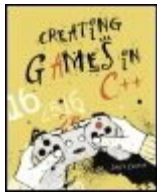


# Creating Games in C++: A Step-by-Step Guide

By David Conger, Ron Little



.....  
Publisher: **New Riders**

Pub Date: **February 21, 2006**

Print ISBN-10: **0-7357-1434-7**

Print ISBN-13: **978-0-7357-1434-2**

Pages: **464**

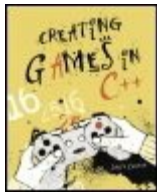
[Table of Contents](#) | [Index](#)

## Overview

Do you love video games? Ever wondered if you could create one of your own, with all the bells and whistles? It's not as complicated as you'd think, and you don't need to be a math whiz or a programming genius to do it. In fact, everything you need to create your first game, "Invasion of the Slugwroths," is included in this book and CD-ROM. Author David Conger starts at square one, introducing the tools of the trade and all the basic concepts for getting started programming with C++, the language that powers most current commercial games. Plus, he's put a wealth of top-notch (and free) tools on the CD-ROM, including the Dev-C++ compiler, linker, and debugger--and his own LlamaWorks2D game engine. Step-by-step instructions and ample illustrations take you through game program structure, integrating sound and music into games, floating-point math, C++ arrays, and much more. Using the sample programs and the source code to run them, you can follow along as you learn. Bio: David Conger has been programming professionally for over 23 years. Along with countless custom business applications, he has written several PC and online games. Conger also worked on graphics firmware for military aircraft, and taught computer science at the university level for four years. Conger has written numerous books on C, C++, and other computer-related topics. He lives in western Washington State and has also published a collection of Indian folk tales.

# Creating Games in C++: A Step-by-Step Guide

By David Conger, Ron Little



.....  
Publisher: **New Riders**

Pub Date: **February 21, 2006**

Print ISBN-10: **0-7357-1434-7**

Print ISBN-13: **978-0-7357-1434-2**

Pages: **464**

[Table of Contents](#) | [Index](#)

[Copyright](#)

[Acknowledgments](#)

[Introduction](#)

[What's Different About This Book](#)

[What's in This Book?](#)

[System Requirements](#)

[Free Stuff on the CD](#)

[Who Am I?](#)

[Part 1: The Really Basic Stuff](#)

[Chapter 1. What it Takes to be a Game Programmer](#)

[Programming Skills](#)

[Computer Graphics Skills](#)

[Game Design Skills](#)

[Art Skills](#)

[Sound and Music Skills](#)

[Summary](#)

[Chapter 2. Writing C++ Programs](#)

[Introducing the Dev-C++ Compiler](#)

[Programming in C++](#)

[Essential Math Operators](#)

[Loops](#)

[While Loops](#)

[Do-while loops](#)

[Windows Programming](#)

[Game Programming](#)

[Summary](#)

[Part 2: ObjectOriented Programming in Games](#)

[Chapter 3. Introducing Object-Oriented Programming](#)

[Software Objects](#)

[Classes](#)

[Logical Operators](#)

[The If-Else Statement](#)

[Namespaces and Scope Resolution](#)

[A Brief Word About Structures](#)

[Summary](#)

#### [Chapter 4. Introducing the LlamaWorks2D Game Engine](#)

[A Step-by-Step Overview](#)

[How Does Llamaworks2D Work?](#)

[A Stationary Ball](#)

[A Bouncing Ball](#)

[Getting Good Results](#)

[Summary](#)

#### [Chapter 5. Function and Operator Overloading](#)

[What Is Overloading?](#)

[Implementing a Vector Class with Overloading](#)

[Summary](#)

#### [Chapter 6. Inheritance: Getting a Lot for a Little](#)

[What Is Inheritance?](#)

[Deriving Classes](#)

[Protected Members](#)

[Overriding Base Class Functions](#)

[Customizing Your Game with Inheritance](#)

[Summary](#)

#### [Part 3: The Essentials of Game Development](#)

##### [Chapter 7. Program Structure](#)

[Program Structure](#)

[File Structure](#)

[A Game Called Ping](#)

[Summary](#)

##### [Chapter 8. Sound Effects and Music](#)

[Sound Effects and Music Are Emotion](#)

[Storing Sound Data](#)

[Sound Effects in LlamaWorks2D](#)

[Noise, Sweet Noise](#)

[Play That Funky Music, Geek Boy](#)

[Summary](#)

#### [Part 4: Graduating to Better C++](#)

##### [Chapter 9. Floating-Point Math in C++](#)

[Getting into the Guts of Floating-Point Numbers](#)

[Case Study: Floating-Point Numbers and Gamespaces](#)

[Summary](#)

##### [Chapter 10. Arrays](#)

[What Are Arrays?](#)

[Declaring and Using Arrays](#)

[Initializing Arrays](#)

[Problems with Array Boundaries](#)

[Summary](#)

## [Chapter 11. Pointers](#)

[Why Are Pointers Important to Games?](#)

[Declaring and Using Pointers](#)

[Pointers and Dynamic Memory Allocation](#)

[Pointers and Inheritance](#)

[Arrays Are Pointers in Disguise](#)

[Summary](#)

## [Chapter 12. File Input and Output](#)

[Games and File I/O](#)

[Types of Files](#)

[Summary](#)

## [Chapter 13. Moving into Real Game Development](#)

[Sprites that Come Alive](#)

[High-Speed Input](#)

[Summary](#)

## [Part 5: The Big Payoff](#)

### [Chapter 14. No Slime Allowed: Invasion of the Slugwroths](#)

[What It Takes to Make a Real Game](#)

[Essential Game Design](#)

[Designing Invasion of the Slugwroths](#)

[Summary](#)

### [Chapter 15. Captain Chloride Gets Going](#)

[Introducing Captain Chloride](#)

[Pulling It Together In The Game Class](#)

[Summary](#)

### [Chapter 16. The World of Captain Chloride](#)

[The New Captain Chloride](#)

[Levels in LlamaWorks2D](#)

[Summary](#)

### [Chapter 17. Captain Chloride Encounters Solid Objects](#)

[Bumping into a Solid Door](#)

[Picking Up a Key](#)

[Making the Door Open and Close](#)

[Summary](#)

### [Chapter 18. That's a Wrap!](#)

[Time for Consolidation](#)

[Enter villains, Stage Left](#)

[Additions to the Game](#)

[Epilogue: Not the End](#)

[Glossary](#)

[Index](#)

# Copyright

## Creating Games in C++: A Step-by-Step Guide

David Conger with Ron Little

New Riders  
1249 Eighth Street  
Berkeley, CA 94710  
510/524-2178  
800/283-9444  
510/524-2221 (fax)

Find us on the Web at: [www.newriders.com](http://www.newriders.com)

To report errors, please send a note to [errata@peachpit.com](mailto:errata@peachpit.com)

New Riders is an imprint of Peachpit, a division of Pearson Education

Copyright © 2006 by David Conger

Project Editors: Davina Baum, Kristin Kalning  
Development Editors: Davina Baum, Denise Santoro Lincoln  
Production Editor: Myrna Vladic  
Copyeditor: Liz Welch  
Tech Editor: Ron Little  
Compositor: WolfsonDesign  
Indexer: Karin Arrigoni  
Cover design: Aren Howell  
Interior design: WolfsonDesign

## Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact [permissions@peachpit.com](mailto:permissions@peachpit.com).

## Notice of Liability

The information in this book is distributed on an "As Is" basis without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

## **Trademarks**

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

## **Dedication**

*This book is dedicated to my mother, Jan Conger, for all the good that she has done and still does.*

*Thanks, Mom.*

# Acknowledgments

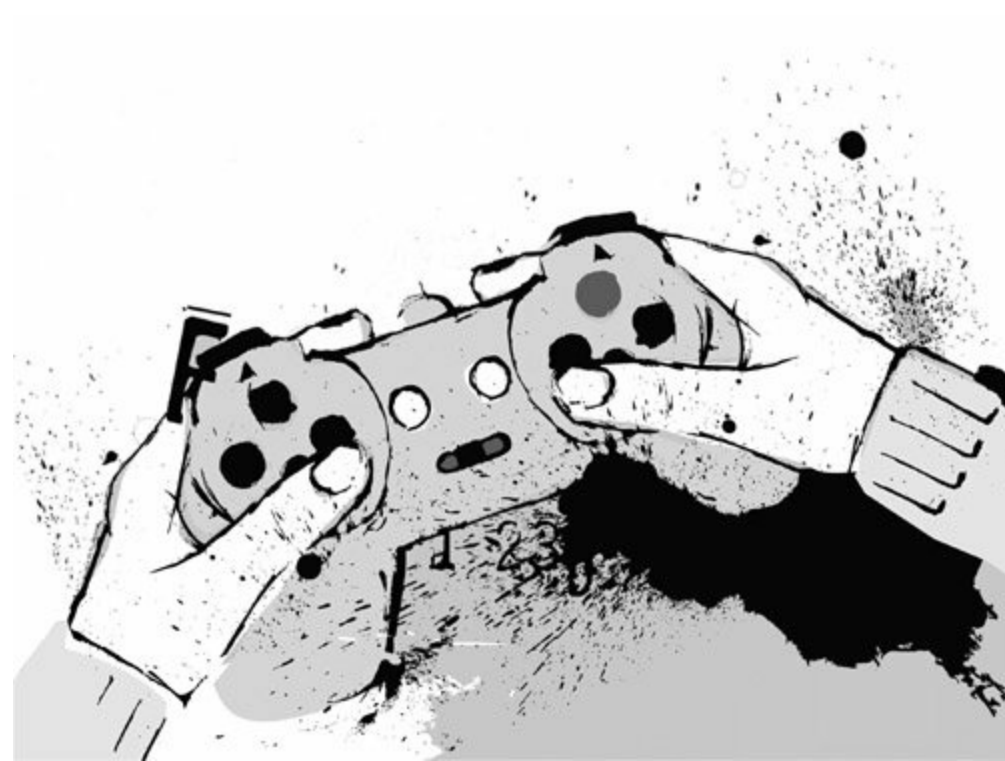
This book has been one of the hardest projects I've ever worked on. Not because the book itself was hard to write, but because of three major computer crashes (and two replacement motherboards), two serious illnesses, two surgeries, having to change compilers after the third chapter, and having to entirely rewrite the game engine after the fourth. I can honestly say that I could not have got through this book without the support of all those who worked on it. I especially want to thank Kristin Kalning, who is simply the best project editor I've ever worked with. Thanks also to Davina Baum, Liz Welch, Myrna Vlastic, Owen Wolfson, Eric Geoffroy, Aren Howell, and Denise Lincoln.

I especially want to thank Ron Little. His technical edits were excellent. And without his help on the sample programs for the last five chapters, I'd probably still be working on this book.

# Introduction

The first video game I ever played (at age 13) was Pong. It was a very simple ping-pong simulation. During my teenage years, a few, more advanced games appeared on the market. Most notable were the Atari games such as Missile Command, a nuclear warfare simulator. When I was 19, I went to live for a couple of years in Japan. There I discovered a whole new world several, in fact.

Around the time I arrived in Japan, the game Space Invaders had just crested its phenomenal wave of popularity. I had never seen anything so cool. By the time I returned to the United States, video games were everywhere.



In the fall of 1981, I started college. Keith, a longtime friend and roommate, pointed to a class in the university's catalog and told me, "You have to take this programming class. I know you'll love it." He was right. By the time two weeks had passed, I knew I would be working with computers for the rest of my life.

And what did I specialize in? Graphics and games, of course.

When I started writing games in college, it was a long and difficult task. The only people who had a prayer of writing decent games were complete geeks (like me).

A lot has changed since then. With the tools available now, nearly anyone can write an original and inventive game.





# What's Different About This Book

There are lots of beginning game programming books. In the end, the question everyone asks is, "What's special about this book in particular?"

I'm glad you asked.

The short answer to your question is that this book is unlike 99 percent of all beginning game programming books in several respects.

## It Doesn't Assume You Know How to Program Computers

Most people who want to get started in game programming don't know how to program at all. They are bright, creative, and innovative and they like games. But they can't yet write computer programs.

Virtually every beginning game programming book I've ever seen assumes you know a programming language like C or [C++](#). That means that most beginning game programmers have to spend months or years learning to program before they can even start a game.

What a waste.

If you want to be a game programmer, but you have little or no programming experience, this book is for you. As I mentioned earlier, the tools available for writing games make the whole experience *much* easier than when I started in the industry more than 25 years ago. There's no reason that a bright and creative person like you should have to jump through a lot of silly hoops learning to program *before* you learn to write games. **You can learn to program computers and write games at the same time.** Unfortunately, that thought hasn't occurred to most authors of beginning game programming books.

## It Teaches You Real Game Programming Skills

There are a few, very rare, beginning game programming books that are written for nonprogrammers. To try and make things easy, they teach you to write games in a programming language called Basic. Unfortunately, no games are written in Basic. It's far too slow for real games. Instead, game programmers use a language called C++ (pronounced *see-plus-plus*). So if you

read one of these books, you'll have to start from scratch in a new programming language if you want to write a real game.

Unlike any other book on the market, this book teaches nonprogrammers how to write games in C++. When you finish reading this book, you'll be reasonably proficient with the programming language that *real* game programmers use.

Also, I explain and demonstrate the techniques used by professional game programmers for animation and sound. In addition, I demonstrate that it is surprisingly easy to add essential physics to your games.

## **It Teaches You How to Build A Real Game**

Many beginning programming books give you lots of nifty little sample programs that demonstrate the concepts they're teaching. However, very few show you how to pull all of those concepts into a complete game. That's a skill in itself. And the process of building a complete game often stumps people who clearly understand how to write the individual pieces.

By the end of this book, you'll see how to write a complete game. The game we'll be building is called "Invasion of the Slugwroths." It's a simple side-scroller (I'll explain that term later) similar to many games that were popular in the 1980s and '90s.

Aren't most games today written in 3D?

Well, yes. But writing 3D games is much harder. If you start with a side-scroller like *Invasion of the Slugwroths*, you'll learn fundamentals of game programming before you have to deal with 3D concepts. It makes the learning process much easier and much more fun.

## **It Provides Everything You Need to Write Games**

Not only does this book teach you everything you need to get started in game programming, it provides you with all of the tools as well. To write computer programs, you need a compiler, linker, and debugger. These can easily cost \$500. Have no fear. You don't need to cough up your hard-earned cash. You'll find them all on the CD that comes with this book, at no extra cost.

To save themselves from having to write program code that nearly every game

uses, many game programmers use a game engine. A game engine supplies program code that performs the most common tasks in games. Professional game engines typically start at about \$100. But don't go out and buy one. I've supplied one for you for free on the CD.

In addition, I've provided programs for making music, creating a game's graphics, and testing your animations. With the development tools and the game engine you get on the CD, the CD alone is well worth the cost of the book.

## **Both Teens and Adults Can Use This Book**

I have been asked for years by parents what book they should buy to get their teenaged son or daughter started in game programming. Adults in their twenties and thirties also often ask me how they can get started. I have difficulty recommending most books because they are really too technical. They're often too hard to understand for people who don't have a college degree in math, engineering, or computers.

Although this book is not written specifically for teens, it is usable by everyone. If you're 15 or over, you can use this book. Both teens and adults will find everything they need right here. This book explains the terms and ideas it uses. It provides you with a lot of the essential program code (programs are built from program code) you need to write games. It teaches programming, computer graphics, and games. It's one-stop shopping.

# What's in This Book?

This book is divided into five main parts.

- [Part 1](#) gives an overview of the tools and skills you'll need to write games. It also covers the most basic programming concepts.
- [Part 2](#) dives into object-oriented programming, which is a style of programming used by all professional game programmers.
- [Part 3](#) shows how game programs are constructed. Here, you'll write your first game Ping, a clone of the ancient Pong game. You'll also learn to add sounds to your games.
- [Part 4](#) raises your C++ skills to a level that enables you to write real games.
- [Part 5](#) enables you to pull everything you've learned together to write the game Invasion of the Slugwroths, and leaves you fully equipped to go on and write your own games.

In addition, I provide you with a glossary that explains all of the technical terms presented in the book. I've also compiled an extensive list of books that I recommend you read after you finish reading this one. That will help you move forward into topics like 3D graphics. You'll find the list of recommended reading on the CD that comes with this book.

# System Requirements

To use the tools included with this book and to run the sample programs you compile as you read the chapters, you'll need a computer with at least the following:

500 MHz Pentium III computer.

Minimum 128 MB of system RAM.

Windows 98 Second Edition or later operating system.

OpenGL-compatible video adapter card. Virtually all video cards today are compatible with OpenGL.

300MB of free space on your computer's hard drive so that you can install all of the tools and source code. You might not need this much space if you choose not to install some of the tools.

# Free Stuff on the CD

## Note

If the HTML page is not displayed when you insert the CD, click the Windows Start button in your Taskbar and select Run. In the dialog box that appears, type `<d>:\AutorunPro.exe`, where `<d>` is the letter of your CD/DVD-ROM drive.

On the CD, you'll find an assortment of essential tools for game programmers. Although I list them here, you can get more information on them by inserting the CD into your CD- or DVD-ROM drive. When you do, an HTML page will automatically display. That page contains a list of everything on the CD. In that list is an item called Tools, Tools, Tools. That item provides a link to a page in the CD that explains what the tools are and how to install them.

**LlamaWorks2D** This is the game engine you'll use to write your games. I wrote this version of LlamaWorks2D especially for this book. Building games with LlamaWorks2D will save you many hours of programming. In addition, it handles many of the repetitive and tedious tasks you have to do in order to get a game up and running. Using LlamaWorks2D, you'll spend less time with the mechanics of Windows programs and more time on games.

**Dev-CPP** The free Dev-C++ compiler is a combination of a compiler, linker, debugger, and program editor. You need all of these tools in order to write games. Therefore, *you must install this program first.*

**Audacity** Audacity is an excellent sound editor that is available for free. With Audacity, you'll be able to record sound effects and edit them for your games.

**GIMP** You'll need a program for drawing the images and animations your games use. Windows Paint, which is a free program that comes with Windows, just doesn't have enough features for you to produce professional graphics. Therefore, I've provided a program called GNU Image Manipulation Program (GIMP). With this powerful program, you'll have what you need to draw everything that appears on the screen when

your game runs.

**And Much More** In addition to the tools mentioned here, you'll find documentation for important graphics and sound technologies, an outstanding music production program, a graphics file format conversion tool, a 3D image production program, and an animation tester. Please see the CD for more details.

## Installation Instructions

I've made installing the programs in the CD extremely straightforward. On the main page that is displayed when you insert the CD, there is an entry called Tools, Tools, Tools. In that item is a link. When you click the link, you'll see a page called Installing the Free Tools. Find the tool you want to install on that page. For most of the tools, there is an installation link. Clicking this link runs the install program automatically. A few of the tools must be copied to your hard drive. The Installation section for that tool contains instructions on how to do that.

## Compiling The Sample Programs

This book contains numerous sample programs that demonstrate the concepts it teaches. In order to view the programs, you must first compile them. The instructions for compiling them are slightly different for each program. To make things more convenient for you, I've written instructions for how to compile every program. You'll find the compilation instructions on the CD. To see them, insert the CD into your CD/DVD-ROM drive. On the HTML page that appears, you'll find a list item called Compilation Instructions. Click that link.

At this point, an HTML page appears called Compiling the Sample Programs. This page contains a list of all of the sample programs grouped by chapter. To see how to compile a particular program, click on its link in the list. The compilation instructions for that program will appear.

In addition, I've provided a compiled version of each program. They're in the Bin (short for binary) folder for their respective chapters. You'll see a link for each Bin folder on the Compiling the Sample Programs page.



# Who Am I?

Hi, I'm David Conger. I've been in the computer industry since 1981, when I took my first professional programming contract shortly after entering college. Most of my career has been focused on graphics, games, and network programming. After graduating from college, I wrote firmware for graphics display controllers used on military aircraft. After that, I taught college-level programming classes for several years. My next career change enabled me to attain my long-standing goal of becoming a professional game programmer. I wrote games for American Laser Games, For Her Interactive Inc., and Microsoft Corporation.

I started writing books in 1987. My first book was a collection of folktales from the Far East and India retold for Western children. Since then, I've written a fairly steady stream of computer books.

I play way too many games. If it wasn't for my wife and kids, I would hardly ever visit reality.

I have a lot of experience in computer graphics, games, writing, and teaching. I want to help you to get into game programming as fast and painlessly as possible. Game programming lets you get paid for being a programmer, designer, artist, composer, and general nutcase.

So read on and have fun. That's what games are for.

# Part 1: The Really Basic Stuff

[Chapter 1. What it Takes to be a Game Programmer](#)

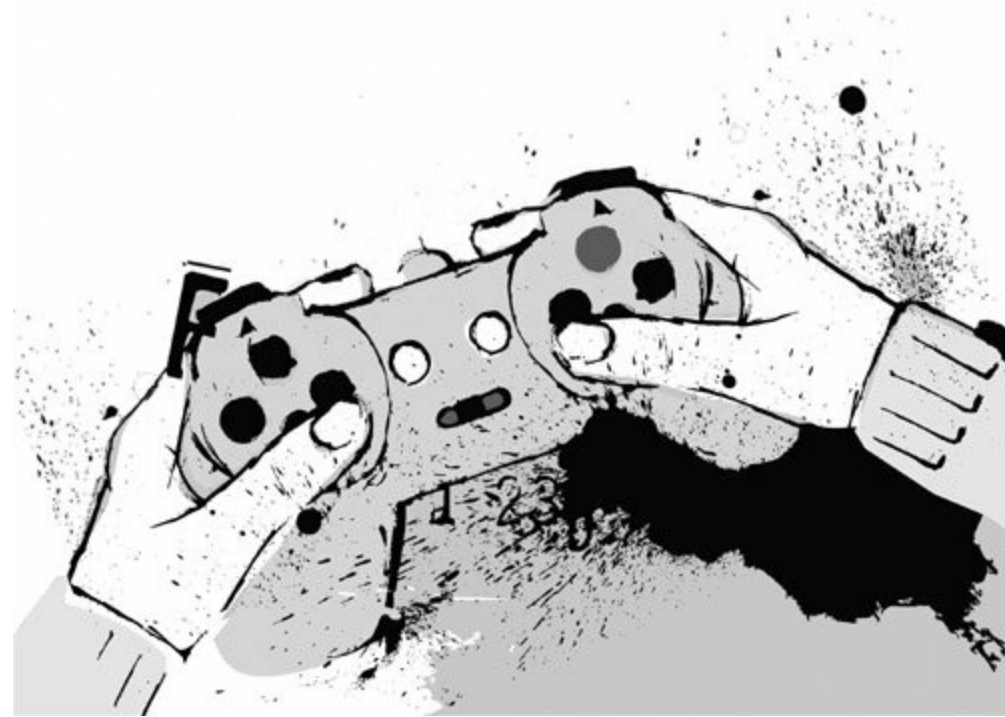
[Chapter 2. Writing C++ Programs](#)

# Chapter 1. What it Takes to be a Game Programmer

You're reading this book because you want to be a game programmer. You may not have any programming experience at all. That doesn't matter. This book shows you everything you need to know to get started. Before you finish reading this book, you'll see how to develop your own innovative games.

So what does it take to be a game programmer?

Game programming can include a wide range of skills. To be a game programmer, you must, of course, know a bit about programming. You should also know the basics of computer graphics. In addition, it helps to know how to design games. You'll learn all of these skills right here.



It also helps to be a musician and an artist. That's not really required. However, you do have to know how art and music are handled on a computer. If you've ever drawn something using the paint program that comes with Windows, you know the basics of creating art on a computer. If you know how to rip MP3s, you know how to prepare music for games. But if you've never done either of these, don't worry. I'll teach you how to handle both art and music for your games.

This chapter presents a brief overview of the most essential skills you'll need: programming, computer graphics, game design, art, and the ability to use sound and music. Because it's an overview, don't worry if you run across something that you don't understand. Everything discussed in this chapter is presented again in greater detail in later chapters.



# Programming Skills

People are often fearful of learning to program computers. Programming can get pretty hairy at times, but learning to program proficiently is not as difficult as it may seem. In fact, if you like to tinker with things and find out how they work, programming often seems like play.

The first step in learning to program is to understand what a program is.

## What is a Computer Program?

### Note

Computer programs are just complex sets of instructions. The instructions tell the computer how to perform a task, such as playing a game.

Imagine you want to bake a cake. If you're like me, you don't know how to do that. In that case, you'd probably go to a cookbook and find a recipe. The recipe is a set of instructions. If you follow the instructions exactly, you get a cake. If not, you may wind up with a foul-tasting mess.

A computer program is like a recipe. It's a set of instructions. The instructions in a program tell the computer how to be a game machine. If you write the instructions properly, you get a game. If not, you get a mess.

Each instruction in a computer program is made up of one or more statements in a [programming language](#).

*A what?*

Computers don't understand human languages like English or Japanese. In fact, they don't really "understand" anything at all. But the computer's [microprocessor](#), which is also called its [central processing unit \(CPU\)](#) can execute commands.

### Note

Every computer has a microprocessor. You can think of the microprocessor as the computer's "brain."

To be executable, the commands in a program must be in binary. If they're not, the microprocessor won't be able to execute them.

So what's binary?

Binary is a number system. It's also called the base 2 number system. People normally use the decimal, or base 10, number system. In other words, we count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and so on. Binary is base 2, so you count 0, 1, 10, 11, 100, 101, 110, 111, 1000, and so forth. You really don't have to know the details of using binary in order to program computers. All you have to know is that a binary number is a group of 0s and 1s. All instructions you give to a microprocessor must be 0s and 1s, or the microprocessor won't be able to execute them.

When a program is running, all of its binary instructions are stored as [bits](#) in your computer's memory. A bit is simply a binary 0 or 1. Therefore, the binary number 10 is 2 bits long. In computers, the bits in memory or on a disk are grouped together. A group of 8 bits is a [byte](#). The binary number 10011100 is 8 bits long so it fits in 1 byte. A group of 1024 bytes is a [kilobyte](#); 1024 kilobytes is a [megabyte](#); 1024 megabytes is a [gigabyte](#); 1024 gigabytes is a [terabyte](#). This is illustrated in [Table 1.1](#).

**Table 1.1. Groupings of Bits and Bytes in a Computer**

A Group of...	Equals
8 bits	1 byte
1024 bytes	1 kilobyte
1024 kilobytes	1 megabyte
1024 megabytes	1 gigabyte
1024 gigabytes	1 terabyte

# The C++ Programming Language

When I started in computer programming, you *had* to understand binary to be able to write programs. There wasn't really any way to avoid it. These days, that's not necessary. Consider yourself lucky. Programming in binary isn't much fun. Instead, you can now write programs in languages that are similar to the languages people speak. Most games are written in a language called C++ (pronounced *see-plus-plus*).

The C++ programming language enables you to write statements that let you control all parts of a computer, such as the display, sound card, or joystick. With C++ statements, you can display aliens on the screen, let the player control a space fighter, and shoot the aliens into space dust. C++ also lets you play music and add sound effects. It is a fast, efficient language that you must know to be a professional game programmer.

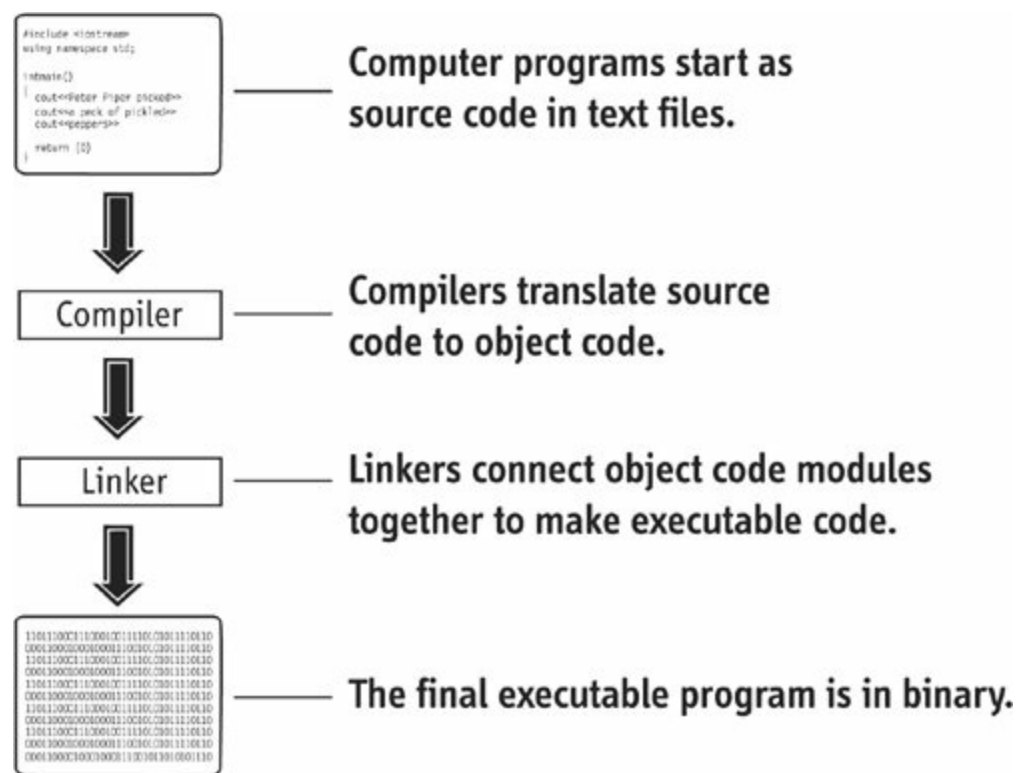
C++ programs start as C++ instructions that we store in [text files](#). A text file is just what it sounds like: It's a file on the disk that contains text letters and numbers. C++ text files are also called [source files](#). The entire collection of C++ instructions is called the [source code](#) of the program.

## Tools of the Trade

Recall that microprocessors only execute commands that are binary numbers. C++ is not binary; it's more like English. How do you translate C++ statements into binary commands? [Figure 1.1](#) gives the answer.

**Figure 1.1. Translating a C++ source file into binary.**

[\[View full size image\]](#)



[Figure 1.1](#) shows that there's a fair amount of work involved in getting source code translated into an executable binary program. In the next few sections, I'll explain each step.

## Compilers

[Figure 1.1](#) shows that you need a special program called a [compiler](#) to translate your C++ source code into binary. Therefore, you'll need a compiler to write your games. I've provided you one for free. It's called Dev-C++ and you'll find it on the CD that comes with this book. The instructions for installing it are in the Introduction.

## Linkers

Compilers translate source code into an intermediate form called [object code](#). Object code is binary, but it is not executable. Object code must be converted to [executable code](#), which is an actual program that you can run on your computer. The tool that converts object code to executable code is a [linker](#).

You'll need a linker to create your games. A linker is included with the Dev-C++ compiler on the CD and it's installed when you install Dev-C++.



Whenever you compile your program with Dev-C++, the linker runs automatically, so your program is both compiled and linked in one step.

## **Note**

To access the Dev-C++ compiler, just open the DVD and go to the \Tools\DevCPP folder. If you have not installed Dev-C++ on your computer, please turn to the Introduction and do that now.

## **Warning**

All of the C++ source code you will encounter in this book is designed to work with Dev-C++. It may or may not compile with other compilers, such as Microsoft's Visual C++. Ideally, C++ source code should work with all compilers, but in reality, subtle differences exist between compilers produced by different companies. Therefore, I strongly recommend that you use Dev-C++ for all the programs in this book. After you become an experienced C++ programmer, you should have no problem compiling this book's programs with Visual C++ with only minimal changes.

## What Is Dev-C++?

Although you'll often see me refer to Dev-C++ as a compiler, it is really an integrated development environment (IDE). An IDE contains everything you need to compile a program. This includes a program to edit your C++ source files, a compiler, a linker, and a debugger (linkers and debuggers will be explained shortly). The compiler, linker, and debugger that the Dev-C++ IDE uses are Windows versions of the GNU Compiler Collection (GCC). The GCC is an open source project that is written by a large and vibrant developer community. You can find the GCC at <http://gcc.gnu.org>. The Windows version of the GCC is also an open source project whose home page is [www.mingw.org](http://www.mingw.org).

## Debuggers

It's not possible for us to write perfect games in just one try we'll make a lot of mistakes. In programming, mistakes are called *bugs*. It's normal for programmers to accidentally create thousands of bugs in each game they write. Yes, I did say *thousands*.

As a result of the mistakes we make, you and I need a tool to help us find and fix bugs. Appropriately enough, that tool is called a *debugger*. You get a debugger for free with Dev-C++. Like the linker, it's installed automatically.

## Graphics Libraries

Many of the tasks involved in games are universal in game programming. For example, several times a second, every type of game gets user input, reacts to it, and updates the contents of the screen. No matter what type of game you're writing, your game follows this design.

The basic tasks in all games are by far some of the hardest. It's not easy to draw circles, lines, and so forth on the screen. A lot of very smart people had to think for a long time to learn how to do these tasks efficiently. The methods they use are very advanced and very involved. This stuff is not easy, and it used to be the case that every game programmer had to know how to do it all.

Fortunately, you no longer have to bother with learning how to do these basic tasks. When you write your games, you'll use a tool called a *graphics library*. A graphics library contains code that does all the basic graphics tasks such as drawing lines and circles, or displaying pictures on a screen. The library uses all the best methods and techniques that were developed by all the smart

people I mentioned in the preceding paragraph. As a result, you already have code for the basic tasks you'll do when you write games. You don't have to write it yourself.

By the way, the graphics library you'll use is called OpenGL. OpenGL is used in high-powered games such as Quake and Doom from id Software. There is a version of the OpenGL graphics library for Dev-C++. The instructions for installing it are in the Introduction.

Microsoft also provides a graphics library called DirectX Graphics (more commonly called Direct3D). Like OpenGL, Direct3D is used for professional games. Direct3D is part of DirectX, which contains libraries for adding sound, music, and networking to games.

Whether you use OpenGL or Direct3D is largely a matter of preference. They both will get you where you need to go. Although this book primarily uses OpenGL, I've also provided the Microsoft DirectX Software Development Kit (SDK) on the DVD that comes with this book. It's in the folder\Tools\Microsoft DirectX SDK.

## Tip

If you're planning on becoming a professional game programmer, it's best to be familiar with both OpenGL and DirectX.

If you have not installed Dev-C++ and OpenGL on your computer, please turn to the Introduction and do that now.

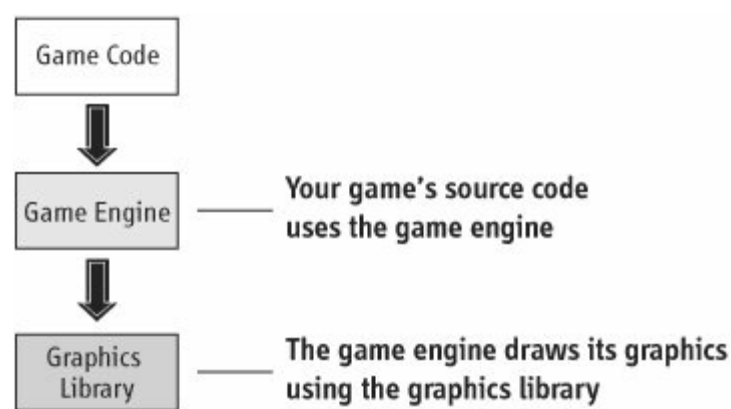
## Game Engines

Games that are of the same type pretty much all work the same way. For example, all first-person shooters like Doom need code that does essentially the same set of tasks. All side-scrolling games, such as the old Super Mario Brothers or Sonic the Hedgehog, also use code that does basically the same things.

To make game programmers' lives easier, other programmers often develop *game engines*. A game engine performs the most common tasks that a particular type of game does. For instance, a game engine for first-person

shooters handles all of the tasks associated with displaying scenery, drawing bad guys, and so forth. You have to insert your own scenery, code for your weapons, and any other code that is specific to your game. The game engine handles displaying buildings, trees, and things like that. [Figure 1.2](#) shows how game engines are used.

## Figure 1.2. Games are built in layers of code.



As [Figure 1.2](#) shows, games use game engines to get their work done. Game engines, in turn, use graphics libraries to do all of their drawing. If you have a game engine and a graphics library, it saves you from writing a tremendous amount of code. All you have to do is add your game code on top of the game engine. That's the only part of the game that you have to write. Take my word for it; that's all you want to worry about. You'll find writing your game code challenging enough.

### Note

The LlamaWorks2D game engine is on the CD in the folder `\Tools\LlamaWorks2D`, and you'll find instructions for installing LlamaWorks2D in the Introduction.

As mentioned earlier, different types of games each need their own game engine. So you may run across game engines that are for first-person shooters, others that are for flight simulators, and so on. The best game engines are generic enough to be used for more than one type of game. Many excellent game engines are available commercially. The people who write

them usually require you to buy the engine or pay a royalty when you sell your game. Some require both.

There are also outstanding game engines you can get for free. In fact, I've written one for you and provided it on the CD that accompanies this book. It's called LlamaWorks2D. Throughout the rest of this book, I'll show you how to use it and how to add your own code to it to produce many types of games.

If you have not installed the LlamaWorks2D game engine on your computer, please turn to the Introduction and do that now.



## Games in 2D?

From the name LlamaWorks2D, you've probably figured out that this game engine does 2D rather than 3D games. You may also be wondering why I don't teach you how to do 3D games. The answer is simple: It's hard.

This book teaches the essential skills every professional game programmer must have. It does so by teaching you to write games in 2D, which is considerably simpler than doing 3D. However, don't think that your game programming skills will be anything less than professional when you get done. Many games today are written in 2D. Command and Conquer by Westwood Studios and SimCity by Maxis are two examples. They are both top-selling games that are written with 2D graphics.

Also, many 3D game programming techniques require that you first understand 2D game programming. You would be surprised how much 2D programming there is in 3D games. Therefore, the best way to start learning to write games is with 2D graphics. It gives you the fundamentals, which makes learning 3D game programming much easier.

# Computer Graphics Skills

In addition to programming skills, you'll need to understand how computer graphics work. If you want to become a guru in computer graphics, be prepared to do almost nothing but that for the rest of your career. However, you don't have to be a guru to start writing games. You just need to know a few fundamental concepts.

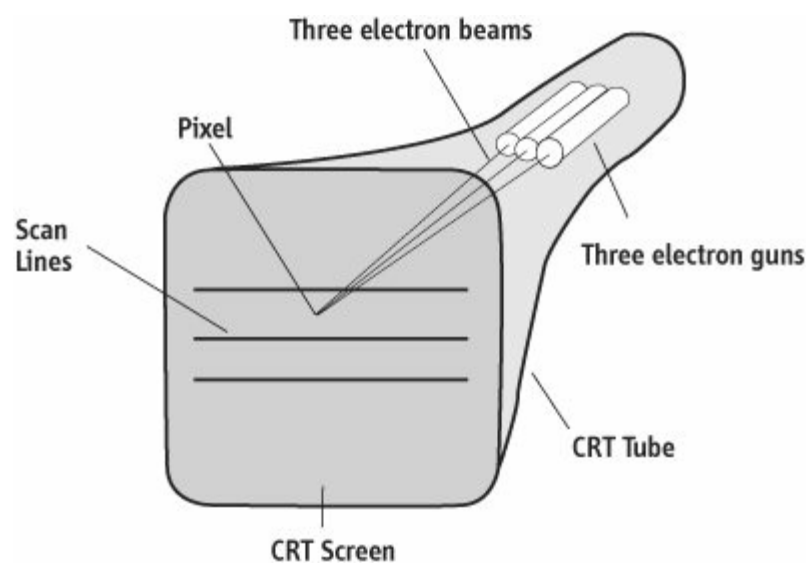
## The Basics of Computer Graphics

The tool you'll use for displaying graphics in your games is a monitor, which is also commonly called a screen, or display. An understanding of monitors is the starting point for developing computer graphics skills.

### How Computer Monitors Work

The internals of monitors aren't as complex as you might think. [Figure 1.3](#) illustrates how they work.

**Figure 1.3. How computer monitors work.**



These days, many computer screens that do not use electron guns. Instead, they use an electrical current to make small cells of plasma glow different colors. These gas plasma displays are nice because they can be made quite flat. For this reason, they have long been used in laptop computers. Increasingly, they are also being used for desktop displays. Gas plasma displays are arranged just like monitors that use electron guns. Specifically, they contain rows of pixels and each pixel is composed of a red, a green, and a blue dot.

A monitor, also called a screen or display, is coated on its front with a chemical called phosphorus. When electrons hit the phosphorus coating, the phosphorus glows. The color that the phosphorus emits depends on what chemicals you add to it. The phosphorus coating on the front of your screen is made up of groups of red, green, and blue dots. One group of dots is called a [\*pixel\*](#). Every pixel has a red, green, and blue dot in it. To make a pixel turn a particular color on your screen, you mix different amounts of red, green, and blue.

## Tip

When you write a game, you can assume that everyone who plays it will have a computer that supports the 800x600x24 graphics mode. That's a resolution of 800 pixels across and 600 rows of pixels with a color depth of 24 bits per pixel. If you want to write a cutting-edge game (not recommended when you're just starting out), you can assume that players' computers support the 1024x768x24 graphics mode.

The electrons that make pixels glow come from electron guns inside your computer monitor. That's why the darn thing is so deep. The bigger the screen, the farther back the electron gun has to be so it can hit all the pixels. As [Figure 1.3](#) shows, color screens actually have three electron guns, one each for red, green, and blue.

As you can see from [Figure 1.3](#), all of the pixels on the screen are arranged in rows. The rows are called [\*scan lines\*](#). The electron guns fire at each pixel, one after the other. They start at the top scan line and work their way down to the



bottom. When they reach the end of the last scan line, they start again at the top of the screen. The time that it takes for the electron guns to hit every pixel on the screen is called the screen's [refresh rate](#).

## Selecting Graphics Modes

Computer monitors display better pictures if they have a high number of scan lines as well as a high number of pixels per scan line. In other words, if there are lots of pixels per row and lots of rows of pixels on the screen, your game looks better. It has a high [resolution](#).

You specify a monitor's resolution by stating the number of pixels per row followed by the number of rows on the screen. So, for instance, a common resolution used by games is 800x600. That's 800 pixels per row and 600 rows on the screen.

All monitors are capable of displaying more than one resolution. Each resolution is called a [video mode](#). You'll often hear game programmers talk about 800x600 mode or 1024x768 mode. Obviously, the higher the resolution, the better your graphics look. The problem with that? Higher-resolution modes require more memory because drawing those graphics requires more processing power. In addition, not every monitor has support for all high-resolution modes. The monitor itself is connected to a [video adapter](#). The capabilities of the computer's video adapter also affect which modes a computer can set its monitor to. [Table 1.2](#) shows some common modes.

**Table 1.2. Common Video Modes**

Mode	Description
640x480	Supported on all monitors and adapters. Low resolution. Poor graphics.
800x600	Supported on all monitors and adapters. Decent resolution with decent graphics.
1024x768	Supported on most (but not all) monitors and adapters. High resolution with very good graphics.

When you select a mode for your game, you want to select one that's high enough to draw good graphics, that's likely to be supported on nearly

everyone's computer, and that doesn't require too much memory and processing power. Typically, that ends up being either 800x600 or 1024x768.

The graphics mode also determines [color depth](#). Color depth determines the number of colors a monitor can display. Every color is specified by a unique number. The number contains pixel values for the colors red, green, and blue. Every other color a monitor displays is created by mixing red, green, and blue.

## Tip

With a color depth of 24 bits, you can get a good range of colors. Almost all video adapters support that color depth.

Suppose a graphics mode on a monitor has a color depth of 8 bits each for red, green, and blue, for a total color depth of 24 bits. Each set of 8 bits specifies an intensity of its respective color. Eight bits enables you to specify 256 individual numbers, 0-255. A value of 255 for red means that the pixel's red is set to full intensity; 128 is half intensity; 0 is off, no red. With a 24-bit color depth, you can, for instance, set the 8 bits for red to 128, the 8 bits for green to 0, and the 8 bits for blue to 255 to get a color that contains some red, no green, and lots of blue all mixed together. The result is a nice shade of purple.

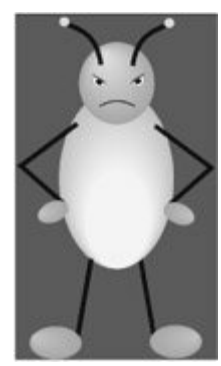
The vast majority of video adapters actually support a color depth of 32 bits. They still use 24 bits for color. The extra 8 bits are used for special effects.

## Displaying Pictures on a Computer Monitor

The images games display on screens are called [bitmaps](#), [pixel maps](#), or [pixmap](#)s. A bitmap contains one item of data for each pixel you want to color on the screen. As a result, we often end up calling the data items pixels. That's not strictly correct, since they're not really pixels. They contain data for pixels, so we tend to apply the name to them.

**Figure 1.4** shows a Krelnorian who's not at all happy you showed up on his planet (Krelnor, which is just a hop, skip, and a hyperjump away from the popular tourist destination planet of Jaglon Beta).

### **Figure 1.4. A bitmap of an alien.**



If this picture is stored in a computer, it's stored in a bitmap. Notice that the bitmap is rectangular—all bitmaps are. But the likelihood is that you just want to display the Krelnorian. Your game probably doesn't display the bitmap's background. What your game needs to do is combine the image of the Krelnorian with whatever's on the screen without displaying the background pixels in this bitmap. To get just the Krelnorian and not the background pixels, your game must make the background pixels transparent. The graphics library (OpenGL) and game engine (LlamaWorks2D) take care of that for you. When you add your bitmaps to a game, you'll need to tell the game engine which color is the background color. The library and the engine display the Krelnorian but not the background color. As we get into animation in later chapters, I'll show you how this is done.

## Animating Your Game

### Tip

To get good animation, you need to display 30 frames per second or more.

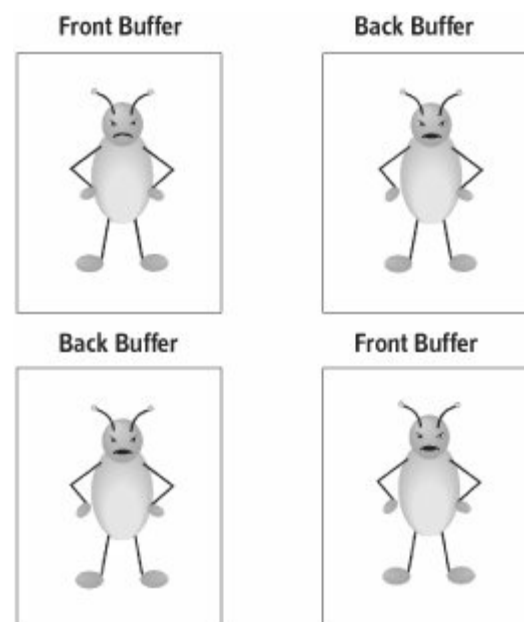
Games aren't much fun if nothing moves. If you want to make the Krelnorian in [Figure 1.4](#) (or anything else) move, you use the same techniques that have been used in movies and cartoons for over a century.

Think about how movies work. A movie is just a collection of pictures on a long reel of film. Each picture is called a [frame](#). In each frame, the scene changes slightly. You get the illusion of movement by displaying the frames on the movie screen one after the other very quickly.

This technique is exactly how animation works in computer graphics. Your game draws a frame onto the screen, displays it for a portion of a second, and then draws the next frame.

In reality, what games do is draw each frame into a chunk of the computer's memory. The chunk of memory is called a *buffer*. The image that is displayed on the screen is in the *front buffer*. While the player is looking at the frame in the front buffer, the game draws the next frame into another buffer called the *back buffer*. When it's done, it switches the front buffer and the back buffer. Because the back buffer is now the front buffer, the image in it is displayed on the screen. This process is illustrated in [Figure 1.5](#).

**Figure 1.5. Front and back buffers switch for each frame of animation.**



[Figure 1.5](#) shows two rows of pictures. Each row represents what's going on in a frame of animation. The picture marked as the front buffer is what's currently displayed on the screen. In the top row, the picture on the left is the front buffer showing the unhappy Krelnorian. While that's being displayed, the game draws the next picture into the back buffer. When it is drawn, the game switches the buffers. The result is shown in the next frame, which is depicted in row two.

Row two shows that the front and back buffers have been switched. The front buffer is now the picture on the right. The game is drawing the Krelnorian's next movements into the back buffer on the left. When it's done, the buffers

will be switched again. With each buffer switch, the Krelnorian appears to move slightly.

Doing this in your game isn't hard. All your code has to do is draw each frame. The graphics library and the game engine handle everything else.

# Game Design Skills

The term "game design skills" means different things to different people. If you're a programmer, it usually means that you design your code well. On the other hand, if you're someone playing a game, it means designing a game that is playable and fun. Let's take a quick look at both of those viewpoints.

## Structuring Game Code

From a programming point of view, you must learn to design your game program code in such a way that the program is robust (in other words, it doesn't crash) and operates the way that games should. No matter what the game is, the program code performs the same essential set of tasks as every other game. Game programmers have found by experience that there are certain ways of putting game program code together that work well. There are other ways that don't work well. This book not only shows you how to write games, it also explains and demonstrates how to build your programs in ways that work well. As we go through the rest of the book, I'll point out ways to structure your game code properly.

### Note

You'll find a list of books on game design in the list of suggested reading on the CD.

## Making Games Good

If you want to write your own games, you must spend time and effort (and probably money) finding out what makes a good game good. Also, you need to know what types of games there are and what sells.

Asking what sells seems the same as asking what makes a good game good. Not so. Mediocre games can sometimes outsell good games. Why is that? Let me give you an example. About a year after the first version of Doom came out, I was at an industry show called Comdex. Nearly every game company was selling a Doom-style first-person shooter. When you asked them what

their game was like, they said, "It's like Doom, only better." For the most part, they were wrong. The games were like Doom but not better.

At that time, the market was saturated with first-person shooters. Some of the games I saw that year at Comdex actually *were* better than Doom. However, because the market was so flooded with first-person shooters that year, not many of them were selling well (except Doom). As a result, even some very good games had mediocre sales.

To top it off, I saw some games that were very mediocre but were selling well. Why? Because they were different than most of what was on the market.

## **Make it Unique**

When you approach game design, you have to come up with something unique. Just rehashing what's already out there is a waste of time. That doesn't mean you have to think up an entirely new type of game. Your game can still fit into a well-established category. However, it has to be significantly different and better than the other games in that same category. And when I say "better," I don't mean that it just has better graphics. Players will often settle for graphics that are like everything else out there *if the game itself is significantly better*.

An example of what I mean is Halo, from Microsoft. Shooters like Halo were on the market for about 10 years before it came out. But Halo did quite well, especially on Microsoft's Xbox.

Why?

First, Halo had a unique story line that was well integrated into the gameplay. It also had a wider array of vehicles to drive and fly than most games. The vehicles added a lot of fun to the experience of playing the game. Halo also lets you move seamlessly between indoor and outdoor battles. At the time it was released, there weren't a lot of games that let you do that so easily and the outside areas of most games weren't nearly as large as those in Halo. Lastly, Halo had excellent music and sound that stood head and shoulders above most of the competition.

## **Tip**

Generally, if you describe your game as being "like Game X, only better," don't bother writing it. Your game is not unique enough. People will just buy Game X instead.

Because of these unique elements, Halo did quite well even though there was a decade of similar shooters on the market. Of course, Halo's excellent graphics did help, but other games with similar graphics have not done as well. Upon its release, Halo's unique features set a new standard for all 3D shooters.

## **Make it Plausible**

If aliens were to invade the earth, the Powers That Be wouldn't send just me to take care of the problem by myself. They'd send a whole bunch of people. I don't know how many games I've played in which one person was sent to save the entire planet (or galaxy or universe). It's completely implausible.

If I am the only person who can save the world, there has to be a reason. For example, I sometimes play a game called Gun Metal, from Yeti Studios. In this game, the player is the pilot of a prototype vehicle that can transform from being a battle robot to being a jet fighter. The idea works because the vehicle is a prototype, making it the only one of its kind. The player goes up against an army that is much better equipped and rather formidable. You have to be clever about how you approach each situation. Just diving in and shooting anything that moves isn't usually the best approach.

The nice thing about the setting of Gun Metal is that you have others helping you. Ground troops provide protection for your energy recharge unit. Drop ships move troops into position after you've cleared out the enemies. Battle cruisers take on enemy ships, with you providing support. You get the feeling that you're part of a much larger force, which is plausible in an invasion.

I've also played games that began with some kind of accident. The player in that case is usually the only survivor in a hostile environment. That's another plausible reason to be alone.

## **Make it Playable**

To be playable, games must enable the user to control the character or vehicle in a smooth and intuitive way. A good example of this is the old Super Mario 64. When it came out, it was a big step forward. It was very easy to make Mario run, climb, jump, or whatever. Even very young kids could master it quickly. Games also need challenges that are hard but not too hard. Let me give you an example of both of these ideas.



I once played a game for the now-defunct 3DO game console in which I was the pilot of a vehicle that was to save the world from invasion. No reason was given as to why I was the only one sent (implausible). In any case, as I played, I quickly discovered that it was next to impossible to fly the vehicle well with the 3DO controller. Also, this game placed some of the power-ups I needed to continue out in areas that were constantly being bombarded from orbit. You could only get the power-ups and return to the safe area if your health was 100%. It was next to impossible to fight your way through to the correct position with your health at 100%; there weren't enough recharges along the way for that. Because of this situation and the difficulty in controlling the vehicle, the game was too hard to play after the fifth level. I just quit. It probably wouldn't surprise you to know that, although the game had excellent graphics integrated with outstanding video, very good music, and killer sound effects, it sold quite poorly. It was so unplayable that word spread rapidly among 3DO owners, and they avoided it like the plague.

## **Make the Player Think**

Let's face it: Anyone who's played games has played games in which you just go around and shoot everything that moves. That's been done to death. You've got to have a wider range of activities, and the player has to figure out what activities to use and when to use them. Let's go back to Halo for a good example of this.

If you play Halo, you can play it much like the thousands of shooters that came before it. That is, you can just shoot everything up. However, you can be cleverer about it, if you want to.

For example, instead of just stomping into an area and shooting up all the aliens, Halo lets you sneak up on them while they sleep. Using your weapon as a club, you can kill the aliens by hitting them. It's possible to clear an entire large room of aliens by silently whopping them each on the back of the head.

Another interesting thing about Halo is that you have constantly shifting alliances during the game. Sometimes, it's best to let your enemies fight their enemies and just sneak on by without firing a shot.

It always increases the player's interest level if you provide multiple approaches when confronting a situation in a game. For instance, there's one spot in Halo where you're crossing a long bridge-like structure. You can fight your way across it, take an elevator down to ground level, and then climb a pyramid. Or, if you're a bit smarter, you can sneak onto an air vehicle called a

banshee and fly over to the pyramid. But in order to do that, you have to get the invisibility cloak, which is quite a way back from the bridge.

Another approach to the bridge is to fight your way to the middle (or sneak there with the invisibility cloak), and then jump down onto one of the struts that hold up the pyramid structure. But to do so, you have to have good health and you have to have a weapon called a needler. The reason you want the needler is that, as you're running down the strut to ground level, aliens are shooting at you. Because you need to concentrate on running down the narrow strut, you can't aim well. The needler is a heat-seeking weapon that doesn't require much aiming.

If you run down the strut fast enough, you can jump into a pit at the bottom. There you'll find a sniper rifle and a rocket launcher. You can grab those two weapons and go back up the strut. With the sniper rifle, you can kill all the aliens that shoot at you. With the rocket launcher, you can blow up the tank at the base of the pyramid. You can then run back down the strut and steal a vehicle to take you up the pyramid.

The result of all of these game features is that you have the option of just blindly shooting your way through the game or thinking up better strategies. Games that allow the possibility of flexible thinking on the part of the player are much more fun.

One caution here: Making players think is not the same thing as making them guess your clever plot trick. An example will help clarify what I mean.

I once played a game in which the main character was a janitor on a starship. Of course, the starship was invaded by evil aliens and the rest of the crew killed (the janitor was sleeping in the broom closet). The aliens set the ship to blow itself up, so the hapless janitor needed to get off fast. To get him off the ship, you had to get him to open doors to the shuttle bay.

## **Tip**

Giving players many ways to solve problems in games makes the games more fun. It also gives games a longer lifespan. People will play through your game again and again to try and find every possible way to solve each challenge.

At that time, it was normal to type text commands into games (mice weren't common yet). So to get the janitor to open the doors, you had to walk him into the proper control room and type a command that would make him push a button. Unfortunately, the designers of the game decided to be clever here and make the player guess the exact command that was required to get the janitor to push the button. I tried "Push button." No luck. Next I tried "Push door button," "Press button," "Press door button," and so on. It wasn't all that long before I was typing "Press that @#!\$% button you stupid \*%^\$#!"

The people who made this game didn't understand that making players guess the designer's cute trick is not the same as making the player think. The challenge should have been something like finding the card key that gave authorization to open the door (I found it lying on the floor). Making players guess the right set of words to get the door open made them focus on the game's interface, not on the game. That's not good game design.

# Art Skills

You don't have to be an artist to write games. However, it does help. If you're not an artist, perhaps you have a friend who is. Either way, you have to know a bit about how computers draw pictures. As mentioned earlier, computers display images on their screens as bitmaps. They can also store bitmaps in files on disks. You produce bitmap files with a paint program, like the one that comes with Windows. The Windows Paint program is OK for starting out with. However, if you're serious about writing games, I encourage you to eventually invest in a professional paint program.

## Note

Paint programs produce files that are in a variety of bitmap format. These include formats such as Windows bitmap (BMP), Tagged Image File format (TIF), Windows Metafile (WMF), JPEG (JPG), GIF, and many others. The LlamaWords2D game engine supports BMP, JPG, and GIF.

At game companies, game programmers use what they call "placeholder art" to indicate what artwork they need in a game and where it goes. Usually, the placeholder art that programmers create must be the same size as the final art. Other than that, it doesn't have to look much like the final art at all. An artist replaces the placeholder art with something that looks more professional.

If you're writing games on your own, you have to draw all the artwork yourself. This includes all the characters, monsters, guns, balls, hockey sticks, or whatever else you want players to see. The skills you need are the same skills needed by all artists. You can acquire them by taking classes at your local community college or high school. Many art supply stores also have listings of art classes. It's wise to focus specifically on classes that involve producing art on computers.

Even 2D games usually use art that looks 3D. There are two common ways to produce 3D art. The first is to just draw it as a 2D picture that looks like it's 3D. Painters and other artists do this all the time. The other common way to produce 3D art for 2D games is to use a 3D modeling program to produce 3D objects and then render them to 2D. When you render a 3D object, you

essentially take a 2D picture of it. If you want to get into 3D art, I suggest you take a look at a free program called Persistence of Vision (POV) Raytracer. You'll find it on the Web at [www.povray.org](http://www.povray.org).

## **Note**

I've provided a copy of the POV Raytracer software on the CD in the folder \Tools\POV.

# Sound and Music Skills

## Tip

Many people have excellent recording programs and don't even know it. They often come free with your sound card in your computer. If you got a CD with your sound card, take a look at the programs it contains. You might find a feature-rich sound recording program there. For example, many computers have sound cards from Creative Labs. If yours does, check the CD for a program called Creative WaveStudio. It's a great free program for recording sound.

Great games have great music and sound effects. You don't have to be a musician to write games if you can find one to work with you. If you do have a musician for a partner, or you are able to hire one, all you really have to know how to do is play their music during the game. However, the more musical skills you have, the better your game will be. Even just being a hobbyist musician can help you develop a better ear for music. You'll need it when you're selecting music for the different portions of your game. Good music can tremendously intensify the experience of a game. If you doubt this, try playing your favorite game with the sound off. The experience is usually rather lifeless and sterile.

Computers store music in one of two ways. One way is to store a digitized version of the music itself. You usually do this with sound recording programs like Windows Recorder, which comes with Windows. That program will help you get started, but you'll eventually need a more professional program for recording and mixing music.

Digitized recordings of music are generally stored in WAV or MP3 files. WAV files give better sound, but MP3 files are *a lot* smaller. You need to decide which is more important for your game: sound quality or disk space.

If you have an electronic musical keyboard, you can easily record music on your computer. Just plug one end of a cable into the headphone jack on the keyboard and the other end into the line-in jack on your computer's sound card. Use your recording program (even if it's only Windows Recorder) to record whatever you play.

You can use the same technique for digitizing music you play on your electric guitar. Most guitar amplifiers have a headphone jack. Like the headphone output from a musical keyboard, an amplifier's headphone output can be plugged into the line-in jack of your computer's sound card.

The other way that computers store music is in MIDI files. A MIDI file doesn't store the music itself. Instead, it stores a set of commands that tell how to make the music. The computer then acts as a synthesizer and produces the music from the commands.

## **Warning**

The technique of recording music by routing your instrument's headphone output into the line-in jack of your computer's sound card is the penny-pincher's approach. It may not give you the high-quality sound recordings that are common in professional games. If it doesn't, you need to explore other options, such as buying a better sound card or external audio-to-digital conversion hardware.

MIDI files are much smaller than digitized music files. However, not all computers have the same ability to synthesize music that your computer does. Some are better at it than your computer. Others are not as good at it. Either way, you cannot guarantee that the music will sound the same across all computers. For that reason, most game developers use digitized music files rather than MIDI music files. Many will use MIDI instruments, such as electronic keyboards, to produce their music. However, they'll record the MIDI music into an MP3 file so that they know the final product sounds the same on all computers.

Most electronic musical keyboards have a MIDI output jack. That means you can hook your musical keyboard into your computer to record your MIDI music. Virtually all computers have a jack on their sound cards that can be used either for a MIDI instrument or a joystick. You can plug your MIDI keyboard into that. If you want to then digitize the resulting MIDI song, you use a music player like Windows Media Player or Winamp to play the song. At the same time, use Windows Recorder (or something similar) to digitally record the song.

## Cheating Your Way to Great Music and Sound Effects

If you're not a musician, and you're not likely to become one soon, you can still get good music for your games. Many little-known musicians can be found on the Web on sites such as GarageBand.com. They will often donate a song or two to your game to get the free advertising. Some of them sell royalty-free music that is designed for use in games.

Another way to get good music is to use a music generation program. There are several around, but the one I recommend is Band-in-a-Box. It enables you to quickly create great music by doing little more than clicking your mouse a few times.

For sound effects, you can get sound effect generator programs. However, I haven't been particularly impressed by any of the low-cost programs. Instead, you might consider using sound effect collections, which you can purchase on CDs. There are also free collections you can download from the Web.

As with learning art skills, you can take music classes at your local community college or high school. Because the easiest way to get music into a computer is with an electronic musical keyboard, you should probably concentrate on learning to play piano. However, it also helps to know how to play guitar and drums as well.

Whether or not you're a musician, you *must* deal with music and sound effects for all games. Without them, the experience of playing a game becomes extremely sterile and bland. Good music and sound effects are absolutely essential to games.



# Summary

As you can see, it takes some skill to be a game programmer. You must know how to program computers, usually in C++. I'll begin showing how to program in C++ in the next chapter.

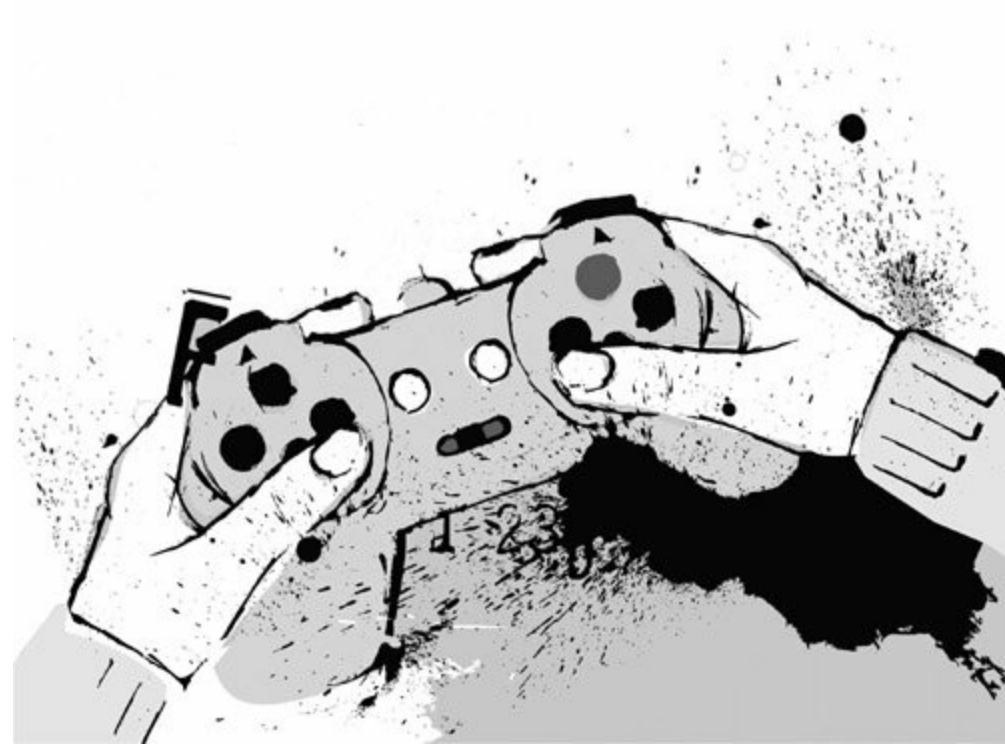
You also need to know the basics of computer graphics and animation. Computer animation works just like movie animation. That is, you show a series of pictures that differ slightly to produce the illusion of movement.

You're probably reading this book because you want to write your own games and not just work for a game company where someone else designs the games. If that's the case, you need to know something about game design. Specifically, you need to make your games unique, plausible, and playable. Also, players will enjoy your game much more if you make them think.

You'll also need to know a bit about art and music. If you are not an artist or musician, it's a good idea to find friends or relatives who can help you with your game or buy collections of art and music that you can use in your game. It really is possible for two people, a programmer and an artist/musician, to develop a game that takes the industry by storm. To begin learning the programming skills you need to develop the next killer game, read on to [chapter 2](#).

# Chapter 2. Writing C++ Programs

[chapter 1](#) discussed how C++ programs, such as games, are built. Specifically, it showed that you write C++ statements in text files, use a compiler to convert the text files to object code, and then use a linker to convert the object code to an executable program. In this chapter, you'll get hands-on experience with how this process works. You'll start by writing a simple C++ program; then you'll compile and link your program.



# Introducing the Dev-C++ Compiler

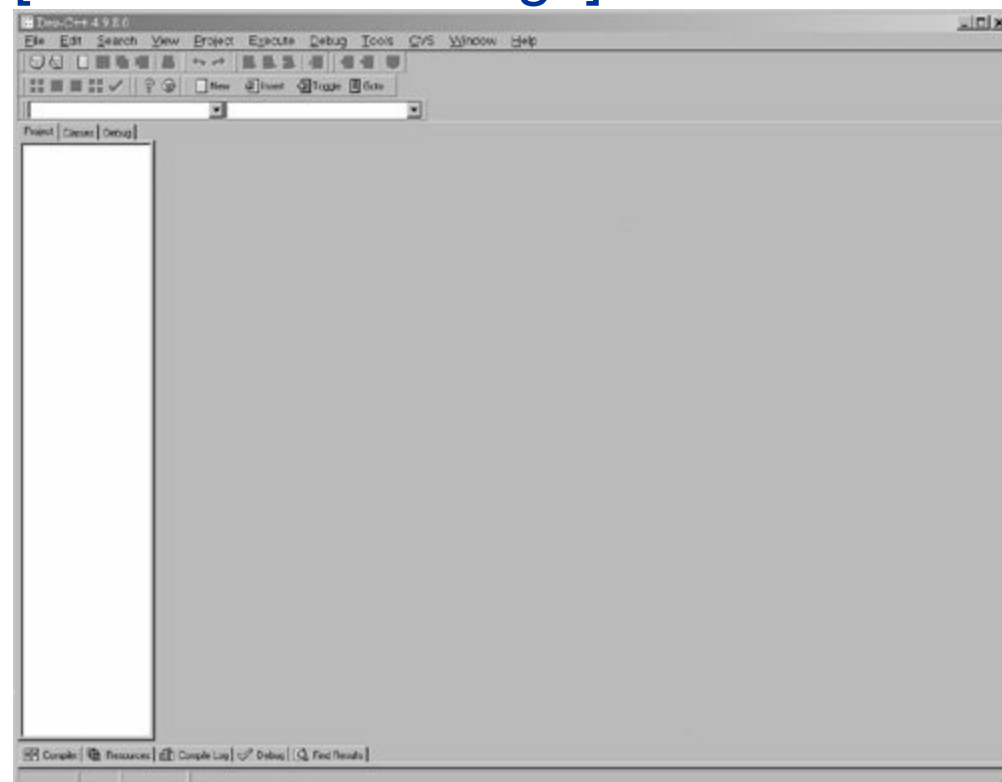
Compilers today are more than just programs that translate source code into object code. They are complete integrated development environments (IDEs). With compilers like Dev-C++, you can write, compile, link, and debug your programs all in one development environment. So before we get into how to write C++ programs, let's take a brief look at Dev-C++.

At this point, I assume that you've installed Dev-C++ on your computer. If you haven't, please do it now. I'll wait.

Okay, now that you've installed Dev-C++, start it up. When you do, the program's main window resembles [Figure 2.1](#).

**Figure 2.1. The Dev-C++ program's main window.**

[\[View full size image\]](#)



We'll see what each part of the Dev-C++ main window is used for in the next few sections.

## Warning

I think it's worthwhile to take a moment to repeat a warning that I gave in [chapter 1](#). All of the C++ source code you will encounter in this book is designed to work with Dev-C++. It may or may not compile with other compilers, such as Visual C++. Ideally, C++ source code should work with all compilers, but in reality, subtle differences exist between compilers produced by different companies. Therefore, I strongly recommend that you use Dev-C++ for all the programs in this book. After you become an experienced C++ programmer, you should have no problem compiling this book's programs with Visual C++ with only minimal changes.

## Can I Use Other Compilers?

You do not have to use the Dev-C++ compiler if you already have another one. For example, many programmers today use Microsoft's Visual Studio.

Commercial compilers such as Visual Studio are great, but they also cost a lot of money. Dev-C++ gives you the tools you need to get started, but it doesn't have all of the fancy Web programming tools, database access tools, and so on that you get with Visual C++. Most of these tools are not needed for game development. By itself, Dev-C++ does pretty much everything you need for game development in one IDE.

I recommend you start with Dev-C++. If you find that you need a bigger, more powerful tool set, you can always spend the money for a commercial compiler later.

## Creating a Project

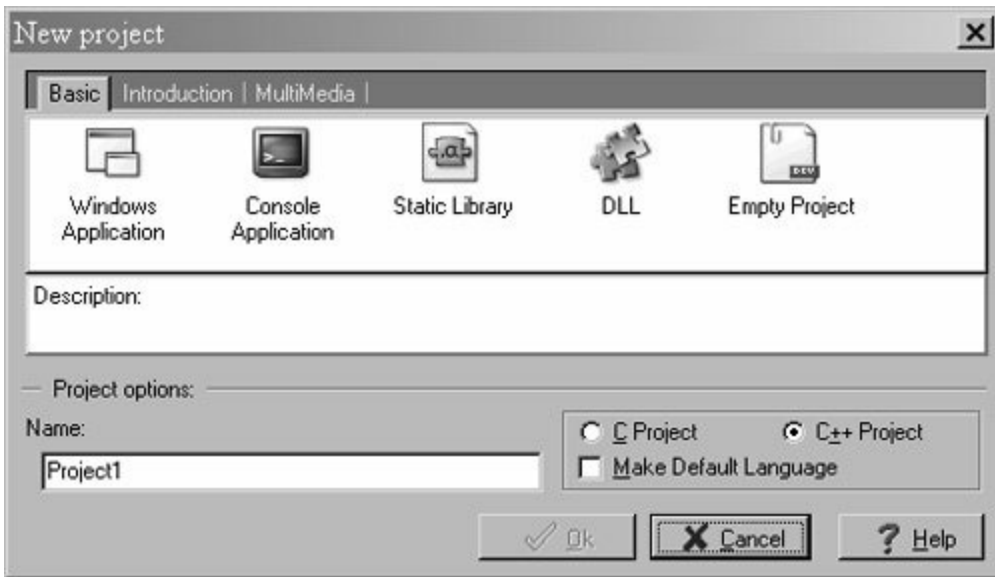
The first step in using Dev-C++ to write a C++ program is to create a new project. Here's how it's done.

### Creating a Project File

Start by choosing File from the main menu. Now select New and then Project. The dialog box in [Figure 2.2](#) appears.

### **Figure 2.2. The New Project dialog box.**

[\[View full size image\]](#)



There are several options in the dialog box shown in [Figure 2.2](#). Let's keep things simple and ignore most of them for now.

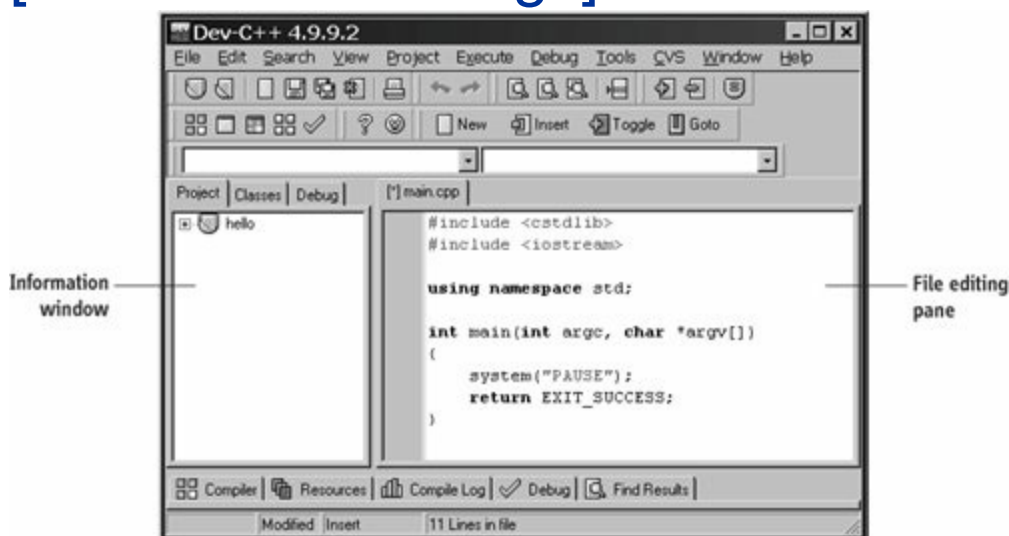
2. Click on the Console Application icon.
3. In the Name box, type **hello** and then click the Ok button.

In the dialog box that appears, choose a directory to save your project file

4. into. The main window now resembles [Figure 2.3](#)

## Figure 2.3. The Dev-C++ main window after creating a new project.

[\[View full size image\]](#)



The tabbed box down the left side of the Dev-C++ window in [Figure 2.3](#) contains information about the current project. If you click the + sign next to the folder icon by the word *hello*, Dev-C++ shows a list of all files in your project. Right now, the only file in the project is called `main.cpp`.

The larger pane on the right side of the window has a tab with the filename in it. This is the file editing pane. In this pane, you'll see your first C++ program Dev-C++ generated it for you. This program doesn't actually do anything. However, it contains all of the fundamental parts of a C++ program. We'll look at each one of them shortly.

## Writing Programs

Let's go through the program in [Figure 2.3](#) line by line. It's presented here in [Listing 2.1](#) for your convenience.

### Listing 2.1. A very basic C++ program

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     system("PAUSE");
9     return EXIT_SUCCESS;
10 }
```

The first two lines in the file `main.cpp` contain the following text:

```
#include <cstdlib>
#include <iostream>
```

These statements are called [include statements](#). They help you access code in the C++ Standard Libraries. Every C++ compiler comes with this standard set of libraries. The libraries contain object code that performs many common

tasks that almost all C++ programs do. The main job of your linker is to link your program to the C++ Standard Libraries. You can also use it to link to other libraries. When you use other libraries, you will need to put special include statements in your program, similar to the ones shown in [Figure 2.3](#).

On line 4 of the file main.cpp, you'll find this statement:

```
using namespace std;
```

At this point, I'm not going to explain what this does. However, I will say that you need to put it at the beginning of most of your .cpp files. We'll revisit this statement, and others like it, in later chapters.

Lines 610 of main.cpp contain a block of code called the `main()` function. The [main\(\) function](#) is the heart of every C++ program. I'll explain it in detail soon. Right now, we'll take a look at how to add C++ statements to the `main()` function.

## Warning

I added line numbers to [Listing 2.1](#) to make it easy to discuss the code. These line numbers are *not allowed* in C++ programs. *Do not type them into your program*. It will not compile. They're just shown in the listing for convenience.

## Modifying the Main() Function

Delete the text on line 8 of main.cpp. Do this by using your mouse cursor **1.** to highlight both lines and then pressing the Delete key on your keyboard.

Change the `main()` function so that it matches the code in [Listing 2.2](#). The changes are shown in **bold** text. Of course, they won't be bold when you **2.** type them into Dev-C++; they're just shown in bold here to make it easier for you to see what to add.



## Listing 2.2. The new version of main()

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     string yourName;
9
10    cout << "Please type your name and press Enter: "
11    cin >> yourName;
12
13    cout << endl << "Hello, " << yourName << endl << endl;
14
15    cout << "Press c and then Enter to continue...";
16
17    char justWait;
18    cin >> justWait;
19
20    return (EXIT_SUCCESS);
21 }
```

3. Save main.cpp by selecting File and then Save from the main menu. The dialog box that appears enables you to enter a filename. In this case, you can just accept the name main.cpp by clicking the Save button.

### Note

I've compiled all of the code listings in this chapter into executable programs. You'll find them on the CD in the folder\Source\Chapter02\Bin.

### Note

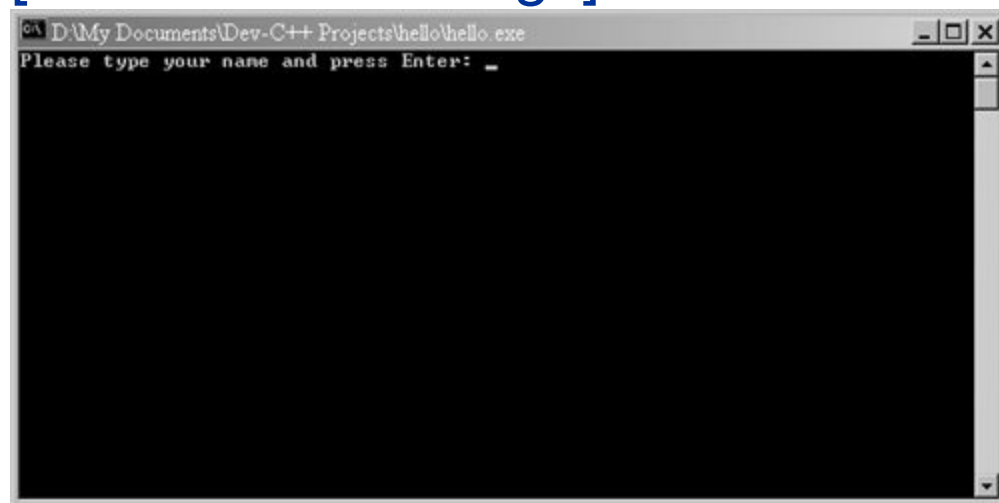
When the length of a line of code exceeds the character limit for this book, you'll see a little continuation arrow on the next line. Treat this as a single line of code: don't use a hard return.

## Compiling and Linking Programs

After you make the changes in [Listing 2.2](#), press the F9 key on your keyboard to compile, link, and run the program. If you typed it correctly, you'll see a window that looks like [Figure 2.4](#)

**Figure 2.4. The output from your first program.**

[\[View full size image\]](#)



Congratulations! You've compiled, linked, and run your first C++ program. This is a console program. If you've been using computers a while, you know that it is essentially no different than old DOS programs. [Figure 2.4](#) shows that this program is waiting for you to type your name and press Enter. After you do, it greets you with a friendly hello. It then asks you to press the C key followed by Enter. When you do, the program ends.

### Tip

If you prefer, you can just compile and link your program without running it. Do this by pressing Ctrl+F9. You can accomplish the same thing by selecting Execute from the main menu and then choosing Compile.



# Programming in C++

Let's analyze the program we wrote in [Listing 2.2](#). We're going to ignore the first five lines of the listing. For now, we'll just jump down to line 6, which is the beginning of the `main()` function.

## The `main()` Thing

The `main()` function includes everything from line 6 through line 21. `main()` is the *program entry point*. That's a fancy way of saying that the `main()` function is the place where your C++ programs start executing. The program starts at `main()` and executes one statement after another.

Actually, I should be more specific. When a C++ program executes, it starts with the first statement inside the `main()` function. The statements inside the `main()` function are all contained within the opening and closing braces, which are the symbols `{` and `}`.

### Note

Notice that every statement in the `main()` function in [Listing 2.2](#) ends with a semicolon. Almost all statements in C++ must end with semicolon. I'll point out the exceptions as we go along.

The left brace appears on line 7. Programmers often call it the *opening brace* of the function. The *closing brace*, which of course is the right brace, appears on line 21. All statements in the `main()` function must appear between the opening and closing braces. The first statement in `main()` is the declaration of a *variable*. The next section explains exactly what that means.



## Compiling the Example Programs Yourself

As mentioned previously, you'll find a folder for every example program in this chapter on the CD in the \Source\Chapter02 folder. There's also a folder in \Source\Chapter02 called Bin. The Bin folder contains the compiled executables for every example program in [chapter 2](#). However, if you would like to compile these programs yourself, use the following procedure:

1. Create a folder on your hard disk for the program. You can put it wherever you like.
2. Copy the .cpp file you want to compile into the folder you just created.

Create a project for the program by starting Dev-C++ and selecting File from the main menu. Next, select New and then Project. All of the programs in this chapter are console programs, so

3. choose Console Application. In the Name box, type the name of the project. In the dialog box that appears, navigate to the folder you created in step 1 and save the project there.
4. From the File menu, select Close to close the edit window.

In the pane on the left side of the Dev-C++ window (this is called the Project pane), you'll see a folder with a plus sign on its left and the name of the program on its right. Click the plus sign. When you do, you'll see the filename main.cpp. Right-click that filename and select Remove File from the menu that appears.

5. Right-click the folder in the Project pane and select Add to Project from the menu that appears. This displays the Open File dialog box.

In the Open File dialog box, navigate to the folder you created in step 1 and select the .cpp file you copied into it in step 2.

6. From the Execute menu, select Compile & Run. The program compiles and then runs.

## Variables

Line 8 of [Listing 2.1](#) declares a variable. Think of a variable as simply a container, much like a mailbox. People put mail in mailboxes and take it out later. Each mailbox has an address. Variables work very much like that. Every byte in your computer's memory has an address, which is a number. Memory addresses can be specified either in binary or hexadecimal. Hexadecimal is yet another number system it's base 16.

You and I are humans. We don't want to work with binary or hexadecimal address numbers if we can avoid it. It's far easier for us to give a memory location a name and use that instead of an address. We can let the compiler translate the variable name into an actual address in memory. Again, it's much like a mailbox. After a mail carrier has been on a mail route for a long time,

she knows that the address 1234 South 5<sup>th</sup> Street is where the Wobbenflatz family lives. When she sees a letter for the Wobbenflatzes, she can put it in the right mailbox without having to look up the address. She uses the name Wobbenflatz for the mailbox rather than the address 1234 South 5<sup>th</sup> Street. You and I do the same with variables in programs. We use the variable names rather than the addresses because they're easier for us to deal with.

The variable on line 8 is called `yourName`. It's a spot in memory where the program stores your name. And once again, it works very much like a mailbox. The program can store your name in the variable and leave it there like people store mail in mailboxes. Later, when the program needs the data, it pulls the data out of `yourName` just as you'd retrieve your mail.

Notice that the variable name `yourName` describes the data that gets stored in the variable. All variable names should be like this. Variable names must conform to the following rules:

Variable names *must* start with a letter. The letter can be uppercase or lowercase.

Most compilers let variable names be as long as 128 characters; some allow them to be longer. However, virtually all compilers require that all variable names be unique in the first 32 characters. So if two variable names use exactly the same letters in the same order for the first 32 characters, the compiler sees them as the same name. It does not check any more than 32 characters in the name.

After the first character, variable names may contain any letter or number.

Variable names can also contain the underscore character (`_`).

Variable names are case sensitive. Therefore, `yourName` is not the same as `YourName` or `YOURNAME`. The compiler sees these as three different variables.

C++ commands, which are called keywords, cannot be used as variable names. We'll talk more about C++ keywords later.

Every variable must have a data type. More on this in a moment.

Spaces are not allowed.

If your variables do not follow these rules, your program won't compile.

As noted above, every variable must have a data type. The data type specifies exactly what kind of information the variable can hold. For instance, the variable `yourName` on line 8 of [Listing 2.2](#) is of type `string`. A string is a collection of characters, such as the upper- and lowercase letters of the alphabet. As you read through this book, you'll see data types that contain integers, floating-point (real) numbers, logical values, and many other kinds of data. When you declare a variable to store each of these different kinds of data, you must specify the appropriate data type. [Table 2.1](#) shows many of the variable types that are built into C++.

**Table 2.1. The Most Common C++ Built-In Data Types**

Type	Description
<code>int</code>	Integer. A number with no decimal point. The values 0, 1, -57, and 1000 are examples of integers. Integers usually contain 4 bytes of data.
<code>short</code>	Short integer. Same as an integer, but it usually contains 2 bytes of data.
<code>long</code>	Long integer. Same as an integer and usually contains 4 bytes of data, but with some compilers, it can contain 8 bytes of data.
<code>unsigned</code>	Unsigned integer. Can only contain values that are greater than or equal to 0. The unsigned integer usually contains 4 bytes of data.
<code>float</code>	Floating-point number. A number with a decimal point. The values 0.0, 1.2, 0.00012345, and -10.5 are all examples of floating-point numbers. Floating-point numbers usually contain 4 bytes of data.
<code>double</code>	Double-sized floating-point number. Same as a floating-point number, but it usually contains 8 bytes of data.
<code>char</code>	Character. Contains characters. The <code>char</code> type contains 1 byte of data.
<code>bool</code>	Boolean value. Can be set to either true or false. The <code>bool</code> type usually contains 1 byte of data.

Notice that the type `string` is not in [Table 2.1](#). The `string` type is added on by the

C++ Standard Libraries. The C++ Standard Libraries contains a huge number of very useful types. We'll discuss some of the most common as we write our example programs. You'll find more information on the C++ Standard Libraries in the books listed in Appendix C.

You may be wondering if you have to use the style of variable name that I use in this book. The answer is no. C++ does not force you to use a particular style of variable name. As a result, you can create names such as `yourName`, `your_name`, `YourName`, and `YOURNAME` in your programs. This leads to confusion for new programmers.

Some common questions they ask are:

What letters should I capitalize and what should be lowercase?

Should I use underscores?

Do most people use the style in this book?

Should I use Microsoft's Hungarian notation?

The most common style in C++ programming is the style I used in [Listing 2.2](#), which is called camel notation. In this style, the first word of the variable is lowercase. All subsequent words start with capital letters.

## Note

There are other data types built into C++ that are not presented in [Table 2.1](#). I'll discuss them later as needed. These are the data types that you'll use most often in your games.

I strongly encourage you to use camel notation. Programmers tend to use it because everyone recognizes that anything in that notation is a variable. It's often the case that many programmers are working together on games. Other programmers on the project can just look at a variable name in camel notation and tell it's a variable. That helps clarify what they're reading.

Other variable name styles do exist. The one you'll run into most often is Microsoft's Hungarian notation. They created it for the C programming



language, which was the language that C++ is based on.

Hungarian notation puts an abbreviation at the beginning of the variable name that describes what type of data is in the variable. For example, a string variable would be `sYourName`. However, for reasons that have to do with the C++ language, it could also be `strYourName`, `cpYourName`, `arrYourName`, or even `bpYourName`. This shows one of the big problems with Hungarian notation—it's not standardized in the least.

Hungarian notation made sense in C, but it is not useful in C++ for a lot of reasons that I won't go into now. I'm just going to say that in C++, Hungarian notation causes more problems than it solves. You'll do well to avoid it.

## Output to the Screen

So far, we've seen that the program in [Listing 2.2](#) contains a `main()` function. The first thing that this function does is declare a variable so it can store your name. Its next task, on line 10, is to output a string of characters to the screen. The string it outputs is

"Please type your name and press Enter: "

I typed the string directly into the program in quote marks. Strings that are typed directly into programs are called [literal strings](#). Literal strings must always be in quote marks.

To get output to the screen, C++ uses a [stream](#). Streams in C++ contain data that flow into and out of the program. When you want to send data from the program to the screen, you have to insert the data into an output stream. The data flows out from the program to the screen.

Streams can flow out to almost any device. The C++ Standard Libraries define a special stream called `cout` that flows to the screen. Any time you insert data into the `cout` stream, it automatically goes to the screen. You don't have to worry about how; all you need to do is insert the data. The C++ Standard Libraries take care of everything else.

### Note

The stream `cout` is pronounced *see-out*.

To insert data into a stream, your program uses the insertion operator. The insertion operator is two less-than signs (<<). It's shown on line 10 between the `cout` stream and the literal string.

Sending information from the program to the screen is very much like standing by a stream of water that flows into a nearby lake and dropping in a toy boat. All you need to do is insert the boat into the stream and let the stream of water take it out to the lake. Similarly, we insert data from C++ programs into data streams. The streams carry the data to their destination. The destination of the `cout` stream is the screen.

If we were to translate line 10 of [Listing 2.1](#) into English, we would read it something like this: "Insert the literal string specified in these quote marks into the `cout` stream so that it can go to the screen."

Lines 13 and 15 show more output to the screen. On line 13, the program sends a special value called an `endl`. This moves the cursor on the screen to the beginning of the next line. So line 13 essentially prints a blank line, and then the string "Hello," (the word *Hello* followed by a comma and a space). Next, it prints the string that the variable `yourName` contains, followed by an `endl`. Finally, it prints another blank line.

## Input from the Keyboard

Streams aren't just for outputting information to the screen. C++ programs also use them to get input from the keyboard. Line 11 of [Listing 2.2](#) demonstrates how this is done.

On line 11, the `main()` function uses a stream called `cin`, which is defined by the C++ Standard Libraries. This stream is connected to the keyboard. Data flows from the keyboard to the program through the `cin` stream. C++ programs must extract the data from the input stream using the extraction operator, which is two greater-than signs (>>).

The statement on line 11 extracts characters from the `cin` stream. The characters it extracts are the characters that the user types in.

### Note

The stream `cin` is pronounced *see-in*.

It is important to note that the extraction operator does not get any input until after the user presses the Enter key. When the program uses the extraction operator, it just waits for input until the user presses Enter. This is fine for entering strings, but not so great for entering single characters. Line 15 prompts the user to press the C key and then press Enter. This causes the program to wait, giving the user time to read the message printed on line 13.

Normally, you do not want your programs to behave like this. If your program pauses, you usually want the user to just press one key to make it continue. We'll see how to do this later.

Although streams such as `cin` and `cout` are fine for simple input and output, we do not use them in games. We learned streams here because they are the simplest form of input and output. Also, all file input and output is based on streams. If you know how to use streams, writing saved games to files (and reading them back from files) is not all that much more complex than writing strings to the screen.

## Tip

Use the insertion operator, which is `<<`, to make data flow out of a program. Use the extraction operator, the `>>` symbol, to get data into a program.

## Streams and Include Statements

I mentioned earlier that the streams `cin` and `cout` are defined by the C++ Standard Libraries. To get access to these definitions, you must use include statements in your `.cpp` files. The include statement on line 2 of [Listing 2.2](#) includes a file from the Standard Libraries called `iostream`. That is the file that contains the definitions of the `cin` and `cout` streams. To use `cin` and `cout`, you must put the statement

```
#include <iostream>
```

at the beginning of your .cpp files.

You might be wondering why we need include statements when the linker connects our programs to the Standard Libraries. The answer is that programs must have the definitions of `cin` and `cout` for the compiler so that the compiler knows that we're using those streams correctly. The linker links our programs to the actual library code that implements the streams. The definitions tell how they are used. The implementation actually performs the work. If our programs don't include the definitions, they won't compile. If our programs don't link to the implementations, they won't work.

## Factoid

The LlamaWorks2D game engine, which is included on this book's CD, uses the OpenGL graphics library. That means that all of your programs that use LlamaWorks2D must link to OpenGL. Fortunately, you do not have to tell the linker that. When you followed the installation instructions in the Introduction, you configured the linker to automatically link to the OpenGL libraries whenever you compile an OpenGL program.

In general, this is how things work in C++. All code libraries provide definitions and implementations. The definitions are in files with an extension of .h or .hpp. There is a trend lately to put the definitions in a file with no extension on the filename. The new version of the C++ Standard Libraries uses this style. So the filename is `iostream` rather than `iostream.h` or `iostream.hpp`.

The implementations for code libraries are always in object files that have extensions such as .a or .lib. Your program must include the definition files and link to the implementation in the object files.

## Functions

I've said repeatedly that `main()` is a function. However, I haven't really defined a function. A function is a chunk of code that your program can use over and over. [Listing 2.3](#) illustrates what I mean.

## Note

Just a quick reminder: Don't type in the line numbers you see in the listings. C++ does not allow line numbers in actual source code. They're just there in the listing to make it easy to talk about the individual statements in the program.

### Listing 2.3. A function called by main()

```
1  #include <iostream>
2  #include <stdlib.h>
3
4  using namespace std;
5
6
7
8  int Squared(int numberToSquare);
9
10
11
12 int main(int argc, char *argv[])
13 {
14     // Prompt the user for an integer.
15     cout << "Please type an integer number and then press Enter: ";
16
17     // Declare the integer variable.
18     int theNumber;
19
20     // Get the integer from the user.
21     cin >> theNumber;
22
23     // Declare another variable.
24     int theNumberSquared;
25
26     // Square the number.
27     theNumberSquared = Squared(theNumber);
28
29     // Print out the answer.
30     cout << theNumberSquared << endl;
31
32     //
33     // Now do it all again.
34     //
35
36     // Prompt the user for an integer.
37     cout << "Please type an integer number and then press Enter: ";
38
39     // Get the integer from the user.
40     cin >> theNumber;
```

```
41
42 // Square the number.
43 theNumberSquared = Squared(theNumber);
44
45 // Print out the answer.
46 cout << theNumberSquared << endl;
47
48 system("PAUSE");
49
50 return (0);
51 }
52
53
54 int Squared(int numberToSquare)
55 {
56 int theNumberSquared = numberToSquare * numberToSquare;
57
58 return (theNumberSquared);
59 }
```

[Listing 2.3](#) shows a function called `Squared()`. It's defined on lines 54-59. Normally, the program executes each statement in `main()` one after another. However, each time `main()` calls the `Squared()` function, the program jumps to the beginning of `Squared()` and executes the statements in `Squared()`. The `main()` function can call the `Squared()` function as many times as needed. As you can see, there are calls to `Squared()` on lines 27 and 43.

## Tip

It's usually best to name your functions by capitalizing the first letter of each word in the name and using lowercase for all the other letters. Here's an example: `ASampleFunctionName()`. When other programmers see this, they recognize it as a function name from the style of capitalization used.

The next few topics use [Listing 2.3](#) to discuss how to create a function, how to pass information to it, and how to get information out of it.

## Creating Functions

Every function has a few required elements. First, functions must have names. The rules for naming functions are the same as the rules for naming variables.

People use different styles for naming their functions than they use for naming variables. For example, some people use all lowercase, while others use all uppercase. In general, though, most programmers name their functions by capitalizing the first letter of each word in the name and using lowercase for all the other letters.

It's probably pretty obvious that the name of the function in [Listing 2.3](#) is `Squared()`. Any time the name `Squared()` appears in another function, such as `main()`, we say that the program is calling or invoking the `Squared()` function. Program execution actually jumps to the beginning of `Squared()`. The program then executes all the statements in `Squared()`. When it hits the end of `Squared()`, which is marked by the closing brace (`}`), the program jumps back to the point where it called `Squared()`. It then resumes executing statements from that point.

## Passing Values to Functions

In addition to a name, functions must have a parameter list. Programs use a function's parameter list to send information to the function. If you look on line 54 of [Listing 2.3](#), you'll see the `Squared()` function's parameter list in parentheses. There is only one item in the parameter list: `numberToSquare`.

Function parameters are very much like variables. You use them in just the same way. The primary difference between a variable and a parameter is how they get their values. Your program usually uses the assignment operator (`=`) to assign a value to a variable. Parameters get their values when the function is called. Let's look at [Listing 2.3](#) to see how this works.

On line 27, you can see that the `main()` function calls `Squared()`. The parentheses next to the name `Squared()` contain the variable `theNumber`. This means that `main()` is passing the value in `theNumber` as the parameter to the `Squared()` function. This causes the value in `theNumber` to be copied into the `Squared()` function's `numberToSquare` parameter. Program execution then jumps down to the `Squared()` function. The `Squared()` function executes and uses the parameter `numberToSquare` just as if it were a variable. When `Squared()` ends, program execution jumps back to line 27 of [Listing 2.3](#).

The long and short of this is that passing a value as a parameter means that the value is copied into the parameter. The function being called can then use the parameter as if it were a variable.

## The Body of a Function

The reason we write functions is to get them to do work for us. That means we need to have commands, or statements, inside the function. A function's statements are all contained in the [function body](#). The function's body consists of all of the statements between the opening and closing braces (the { and } symbols).

When a function is called, program execution jumps down to the beginning of the function and executes all of the statements in the body. When the program reaches the end of the function's body, it jumps back to the point where the function was called.

The statements in the body of the `Squared()` function are on lines 56-58. The statement on line 56 multiplies the number in `numberToSquare` by itself. That is how you calculate the square of a number. The statement stores the answer in a variable called `theNumberSquared`. Even though this variable has the same name as a variable in the `main()` function, they are two different variables. We'll talk more about this shortly.

The last statement in the body of the `Squared()` function sends the answer back to the `main()` function. Let's take a closer look at how that's done.

## Note

It is possible to have more than one parameter in a function's parameter list. If there is more than one, you separate each parameter in the list with a comma. Each parameter must have its own name and type.

## Returning Values from Functions

The purpose of the `Squared()` function in [Listing 2.3](#) is to calculate the square of the number that is sent in through the parameter list. That means we want `Squared()` to give us back an answer. That answer is a piece of data. All data in programs must have a type.

When we send data from a function back to the point where the function was called, we say that we're returning a value from the function. When the `Squared()` function in [Listing 2.3](#) is called on line 27, the program copies the value in the parameter list into the parameter `numberToSquare`. The program then executes the



statements in the function's body. The last statement is a `return` statement. The `return` statement tells the program to return a value. The `return` statement is followed by the value to return. In this case, the value is the number in the variable `theNumberSquared`. So whatever's in `theNumberSquared` gets sent back to line 27, which is where `Squared()` was called from.

Notice that the return type of the `Squared()` function is `int`, which means integer. That is also the type of the parameter `numberToSquare`. Function return types do not have to be the same as any of types of the function's parameters. They don't have to match. In this case, they do match, but that's not a general requirement for all functions.

## Note

It's not strictly true that *all* data in programs *must* have a type. It is possible to create data without a type, but it can cause complications. I strongly recommend that you don't create data without a type until you feel you're a *strong* C++ programmer.

Let's summarize what we've covered about functions so far. Every function must have a name, a parameter list, a return type, and a body. The name distinguishes the function from all other functions in the program. The parameter list is a way of getting information into the function. The return type specifies the type of the data the function returns. Returning data is how we get information out of functions.

## Reminder

An integer has no decimal point. 0, -42, and 1000 are examples of integers.

The `Squared()` function has a name and a parameter list containing one parameter. When the function's name appears in `main()` on lines 27 and 43, the program copies the value in the parameter list into the `numberToSquare` parameter. The program then jumps to the beginning of the `Squared()` function's body on line 56. It performs the calculation on line 56 and then encounters the `return`

statement on line 58. The `return` statement tells the program to send the value in `theNumberSquared` back to the place where there `Squared()` function was called from. If we look at lines 27 and 43, we see that they both use the assignment operator (the `=` sign) to assign the value that comes back from `Squared()` into a variable. This is what we often do with the return value of a function. You could also use the return value in a calculation, or print it to the screen with `cout` and the insertion operator (`<<`).

## Factoid

Every function must have a name, type, parameter list, and body.

## Scope in Functions

One thing that [Listing 2.3](#) teaches is that variables in different functions can have the same name. The reason for this is that they have different [scope](#).

So what is scope?

The scope of a variable is the portion of the code over which the variable can be seen and used.

Say what?

Let's go back to [Listing 2.3](#) again to get a better handle on what scope is. The `main()` function declares a variable called `theNumberSquared` on line 24. The `Squared()` function declares a variable called `theNumberSquared` on line 56. These are two different variables because they each have a different scope. The variable declared on line 24 can only be accessed by statements in the `main()` function. Technically, we say that the scope of that variable is limited to the `main()` function.

On the other hand, the variable declared on line 56 can only be accessed by statements in the `Squared()` function. Therefore, its scope is limited to the `Squared()` function.

Any statements in the `main()` function that refer to a variable called `theNumberSquared` are referring to the one declared on line 24. Statements in the `Squared()` function that refer to a variable called `theNumberSquared` are referring to the one declared on

line 56. The compiler never confuses the two.

As you can see from [Listing 2.3](#), the scope of any variable is limited to the function in which it is declared. It cannot be seen outside that function. This enables us to use variables of the same name over and over in many different functions. You'll see the importance of this in later chapters.

Everything in a C++ program has scope. For example, the function `Squared()` has a specific scope. It can be accessed by any other function in the program. Most functions have this sort of scope. We can make functions that can only be seen by certain parts of the program. As we'll see in [Part 2](#), there are definite reasons for doing so.

For now, though, we won't fuss much with scope. The variables we'll declare will only be visible in the functions in which they're declared. The functions we'll write will be visible throughout the entire program in which they appear. We'll continue in this fashion until [Part 2](#).

## Warning

It's not absolutely true that variables cannot be seen outside the functions where they are declared. However, making them visible outside their natural scope is an advanced programming topic that most C++ developers never need. I strongly recommend that you avoid changing the natural scope of variables.

## Prototypes

Take a look at line 8 of [Listing 2.3](#). Notice that it contains a statement that looks just like the first line of the `Squared()` function. This is a special type of statement called a function *prototype*. A prototype is like an introduction to a function. It gets the compiler and the function acquainted with each other. A fancier way of saying this is that the prototype is a template of how the function looks and acts. It describes the function's name, parameter list, and return type.

You're probably asking, "What are prototypes for?"

When a compiler compiles a program, it does so line by line. When it compiles

`main()`, it examines each line and determines whether it is correct C++. If the statement is correct, the compiler translates it into object code. If the statement is not correct, the compiler spits out an error or warning message depending on how bad the problem is.

Suppose you're the compiler. You start at the beginning of `main()` and go through each statement. When you get to line 27 you see that the `main()` function calls a function named `Squared()`. Do the parentheses contain the correct number of parameters? Is the parameter the correct type? What about the return value? Can it be stored in an `int` variable, or is that an error?

The only way the compiler can answer such questions is to have a description of the `Squared()` function. That's what the prototype is for. It describes the parameter list and return type for the `Squared()` function. Because the prototype appears at the beginning of the file, we can use the `Squared()` function anywhere in the program. The compiler knows whether we're using it correctly. If we're not, it can let us know.

You'll need prototypes for all of the functions that you write. In later chapters, I'll show you how to organize the prototypes so they can be used easily and efficiently. There's a bit of a trick to it, but it's not hard to do if you just follow the few simple rules we'll discuss later.

## A Few Final Words about [Listing 2.3](#)

There are a couple more items in [Listing 2.3](#) worth discussing before we move on. First, notice that line 48 shows a call to the `system()` function. The `system()` function is provided by the C++ Standard Libraries. Its prototype is in the file `stdlib.h`.

The `system()` function takes one parameter. That parameter is a string that `system()` passes to the operating system. On line 48, the string that is passed is the PAUSE command. This is an old DOS command that was used a lot in batch files.

### **Factoid**

The name `stdlib` stands for "standard library."

The PAUSE command causes DOS, or a DOS window under Windows, to pause and wait for the user to press a key. This is a different way to add a pause to a program than we saw in [Listing 2.1](#). Either one works. You can use whichever you like better.

An item in [Listing 2.3](#) that doesn't appear in [Listing 2.2](#) is the comment. Programmers put comments in their programs to describe what they're doing. For example, line 14 contains a comment. It tells the reader that the program is about to prompt the user for an integer.

Comments in C++ programs begin with a pair of slash marks (`//`). Whenever the compiler sees a pair of slash marks on a line of code, it knows that the line contains a comment. The compiler always ignores comments; they are there strictly for people to read. Comments that begin with a `//` symbol end at the end of the line. They do not span more than one line of text.

Another interesting item in [Listing 2.3](#) that is worth discussing is the fact that `main()` is a function. Therefore, it has all of the things that any other function has. Notice that `main()` has parameters. These parameters enable your program to receive strings as parameters when it starts up. For example, suppose you ran this program from the command line of a DOS window with the following command:

```
prog_02_03_2 this that theOther
```

What this does is run the program `Prog_02_03.exe`, which is shown in [Listing 2.3](#). It also passes the strings `"this"`, `"that"`, and `"theOther"` into `prog_02_03` as parameters in the parameter list.

Passing parameters from the operating system into `main()` used to be an important technique for all programmers. However, now that we're not using operating systems based around command lines any more, it is much less important. We won't discuss how it's done. However, if you want to know more, you'll find information on it in the C++ programming books I recommend in Appendix C.

The last item we should mention is the fact that `main()` returns a value like any other function. On line 50, it returns the value 0. Returning 0 tells the operating system that the program ended with no errors. All operating systems expect this, so you should put it at the end of all of your programs. If your program wants to return another value besides 0, it can. If it does, it usually means that an error occurred. Typically, the error values that `main()` returns are

negative numbers.

## Note

You can create comments that span multiple lines of text in C++ programs. To do so, start the comment with slash followed by an asterisk (`/*`). End the comment with an asterisk followed by a slash (`*/`).

The libraries that come with the Dev-C++ compiler define a special value called `EXIT_SUCCESS`. It is equal to 0. If you create a new project in Dev-C++, the `main()` function it generates for you returns the value `EXIT_SUCCESS`. So your programs can return `EXIT_SUCCESS` or 0. It's up to you which you want to use.

# Essential Math Operators

I know a lot of people don't like math very much, but you have to be able to do some math to write games. Fortunately, many types of games don't require heavy-duty math. In fact, you can do a large amount of programming with nothing more than the basic math operators: addition, subtraction, multiplication, and division.

## Note

You may have noticed that each line of code in the listings in this book is rather short. Because of the limitations of the printed page, I can have only about 70 characters per line of code (including spaces). In your programs, you have no such limitations, so you can make each line of code as long as you want it. However, most programmers keep their code to 80 characters per line to make them more readable.

## Addition, Subtraction, Multiplication, and Division

The symbols you use for addition and subtraction in C++ programs are the symbols you'd expect, + and -. For division, you use the slash (/). Multiplication uses the asterisk (\*) rather than an x. Let's do a quick program ([listing 2.4](#)) to demonstrate.

### Listing 2.4. A short demonstration of +, -, \* and /

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     int anInt, anotherInt;
9
10    // Prompt the user for an integer.
11    cout << "Please type an integer and then press Enter: ";
12    cin >> anInt;
```

```

13 cout << "Please type another integer and then press Enter: ";
14 cin >> anotherInt;
15
16
17 cout << anInt << " + " << anotherInt << " = ";
18 cout << anInt + anotherInt << endl;
19 cout << anInt << " - " << anotherInt << " = ";
20 cout << anInt - anotherInt << endl;
21 cout << anInt << " * " << anotherInt << " = ";
22 cout << anInt * anotherInt << endl;
23 cout << anInt << " / " << anotherInt << " = ";
24 cout << anInt / anotherInt << endl;
25
26 float aFloat, anotherFloat;
27 cout << "Please enter a floating-point number";
28 cout << " and then press Enter: ";
29 cin >> aFloat;
30
31 cout << "Please type another floating-point number";
32 cout << " and then press Enter: ";
33 cin >> anotherFloat;
34
35 cout << aFloat << " + " << anotherFloat << " = ";
36 cout << aFloat + anotherFloat << endl;
37 cout << aFloat << " - " << anotherFloat << " = ";
38 cout << aFloat - anotherFloat << endl;
39 cout << aFloat << " * " << anotherFloat << " = ";
40 cout << aFloat * anotherFloat << endl;
41 cout << aFloat << " / " << anotherFloat << " = ";
42 cout << aFloat / anotherFloat << endl;
43
44 system("PAUSE");
45 return 0;
46 }

```

The output from the program in [Listing 2.4](#) is shown in [Figure 2.5](#).

## Figure 2.5. The output from Prog02\_04.

[\[View full size image\]](#)

```

Please type an integer and then press Enter: 2
Please type another integer and then press Enter: 3
2 + 3 = 5
2 - 3 = -1
2 * 3 = 6
2 / 3 = 0
Please enter a floating-point number and then press Enter: 2.3
Please type another floating-point number and then press Enter: 4.5
2.3 + 4.5 = 6.8
2.3 - 4.5 = -2.2
2.3 * 4.5 = 10.35
2.3 / 4.5 = 0.511111
Press any key to continue . . . _

```



The program in [Listing 2.4](#) begins by declaring two integer variables on line 8. Next, it prompts the user to type in an integer and reads in the response. The program then asks for another integer and reads it into the program on lines 14-15.

## Note

You can declare more than one variable on a line of code by separating the variables with commas. All of the variables declared on one line of code are the same type.

On line 17, the program prints the first integer to the screen. As in the previous example programs, it sends output to the screen using the `cout` stream and the insertion operator. Notice that you can use the insertion operator more than once on a line of code. On line 17, the insertion operator appears four times. The result is that line 17 first sends the value in the variable `anInt` to the screen. It then sends a string containing a space, a plus sign, and another space, in that order. Next, the program prints the second integer the user typed in, followed by a space, an equal sign, and another space. Line 18 prints the result of adding the two integers. Finally, it prints an newline.

The result of lines 17-18 is shown in [Figure 2.5](#). When prompted, I typed in the numbers 2 and 3 for the two integers. Lines 17-18 used them to print the following output:

```
2 + 3 = 5
```

Notice that even though it took two lines of code to print the output, the result contains only one line of output. That is because the program prints all output on the same line until it encounters the newline symbol.

Lines 19-20 print the result of subtracting the two integers. Notice from the output shown in [Figure 2.5](#) that the result in this case is negative, which is what you'd expect. Lines 21-22 print the result of multiplying the two integers. The result in this case is also what you'd expect: the number 6. Lines 23-24 print the result of dividing the two integers. This result is not what most people expect. The output in [Figure 2.5](#) shows that the result of dividing 2 by 3 is 0. How did that happen?

When you use integers in programs, they cannot have any decimal points. Put another way, integers have no fractional parts. They are whole numbers only.

When you divide 2 by 3, the real answer is the fraction 2/3. Using a decimal point, that comes out to 0.6666666 (the 6 repeats infinitely). However, the program is using integers here because the variables `anInt` and `anotherInt` are declared as type `int` on line 8. As a result, the answer must also be an integer. Although the real answer is 0.6666666, the program forces the answer to be an integer by throwing away the decimal point and everything to the right of it. Only the 0 is left.

Starting on line 27 of [Listing 2.4](#), the program uses floating-point numbers. Floating-point numbers have fractional parts, so they can use decimal points. The program demonstrates adding, subtracting, multiplying, and dividing floating-point numbers. Notice that the division has no problem with the fact that the answer has a fractional part in it.

## Tip

If the answer to your calculations is going to be a number with a fractional part, be sure to use floating-point numbers (type `float` or `double`) rather than integers.

## Increment and Decrement Operators

In addition to the four standard math operators, C++ provides operators to add or subtract one to the value in a variable. Adding one to a variable is called *incrementing* it. Subtracting one is called *decrementing* it. Incrementing and decrementing are tasks that your programs need to perform a lot, so it's important to learn them.

The increment operator is two plus signs with no space between them (`++`). Likewise, the decrement operator is a pair of minus signs (`--`). [Listing 2.5](#) shows how to use them.

### Listing 2.5. Using the increment and decrement operators

```
1 #include <iostream>
2 #include <stdlib.h>
3
```

```

4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      // Declare an integer variable.
9      int anInt;
10
11     // Prompt the user for an integer.
12     cout << "Please input an integer and press Enter: ";
13     cin >> anInt;
14
15     // Increment the integer and show the result.
16     anInt++;
17     cout << "After increment: " << anInt << endl;
18
19     // Decrement the integer and show the result.
20     anInt--;
21     cout << "After decrement 1: " << anInt << endl;
22
23     // Decrement the integer and show the result.
24     anInt--;
25     cout << "After decrement 2: " << anInt << endl;
26
27     system("PAUSE");
28     return 0;
29 }

```

You can see the output from the program in [Listing 2.5](#) in [Figure 2.6](#)

## Figure 2.6. The results of using the increment and decrement operators.

```

Please input an integer and press Enter: 0
After increment: 1
After decrement 1: 0
After decrement 2: -1
Press any key to continue . . .

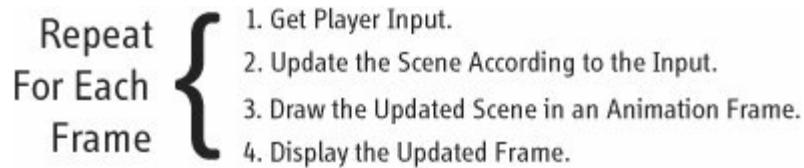
```

The program in [Listing 2.5](#) starts by declaring an integer variable. It prompts the user to input an integer. Whatever value the user enters, the program increments it on line 16. It outputs the result on line 17. On line 20, it decrements the value in the variable `anInt`, outputs the result, decrements it again, and outputs the result one final time. Line 27 causes the program to pause until the user presses a key on the keyboard. Line 28 returns the value 0 to the operating system to indicate that the program completed successfully.

# Loops

Most of what happens in computer programs happens repeatedly. For instance, I mentioned in [chapter 1](#) that computer animation happens one frame at a time. During each frame, the game performs a certain set of steps. Specifically, the game gets user input, updates the scene according to the input, draws the updated scene into the frame buffer, which is the back buffer we discussed in [chapter 1](#). It then displays the contents of the frame buffer by swapping the front and back buffers. [Figure 2.7](#) illustrates this process.

## **Figure 2.7. During each frame of animation, games perform these steps.**



Programs perform repetitive tasks like the one shown in [Figure 2.7](#) by using loops. There are several types of loops you can use. For now, I'll introduce the `while` and `do-while` loops.

# While Loops

It's easier to demonstrate `while` loops than to explain them. They're really simple. [listing 2.6](#) presents the code for a simple program that uses a `while` loop.

## Listing 2.6. Using a while loop

```
1  #include <iostream>
2  #include <stdlib.h>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      int anInt;
9
10     // Prompt the user for an integer.
11     cout << "Please input an integer and press Enter: ";
12     cin >> anInt;
13
14     int i=0;
15     while (i < anInt)
16     {
17         cout << i++ << endl;
18     }
19
20     system("PAUSE");
21     return 0;
22 }
```

This short program illustrates how easy using `while` loops is. After declaring an integer variable on line 8, the program prompts the user and gets a value for the variable. Next, it declares another integer called `i`. For reasons that had to do with the structure of early programming languages (specifically, early versions of a language called FORTRAN), it's traditional to name your loop counters `i`, `j`, or `k`. I follow that style of coding, so you'll see me use variables called `i`, `j`, and `k` whenever I use loops. Normally, we want our variables to have more descriptive names than `i`, `j`, and `k`. However, with loop counters, we can use these names because everyone knows what they are. If programmers see variables named `i`, `j`, or `k`, they expect those variables to be loop counters.

### Note

You can always initialize variables at the time you declare them. In fact, it's preferable to do so for most variables.

Line 14 of the program declares a loop counter called `i` and sets it to 0. On line 15, the program compares the value in `i` to the value in `anInt`. The comparison is in the parentheses after the C++ keyword `while`.

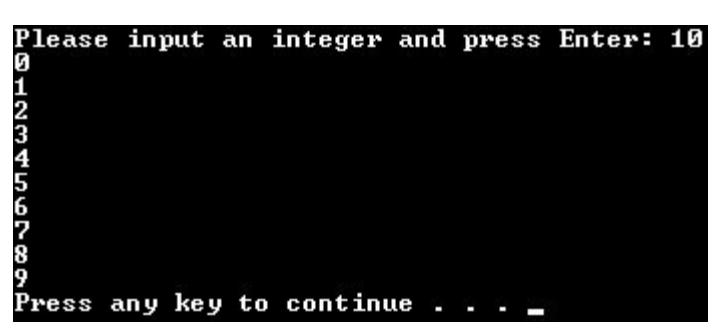
Comparisons, such as the one at the beginning of the `while` loop on line 15, can evaluate to `true` or `false`. Those are actual values in C++, just like numbers are. The condition on line 15 evaluates to `TRue` if the value in `i` is less than the value in `anInt`. If the condition evaluates to `true`, the body of the loop executes. The body of the loop is contained in the opening and closing braces on lines 16 and 18.

The body of the `while` loop in [Listing 2.5](#) contains just one statement. This statement does quite a bit. Let's look at it part by part.

First, the statement on line 17 says the program is sending a value to the stream `cout`. The value it's sending is the value in the variable `i`. Next, the statement increments the value. Finally, it sends an newline out to the `cout` stream. That's a lot to do in one statement, but this kind of programming is common in C++.

You've probably already figured out what the output of this program looks like. Nevertheless, I've provided it for your reference in [Figure 2.8](#)

### Figure 2.8. The output from the program in [Listing 2.6](#).



```
Please input an integer and press Enter: 10
0
1
2
3
4
5
6
7
8
9
Press any key to continue . . . _
```

In [Figure 2.8](#), I entered the number 10 when the program prompted me for input. The loop caused the program to print out the values 09 with one value printed per line.

After executing the statement in the loop's body, the program performs the loop's test again. If it evaluates to `TRue`, the program executes the loop's body

again. This continues until the condition evaluates to `false`. The condition evaluates to `false` if `i` is greater than or equal to the value in `anInt`. As soon as the condition evaluates to `false`, the loop stops.

It's important to note that I was careful about the number I typed in for the output in [Figure 2.8](#). I specifically entered a number that would make the loop execute. The `while` loop executes only for as long as its condition evaluates to `TRue`. If I had typed in 0 or a negative number, the value in `i` would not have been less than `anInt`. The condition on the `while` loop would have evaluated to `false` the first time it was tested. As a result, the loop would never have executed. [Figure 2.9](#) demonstrates this.

## Figure 2.9. The `while` loop won't execute if its condition is never true.

```
Please input an integer and press Enter: 0
Press any key to continue . . . _
```

Because I typed in 0 when I ran the program this time, the `while` loop's condition evaluated to `false` right away. As a result, the loop's body was never executed.

### Note

The statement on line 17 of [Listing 2.5](#) uses a particular type of increment operator called a *postincrement*. A postincrement operator performs its increment after the value in the variable has been used in the statement. So, for example, if the statement `a=b++;` appears in a program, it tells the program to assign the value in `b` to the variable `a` and then perform the increment. Because the increment is performed after the value has been used, it is called a postincrement.

# Do-while loops

The `do-while` loop works just like the `while` loop almost. [Listing 2.7](#) shows the difference.

## Listing 2.7. Using a do-while loop

```
1  #include <iostream>
2  #include <stdlib.h>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      int anInt;
9
10     // Prompt the user for an integer.
11     cout << "Please input an integer and press Enter: ";
12
13     // Get the integer.
14     cin >> anInt;
15
16     int i=0;
17     do
18     {
19         cout << i++ << endl;
20     } while (i < anInt);
21
22     system("PAUSE");
23     return 0;
24 }
```

The `do-while` loop has its test at the end of the loop instead of the beginning. As a result, the program executes the body of the loop first, then performs the test to determine whether it should loop again. [Figure 2.10](#) shows the output of the program in [Listing 2.7](#).

**Figure 2.10. The output of a `do-while` loop.**



```
Please input an integer and press Enter: 10
0
1
2
3
4
5
6
7
8
9
Press any key to continue . . . _
```

[Figure 2.10](#) looks the same as [Figure 2.8](#). So what's the difference between a `while` and a `do-while` loop? To see the answer, look at [Figure 2.11](#).

**Figure 2.11. The `do-while` loop executes at least once.**

```
Please input an integer and press Enter: 0
0
Press any key to continue . . . _
```

If you compare [Figure 2.11](#) to [Figure 2.9](#), you'll see that if the condition of a `while` loop never evaluates to `TRue`, the program never executes the body of the `while` loop. On the other hand, the body of a `do-while` loop is guaranteed to execute at least once. Because a `do-while` loop performs its test at the end of the loop instead of the beginning, the body *must* execute at least once before the condition is ever tested.

Both the `while` and the `do-while` loops see a lot of use in all types of programs. There is one more type of loop in C++: the `for` loop. It's a more complex looping statement, so we'll put off learning about it until the next chapter.

## Factoid

Because the `while` loop performs its test at the beginning of the loop, it is called a [pretest loop](#). Likewise, because the `do-while` loop performs its test at the end, it is called a [posttest loop](#).



# Windows Programming

One of the biggest obstacles to game programming for new programmers is learning to write games that work on the Windows operating system. Windows programming is a huge and complex topic. But don't worry; you'll be able to sidestep most Windows programming issues. The LlamaWorks2D game engine enables you to write games for Windows without having to deal much with the operating system itself.

## WinMain()

If you write a Windows program without using the LlamaWorks2D game engine, the first task you'll need to tackle is writing a `WinMain()` function. In a Windows program, the `WinMain()` function replaces the normal `main()` function used in other types of C++ programs. Your `WinMain()` function must define the type of window that the program uses by creating what is called a window class. Also, it must declare the window class, which is conceptually similar to declaring a variable in a function. Next, `WinMain()` has to display the window. When the window is on the screen, `WinMain()` dispatches messages to its message-handling function (more on this in a moment).

Getting all of the `WinMain()` function's tasks done usually takes a couple of hundred lines of code. Yes, I did say *hundred*. Every Windows program must go through this sequence of tasks in order to operate. The difference in the code from program to program is really minimal. Most people just write `WinMain()` once and reuse it in all their different Windows programs with minor variations. However, writing `WinMain()` isn't easy the first time you do it.

To save you a considerable amount of work, the LlamaWorks2D game engine provides a `WinMain()` function for you, so you don't have to write it. To set up your Windows program the way you want, you must pass LlamaWorks2D some initialization information that `WinMain()` uses. I'll show you how to do that in the next chapter. Other than passing in the initialization information, you don't need to do anything to get your game's window set up.

### Note

If you want to see what `WinMain()` looks like, you can look in the source code for the game engine on the CD. You'll find the `WinMain()` function in a file called `LW2DApp.cpp`, which is in the `\Tools\LlamaWorks2D` folder.

# Messages and Message Processing

Windows is based on the idea of passing messages to a program's windows. In fact, *everything* in Windows is a window. That includes buttons, menus, menu items, sliders, and, of course, windows. Nearly everything in Windows can send or receive messages. As a result, most of Windows programming is learning how to respond to messages. I'll demonstrate techniques for doing this throughout the rest of this book.

Windows messages are generated by events. Most events are a result of the program's interaction with a person. For example, if the program's user clicks an onscreen button or selects a menu item, Windows triggers events. The events generate messages, and your program responds to the messages in a special message-handling function.

LlamaWords2D simplifies the process of handling messages in Windows somewhat. As much as possible, the game engine provides the help and tools you'll need to handle messages. As a result, you'll find that the techniques I demonstrate are remarkably simple when compared with regular Windows programming.

# Game Programming

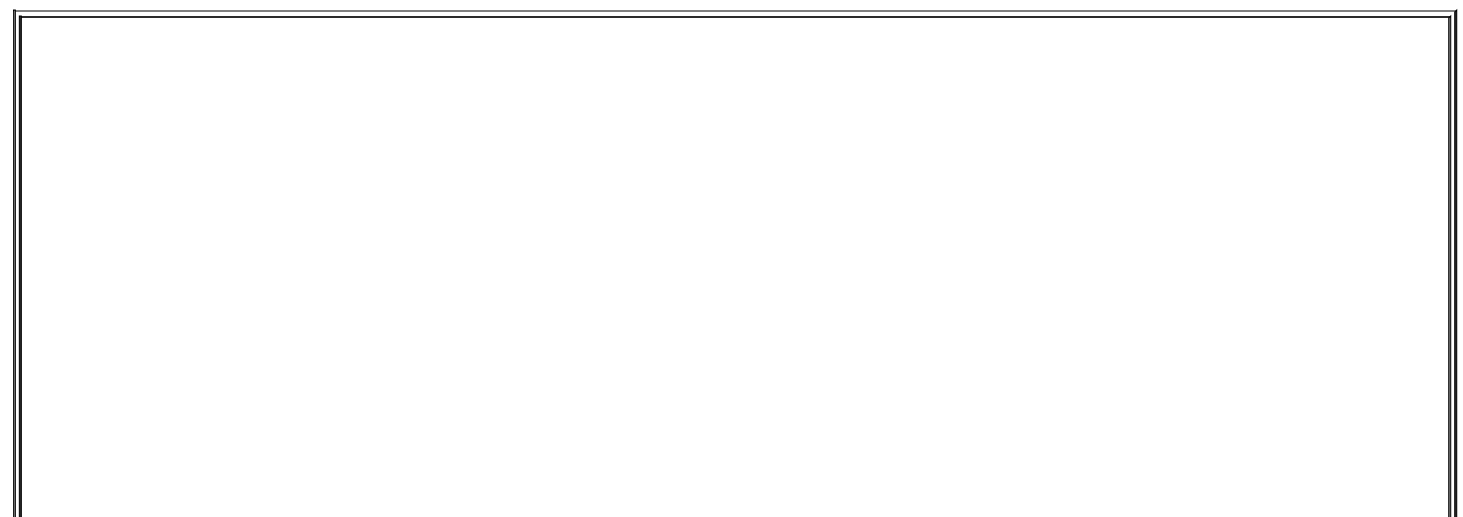
Game programming means graphics and sound programming. As mentioned in [chapter 1](#), this book uses the Screaming Llama LlamaWorks2D game engine and the OpenGL graphics library. For sound, it uses OpenAL. OpenGL and OpenAL are industry standard tools that are well documented and widely available.

## What is OpenGL?

The Open Graphics Library (OpenGL) was first invented at Silicon Graphics, Inc. many years ago. OpenGL is an open standard. That means it's a standard library worked out by an industry group. Everyone who creates a version of OpenGL conforms to the standard, so all versions of OpenGL on all types of computers work essentially the same way. The standard allows some differences, but you don't have to worry about them if you don't want to use the features they define.

OpenGL provides you and me with code to do the most important graphics tasks. For example, it enables our programs to easily draw bitmaps and geometric shapes (circles, rectangles, and so forth) into a game's back buffer. When the final frame is ready, our programs can easily switch the front and back buffers to display the frame on the screen.

Programming with OpenGL is not difficult. After a bit of introduction, you'll be using it like a pro. Unfortunately, one of the hardest tasks to accomplish when you're using OpenGL is to get it up and running under Windows. However, the LlamaWords2D game engine does that for you. All you have to do is pass the engine some startup information that I'll explain in [chapter 3](#). The engine takes it from there.



## Where Can I Get More Information on OpenGL and OpenAL?

Information on both OpenGL and OpenAL is available from many sources on the Web. Probably your best starting point for OpenGL information is [www.opengl.org](http://www.opengl.org). This is the home page for the organization that maintains OpenGL. Here you'll find documentation, articles, and discussion forums where you can ask industry experts your OpenGL questions.

If you want documentation on how to use OpenGL with Windows, you can get it at [www.msdn.microsoft.com/library/en-us/dnanchor/html/opengl.asp?frame=true.if](http://www.msdn.microsoft.com/library/en-us/dnanchor/html/opengl.asp?frame=true.if) that address is too long for you to type (it is for me too), you can find this documentation set by going to [www.msdn.microsoft.com/library/default.asp](http://www.msdn.microsoft.com/library/default.asp) and typing `OpenGL` in the Search box.

For information on OpenAL, I suggest you start with [www.openal.org](http://www.openal.org). This site contains documentation, example programs, and links to articles and other Web pages about OpenAL.

## What is OpenAL?

Like OpenGL, the Open Audio Library (OpenAL) is an open standard. OpenAL enables our games to play and mix sounds. Mixing sounds is important for proper special effects. If you have two characters firing guns, you want to hear both bangs. To make that happen, your game has to mix the two sounds together in a memory buffer and then play the contents of the buffer.

With OpenAL, you can play music and special effects at the same time. When you get proficient with it, you can use OpenAL to do 3D sounds. For instance, if there's an explosion off to the player's left, you can get OpenAL to make the sound come from the left speaker. You can also do effects such as diminishing sounds over distance, echoing, and so on.

## OpenGL, OpenAL, and DirectX

As mentioned in [chapter 1](#), Microsoft provides a library that does graphics, sound, and other game-related tasks. It's called DirectX. The DirectX graphics library is called DirectX Graphics or Direct3D (Direct3D is by far the more common name). Its sound libraries are called DirectSound and DirectMusic.

Programmers new to game programming often ask, "Which is better, OpenGL and OpenAL or DirectX?"

If you're a beginner, "better" probably means "easier." In that case, OpenGL and OpenAL are by far the better choice. That's why we're using them in this

book. DirectX is complex. There's nothing easy about it.

If you're a proficient C++ programmer with a background in graphics, DirectX is the better choice. It gives you cutting-edge graphics and sound. OpenGL and OpenAL are professional tools used in real games. However, they do not give the screaming performance and bleeding-edge graphics and sound that DirectX provides.

Many great games do not need cutting-edge graphics and sound. Professional games such as Blizzard Entertainment's WarCraft have long used 2D graphics. WarCraft, and games like it, do not depend on blindingly fast graphics and split-second player response times. You can write games just like it with OpenGL and OpenAL.

The long and short of all this is that OpenGL and OpenAL are a better place to start. If you then want to move to top-level graphics and sound, you'll have to learn DirectX. To be a professional game programmer generally requires a strong knowledge of both OpenGL/OpenAL and DirectX.

# Summary

The first step in learning to write games is learning to program in C++. This chapter provided a brief introduction to writing programs in C++ using the free Dev-C++ compiler provided on the CD that comes with this book. For each program you write, you must create a project file and add source files to it. Your source files contain functions. One of the source files must contain the `main()` function, which is where every C++ program starts and ends. When you write your program, you must also compile and link it before you can run it.

Functions use the various tools that C++ provides. These tools include data types, variables, and streams. Functions such as `main()` pass information to other functions and get information back from them using parameter lists and return values. Functions often use the basic C++ math operators to do their work. The most common are addition, subtraction, multiplication, division, increment, and decrement. Functions use loops, such as the `while` and the `do-while` loops, to perform repetitive tasks.

Windows programs use a `WinMain()` function rather than a `main()` function. They process messages that are generated in response to events. Events are caused by such things as player input. The LlamaWords2D game engine provides a `WinMain()` function and some message-handling tools for you so that you can more easily overcome the challenge of learning Windows programming.

We've covered a lot of information in one chapter. However, the concepts presented in this chapter are fundamental to game programming in C++. You will use them in every program you write. So if it still all seems a bit overwhelming at this point, don't be concerned. Because you'll use these concepts in every program, you'll get lots of practice with them. I'll walk you through the process of applying these ideas to your games, beginning in the next chapter.



# Part 2: ObjectOriented Programming in Games

[Chapter 3. Introducing Object-Oriented Programming](#)

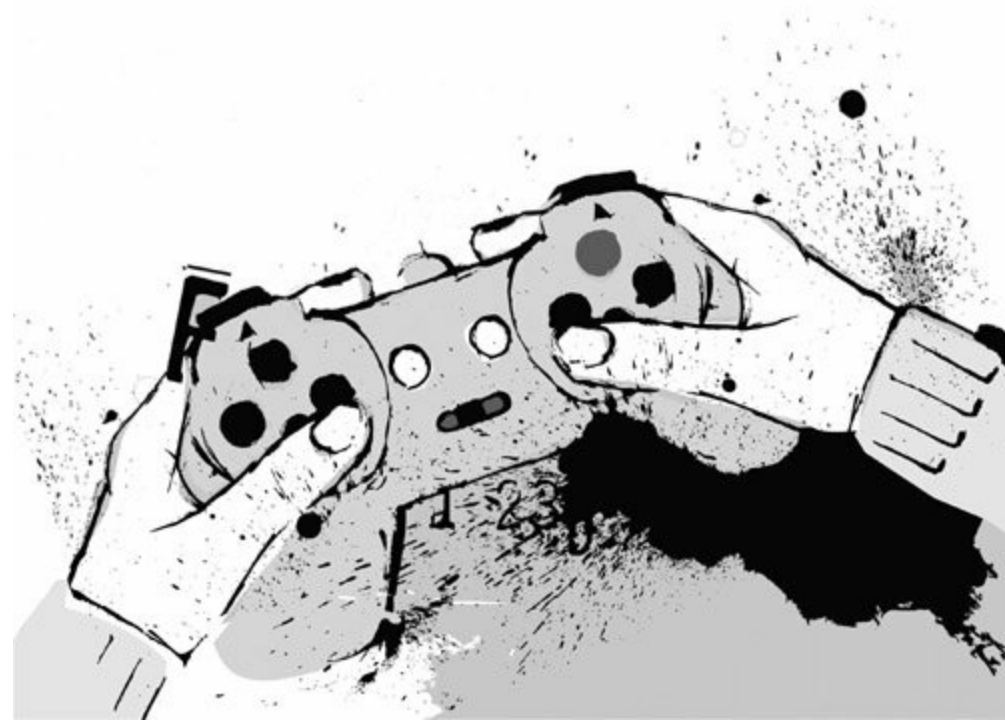
[Chapter 4. Introducing the LlamaWorks2D Game Engine](#)

[Chapter 5. Function and Operator Overloading](#)

[Chapter 6. Inheritance: Getting a Lot for a Little](#)

# Chapter 3. Introducing Object-Oriented Programming

Before we start writing games, we should talk a bit more about C++. Specifically, we need to discuss how to create your own data types, how to write conditions, and how to use `if-else` statements to make decisions. In addition, I'll explain how to keep the type names you create from conflicting with other type names in your game.



# Software Objects

[chapter 2](#) showed that C++ programs use variables to store information. Every variable has a data type. As you may recall, the purpose of the data type is to specify what kind of information the variable holds. For instance, integer variables hold integers (numbers with no fractional parts), floating-point numbers contain real numbers (numbers with fractional parts), strings hold groups of characters, and so on.

## Factoid

One of the most common game programming tasks is creating software objects to represent things that players see on the screen.

The C++ language would be extremely limited if it could only use the small group of data types it defines. We would have a tough time writing our programs that way. As game programmers, we usually need to create data types that represent things the player sees on the screen.

For example, when your game shows a dragon on the screen your code should be able to define a variable of type `dragon`. The `dragon` variable would store all the information needed to represent a dragon in the program. This information might include the dragon's current position, how far it can spit fire, and how many fireberries it's eaten (more fireberries means it spits fire farther). When the dragon moves, the `dragon` variable must keep track of which way it's moving and how fast.

To define our own types to represent things we need in games, we use a technique called [object-oriented programming](#). In object-oriented programming, you define your types to represent anything you want. When you declare a variable of that type, you are said to be creating a [software object](#). The objects you create can be anything in the real world or anything you can imagine. If you want to define an object to represent dragons, you can. If you need an object to represent walls, cars, feet, or trees, you can define those too. Anything can become objects in software you're limited only by your own imagination.

Software objects contain data. The data describes the thing the object represents. In our dragon example, a `dragon` variable stores such information as

the dragon's current position, how far it can spit fire, and how many fireberries it's eaten. That information describes a dragon. This is the way software objects work. We use them so that we can create variables that store all the information needed to describe an object.

## Note

Because an object's data describes whatever the object represents, programmers often say that the set of data the object contains is its *[attributes](#)*.

All software objects behave exactly the way you tell them to. You define the set of actions, also called operations, that the software can perform on your objects. For example, consider the case of integers. What operations can we perform on integers?

Because we all know basic math, you and I can easily state that the most fundamental operations we can perform on integers are addition, subtraction, multiplication, and division. In C++, these four operations are built into the `int` data type. We saw how to use them in [chapter 2](#).

All of the data types in C++ have a specific group of operations that programs can perform on them. If you make your own software objects, you're making your own data types. When you do, you must also define the set of operations that programs can perform on them. All of this is accomplished with C++ classes, so that's what we'll discuss next.

# Classes

To define objects in C++, you create classes. Defining your own classes is actually easier than explaining it. So rather than starting with a long explanation, let's go through the sample program in [Listing 3.1](#).

## Listing 3.1. Defining a software object with a C++ class

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  class my_object
7  {
8  public:
9      int i;
10     float f;
11 };
12
13 int main(int argc, char *argv[])
14 {
15     my_object anObject;
16
17     anObject.i = 10;
18     anObject.f = 3.14159;
19
20     cout << "i = " << anObject.i << endl;
21     cout << "f = " << anObject.f << endl;
22     system("PAUSE");
23     return (EXIT_SUCCESS);
24 }
```

The short program in [Listing 3.1](#) demonstrates how to define a class in C++. It uses the C++ keyword `class` on line 6 to define a new type: `my_object`. Everything in the class definition must be inside the opening and closing braces. The definition of the contents of the `my_object` type spans lines 7-11.

This program uses the `my_object` type to declare a variable in the `main()` function on line 15. As you can see, declaring a variable of type `my_object` is done in just the same way as declaring integers or floating-point numbers. Just as with any other variable type, variables of type `my_object` can be accessed in the functions where they are declared. Another way to say that is that the rules of scope work the same way for both objects and built-in types such as `int`.

# Member Data

The `my_object` type contains an integer called `i` and a floating-point number called `f`. Programs using the `my_object` type store data in `i` and `f`. Therefore, `i` and `f` are collectively referred to as the class's *member data*. Confusingly, when we talk about just `i` or `f` by themselves, we call them data members. In other words, one item of member data is called a data member. And I'll bet you thought programmers had no sense of humor.

A class's member data stores all of the information that describes the object the class represents. As mentioned earlier, that information is also called the class's attributes.

The member data in the `my_object` class in [Listing 3.1](#) is defined as being *public*. This means that any function can access the member data directly. For example, on line 17, the `main()` function stores an integer in `i`. It does so by stating the variable name (`anObject`), followed by a period, and then the data member name (`i`).

After storing values in the member data of the variable `anObject`, the program in [Listing 3.1](#) prints the values it stored. When it does, line 20 shows that it uses `anObject.i` just as if it was a regular `int` variable. Specifically, it uses the insertion operator to put the value in `anObject.i` into the stream `cout`. On line 21, it does the same with `anObject.f`. [Figure 3.1](#) shows the output.

## Figure 3.1. The output of the program in [Listing 3.1](#).

```
i = 10
f = 3.14159
Press any key to continue . . .
```

As you can see, the program printed the values in `anObject.i` and `anObject.f` in just the way it prints integer or floating-point variables.

### Note

In general, programs can use member data in the same way they use variables of the same type.

You can declare more than one object of the same type in a function. [Listing 3.2](#) shows how.

## Listing 3.2. Declaring two objects of the same type in a function

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  class my_object
7  {
8  public:
9      int i;
10     float f;
11 };
12
13 int main(int argc, char *argv[])
14 {
15     my_object anObject, anotherObject;
16
17     anObject.i = 10;
18     anObject.f = 3.14159;
19
20     anotherObject.i = -10;
21     anotherObject.f = 0.123456;
22
23     cout << "anObject.i = " << anObject.i << endl;
24     cout << "anObject.f = " << anObject.f << endl;
25
26     cout << "anotherObject.i = " << anotherObject.i << endl;
27     cout << "anotherObject.f = " << anotherObject.f << endl;
28
29     system("PAUSE");
30     return (EXIT_SUCCESS);
31 }
32
```

As the program in [Listing 3.2](#) illustrates, you can declare as many objects of the same type as you need. Each object has its own member data, in this case `i` and `f`. The value in `anObject.i` has nothing to do with the value in `anotherObject.i`; they are completely independent of each other. The same is true for `anObject.f` and `anotherObject.f`. You can see this from the output of the program, which appears in [Figure 3.2](#).

## Figure 3.2. The values in the two objects are independent of each other.

```
anObject.i = 10
anObject.f = 3.14159
anotherObject.i = -10
anotherObject.f = 0.123456
Press any key to continue . . .
```

In [Figure 3.2](#), the program stores and prints two different sets of values. This shows that the two objects contain different information; they are independent of each other.

## Member Functions

As mentioned earlier, programmers define more than just the data for the objects they create. When a game programmer creates a type, she also writes the set of operations that can be performed on the type. Those operations are functions, which we introduced in [chapter 2](#). The functions that form the set of valid operations that can be performed on a type are all members of the class. Just as we can define member data in classes, we can also define member functions. Member functions form the set of valid operations a program can perform on an object. Again, it's easier to show what I mean rather than launching into a long explanation. So let's examine the program in [Listing 3.3](#).

### Listing 3.3. Creating and using member functions

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  class my_object
7  {
8  public:
9      void SetI(int iValue);
10     int GetI();
11
12     void SetF(float fValue);
13     float GetF();
14
15 private:
16     int i;
```



```

17     float f;
18 };
19
20
21 void my_object::SetI(int iValue)
22 {
23     i = iValue;
24 }
25
26
27 int my_object::GetI()
28 {
29     return (i);
30 }
31
32
33 void my_object::SetF(float fValue)
34 {
35     f = fValue;
36 }
37
38
39 float my_object::GetF()
40 {
41     return (f);
42 }
43
44
45 int main(int argc, char *argv[])
46 {
47     my_object anObject, anotherObject;
48
49     anObject.SetI(10);
50     anObject.SetF(3.14159);
51
52     anotherObject.SetI(-10);
53     anotherObject.SetF(0.123456);
54
55     cout << "anObject.i = " << anObject.GetI() << endl;
56     cout << "anObject.f = " << anObject.GetF() << endl;
57
58     cout << "anotherObject.i = " << anotherObject.GetI() << endl;
59     cout << "anotherObject.f = " << anotherObject.GetF() << endl;
60
61     system("PAUSE");
62     return (EXIT_SUCCESS);
63 }

```

The program in [Listing 3.3](#) is a modification of the program in [Listing 3.2](#). If you compare the two programs, you'll see some very important changes.

First, the program in [Listing 3.3](#) changed the member data from public to private. You'll see this on lines 15-17. Recall that any function in a program can access public member data. The same is true for public member functions. However, private members are different. The private member data on lines 16

and 17 can only be accessed by member functions. It's as if the class is a very exclusive club; only members can use the club's private goodies.

The list of member functions is shown on lines 913. Notice that all of these functions are public. This is typically how programmers define classes. You should only rarely make member data publicly accessible. Normally, the member functions should be public and the member data private.

The reason programmers make class data private and functions public is simple: control. If a class's member data can only be accessed by the member functions, then we're controlling how the data can be changed. If the member functions never let the data get into an invalid state, then the data will always be valid.

Remember our dragon example we used when we first started talking about classes? Recall that the dragon in the game could eat fireberries to spit fire farther. Suppose the number of fireberries the dragon has eaten is stored in the `dragon` class as a data member. What would happen in the game if that number somehow got to be negative? Oops. Now the dragon swallows fire rather than spits it. The game will probably crash. That's why it's vital to keep all data valid at all times.

## Tip

Except in very rare circumstances, class data should be kept private. Usually, the member functions are public. Only the member functions can access the private member data. Using this rule when defining classes goes a long way toward keeping the member data in a valid state at all times.

When we define objects, controlling the access to member data helps keep the data in a valid state. No functions outside the class can change private data. As long as the member functions never let the data become invalid, the data is always in a valid state. This is such an important point that I'm going to emphasize it with a special Tip.

If you look back at [Listing 3.3](#), you'll see the prototypes for the member functions on lines 913. Putting the prototypes in the class tells the compiler which functions are members. The member functions for this class are very simple: All they do is set or get the values of the member data. We'll see in

later chapters that member functions can also check the values before they set the member data.

The actual functions are given on lines 2142. Let's focus on the first line of the `SetI()` function, which appears on line 21. Notice that the first line of the function includes the name of the class followed by two colons. Two colons used together like that known as the C++ scope resolution operator is a way to tell the compiler, "This is the function `SetI()` that I said was a member in the `my_object` class definition." The compiler replies, "Aha. So every time I see a call to the `my_object` class's `SetI()` function, this is the function I'll use." All member functions must be written in this way.

The `SetI()` function sets the value of `i`. You might be thinking, "Duh. I could tell that from the name of the function." That's exactly why the function is named `SetI()`. When you write your member functions, name them in a way that anyone looking at your code can tell what the function does by its name. If you try to explain what the function does, "Duh" is a good response to get it means you named your function well.

## Note

It is possible to write member functions a bit differently: as inline member functions. We discuss inline member functions later in this chapter. You'll see that even inline member functions can be written the way the functions in [Listing 3.3](#) are written. In fact, it is actually preferable to do so. Therefore, I recommend that you use the style shown here.

As I mentioned, the `SetI()` function sets the value of `i` to the value in the `Value` parameter. In most programs, the `SetI()` function would check the value of `Value` before saving it into `i`. I'll show how to do that sort of data validation when we discuss `if-else` statements in "The `if-else` Statement" section later in this chapter.

Now take a look at the `main()` function in [Listing 3.3](#). On lines 4950, you'll see calls to the `SetI()` and `SetF()` functions. Notice that the program calls member functions by stating the name of the object, followed by a period, and then the name of the function. There is an important reason for doing it this way: If you look on lines 5253, you'll see two more calls to the `SetI()` and `SetF()` functions. The difference between these two pairs of function calls is that the first pair sets the member data in the `anObject` variable, while the second pair sets the

member data of the `anotherObject` variable. So any time you call a member function, you call that function on an object. The function uses the member data of that particular object and no other. This enables you to specifically state which object you're changing.

## Tip

Make every function name describe what the function does.

Lines 5559 demonstrate calls to the `GetI()` and `GetF()` functions. Again, they follow the same pattern as calls to the `SetI()` and `SetF()` functions. Specifically, they state the variable name, then a period, and then the function name. Your programs always call member functions in this way.

## Constructors and Destructors

Classes can have special member functions called [\*constructors\*](#) and [\*destructors\*](#). Constructors are called automatically when the object is created. Constructors are used to initialize the object's member data into known states before any other function can possibly use the object. Destructors are called automatically when the object is destroyed. They perform any cleanup tasks on the object that may be required. [Listing 3.4](#) demonstrates the use of constructors and destructors.

### Listing 3.4. Constructors and destructors in classes

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  class my_object
7  {
8  public:
9      my_object();
10     ~my_object();
11
12     void SetI(int iValue);
13     int GetI();
14
```

```
15 void SetF(float fValue);
16 float GetF();
17
18 private:
19     int i;
20     float f;
21 };
22
23
24 my_object::my_object()
25 {
26     cout << "Entering constructor." << endl;
27     i = 0;
28     f = 0.0;
29 }
30
31
32 my_object::~~my_object()
33 {
34     cout << "Entering destructor." << endl;
35 }
36
37
38 void my_object::SetI(int iValue)
39 {
40     i = iValue;
41 }
42
43
44 int my_object::GetI()
45 {
46     return (i);
47 }
48
49
50 void my_object::SetF(float fValue)
51 {
52     f = fValue;
53 }
54
55
56 float my_object::GetF()
57 {
58     return (f);
59 }
60
61
62 int main(int argc, char *argv[])
63 {
64     my_object anObject, anotherObject;
65
66     anObject.SetI(10);
67     anObject.SetF(3.14159);
68
69     anotherObject.SetI(-10);
70     anotherObject.SetF(0.123456);
71
72     cout << "anObject.i = " << anObject.GetI() << endl;
73     cout << "anObject.f = " << anObject.GetF() << endl;
74
75     cout << "anotherObject.i = " << anotherObject.GetI() << endl;
```

```
76     cout << "anotherObject.f = " << anotherObject.GetF() << endl;
77
78     system("PAUSE");
79     return (EXIT_SUCCESS);
80 }
```

Looking at lines 910 of [Listing 3.4](#), you'll see that the `my_object` class now contains prototypes for two additional member functions: the constructor and the destructor, respectively. These prototypes are very different from any that we have introduced so far. Unlike other member functions, the names of constructors exactly match the name of the class. The only difference between the names of a constructor and a destructor is that the destructor name is preceded by the tilde (pronounced TILL-duh) character, which looks like this: "~".

## Note

Nearly every class you write needs a constructor. However, many classes do not need destructors. [chapter 11](#), "Pointers," explains exactly when to use destructors.

The code for the constructor is on lines 2429. Like all member functions, the constructor must state the class that it's a member of by using the class name and then the scope resolution operator. That's why line 24 has the name of the class, then the two colons followed by the name of the constructor (which is also the name of the class).

Notice that neither the constructor nor the destructor has a return type. The constructor doesn't need a return type because it creates an object of the class that it's a member of. In [Listing 3.4](#), the constructor creates an object of type `my_object`. In "programmer-speak," we say that the constructor's return type is implicit in its name.

Destructors, on the other hand, delete an object of their class type. There is no return type because there's nothing to return. The object is gone when the destructor ends.

When the constructor starts, it prints the message *Entering constructor* to the screen. Most constructors don't do this; I did it here just for demonstration purposes. The primary purpose of a constructor is to set the member data into

a known state when the program creates an object of that type. That's exactly what happens on lines 2728 of [Listing 3.4](#). The constructor sets the member data to 0.

## Tip

For every class you write, it is important to make sure that the class's constructor sets all member data to a known state. Often, that just means setting all data members to 0.

The destructor appears on lines 3235. This class has nothing to clean up, so the destructor doesn't do anything significant. For demonstration purposes, it prints the message *Entering destructor* when it starts.

A look at the `main()` function on lines 6479 shows that it is no different than the one in [Listing 3.3](#). That probably seems strange. The logical question to ask is, "Why aren't the constructor and destructor called?" In fact, they are. **The program output in [Figure 3.3](#)** proves it.

## Figure 3.3. Calling the constructor and destructor.

```
D:\>prog_03_04
Entering constructor.
Entering constructor.
anObject.i = 10
anObject.f = 3.14159
anotherObject.i = -10
anotherObject.f = 0.123456
Press any key to continue . . .
Entering destructor.
Entering destructor.
```

## Note

The last two lines of output will not be seen unless you are running `Prog_03_04.exe` from the Windows command line.

When I ran the program in [Listing 3.4](#), it began with the `main()` function, as all C++ programs do. The first thing `main()` did was to declare two variables of type `my_object`. At that time, the program automatically called the constructor once for each object that `main()` created. The first two lines of the output in [Figure 3.3](#) show that that's exactly what happened; the constructor was called twice. Each time the constructor executed, it printed the message *Entering constructor*.

On lines 6670, `main()` set the member data in both of its `my_object` variables. It then printed the values in the member data. We can see that in the output in [Figure 3.3](#) The program paused and waited for me to press a key. When I did (just in case you're curious, I pressed the letter Z), the program executed the `return` statement on line 79. This caused the `main()` function to end. When `main()` ended, the two variables it declared on line 64 were no longer being used. Programmers say that the variables went "out of scope." When variables go out of scope, C++ programs automatically calls the appropriate destructors once for each class variable. In this case, it called the `my_object` destructor twice, as you can see from the output in [Figure 3.3](#)

## Inline Member Functions

As you begin to read professional C++ programming literature, you'll see that many programmers put the code for their member functions into the class definitions. This approach is allowed in C++ programming. Functions defined this way are called *inline member functions*. [listing 3.5](#) demonstrates the use of inline member functions.

### Listing 3.5. A class with inline member functions

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  class point
7  {
8  public:
9      point()
10     {
11         x = y = 0;
12     }
13
14     void SetX(int xValue)
15     {
16         x = xValue;
17     }
```



```

18     int GetX(void)
19     {
20         return (x);
21     }
22
23     void SetY(int yValue)
24     {
25         y = yValue;
26     }
27
28     int GetY(void)
29     {
30         return (y);
31     }
32 }
33
34 private:
35     int x, y;
36 };
37
38 int main(int argc, char *argv[])
39 {
40     point rightHere;
41
42     rightHere.SetX(10);
43     rightHere.SetY(20);
44
45     cout << "(x,y)=" << rightHere.GetX();
46     cout << ", " << rightHere.GetY() << " ";
47     cout << endl;
48
49     rightHere.SetX(20);
50     rightHere.SetY(10);
51
52     cout << "(x,y)=" << rightHere.GetX();
53     cout << ", " << rightHere.GetY() << " ";
54     cout << endl;
55
56     system("PAUSE");
57     return (EXIT_SUCCESS);
58 }

```

Rather than just prototypes for member functions, the class in [Listing 3.5](#) contains the member functions themselves. All of the code for each function appears in the class. The `main()` function on lines 38-58 demonstrates that your programs can use inline member functions in exactly the same way they use member functions defined outside of a class.

## Note

Member functions whose code appears outside of the class definition are called [out-of-line member functions](#).

You're probably quite logically wondering what, if any, differences exist between inline and out-of-line member functions. There's really only one difference: When you define a member function inline, the compiler substitutes the code from the inline member function into your program. With out-of-line functions, that doesn't happen. For instance, if you look back at [Listing 3.5](#), you'll see the statement

```
rightHere.SetX(10);
```

on line 42. When this statement gets compiled, the compiler substitutes the code for the `SetX()` function right into the statement. That is, it puts the equivalent of

```
rightHere.x=10;
```

into the compiled program. Of course, it doesn't change the source code in the .cpp file. It performs the substitution in the object code that it generates. It does this everywhere in every function that calls the inline function. The program in [Listing 3.5](#) makes one call to the constructor, and two calls each to `SetX()` and `SetY()`. That means it substitutes code from the member functions into the `main()` function five times.

## Factoid

Dev-C++ puts its object code in .o files. Other compilers, such as Microsoft's Visual C++, put object code into .obj files. Whether the filename ends with .o or .obj doesn't matter. Either way, the file contains object code.

When I was teaching college-level C++ programming classes, students would often ask, "Why not just make all functions inline?"

The first reason is that the compiler won't allow it. If the compiler determines that the function is too long or too complex, it won't compile the member

function as an inline function. It still compiles the function without a problem. However, it converts the member function into an out-of-line function in the compiled code. And it does this without telling you. It is completely up to whoever writes the compiler whether or not a function can be made inline. You and I can't control it. Writing functions inline is more of a suggestion than a command.

Usually functions remain inline if all they do is set or get values from members in a class. They generally also remain inline if they perform simple calculations. However, member functions that contain loops or call other functions are not likely to remain inline.

## **Note**

In general, inline member functions should not do much more than get or set the values of member data.

Another reason why it might not be a good idea to make all member functions inline is that inline member functions can make programs very large. Because the C++ compiler performs code substitution with inline member functions, there are fewer function calls in programs so they run faster. However, it also makes them bigger because the code for the inline functions gets inserted repeatedly. Using inline functions means that there are lots of copies of the inline function in the compiled program. With out-of-line functions, the program jumps to the one and only copy of the member function. That's a tiny bit slower, but it makes the program smaller. So when you're writing programs, you have to decide which is more important: speed or size.

The last reason why you might not make all of your member functions inline is that it makes your classes hard to read. Complex classes with lots of member functions become huge when you use inline member functions. Other programmers tend to find them difficult to deal with. Plenty of programmers disagree with this point of view; you have to decide for yourself.

## **Tip**

In game programming, the real memory and disk hog tends to be the graphics, images, and sounds your game displays. Even if you use lots of inline member functions, your code is small by comparison. Although it's not good to try and make *every* member

function inline, you shouldn't worry a lot about program size. Speed tends to be the more important factor.

One way to have the advantages of inline member functions without cluttering up your class definitions is to use out-of-line inline functions.

Say what?

Amazingly enough, C++ lets you define out-of-line member functions that are inline. It sounds kooky, but it's actually a nice feature. [listing 3.6](#) illustrates how to create out-of-line inline functions.

### Listing 3.6. Making out-of-line functions inline

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  class point
7  {
8  public:
9      point();
10     void SetX(int xValue);
11     int GetX(void);
12     void SetY(int yValue);
13     int GetY(void);
14
15 private:
16     int x,y;
17 };
18
19
20 inline point::point()
21 {
22     x = y = 0;
23 }
24
25 inline void point::SetX(int xValue)
26 {
27     x = xValue;
28 }
29
30 inline int point::GetX(void)
31 {
32     return (x);
33 }
34
```

```

35 inline void point::SetY(int yValue)
36 {
37     y = yValue;
38 }
39
40 inline int point::GetY(void)
41 {
42     return (y);
43 }
44
45
46
47 int main(int argc, char *argv[])
48 {
49     point rightHere;
50
51     rightHere.SetX(10);
52     rightHere.SetY(20);
53
54     cout << "(x,y)=" << rightHere.GetX();
55     cout << "," << rightHere.GetY() << ")";
56     cout << endl;
57
58     rightHere.SetX(20);
59     rightHere.SetY(10);
60
61     cout << "(x,y)=" << rightHere.GetX();
62     cout << "," << rightHere.GetY() << ")";
63     cout << endl;
64
65     system("PAUSE");
66     return (EXIT_SUCCESS);
67 }

```

In [Listing 3.6](#), all of the functions from [Listing 3.5](#) are now moved out of line. Only the function prototypes remain in the class definition. However, if you look at the member functions, you'll see that each one begins with the C++ keyword `inline`. An example is the `SetX()` function on line 25. Putting the keyword `inline` at the beginning of the function definition makes `SetX()` an inline function even though the code appears out of line. Using this style gives you shorter, more readable class definitions and still provides the advantages of inline functions. When we examine the source code for the LlamaWorks2D game engine in the next chapter (the source code is also provided on the CD), you'll find that this is the style it uses for nearly all of its inline functions.

# Logical Operators

The sample programs presented so far generally execute one statement after another. They usually do exactly the same thing each time they run. Although this is a good way to learn C++ programming, that's not how games operate. Games need to present players with different experiences each time they play. The most important tool for doing this is the C++ `if-else` statement, which we will discuss shortly. However, to use the `if-else` statement, programmers have to understand logical operators. Therefore, we'll take a brief look at them now.

Logical operators are pretty straightforward. In fact, you have seen them before in your math classes in school. [Table 3.1](#) shows the C++ logical operators.

**Table 3.1. The C++ Logical Operators**

---

Operator	Description
----------	-------------

<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal
!	Not
>=	Greater than or equal to
>	Greater than
&&	And
	Or

---

Although you may have used different symbols for these in school, you're

probably familiar with what they do. You use these operators in comparisons, which programmers call *logical expressions*. All logical expressions evaluate to `TRue` or `false`. Recall from [chapter 2](#) that `true` and `false` are actual values in C++, just like numbers.

## Note

Comparisons in logical expressions are also called *conditions*.

You've actually used logical operators in your programs already. [chapter 2](#) showed their use with `while` and `do-while` loops. [Table 3.2](#) gives some more examples of how to use logical operators.

**Table 3.2. Using Logical Operators In Expressions**

Expression	Meaning
<code>thisValue &lt;= thatValue</code>	Evaluates to <code>true</code> if the variable <code>thisValue</code> contains a value that is less than or equal to the value in the variable <code>thatValue</code> . Otherwise, it evaluates to <code>false</code> .
<code>thisValue != 100</code>	Evaluates to <code>true</code> if <code>thisValue</code> is not equal to 100. Otherwise, it evaluates to <code>false</code> .
<code>thisValue == 100</code>	Evaluates to <code>TRue</code> if <code>thisValue</code> is equal to 100. Otherwise, it evaluates to <code>false</code> .
<code>!(thisValue == 100)</code>	Evaluates to <code>true</code> if <code>thisValue</code> is not equal to 100. Otherwise, it evaluates to <code>false</code> .
<code>(thisValue &lt; 100) &amp;&amp; (thatValue != 10)</code>	Evaluates to <code>true</code> if <code>thisValue</code> is less than 100 and <code>that Value</code> is not equal to 10. Otherwise, it evaluates to <code>false</code> .
<code>(thisValue &lt; 100)    (thatValue != 10)</code>	Evaluates to <code>true</code> if <code>thisValue</code> is less than 100 or <code>that Value</code> is not equal to 10. Otherwise, it evaluates to <code>false</code> .

Some of the expressions in [Table 3.2](#) deserve particular attention. The

expression

```
!(thisValue == 100)
```

says the same thing as

```
thisValue != 100
```

To understand this expression, you need to know that in C++, everything in parentheses is done first. To evaluate

```
!(thisValue == 100)
```

you first evaluate

```
thisValue == 100
```

because it's in the parentheses. This expression evaluates to `true` when the variable

`thisValue` contains 100. If it contains any other value, the expression is `false`. If you then add the Not operator, which is the exclamation point, it negates the expression. That means if

```
thisValue == 100
```

evaluates to `TRue`, then

```
!(thisValue == 100)
```

evaluates to `false`. And any time



`thisValue == 100`

evaluates to `false`, it makes

`!(thisValue == 100)`

evaluate to `true`. The `!` operator negates a logical condition.

## Factoid

My experience as a college professor is that beginning programmers are often uncomfortable with using the `!` operator. If you feel that way, don't bother using it. Any logical expression you can write with the `!` operator can be written without it if you use other logical operators.

The last two expressions in [Table 3.2](#) are also of particular interest. These are often called *[compound logical expressions](#)* because they put together more than one comparison or condition. When you're writing your programs, you make compound logical expressions with the And operator. You can also create them with the Or operator. How is this done?

The expression

`(thisValue < 100) && (thatValue != 10)`

has two conditions:

`thisValue < 100`

and

thatValue!=10

In order for the entire expression to evaluate to `true`, both the first *and* the second conditions must evaluate to `true`. If either of them evaluates to `false`, then the whole expression is `false`. As a result, expressions with an And operator tend to evaluate to `false` more than they evaluate to `true`.

If we rewrite the expression to make this

```
(thisValue<100) || (thatValue!=10)
```

then we're using the Or operator. This says that the entire expression is `true` if either the first *or* the second condition evaluates to `true`. The only time it's `false` is if both conditions are `false`.

# The If-Else Statement

The `if-else` statement gives C++ programs a way to make decisions based on the information the program has as it's running. Let's use [listing 3.7](#) to see how it works.

## Listing 3.7. A simple `if-else` statement

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      cout << "Please input a number greater than or equal to 0: ";
9
10     int aNumber;
11
12     cin >> aNumber;
13
14     if (aNumber >= 0)
15     {
16         cout << "Very good." << endl;
17     }
18     else
19     {
20         cout << "Not very good." << endl;
21     }
22
23     system("PAUSE");
24     return (EXIT_SUCCESS);
25 }
```

You create an `if-else` statement to make a decision, as [Listing 3.7](#) illustrates. In this example, the program asks the user to enter a number. It makes the decision on lines 14-21. If the user entered a number greater than or equal to 0, the program prints a compliment (line 16). If the number the user entered is less than 0, the program prints another message (line 20).

All `if-else` statements follow the format of the one in [Listing 3.7](#). To be specific, they all start with the C++ keyword `if`, which is always followed by a logical expression in parentheses. Like a loop, `if-else` statements can have bodies. Of course, the bodies are enclosed in opening and closing braces. The `if-else` statement is actually composed of two parts, or clauses. They all have a required `if` clause and an optional `else` clause. The `if` clause for the program in

[Listing 3.7](#) appears on lines 14-17. As you can see, the `if` clause has a code body on lines 15-17. The body of the `else` clause spans lines 19-21.

Because the `if` and `else` clauses in an `if-else` statement each have their own code bodies, the program executes different blocks of code based on the condition. In other words, `if` the condition on line 14 is `true`, the program executes the body of the `if` clause on lines 15-17. On the other hand, when the condition on line 14 is `false`, the program executes the code body of the `else` clause. As a result, the program prints the string on line 16 if the condition is `TRue`. If the condition is `false`, it prints the string on line 20.

With all `if-else` statements, the `else` clause is optional. You don't have to put it in. [Listing 3.8](#) shows the same `if-else` statement without an `else` clause.

### Listing 3.8. An `if` statement without an `else` clause

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     cout << "Please input a number greater than or equal to 0: ";
9
10    int aNumber;
11
12    cin >> aNumber;
13
14    if (aNumber >= 0)
15    {
16        cout << "Very good." << endl;
17    }
18
19    system("PAUSE");
20    return (EXIT_SUCCESS);
21 }
```

If you run this program and the condition is `TRue`, it executes the body of the `if` statement. However, when you run this program and the condition evaluates to `false`, the program skips to the next statement after the `if`, which is on line 19. Essentially, this `if` does nothing when its condition is `false`.

### Factoid

Programmers often refer to an `if` without an `else` as an `if` statement.

When an `else` clause is present, they tend to call it an `if-else` statement.

# Namespaces and Scope Resolution

In C++, the name of every class in a program has to be unique; there can't be two classes with the same name. This presents a potential problem. Suppose you write a game that uses the LlamaWorks2D library. LlamaWorks2D contains a class called `vector`, which I'll present in [chapter 5](#), "Function and Operator Overloading." Games contain a lot of math. Imagine that you decide that you will also use a math library written by someone else in your game. It's very likely that the math library also contains a class called `vector`. When your game declares a variable of type `vector`, which class should the compiler use, the one in LlamaWorks2D or the one in the math library?

C++ provides a solution to the problem of conflicting class names: [namespaces](#). A namespace is a way of grouping related classes together. In fact, it can be used for more than classes. Namespaces group related types, functions, and other kinds of C++ constructs that I'll discuss later. You create a namespace with the `namespace` keyword, as shown in [Listing 3.9](#).

## Listing 3.9. Defining a namespace

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  // Beginning of first namespace.
7  namespace anamespace
8  {
9
10 class point
11 {
12 public:
13     point()
14     {
15         x = y = 0;
16     }
17
18     void SetX(int xValue)
19     {
20         x = xValue;
21     }
22
23     int GetX(void)
24     {
25         return (x);
26     }
27
28     void SetY(int yValue)
29     {
```

```
30     y = yValue;
31 }
32
33 int GetY(void)
34 {
35     return (y);
36 }
37
38 private:
39     int x,y;
40 };
41
42 // End of first namespace.
43 };
44
45
46
47 // Beginning of second namespace.
48
49 namespace anothernamespace
50 {
51     class point
52     {
53     public:
54         point()
55         {
56             x = y = 0;
57         }
58
59         void SetX(int xValue)
60         {
61             x = xValue;
62         }
63
64         int GetX(void)
65         {
66             return (x);
67         }
68
69         void SetY(int yValue)
70         {
71             y = yValue;
72         }
73
74         int GetY(void)
75         {
76             return (y);
77         }
78
79         void Reset()
80         {
81             x = y = 0;
82         }
83
84     private:
85         int x,y;
86     };
87
88 // End of second namespace.
89
90 };
```

```

91
92
93
94 int main(int argc, char *argv[])
95 {
96     anamespace::point rightHere;
97
98     rightHere.SetX(10);
99     rightHere.SetY(20);
100
101     cout << "(x,y)=" << rightHere.GetX();
102     cout << ", " << rightHere.GetY() << ")";
103     cout << endl;
104
105     anothernamespace::point rightThere;
106     rightThere.SetX(20);
107     rightThere.SetY(10);
108
109     cout << "(x,y)=" << rightThere.GetX();
110     cout << ", " << rightThere.GetY() << ")";
111     cout << endl;
112
113     rightThere.Reset();
114
115     cout << "(x,y)=" << rightThere.GetX();
116     cout << ", " << rightThere.GetY() << ")";
117     cout << endl;
118
119     system("PAUSE");
120     return (EXIT_SUCCESS);
121 }

```

This sample program defines two namespaces: one called `anamespace` and the other called `anothernamespace`. Both namespaces contain a class called `point`. If you examine the two classes, you'll find that they're nearly identical. The difference is that the `point` class in `anothernamespace` has a member function called `Reset()`. The `point` class in `anamespace` doesn't have that member function.

The program in [Listing 3.9](#) uses both `point` classes without a problem. The compiler can tell which `point` class is being referred to because the `main()` function uses the scope resolution operator, which is the double colon (`::`). You see it on lines 96 and 105. On line 96, the `main()` function states that it is declaring a variable of type `point` and using the `point` class in the namespace called `anamespace`. Line 105 also declares a variable of type `point`, but it is the `point` class in the namespace called `anothernamespace`.

Every time you declare a variable that uses a type from a namespace, you must use the scope resolution operator. This makes your code really wordy. Most programmers don't care for this. The way you get around it is to put the `using` keyword into your program. On line 4 of [Listing 3.9](#), you can see an



example of the `using` statement. This line specifies that the program is using a namespace called `std`. The `std` namespace is defined by the C++ Standard Libraries in the file `iostream`, which is included on line 2. Pretty much all of the programs presented so far have used this namespace; most C++ programs do.

When you start using the LlamaWorks2D game engine, you'll see that all of its types are defined in a namespace called `llamaworks2d`. As a result, you'll put the statement

```
using namespace llamaworks2d;
```

at the beginning of your source code for your games. Having the `using` statement at the beginning of your files means that you can use types that LlamaWorks2D provides, such as `vector`, without having to specify `llamaworks2d::vector` throughout your code. If you use a `vector` class from another library, that's okay. Any time there's a conflict, you can resolve it by using the scope resolution operator to specify exactly which `vector` class you mean.

# A Brief Word About Structures

Before we dive into writing games with LlamaWorks2D, which is the subject of the next chapter, there's one more topic I'd like to touch on briefly. That topic is structures.

Occasionally you will want to create a class that is so simple that it doesn't seem worth it to write member functions to get and set its values. If that's the case, you should not make it a class. Instead, you should make it a structure.

A structure is almost exactly the same as a class. A structure can have member functions, constructors, destructors, and member data just like classes. It can have public data, private data, public functions, and private functions just the same as a class. The only difference between classes and structures is the default for all class members is private while the default for structures is public.

For example, suppose I define a class like this:

```
class bitmap_region
{
    int top,left,bottom,right;
};
```

This class does not contain the keywords `public` or `private`. By default, the four data members are set to private access. That means only member functions can access them. However, there are no member functions. If I change this to a structure, the scope changes as well. Here's the same thing as a structure:

```
struct bitmap_region
{
    int top,left,bottom,right;
};
```

## Factoid

The LlamaWorks2D game engine provides only a few types that are structures. They are mostly used for program initialization and a few other minor tasks. LlamaWorks2D provides far more classes

than structures.

As shown here, all structures begin with the keyword `struct`. This structure does not contain the keywords `public` or `private`. By default, its members are public. They can be accessed by any function in the program that has a `bitmap_region` variable.

In general, you should not use many structures in your program because their members are public by default. Whenever you can, it's better to use classes that keep their data private and their functions public.

# Summary

We've covered a lot of territory in one chapter. You've had a whirlwind introduction to object-oriented programming. Don't expect to master everything presented here right away; we'll practice everything you learned in this chapter throughout the rest of the book. Along the way, I'll explain in detail some of the C++ features I just touched on in this chapter. Before the end of this book, you'll have used all of these concepts and techniques repeatedly so that when you start writing your own games, you'll have a strong background to draw from.

With the tools that object oriented programming provides, you'll be able to make software objects that you can use and reuse in your games. You'll also be able to use the LlamaWorks2D game engine, which is based on software objects.

# Chapter 4. Introducing the LlamaWorks2D Game Engine

At last, it's time to do some actual game programming. First, we'll present a short overview of the LlamaWorks2D game engine. Next, we'll examine some simple programs written with LlamaWorks2D. However, before we do, I want to emphasize that this chapter is an *overview*. As a result, there will be things in this chapter that I'll show you how to do but put off explaining why until the next few chapters. This style of presentation makes it easier to get right into game programming.



# A Step-by-Step Overview

I mentioned in previous chapters that when I started into graphics programming, you had to do *everything* yourself. When graphics libraries such as OpenGL were released, it helped a lot. However, if you ever attempt to write an OpenGL program without LlamaWorks2D, you'll quickly discover one thing: there are lots of ways to get a blank screen. It's very easy to spend many frustrated hours trying to figure out why you're not seeing anything but an empty screen. Been there, done that, got the t-shirt.

The great thing about LlamaWorks2D is that it's incredibly easy to use. I designed it to make game programming as simple as possible. When you're starting out, you can ignore almost everything about the engine and you'll still get a decent result. It is written so that it provides a reasonable default value for virtually everything. Therefore, you can often avoid specifying information that the engine needs because anything you don't specify uses the default value. And when you use the default values, you should still see something reasonable on the screen.

To write a LlamaWorks2D program you must create a project in Dev-C++, configure it appropriately, and then start writing your game code. In your game code, you've got to create a software object called a [\*game class\*](#). The game class requires some important information, but LlamaWorks2D provides straightforward ways of specifying it. Typically, an OpenGL or DirectX program requires that you write hundreds of lines of code before you can even get a working, but empty, window on the screen. With LlamaWorks2D, you can do that in fewer than 20 lines of code. Let's see how.

## Phase 1: Create a Dev-C++ Project

It's time to create the project that will become your first LlamaWorks2D program. Here's how it's done.

### Create a Project File and Add a .Cpp File to It.

1. In your My Documents folder, create a folder called **Dev-c++ projects**.
2. In the Dev-C++ Projects folder, create a folder called **Simplegame**.

3. Insert the CD that comes with this book. Navigate to the folder `<d>:\Source\Chapter04\Prog_04_01`, where `<d>` is the letter of your CD-ROM drive. In that folder, you'll find a file called `Prog_04_01.cpp`. Copy it to the SimpleGame folder you created in step 2.
4. Start Dev-C++ if it is not already running. From the File menu, select New and then Project.
5. In the New Project dialog box, click on the Windows Application icon.
6. Enter the name **simplegame** in the Name edit box. Make sure that the C++ Project radio button is selected. Click OK.
7. In the dialog box that appears, navigate to the SimpleGame folder you created in step 2. Click Save.
8. Right-click the `main.cpp` tab at the top of the edit window. In the menu that appears, choose Close, and then select No in the dialog box.
9. Right-click the name SimpleGame in the Project pane that is along the left side of the Dev-C++ window. Choose Add to Project.
10. In the dialog box that appears, find the SimpleGame folder that you created in Step 2. When you do, select the file `Prog_04_01.cpp`. Now click the Open button. This adds the file to the project.

That's all it takes to create the project.

## Phase 2: Configure the Dev-C++ Project

The next task is to configure your project so that you can use it with LlamaWorks2D and OpenGL. When you installed Dev-C++ and OpenGL in [chapter 2](#), you hopefully configured Dev-C++ to use OpenGL in all of your programs. Check to ensure that's done with the following instructions.

### Ensure that Dev-C++ is Ready to Use OpenGL.

1. From the Dev-C++ Tools menu, select Compiler Options.

2. Ensure that there is a check in the check box that's labeled "Add these commands to the linker command line". If the check isn't there, add it now.

In the edit box associated with the "Add these commands to the linker command line" check box, add the following command if it is not there already:

```
<d>:\Dev-Cpp\lib\libopengl32.a
```

3. Replace the `<d>` in this command with the letter of the drive where you installed Dev-C++. This is usually the C drive. If that's the case on your computer, the command should read

```
C:\Dev-Cpp\lib\libopengl32.a
```

4. Clear the check from the box labeled "Use fast but imperfect dependency generation". Click OK.

You're now ready to use OpenGL.

You also have to configure your project to use LlamaWorks2D. In the Introduction, the instructions had you install LlamaWorks2D into a folder in your Dev-C++ program directory. If you did not do that, please do it now using the instructions from the Introduction in the section called "Installing LlamaWorks2D." Once that's done, you can configure your project to use LlamaWorks2D.

## Warning

Setting Dev-C++ to always link to the OpenGL library makes things much easier for the purposes of this book. However, if you want to use Dev-C++ to develop programs that do not use OpenGL, you must uncheck the box you checked in Step 2 and remove the command you inserted in Step 3.



## Put LlamaWorks2D Into Your Program.

1. Use Windows Explorer to copy the LlamaWorks2D files into the SimpleGame folder. If you followed the instructions in the Introduction, it will be in your Dev-Cpp folder, which is probably on your C drive. The whole path will be `<d>:\Dev-Cpp\LlamaWorks2D\Source`, where `<d>` is the letter of the drive where your Dev-Cpp folder resides.

2. After you copy all the LlamaWorks2D files into the SimpleGame folder, right-click the project name in the Project pane. Select Add folder from the menu that appears. In the resulting dialog box, type **llamaworks2d** and then click OK.

3. Right-click the LlamaWorks2D folder in the Project pane, and choose Add to Project from the context menu. In the dialog box that appears, navigate to the SimpleGame folder. Select all of the LlamaWorks2D files that you copied in Step 1. Click Open.

4. Right-click the project name (SimpleGame) in the Project pane and select Project Options from the context menu.

5. In the Project Options dialog box, click the Directories tab. On the Directories page, click the Include Directories tab. In the lower, single-line edit box, type the path to your LlamaWorks2D\Source directory. Most likely, the path will be `C:\Dev-Cpp\LlamaWorks2D\Source`. Click the Add button and then click OK.

Your Dev-C++ project is ready. Now let's write a LlamaWorks2D program.

## Phase 3: Write a Game Class

To start writing a game with LlamaWorks2D, you must create a game class. Writing a game class is essentially like writing any other C++ class. Game classes have public member functions and private member data. They also require some additional features that typical C++ classes may not have.

I've provided a file, called `Prog_04_01.cpp`, that contains a minimal game class. It's on the CD in the folder `\Source\Chapter04\Prog_04_01`. You added

it to your project in Phase 1. [Listing 4.1](#) shows its contents.

## Listing 4.1. The smallest possible LlamaWorks2D program

```
1 #include "LlamaWorks2d.h"
2
3 using namespace llamaworks2d;
4
5 class my_game : public game
6 {
7 public:
8     ATTACH_MESSAGE_MAP;
9 private:
10 };
11
12
13 CREATE_GAME_OBJECT(my_game);
14
15 START_MESSAGE_MAP(my_game)
16 END_MESSAGE_MAP(my_game)
```

The file Prog\_04\_01.cpp begins by including the file LlamaWorks2D.h. Recall from [chapter 2](#) that `#include` statements are a way of providing the compiler with the definitions that your program needs to compile. The file LlamaWorks2D.h is a kind of "ultimate include file" in the LlamaWorks2D game engine. All of the files you create for your games must include LlamaWorks2D.h. So you should put the statement you see on line 1 of [Listing 4.1](#) at the beginning of all your C++ source files in your games.

### Factoid

Programmers have a couple of different names for files they use with `#include` statements. They'll refer to them as either include files or header files.

The statement on line 3 must also be present to use the LlamaWorks2D game engine. Back in [chapter 3](#), you saw that namespaces help group types, functions, and so forth into related collections. Everything the LlamaWorks2D game engine provides is in the `llamaworks2d` namespace. The `using` statement on line 3 saves you from having to put `llamaworks2d::` at the beginning of every variable

declaration in your program.

The definition of the game class itself begins on line 5. Notice at the end of the line that there's a colon, followed by the C++ keyword `public`, followed by the name `game`. I'll explain exactly what all that means in [chapter 6](#), "Inheritance: Getting a Lot for a Little." For now, just know that it's required in your game class definition.

The statement on line 8 is a special marker called a *macro*. What it does is copy a whole bunch of C++ source code into your game class for you. The source code this macro inserts is part of the LlamaWorks2D engine. It adds a *message map* to your game class. A message map is a technique for assigning specific functions to handle player input. Message maps are widely used in all types of programs. (Message maps are presented in more detail in [chapter 7](#), "Game Program Structure.") Every game class must have a message map. It does not matter whether your LlamaWorks2D program actually accepts player input. In fact, the program in [Listing 4.1](#) ignores all player input. Even so, the game class in [Listing 4.1](#) must have the statement shown on line 8 or it will not compile.

In addition to `ATTACH_MESSAGE_MAP`, LlamaWorks2D also provides you with other macros that insert a lot of source code into your game. This simplifies your life by saving you from having to write large amounts of code yourself. Line 13 of [Listing 4.1](#) uses a macro to create an object from your game class. Recall that classes define what objects contain, but you don't have an object until you declare a variable. The macro on line 13 creates a variable of type `my_game`.

## Warning

If you forget the `ATTACH_MESSAGE_MAP;` statement in your game class, the compiler outputs an error telling you that the function `ProcessInputMessageMap()` is not a member of your game class. If you see this error, you know you are missing the `ATTACH_MESSAGE_MAP;` statement in your game class.

If you change the name of your game class to something other than `my_game`, you need to put that name within the parentheses of the `CREATE_GAME_OBJECT()` macro; it has to contain the type name of your game class. In addition, this macro must always appear right after the definition of your game class.

The macros you see on lines 15 and 16 mark the beginning and ending of the game class's message map. As I mentioned, the message map does nothing at this point. However, it must be present for your game to compile.

## Warning

If the compiler outputs an error stating that it cannot find the function `CreateGameObject()`, you did not put the `CREATE_GAME_OBJECT()` macro in your program. If it tells you that it cannot find the function `ProcessInputMessageMap()`, you don't have the `START_MESSAGE_MAP()` macro in your program. Whenever the compiler tells you that it found the beginning of another function before the end of `ProcessInputMessageMap()`, that means you left out the `END_MESSAGE_MAP()` macro.

## Phase 4: Compile and Run the Program

Now comes the easy part. If you've done everything correctly, all you have to do is press F9 or select Compile & Run from the Dev-C++ Execute menu. Assuming all goes well, you'll see a nice blue window that covers most of your screen. Press Alt+F4 to close it.

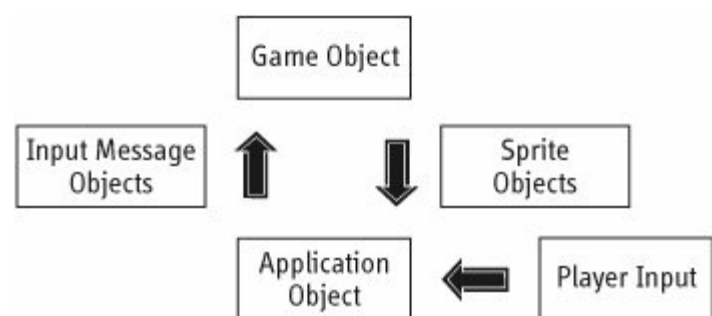
That blue window may not seem like much, but it hides a lot that is going behind the scenes. The game engine initialized Windows and OpenGL for you. It sized and positioned to the window, and set its background color. While you gaze into the window's relaxing blueness, the game engine is displaying lots and lots of empty frames per second. It's checking for player input, even though it doesn't do anything with the input. It sits patiently waiting for you to add your unique and dazzling game elements so that it can display them. Not bad for just 16 lines of code.

# How Does Llamaworks2D Work?

LlamaWorks2D is built around objects. Remember how I said in [chapter 3](#) that objects could represent anything? LlamaWorks2D uses that ability to hide many of the mundane tasks of game programming from you. That's a good thing. The tasks it hides from you are the ones that are not directly related to making games. They're all about Windows and OpenGL.

LlamaWorks2D uses an application object and a game object to do most of its work. The application object represents the program itself. The game object is your game class that you saw back in [Listing 4.1](#) [Figure 4.1](#) illustrates how these objects work.

**Figure 4.1. The design of LlamaWorks2D.**



## Warning

When you follow the instructions presented here to create and compile your program, the compiler names the program SimpleGame.exe. On the CD, I've provided a compiled version of this program. However, it's called Prog\_04\_01.exe I did that to make it easier to figure out which program goes with which example. When you're writing games, the name of your game's executable file should be something that resembles the name of the game.

The game object in [Figure 4.1](#) represents the portion of the program that is

your game. When you write a game with LlamaWorks2D, you put all of your game's data and all of the functions that make your game operate into your game class. Note that you can also define other classes that your game class uses. But the code that defines what your game is and how it works goes into the game class. Of all the objects in [Figure 4.1](#), the game class is the only one you write. LlamaWorks2D provides you with the message objects, sprite objects (I'll explain sprite objects later in this chapter when we create a stationary ball), and the application object.

By using an application object and a game object, you ensure that your game code is separated from the code that makes the overall program run. That means you don't have to bother with the code that gets Windows and OpenGL up and running. You don't have to know all of the ins and outs of handling user input (there are many). [Figure 4.1](#) shows that player input comes into the application object, which packages that input as message objects that your game can handle fairly easily.

## Note

You may be wondering whether you should learn how to initialize Windows and OpenGL yourself. The answer is yes, but not now. When you feel like you're a proficient C++ programmer, feel free to go through the source code for LlamaWorks2D. It will help you learn both Windows and OpenGL programming. Also, you should go through the Windows and OpenGL programming documentation at [www.msdn.microsoft.com](http://www.msdn.microsoft.com).

In addition, you don't have to know how OpenGL does bitmap animation. It's not a topic most beginners want to tackle. Instead, LlamaWorks2D provides a `sprite` object that handles all that for you. Almost all the animation you do in 2D games can be handled with just the `sprite` class.

As you become more proficient with C++, Windows, and OpenGL, you can easily enhance or customize the objects LlamaWorks2D provides. There is nothing in LlamaWorks2D that prevents you from adding your own objects. You can even bypass any of the engine's components and handle the tasks yourself. This enables you to perform those tasks in the way that is best for your game.

LlamaWorks2D creates an application object itself. However, the engine does

need a bit of configuration information to do so. If you don't provide that information (we didn't in [Listing 4.1](#)), LlamaWorks2D uses a reasonable set of default values. The application object is accessible from anywhere in the program. It is the only object in the program that is set up that way.

LlamaWorks2D also creates the game object using your game class when you use the macro `CREATE_GAME_OBJECT()`. Functions in your game will need to access the game object at times. When they do, they call a function in the application object that provides them with access to the game object. The name of that function is `Game()`.

# A Stationary Ball

To how see the LlamaWorks2D game engine draws things on the screen, we'll create a program that displays a ball. Then in the following section, we'll animate it by bouncing it around the screen.

The basic process for displaying a ball on a screen is straightforward. Your game class must create the ball as a sprite. A sprite is anything that moves on the screen. That includes monsters, people, oozing lava, or anything else. Even though the ball for this program won't move, we'll still create it as a sprite so that we can animate it easily in the next section.

After it creates the sprite, your game must display the sprite on the screen. Graphics programmers call this process [rendering](#). For each frame, the function that does the rendering must clear the screen and draw the ball. The position and image of the ball are stored in the `sprite` object. [Listing 4.2](#) shows how to make all of this happen.

## Listing 4.2. Creating and rendering a stationary ball

```
1  #include "LlamaWorks2d.h"
2
3  using namespace llamaworks2d;
4
5
6  class my_game : public game
7  {
8  public:
9      bool InitGame();
10     bool RenderFrame();
11
12     ATTACH_MESSAGE_MAP;
13 private:
14     sprite theBall;
15 };
16
17
18 CREATE_GAME_OBJECT(my_game);
19
20 START_MESSAGE_MAP(my_game)
21 END_MESSAGE_MAP(my_game)
22
23 bool my_game::InitGame()
24 {
25     bool initOK;
26
27     theBall.BitmapTransparentColor(color_ rgb(0.0f,1.0f,0.0f));
28     initOK =
```



```

29  theBall.LoadImage(
30  "ball.bmp",
31  image_file_base::LWIFF_WINDOWS_BMP);
32  if (initOK==true)
33  {
34  theBall.X(1);
35  theBall.Y(1);
36  }
37  else
38  {
39  ::MessageBox(
40  NULL,
41  theApp.AppError().ErrorMessage().c_str(),
42  NULL,
43  MB_OK | MB_ICONSTOP | MB_SYSTEMMODAL);
44  }
45  return (initOK);
46  }
47
48  bool my_game::RenderFrame()
49  {
50
51  theBall.Render();
52  return true;
53  }

```

As you might expect, the game class has changed a bit for this program. Lines 910 of [Listing 4.2](#) show the prototypes for two new functions. The game engine requires these functions, but it provides them for you if you don't put them in.

LlamaWorks2D automatically calls the `InitGame()` function when your game starts up. Therefore, you should put all of the code that is required to get your game going into the `InitGame()` function. If your game doesn't use elaborate startup menus, this is not very much code.

## Factoid

In case you're interested, the way that LlamaWorks2D provides the `InitGame()` and `RenderFrame()` functions when you don't write them is through inheritance. The inheritance happens on line 6. However, if you don't know what inheritance is yet and don't want to fuss with it, then just continue reading this chapter. As I mentioned previously, inheritance is explained in [chapter 6](#), "Inheritance: Getting a Lot for a Little."

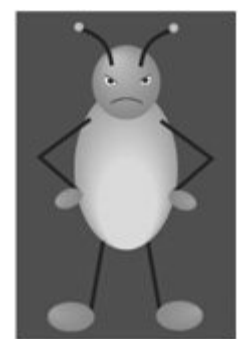
During every frame of animation, LlamaWorks2D calls the `RenderFrame()` function. All of the code that draws one animation frame goes here. At the beginning of each frame, LlamaWorks2D automatically clears the background. So your code should paint a background, if you need one, and then draw anything that goes in front of it. The code in the `RenderFrame()` function should work the same way people paint paintings that is, everything should be painted from back to front. That way, objects that are closer to the viewer will appear to be in front of objects that are farther away.

Right now, the `RenderFrame()` function is quite simple. It just calls the `sprite` class's `Render()` function and the ball renders itself. In general, you'll render your entire scene this way. It often makes rendering one of the easiest tasks in the game.

In addition to the two new member functions, the `my_game` class gained a private data member, as you can see on line 14. The data member is of type `sprite`. As [Figure 4.1](#) showed, LlamaWorks2D provides the `sprite` class for your games. Sprites represent anything moving on the screen. When your program creates an object of type `sprite`, it gives itself a place to store information about the ball.

The code for the two new member functions appears on lines 2353. The first is the `InitGame()` function. After the `InitGame()` function declares a variable, it sets the ball's transparent color. The transparent color is the color that you don't want displayed. [Figure 4.2](#) illustrates what the transparent color is used for.

### **Figure 4.2. The return of the Krelnorian.**



[Figure 4.2](#) shows the Krelnorian you saw in [chapter 1](#). The Krelnorian's bitmap is rectangular, but the Krelnorian itself isn't. Most things aren't rectangular, but bitmaps are. That means there is essentially empty space in the bitmap that we don't want shown. In [Figure 4.2](#), we don't want the game to show the dark area around the Krelnorian. To make part of a bitmap transparent, you first set the entire transparent area to the same color. Then you tell the game

engine what the transparent color is, and it handles the rest. It displays the bitmap without displaying the transparent area.

## Warning

Be careful that you do not accidentally use the transparent color in parts of the bitmap you do not want to be transparent.

When your game loads the ball's bitmap image, it tests every pixel against the transparent color. The `sprite` class works some special magic on all pixels that are of the transparent color to convert them into transparent pixels. On line 27, all pixels that are pure green (no red, no blue, and green set to maximum) are made transparent.

The files that go with [Listing 4.2](#) are on the CD in the folder `Source\Chapter04\Prog_04_02`. The program file is `Prog_04_02.cpp`. The folder also contains `Ball.bmp`, which is the bitmap image of the ball. All of the transparent pixels in `Ball.bmp` are set to pure green. The floating-point color values for pure green are (0,0, 1.0, 0.0). The integer color values for pure green are (0,255,0).

## Factoid

You can specify the transparent color as floating-point numbers or as integers. If you use floating-point numbers, the color values must be between 0.0 and 1.0, inclusive. If you use integers, valid color values are between 0 and 255, inclusive.

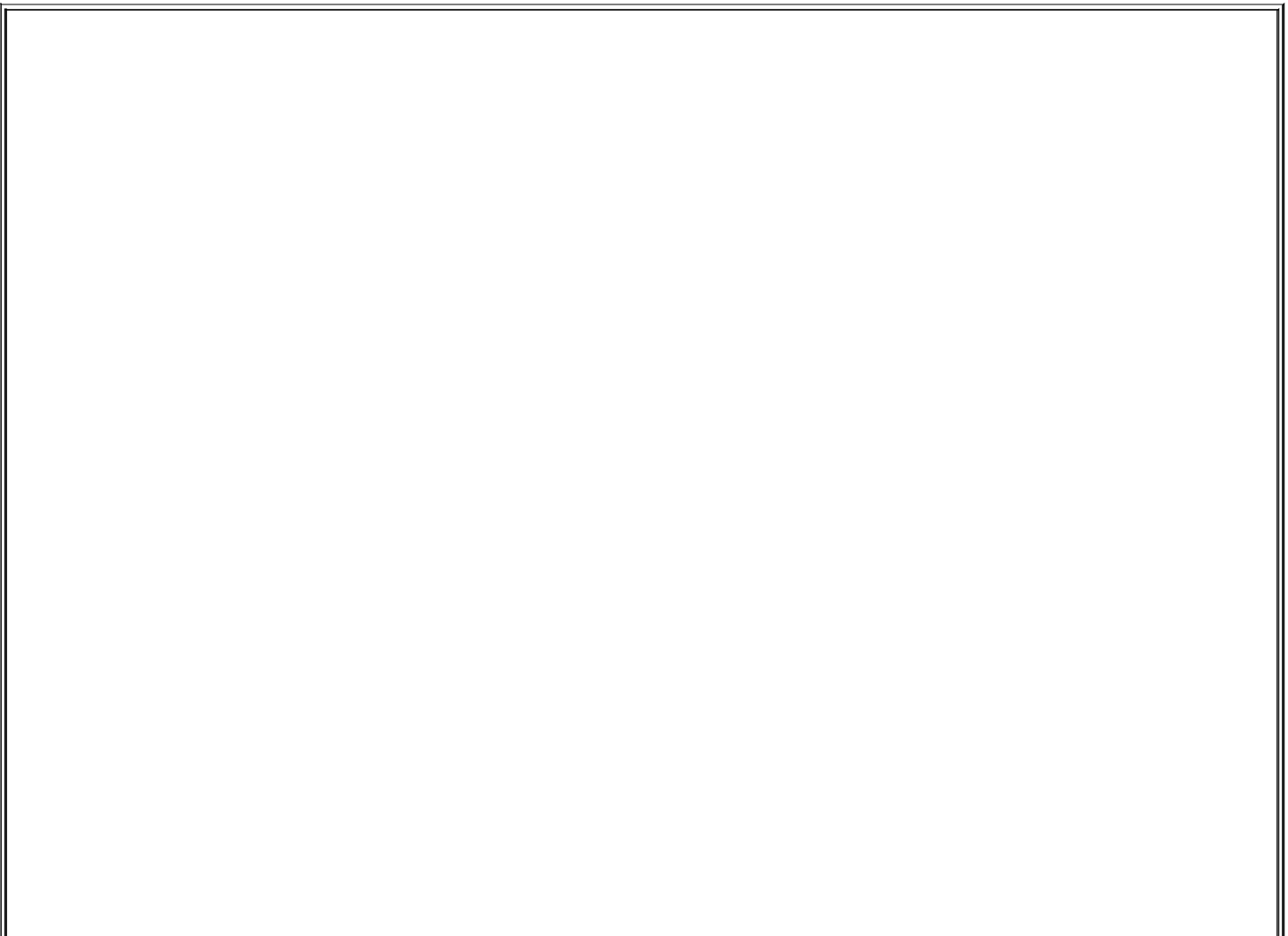
Next, the `InitGame()` function uses the `initOK` variable to store the results of the initialization. As lines 2831 show, the initialization is successful if the ball's image is loaded. The `if` statement on lines 3244 tests to see whether the initialization is successful. If it is, the statements on lines 3435 use the `sprite` class's `x()` and `y()` functions to set the position of the upper-left corner of the ball's bitmap image. If the initialization is not successful, the statements on lines 3943 display an error message.

## Warning

The source code for this program is on the CD in the folder Source\Chapter04\Prog\_04\_02. If you compile this program yourself, you'll need to copy both the .cpp and .bmp files in the folder into a folder on your computer's hard drive. If you forget to copy the file ball.bmp, the program will compile properly; however, it won't be able to load the ball's bitmap image because it's not in the same folder as the file Prog\_04\_02.cpp.

The `RenderFrame()` function, which begins on line 48, is where your game renders all of its sprites. In this case, there's only one sprite, so the function uses the `sprite` class's `Render()` function to display the ball's bitmap image.

It's important to note that both the `InitGame()` and `Render()` functions must return `true` if they succeed and `false` if they do not. This is how LlamaWorks2D knows whether or not there is an error.



## BMP Files, Transparent Pixels, and Professional Game Art

The LlamaWorks2D `sprite` class makes it easy for you to use Windows BMP files with your games. It enables you create all the art for your games with nothing more complicated than Windows Paint.

You should be aware, however, that professional games rarely use BMP files for their game art. One of the primary reasons for this is that BMP files are not compressed. They are huge compared to other file formats. Common compressed file formats are JPEG, GIF, TIF, and TGA. To enable you to use these file formats, I've included a powerful free paint program on the CD that you can use in place of Windows Paint. The program is called Gnu Image Manipulation Program, or GIMP. I strongly encourage you to use it rather than Windows Paint. You'll find it in the `\Tools\GIMP` folder.

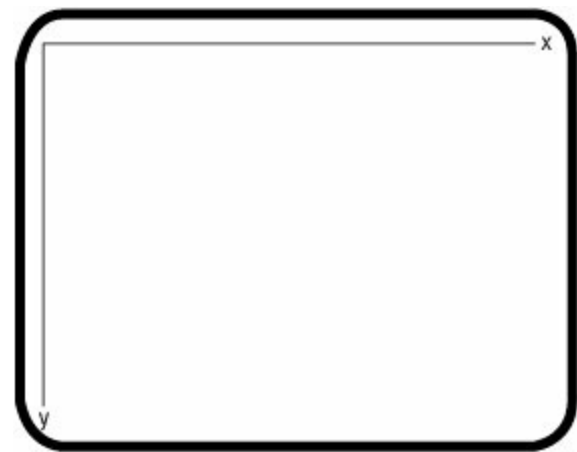
Another reason for not using BMP files is that BMP files are not designed for transparent pixels. As the program in [Listing 4.2](#) shows, I have done some special things with the `llamaworks2d::sprite` class to add transparent pixels to BMP files. The disadvantage to the technique I use here is that *all* pixels that are pure green are converted to transparent pixels. You cannot use the transparent color for opaque pixels. Unlike BMP files, file formats such as TGA (Truevision's TARGA format) do support transparent pixels. You can use GIMP to mark any pixels in a TGA file as transparent. It doesn't matter what color the pixel is. Therefore, you don't have to use a specific color as a transparent color. Instead, you add an extra 8 bits of information called an [alpha channel](#). The alpha channel specifies how transparent a pixel is. An alpha value of 0.0 means the pixel is opaque, and an alpha value of 1.0 means it is transparent.

The free version of LlamaWorks2D that you get with this book does not support TGA files or alpha channels.

# A Bouncing Ball

The final example program for this chapter demonstrates how to move the ball. Because sprites are supposed to move, the `sprite` class is able to keep track of how much it should move and in what direction. It has a member function called `Movement()` that your game can use to set the direction and speed of the sprite's movement. To do this, your game passes a `vector` object that contains x and y values to the `Movement()` function in its parameter list. The movement you specify is in pixels. The x and y values are values in a Cartesian coordinate system. Every computer screen, and every window on a computer screen, has an x and a y axis. Windows sets the positive x axis so that it points from left to right and the positive y axis so that it points down. This is probably different than what you did when you learned Cartesian coordinate systems, but it is valid. However, it means that location 0,0 is at the *upper-left* corner of the screen. This is illustrated in [Figure 4.3](#).

**Figure 4.3. The origin of the Windows screen coordinate system is at the upper-left corner of the screen.**



## Note

If you ever decide to do games with DirectX, you will use the Windows-style coordinate system with the origin at the upper-left corner of the screen. DirectX provides a function that helps you load a bitmap properly by inverting it when it's loaded.

The Windows-style coordinate system means that you have to invert all of your bitmaps when you load them or they will be displayed upside down. However, the `sprite` class takes care of that for you by inverting bitmaps when it loads them.

As you might expect, the bouncing ball program is much larger than the stationary ball program. During each frame, the program has to move the ball. It also has to check whether the ball hit the edge of the screen or went off the edge. If so, it may have to adjust the ball's position before it bounces the ball.

[Listing 4.3](#) shows how it's done.

## Listing 4.3. The bouncing ball program

```
1 #include "LlamaWorks2d.h"
2
3 using namespace llamaworks2d;
4
5
6 class my_game : public game
7 {
8 public:
9     bool OnAppLoad();
10    bool InitGame();
11    bool UpdateFrame();
12    bool RenderFrame();
13
14    ATTACH_MESSAGE_MAP;
15 private:
16    sprite theBall;
17 };
18
19
20 CREATE_GAME_OBJECT(my_game);
21
22 START_MESSAGE_MAP(my_game)
23 END_MESSAGE_MAP(my_game)
24
25 bool my_game::OnAppLoad()
26 {
27     bool initOK = true;
28
29     //
30     // Initialize the window parameters.
31     //
32     init_params lw2dInitParams;
33     lw2dInitParams.openParams.fullScreenWindow = true;
34     lw2dInitParams.openParams.millisecondsBetweenFrames = 0;
35
36     // This call MUST appear in this function.
37     theApp.InitApp(lw2dInitParams);
38
39     return (initOK);
40 }
```

```

41
42 bool my_game::InitGame()
43 {
44     bool initOK;
45
46     theBall.BitmapTransparentColor(color_rgb (0.0f,1.0f,0.0f));
47     initOK =
48         theBall.LoadImage(
49             "ball.bmp",
50             image_file_base::LWIFF_WINDOWS_BMP);
51
52     if (initOK==true)
53     {
54         theBall.X(1);
55         theBall.Y(1);
56         theBall.Movement(vector(5,3));
57
58         bitmap_region boundingRect;
59         boundingRect.top = 0;
60         boundingRect.bottom = theBall.BitmapHeight();
61         boundingRect.left = 0;
62         boundingRect.right = theBall.BitmapWidth();
63         theBall.BoundingRectangle(boundingRect);
64     }
65     else
66     {
67         ::MessageBox(
68             NULL,
69             theApp.AppError().ErrorMessage().c_str(),
70             NULL,
71             MB_OK | MB_ICONSTOP | MB_SYSTEMMODAL);
72     }
73     return (initOK);
74 }
75
76
77
78 bool my_game::UpdateFrame()
79 {
80     bool updateOK = true;
81
82     vector ballVelocity = theBall.Movement();
83     theBall.Move();
84     if (theBall.X()<=0)
85     {
86         theBall.X(1);
87         ballVelocity.X(ballVelocity.X() * -1);
88     }
89     else if (theBall.X() + theBall.BoundingRectangle().right >=
90             theApp.ScreenWidth())
91     {
92         theBall.X(theApp.ScreenWidth() -
93                 theBall.BoundingRectangle().right - 1);
94         ballVelocity.X(ballVelocity.X() * -1);
95     }
96     else if (theBall.Y()<=0)
97     {
98         theBall.Y(1);
99         ballVelocity.Y(ballVelocity.Y() * -1);
100 }
101 else if (theBall.Y() + theBall.BoundingRectangle().bottom >=

```



```

102     theApp.ScreenHeight())
103 {
104     theBall.Y(theApp.ScreenHeight() -105 theBall.BoundingBox().bottom - 1);
106     ballVelocity.Y(ballVelocity.Y() * -1);
107 }
108 theBall.Movement(ballVelocity);
109
110 return (updateOK);
111 }
112
113
114 bool my_game::RenderFrame()
115 {
116
117     theBall.Render();
118     return true;
119 }

```

Like the previous examples, this program creates a game class. If you look on lines 9 and 11 of [Listing 4.3](#), you'll see that there are two new member functions. The first is called `OnAppLoad()`. The `OnAppLoad()` function is required by the game engine. However, if your game class doesn't have it, the engine provides one.

## Warning

I cannot overemphasize the importance of calling `theApp.InitApp()` in your game class's `OnAppLoad()` function. Your `OnAppLoad()` function must pass a variable of type `init_params` to the `theApp.InitApp()` function. When it does, the application object's `InitApp()` function copies all of the initialization information your `OnAppLoad()` function into the application object so that the application object can use it to set up Windows, OpenGL, and OpenAL properly. Without a call to `theApp.InitApp()`, LlamaWorks2D cannot initialize your game properly.

The code for `OnAppLoad()` begins on line 25. The purpose of this function is to set the initialization parameters for Windows, OpenGL, and OpenAL. The game engine contains reasonable defaults for any initialization values you don't set. LlamaWorks2D calls the `OnAppLoad()` function when it starts the program but before it initializes the program's window, OpenGL, or OpenAL. This enables you to set initialization information before the program gets very far along. On line 33 of [Listing 4.3](#), the `OnAppLoad()` function tells the program to run in full-

screen mode. Most games run in full-screen mode; I recommend you consider using it for your games.

Just before it returns, the `OnAppLoad()` function calls another function. Remember that LlamaWorks2D hides a lot of the hard work of Windows programming in an object that represents the program itself. That object is available to all functions in the program. The name of the object is `theApp`. Anytime your functions need access to the application object, they access it through the name `theApp`. On line 37, you can see that the `OnAppLoad()` function calls one of the application object's member functions, `InitApp()`. Your `OnAppLoad()` function *must* call `InitApp()`. If it doesn't, your program will compile, but it won't run. It just prints an error message saying that it can't initialize Windows and then quits.

After the program initializes itself, it must initialize your game. To do so, it calls your game class's `InitGame()` function. We saw this function in the previous example. However, this version, which starts on line 42, is more complex. The first thing it does is load the ball's bitmap image, which is stored in the file `Ball.bmp`. If `Ball.bmp` is not in the same directory as the game itself, the `InitGame()` function won't be able to load it. As a result, the game won't initialize properly.

The `InitGame()` function sets the transparent color for the ball's bitmap on line 46. It then calls the `sprite` class's `LoadImage()` function to load the bitmap.

If the `InitGame()` function successfully loads the ball's bitmap image, it sets the ball's starting location on the screen. The location is specified in terms of x and y coordinates. The `sprite` class has functions called `X()` and `Y()` that set the values of the sprite's x,y location. `InitGame()` calls them on lines 5455.

Next, the program specifies the direction and speed of the ball's movement. It does this in a nifty shorthand way. The `sprite::Movement()` function requires a `vector` as its only parameter. One way to pass it a `vector` is to create a `vector` variable, as in the following code fragment:

```
vector ballMovement;  
ballMovement.X(5);  
ballMovement.Y(3);  
theBall.Movement(ballMovement);
```

## Warning

Loading bitmaps is probably the slowest task in your game. It

should not be done while the player is in a level. It will slow or stop gameplay momentarily and players don't like that a bit. To solve this problem, load all your bitmaps in the `InitGame()` function.

This style of programming is easy to read, but it's not necessarily the most efficient. Notice that the statements on lines 54-63 of [Listing 4.3](#) only use the `vector` to specify the ball's movement. It's not used for anything else. If the `InitGame()` function created a variable like the code fragment shown here, it would not reuse the variable. It would be more efficient if `InitGame()` could specify the vector without having to create a variable for it. That would save some memory as the program executes.

Rather than create a variable that will be used in only one statement, we can save memory and CPU time by taking advantage of a special feature of C++ called [nameless temporary variables](#). You create a nameless temporary variable by calling a class's constructor. That's what's happening on line 56. The `InitGame()` function creates a nameless `vector` variable by calling the `vector` constructor and passing it the x and y values for the `vector`.

## Tip

It's a good idea to use nameless temporary variables when you're passing information in a parameter list and you won't use the variable anywhere else. This approach makes your code shorter. You may also save memory and CPU time because many compilers recognize nameless temporary variables and perform special optimizations on them. Some of them can actually optimize it right out of existence so that the program never needs to allocate memory for it.

The ball's movement is specified in pixels. Line 56 says that the ball moves 5 pixels to the right and 3 pixels down on every frame of animation.

Next, `InitGame()` sets the ball's bounding rectangle on lines 58-63. The bounding rectangle is the smallest rectangle that will enclose the ball. As you'll see shortly, the game uses the bounding rectangle to test for when the ball hits

the edges of the screen. This technique is called [\*collision detection\*](#). Collision detection is exactly what the name implies; it's the act of detecting when sprites hit something. Every time the ball moves, the game tests to see if it hit anything. Collision detection is an extremely common and necessary task in games. It can become very complex, especially in 3D games. However, 2D games can usually detect collisions between sprites just by testing whether their bounding rectangles overlap. It's a fast, easy test that doesn't require a lot of CPU time. I'll demonstrate it throughout the remaining chapters in this book.

The `InitGame()` function sets the ball's bounding rectangle by creating and initializing a variable of type `bitmap_region`. This is a type that's defined in `LlamaWorks2D`. If you look at the code on lines 5962, it appears that `InitGame()` is accessing a class's public member data. However, the `bitmap_region` type is not a class; it's a structure. Therefore, its members are public by default. Your functions can access them directly. Lines 5962 set the top, bottom, left, and right edges of the sprite's bounding rectangle. These values are distances away from the upper-left corner of the sprite's bitmap image. Setting the top and left both at 0 means that the upper-left corner of the bounding rectangle is at the upper-left corner of the bitmap. Setting the bottom and right edges to the height and width means that the bottom-right corner of the bounding rectangle is at the bottom-right corner of the bitmap. This does not have to be the case, but often, this is exactly how you will set up your sprites.

On line 63, the `InitGame()` function calls the `sprite` class's `BoundingBox()` function and passes it the variable `boundingRect`. This sets the `sprite` object's bounding rectangle.

If the bitmap was not loaded correctly, the `InitGame()` function executes the code on lines 6771. These statements use the standard Windows `MessageBox()` function to display an error message. If you want more information on `MessageBox()`, you can find it on the Web at [www.msdn.microsoft.com](http://www.msdn.microsoft.com).

## Tip

As the statements on lines 6771 show, your game can call Windows functions whenever it needs to. This does not interfere in any way with OpenGL or `LlamaWorks2D`.

Beginning on line 78, [Listing 4.3](#) presents the `UpdateFrame()` function. This is another new function in the `my_game` class. Every game class must have this

function. As you've probably guessed by now, LlamaWorks2D provides one for you if you do not write it yourself. The purpose of the `UpdateFrame()` function is to calculate the new positions for each sprite in the frame. To do that, it first gets the ball's movement vector and stores it in a variable.

The `UpdateFrame()` function next calls the `sprite::Move()` function on line 83. This tells the ball to move based on its movement vector. After the `Move()` function finishes, it is possible that the ball will have moved off the screen or collided with the edge of the screen. The series of chained `if-else` statements beginning on line 84 tests for that. This style of `if-else` statement puts a new `if` onto the end of an `else`. It's kind of tricky, but almost every C++ programmer does it. You'll have to get used to it if you want to work with other C++ programmers. The statement on line 89 says that if the ball didn't hit the right edge of the screen, `UpdateFrame()` checks to see if it hit the left edge. The chained `if-else` statement continues and tests for collisions with the top and bottom of the screen as well.

The statement on line 84 tests the position of the ball's upper-left corner against the left edge of the screen, which is located at an x position of 0. If you look at the statement on line 84, you'll see that it gets the ball's x position by calling the `sprite::X()` function. We used that same function on line 54 to set the x position of the ball.

What's up here? The `sprite` class's `X()` and `Y()` functions set the x and y coordinates in the `InitGame()` function. Now they're getting those values?

The answer is yes. We can have functions do this in C++. Programming this way cuts down on the number of function names you have to remember when you're using LlamaWorks2D. I'll show you how to make functions like this for your own classes in [chapter 5](#), "Function and Operator Overloading".

## Tip

It's very common to add very small fudge factors to the position of objects in games to simplify code. This is a technique that you should not hesitate to use when appropriate.

If the ball has gone off the left edge of the screen, it's time to bounce the ball. The statement on line 86 sets the ball's x position to 1. That shifts the ball over 1 pixel away from the edge of the screen so that the `UpdateFrame()` function

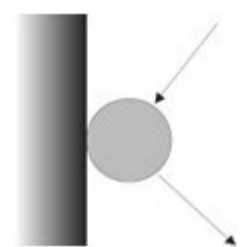
won't bounce the ball again in the next frame of animation. No one will notice that the position of the ball has shifted by 1 pixel. However, shifting the ball over 1 pixel makes the program simpler because it doesn't have to keep track of whether there was a bounce on the previous frame. This technique is a way of adding a fudge factor that keeps your life simple.

## Note

The simple method of bouncing a ball shown here assumes that there is no friction. As a result, the speed of the ball does not change; only its direction changes. If we were doing a more realistic game, we would probably need to make the ball slow down with each bounce.

The statement on line 87 bounces the ball by multiplying its x direction by  $-1$ . Because multiplying by  $-1$  makes positive numbers negative and negative numbers positive, this has the effect of switching the x direction of the ball. This easy technique for bouncing sprites in games is illustrated in [Figure 4.4](#).

### Figure 4.4. The ball's x direction reverses when you multiply it by $-1$ .



[Figure 4.4](#) shows the ball hitting the left edge of the screen after it has been moving down and to the left. Multiplying the ball's x direction by  $-1$  points it in the opposite direction. The y direction remains unchanged. So in the next frame of animation, the ball continues to move down. However, it is now moving to the right.

If the ball has not hit the left edge of the screen, the `UpdateFrame()` function tests to see whether it hit the right edge of the screen on lines 8990. It does so by

adding the width of the bounding rectangle to the ball's x position. That calculates the position of the right edge of the bitmap. If the right edge has gone off the right edge of the screen, the statements on lines 9294 set the ball so that it is 1 pixel away from the right edge of the screen and then bounce the ball.

## Note

If you compile and run this program, you'll need to press Alt+F4 when you want to quit. If you don't, the program keeps running indefinitely.

Lines 96100 test to see whether the ball has hit the top of the screen. If so, it sets the ball's y position to 1 and then bounces the ball. On lines 101107, `UpdateFrame()` performs the test for the bottom of the screen. If the ball has hit the bottom, it moves the ball so that it is 1 pixel above the bottom of the screen and then bounces the ball.

# Getting Good Results

It is possible that the movement of the ball is too slow or too jerky. There are a couple of reasons for this. First, your computer may be old and slow. If that's the case, you'll need to upgrade as soon as you can afford it. A game programmer should have a computer whose microprocessor runs no slower than 500 MHz (500 megahertz). Even that's on the slow side. In reality you should be using a computer that runs no slower than 1 GHz (1 gigahertz, which is 1000 megahertz).

The simple bouncing ball program shown in [Listing 4.3](#) can run slow or jerky even if you have a fast computer. It might be that your graphics card (also called a video adapter) is old and slow. If you bought your graphics card more than 4 years ago, it's time to replace it.

It is possible that the program runs slow on your *very fast* computer with an *excellent* graphics card. That's because your screen might be in a very high resolution video mode. For example, my screen is set to 1280 pixels across by 1024 pixels vertically. That's a lot of pixels.

Having your display set to a high-resolution mode usually means that its refresh rate is set to 60 Hz (hertz). That means that the electron gun in your screen refreshes the screen 60 times every second. This is actually a slow refresh rate. If your screen is set to 60 Hz, it limits the number of frames that OpenGL can display on the screen each second. Usually it means that you're getting fewer than 30 frames of animation per second. To get smooth, quick animation, you need a frame rate of 30 frames per second or higher. Your program does that by setting the screen into a lower resolution mode with a higher refresh rate than 60 Hz. I'll demonstrate how to do that with LlamaWorks2D in [chapter 7](#), "Game Program Structure."

If your game does not specify a video mode and refresh rate, LlamaWorks2D just uses the current mode and refresh rate. This is not generally a good idea because you cannot guarantee that the game looks right and has a proper frame rate on all computers.



# Summary

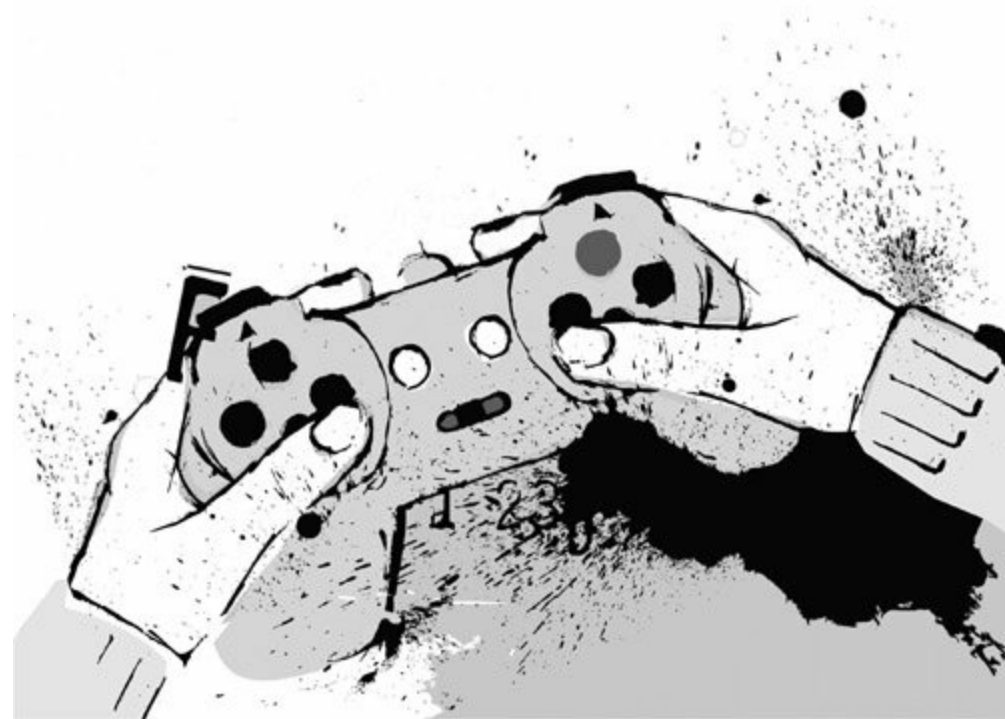
In this chapter, you got an overview of game programming with LlamaWorks2D. The chapter showed that you really don't have to write that much code to get a game up and running. By focusing on creating your game class and leaving the rest up to LlamaWorks2D, you can stick to the programming tasks that are the most fun.

In the next two chapters, you get a closer look at some of the programming techniques demonstrated in this chapter. You'll see how you can use them to make your game code simpler, shorter, and easier to understand.

# Chapter 5. Function and Operator Overloading

Before we go much further in game programming, you must know how to write programs in which multiple C++ functions have the same name. This is an important feature of C++; its formal name is *function and operator overloading*.

Also, every game programmer must know some math. I realize some people may experience physical pain when I say that, but the math needed for games isn't actually very hard. Specifically, we're going to talk about vectors. Vectors are so common that every game programmer has to be able to use them. Before we can move on to talking about how to write even simple games, I have to cover them.



Although overloading and vectors may seem like unrelated topics, they are actually closely connected in game programming. To be specific, when you write a vector class, you must use overloading. Therefore, in this chapter, I explain overloading and then show how to use overloading to write a vector class.

# What Is Overloading?

In early programming languages, such as C, BASIC, FORTRAN, and Pascal, the name of every function in a program had to be unique. No two functions could have the same name. In C++, that's not the case. It's possible, and often desirable, to assign the same name to two or more functions; that's overloading.

There are two basic kinds of overloading. The first is function overloading, which involves having multiple functions with the same name. In addition, you can overload operators, such as `+` and `*`.

Let's first examine function overloading.

## Overloading Functions

C++ allows functions in different classes to have the same name. That is not overloading.

### Tip

When programmers talk about member functions, they often use the scope resolution operator to show which function they mean. If you use that style, the function you're talking about is clear. For example, it's easy to distinguish `snert::Zop()` from `blarg::Zop()`. It's clear that they are two different and unrelated functions in two different classes.

Why? Because the class name differentiates the functions. If two classes, one called `snert` and the other called `blarg`, each have a function named `Zop()`, the compiler differentiates between them by their types. The class that they are a member of tells the compiler which one to use. In fact, the compiler sees them as having nothing to do with each other even though they are both called `Zop()`.

To be overloaded, the functions must be in the same class. Now why in the world would anyone want to have two functions in the same class with the same name?

In [chapter 4](#), "Introducing the LlamaWorks2D Game Engine," you were able to both set and get the x location of a `sprite` object with a function called `X()`. Likewise, you were able to both set and get the y location with a function called `Y()`. This worked because the `X()` and `Y()` functions are overloaded. The `sprite` class contains one `X()` function that sets the x location of the `sprite` object and another that gets the x location. There are also two functions in the `sprite` class called `Y()` that set and get the y location, respectively. Using function overloading enables you to access the values of the private x and y data members using very intuitive function names. This style of programming helps reduce the number of function names you have to remember in order to use objects.

The first `X()` function in the LlamaWorks2D `sprite` class sets the value of the class's private data member `x`. Here's the source code for it:

```
inline void sprite::X(int upLeftX)
{
    x=upLeftX;
}
```

As you can see, this function assigns the value of its parameter to the private data member `x`. Because LlamaWorks2D is a teaching tool as much as a game engine, I've purposely left out some error checking. For instance, this function should ensure that the value of `upLeftX` is on the screen.

The other version of the `sprite::X()` function gets the value of the private data member `x`. Here's its source code:

```
inline int sprite::X()
{
    return (x);
}
```

All this version does is return `x`.

How can we, or the compiler for that matter, tell which overloaded function a program is calling? Like the compiler, we can differentiate between the functions by their return types and parameter lists. For example, the `X()` functions in the `sprite` class both have different return types and a different numbers of parameters. Therefore, if you see code like this:

```
sprite theBall;  
theBall.X(100);
```

you use the fact that this call passes a parameter to `x()` to determine which version of the function is being called. Because this call passes the value 100 to `x()`, it has to be calling the version with one parameter. That version sets the x location of the sprite. It follows that if you see code like this:

```
int ballX = theBall.();
```

you can tell which function is being called based on the return type and parameter list. The first version of `x()` in the `sprite` class requires one parameter and doesn't return a value, so the code can't be calling that one. The second version of `x()`, takes no parameters and returns an integer. That's the one that fits this usage.

Another reason C++ programmers use function overloading is to be able to provide different functions of the same name that all do the same thing in a slightly different way. [Listing 5.1](#) gives an example of this.

## Warning

You cannot have two functions in a class with the same name, parameter list, and return type. If you do, the compiler won't be able to tell them apart. Also, if two functions have the same name and parameter list but different return types, the compiler won't be able to differentiate between them.

## Listing 5.1. A class with overloaded constructors

```
1 class point_2d  
2 {  
3 public:  
4     point_2d();  
5     point_2d(int xComponent, int yComponent);
```

```

6     void X(int xComponent);
7     int X();
8
9
10    void Y(int yComponent);
11    int Y();
12
13    private:
14        int x, y;
15 };
16
17
18    point_2d::point_2d()
19    {
20        x=y=0;
21    }
22
23    point_2d::point_2d(int xComponent, int yComponent)
24    {
25        x=xComponent;
26        y=yComponent;
27    }

```

[Listing 5.1](#) contains a class called `point_2d`, which is a class you might use in a game. Notice that the class has two constructors. Their prototypes are on lines 45. The first takes no parameters and the second requires two parameters. The listing shows the code for these constructors on lines 18-27. The first constructor, which is called the [default constructor](#), initializes the `x` and `y` data members to 0. The second constructor initializes the `x` and `y` data members to the values passed in through the parameter list. These constructors enable programs to declare `point_2d` variables in the following ways:

```

point_2d point1;
point_2d point2(10,20);

```

These two functions do essentially the same thing: they both initialize a new `point_2d` object. However, they do it in slightly different ways. Both ways of initializing this object are intuitive and useful to programmers. Therefore, it's sensible for the class to provide both functions. Function overloading enables it to do exactly that.

## Tip

You can overload any function in a class except the destructor.

# Overloading Operators

C++ enables you to create operators for your classes. Examples of operators are `+`, `-`, `*`, and `/`. It may seem odd to be able to create operators, but programmers find it really handy once they give it a try.

We all know how to use operators with integers. For instance, the addition operator for integers adds two numbers together. However, you can also define an addition operator for other types. One example is the `string` class that is defined in the C++ standard libraries. Suppose you saw the following code in a program:

```
string thisString = "Mary had a ";  
string thatString = "little lamb."  
string theOtherString = thisString + thatString;
```

Looking at this code, most programmers expect the result in `theOtherString` to be "Mary had a little lamb." In this case, the `+` operator is used to combine the contents of two string variables. This is conceptually similar to adding two integer variables. Any time you want to perform an operation on a class that is conceptually similar to addition, you can define a plus operator for that class. The same is true of other operators.

The best way to explain how to overload operators is to demonstrate it. I'll do that by creating a class to represent mathematical vectors in the next section.

# Implementing a Vector Class with Overloading

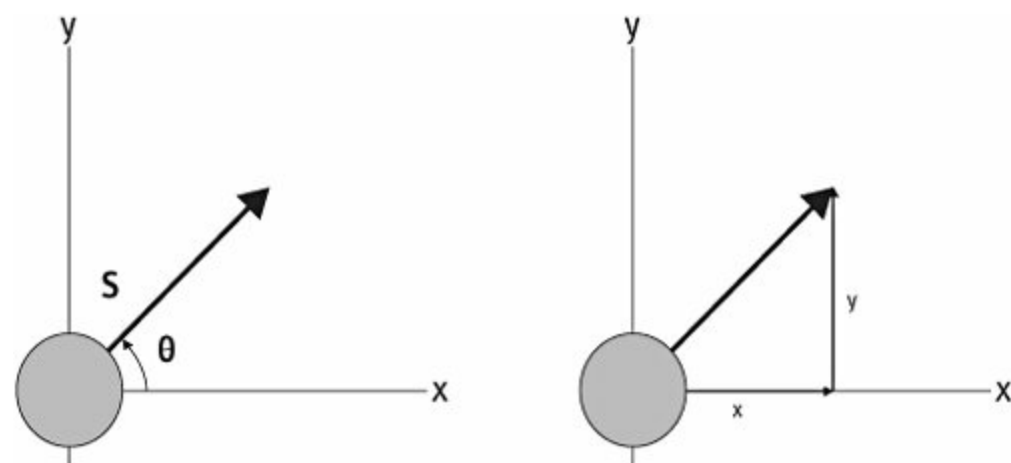
In [chapter 4](#), I briefly introduced vectors and stated that a vector contains a magnitude and a direction. There's really no better way to specify how sprites in games should move than with vectors. In fact, if you ever study the physics topics of dynamics and kinematics, you'll see that physicists specify movement almost completely in vectors.

If a vector has a magnitude and direction, how does that help you in games?

If your game specifies an object's movement as a vector, the vector's magnitude tells how fast the object moves. The direction of the object's movement is specified by the vector's direction. Also, vectors can be applied to any coordinate system. This enables you to define your vectors any way you want. For example, if you live outside the United States, the speedometers of cars in your country probably give the car's speed in kilometers per hour. If you live in the United States, it's miles per hour. Either way, specifying a car's speed with vectors works exactly the same. Whether your game has a coordinate system that uses kph or mph, you can use vectors to describe the motion of its sprites.

You can specify a vector's direction and magnitude in multiple ways. One is to just specify the vector's direction and magnitude. That may sound a bit silly to say, but it's actually not. The reason is that you can also specify a vector's direction and magnitude by breaking the vector into components. [Figure 5.1](#) illustrates what I mean.

**Figure 5.1. Two ways to specify vectors.**





[Figure 5.1](#) shows two views of a ball at the origin of an  $x,y$  coordinate system. The one on the left has a vector that specifies the direction and speed that the ball is moving. The direction is stated in terms of an angle represented by the Greek letter theta ( $\theta$ ). If you're familiar with specifying angles, you probably know that they can be specified either in degrees or radians. In this coordinate system, we'll use degrees and say that  $0^\circ$  is at the positive  $x$ -axis. The angle increases in the counterclockwise direction. The speed of the ball is represented by  $S$ . The speed and the direction specify the magnitude and direction of the ball's motion vector.

The second diagram in [Figure 5.1](#) (on the right) also shows the ball's movement vector. This time, it's specified in terms of its component vectors. This is an extremely easy way to specify vectors in games. You can take the  $x$  and  $y$  components as pixel displacements, as we did in the final program in [chapter 4](#). Specifying an  $x$  component of 5 for the ball's motion vector and a  $y$  component of 10 means that the ball will move 5 pixels in the  $x$  direction and 10 in the  $y$  direction in each frame. This is exactly how most 2D games use vectors.

## Tip

When you write 2D games, I recommend that you specify vectors in terms of  $x$  and  $y$  components rather than direction angles and magnitudes. It's much easier if you do.

We're ready to see overloading in action in the `vector` class. First, let's examine the class definition itself (see [Listing 5.2](#)). After that, we'll see how overloaded functions and operators work.

## Listing 5.2. A `vector` class

```
1 class vector
2 {
3 public:
4     vector();
5     vector(
6         int xComponent,
7         int yComponent);
8
9     void X(
10        int xComponent);
```

```

11 int X(void);
12
13 void Y(
14     int yComponent);
15 int Y(void);
16
17 vector operator +(
18     vector &rightOperand);
19 vector operator -(
20     vector &rightOperand);
21
22 vector operator *(
23     int rightOperand);
24 friend vector operator *(
25     int leftOperand,
26     vector &rightOperand);
27
28 vector operator /(
29     int rightOperand);
30 friend vector operator /(
31     int leftOperand,
32     vector &rightOperand);
33
34 vector &operator +=(
35     vector &rightOperand);
36 vector &operator -=(
37     vector &rightOperand);
38 vector &operator *=(
39     int rightOperand);
40 vector &operator /=(
41     int rightOperand);
42
43 int Dot(
44     vector &rightOperand);
45
46 int Magnitude();
47 int MagnitudeSquared();
48 vector Normalize();
49
50 private:
51     int x,y;
52 };

```

This `vector` class makes heavy use of function and operator overloading. It provides two constructors, two `X()` functions, two `Y()` functions, and two `*` operators. It also has the operators `+`, `-`, and `/`. In addition, it provides some new operators: `+=`, `-=`, `*=`, and `/=`.

## Overloaded Vector Constructors

Like the `point_2d` class in [Listing 5.1](#), the `vector` class in [Listing 5.2](#) contains two overloaded constructors. The default constructor (the one with no parameters)

sets the values of the private data members `x` and `y` to 0. The other, whose prototype is shown on lines 57, sets `x` and `y` to the value of its parameters. Here is the code for both functions:

```
inline vector::vector()
{
    x = y = 0;
}
```

```
inline vector::vector(
    int xComponent,
    int yComponent)
{
    x = xComponent;
    y = yComponent;
}
```

Programs automatically call the default constructor whenever they declare a `vector` variable without initializing it, like this:

```
vector thisVector;
```

## Note

Using overloaded constructors, you can provide any type of initialization that makes sense for the class. This includes an initialization with no parameters, one parameter, two parameters, and so on.

To initialize a `vector` variable, programs must either provide no parameters and accept the default initialization, or provide two parameters. There are no other choices because there are no other constructors. It is possible to add a constructor that has only one parameter. However, for the `vector` class, that approach doesn't make sense.

When a program calls the constructor with two parameters, it uses a function-style declaration, like the following statement:

```
vector thatVector(20,30);
```

## Factoid

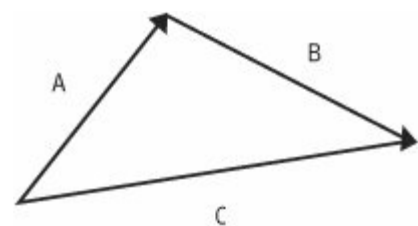
C++ programmers also call parameters "arguments"-so they will often call the default constructor the "no-arg constructor" to indicate that it has no arguments. Similarly, they say that one-parameter constructors are one-arg constructors, two-parameter constructors are two-arg constructors, and so on.

This example is similar to the `point_2d` example earlier in this chapter. The variable name `thatVector` is followed by parentheses as if it were a function name. However, it is not. Instead, this notation tells the program to call the `vector` constructor and pass it two parameters. This causes the program to call the constructor that has two parameters and set the values of `xComponent` and `yComponent` to 20 and 30, respectively. The two-parameter constructor sets the `x` and `y` values of `thatVector` to 20 and 30.

## Addition and Subtraction of Vectors

When you add or subtract vectors, it's very much as if you're laying them end to end. [Figure 5.2](#) illustrates adding vectors.

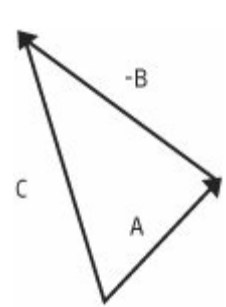
**Figure 5.2. Adding vectors A and B produces vector C.**



To add vector A in [Figure 5.2](#) to vector B, you essentially put the start of B at the end of A. You then use the starting point of A and the ending point of B for the start and end of a new vector, called C, that is the answer to the addition. Whenever you add two vectors, you always get another vector.

Subtracting vectors is the same as adding the negative, as [Figure 5.3](#) demonstrates.

### Figure 5.3. Subtracting vectors A and B produces vector C.



In this figure, I've used vectors A and B from the previous diagram. To subtract A from B, I reversed the direction of B. This turns B into  $-B$ . Now it is easy to lay the two vectors end to end and add them. The result is a new vector, C.

The `vector` class in [Listing 5.2](#) supports both addition and subtraction. It does so by providing its own functions for those operators. The prototypes are on lines 1720. In general, a prototype for overloading any operators follows the pattern that these two use. That is, it first states the return type, which is often the name of the class. Next, it uses the C++ keyword `operator` to indicate that an operator is being overloaded for the class. That is followed by the symbol for the operator itself—in this case, the `+` and `-` signs for the addition and subtraction operator. Finally, the prototype presents the parameter list.

Most of the operators that you'll overload are binary operators. That means they have two operands. Addition is a perfect example. Here's an integer addition:

```
3+5
```

You always add two things together, so you need two operands and an operator. In this case, the 3 and the 5 are the operands and the `+` is the operator. If you're adding vectors, it works the same way. Here's a vector addition:

## Warning

It is possible for you to overload operators in unusual ways. For example, you can overload the + operator to do subtraction and the - operator to do addition. This is not a good idea. You should always overload the operators to do what most people expect.

### **a + b**

The format is just the same as an integer addition. The equation contains two vector operands, **a** and **b**, and the + operator.

When you add two integers in a C++ program, you use a notation that is similar to normal integer addition, like the one shown here:

```
int a = 3, b = 5;  
int c;  
c = a + b;
```

This simply declares two integer variables, **a** and **b**, adds them, and stores the result in the variable **c**. This code looks very normal and understandable to anyone familiar with addition. When we overload the addition operator for any class, such as the `vector` class, we want our code to look essentially the same way. Specifically, we'd like to be able to write our code like the following code fragment:

```
vector a(10,20);  
vector b(30,40);  
vector c;  
c = a + b;
```

This is exactly the way that you can write your code when you overload the + operator.

## Note

Notice that the vectors **a** and **b** are now printed in bold lettering. This is actually the proper way to show vectors. Printing them in bold type indicates that you're adding vectors, not integer or floating-point variables. Of course, this is just the convention used in books. Happily, we do not have to try and bold the variables in our programs. Compilers don't like that sort of thing.

Here's the code for the `+` and operators for the `vector` class:

```
inline vector vector::operator +(
    vector &rightOperand)
{
    vector temp;

    temp.x = x + rightOperand.x;
    temp.y = y + rightOperand.y;

    return (temp);
}
```

```
inline vector vector::operator -(
    vector &rightOperand)
{
    vector temp;
    temp.x = x - rightOperand.x;
    temp.y = y - rightOperand.y;
    return (temp);
}
```

This code fragment calls these two functions:

```
vector v1(10,20), v2(20,30), v3;
v3 = v1 + v2;
v3 = v1 - v2;
```

Bear in mind that when your program calls a class's member function, it must use an object. For instance, calling the `x()` function from the `vector` class would require a

`vector` variable:

```
vector a;  
a.X(10);
```

Here, the variable `a` is used to call `x()`. Calling your overloaded operator functions also requires a class variable. In the previous `vector` example, the object that calls the `+` function is `v1`. The variable `v2` gets passed to the `+` function through the parameter list. If you look back at [Listing 5.2](#), you'll see that the name of the parameter for the `+` operator is `rightOperand`. Now you know why.

The addition function contains the statement

```
temp.x = x + rightOperand.x;
```

Notice that there are three references to the private data member called `x`. The leftmost one is the `x` for the `temp` variable. The rightmost `x` (`rightOperand.x`) is the `x` for the right-hand operand in the addition. The one in the middle is just `x`. It doesn't have an object name in front of it. That's because it's the `x` for the object that was used to call the `+` function. The object that's used to call the addition function is always the left-hand operand. Therefore, in the statement

```
v3 = v1 + v2;
```

the variable `v1` calls the `+` function and `v2` is copied into the parameter `rightOperand`. The function adds the `x` and `y` values of `v2` to the `x` and `y` values of `v1`, stores them in the variable `temp`, and then uses the `return` statement to send the answer back. The answer is then stored in `v3`.

## Factoid

Even if your class only provides one version of an operator function, such as a `+` operator, it is said to be overloading that



operator for the class.

## Multiplication of Vectors

Vectors are different than regular numbers in that there are three ways to multiply vectors. Not only that, but when you do vector multiplication, you have to write four multiplication functions to cover all three methods. That seems odd, but there's a good reason for it, which we'll get to shortly.

The three ways of multiplying vectors are:

Multiplying vectors by scalars

Multiplying a vector by a vector to get the dot product

Multiplying a vector by a vector to get the cross product

We're going to simplify things a bit. In this book, we're only working in two dimensions. You must use 3D to find a cross product, so we'll just ignore that completely. The only kinds of vector multiplication I'll present are those that work in 2D. Therefore, we'll look at multiplying vectors by scalars and finding the dot product. When we multiply vectors and scalars, we actually have to write two functions to get it done.

### Multiplying Vectors by Scalars

When you multiply a vector by a number, you're said to be multiplying it by a *scalar*. A scalar can be either an integer or a floating-point number. The reason we call them scalars becomes clear when you look at the code for the `*` operator for the `vector` class:

```
inline vector vector::operator *(
    int rightOperand)
{
    vector temp;
```

```
temp.x = x * rightOperand;  
temp.y = y * rightOperand;
```

```
return (temp);
```

```
}
```

The form of this `*` operator follows the same pattern that we saw in the `+` operator. Specifically, the return type is the class name and the parameter is the right operand. This is how you'd use the previous function:

```
vector v1, v2(20,30);  
v1 = v2 * 5;
```

When your program multiplies a vector by a scalar, such as the integer 5, it calls the function for the multiplication operator. The `vector` object is the left operand. It is the object that calls the `*` function. The 5 gets passed to the `*` function as the value for the parameter `rightOperand`. When it does, the `*` function multiplies the vector's x and y values by the value in `rightOperand`. What this does is multiply the length of the vector by 5. That's why, when you're speaking of vectors, regular numbers such as integers and floating-point numbers are called scalars. They scale the length of the vector. In this case, it made the vector five times longer. If you want to make it half as long, you multiply it by 0.5:

```
vector v1, v2(20,30);  
v1 = v2 * 0.5;
```

## Multiplying Scalars by Vectors with Friend Functions

Multiplying vectors by scalars should mean that we can multiply scalars by vectors. That is, if we can do this:

```
v1 = v2 * 5;
```

we should also be able to do this:

```
v1 = 5 * v2;
```

The rules of math say that's okay. However, we're also working with C++, and its rules say that member functions must be called by objects. The statement

```
v1 = v2 * 5;
```

calls the `operator*()` function because `v2` is a vector object and it is on the left of the `*` operator. However, if we change that to

```
v1 = 5 * v2;
```

then the left-hand operand is an integer. Integers won't call the `*` function for the `vector` class. In situations like this, it'd be nice to have a friend to help us out. In fact, C++ provides a special type of function called a [friend function](#) to solve problems like this. If you look back at [Listing 5.2](#), you'll see the declaration of a friend function on lines 24-26. Its prototype begins with the keyword `friend`. The function itself is not a member of the class but just a friend. Because it's not a member, it doesn't need to be called with a `vector` object on the left side of the `*` operator. Because it's a friend, it can access all the class's private member data. It's as if the class is an exclusive club that lets in only members-and their friends.

Notice that the friend function needs parameters for both the left and right operands. The left operand is an integer and the right one is a vector. Notice also that there's an ampersand sign (`&`) in front of the parameter `rightOperand`. You don't have to pay attention to that now. I'll explain it shortly.

Here's the code for the friend function:

```
inline vector operator *(
    int leftOperand,
    vector &rightOperand)
{
    return (rightOperand * leftOperand);
}
```

An important point here is that the friend function is not a member of the class. As a result, it does not have `vector::` in front of the word `operator` as the previous `*` function did.

The friend function switches the order of the left and right operands and performs the multiplication. That puts the `vector` object that came into the friend function as the right operator on the left. That's important; it means that the `vector` object now calls the `*` function. Which `*` function does it call? The one we saw in the previous section that is a member of the `vector` class. That function actually does the multiplication by a scalar. When it returns its answer to the friend function, the friend function just returns that answer to the calling program.

With the friend function in place, we can now have statements such as the following in our programs:

```
v1 = 5 * v2;
```

## The Dot Product

Multiplying a vector by a scalar, or scalar by a vector, results in a vector. There is also a way of multiplying two vectors to get a scalar. This process is called the [\*dot product\*](#), so called because you use a dot symbol when you write the equation, as shown here.

$$\mathbf{a} \cdot \mathbf{b}$$

The equation finds the dot product of the vectors **a** and **b**. C++ does not supply a dot symbol. As a result, most programmers simply name their functions that perform the dot product `Dot()`. If you look on lines 4344 of [Listing 5.2](#), you'll see that I've followed this convention in the `vector` class.

When you're performing this kind of vector multiplication, you simply multiply the components of the two vectors involved. The resulting code is quite simple:

```
inline int vector::Dot(
    vector &rightOperand)
{
    return (x*rightOperand.x + y*rightOperand.y);
}
```

## Warning

Some programmers like to overload the \* symbol for their dot product. This is okay if you know your code will always be used by an experienced graphics programmer. However, beginners find it terribly confusing. For that reason, it's best to avoid using the \* symbol for the dot product.

Although the name of the parameter for this function is `rightOperand`, your programs cannot use this function like it does operators. A function call to the `Dot()` function resembles normal function calls:

```
vector v1(10,20), v2(20,30);
int answer;
answer = v1.Dot(v2);
```

As you get into game programming, you'll find that the primary use of the dot product is to calculate the physics of moving objects. Since we're not getting into that much physics, we won't delve into the dot product further. However, its relevance to the current discussion is that it shows the limits of operator overloading. It demonstrates that we cannot overload any symbol we would like. There is a definite set of symbols that can be overloaded, as shown in [Table 5.1](#).

## Note

I've deliberately omitted the following operators from [Table 5.1](#): `->`, `->*`, `()`

These operators can be overloaded, but should not be unless the

circumstances are extremely exceptional.

**Table 5.1. The C++ Operators That Can Be Overloaded**

<b>Operator</b>	<b>type</b>	<b>Description</b>
new	N/A	Allocates dynamic memory.
delete	N/A	Releases dynamic memory.
new[]	N/A	Allocates dynamic array.
delete[]	N/A	Releases a dynamic array.
+	Unary.	Indicates a positive value.
+	Binary.	Addition.
-	Unary.	Makes a value negative.
-	Binary.	Subtraction.
*	Unary.	Dereference.
*	Binary.	Multiplication.
/	Binary.	Division.
%	Binary.	Modulus.
^	Binary.	Bitwise Exclusive Or.
&	Unary.	Reference.
&	Binary.	Bitwise And.
	Binary.	Bitwise Or.
~	Unary.	Bitwise negation.

!	Unary.	Logical Not.
=	Binary.	Assignment.
<	Binary.	Less than.
<=	Binary.	Less than or equal to.
==	Binary.	Equal to.
!=	Binary.	Not equal to.
>=	Binary.	Greater than or equal to.
>	Binary.	Greater than.
&&	Binary.	Logical And.
	Binary.	Logical Or.
+=	Binary.	Add assign.
-=	Binary.	Subtract assign.
*=	Binary.	Multiply assign.
/=	Binary.	Divide assign.
%=	Binary.	Modulus assign.
&=	Binary.	Bitwise And assign.
=	Binary.	Bitwise Or assign.
^=	Binary.	Bitwise Exclusive Or assign.
<<	Binary.	Left shift.
>>	Binary.	Right shift.
<<=	Binary.	Left shift assign.
>>=	Binary.	Right shift assign.

++	Unary.	Increment.
--	Unary.	Decrement.
[]	Unary.	Array item dereference.

---

## Division of Vectors

Vectors can be divided by scalars, but not by other vectors. Therefore, writing an `operator /()` function for the `vector` class is just a matter of writing a function that divides a vector's components by a scalar. If you look in [Listing 5.2](#), you'll see the prototype for the `operator /()` function on lines 2829. Here's the code:

```
inline vector vector::operator /(  
int rightOperand)  
{  
vector temp;  
  
temp.x = x / rightOperand;  
temp.y = y / rightOperand;  
return (temp);  
}
```

In a manner similar to the `operator *()` function, the `operator /()` function creates a temporary variable. It divides the x and y components of the vector by the right operand and stores the results in the temporary variable. Lastly, it returns the contents of the temporary variable to the calling program.

### Factoid

Variables in functions are called temporary or local variables.

When your program calls the `operator /()` function, it uses the style you would



expect for division. For example:

```
vector v1(10,20);  
vector v2;  
v2 = v1 / 10;
```

As with addition and multiplication, the `vector` object on the left of the `/` operator calls the `operator /()` function. That is normally how we would expect division to work. If we were to reverse the order, as in the following code fragment, it looks strange:

```
vector v1(10,20);  
vector v2;  
v2 = 10 / v1;
```

This is not following the normal rules of division. Just as  $10/2$  is not the same thing as  $2/10$ ,  $v1/10$  is not the same thing as  $10/v1$ . Nevertheless, to provide another example of the use of friend functions, I'll implement an operator that allows you to write equations such as  $10/v1$  in your programs. The prototype for this function is on lines 3032 of [Listing 5.3](#).

```
inline vector operator /(  
    int leftOperand,  
    vector &rightOperand)  
{  
    return (rightOperand / leftOperand);  
}
```

You've already seen how to call this function, so I won't provide another example. I want to emphasize that function does not follow the normal rules of division. If I weren't providing another example of friend functions, I would not include it in `LlamaWorks2D`.

## Operator-Equals Operators

C++ provides a useful shorthand version of the basic math operators that can

be applied to many classes, including vectors. To see why these might be useful, suppose for a moment that you want to add a number to a variable and store the result in the same variable. In such a case, you might write code like this:

```
int i = 10;  
i = i + 2;
```

This code declares a variable and initializes it to 10. The expression

```
i + 2
```

gets the value in the variable `i` and adds 2 to it. The assignment operator, which is the equal sign, stores the result back in the variable `i` in the statement

```
i = i + 2;
```

There is a shorter way to write statements like this. C++ provides a `+=` (pronounced "plus-equal") operator that gets the value from a variable, adds a number to it, and stores the result in the same variable. As a result, we can change statements such as

```
i = i + 2;
```

to

```
i += 2;
```

Although such a statement may look odd at first, programmers rapidly come to find the `+=` operator very handy.

In addition, C++ provides similar operators for other operations such as subtraction, multiplication, and division. The general name for this type of operator is the *operator-equals* operators. Some of the most commonly used

operator-equals operators are `-=`, `*=`, and `/=`.

## Note

C++ provides more operator-equals operators than I show here. However, we will not cover them at this time because most of them involve operations we have not yet discussed.

The prototypes for the `+=`, `-=`, `*=`, and `/=` operators for the `vector` class appear on lines 3441 of [Listing 5.2](#). We've already seen what it means to add and subtract vectors, so you can probably imagine how operators such as `+=` might be used for a `vector` class. Here's an example:

```
vector v1(10,20), v2(20,30);  
v1 += v2;
```

This statement is the same as writing

```
v1 = v1 + v2;
```

It results in vector variable `v1` being added to vector variable `v2`. The result is stored in `v1`.

Because the implementations of these four functions contains a fair bit of code, I'll show them in [listing 5.3](#) rather than just displaying them in among the paragraphs of text as I have been doing with the other functions in the `vector` class.

## Listing 5.3. The operator-equals functions for the `vector` class

```
1  inline vector & vector::operator +=(  
2  vector &rightOperand)  
3  {  
4      x += rightOperand.x;  
5      y += rightOperand.y;  
6      return (*this);
```

```

7 }
8
9 inline vector & vector::operator -=(
10     vector &rightOperand)
11 {
12     x -= rightOperand.x;
13     y -= rightOperand.y;
14     return (*this);
15 }
16
17 inline vector & vector::operator *=(
18     int rightOperand)
19 {
20     x *= rightOperand;
21     y *= rightOperand;
22     return (*this);
23 }
24
25 inline vector & vector::operator /=(
26     int rightOperand)
27 {
28     x /= rightOperand;
29     y /= rightOperand;
30     return (*this);
31 }

```

The operator-equals operator functions in [Listing 5.3](#) each use the operator-equals operators for the vector components. For instance, the operator `+=()` function on lines 17 uses the `+=` operator for integers to add the x component of the right operand to the x component of the left operand and store the result in the left operand. It does the same on line 5 for the y components.

These four operator functions work differently than any we've discussed so far. First, they do not declare temporary variables. Second, they return something called `*this`. Third, the return type is `vector &`, not just `vector`. It's natural to wonder what all this means.

To understand what's happening here, let's first look at what happens when your functions return values. Normally, the value a function returns is automatically copied into a special area of memory called the [program stack](#). The program uses its stack to store values that functions return. We can see how the stack works by looking back at the `vector::X()` function whose prototype is on line 11 of [Listing 5.2](#) I'll repeat its code here for convenience:

```

inline int vector::X(void)
{
    return (x);
}

```

This function does nothing but return a value. Suppose a program contains statements like the following:

```
vector v1(10,20);  
int thisInt = v1.X();
```

When the program executes the statement

```
int thisInt = v1.X();
```

it calls the `x()` function. The `x()` function in turn returns the value of `v1`'s private data member `x`. When it does, the program automatically copies the value being returned onto the stack. It then jumps back to the point at which the `x()` function was called. Of course, that point was the statement

```
int thisInt = v1.X();
```

The value being returned is still on the program's stack. The assignment operator (`=`) causes that value to be copied into the variable `thisInt`. The value on the stack is then thrown away, but that's okay because the program copied it into a variable. That's how things *normally* work when your function returns a value.

It is possible for function to bypass the normal mechanisms of returning a value which is what's happening in [Listing 5.3](#) with the operator-equals functions. Whenever the return type of a function is followed by an ampersand (`&`), it means that the function returns a [reference](#) instead of a value. When a function returns a reference, it returns an actual variable rather than a copy of the variable.

What does this all mean?

Look once more at how programs use the `+=` operator. If your program contains the statements

```
vector v1(10,20), v2(20,30);
```

```
v1+=v2;
```

then it is changing the contents of `v1`, and the results of the addition are stored there. If you write the `operator +=()` function for the `vector` class like this:

```
inline vector vector::operator +=(  
    vector &rightOperand)  
{  
    vector temp;  
    temp.x = x + rightOperand.x;  
    temp.y = y + rightOperand.y;  
    return (temp);  
}
```

This function has a problem. Do you see what it is?

In this example, the `operator +=()` function declares a variable called `temp`, which stores the results of the addition. The function then returns that variable. But the whole idea behind the `+=` operator is to change the variable that called the function.

This version of the `operator +=()` function does not do that. At no time does it change the `x` or `y` value of the left operand.

"Well," you might say, "that's easy to fix. Just write it like you had it before, but return a `vector` instead of a `vector &`." If we did that, the code would look like this:

```
inline vector vector::operator +=(  
    vector &rightOperand)  
{  
    x += rightOperand.x;  
    y += rightOperand.y;  
}
```

What happened to the `return` statement? It's gone. When we get rid of the temporary variable `temp`, we no longer have anything to return. "That's okay," you might say. "I don't want to return anything anyway." You may not want

to, but you need to. According to the normal rules of C++, all operator-equals functions must return a value. This enables you to write statements such as

```
vector v1(10,20), v2(20,30), v3;  
v3 += v2 += v1;
```

C++ does not force you to return a value from an `operator +=()` function. However, if you don't, other programmers will have problems with your code. They expect all `operator +=()` functions to return a value.

To solve this problem, the `operator +=()` function must return the left operand. The way it does that is with the statement

```
return (*this);
```

This statement is a way of saying, "Return this object." And in all cases, "this object" is the one that called the function.

If a function contains the statement

```
return (*this);
```

it cannot return a copy of an object. It must return a reference to an object instead. That's why the operator-equals functions in [Listing 5.3](#) all return references. They have to bypass the normal return mechanisms and return a reference rather than a value because they all end with the statement

```
return (*this);
```

By using a reference for the return type, and by returning `*this`, your operator-equals functions all work properly. Your functions can change the contents of the left operand. They let you write statements such as

```
v3 += v2 += v1;
```

in your programs. This is how operator-equals functions are supposed to work.

## Magnitude

Finding the length, or magnitude, of a vector does not have anything to do with overloading. However, I thought I'd cover it here because it's part of the `vector` class.

It is not unusual in games to need to get the length of a vector. For instance, you might want to compare the lengths of two vectors that represent the speed of sprites. That way, you could see which is moving faster.

The `vector` class stores the length of the vector's components, not the length of the vector itself. It has to calculate that. The formula for calculating a vector's length is given here.

$$\text{length} = \sqrt{x^2 + y^2}$$

To find the length of a vector from its components, you first square the components. Next, you add the squares together and find the square root of the result.

The `vector` class contains a function called `Magnitude()` that uses the formula above to calculate the length of the vector from its components. Here's the code:

```
inline int vector::Magnitude()
{
    return (::sqrt(x*x + y*y));
}
```

Using the formula, this function squares the vector's x and y components. It does so by multiplying the private data members `x*x` and `y*y`. It then adds the results. Next, the `Magnitude()` function calls the C++ Standard Library `sqrt()` function to find the square root. Whenever your program uses the `sqrt()` function, it must include the file `math.h`. So you should put the statement

```
#include <math.h>
```

at the beginning of your program's file.



## Note

Putting the scope resolution operator (::) with no class name in front of a function call, as shown in the `Magnitude()` function, tells the compiler that the function (in this case `sqrt()`) is a member of the global namespace. This enables it to be accessed from anywhere in your program. In general, most things in the C++ Standard Library are either in the global namespace or the `std` namespace.

Finding a square root is a high-overhead operation. If you can avoid it, you should. This should not cause you a problem in your game. One of the main reasons to find the length of a vector is to compare it to the length of another vector or compare it to 0. In either case, you can use the square of the length instead.

Suppose the variable `length1` contains the length of one vector and `length2` contains the length of another. Imagine that `length1` is greater than `length2`. If that's the case, then it's also true that `length12` is greater than `length22`. Therefore, we don't need to compare the length of the vectors; we can compare the length of the squares of the vectors instead. The `vector` class enables you to do that by providing a function called `MagnitudeSquared()`. The `MagnitudeSquared()` function returns the length of a vector squared. Here's the code:

```
inline int vector::MagnitudeSquared()
{
    return (x*x + y*y);
}
```

This is essentially the same as the `Magnitude()` function, except that it doesn't calculate the square root. Therefore, this function is much faster than `Magnitude()`.

You should use it in your games instead of `Magnitude()` whenever you can.

## Unit Vectors

A *unit vector* is a vector that has a length of 1. The `vector` class provides a

function called `Normalize()`. This function calculates the unit vector for any vector variable. The resulting unit vector points in the same direction as the vector variable. Here's the code for `Normalize()`:

```
inline vector vector::Normalize()
{
    vector temp;
    int length = Magnitude();
    if (length != 0)
    {
        temp.x = x/length;
        temp.y = y/length;
    }
    return (temp);
}
```

After declaring a local variable, this function tests to see if the length is not equal to 0. If it's not, the function divides the x and y components of the current `vector` object by the length and stores the result in `temp`. It then returns `temp`.

# Summary

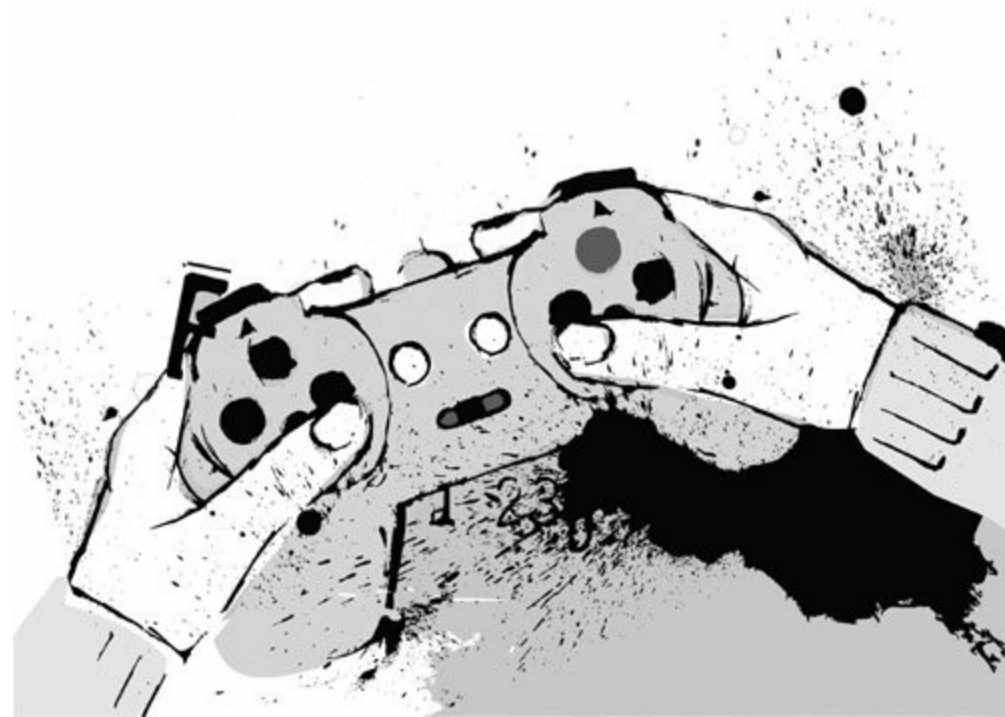
Most professional game engines and code libraries make heavy use of overloading. A good example is LlamaWorks2D. As you continue reading this book, you'll see that almost every class in the LlamaWorks2D game engine has overloaded constructors, member functions, or operators. Therefore, you'll see many examples of everything you've learned in this chapter.

## **Warning**

Never let your program divide by 0. The value 0 should never be in the right operand. If you let this mistake occur, it will crash your program.

# Chapter 6. Inheritance: Getting a Lot for a Little

When I was just starting out in the computer industry, I learned a very important concept from a wise and experienced college professor. He told me that successful programmers are selectively lazy. When I became a college professor, I taught the same concept a bit differently; I used to teach my students how to be successfully lazy. Indeed, laziness in programming, when properly used, can be a virtue.



How can laziness help programmers? If you ever decide to take a job as a game programmer, you'll find that there are two kinds of programmers. The first is incredibly hard working. When given the assignment to write a new game or a part of a game, he immediately sits down at his computer and starts coding. He works long, long hours and is extremely dedicated. Unfortunately, he doesn't have much of a life.

On the other hand, a successfully lazy (and smart) programmer approaches a new assignment a bit differently. She looks around for code she can reuse or repurpose. She searches for code on the Internet, in books and magazines, and in commercial code libraries. As she does, she usually finds that she can reuse or repurpose code for about 50%–80% of her task. This saves her a tremendous amount of effort and enables her to finish much more quickly than the hard-working (but not quite so smart) programmer. She has then time to concentrate her creativity on the most important parts of the game so that she can make a more innovative and unique game.

In programming, reusing code, or being successfully lazy, is a subject called inheritance, which we'll discuss in this chapter. The most important reason for studying and using inheritance can be summed up with the old saying, "Work smarter, not harder." Inheritance helps us do just that.

# What Is Inheritance?

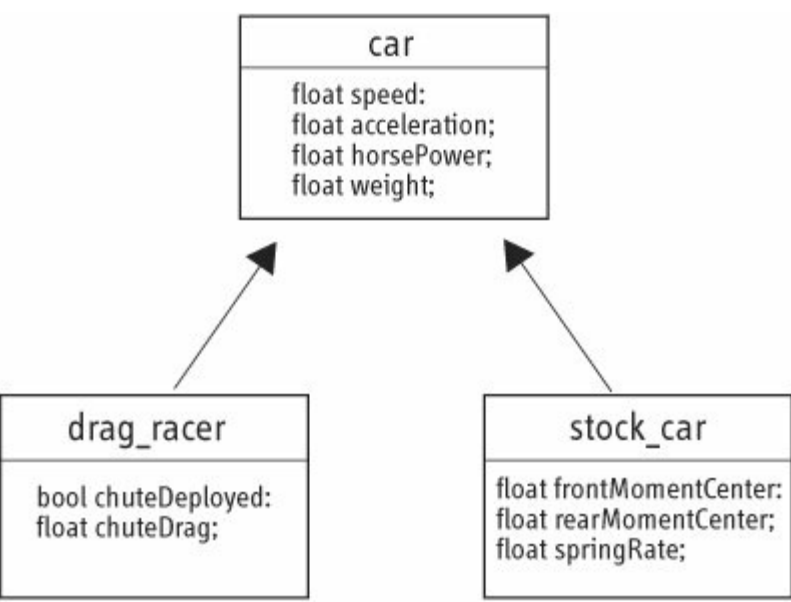
In object-oriented programming languages, inheritance is a way of enhancing existing classes or altering their functionality without rewriting them. Imagine you're writing a racing game. It would be sensible in that situation to create a class called `car`. The `car` class would contain such information as the car's current speed, weight, horsepower, and so forth.

The `car` class is generic. Ideally, you would like to use it for drag racers, stock cars, and so on. Inheritance lets you do that. You can use the `car` class to create such classes as `drag_racer` and `stock_car`. The `drag_racer` and `stock_car` classes have all of the functionality of the `car` class. They also add functionality that is specific to their own types of racecars. For example, drag racers have parachutes, but generic and stock cars don't. Therefore, the `drag_racer` class can add a data member that specifies the drag produced by the parachute. It should also have a data member that is `true` if the parachute is deployed and `false` if not.

Inheritance enables you to write a generic car class containing the basic functionality of all cars. You can use it to implement the `drag_racer` class, and the `drag_racer` class adds more features. In addition, you can use the same `car` class as the basis for implementing the `stock_car` class. This saves you effort. In other words, you get to be successfully lazy.

Game programmers often draw inheritance diagrams to help them figure out the inheritance of all the objects in their games. [Figure 6.1](#) shows an example of an inheritance diagram.

## **Figure 6.1. Inheritance in the racing game.**



As this diagram shows, both the `drag_racer` and `stock_car` classes inherit from the `car` class. You can tell this by the fact that the arrows point from `drag_racer` and `stock_car` to `car`.

When classes inherit from another class, the class they inherit from is called the *base class* or *parent class*. The classes that inherit from the base class are called the *derived classes* or *child classes*. In [Figure 6.1](#), the `car` class is the base or parent class and the `drag_racer` and `stock_car` classes are the derived or child classes.

It's important to note that when child classes derive from parent classes, the children inherit all of the member data and functions of the parent. For instance, [Figure 6.1](#) shows that the `drag_racer` class contains two items of member data that both relate to the drag racer's parachute. However, because `drag_racer` inherits from `car`, it essentially has a `car` object built into it automatically. As a result, `drag_racer` inherits all of the member data you see in `car`. [Figure 6.2](#) illustrates this.

**Figure 6.2. Child classes actually contain a parent class object.**

drag\_racer

car

float speed;  
float acceleration;  
float horsePower;  
float weight;

bool chuteDeployed;  
float chuteDrag;

stock\_car

car

float speed;  
float acceleration;  
float horsePower;  
float weight;

float frontMomentCenter;  
float rearMomentCenter;  
float springRate;

As the figure shows, the derived `drag_racer` and `stock_car` objects contain a copy of a `car` object. The embedded `car` object is invisible and unnamed. Its functions are part of the child object. In this way, the child objects inherit all of the member data and member functions of the parent.



# Deriving Classes

In games, it's not uncommon to define classes that represent points in 2D spaces. Programmers also do the same for points in 3D spaces. However, points in 2D spaces have a lot in common with points in 3D spaces. For instance, they both have x and y coordinates. They both need functions to get and set the x and y coordinates. This suggests that code from the class that represents the 2D point can be reused in the class that represents the 3D point. [Listing 6.1](#) demonstrates how to use inheritance to define 2D and 3D point classes.

## Listing 6.1. Deriving the **point3d** class from the **point2d** class

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 class point2d
7 {
8 public:
9     point2d();
10    point2d(int xValue,int yValue);
11
12    void X(int xValue);
13    int X(void);
14    void Y(int yValue);
15    int Y(void);
16
17 private:
18    int x,y;
19 };
20
21
22 inline point2d::point2d()
23 {
24     x = y = 0;
25 }
26
27
28 inline point2d::point2d(int xValue,int yValue)
29 {
30     x = xValue;
31     y = yValue;
32 }
33
34 inline void point2d::X(int xValue)
35 {
36     x = xValue;
37 }
38
39 inline int point2d::X(void)
```

```
40 {
41     return (x);
42 }
43
44 inline void point2d::Y(int yValue)
45 {
46     y = yValue;
47 }
48
49 inline int point2d::Y(void)
50 {
51     return (y);
52 }
53
54
55 class point3d : public point2d
56 {
57 public:
58     point3d();
59     point3d(int xValue,int yValue,int zValue);
60
61     void Z(int zValue);
62     int Z(void);
63 private:
64     int z;
65 };
66
67 inline point3d::point3d() : point2d()
68 {
69     z = 0;
70 }
71
72
73 inline point3d::point3d(
74     int xValue,
75     int yValue,
76     int zValue) : point2d(xValue,yValue)
77 {
78     z = zValue;
79 }
80
81 inline void point3d::Z(int zValue)
82 {
83     z = zValue;
84 }
85
86 inline int point3d::Z(void)
87 {
88     return (z);
89 }
90
91
92 int main(int argc, char *argv[])
93 {
94     point3d rightHere(40,50,60);
95
96     cout << "(x, y, z)=" << rightHere.X();
97     cout << "," << rightHere.Y();
98     cout << "," << rightHere.Z() << ")";
99     cout << endl;
100
```

```

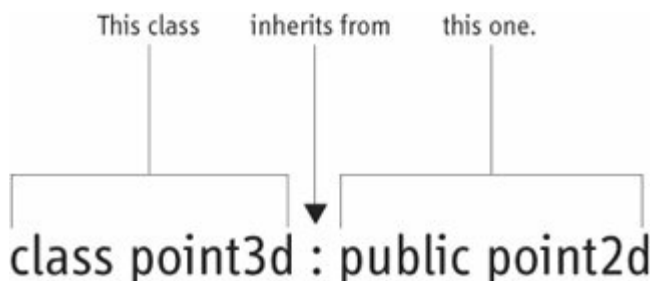
101 rightHere.X(10);
102 rightHere.Y(20);
103 rightHere.Z(30);
104
105 cout << "(x, y, z)=" << rightHere.X();
106 cout << "," << rightHere.Y();
107 cout << "," << rightHere.Z() << " ";
108 cout << endl;
109
110 rightHere.X(30);
111 rightHere.Y(20);
112 rightHere.Z(10);
113
114 cout << "(x, y, z)=" << rightHere.X();
115 cout << "," << rightHere.Y();
116 cout << "," << rightHere.Z() << " ";
117 cout << endl;
118
119 system("PAUSE");
120 return (EXIT_SUCCESS);
121 }

```

[Listing 6.1](#) declares a class called `point2d` on lines 619 that represents a point in a 2D Cartesian coordinate system. As you might expect, it has two constructors as well as member functions, for setting and getting the x and y values of the point.

Line 55 shows the definition of a class called `point3d`, which represents a point in a 3D Cartesian coordinate system. It looks very much like the definition of any other class. However, the name of the `point3d` class is followed by a colon. After the colon is the keyword `public`, which is followed by the name `point2d`. The colon indicates that the `point3d` class inherits from another class. The keyword `public` and the name `point2d` specify that the `point3d` class inherits from the `point2d` class, as illustrated in [Figure 6.3](#).

**Figure 6.3. The `point3d` class inherits from `point2d`.**



Lines 5862 show that the `point3d` class has two constructors and a pair of functions for setting and getting the z coordinate of the point in 3D space. Line 64 defines a private data member called `z`. That is the only member data that the `point3d` class needs.

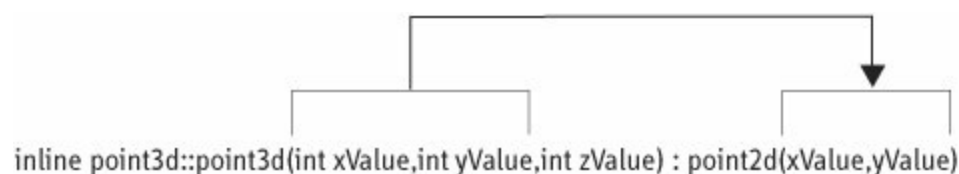
## Note

Recall that C++ programmers often refer to parameters as arguments. So a 3-arg constructor is a constructor with three arguments or parameters.

If you skip down to line 94 of [Listing 6.1](#), you'll see that the program declares a `point3d` variable called `rightHere`. When the program executes this statement, it automatically calls the 3-arg constructor for the `point3d` class. The code for the 3-arg constructor is given on lines 7379. The constructor's three parameters contain the x, y, and z values passed in from line 94.

When the statement on line 94 declares the `point3d` variable, the program automatically creates an invisible `point2d` variable inside it. That invisible `point2d` variable must be initialized whenever the `point3d` constructor is called. That's why that the last parameter on line 76 is followed by a colon and then a call to the `point2d` constructor. As [Figure 6.4](#) shows, writing the `point3d` constructor using this syntax invokes the 2-arg `point2d` constructor and passes it the parameters `xValue` and `yValue`.

**Figure 6.4. Calling the base class constructor from the derived class.**



The 2-arg constructor for the `point2d` class is on lines 2832. It sets the private data members `x` and `y` to the values it receives through its parameter list. When the 2-arg constructor for the `point2d` class ends, the program jumps back to the body of the 3-arg constructor for the `point3d` class beginning on line 77.

Because the constructor for the `point2d` class was called on line 76, you can be assured that the invisible `point2d` object is in a known state before the body of the `point3d` constructor executes. The only remaining task is to initialize the member data declared in the `point3d` class. That's exactly what happens on line 78.

Program execution now jumps back to line 94 and continues on line 96. At this point, the program calls the `X()` function using the `point3d` variable `rightHere`. However, if you look at the definition of the `point3d` class, you'll see that there is no function called `X()`. That is not a problem because `point3d` inherits the `X()` function from `point2d`. Your programs can call functions in base classes using variables of derived types as this program does on lines 96 and 97 when it calls the `X()` and `Y()` functions using the `point3d` variable.

The program in [Listing 6.1](#) demonstrates that the `point3d` class inherits all of the member data and functions of the `point2d` class. After you write the `point2d` class, you don't have to rewrite any of its functionality for the `point3d` class. It got that automatically through inheritance, which saved you a lot of work.

# Protected Members

C++ has a way for base classes to provide derived classes with access to some of their private member data or functions. By changing private member data or functions to protected, all derived classes can access them. [Listing 6.2](#) gives an example.

## Listing 6.2. Protected member data

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 class base
7 {
8     protected:
9         int anInt;
10 };
11
12 class derived : public base
13 {
14     public:
15         void AFunction(int theInt);
16 };
17
18 inline void derived::AFunction(int theInt)
19 {
20     anInt = theInt;
21
22     cout << anInt << endl;
23 }
24
25 int main(int argc, char *argv[])
26 {
27     derived temp;
28
29     temp.AFunction(5);
30
31     system("PAUSE");
32     return EXIT_SUCCESS;
33 }
```

This short program defines two classes. One is called `base` and the other is named `derived`. On line 8, access to the single item of member data in `base` is specified using the keyword `protected` rather than `private`. If `anInt` was specified as `private` rather than `protected`, it could be accessed only by functions that are members of the `base` class. Because this example uses `protected` instead, the member function in the

derived class can directly access `anInt`. This is demonstrated on lines 2022.

It is important to note that only functions in classes derived from `base` can access `anInt` directly. Other functions do not have access. It would be a mistake, for example, to put the statement

```
temp.anInt = 5;
```

in the function `main()`. The `main()` function is not a member of a class derived from `base`. Therefore, it does not have access to protected items in `base`.

In general, I recommend that you avoid the use of protected member data. It can easily be used to bypass the normal access mechanisms you build into your classes. In other words, when you provide a protected data member in a base class, a derived class can contain a function that sets that data member to an invalid value. It is my experience that this actually happens all too often. One programmer writes a class that contains a protected data member and another uses the class in unexpected ways by setting the protected data member to an unusual value.

If you find that you need to provide derived classes with access to a base class's private data, there's a better approach, as demonstrated in [Listing 6.3](#). Instead of providing protected member data, provide protected member functions to get and set the data.

### Listing 6.3. A better style of protected access

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 class base
7 {
8 protected:
9     void AnInt(int intValue);
10    int AnInt();
11
12 private:
13    int anInt;
14 };
15
16
17 inline void base::AnInt(int intValue)
18 {
```

```

19     if (intValue >= -10)
20     {
21         anInt = intValue;
22     }
23     else
24     {
25         anInt = -10;
26     }
27 }
28
29 inline int base::AnInt()
30 {
31     return (anInt);
32 }
33
34
35 class derived : public base
36 {
37     public:
38     void AFunction(int theInt);
39 };
40
41 inline void derived::AFunction(int theInt)
42 {
43     AnInt(theInt);
44
45     cout << AnInt() << endl;
46 }
47
48 int main(int argc, char *argv[])
49 {
50     derived temp;
51
52     temp.AFunction(5);
53
54     system("PAUSE");
55     return EXIT_SUCCESS;
56 }

```

In this version of the program, the member data is private rather than protected. Also, the base class adds two protected functions. These functions set and get the value of `anInt`, respectively. If you look at `derived::AFunction()` on lines 41-46, you can see that it must access the information in `anInt` through the protected member functions. It no longer has direct access to `anInt`. Classes that are derived from `base` can use the overloaded `AnInt()` functions to set or get the value of `anInt`. Therefore, `derived::AFunction()` can do everything it could do in the previous version. However, because all access to the private data member `anInt` must occur through the protected `AnInt()` functions, `AnInt()` can use an `if-else` statement to prevent `anInt` from being set to an invalid value, as shown in lines 19-26.

## Tip



I strongly recommend that, rather than create classes with protected member data, you provide protected member functions to get and set the data. This approach helps ensure that member data in the base class can never be set to invalid values.

# Overriding Base Class Functions

In [chapter 5](#), "Function and Operator Overloading," you saw how C++ enables you to override constructors, functions, and operators. It also lets you do the same thing between base and derived classes. You can override base class functions in derived classes. You do this when you want to enhance or change the way the base class function operates. [Listing 6.4](#) demonstrates how to override base class functions in a derived class.

## Note

The entire program for [Listing 6.4](#) is on the CD in the folder \Source\Chapter06\Prog\_06\_04. It's in a file called Prog\_06\_04.cpp.

## Listing 6.4. Overriding the **Print()** function in the **point2d** class

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 class point2d
7 {
8 public:
9     point2d();
10    point2d(int xValue, int yValue);
11
12    void X(int xValue);
13    int X(void);
14    void Y(int yValue);
15    int Y(void);
16
17    void Print();
18
19 private:
20    int x, y;
21 };
22
23
24
25 inline void point2d::Print()
26 {
27     cout << x << ", " << y;
28 }
```

```

29
30
31 class point3d : public point2d
32 {
33 public:
34     point3d();
35     point3d(int xValue, int yValue, int zValue);
36
37     void Z(int zValue);
38     int Z(void);
39
40     void Print();
41
42 private:
43     int z;
44 };
45
46
47 void point3d::Print()
48 {
49     point2d::Print();
50     cout << "," << z;
51 }
52
53 int main(int argc, char *argv[])
54 {
55     point3d rightHere(40,50,60);
56
57     cout << "(x, y, z)=";
58     rightHere.Print();
59     cout << ")";
60     cout << endl;
61
62     rightHere.X(10);
63     rightHere.Y(20);
64     rightHere.Z(30);
65
66     cout << "(x, y, z)=";
67     rightHere.Print();
68     cout << ")";
69     cout << endl;
70
71     rightHere.X(30);
72     rightHere.Y(20);
73     rightHere.Z(10);
74
75     cout << "(x, y, z)=";
76     rightHere.Print();
77     cout << ")";
78     cout << endl;
79
80     system("PAUSE");
81     return (EXIT_SUCCESS);
82 }

```

[Listing 6.4](#) gives a modified version of the program from [Listing 6.1](#). It doesn't actually present the entire program. To save some space and focus on the

discussion at hand, I've omitted all of the member functions you saw in [Listing 6.1](#). The only member functions that appear in [Listing 6.4](#) are two functions that are both named `Print()`. The `point2d` and `point3d` classes each contain their own version of `Print()`.

The version of `Print()` in the `point2d` class prints the value in the private data member `x`, then prints a comma, and finally prints the value in the private data member `y`. The version of `Print()` in the `point3d` class begins on line 47. It calls the `Print()` function in the `point2d` class. The style of function call you see on line 49 is how functions in derived classes call the functions in base classes that they override. The call must specify the name of the base class, followed by the scope resolution operator, and then the name of the function.

## Note

Functions in derived classes that override base class functions do not *have* to call the base class functions they override. They often do, but it is not required.

As a result of calling the `Print()` function in the `point2d` class, the `x` and `y` values for the point are printed to the screen. After the `Print()` in the `point3d` class calls the `Print()` function in the `point2d` class, it prints a comma. It then prints the value in the private data member `z`. You can see on line 58 that the `main()` function calls the `Print()` function for the `point3d` class. Looking at just that one line of code, you cannot tell (nor should you care) that the `Print()` function in the `point3d` class invokes the `Print()` function in the `point2d` class. The fact that it gets its work done properly is all that counts. However, when you're writing the `point3d` class, it saves you time and effort to utilize the `Print()` function from the `point2d` class. Again, that's precisely the purpose of inheritance.

# Customizing Your Game with Inheritance

The main reason I delved into the topic of inheritance was so that we could use it in games. If you're wondering how inheritance might be used in your games, the answer is that you already know how. The class in [Listing 6.5](#) repeats the game class you saw in [Listing 4.3](#) in [chapter 4](#), "Introducing the LlamaWorks2D Game Engine." At the time I originally presented this game class, I didn't explain what the statement

`: public game`

was at the end of the first line of the class. Now you know.

## Listing 6.5. A game class that uses inheritance

```
1 class my_game : public game
2 {
3 public:
4     bool OnAppLoad();
5     bool InitGame();
6     bool UpdateFrame();
7     bool RenderFrame();
8
9     ATTACH_MESSAGE_MAP;
10
11 private:
12     sprite theBall;
13 };
```

Every game you write with LlamaWorks2D requires that you create your own game class. Your game class inherits from a class I've provided called `game`. If your game class doesn't inherit from `game`, LlamaWorks2D simply won't recognize it as a game class.

If you glance back at [chapter 4](#), you'll see that I indicated that if you don't provide the functions whose prototypes are shown on lines 47 of [Listing 6.5](#), the LlamaWorks2D provides them for you. Those functions are in the LlamaWorks2D `game` class. When you write the functions such as `OnAppLoad()` or `InitGame()` in your derived class, you're overriding the functions in the base class.

If you don't override the functions in the base class, your game class inherits these functions from the LlamaWorks2D `game` class. Inheritance is one of the main ways that LlamaWorks2D provides you with reasonable default implementations of tasks that need to be done in games.

# Summary

This chapter demonstrated that inheritance is a valuable tool for game programmers. It saves you a huge amount of work as you write games by enabling you to reuse your code. In addition, inheritance lets you customize code that someone else provides as is the case with LlamaWorks2D. With inheritance, you save yourself a lot of effort by extending the game class I provide rather than writing one from scratch. You're being "successfully lazy" by "working smarter, not harder."

As you'll see in the next chapter, LlamaWorks2D makes heavy use of inheritance. In fact, without an understanding of inheritance, you really can't use LlamaWorks2D effectively. But now that you've seen inheritance in action, you're ready to start using LlamaWorks2D to write games. That's the subject of [chapter 7](#).

# Part 3: The Essentials of Game Development

[Chapter 7. Program Structure](#)

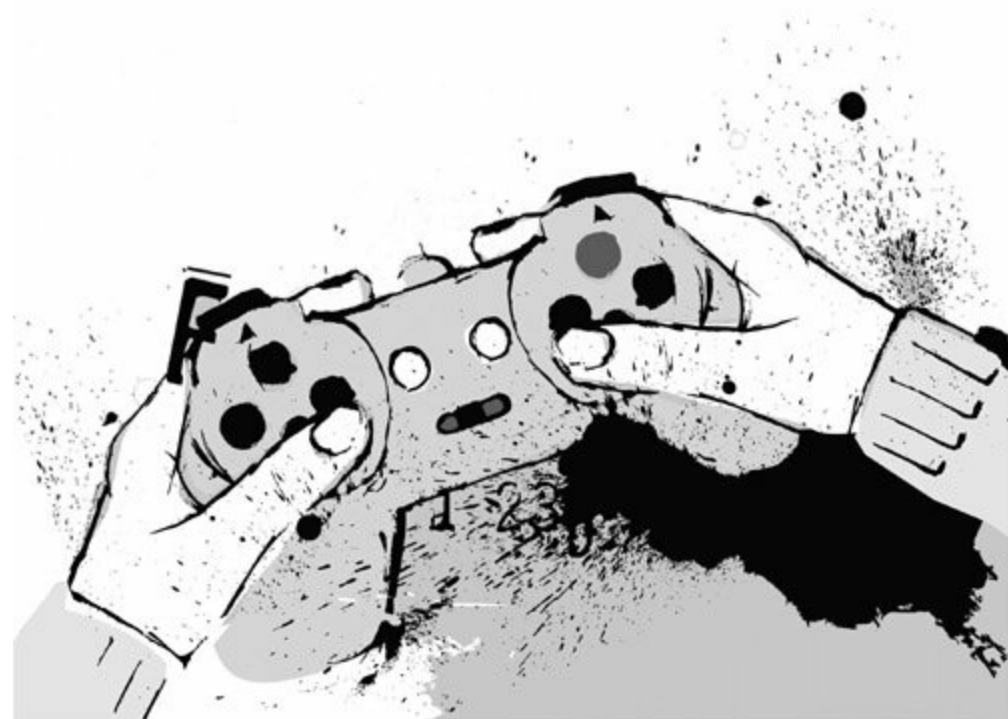
[Chapter 8. Sound Effects and Music](#)



# Chapter 7. Program Structure

You've made it through six chapters so far, and you've learned a lot. This chapter provides a payoff for your persistence. It shows how to pull together everything you've learned to design and implement an actual game.

In my experience, one of the worst things game programmers can do when they're writing a game is to just jump in and start writing program code. That's a recipe for disaster (or at least a big mess). The typical result is that a lot of code has to be rewritten because it doesn't quite do what's needed.



Instead, programmers should take some time to design the game program. This is different than designing the game. Designing the game includes tasks such as figuring out where to hide the Singing Sword of Power or what happens when you fire a 1000-MegaHurt InstaKill Thunderbomb at the Disgusting Fangbeast of Blort. We're not discussing game design in this chapter; we're talking about designing a game program.

When you design a game program, you're designing two important features of the code: the game's program structure and its file structure. This chapter explains how professional game programmers design the structure of their game programs. It also provides an overview of how they structure the files in a game. It demonstrates these concepts by creating a game called Ping.

In addition, the Ping example illustrates how to write a real game. It demonstrates dividing a game into levels. In this case, each level will be a ping-pong match. Ping also shows how to create and use message maps. It

shows how easy message maps make it to respond to user input by updating the state of the game. Finally, the Ping program demonstrates game startup and shutdown.

# Program Structure

Every computer program ever written is about processing data. As a result, when you design your game program structure, you must specify what data the program processes and what processing operations it performs. This two-step process tells you how to write your program.

To specify the data your program processes, you design the program's objects. Specifying the operations the program performs involves designing the member functions for all of the objects.

## Designing Game Objects

Software objects were introduced in [chapter 3](#), "Introducing Object-Oriented Programming." In general, figuring out what objects a game needs is pretty straightforward. Essentially, you create objects for the things you see on the screen. In fact, writing games is one of the best ways to learn object-oriented design because you can generally see the objects you're designing. [Figure 7.1](#) shows a screen shot of the game Ping, which we'll write later in this chapter. Ping provides good practice for learning game software design.

**Figure 7.1. Ping requires objects for the ball and the paddle.**



The Ping game shown in this figure is a simple simulation of ping-pong or table tennis. Players use the paddles at the left and right edges of the screen to hit the ball back and forth. As you can see by looking at this screenshot, it's very intuitive that the game requires at least two types of objects in the software:

one to represent the ball and another to represent the paddles.

## Note

You'll find the files for Ping on the CD in the folder \Source\Chapter07\Prog\_07\_01. There's a compiled executable version of Ping that you can run in the \Source\Chapter07\Bin folder.

After you figure out what objects a game needs, you must design the objects themselves. Specifically, you design your objects identifying the important characteristics or attributes of the object. For example, the ball in [Figure 7.1](#) would require an object that has a position, a height, and width, and a vector describing the ball's movement, and a bounding rectangle. [chapter 4](#), "Introducing the LlamaWorks2D Game Engine," demonstrated the use of such an object by introducing the `llamaworks2d::sprite` class.

Obviously there is a lot more to designing objects than I've explained in these few paragraphs. In fact, entire books have been written about this topic. However, this short introduction should be enough to get you started. After you finish this book, it's a good idea to invest in a book on object-oriented design. You'll find a few such books in the Suggested Reading list on the CD.

## Designing Game Tasks

Although each game you write is unique, nearly every game performs the same set of basic tasks, which are listed below. After you've written code for these tasks for a couple of games, you will become very proficient at it. As a result, you'll be able to rapidly move into the tasks that are unique to the game you're working on.

### Factoid

Game programmers have various terms for the message loop. They may call it the update loop, the rendering loop, the main game loop, or the game loop.

- **Game initialization** During game initialization, your game creates the objects that it uses throughout the entire game. At this point, it does not create objects that it uses on individual levels. Instead, it performs tasks such as loading bitmaps for opening screens and for backgrounds the player sees on all levels of the game. It can also be a time for loading menus, setting up game controllers, initializing the player's score, and so forth.
- **Level initialization** When your game initializes a level, it creates all the objects used on that level. This is when you should load all of the bitmaps the level requires. As mentioned in previous chapters, loading bitmaps is not a task you want your game doing while it is running. It's a lengthy task and will bring the game to a standstill. That's why your game should take care of it during level initialization. Level initialization is also a good time to load music clips that the level uses.
- **Processing messages in the message loop** Every game uses a message loop. During the message loop, the game checks the input devices, such as the keyboard or mouse, for input messages. If there are input messages, the game calls functions that react to the messages by updating the game's state. The Ping game shown in [Figure 7.1](#) is a good example. The player using the right paddle presses the up or down button on the keyboard to move the right paddle. The player on the left uses the A and Z keys to move the left paddle up and down, respectively. Every time Ping's message loop executes, it checks the keyboard for those four keystrokes. When it gets messages saying that one of those four keys has been pressed, the loop reacts by adjusting the position of the appropriate paddle.
- **Updating and rendering a frame** As you've seen in previous chapters, every game must update the state of its data during each frame. The update may be based on the input the game gets from the player. It may also be based on information from the previous frame, such as using a ball's velocity to move it to a new position on the screen. In addition, the update can include playing the appropriate sounds and music. The game must also render all onscreen objects during each frame of animation.
- **Level cleanup** During level cleanup, your game deletes any objects that it will not use on the next level. It also releases the memory for bitmaps that

will no longer be used, displays the player's score for the level, saves the score, and so on.

- **Game cleanup** Your game performs game cleanup just before the program exits. This task involves deleting all objects used by the game, releasing all memory used by bitmaps and sound clips, displaying the player's score for the session, saving the level number that the player last completed, and so forth.

The example game Ping will demonstrate how to handle all these tasks.

# File Structure

After you've figured out what objects a game needs and the tasks it performs, designing the game's file structure is fairly straightforward. For each class you define, you should create a .h and a .cpp file. The .h file holds the class definition and the inline member functions. The .cpp file contains the member functions that are too big or too complex to be inline. That's all there is to it. If you follow these guidelines, you'll find it easy to use your objects in your game.

Recall that the LlamaWorks2D game engine uses objects to represent the application and the game. The game engine provides the code for the application class and creates the application object itself. However, you have to use the `llamaworks2d::game` class the engine provides to create your own game class. Therefore, in addition to a pair of files for your objects, you'll need a .cpp file for your game class. If your game is complex, you may need a .h file for your game class too.

You can see an example of how game files should be structured by looking at the Ping game. Ping has one class for the ball and another for the paddles. Therefore, it has a file called Ping.cpp, a pair of files called Ball.h and Ball.cpp, and another pair named Paddle.h and Paddle.cpp.

# A Game Called Ping

It's time to put all the information presented in this book so far into practice by writing Ping. As you may have guessed, this game is a knockoff of Pong, the granddaddy of all video games. Chances are fairly good that you're not old enough to remember Pong, but I am. I remember being rather excited over it when it first came out. When kids today see it, they usually just laugh at how sad it is compared to today's games. Nevertheless, our games still do all of the same basic tasks that Pong did. Therefore, writing Ping provides a straightforward introduction to building games.

## Introducing Ping

Ping is a simple two-player game modeled on ping-pong. The ball is served randomly to the left or right player. When the ball moves toward a player's paddle, the player moves the paddle up or down to block the progress of the ball. If the ball hits the paddle, it bounces back toward the other side. If not, it passes off the edge of the screen. When it leaves the screen, the player on the opposite side scores a point. Whenever a player scores a point, a small red dot appears on that player's side of the screen near the bottom. If the ball hits the top or bottom of the area of play, it bounces.

Now that we've defined the rules of the game, let's build it.

## Writing Ping

Implementing Ping takes a surprising amount of code. As a result, I'll show pieces of the game in several code listings to make it all a bit more comprehensible.

The game Ping has three classes in it. The first is the game class. It, and its member functions, are stored in the file Ping.cpp. In addition, Ping has a `ball` class and a `paddle` class. There are a pair of files for each of these two classes.

The major steps that Ping must perform when it starts are:

- Create the game's objects.



- Initialize the program.
- Initialize the game.
- Initialize a level.
- Create the message map.
- Update a frame.
- Render a frame.
- Clean up a level.
- Clean up the game.

## Creating the Game's Objects

The game object for Ping is defined in the class `ping` in the file `Ping.cpp`. This class is only moderately more complex than the `my_game` class in [Listing 4.3](#) in [chapter 4](#). [Listing 7.1](#), which gives the first portion of `Ping.cpp`, shows the definition of the `ping` class.

### Listing 7.1. The `ping` class and its message map

```
1  #include "LlamaWorks2d.h"
2
3  using namespace llamaworks2d;
4
5  #include "Ball.h"
6  #include "Paddle.h"
7
8  class ping : public game
9  {
10 public:
11     bool OnAppLoad();
12     bool InitGame();
13     bool InitLevel();
14     bool UpdateFrame();
15     bool RenderFrame();
16
17     ATTACH_MESSAGE_MAP;
18
```

```

19  bool OnKeyDown(
20      keyboard_input_message &theMessage);
21
22  bool BallHitPaddle();
23
24  void SetBallStartPosition();
25  void SetBallStartDirection();
26
27  bool DoLevelDone();
28  bool DoGameOver();
29
30  private:
31      ball theBall;
32      paddle leftPaddle;
33      paddle rightPaddle;
34      sprite theTrophy;
35
36      sprite scoreMarker;
37      int player1Score;
38      int player2Score;
39
40      bitmap_region areaOfPlay;
41  };
42
43
44  CREATE_GAME_OBJECT(ping);
45
46  START_MESSAGE_MAP(ping)
47      ON_WMKEYDOWN(OnKeyDown)
48  END_MESSAGE_MAP(ping)

```

The first thing to notice is that the `#include` statements for `Ball.h` and `Paddle.h` on lines 56 both appear after the `using` statement on line 3. This is because they both use the `llamaworks2d` namespace. Many C++ programmers do not like this style. They prefer to keep all of their `#include` statements together. Generally, I agree that is a better idea. If that is your preference as well, you can copy the `using` statement on line 3 to the beginning of `Ball.h` and `Paddle.h`. I showed this style of definition in this program because you will encounter it when looking at code written by other game programmers.

The `ping` class has 11 member functions. Several of them are required by the game engine. As you might expect, the engine provides reasonable default functions if you don't write your own.

The functions `OnAppLoad()`, `InitGame()`, `InitLevel()`, `UpdateFrame()`, `RenderFrame()`, `DoLevelDone()`, and `DoGameOver()` are all required member functions that the `ping` class inherits from the `llamaworks2d::game` class. The `ping` class overrides the versions of these functions it inherits so that they can be customized for this game.

On line 17, the `ping` class attaches its message map. The prototype for its one

and only message-handling function is given on lines 1920. Because of the message map, which appears on lines 4648, the game engine automatically calls the `OnKeyDown()` function whenever a player presses a key on the keyboard. I'll discuss this in more detail shortly.

The three functions on lines 2225 are called by the other functions in `Ping.cpp`. They are used to implement some of the game logic.

Lines 3040 of [Listing 7.1](#) show the private member data for the `ping` class. The first three are `ball` and `paddle` objects. I'll show the definition for these in a moment. Line 34 declares a `sprite` member variable called `theTrophy`. You saw how to use `sprite` objects in [chapter 4](#). When a player wins the game, it displays a trophy on the winner's side of the screen.

## Tip

You'll find detailed information on Windows messages on the Web at [www.msdn.microsoft.com](http://www.msdn.microsoft.com).

The `ping` class also uses a `sprite` object for its score markers. Ping displays them across the bottom of the screen to show the player's current score. The scores themselves are kept in the member variables defined on lines 3738. Lastly, the `ping` class defines a variable that specifies the portion of the screen that it uses to play the game.

On lines 4648, the `ping` class defines its message map. The purpose of the message map is to connect specific messages sent by Windows to functions in the game. For example, the macro on line 47 connects the Windows message `WM_KEYDOWN` to the `OnKeyDown()` function. There are literally hundreds, if not thousands, of different messages that Windows can send to your game.

## Factoid

Packaging data into an easy-to-use object is often called "cooking" the data. `LlamaWorks2D` takes "raw" input messages and "cooks" for you.

Windows sends your game a `WM_KEYDOWN` message whenever the player presses a key on the keyboard. The message contains an identifier for the key that was pressed. These identifiers are defined by LlamaWorks2D in the file `LW2DInputDevice.h`.

When Windows sends a `WM_KEYDOWN` message to your game, it sends the message in a very "raw" form. Those new to Windows programming often find it hard to extract the information they need from the message. To make things easier, LlamaWorks2D automatically packages the message into a `keyboard_input_message` object, which is also defined in `LW2DInputDevice.h`. Your game uses the member functions of the `keyboard_input_message` class to extract the message's information.

Because input from the `WM_KEYDOWN` message is in a very raw form, it does not, for instance, send your game an 'A' or 'a' character when the player presses the A key. Instead, it gets the key code `KC_A`, which is defined by LlamaWorks2D. Microsoft calls this a *virtual key code*. You can find more information on virtual key codes at [www.msdn.microsoft.com](http://www.msdn.microsoft.com).

If you want your game to receive characters, rather than key codes, LlamaWorks2D helps you out. It defines a special macro called `ON_WMCHAR()` that connects the Windows `WM_CHAR` message to a function. The `WM_CHAR` message provides characters rather than key codes.

You may wonder why it's worth bothering to use key codes rather than characters. The answer is that many keystrokes are not in the set of characters that generate a Windows `WM_CHAR` message. For example, the up and down arrows do not generate a `WM_CHAR` message. They only generate `WM_KEYDOWN` messages. In general, games respond to the `WM_CHAR` message when the player is typing in a string of characters. During gameplay, they use `WM_KEYDOWN` messages instead.

One word of warning before we move on: LlamaWorks2D does not let you put the macros `ON_WMKEYDOWN()` or `ON_WMCHAR()` more than once in a message map. You can actually write them in more than once and your program will compile without an error. However, only the first use of these macros will work.

## Mapping Other Windows Messages

LlamaWorks2D enables you to connect, or map, any Windows message to any function you want. To make this happen, use the LlamaWorks2d `ONMESSAGE()` macro. For example, if you want to map the message `WM_ACTIVATE` to a function

in your game class named `OnActivate()`, you can do so by putting the following statement in your game's message map:

```
ONMESSAGE(WM_ACTIVATE, OnActivate)
```

This statement tells the game engine to automatically call the `OnActivate()` function any time the game receives a `WM_ACTIVATE` message.

The first parameter to the `ONMESSAGE()` macro must always be the Windows message you want map. The second parameter is the name of the function to map the message to.

## Warning

It is possible to set a variable of an enumerated type to a value other than the ones specified in the enumeration. For instance, there is a way to set a `bounce_direction` variable to 10. However, you should not do this because it can cause unexpected results in programs. LlamaWorks2D is a good example. It expects all variables of type `bounce_direction` to be assigned one of the values on the list on lines 910 of [Listing 7.2](#). If you manage to assign it a value not in the list, LlamaWorks2D will not know what to do with that value. It is likely to cause the game engine to crash.

In addition to its game class, Ping must create a `ball` object and two `paddle` objects. Both the `ball` and `paddle` classes are derived through inheritance from the `llamaworks2d::sprite` class. [Listing 7.2](#) provides the code for the `ball` class.

## Listing 7.2. The `ball` class

```
1 #ifndef BALL_H
2 #define BALL_H
3
4 class ball : public sprite
5 {
6 public:
7     enum bounce_direction
8     {
```

```

9     X_DIRECTION,
10    Y_DIRECTION
11 };
12 public:
13     void Bounce(
14         bounce_direction dir);
15 };
16
17 #endif

```

The `ball` class is extremely simple because the `sprite` class does most of the actual work. The only thing the `ball` class really needs to do is to bounce the ball whenever the program tells it to. Everything else the `ball` class needs to do is handled by the `sprite` class.

On 711 of [Listing 7.2](#), the `ball` class defines an enumerated type. C++ provides enumerated types as a way of creating a type that can only be set to a specific set of values. It also lets you define what the values are that variables of that type can be set to. In this case, the name of the type is `bounce_direction`. Variables of type `bounce_direction` can only be set to the values `X_DIRECTION` and `Y_DIRECTION`.

The `bounce_direction` type is used for the parameter of the public member function `Bounce()`. The prototype for `Bounce()` is on lines 13-14. The code for the `Bounce()` function appears in [Listing 7.3](#).

### Listing 7.3. Bouncing the ball

```

1 void ball::Bounce(
2     bounce_direction dir)
3 {
4     vector ballVelocity = Movement();
5     switch (dir)
6 {
7     case X_DIRECTION:
8         ballVelocity.X(ballVelocity.X() * -1);
9         break;
10
11     case Y_DIRECTION:
12         ballVelocity.Y(ballVelocity.Y() * -1);
13         break;
14     }
15     Movement(ballVelocity);
16 }

```

The `Bounce()` function in [Listing 7.3](#) begins by calling the `sprite::Movement()` function to

get the ball's movement vector. It uses a `switch` statement with two cases. `Bounce()` executes the `case` statement on lines 79 whenever the parameter `dir` is equal to `X_DIRECTION`. That `case` statement bounces the ball in the x direction by multiplying the x component of its movement vector by `-1`. So if the ball was moving left, it will move right after the statement on line 8 gets executed. If it was moving right, it will move left after the multiplication on line 8.

## Factoid

Programmers often say that classes like `ball` are a specialization of `sprite`. A `sprite` is a generic concept. Many things can be `sprites`. On the other hand, a `ball` is a specific thing. In a game, a `ball` is a specific or special type of `sprite`. Base classes are often generic, while derived classes are typically specializations.

If the `dir` parameter is equal to `Y_DIRECTION`, `Bounce()` executes the `case` statement on lines 1113. The statement on line 12 bounces the ball in the y direction.

That's all there is to the `ball` class. Inheritance makes it remarkably simple.

Like the `ball` class, the `paddle` class derives from `sprite`. The primary difference between a `sprite` and a `paddle` is that paddles in Ping can only move up and down. [Listing 7.4](#) illustrates this difference.

## Listing 7.4. The `paddle` class

```
1 #ifndef PADDLE_H
2 #define PADDLE_H
3
4
5 class paddle : public sprite
6 {
7 public:
8     bool Move();
9     void Movement(
10     vector direction);
11     vector Movement();
12 };
13
14 inline vector paddle::Movement()
15 {
16     return (sprite::Movement());
17 }
```

```
18
19 #endif
```

As this listing shows, the `paddle` class has three member functions that control how paddles move. The `Move()` and `Movement()` functions, whose prototypes appear on lines 811, override the same functions in the `sprite` class. The code for one of the `Movement()` functions is given on lines 1417. All it does is call the corresponding function in the `sprite` class. Since the `sprite` class already provides a `Movement()` function that gets the movement vector, you may wonder why it's included at all. It's there because of the compiler. The Dev-C++ compiler, which is based on the Gnu C++ compiler (GCC), requires that if you override one overloaded function, you must overload all functions of the same name. The `paddle` class must override the `Movement()` function that sets the movement vector (the one whose prototype is on lines 910). Because it overrides that version of `Movement()`, it must override all versions of `Movement()` in the `sprite` class. The `sprite` class contains two `Movement()` functions, so the `paddle` class must contain either zero or two `Movement()` functions.

## Note

Not all compilers require that you override all base class functions of the same name.

The code for the rest of the `paddle` class's member functions is given in [Listing 7.5](#).

## Listing 7.5. Controlling a paddle's movement

```
1  bool paddle::Move()
2  {
3      bool moveOK = true;
4
5      vector velocity = Movement();
6      Y(Y() + velocity.Y());
7      velocity.Y(0);
8      Movement(velocity);
9
10     return (moveOK);
11 }
12
```



```
13
14 void paddle::Movement(
15     vector direction)
16 {
17     ASSERT(direction.X()==0);
18     sprite::Movement(direction);
19 }
```

Normally sprites move in both the x and y directions. The `paddle::Move()` function moves the paddle only in the y direction on line 6. There is no possibility of it moving in the x direction.

Notice that the `Move()` function sets the paddle's movement vector to 0 on line 7. If it did not do this, the paddle would keep moving. In Ping, the paddle should only move when the player presses one of the appropriate keys. Setting the movement vector to 0 means that the only way the paddle can move in the next frame is if the player presses a key again. That's exactly as it should be.

The `paddle::Movement()` function, which begins on line 14, does two things. First, it uses a macro defined in `LlamaWorks2D` to make sure that no attempt is being made to move the paddle in the x direction. The `ASSERT()` macro takes a condition as its only parameter. On line 17, the condition tests to see whether the x value of the direction vector is 0. If it is, all is well. If that assertion is not equal to `true`, then the program crashes.

What?!?!?

## Tip

Professional programmers use assertions a lot in their programs. I strongly recommend that you do the same.

Yes, it's true. `ASSERT()` deliberately causes the program to crash if the assertion is not `true`. If the x value of the direction vector is not 0, then you made a programming error. You want to find all programmer errors *before* you release your game for sale. Assertions help you do that. You can't possibly miss it when your game crashes completely, so it's very easy to find errors with assertions.

Assertions are for debugging. Hopefully, they will help you catch your programming errors before you sell your game. When you're ready to release your game for sale, you remove all assertions. You don't have to go through and take them out yourself. Instead, uncomment the `#define` statement on line 5 of `LlamaWorks2D.h` and then recompile your program. A `#define` statement is a special type of command recognized by the C++ preprocessor, which is part of the compiler. The `#define` statement on line 5 of `LlamaWorks2D.h` tells the compiler that you don't want debugging stuff in your code. When you uncomment the `#define` statement, all assertions are automatically removed. It's a rather slick trick commonly used by C++ programmers.

## Warning

Never use assertions for anything other than programmer errors. If you utilize them to handle such problems as input errors, players won't like your game much.

## Initializing the Program

As previous chapters mentioned, initializing a Windows program is not for the faint of heart. For the most part, `LlamaWorks2D` handles the initialization for you. However, `LlamaWorks2D` calls your game class's `OnAppLoad()` function after the program starts but before it creates the program's window. This gives you the opportunity to control how the initialization takes place. [Listing 7.6](#) gives the `OnAppLoad()` function for the `ping` class.

### Listing 7.6. The `OnAppLoad()` function

```
1  bool ping::OnAppLoad()
2  {
3      bool initOK = true;
4
5      //
6      // Initialize the window parameters.
7      //
8      init_params lw2dInitiParams;
9      lw2dInitiParams.openParams.fullScreenWindow = true;
10     lw2dInitiParams.winParams.screenMode. AddSupportedResolution(
11         LWSR_1024X768X32);
```

```

12  lw2dInitiParams.winParams.screenMode. AddSupportedResolution(
13      LWSR_1024X768X24);
14  lw2dInitiParams.winParams.screenMode. AddSupportedResolution(
15      LWSR_800X600X32);
16  lw2dInitiParams.winParams.screenMode. AddSupportedResolution(
17      LWSR_800X600X24);
18  lw2dInitiParams.openParams.millisecondsBetweenFrames = 0;
19
20  // This call MUST appear in this function.
21  theApp.InitApp(lw2dInitiParams);
22
23  return (initOK);
24  }

```

This function sets the game to run in a full-screen window on line 9. Next, it states the screen resolutions it supports on lines 10-17. The `init_params` variable declared on line 8 has a member that contains information about initializing Windows when the program starts up. Part of that information is a collection of screen resolutions that the game supports. That collection is kept in `screenMode`, which is an object of type `screen_mode`. The `screen_mode` class has a member function called `AddSupportedResolution()`. Each time your game calls `AddSupportedResolution()`, it passes in a value of type `screen_resolution`. The `screen_resolution` type contains a list of values that specify screen resolutions that LlamaWorks2D recognizes. [Table 7.1](#) shows the available screen resolutions.

**Table 7.1. Llamaworks2D Can Set The Screen to These Resolutions**

Value	Resolution
<code>LWSR_UNKNOWN</code>	The resolution is unknown or not specified.
<code>LWSR_640X480X24</code>	Sets the screen to 640 x 480 x 24. Poor image quality.
<code>LWSR_640X480X32</code>	Sets the screen to 640 x 480 x 32. Poor image quality.
<code>LWSR_800X600X24</code>	Sets the screen to 800 x 600 x 24. Good image quality.
<code>LWSR_800X600X32</code>	Sets the screen to 800 x 600 x 32. Good image quality.
<code>LWSR_1024X768X24</code>	Sets the screen to 1024 x 768 x 24. Very good image quality.
	Sets the screen to 1024 x 768 x 32. Very good

LWSR_1024X768X32	image quality.
LWSR_1152X864X24	Sets the screen to 1152 x 864 x 24. Excellent image quality.
LWSR_1152X864X32	Sets the screen to 1152 x 864 x 24. Excellent image quality.

---

The resolutions given in [Table 7.1](#) are the most common ones that games use. The first number in the resolution is the number of horizontal pixels. The second is the number of vertical pixels. The third number is the number of bits per pixel. So the value `LWSR_800X600X32` sets the screen to a resolution of 800 pixels across and 600 pixels tall with 32 bits per pixel.

## Factoid

The most common screen resolution for games today is 800 x 600 x 24. It is supported on virtually all computers currently in use.

The `OnAppLoad()` function in [Listing 7.6](#) adds four supported screen resolutions. Most games support multiple resolutions. The reason is that you cannot guarantee that a player's screen supports the resolution you most want to use. Therefore, your game has to be able to find an acceptable resolution that the player's computer does support. LlamaWorks2D lets you add as many screen resolutions as it offers. When it attempts to set the screen resolution, it starts with the first resolution your `OnAppLoad()` function gives it. If that one doesn't work, it goes to the next one in the list, and so on, until it reaches the end of the list. If it can't find a supported resolution, the engine displays an error message saying that the game can't run on the player's computer.

The vast majority of computers that people are using today support the resolutions offered by LlamaWorks2D. If a player's computer doesn't support any of those resolutions, chances are their computer is far too old to run your game.

## Initializing the Game

Initializing Ping is a more complex task than you might think. To initialize Ping, the program must do the following tasks:

1. Set the size of the playable area.
2. Load the ball's image.  
Initialize all of the ball's member data except the position and movement
3. vector.
4. Load the bitmap for the paddles.  
Initialize all of the paddles' member data except the position and
5. movement vectors.
6. Load the bitmap for the score markers.
7. Load the bitmap for the trophy.

All of this initialization is handled by the `ping` class's `InitGame()` function. As you may recall from [chapter 4](#), `LlamaWorks2D` calls `InitGame()` automatically. [Listing 7.7](#) shows `InitGame()`.

## Listing 7.7. The `ping::InitGame()` function

```
1  bool ping::InitGame()
2  {
3      bool initOK = true;
4      bitmap_region boundingRect;
5
6      areaOfPlay.top = 0;
7      areaOfPlay.bottom = theApp.ScreenHeight() - 20;
8      areaOfPlay.left = 0;
9      areaOfPlay.right = theApp.ScreenWidth();
10
11     theBall.BitmapTransparentColor(color_ rgb(0.0f,1.0f,0.0f));
12     initOK =
13         theBall.LoadImage(
14             "ball2.bmp",
15             image_file_base::LWIFF_WINDOWS_BMP);
16
17     if (initOK==true)
18     {
19         srand((unsigned)time(NULL));
```

```
20     boundingRect.top = 0;
21     boundingRect.bottom = theBall.BitmapHeight();
22     boundingRect.left = 0;
23     boundingRect.right = theBall.BitmapWidth();
24     theBall.BoundingRectangle(boundingRect);
25 }
26 else
27 {
28     ::MessageBox(
29         NULL,
30         theApp.AppError().ErrorMessage().c_str(),
31         NULL,
32         MB_OK | MB_ICONSTOP | MB_SYSTEMMODAL);
33 }
34
35 if (initOK==true)
36 {
37     leftPaddle.BitmapTransparentColor(
38         color_rgb(0.0f,1.0f,0.0f));
39     initOK =
40     leftPaddle.LoadImage(
41         "paddle.bmp",
42         image_file_base::LWIFF_WINDOWS_BMP);
43     if (!initOK)
44     {
45         ::MessageBox(
46             NULL,
47             theApp.AppError().ErrorMessage().c_str(),
48             NULL,
49             MB_OK | MB_ICONSTOP | MB_SYSTEMMODAL);
50     }
51 }
52
53 if (initOK==true)
54 {
55     scoreMarker.LoadImage(
56         "marker.bmp",
57         image_file_base::LWIFF_WINDOWS_BMP);
58     if (!initOK)
59     {
60         ::MessageBox(
61             NULL,
62             theApp.AppError().ErrorMessage().c_str(),
63             NULL,
64             MB_OK | MB_ICONSTOP | MB_SYSTEMMODAL);
65     }
66 }
67
68 if (initOK)
69 {
70     theTrophy.BitmapTransparentColor(color_rgb(0.0f,1.0f,0.0f));
71     initOK =
72     theTrophy.LoadImage(
73         "trophy.bmp",
74         image_file_base::LWIFF_WINDOWS_BMP);
75 }
76
77
```

```

81  if (initOK==true)
82  {
83      boundingRect.top = 3;
84      boundingRect.left = 10;
85      boundingRect.bottom = 63;
86      boundingRect.right = 22;
87      leftPaddle.BoundingBox(boundingRect);
88
89      rightPaddle = leftPaddle;
90
91      player1Score = player2Score = 0;
92      InitLevel();
93  }
94
95  return (initOK);
96  }

```

The `InitGame()` function begins by specifying the portion of the screen that is used for playing the game. The game does not use the entire screen, as [Figure 7.2](#) shows.

**Figure 7.2. The area of play is marked by the black rectangle.**



The black rectangle in [Figure 7.2](#) indicates the limits of the playable area. The game specifies the area of play because it needs a place to put the score markers. By limiting the portion of the screen that can be used for playing, the game leaves a blank area across the bottom of the screen where it can place score markers.

On lines 1115 of [Listing 7.7](#), the `InitGame()` function sets the transparent color of the ball's bitmap and then loads it. If the bitmap is properly loaded, `InitGame()` calls the C++ Standard Library function `srand()` to seed the random number generator. The C++ Standard Library contains a function named `rand()` that generates random numbers. Ping uses it to set the initial speed and direction of the ball. When Ping starts up, it calls the `srand()` function to seed the random number generator with the current time. This helps ensure that the numbers that `rand()` generates are truly random.

Lines 2125 set the bounding rectangle of the ball. This is used to detect when the ball collides with the boundaries of the area of play or with the paddles. On lines 3944, `InitGame()` sets the transparent color of the left paddle's bitmap and loads it. If it is loaded properly, `InitGame()` loads the bitmap for the score marker on lines 5860. On lines 7478, `InitGame()` loads the bitmap for the trophy.

The final initialization that `InitGame()` performs on the left paddle is to set its bounding rectangle. It's at this point that `InitGame()` does something very tricky. Line 89 sets the right paddle equal to the left paddle. What this does is copy information in `leftPaddle` into `rightPaddle`. That includes integer values it contains such as the size of the bounding rectangle. However, unlike the integer values in the object, the `paddle` class uses a tool called a pointer that does not let the bitmap image be copied from paddle to paddle. Instead, they *share the bitmap*.

That's right. Both the left paddle and the right paddle share the very same bitmap image. This is a very cool trick that saves *lots* of memory. Pointers are extremely powerful tools that we'll cover in [chapter 11](#), "Pointers." It is very common for games to share bitmaps between objects. If they couldn't, their graphics would take up a lot more memory and therefore the individual levels would be significantly smaller.

The `InitGame()` function finishes up by setting both players' scores to 0 and calling the `InitLevel()` function to initialize the first level of the game.

## Initializing a Level

By comparison to initializing the game, initializing an individual level of Ping is considerably simpler. The `llamaworks2d::game` class contains a function named `InitLevel()` that you can override to initialize your levels. The game engine calls `InitLevel()` automatically at the beginning of each level. The tasks that this function performs are:

1. Sets the initial position of the paddles



2. Sets the initial position of the ball
3. Sets the ball's initial movement vector

[Listing 7.8](#) provides the `InitLevel()` function for the `ping` class.

### Listing 7.8. The `ping::InitLevel()` function.

```
1  bool ping::InitLevel()
2  {
3      bool initOK = true;
4      bitmap_region boundingRect =
5          leftPaddle.BoundingRectangle();
6
7      leftPaddle.X(5);
8      int paddleHeight = boundingRect.bottom - boundingRect.top;
9      int initialY =
10         ((areaOfPlay.bottom - areaOfPlay.top)/2) -
11         (paddleHeight/2);
12     leftPaddle.Y(initialY);
13
14     int bitmapWidth = rightPaddle.BitmapWidth();
15     int initialX = areaOfPlay.right - bitmapWidth - 5;
16     rightPaddle.X(initialX);
17     rightPaddle.Y(initialY);
18
19     SetBallStartPosition();
20     SetBallStartDirection();
21
22     LevelDone(false);
23     return (initOK);
24 }
```

On line 4 of [Listing 7.8](#), the `InitLevel()` function declares the variable `boundingRect` that it uses to temporarily hold the bounding rectangle of the left paddle. It sets the x position of the left paddle on line 7. Next, it calculates the height of the paddle's bounding rectangle. On lines 9-11, it uses the height to calculate the paddle's y position. The calculation centers the paddle halfway between the top and bottom of the area of play. `InitLevel()` sets the left paddle's y position on line 12.

At the beginning of each level, the y position of both paddles is the same. They are both set halfway down the area of play. So on line 17, `InitLevel()` uses the value it calculated on lines 9-11 as the y value of the right paddle. The x position is 5 pixels to the left of the right edge of the area of play. That value is calculated on line 15 and passed to the right paddle on line 16.

The `InitLevel()` function ends by calling functions to calculate and set the initial position, speed, and direction of the ball, and by telling the `lamaworks2d::game` class that the level is not done.

[Listing 7.9](#) gives the functions that the `ping` class uses to set the ball's starting position, speed, and direction.

## Listing 7.9. Setting the initial values of the ball

```
1 void ping::SetBallStartPosition()
2 {
3     int startX = rand();
4
5     while (startX >= areaOfPlay.right)
6     {
7         startX /= 10;
8     }
9
10    int startY = rand();
11
12    while (startY >= areaOfPlay.bottom)
13    {
14        startY /= 10;
15    }
16
17    theBall.X(startX);
18    theBall.Y(startY);
19 }
20
21 void ping::SetBallStartDirection()
22 {
23     int xDirection = rand();
24
25     while (xDirection > 10)
26     {
27         xDirection /= 10;
28     }
29
30     if (xDirection < 3)
31     {
32         xDirection = 3;
33     }
34
35     int yDirection = rand();
36
37     while (yDirection > 10)
38     {
39         yDirection /= 10;
40     }
41
42     if (yDirection < 3)
43     {
44         yDirection = 3;
45     }
46 }
```

```

47  if (theBall.X() >= (areaOfPlay.right - areaOfPlay.left)/2)
48  {
49      xDirection *= -1;
50  }
51
52  if (theBall.Y() >= (areaOfPlay.bottom - areaOfPlay.top)/2)
53  {
54      yDirection *= -1;
55  }
56
57  theBall.Movement(vector(xDirection, yDirection));
58  }

```

[Listing 7.9](#) contains the functions `SetBallStartPosition()` and `SetBallStartDirection()`. The `SetBallStartPosition()` function begins by calling the C++ Standard Library `rand()` function to generate a random number. The loop on lines 58 test the random number to see if it's larger than the right edge of the area of play. If it is, the loop divides the number by 10. Because this is an integer division, there is no fractional part in the answer. Everything in the answer that is to the right of the decimal point is thrown away. This process essentially throws away the rightmost digit of the random number. It keeps throwing away the rightmost digit until the number is less than or equal to the right edge of the area of play. The result is used for the ball's x position.

Next, `SetBallStartPosition()` generates a random number for the ball's y position. While the value of the y position is larger than the bottom of the area of play, `SetBallStartPosition()` divides the value by 10. By the time that the `SetBallStartPosition()` function gets to lines 17 and 18, it has generated x and y values that are guaranteed to be within the area of play. Therefore, it stores the x and y values in the ball.

The `SetBallStartDirection()` function uses a similar approach to calculate the ball's starting direction and speed. On line 23 it generates a random number for the x value of the ball's movement vector. The loop on lines 2528 ensures that the x value is less than or equal to 10. If the x value is less than 3, the `if` statement on lines 3033 set it to 3. The process then repeats on lines 3545 for the vector's y value.

Because the `rand()` function only generates random numbers greater than zero, the x and y components of the ball's movement vector are always positive. Therefore, the vector points down and to the right. If the ball is on the right half of the area of play, the statement on line 49 points the ball's x direction toward the left. If the ball is in the lower half of the screen, the statement on line 54 reverses its y direction so that it points up. The `SetBallStartDirection()` function ends by setting the ball's movement vector using the values it calculated.

# Handling Messages

As you saw earlier in this chapter, the `ping` class's message map connects the Windows message `WM_KEYDOWN` to the `OnKeyDown()` function. As a result, every time a player presses a key on the keyboard, `LlamaWorks2D` calls the `OnKeyDown()` function, which is shown in [Listing 7.10](#).

## Listing 7.10. The message handler

```
1  bool ping::OnKeyDown(  
2      keyboard_input_message &theMessage)  
3  {  
4      vector paddleDirection;  
5  
6      switch (theMessage.keyCode)  
7      {  
8          case KC_UP_ARROW:  
9              paddleDirection.Y(-15);  
10             rightPaddle.Movement(paddleDirection);  
11             break;  
12  
13             case KC_DOWN_ARROW:  
14                 paddleDirection.Y(15);  
15                 rightPaddle.Movement(paddleDirection);  
16                 break;  
17  
18             case KC_A:  
19                 paddleDirection.Y(-15);  
20                 leftPaddle.Movement(paddleDirection);  
21                 break;  
22  
23             case KC_Z:  
24                 paddleDirection.Y(15);  
25                 leftPaddle.Movement(paddleDirection);  
26                 break;  
27         }  
28         return (false);  
29     }
```

Every time you write a function that handles the `WM_KEYDOWN` message, you must make it return a `bool` value. Recall that the `ping` class is derived from the `llamaworks2d::game` class provided by `LlamaWorks2D`. It is actually possible to make a game class that is derived from `ping`. In fact, you can use inheritance with game classes as much as you want. For example, you can have a class that is derived from a class that is derived from a class that is derived from `game`. In such a situation, you might want to pass the message up the chain of inheritance. To do so, have your message handling function return `true`. If you

do not need to pass the message up the inheritance tree, your message-handling function should return `false`.

## Tip

Your game classes will generally be derived from the `llamaworks2d::game` class. If so, their message-handling functions do not need to pass the message up the inheritance tree to the base class for handling. That means they should return the value `false`.

The `ping` class is derived from the `game` class. The `game` class doesn't do anything with keystrokes, so there's no sense in passing them along. You can if you want to, but it accomplishes nothing. Therefore, the `OnKeyDown()` function for the `ping` class returns `false`.

To find out which key was pressed, the `OnKeyDown()` function uses a C++ `switch` statement. The `switch` statement does the same thing as a group of chained `if-else` statements. If the key code equals the values in the `case` statements on lines 8, 13, 18, or 23, `OnKeyDown()` executes the commands between the `case` and `break` statements. If it doesn't equal any of those values, `OnKeyDown()` skips to the end of the `switch` statement and continues from there.

The `case` statement on line 8 is selected when the key code equals `KC_UP_ARROW`. It sets the y component of the right paddle's movement vector to -15. That makes the paddle move up 15 pixels on the next frame.

Likewise, the `case` statement on line 13 gets selected when the key code is `KC_DOWN_ARROW`. It sets the right paddle so that it moves down 15 pixels on the next frame.

The `case` statements beginning on lines 18 and 23 set the left paddle so that it moves 15 pixels up and down, respectively.

In general, your message-handling function processes all keystrokes it receives through a `WM_KEYDOWN` message in the manner shown here. It uses a `switch` statement to figure out which key was pressed, and then updates the game's current state.

## Updating a Frame

The paddles and ball in the Ping game purposely do not have a lot of smarts built into them. For instance, the ball "knows" how to move and bounce, but it doesn't know when to bounce. The paddles know they move up and down, but they don't know how far. Neither the paddles nor the ball know that the ball bounces off the paddles. Why implement these objects with so little smarts?

Many arcade games use balls and paddles. The `ball` and `paddle` classes could easily be modified to work in those games. You'll do yourself a big favor if you make the objects in your games as reusable as possible. It saves a lot of time if you can reuse objects from previous games you wrote.

To help make your objects reusable, you put the code that enforces the rules of the game into your game class. If you don't, the objects you write become highly dependent on each other. For instance, programming the ball to bounce off paddles means that the `ball` class always has to be used with the `paddle` class. You can't reuse the `ball` class in a game that doesn't have paddles. Instead, you have to rewrite the `ball` class from scratch. If you put the code that enforces the rules of the game into your game class, the ball just has to know how to move and bounce. Lots of games have balls that move and bounce, so a `ball` class written like this is very handy to have around.

The `ping::UpdateFrame()` function demonstrates how to put the game logic in a game class. It tells the ball and paddles when to move. It then tests for conditions such as the ball reaching an edge of the area of play or hitting a paddle. When `UpdateFrame()` discovers that one of events has occurred, it tells the ball and paddles how to react. [Listing 7.11](#) gives the code for the `UpdateFrame()` function.

## Listing 7.11. The `ping::UpdateFrame()` function

```
1  bool ping::UpdateFrame()
2  {
3      bool updateOK = true;
4
5      if (!GameOver())
6      {
7          theBall.Move();
8          rightPaddle.Move();
9          leftPaddle.Move();
10
11         if (BallHitPaddle())
12         {
13             theBall.Bounce(ball::X_DIRECTION);
14         }
15
16         if (theBall.X()+theBall.BitmapWidth() < areaOfPlay.left)
17         {
18             player1Score++;
```

```
19     if (player1Score >= 5)
20     {
21         GameOver(true);
22     }
23     else
24     {
25         LevelDone(true);
26     }
27 }
28 }
29 else if (theBall.X() > areaOfPlay.right)
30 {
31     player2Score++;
32
33     if (player2Score >= 5)
34     {
35         GameOver(true);
36     }
37     else
38     {
39         LevelDone(true);
40     }
41 }
42
43 if (theBall.Y() <= areaOfPlay.top)
44 {
45     theBall.Y(1);
46     theBall.Bounce(ball::Y_DIRECTION);
47 }
48 else if (theBall.Y() +
49     theBall.BoundingRectangle().bottom >=
50     areaOfPlay.bottom)
51 {
52     theBall.Y(areaOfPlay.bottom -
53     theBall.BoundingRectangle().bottom - 1);
54     theBall.Bounce(ball::Y_DIRECTION);
55 }
56
57 if (leftPaddle.Y() < areaOfPlay.top)
58 {
59     leftPaddle.Y(0);
60 }
61 else if (leftPaddle.Y() + leftPaddle.BitmapHeight() >
62     areaOfPlay.bottom)
63 {
64     leftPaddle.Y(
65     areaOfPlay.bottom - leftPaddle.BitmapHeight());
66 }
67
68 if (rightPaddle.Y() < areaOfPlay.top)
69 {
70     rightPaddle.Y(0);
71 }
72 else if (rightPaddle.Y() + rightPaddle.BitmapHeight() >
73     areaOfPlay.bottom)
74 {
75     rightPaddle.Y(
76     areaOfPlay.bottom - rightPaddle.BitmapHeight());
77 }
78 }
79
```

```
80 return (updateOK);  
81 }
```

The first task of the `UpdateFrame()` function is to test whether the game is over. If it is, `UpdateFrame()` doesn't perform the update. If the game is still in progress, it moves the ball and the paddles. Next, `UpdateFrame()` calls a function named `BallHitPaddle()`. (The `BallHitPaddle()` is presented next in [Listing 7.12](#) we'll examine it in a moment.) If the `BallHitPaddle()` function indicates that the ball hit a paddle, the `UpdateFrame()` function calls the `ball` class's `Bounce()` function on line 13 to bounce the ball in the x direction.

## Note

When you use a value from an enumerated type, you generally have to specify the class it's defined in, followed by two colons, followed by the enumerated value. The exception is when you use the value inside the member functions of the class the enumerated type is defined in.

Next, `UpdateFrame()` checks whether the ball went off the left edge of the area of play on line 16. If so, the player on the right (player 1) scores. `UpdateFrame()` increments player 1's score on line 18. If player 1 has scored more than five goals, player 1 is the winner and the game is over. The `UpdateFrame()` lets the game engine know that the game is over by calling the `game::GameOver()` function and passing it the value `TRue`. If player 1 has not won, then `UpdateFrame()` lets the game engine know that the current level (match) is over by calling `LevelDone()`, which is also a member of the `llamaworks2d::game` class.

If the ball did not go off the left edge of the area of play, the `UpdateFrame()` function tests to see if it went off the right edge on line 29. If it did, player 2 scored a point. `UpdateFrame()` checks to determine whether player 2 won the game. If so, it calls the `game::GameOver()` function to let LlamaWorks2D know the game is done. If the game is not yet over, `UpdateFrame()` invokes `LevelDone()` to tell the game engine that it's time for another level (match).

Lines 4347 bounce the ball in the y direction if it hits the top of the area of play. Lines 4855 bounce it when it hits the bottom. The `if` statement on lines 5766 keep the left paddle from going off the top or bottom of the area of play.



The `if` statement on lines 6877 do the same for the right paddle.

The `BallHitPaddle()` function, which was called on line 11 of [Listing 7.11](#), performs collision detection. It appears in [Listing 7.12](#).

## Listing 7.12. Did the ball hit a paddle?

```
1  bool ping::BallHitPaddle()
2  {
3      bool hitPaddle = false;
4      int paddleLeft, paddleRight, paddleTop, paddleBottom;
5      int ballLeft, ballRight, ballTop, ballBottom;
6
7      ballLeft = theBall.X() + theBall.BoundingBox().left;
8      ballRight = theBall.X() + theBall.BoundingBox().right;
9      ballTop = theBall.Y() + theBall.BoundingBox().top;
10     ballBottom = theBall.Y() + theBall.BoundingBox().bottom;
11
12     paddleRight =
13         leftPaddle.X() + leftPaddle.BoundingBox().right;
14     paddleTop =
15         leftPaddle.Y() + leftPaddle.BoundingBox().top;
16     paddleBottom =
17         leftPaddle.Y() + leftPaddle.BoundingBox().bottom;
18
19     bool leftEdge = (ballLeft <= paddleRight) ? true: false;
20
21     bool topEdge =
22         ((ballTop >= paddleTop) &&
23          (ballTop <= paddleBottom)) ?
24         true:false;
25
26     bool bottomEdge =
27         ((ballBottom >= paddleTop) &&
28          (ballBottom <= paddleBottom)) ? true:false;
29
30     if ((leftEdge) && ((topEdge) || (bottomEdge)))
31     {
32         theBall.X(paddleRight + 1);
33         hitPaddle = true;
34     }
35     else
36     {
37         paddleLeft =
38             rightPaddle.X() + rightPaddle.BoundingBox().left;
39         paddleTop =
40             rightPaddle.Y() + rightPaddle.BoundingBox().top;
41         paddleBottom =
42             rightPaddle.Y() +
43             rightPaddle.BoundingBox().bottom;
44
45         bool rightEdge = (ballRight >= paddleLeft) ? true:false;
46
47         topEdge =
48             ((ballTop >= paddleTop) &&
```

```

49     (ballTop <= paddleBottom)) ? true:false;
50
51     bottomEdge =
52         ((ballBottom >= paddleTop) &&
53         (ballBottom <= paddleBottom)) ? true:false;
54
55     if ((rightEdge) && ((topEdge) || (bottomEdge)))
56     {
57         theBall.X(paddleLeft - theBall.BitmapWidth() - 1);
58         hitPaddle = true;
59     }
60 }
61 return (hitPaddle);
62 }

```

The `BallHitPaddle()` function first calculates where the left, right, top, and bottom edges of the ball's bounding rectangle are on the screen. It also finds the right, top, and bottom of the left paddle on lines 1217. On lines 1934 it uses the positions of the ball and left paddle to find out if there was a collision between the two. Let's take a close look at how that happens.

On line 19, the `BallHitPaddle()` function uses the C++ conditional operator, which is made up of the symbols `?` and `:`, to test for an overlap between the left edge of the ball's bounding rectangle and the right edge of the paddle's bounding rectangle. The conditional operator tests the condition, which I've put in the parentheses before the question mark. If the condition is `true`, the conditional operator evaluates to its `true` expression, which is between the question mark and the colon. If the condition is `false`, the conditional operator evaluates to its `false` condition, which is between the colon and the semicolon. In this case, the values it evaluates to are `true` or `false`. The conditional operator is a C++ shorthand for an `if` statement. If you were to write the statement on line 19 using an `if`, it would look like the following code:

```

if (ballLeft <= paddleRight)
{
    leftEdge = true;
}
else
{
    leftEdge = false;
}

```

Lines 2124 use a conditional operator to determine if the top of the ball is

between the top and bottom of the left paddle. On lines 2628, `BallHitPaddle()` tests whether the bottom of the ball is between the top and bottom of the paddle. `BallHitPaddle()` uses the results of these tests in the `if` statement on line 30. If the left edge of the ball's bounding rectangle is past the right edge of the left paddle, and if either the top or the bottom of the ball is between the top and bottom of the paddle, a collision occurs between the ball and the paddle. This type of collision is shown in [Figure 7.3](#).

**Figure 7.3. The ball hit the left paddle.**



You can see that the left edge of the ball's bounding rectangle has moved past the right edge of the paddle's bounding rectangle. The top edge of the ball's bounding rectangle is between the top and bottom of the paddle. In this situation, the `if` statement on line 30 is `true` and `BallHitPaddle()` executes the statements on lines 32-33. If not, `BallHitPaddle()` jumps down to the `else` statement on line 35.

On lines 37-43, the `BallHitPaddle()` function finds the location on the screen of the right paddle's bounding rectangle. It uses a conditional operator to determine if the top edge of the ball is between the top and bottom of the right paddle on lines 47-49. The conditional operator on lines 51-53 test to determine whether the bottom edge of the ball's bounding rectangle is between the top and bottom of the right paddle.

Beginning on line 55, `BallHitPaddle()` tests to see if the right edge of the ball has gone past the left edge of the right paddle, and if the top or bottom edge of the ball is between the top and bottom of the right paddle. If so, the ball hit the right paddle. In that case, `BallHitPaddle()` adjusts the position of the ball on line 57 to ensure that it is one pixel to the left of the right paddle. It then sets a Boolean variable indicating that the ball hit the paddle.

## Approximation in Games

Ping uses a very simple method of bouncing a ball. It just reverses the sign of the movement vector in the direction of the bounce. This is not an accurate simulation of a bouncing ball. But for Ping, it's a good enough approximation.

It's often the case that professional game programmers substitute approximations into their games when they can. The approximation makes the game look as real as necessary, without incurring the performance decrease that would result from an accurate simulation of the relevant physics.

Approximation is a very good technique to use when you can. But be careful not to overuse it. If your approximations are too simple for the game you're writing, it makes the game look unrealistic.

So basically, use approximation, but don't overuse it. If your approximation looks right in your game, then it *is* right. If it looks unrealistic, you need to use better physics.

Before we move on, it's worth noting that professional programmers seldom break their conditions up as I've done in [Listing 7.12](#). For example, on lines 1928 I repeatedly used the conditional operator to perform three tests. The results of the tests are stored in variables of type `bool` and then used in the condition of the `if` statement. I did that to make each condition more readable. Professional programmers generally just put all of them into the `if` statement's condition. It sometimes makes for very complicated conditions that can be difficult for new programmers to read. If you're not comfortable attempting to build large, complex conditions in `if` statements, then

## Rendering a Frame

During each frame of animation, Ping must render all of its visual elements. This includes both paddles, the ball, and the score markers. As you've already seen, all of the visual elements in Ping are implemented as objects of type `sprite` or as objects derived from the `sprite` class. Ping can render these objects fairly easily using the `sprite` class's `Render()` function. As a result, Ping's `RenderFrame()` function is quite a bit simpler than its `UpdateFrame()` function. [Listing 7.13](#) gives the `RenderFrame()` function.

### Tip

If you find statements using the conditional operator to be difficult to read, you can use an `if-else` statement instead.

## Listing 7.13. Rendering is the easy part.

```
1  bool ping::RenderFrame()
2  {
3      theBall.Render();
4      rightPaddle.Render();
5      leftPaddle.Render();
6
7      scoreMarker.Y(areaOfPlay.bottom + 5);
8      int i=1;
9      while (i<=player1Score)
10     {
11         scoreMarker.X(
12             areaOfPlay.right -
13             (scoreMarker.BitmapWidth() + 5) * i);
14         scoreMarker.Render();
15         i++;
16     }
17
18     i=1;
19     while (i<=player2Score)
20     {
21         scoreMarker.X(
22             areaOfPlay.left +
23             (scoreMarker.BitmapWidth() + 5) * i);
24         scoreMarker.Render();
25         i++;
26     }
27
28     return true;
29 }
```

The `RenderFrame()` function renders the ball and paddles on lines 35 of [Listing 7.13](#). On line 7, it sets the y position of all of the score markers it is about to display. All score markers are rendered 5 pixels below the bottom of the area of play.

### Note

When there are parentheses in an equation, the program always evaluates the contents of the parentheses first.

The `while` loop on lines 916 renders the score markers for player 1. The equation on lines 1213 can be a little confusing unless you start with the part in the parentheses. The first thing that the program does when evaluating this equation is to add 5 to the width of the score marker's bitmap. The extra 5 pixels are added to provide a bit of blank space between each score marker. Next, the program multiplies the answer by the loop counter variable `i`. It then subtracts the resulting number from the right boundary of the area of play. This gives an `x` position for the score marker.

On lines 1926, the `RenderFrame()` function uses another `while` loop to position the score markers for player 2. Because player 2 operates the left paddle, the `RenderFrame()` function positions the score markers to the right of the left edge of the area of play.

That's really all there is to rendering. It's pretty straightforward.

## Cleaning Up a Level

At the end of a level, the `UpdateFrame()` function calls the `sprite::LevelDone()` function to tell the game engine that the level is finished.

Each time the game engine goes through its message loop, it tests to see if the level is done. If so, the game automatically calls the `DoLevelDone()` function for your game class. You can have the `DoLevelDone()` function do any processing you want done when a level is finished. [Listing 7.14](#) shows the `DoLevelDone()` function for the `ping` class.

### Listing 7.14. Time for a nap

```
1  bool ping::DoLevelDone()
2  {
3      ::Sleep(2000);
4      return (true);
5  }
```

When a level (match) finishes, the score is updated by the `UpdateFrame()` function and an additional score marker is displayed on the screen by the `RenderFrame()` function. The only thing that `DoLevelDone()` does when a level finishes is pause for 2,000 milliseconds (2 seconds). This gives the players a chance to get themselves ready for the next match.

`DoLevelDone()` pauses by calling the Windows `Sleep()` function. `Sleep()` takes an integer number of milliseconds as its only parameter. When 2 seconds expires, the `DoLevelDone()` function returns and the new match starts.

## Cleaning Up the Game

If a player wins, the `UpdateFrame()` function invokes the `game::GameOver()` function. This tells the game engine that the game is done. In response, the engine calls the `DoGameOver()` function for your game class. [Listing 7.15](#) provides the code for the `ping::DoGameOver()` function.

### Listing 7.15. Say bye-bye.

```
1  bool ping::DoGameOver()
2  {
3      bool noError = true;
4
5      if (player1Score >= 5)
6      {
7          theTrophy.X(areaOfPlay.right * 3/4);
8          theTrophy.Y(areaOfPlay.bottom/2);
9      }
10     else
11     {
12         theTrophy.X(areaOfPlay.right * 1/4);
13         theTrophy.Y(areaOfPlay.bottom/2);
14     }
15
16     theApp.BeginRenderNow();
17     theTrophy.Render();
18     theApp.EndRenderNow();
19
20     ::Sleep(4000);
21
22     game::DoGameOver();
23
24     return (noError);
25 }
```

At the end of a game, the `DoGameOver()` function uses the `if-else` statement on lines 5-14 to determine who the winner was. It sets the x and y coordinates of the trophy so that the trophy appears on the winner's side of the screen.

Next, `DoGameOver()` calls the `BeginRenderNow()` function. `BeginRenderNow()` is a function provided by the game engine's application object. It also has a corresponding

`EndRenderNow()` function, which is called on line 18 of [Listing 7.15](#). The `BeginRenderNow()` and `EndRenderNow()` functions provide you with a way of rendering to the screen even when your game is not in the main message processing loop. Your game should never call `BeginRenderNow()` or `EndRenderNow()` from the `RenderFrame()` function. There is no need, and it will make your game crash.

Good places to call the `BeginRenderNow()` and `EndRenderNow()` functions are in the `InitLevel()`, `DoLevelDone()`, and `DoGameOver()` functions. Every time your game calls `BeginRenderNow()`, it must also call `EndRenderNow()` when the rendering is done. If it does not, you'll find out quite soon because your game will crash.

In between `BeginRenderNow()` and `EndRenderNow()`, your game can do any rendering it needs to. You can have your game enter a loop that displays a long animation, plays music, or whatever else you would like. In the case of Ping, the `DoGameOver()` function invokes the trophy's `Render()` function to display the trophy. After it calls `EndRenderNow()`, `DoGameOver()` invokes the Windows `Sleep()` function to pause for four seconds.

The last thing that the `ping::DoGameOver()` function does is to call the `game::DoGameOver()` function to shut down the game. Your `DoGameOver()` function should always call the `game::DoGameOver()` function before it ends because `game::DoGameOver()` sends a message to Windows to tell it to end the program.

## Warning

If your `DoGameOver()` function doesn't call `game::DoGameOver()`, you'll have to write code to post a quit message to Windows yourself.



# Summary

This chapter presented the essentials of game program code design. In professional games, you define classes for each type of object you see on the screen. You implement each class in a .h and a .cpp file. The .h file contains the class definition and the inline member functions. The .cpp file contains the out-of-line member functions.

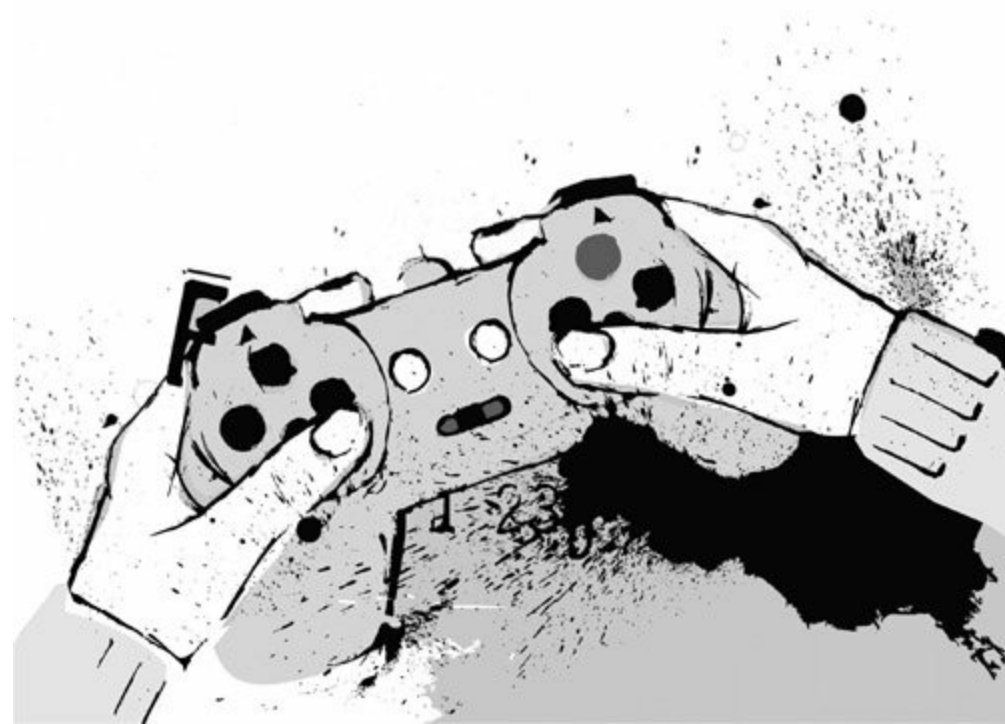
When speaking of the primary tasks that games perform, all game programs follow the same general design. They initialize the game, initialize a level, process messages in the main message loop, and update and render the frame. At the end of a level, they do any necessary level cleanup. At the end of the game, they perform whatever game cleanup is needed.

Unfortunately, Ping isn't a very exciting game because it has no sound or music. The next chapter demonstrates how to fix that.

# Chapter 8. Sound Effects and Music

In [chapter 7](#), "Game Program Structure," you implemented your own replica (named Ping) of the very first video game ever produced (which was called Pong). Like the first version of Pong that I played more than 30 years ago, Ping had no sound effects or music. Playing Ping is a very sterile experience.

While nothing can make a ping-pong simulator very exciting, adding sound effects and music to any game intensifies the experience of playing it. In fact, it is my opinion that sound effects and music are so critical to a game that poor sound effects and music can actually ruin an otherwise good game.



I want to emphasize that you do not have to be a musician in order to make games. There are ways of getting sound effects and music into a game even if your music skills are limited. Having said that, I also have to say that it helps a lot if you have a musician on the project when you're creating a game.

Adding music and sound effects to your game adds life to it in a way that few things do. With the techniques you'll learn in this chapter, you'll see that dealing with music is a lot of fun and not nearly as hard as you might think.

This chapter discusses how to put sound effects and music into your games. It also gives an overview of producing and recording music and sound effects. In addition, it tells you how to find inexpensive sound effects and music if you can't create them yourself.



# Sound Effects and Music Are Emotion

Years ago I used to play the game Descent. I happily wasted major amounts of time playing Descent 1 and 2, and I enjoyed every moment of it. However, when Descent 3 came out, I was bitterly disappointed. For a while I could not put my finger on why. The gameplay was very much the same. The graphics, which really didn't improve much with version 3, were nevertheless good. Nearly everything about version 3 of the game was as good or better than previous versions. At last, I figured out that there was a major difference in the sound. The music was not as good and the sound effects were downright wimpy. In Descent 1 and 2, the sounds of many of the weapons were thunderous. In Descent 3, most of the weapons sounded like popguns and cap pistols. For me, the lack of good sound effects and music so diminished the gameplay experience that I never finished Descent 3. I got about halfway through and lost interest. That is the power of sound effects and music in games.

Sound effects intensify the feeling of reality in games. In the real world, there are a myriad of sounds in even the quietest of spots. In the most soundproof rooms you can hear yourself breathe. If you move your foot across the floor, there's a sound. If it's quiet enough, you can even hear your own eyes blink.

Human beings expect to be immersed in a world of sound. If your game does not provide a world of sound, it doesn't feel real enough. As a result, players are not as immersed in your game as they otherwise could be.

Sounds cause emotion. Think how you would feel if you were in a dark alley and suddenly heard the heavy footfalls of two large men behind you. In a game, the sound of footsteps can cause the same emotional reaction in a player.

Music also causes emotion. In fact, good game designers know that music *is* emotion. Think about someone you dated who meant a lot to you. Did the two of you have a particular song that you thought of as "your" song? Does hearing that song still bring back memories of that special person? What do you feel when those memories come back? That's the power of music.

Next time you watch a movie or TV show in which the character gets into danger, listen to the music. Often, the change in the music is the first clue you have that danger is imminent. It causes an immediate shift in mood.

If your sound effects and music are well done, people won't even notice them. Although that sounds odd, it's true. The sounds and music that come from your

game will just be taken as part of the experience. Players won't even notice them. However, if they're poorly done, players will *really* notice that.

# Storing Sound Data

Both music and sound effects for games are stored in two basic types of files. The first is called [digitized sound](#). The second way to store sound is in MIDI files. This section examines both types of sound files.

## Digitized Sound

Digitized sound or music is recorded on the computer in a digital format. Digitized sound always sounds the same every time you play it back. That's what you want for your games, but it comes at a cost.

Digitized sound files contain a lot of information so they tend to be very large. Most people who record sounds and music use special compression techniques to make the files smaller. So when you record digitized sound, you can choose between compressed and uncompressed formats. Windows WAV files are an example of an uncompressed file format. WAV files result in high-quality sound, but take a lot of room on disks.

On the other hand, MP3 files are compressed. They take up much less room than WAV files. However, the sound quality is not quite as good. Nevertheless, the sound quality of MP3 files is good enough for the vast majority of game programmers. Almost every game stores its sound effects and music in MP3 files.

### Note

LlamaWorks2D can play music off CDs in just the same way it plays music from hard drives.

There is a way to get high-quality sound without requiring huge amounts of space on the player's hard drive: Your game can stream CD-quality sound from the game's CD or DVD.

### Factoid

CD-quality music is also called [Redbook audio](#).

When your game streams sound off a CD or DVD, you may not have to use as much compression for your sound files as you otherwise would. As a result, you can get higher-quality music. Streaming sound also takes very little of the computer's memory.

The problem with streaming sound is that getting music off a CD or DVD is *much* slower than playing sounds stored in memory. It's so slow, in fact, that it is not possible to use streaming sound for sound effects. However, it's often used for background music.

## Midi Files

In addition to digitized sound, another basic form for storing music exists. Instead of storing a recording of the sounds themselves, you can store a set of commands that computers can use to reproduce the sounds. The most common file format of this type is the Musical Instrument Digital Interface (MIDI) file.

Because you store only the commands needed to reproduce sound effects or music in MIDI files, they tend to be much smaller than any digitized sound format. There is one problem with MIDI files, however. The sound that is reproduced depends entirely on the quality and the settings of the sound card in the player's computer. That means your music and sound effects can sound different on different people's computers. As a result, most game programmers don't use MIDI files.

Using MIDI sound has the added advantage of [\*dynamic music generation\*](#). If your game generates music dynamically, it actually composes music as your game runs. Games that employ dynamic sound generation follow a set of rules for automatically composing music. These rules dictate the style, tempo, key, and other features of the music.

### Factoid

It is not uncommon for game developers to use MIDI files to produce music and sound effects with their own high-quality equipment. They then record the music and sound effects into a digital format such as MP3 for distribution with their games.

With dynamic music generation, games can play faster, more intense music as the action picks up and slower, more relaxed music as the game's pace slows down. For instance, in a two-player game in which the players battle each other, you as the game programmer don't know in advance when the action will be fast-paced. However, the game can figure it out from the number of shots being fired, the number of times players are actually hit, and the speed at which the characters are running. At those times, it can use dynamic music generation to compose and play fast, intense music. When no one is firing any weapons and everyone's sneaking slowly around, the game can play slow but scary music to indicate impending doom.

## Note

Microsoft's DirectX supports dynamic music generation that sounds nearly the same on all computers. If you want to incorporate dynamic music generation into your games, I recommend that you learn and use DirectX after you become a proficient game programmer.

## Sound Storage Techniques Compared

With all of these different method of doing audio in games, the question programmers often ask is "Which technique should I use?" As you might expect, the answer depends on your game. [Table 8.1](#) lists the options you have.

**Table 8.1. Comparing Sound Storage Techniques**

Storage Method	Pros	Cons
Uncompressed digitized audio in memory	High-quality sound, fast access, always sounds the same on all computers	Takes huge amounts of memory
Compressed digitized audio in memory	Good-quality sound, fast access, always sounds the same on all computers	Takes medium to large amounts of memory



Streaming digitized audio	High-quality sound, requires little memory, always sounds the same on all computers	Very slow access
MIDI audio	Takes small amounts of disk space and memory	Does not sound the same on all computers

---

You can see from [Table 8.1](#) that if you're creating a small, arcade-style game, then it's probably best to use compressed digitized sound for all music and sound effects and put them in memory. For larger games where the player encounters many different situations, you'll need a wide variety of music and sound effects. In such a case, it may be better to use compressed digitized music streamed from the CD or DVD for background music and compressed digitized sound effects stored in memory.

# Sound Effects in LlamaWorks2D

When I first started writing games many years ago for MS-DOS, the only way to make noises was to use the computer's speaker. It was not a particularly straightforward task. If your game played music, and most did, the music had a "dinkity-dink" quality to the sound.

These days, computers come with wonderful sound cards that can produce excellent music and sound effects. However, for the most part, using them is still not particularly easy.

You learned in previous chapters that the computer industry has developed graphic libraries to simplify the process of creating animated graphics on computer screens. The industry has done the same for sounds and music. The two most commonly used libraries for doing sound and music are Microsoft's DirectX Audio and OpenAL. Of the two, DirectX Audio is more powerful and flexible. However, it is also much harder to use than OpenAL. Therefore, the LlamaWorks2D game engine uses OpenAL for both music and sound effects.

## Factoid

Microsoft used to distribute its audio libraries as two separate components, called DirectSound and DirectMusic. As you might expect from the names, DirectSound was for sound effects and DirectMusic was for music. These days, Microsoft has combined the two components into a single library called DirectX Audio.

The instructions for installing OpenAL are in the Introduction. If you have not already installed OpenAL, please do so now.

LlamaWorks2D uses the same philosophy for sounds and music that it does for graphics. Just as graphics are all represented by the `sprite` class, all sounds, whether they are sound effects or music, are represented by the `sound` class. To play sound effects or music, your game declares a variable of type `sound`.

[Listing 8.1](#) shows the class definition of the LlamaWorks2D's `sound` class.

## Listing 8.1. The `sound` class

```

1 class sound
2 {
3 public:
4     sound();
5     ~sound();
6
7     bool LoadWAV(
8         std::string fileName,
9         bool continuousLoop = false);
10    bool Play();
11    bool Pause();
12    bool Stop();
13    bool Rewind();
14
15 private:
16     ALuint audioBuffer;
17     ALuint audioSource;
18 };

```

The `sound` class has a constructor and destructor that take care of the OpenAL setup and cleanup. It also has two data members, as shown on lines 16-17 of [Listing 8.1](#). These data members contain the information that OpenAL needs to produce sounds.

## Note

LlamaWorks2D can only load WAV files for sound effects and music. If you decide you need the ability to use MP3 files, you can purchase LlamaWorks2D XE at [www.screamingllamasoftware.com](http://www.screamingllamasoftware.com) for a modest fee. LlamaWorks2D XE can utilize MP3 files.

The member functions on lines 7-13 enable your game to load sound data into the `sound` variable, play the sound, pause it, stop it, or rewind it.

You may notice something unusual about the second parameter to the `LoadWAV()` function. As you can see from line 9, the prototype sets the parameter to `false`. This is a C++ feature called a [default parameter](#). If you do not provide a value for the second parameter for the `LoadWAV()` function, the program automatically assigns it the value `false`. The result of using a default parameter is that you can leave off the parameter when you call the function. Therefore, some calls to `LoadWAV()` can have two parameters and some only one. The compiler will not complain. If a game only passes one parameter to the `LoadWAV()` function, the

program sets `continuousLoop` to `false`.

## Loading Sounds

To demonstrate how to use sounds in games, we'll put a sound effect and two short pieces of music into Ping. Let's start by examining how to load sounds.

The new version of Ping declares its sound variables in its `game` class, as [Listing 8.2](#) demonstrates.

### Listing 8.2. Ping with sounds

```
1 class ping : public game
2 {
3 public:
4     bool OnAppLoad();
5     bool InitGame();
6     bool InitLevel();
7     bool UpdateFrame();
8     bool RenderFrame();
9
10    ATTACH_MESSAGE_MAP;
11
12    bool OnKeyDown(
13        keyboard_input_message &theMessage);
14
15    bool BallHitPaddle();
16
17    void SetBallStartPosition();
18    void SetBallStartDirection();
19
20    bool DoLevelDone();
21    bool DoGameOver();
22
23 private:
24     ball theBall;
25     paddle leftPaddle;
26     paddle rightPaddle;
27     sprite theTrophy;
28
29     sprite scoreMarker;
30     int player1Score;
31     int player2Score;
32
33     bitmap_region areaOfPlay;
34
35     sound ballBounceSound;
36     sound gameWonMusic;
37 };
```

The `ping` class declares two data members of type `sound` on lines 3536. If your program contains a variable of type `sound`, it can call the `sound::LoadWAV()` function to load WAV file data into the `sound` variable. As with loading bitmaps, loading sounds takes time. Your game should not load sounds while the player is in a level. Instead, it should load sounds in an initialization function. The new version of Ping loads its sounds in the `InitGame()` function, which is given in [Listing 8.3](#).

### Listing 8.3. Loading sounds in `InitGame()`

```
1  bool ping::InitGame()
2  {
3      // The InitGame() function is really 111 lines long.
4      // I've omitted most of it because it
5      // repeats what you saw in Chapter 7.
6      // The entire InitGame() function is in program
7      // 8.1 on the CD. The statements below demonstrate
8      // how to load and initialize sounds.
9      if (initOK)
10     {
11         initOK = ballBounceSound.LoadWAV("BallBounce.wav");
12     }
13
14     if (initOK)
15     {
16         initOK = gameWonMusic.LoadWAV("GameWon.wav");
17     }
18
19     if (initOK)
20     {
21         InitLevel();
22     }
23
24     return (initOK);
25 }
26 }
```

As the comments in [Listing 8.3](#) indicate, I've omitted most of the `InitGame()` function because it's just a repeat of what's found in [chapter 7](#). If you look on the CD in the folder `Source\Chapter08\Prog_08_01`, you'll find the entire `InitGame()` function in the file `Ping.cpp`.

### Warning

The programs on the CD are numbered independently of the listings in the corresponding chapters. In earlier chapters, there was usually one program per listing so the numbering matched.

However, from this chapter forward, that will rarely be the case. One program is often used for several listings. Program 8.1 is used for [Listings 8.1](#) through 8.5. For [Listings 8.6](#) through 8.9, see Program 8.2. [Listing 8.10](#) is taken from Program 8.3.

Although it's not shown here, the `InitGame()` function does all the tasks it did in past versions. It loads the bitmap images for the ball, the paddles, the score marker, and the trophy. It sets the bounding rectangles and other initialization information. If all of that completes successfully, `InitGame()` uses the `ballBounceSound` member of the `ping` class to call the `LoadWAV()` function. `LoadWAV()` loads the sound effect from the file `BallBounce.wav` and stores the sound data in a buffer. It also prepares the sound to be played. `LlamaWorks2D` takes care of managing the sound buffer and other related tasks, so all you have to do is tell `ballBounceSound` when to play itself.

On line 16 of [Listing 8.3](#), the `InitGame()` function also loads a sound from the file `GameWon.wav`. This file contains music that the game plays when a player wins. As you can see from the statements on lines 11 and 16, there is absolutely no difference between loading a sound effect and loading music. The game engine handles them both in exactly the same manner.

Another thing to notice before we move on is that the `InitGame()` function passed only one parameter in its calls to `LoadWAV()`. Recall that this is allowed because the last parameter to the `LoadWAV()` function has a default value. If the game does not need to change the default value for the second parameter, it can omit the value entirely.

## Playing Sounds

Playing a sound generally occurs as soon as the event that generates the sound occurs. For example, `Ping` plays a sound for the ball bounce at the point in the code where the bounce occurs in the `UpdateFrame()` function. It may seem as if I'm stating the obvious here, but I have seen programmers new to games try to wait to play sounds until the `RenderFrame()` function. They think the sound will synchronize better with the graphics. On the surface, that would seem to be a sensible approach; however, it complicates life unnecessarily.

To understand why delaying sound processing until the `RenderFrame()` function is

not a good idea, let's first examine how Ping handles the ball bounce sound. [Listing 8.4](#) shows Ping's `UpdateFrame()` function.

## Listing 8.4. The `UpdateFrame()` function for the ping class

```
1  bool ping::UpdateFrame()
2  {
3      bool updateOK = true;
4
5      if (!GameOver())
6      {
7          theBall.Move();
8          rightPaddle.Move();
9          leftPaddle.Move();
10
11         if (BallHitPaddle())
12         {
13             theBall.Bounce(ball::X_DIRECTION);
14             ballBounceSound.Play();
15         }
16         // The majority of this function is omitted also.
17         // If you want to see the rest of it, you'll find
18         // it in sample program 8.1.
19
20     }
21
22     return (updateOK);
23 }
```

As with [Listing 8.3](#), I've omitted most of the function so that we can focus on handling sounds. If you look at lines 11-15, you'll see that the `UpdateFrame()` function calls the `sound::Play()` function to play the bounce sound. It makes this call when it detects that the ball has hit the paddle. If Ping were to wait until the `RenderFrame()` function to play the sound, it would have to detect the collision between the paddle and the ball again. That requires *two* calls to the `BallHitPaddle()` function in every frame, which is a waste of time. In a complex game with many sounds, it would most likely produce a slowdown in the frame rate. It's better for your game to play the sound as soon as it discovers the sound needs to be played. Experience shows that it does not cause a loss of synchronization with the action.

The call to the `sound::Play()` function in [Listing 8.4](#) plays a sound effect. Playing music is no different, as [Listing 8.5](#) demonstrates.

## Listing 8.5. Playing music when a player wins

```

1  bool ping::DoGameOver()
2  {
3      bool noError = true;
4
5      if (player1Score >= 5)
6      {
7          theTrophy.X(areaOfPlay.right * 3/4);
8          theTrophy.Y(areaOfPlay.bottom/2);
9      }
10     else
11     {
12         theTrophy.X(areaOfPlay.right * 1/4);
13         theTrophy.Y(areaOfPlay.bottom/2);
14     }
15
16     gameWonMusic.Play();
17     theApp.BeginRenderNow();
18     theTrophy.Render();
19     theApp.EndRenderNow();
20
21     ::Sleep(27000);
22
23     game::DoGameOver();
24
25     return (noError);
26 }

```

Line 16 of [Listing 8.5](#) shows that your game can also play music by calling `sound::Play()`. The music plays until the end of the file and then quits. The `DoGameOver()` function calls the Windows `Sleep()` function to pause for 27 seconds, which is how long it takes to play the triumphal music for the winner.

## Adding Background Music

Many games have looping background music. That is, they play a song over and over while the player is in a level. When the level ends, they usually end the music as well. LlamaWorks2D enables you to do the same thing in your games. To demonstrate how this is done, we'll add looping background music to Ping. The first thing to do is to add another data member to the `game` class, as shown in [Listing 8.6](#).

### Listing 8.6. The `ping` class with a member for background music

```

1  class ping : public game

```



```

2  {
3  public:
4      bool OnAppLoad();
5      bool InitGame();
6      bool InitLevel();
7      bool UpdateFrame();
8      bool RenderFrame();
9
10     ATTACH_MESSAGE_MAP;
11
12     bool OnKeyDown(
13         keyboard_input_message &theMessage);
14
15     bool BallHitPaddle();
16
17     void SetBallStartPosition();
18     void SetBallStartDirection();
19
20     bool DoLevelDone();
21     bool DoGameOver();
22
23 private:
24     ball theBall;
25     paddle leftPaddle;
26     paddle rightPaddle;
27     sprite theTrophy;
28
29     sprite scoreMarker;
30     int player1Score;
31     int player2Score;
32
33     bitmap_region areaOfPlay;
34
35     sound ballBounceSound;
36     sound backgroundMusic;
37     sound gameWonMusic;
38 };

```

Line 36 of [Listing 8.6](#) declares a data member of type `sound` for the background music. This class comes from program 8.2 on the CD.

To get a piece of music to loop over and over, your game must set the second parameter of the `sound::LoadWAV()` function to `true`. [Listing 8.7](#) gives a version of the `InitGame()` function that loads a piece of looping music.

## Listing 8.7. Making music loop

```

1  bool ping::InitGame()
2  {
3      // Most of this function has been omitted as well. To
4      // see the rest of it, look in program 8.2 on the CD.

```

```

5   if (initOK)
6   {
7       initOK = backgroundMusic.LoadWAV("BackMusic. wav",true);
8   }
9
10  if (initOK)
11  {
12      initOK = ballBounceSound.LoadWAV("BallBounce. wav");
13  }
14
15  if (initOK)
16  {
17      initOK = gameWonMusic.LoadWAV("GameWon.wav");
18  }
19
20
21  if (initOK)
22  {
23      InitLevel();
24  }
25
26  return (initOK);
27 }

```

As with previous listings, [Listing 8.7](#) omits a large section of the `InitGame()` function. You can find the entire function in the file `Ping.cpp` on the CD, in the folder `Source\Chapter08\Prog_08_02`.

If you look on line 7 of [Listing 8.7](#), you'll see that `InitGame()` loads the background music by calling `LoadWAV()`. As stated previously, it sets the second parameter of `LoadWAV()` to `true` to tell `LlamaWorks2D` that the song will loop when played.

## Warning

`LlamaWorks2D` does not provide a way to turn song looping on and off while the game is in progress. You can set it only when you load a song file.

After a file is loaded as a looping song, your game starts it playing by calling the `Play()` function. This is usually done at the beginning of the level, which is what `Ping` does. [Listing 8.8](#) demonstrates this.

## Listing 8.8. Starting a looping song

```

1  bool ping::InitLevel()
2  {
3      bool initOK = true;
4      bitmap_region boundingRect =
5          leftPaddle.BoundingRectangle();
6
7      leftPaddle.X(5);
8      int paddleHeight = boundingRect.bottom - boundingRect.top;
9      int initialY =
10         ((areaOfPlay.bottom - areaOfPlay.top)/2) -
11         (paddleHeight/2);
12     leftPaddle.Y(initialY);
13
14     int bitmapWidth = rightPaddle.BitmapWidth();
15     int initialX = areaOfPlay.right - bitmapWidth - 5;
16     rightPaddle.X(initialX);
17     rightPaddle.Y(initialY);
18
19     SetBallStartPosition();
20     SetBallStartDirection();
21
22     backgroundMusic.Play();
23
24     LevelDone(false);
25     return (initOK);
26 }

```

This version of `InitLevel()`, which is also from Program 8.2 on the CD, starts the background music playing each time a level is initialized. You can see the call to the `Play()` function on line 22. That's all it takes. The song will play over and over until the level ends.

When Ping finishes a level, the game stops the background music and rewinds it, as shown in [Listing 8.9](#).

### Listing 8.9. Stopping and rewinding music

```

1  bool ping::DoLevelDone()
2  {
3      backgroundMusic.Stop();
4      backgroundMusic.Rewind();
5      ::Sleep(2000);
6      return (true);
7  }

```

The game engine automatically calls `DoLevelDone()` as soon as a level finishes. The

`DoLevelDone()` function calls the `sound::Stop()` function on `backgroundMusic` to stop the background music from playing. It then calls `sound::Rewind()` to reset play to the beginning of the song. When the game begins the next level and plays the background music, the song starts over from its beginning.

If your game uses background music in the same manner as Ping, other sounds can occur while the background music plays. The fact that the game plays multiple sounds makes no difference at all to the game engine. All of the sounds are automatically mixed together to play at the same time. Sound mixing is not something you need to worry about or pay attention to. Just play sounds whenever you need to. LlamaWorks2D plays multiple sounds properly.

## Controlling the Volume

Almost all modern games provide a way to set the sound volume. In fact, many games enable you to separately control the volume of various types of sounds. Most games that do this feature individual volume controls for special effects, music, and other sounds. For instance, I have a flight combat simulator game in which the commanding officer assigning the missions to the player often shouts out instructions during the mission in a (stereotypical) grizzly soldier voice. I find it extremely annoying and unnecessary because the same instructions are printed along the bottom of the screen. Fortunately, the game has a volume slider in its setup screen that enables me to completely turn off that sound.

### Tip

I strongly recommend that you provide a way for players to set the volume of your game. It's even better to let them set the volumes of different types of sounds, such as sound effects, background music, and so forth.

LlamaWorks2D makes it easy for your game to control the volume of sounds pro-grammatically. The `sound` class provides a `Gain()` function for setting the volume.

Sound engineers call the sound volume the *gain*. Your game can set a sound's gain at any time after it is loaded.

[Listing 8.10](#) gives a version of Ping's `InitGame()` function that sets the gain of the sounds it loads. Once again, I've omitted the parts of the `InitGame()` function that don't deal with sounds. If you want to see the entire function, you'll find it on the CD in the folder `Source\Chapter08\Prog_08_03` in the file `Ping.cpp`.

## Listing 8.10. Setting the gain

```
1  bool ping::InitGame()
2  {
3
4      // Most of this function has been omitted. To see the
5      // rest of it, look in program 8.3 on the CD.
6      if (initOK)
7      {
8          initOK = backgroundMusic.LoadWAV("BackMusic. wav", true);
9      }
10
11     if (initOK)
12     {
13         backgroundMusic.Gain(0.4f);
14         initOK = ballBounceSound.LoadWAV("BallBounce. wav");
15     }
16
17     if (initOK)
18     {
19         ballBounceSound.Gain(0.4f);
20         initOK = gameWonMusic.LoadWAV("GameWon.wav");
21     }
22
23
24     if (initOK)
25     {
26         gameWonMusic.Gain(0.4f);
27         InitLevel();
28     }
29
30     return (initOK);
31 }
```

This version of the `InitGame()` function sets the volume of the game's sounds by calling the `sound::Gain()` function after it loads each sound. The gain is always a floating point number between 0.0 and 1.0, inclusive. As you might expect, 0.0 means no sound, 1.0 is the maximum volume, 0.5 is half volume, 0.75 is 3/4 volume, and so on.

### Note

Setting the gain on individual sounds does not change the volume

of the player's computer; it only changes the volume of that particular sound.

# Noise, Sweet Noise

Now that you know how to play sound effects, you have to somehow come up with sound effects to play. There are two basic approaches that most game programmers use to obtain sound effects. They either find royalty-free effects available commercially or they generate their own.

## Finding Sound Effects

The easiest way to find sound effects is to search the Web. Many sites offer MP3 files containing sound effects; however, a lot of them are copied illegally. Enthusiastic fans often copy sounds from movies, TV shows, cartoons, and similar sources. It may be tempting to use these sounds because they're offered for free. Don't give in to that temptation! Those are copyrighted sounds. If your game becomes popular, it can attract the attention of the copyright holder. That in turn can lead to nasty letters from lawyers, lawsuits, and a host of legal woes.

### **Warning**

Do not use copyrighted sound effects in your games. This includes sounds from movies, TV shows, and other games.

When you're using your favorite search engine to find sound effects on the Web, use phrases such as "royalty free sound effects" as your search keywords. If you include the term "royalty free," you'll find collections of sound effects that very skilled sound effects experts produce and sell. You'll typically pay from \$25 to \$150, depending on the size of the collection you're ordering, for CDs containing hundreds or thousands of sounds.

### **Tip**

Always ensure that sound effect collections you buy are royalty free.

It's important to ensure that the sound effect collections you order specify that you do not have to pay royalties for commercial uses of the sound effects. If the collection is not specified as royalty free, don't use it. If the collection requires you to pay royalties, you'll have to pay some money to the author of the collection for each copy of your game that you sell. This can really cut into your profits.

There's one more thing to be careful of when you're buying collections of sound effects: Some sound effect collections specify that they are for noncommercial use only. You can't legally use them in games that you sell.

## **Making your Own Sound Effects**

Sound effects experts commonly use a few different techniques for creating their own sound effects. Most of the techniques they employ are available to you as well for not a lot of money.

### **Using Real-World Sounds**

One technique that audio experts use is to record sounds they find in the real world and then alter those sounds with audio editor software. Audio editors enable you to cut, paste, and combine sounds. They also let you change the pitch of recorded sounds, slow them down, speed them up, and apply many other sound transformations.

#### **Note**

If you want to be able to save MP3 files with Audacity, you must go to the project's download page at <http://audacity.sourceforge.net/download/windows> and follow the instructions for downloading the LAME MP3 encoder.

Professional audio editing software can get very expensive very quickly. However, to get you started, I've included an open source audio editor called Audacity on the CD with this book; it's in the folder Tools\Audacity. The home page for the Audacity software is at <http://audacity.sourceforge.net>.



## **Generating Sounds on an Electronic Keyboard**

Another approach to making sound effects is to generate them with an electronic keyboard. Many inexpensive musical keyboards come with a standard set of sound effects. You can record those effects by connecting a wire from the keyboard's headphone output jack (or from the Line Out jack) to the microphone jack (or the Line In jack) on your computer's sound card. Use a recording program, such as Windows Recorder, to record the sound. You can also use Audacity for sound recording. After you record the sound, you can use Audacity, or another audio editor, to alter it in any way you want.

## **Using Sound Effects Generation Software**

A third technique for making sound effects is to use sound effects generation software. A good way to find sound effects generation software that meets your needs is to read the software reviews at [emusician.com](http://emusician.com). Sound effects generation programs typically generate effects using MIDI commands. As a result, most game developers digitally record the sound effects and distribute them as MP3 files.

# Play That Funky Music, Geek Boy

To obtain music for their games, game developers usually either find royalty-free music or make their own. It may be surprising, but both of these options work well even for game developers who are not professional-level musicians.

## Finding Music

As with sound effects, it's easy to find music on the Web. And like sound effects, much of the music floating around the Internet is illegal. Game developers should avoid illegal music like the plague. Using it can get them sued into oblivion. It's far better to find one of the many Web sites selling royalty-free music instead. To find collections of music, search the Web using phrases such as "royalty free music" and "music for games".

### Note

This is a shameless bit of self-promotion, but you can find a growing collection of music for games at my Web site, [screamingllamasoftware.com](http://screamingllamasoftware.com). All of it is royalty free and available for commercial use in your games.

As with sound effects, music collections are sometimes royalty free but not licensed for commercial use. Make sure that any music collection that you buy is both royalty free and can be used in commercial applications such as games.

## Making Your Own Music

There are a variety of ways to produce music for your games. Some require advanced musicianship while others are accessible to nonmusicians. Let's examine each of them, from easiest to hardest.

### Band-in-a-Box Programs

For nonmusicians, the easiest way to create original music for games is to use

a music-generation program. These so-called band-in-a-box programs can completely generate original music for you while requiring you to do little more than point and click with your mouse. If you know what type of music you want in your game, you can create it with a music-generation program.

Several music generation programs are on the market. The most popular is actually named Band In A Box; it's produced by PG Music ([www.pgmusic.com](http://www.pgmusic.com)) and costs about \$90.

Music-generation programs work by following a set of rules for music composition that actual composers use. They apply these rules to a style, such as reggae or heavy metal, using a particular band. A band is defined as a collection of instruments.

Most band-in-a-box programs generate the music as a set of MIDI commands, which you can play and edit. Some use sampled sounds. Programs that generate MIDI commands are more flexible in the way they create music. Keep in mind, though, that they are limited by the quality of your sound card. A good sound card produces more realistic and professional-sounding music.

On the other hand, some music-generation programs use small samples of digitized sound to make the instrument sounds. Sampled instruments sound much more realistic. However, programs that use sampled sounds are often less flexible than those that use MIDI commands to produce instrument sounds.

## **Tip**

Whether your music-generation program uses MIDI commands or digital samples to generate music, you should digitally record the final song as an MP3 file.

Which type of music-generation program you use depends on your personal preferences and the quality of your hardware. If your sound card isn't that great, it's probably better to use sampled instrument sounds. If your sound card is one of the best, a MIDI-based program will provide you with the best options for music generation.

## **Loop-Based Programs**

Perhaps you know the basics of music production because you play the piano or guitar. If so, you should consider using a loop-based music-generation program. Loop-based programs enable you to define repeating patterns of music and use them throughout a song. You build entire songs from collections of repeating patterns of sound.

If you think about it, most songs are built from repeating patterns. In particular, the drums repeat a rhythmic pattern throughout most of a song. The bass, which is often used to reinforce the rhythm of a song, is played essentially the same way. Throughout a song, the bass plays a few repeating rhythmic patterns. Rhythm or background guitars also play their chord progressions in repeating patterns. In fact, most songs are built from 3 to 4 repeating patterns for each instrument.

Loop-based software enables you to exploit the repeating patterns of music by defining the patterns, selecting a song's instruments, and creating patterns for each of them. You then loop the patterns by specifying how many times the pattern repeats before the instrument either goes quiet or another pattern starts. Most of them let you enter the music pattern with an electronic musical keyboard. If you don't have one connected to your computer, you can also use your regular keyboard and mouse.

Probably the best loop-based music-generation program I've found is FruityLoops, by Image-Line Software ([www.flstudio.com](http://www.flstudio.com)). At the entry level, they offer Fruity Loops Express for about \$50. Their top-of-the-line product is called FL Studio. I've provided a demo version of FL Studio for you on the CD; you'll find it in the folder Tools\FLStudio.

The nice thing about loop-based programs such as FL Studio is that they offer more than just loop-based music creation. They are more like a PC-based music studio that can be used for generating, recording, mixing, and mastering both music and sound effects. Programs such as FL Studio support both sampled and MIDI instrument sounds, and enable you to save your work as either WAV or MP3 files.

## **MIDI Instruments**

If you're proficient on electronic music keyboards, you can move up a step in music creation. Most music keyboards can connect to a computer through a MIDI port. Increasingly, they simulate a MIDI connection through a universal serial bus (USB) port. Either way, they can send MIDI commands to your computer.

With MIDI instruments, you can make your instrument sound like anything you have a MIDI definition for. In other words, your MIDI keyboard can sound like a rock organ, guitar, drum set, or choir. It all depends on what instrument sound you select. The standard MIDI set that accompanies all MIDI devices contains sound definitions for many common musical instruments. Good keyboards often contain extended MIDI definitions, called *patches*, that provide you with a much wider range of instrument sounds than the standard MIDI set. The software that comes with MIDI keyboards generally enables you to select additional custom patches.

## **Note**

There are some programs around that convert a digital recording of one guitar at a time into MIDI commands. I haven't found one that works very well.

Guitarists often ask if there's a MIDI option for them as well. There are actually two options. The first is to buy a MIDI pickup that mounts on your guitar and an accompanying synthesizer. Look to spend at least \$1,000 for a decent pickup and synthesizer.

The second, less expensive option for guitarists wanting to do MIDI music is very hard to find these days. In the mid-1990s a company called GVOX made a special pickup for guitars. You could attach the GVOX pickup bar on your guitar to connect your guitar to a serial port. The software enabled you to convert the music you played into MIDI commands. It was a very good solution for guitarists on a shoestring budget as it only cost \$100 originally. Unfortunately, GVOX stopped producing hardware in the late 1990s. Nowadays, it only makes software. If you can find the increasingly rare GVOX hardware, you may be able to do MIDI with your guitar very inexpensively.

## **Tip**

If you do manage to find GVOX hardware, you'll see that it does great when you're just starting out. However, its limitations mean that you will probably have to invest in a MIDI pickup/synthesizer set eventually.

Drummers can also get into the MIDI act fairly inexpensively. Starter MIDI drum pads can sell for as little as \$70. Of course, you'll want to invest in better equipment as your games begin to sell.

## **Other Instruments and Tools**

You do not have to use MIDI instruments to record your music. If you have an electric guitar, you can connect its output directly to the input of your computer's sound card. The same is true for electronic music keyboards and electric bases.

For other instruments, and for voice input, you can buy an adapter that enables you to connect a microphone to your sound card. Note that I am not talking about the cheap microphones that often ship with computers; you need a microphone designed for use by musicians. Good starter microphones cost around \$100. You can get better sound by moving up to more expensive mics.

If you're recording music and sound effects, then I strongly recommend you check out professional music studio software. The industry standard is a program called Pro Tools by Digidesign ([www.digidesign.com](http://www.digidesign.com)). The Digidesign Web site provides a free, scaled-down version of its software at [www.digidesign.com/ptfree/](http://www.digidesign.com/ptfree/).

Pro Tools basically does everything. You can use it as an audio editor, loop-based music generator, or sound effects generator. It is a professional-level tool for recording, mixing, and mastering music.

# Summary

Good sound effects and music are critical to the success of games. They evoke emotion and intensify the experience of playing the game. LlamaWorks2D makes it easy to play sounds in games by providing a `sound` class. With the `sound` class, your game can play background music and multiple simultaneous sound effects.

When you need sound effects or music, you can either buy them or make them yourself. If you buy them, you need to make sure you're buying royalty-free sound effects or music and that they are licensed for commercial uses. If, on the other hand, you're making your own sound effects and music, you need the proper tools. For sound effects, you need audio-editing software such as Audacity, which is provided on the CD. You may also need to invest in sound effectsgeneration software.

To make your own music, you can use music-generation programs, such as Band In A Box, and loop-based music software like FL Studio. If you're a musician, you can use MIDI instruments such as keyboards, guitars, and drums. By connecting a good microphone to your computer, you can also record yourself playing any instrument.

# Part 4: Graduating to Better C++

[Chapter 9. Floating-Point Math in C++](#)

[Chapter 10. Arrays](#)

[Chapter 11. Pointers](#)

[Chapter 12. File Input and Output](#)

[Chapter 13. Moving into Real Game Development](#)

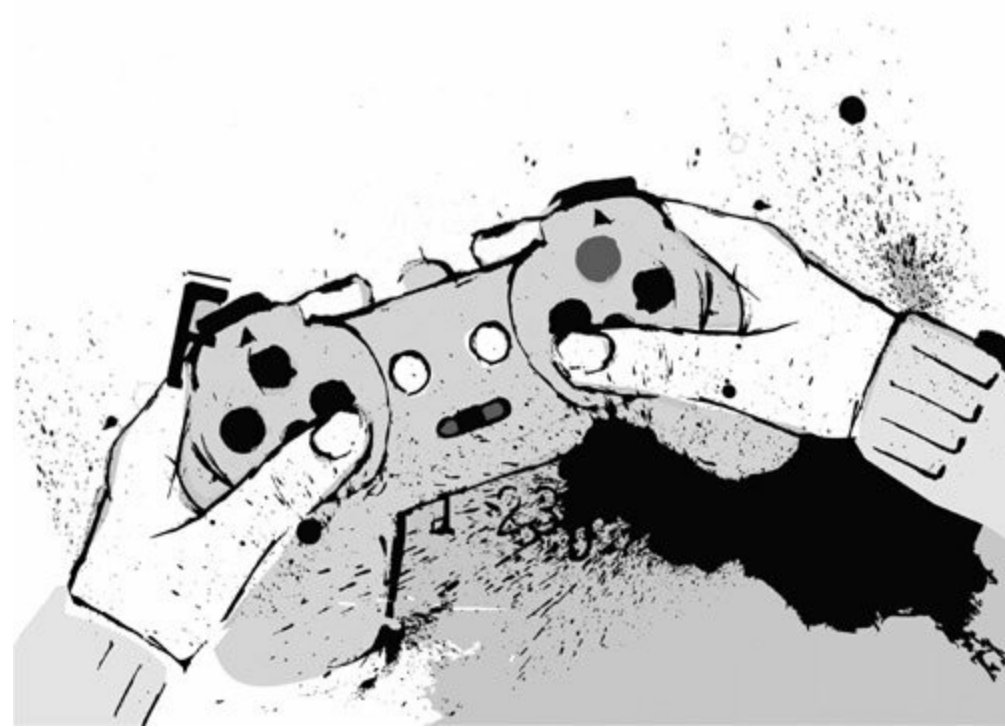


# Chapter 9. Floating-Point Math in C++

This chapter introduces floating-point math.

Oh, No! Not MATH!

Yes, it's true. A certain amount of math is involved in games. For many types of games, the math you have to do is very straightforward and minimal. So if you're not a student of advanced math, don't worry. Most of the math you'll do is just addition, subtraction, multiplication, and division. Having said that, however, you can't avoid floating-point math if you're going to become a game programmer. Floating-point numbers and floating-point math are used in physics. In previous chapters, we simulated the movement of a ball in the Ping game by using just integers. However, the movement of that ball is not very complex. In fact, it's not actually all that realistic. You can get more realistic results with floating-point math.



Previous chapters mentioned that floating-point numbers are numbers that contain a fractional part. However, that is only the most basic definition of floating-point numbers. You must know how C++ implements floating-point numbers to use them well and to avoid the problems you can have with them.

This chapter provides a fairly detailed overview of floating-point numbers in C++. It demonstrates their use by showing you how to do realistic (but surprisingly simple) physics with floating-point numbers.

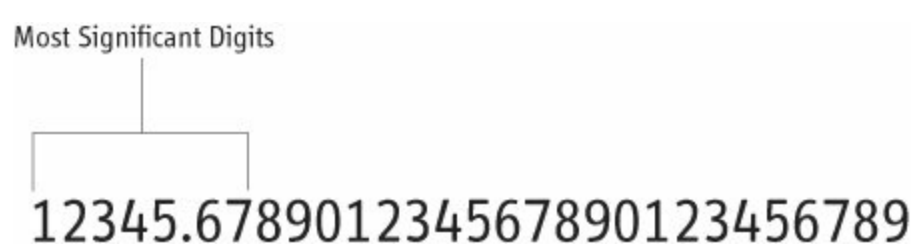


# Getting into the Guts of Floating-Point Numbers

By definition, floating-point numbers have fractional parts. When a program stores a floating-point number in a variable, it actually stores two pieces of information; the [significant digits](#) and the [exponent](#).

The significant digits in a floating-point number are the digits of the number that most significantly affect its value. For example, the number in [Figure 9.1](#) contains a lot of digits. However, not all of them are significant.

**Figure 9.1. The significant digits of a floating-point number.**



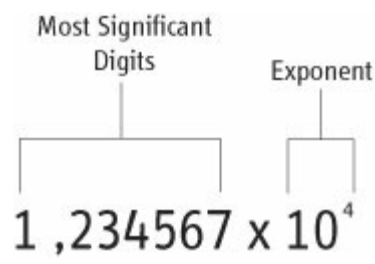
As [Figure 9.1](#) indicates, the leftmost seven digits of the floating-point number are the most significant. Anything to the right of those digits doesn't affect the value of the number much. This can lead to both important advantages and important problems. I'll talk about why in just a few moments.

## Note

Seven significant digits is typical on most personal computers.

As I already mentioned, floating-point numbers store the most significant digits and an exponent. [Figure 9.2](#) illustrates how programs would store the number in [Figure 9.1](#).

**Figure 9.2. The most significant digits and an exponent.**



When you multiply a number by a power of 10, you move the decimal point. Multiplying any number by  $10^4$  moves the decimal point right four places. As a result, the number  $1.234567 \times 10^4$  is the same as 12345.67.

If we take the leftmost seven digits of the number in [Figure 9.1](#) as the most significant, then 12345.67 is approximately the same as 12345.678901234567890123456789. The difference is small. To make things easy for the computer, the decimal point is always assumed to be after the leftmost digit, as shown in [Figure 9.2](#). To put into the correct place, you multiply it by a power of 10. In the case of [Figure 9.2](#), the number must be multiplied by  $10^4$  to get the decimal point into the correct spot. So when a computer stores a floating-point number, it stores the significant digits, in this case 1.234567, and the exponent, which is  $10^4$  in this example.

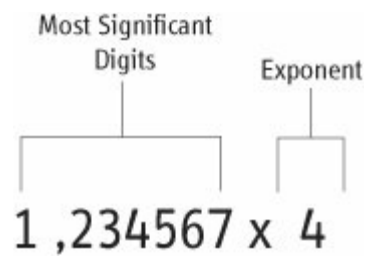
Okay, I lied.

## Note

Multiplying by a negative power of 10 moves the decimal point to the left.

There is one optimization that the computer makes when it stores floating-point numbers. If both we and the computer know that we're always storing the significant digits and a power of 10, then we really don't need to store the 10. Instead, the computer can just store the exponent or power that the 10 is raised to. In the case of [Figure 9.2](#), the computer wouldn't store  $10^4$ ; it would just store the 4. In reality, it's the 4 that's the exponent, and the 10 is just assumed. That's why we say that the computer stores the most significant digits and an exponent. [Figure 9.3](#) shows what *really* gets stored in a floating-point variable.

## Figure 9.3. The truth about floating-point numbers.



When the computer uses the number in [Figure 9.3](#), it understands what the significant digits are for and it knows that the exponent is a power of 10. It also understands that in order to use the floating-point number, it must raise 10 by the exponent and then multiply the result by the significant digits. It does all of this automatically for you whenever you use floating-point numbers.

You may be wondering why you have to understand how C++ implements floating-point numbers. The answer is that this knowledge is necessary to avoid potential problems that can crop up when using floating-point numbers. To find out what they are, keep reading.

### Note

Of course I'm using the words *understands* and *knows* in the preceding paragraph figuratively. Computers, in spite of what we see in the movies, are incapable of understanding or knowing anything.

## Floating-Point Numbers and Precision Errors

If computers don't store floating-point numbers exactly, there's a potential problem. Let's go back to the number 12345.678901234567890123456789. If you declare a variable of type `float` in your program, the program stores 12345.678901234567890123456789 as 1.234567 and 4. It throws away all but the seven most significant digits. What if you need those digits for your calculation to be correct?

This type of problem is called a [precision error](#). It's possible for floating-point variables to lack the precision needed to perform the calculation correctly. In other words, it's possible that the digits are not significant to the computer but are *very* significant to you.

Precision errors can also occur if the significant digits are too far apart. Take the number 900000.00000012. If your game stores that number in a variable of type `float`, it will contain 9.0 and 5. That's  $9.0 \times 10^5$ . The .00000012 got thrown away.

How can you fix precision errors?

The answer is that you can use a bigger data type. For example, on most personal computers, the `float` data type currently contains about seven significant digits. If you need more significant digits, then you can use a `double` instead. The C++ specification states that a `double` is no shorter than a `float`. Usually, it is implemented as twice the number of bits as a `float`. If so, a variable of type `double` contains about 15 significant digits. I doubt you'll ever need anything more precise than that for your games.

## Warning

Using a larger data type means that your program performs calculations more slowly and uses more memory. However, in most instances, you will probably not be able to notice the difference.

## Floating-Point Numbers and Rounding Errors

If you divide 2.0/3.0, which is a floating-point calculation, what do you get?

You and I know the answer is 0.6666666666666666, with the 6's repeating infinitely. However, computers don't have a way of storing an infinite number of 6's. When the computer has to represent an infinite number of floating-point digits in a fixed amount of memory, it rounds. As a result, 0.6666666666666666 becomes 0.6666667.

Depending on the calculations your game is doing, rounding can cause errors. This is particularly true when your game is simulating advanced physics, as many 3D games do these days.

The fix for rounding errors is the same as the fix for precision errors. You use a larger data type. The program still rounds when it has to. However, using a larger data type, such as `double`, means a greater number of digits in the numbers. Therefore, your results are more accurate even if they are still rounded.

# Case Study: Floating-Point Numbers and Gamespaces

It's time to see floating-point numbers in action. To see how they work, you'll do a case study that simulates the firing of a cannon. This simulation is more like real games than the example programs shown so far because it uses floating-point numbers for all of its calculations. The objects in the simulation will also be positioned using floating-point numbers.

As you've already seen, computer screens use integers rather than floating-point numbers. So if all of the simulation's calculations and object positions are in floating-point numbers, how do you connect that to integer locations on the screen?

The answer is that you use a gamespace.

## What is a Gamespace?

A gamespace is a virtual world that all of a game's objects exist in. The gamespace can be any size. It can use any unit of measure. So, for instance, programmers who like the English system can use inches, feet, yards, and miles. Programmers who prefer the metric system can define their gamespaces using centimeters, meters, and kilometers.

In real games, gamespaces are almost always defined using floating-point numbers. This enables them to do much more accurate and realistic physics calculations. In the case study of the cannon, the program must perform calculations that simulate the movement of a cannonball. If it were to use only integers, it would probably not look very realistic because of inaccuracies in the physics calculations. Instead, the cannon simulation creates a gamespace that uses vectors containing floating-point numbers. This enables more accurate calculations than integer vectors.

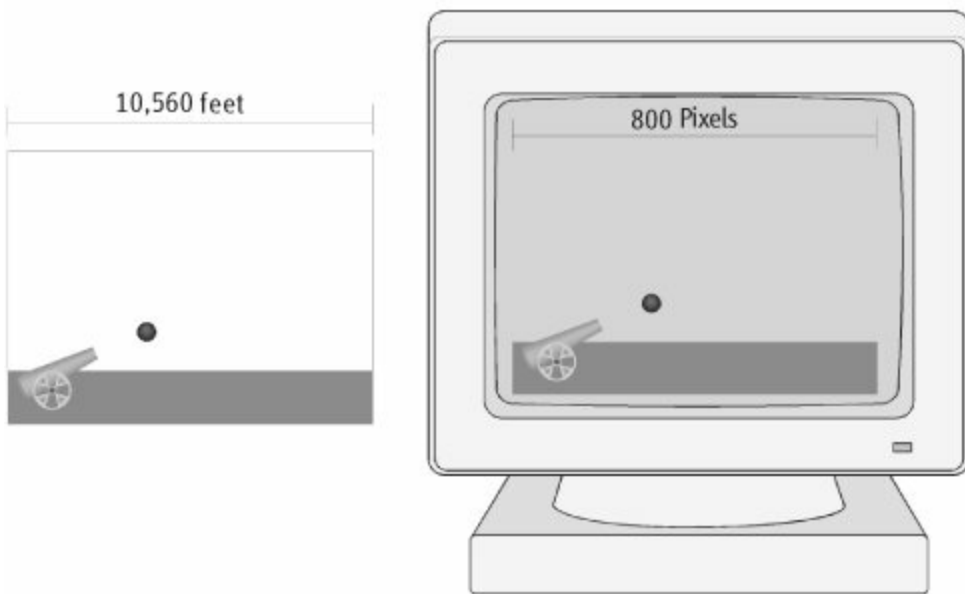
When your game uses a gamespace, it must convert the locations of objects from floating-point game coordinates into integer screen coordinates. This is illustrated in [Figure 9.4](#).

**Figure 9.4. The gamespace on the left contains the game's objects. The images of the objects are drawn in the screen space at the right.**



The Gamespace

The Screen Space



[Figure 9.4](#) shows that the gamespace for the cannon simulation is 10,560 feet (2 miles) wide. The screen resolution is set so that the screen is 800 pixels wide. The position of the cannonball that's in flight is defined in the gamespace with floating-point numbers. The same is true of the cannon's position. During each frame, the position of the cannon and cannonball must be converted to integer screen coordinates so that the cannon and the ball can be rendered at the correct locations on the screen.

## Factoid

Another common name for game coordinates is *world coordinates* because the gamespace can be called a world.

To convert from gamespace coordinates to screen space coordinates, you use the formulas shown in [Figure 9.5](#).

**Figure 9.5. Converting coordinates from the gamespace to the screen space.**

$$x_{SS} = \frac{\text{ScreenWidth}}{\text{GamespaceWidth}} \times x_{GS}$$

$$y_{SS} = \frac{\text{ScreenHeight}}{\text{GamespaceHeight}} \times y_{GS}$$

In the equations in [Figure 9.5](#), the term  $x_{SS}$  stands for the x value in the screen space. The term  $x_{GS}$  is the x value in the gamespace. Likewise,  $y_{SS}$  and  $y_{GS}$  represent the y values in the screen space and the gamespace, respectively.

As the first equation in [Figure 9.5](#) shows, you convert x values in the gamespace to screen space by dividing the width of the screen in pixels by the width of the gamespace. In our cannon example, the height of the gamespace is in feet (10,560.0 feet to be exact). Next, you multiply the result by the x coordinate in the gamespace.

Similarly, you convert y values in the gamespace to screen space by dividing the height of the screen in pixels by the width of the gamespace. In the cannon example, the width of the gamespace is 7920.0 feet. You multiply the result of the division by the y value in the gamespace.

## Tip

I've defined the gamespace for the cannon simulator in English units (feet). However, if you're serious about game programming, I strongly encourage you to become familiar with the metric system if you're not already. You'll find that almost all physics calculations are easier using the metric system.

Converting from screen space coordinates to gamespace coordinates uses the formula in [Figure 9.6](#).

**Figure 9.6. Converting coordinates from the screen space to the gamespace.**

$$x_{GS} = \frac{\text{GamespaceWidth}}{\text{ScreenWidth}} \times x_{SS}$$

$$y_{GS} = \frac{\text{GamespaceHeight}}{\text{ScreenHeight}} \times y_{SS}$$

As with the equation in [Figure 9.5](#), the term  $y_{SS}$  in [Figure 9.6](#) stands for the  $y$  value in the screen space. The term  $y_{GS}$  is the  $y$  value in the gamespace. The terms  $x_{SS}$  and  $x_{GS}$  represent the  $x$  values in the screen space and the gamespace.

## Writing a Vector Class that Uses Floating-Point Numbers

Most gamespaces define the positions and movements of objects using floating-point numbers for greater accuracy. This requires the use of vectors containing floating-point numbers. LlamaWorks2D provides a floating-point vector class called `vectorf`. [Listing 9.1](#) gives the class definition for the `vectorf` class.

### Listing 9.1. The `vectorf` class

```
1 class vectorf
2 {
3 public:
4     vectorf();
5     vectorf(
6 float xComponent,
7 float yComponent);
8
9     void X(
10 float xComponent);
11     float X(void);
12
13     void Y(
14 float yComponent);
15     float Y(void);
16
17     vectorf operator +(
18     vectorf &rightOperand);
19     vectorf operator -(
20     vectorf &rightOperand);
21
22     vectorf operator *(
23     float rightOperand);
24     friend vectorf operator *(
```

```

25     float leftOperand,
26     vectorf &rightOperand);
27
28     vectorf operator /(
29     float rightOperand);
30     friend vectorf operator /(
31     float leftOperand,
32     vectorf &rightOperand);
33
34     vectorf &operator +=(
35     vectorf &rightOperand);
36     vectorf &operator -=(
37     vectorf &rightOperand);
38     vectorf &operator *=(
39     float rightOperand);
40     vectorf &operator /=(
41     float rightOperand);
42
43     float Dot(
44     vectorf &rightOperand);
45
46     float Magnitude();
47     float MagnitudeSquared();
48     vectorf Normalize();
49
50 private:
51     float x,y;
52 };

```

As you can see, the `vectorf` class is exactly the same as the `vector` class, except that it uses floating-point numbers rather than integers.

## Cannonshoot: A Gamespace in Action

With an introduction to gamespaces and the `vectorf` class, you're ready to use floating-point numbers in a game-like simulation. As mentioned previously, this program simulates a cannon firing a cannonball. For the physics calculations that determine where the cannonball is and how it moves, the program uses floating-point numbers and vectors containing floating-point numbers.

## Introducing the Cannonshoot Class

We'll call this program `CannonShoot`. It consists of a game class, named `cannonshoot`, that is stored in a header file called `CannonShoot.h`. The functions for the `cannonshoot` are in the file `CannonShoot.cpp`.

[Listing 9.2](#) gives the definition of the `cannonshoot` class. This class resembles the game classes for programs in previous chapters. By now, lines 112 of [Listing 9.2](#) should look quite familiar.

## Note

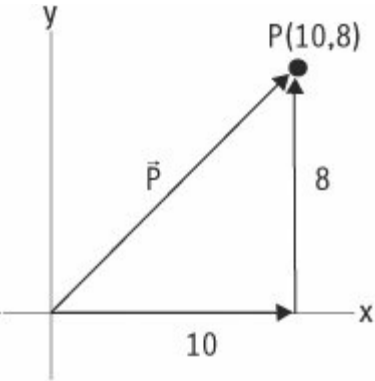
You'll find the source files for the CannonShoot program on the CD in the folder `\Source\Chapter09\Prog_09_01`. There's an executable version of the program in the folder `\Source\Chapter09\Bin`. To play the simulation, press the A key to arm the cannon and the F key to fire. Press Q to quit.

## Listing 9.2. The `cannonshoot` class

```
1 class cannonshoot : public game
2 {
3 public:
4     bool OnAppLoad();
5     bool InitGame();
6     bool UpdateFrame();
7     bool RenderFrame();
8
9     ATTACH_MESSAGE_MAP;
10
11     bool OnKeyDown(
12         keyboard_input_message &theMessage);
13
14     vector GameCoordsToScreenCoords(
15         vectorf theVector);
16 private:
17     cannon theCannon;
18     sprite theGround;
19     float gameSpaceWidth, gameSpaceHeight;
20 };
```

There are a couple of noteworthy items in [Listing 9.2](#). First, the `cannonshoot` class contains a function called `GameCoordsToScreenCoords()`. This function converts vectors in the gamespace to vectors in the screen space. Using vectors enables this function to convert both x,y coordinates and velocity vectors to values in the screen space. [Figure 9.7](#) demonstrates why.

## Figure 9.7. Points can be specified using vectors or x,y coordinates.



In [Figure 9.7](#), the point P is specified as a pair of x,y coordinates *and* as a vector. The vector **p** points from the origin of the gamespace to the point P. The x and y components of vector **p** are the same as the x,y coordinates of point P. This works for screen space as well. No matter what coordinate space you're using, you can specify a point as a vector.

The ability to specify points as vectors means that the CannonShoot program can call the `GameCoordsToScreenCoords()` function to convert either velocity vectors or points into their corresponding screen space values.

Another noteworthy aspect of the `cannonshoot` class is the fact that this game class is in a header file rather than the `.cpp` file that implements its member functions. Various functions in the game call the `GameCoordsToScreenCoords()` function. Therefore, to make `GameCoordsToScreenCoords()` available to all of the game's functions, it's put into a `.h` file that can be included in any of the game's `.cpp` files.

### Tip

Your game code will usually be simpler if you define all of your game's objects in the gamespace.

Notice also that the `cannonshoot` class contains an object of type `cannon` and one of type `sprite`. The `cannon` object is, of course the cannon that this program simulates. The `sprite` object represents the ground. There's a potential problem here. The

x,y coordinates of the `cannon` object are defined in the gamespace. The x,y coordinates of the `sprite` object are defined in screen space. I wrote the simulation this way to demonstrate that it is possible to use both spaces in your game class. Even though this technique is all too common in games, I strongly urge you not to use it. If your game is very complex, having some objects in screen space and others in the gamespace can get very confusing. Also, it typically forces your game to do multiple conversions of coordinates back and forth between the two spaces. These are unnecessary if you just define all of your objects in the gamespace.

## Implementing the Cannonshoot Class

The implementation of the `cannonshoot` class is in the file `CannonShoot.cpp`. The functions in this file fall generally into three categories: object initialization, frame update and rendering, and coordinate conversion.

### Initializing a cannonshoot object

Initializing a `cannonshoot` object is similar to initializing the game classes in previous chapters. However, there are some important differences, as [Listing 9.3](#) illustrates.

### Listing 9.3. The `OnAppLoad` and `InitGame()` functions of the `cannonshoot` class

```
1  #include "Cannonball.h"
2  #include "Cannon.h"
3  #include "Cannonshoot.h"
4
5  CREATE_GAME_OBJECT(cannonshoot);
6
7  START_MESSAGE_MAP(cannonshoot)
8    ON_WMKEYDOWN(OnKeyDown)
9  END_MESSAGE_MAP(cannonshoot)
10
11 bool cannonshoot::OnAppLoad()
12 {
13     bool initOK = true;
14
15     //
16     // Initialize the window parameters.
17     //
18     init_params lw2dInitParams;
19     lw2dInitParams.openParams.fullScreenWindow = true;
```

```

20 lw2dInitiParams.winParams.screenMode. AddSupportedResolution(
21     LWSR_800X600X32);
22 lw2dInitiParams.winParams.screenMode. AddSupportedResolution(
23     LWSR_800X600X24);
24 lw2dInitiParams.openParams.millisecondsBetweenFrames = 30;
25
26 // This call MUST appear in this function.
27 theApp.InitApp(lw2dInitiParams);
28
29 return (initOK);
30 }
31
32 bool cannonshoot::InitGame()
33 {
34     bool initOK = true;
35
36     gameSpaceWidth = 10560.0f;
37     gameSpaceHeight = 7920.0f;
38
39     srand((unsigned)time(NULL));
40
41     theGround.BitmapTransparentColor(color_rgb(0.0f,1.0f, 0.0f));
42     initOK =
43         theGround.LoadImage(
44             "ground.bmp",
45             image_file_base::LWIFF_WINDOWS_BMP);
46
47     if (initOK==true)
48     {
49         theGround.X(0);
50         theGround.Y(484);
51
52         initOK = theCannon.LoadImage();
53     }
54
55     if (initOK==true)
56     {
57         theCannon.X(0.0f);
58         theCannon.Y(gameSpaceHeight-1300.0f);
59
60         initOK =
61             theCannon.LoadSounds(
62                 "CannonFire.wav",
63                 "BallHit.wav");
64     }
65
66     return (initOK);
67 }

```

After including the files it needs, CannonShoot.cpp creates a `cannonshoot` game object on line 5. It also declares its message map on lines 79.

The `OnAppLoad()` function begins on line 11 of [Listing 9.3](#). Notice that this simulation only supports screen resolutions of 800x600. It can use either 32 or 24 bits of color. However, it forces the number of pixels to 600 rows of 800



per row. The conversion of gamespace to screen coordinates is much easier if both the gamespace and the screen space have fixed sizes.

The `InitGame()` function, which starts on line 32, sets the size of the gamespace on lines 36-37. The gamespace is 10,560 feet wide (2 miles) and 7,920 feet tall. I chose a width of 2 miles rather arbitrarily. The height is based on the width. I used the proportions of the screen space to find the height of the gamespace. If you divide the height of the screen space by its width, you get a ratio ( $3/4$ , in case you're interested). I set the height of the gamespace so that it has the same ratio as the screen space. That is, if you divide 7,920 by 10,560, you get  $3/4$ .

Making the gamespace and screen space proportional to each other is not required for your game. However, it makes the game's calculations easier to program. If they are not proportional, you have to scale all of your points and vectors in the gamespace so that they will fit properly in the screen space.

On lines 41-45 of [Listing 9.3](#), the `InitGame()` function loads the bitmap for the ground. If it loads properly, `InitGame()` sets the ground's position on lines 49-50. Remember, the x and y values shown here are in screen space.

Next, the `InitGame()` function loads the cannon's image. As you'll see later in this chapter, the cannon comes complete with a cannonball (no extra charge). Therefore, when `InitGame()` loads the cannon's image, it also loads the cannonball's image.

On lines 57-58, `InitGame()` sets the position of the cannon in the gamespace. The origin of the gamespace is in the upper-left corner, exactly the same as the screen space. I did that to keep the coordinate conversion process simple. Most games set the origin of their gamespaces to the lower-left corner. As a result, they have to invert all the y values in the game when they draw things to the screen. That isn't hard to do I just didn't want to throw too much at you at once.

After `InitGame()` positions the cannon, it loads the sounds the cannon uses and then returns.

## Updating and Rendering a Frame

When the `cannonshoot` object updates a frame of animation, it doesn't actually do much. Most of the work is accomplished by the `cannon` and `cannonball` classes. Rendering is equally straightforward, as shown in [Listing 9.4](#).

## Listing 9.4. The `UpdateFrame()`, `RenderFrame()`, and `OnKeyDown()` functions for the `cannonshoot` class

```
1  bool cannonshoot::UpdateFrame()
2  {
3      bool updateOK = theCannon.Update();
4
5      cannonball &currentBall = theCannon.CannonBall();
6      if ((currentBall.BallInFlight()) &&
7          (currentBall.Y() > theCannon.Y()+650.0f))
8      {
9          currentBall.BallHit();
10     }
11
12     return (updateOK);
13 }
14
15
16 bool cannonshoot::RenderFrame()
17 {
18     bool renderOK = theGround.Render();
19     if (renderOK)
20     {
21         renderOK = theCannon.Render();
22     }
23
24     return renderOK;
25 }
26
27
28 bool cannonshoot::OnKeyDown(
29     keyboard_input_message &theMessage)
30 {
31     vector paddleDirection;
32     float temp = 0.0f;
33
34     switch (theMessage.keyCode)
35     {
36     case KC_A:
37         temp = (float)rand();
38         while (temp > 1.0)
39         {
40             temp /= 10;
41         }
42         theCannon.ArmCannon(temp);
43         break;
44
45     case KC_F:
46         theCannon.FireCannon();
47         break;
48
49     case KC_Q:
50         GameOver(true);
51     }
52     return (false);
53 }
```

The `UpdateFrame()` function, which begins on line 1 of [Listing 9.4](#), first tells the cannon to update itself. Updating the cannon also updates the cannonball. However, the simulation must determine whether or not the cannonball hit the ground. On line 5, the `UpdateFrame()` function calls the `cannon::CannonBall()` function to obtain a reference to the `cannonball` object. Recall that functions normally return copies of objects. When a function returns a reference, it returns the actual object rather than a copy. The `UpdateFrame()` function needs a reference to the cannon's `cannonball` object because it must change the status of the `cannonball` if the `cannonball` hit the ground.

On lines 67, `UpdateFrame()` uses an `if` statement to see if the ball is in flight and it hit the ground. If the ball hit the ground, the `UpdateFrame()` function calls the `cannon::BallHit()` function. As you'll see shortly, the `BallHit()` function plays a sound and erases the cannonball when the ball hits the ground.

Updates occur in this simulation whenever the user presses particular keys on the keyboard. The `OnKeyDown()` function responds to those keystrokes. It begins on line 28 of [Listing 9.4](#). When the user presses the A key on the keyboard, it arms the cannon. Lines 3641 show that the `OnKeyDown()` function generates a random number between 0.0 and 1.0. It then passes that number to the `cannon` class's `ArmCannon()` function. This number is actually a percentage. If, for instance, the number is 0.5, the cannon fires the cannonball with 50% of its maximum charge. If the number is 0.75, it fires the ball with 75% of the maximum charge, and so forth.

The cannon fires when the user presses the F key. If the user presses Q, the simulation ends.

Rendering, as usual, is extremely easy. The `Render()` function, beginning on line 16, simply calls the `Render()` function for the `sprite` class to draw the ground. It then invokes the `Render()` function for the `cannon` class to draw the `cannon` and the `cannonball`.

## Converting Floating-Point Gamespace Coordinates to Integer Screen Coordinates

The final function in `CannonShoot.cpp` is used to convert coordinates in the gamespace to screen space coordinates. It's shown in [Listing 9.5](#).

### Listing 9.5. The `cannonshoot` class's

## GameCoordsToScreenCoords() function

```
1  vector cannonshoot::GameCoordsToScreenCoords(  
2      vectorf theVector)  
3  {  
4      static const float scaleFactor = 1.0f;  
5      vector theAnswer;  
6  
7      theAnswer.X(  
8          (int)((800 / gameSpaceWidth) *  
9          theVector.X()));  
10     theAnswer.Y(  
11         (int)((600 / gameSpaceHeight) *  
12         theVector.Y()));  
13     return (theAnswer);  
14 }
```

This function uses the formulas presented back in [Figure 9.5](#). In both calculations, the `GameCoordsToScreenCoords()` function uses the C++ `(int)` statement to tell the compiler to convert the answers from floating-point numbers to integers. It does this by lopping off everything in the answer to the right of the decimal point.

## Creating a Cannonball

To help make the cannon simulator easier to present, I should discuss the `cannonball` class before the `cannon` class. Although the order of things may seem counterintuitive, it simplifies the presentation quite a bit.

First, I'll cover the design of the `cannonball` class. Because we're now working with gamespaces, the design of the `cannonball` class is not as simple as you might think. After examining the design of the `cannonball` class, I'll discuss how it's implemented.

## Design Considerations for the Cannonball Class

Because the cannon simulator defines its objects in a gamespace rather than screen space, designing the `cannonball` class is different than the example programs of previous chapters. I'll illustrate these differences by presenting the `cannonball` class in [Listing 9.6](#).

## Listing 9.6. The `cannonball` class

```
1 class cannonball
2 {
3 public:
4     cannonball();
5
6     bool LoadImage(
7         std::string fileName,
8         image_file_base::image_file_format fileFormat);
9
10    bool LoadHitSound(
11        std::string fileName);
12
13    void BallHit();
14    bool Move();
15    bool Render();
16
17    void X(
18        float upLeftX);
19    float X();
20
21    void Y(
22        float upLeftY);
23    float Y();
24
25    int BitmapHeight();
26    int BitmapWidth();
27
28    void BoundingRectangle(
29        bitmap_region rectangle);
30    bitmap_region BoundingRectangle();
31
32    void BitmapTransparentColor(
33        color_rgb theColor);
34    color_rgb BitmapTransparentColor();
35
36    void Velocity(
37        vectorf direction);
38    vectorf Velocity();
39
40    bool BallInFlight();
41
42 private:
43     sprite theBall;
44     vectorf velocity;
45     float x,y;
46     bool ballInFlight;
47     sound ballHitSound;
48 };
```

The first and most important difference is that the `cannonball` class does not use inheritance. Instead of being derived from the `sprite` class, the `cannonball` class contains a `sprite` object. The definition of the cannonball's `sprite` object is shown on

line 43 of [Listing 9.6](#).

You may ask why the `cannonball` class does not use inheritance. If the `cannonball` class was derived from the `sprite` class, programs could call all of the `sprite` member functions using a `cannonball` object. That would enable a program to directly set the location of a `cannonball` object in screen space. Because `cannonball` objects keep track of where they are in both the gamespace and screen space, it would be possible for the object to think it's in two different places at once! You prevent that by avoiding inheritance for game objects that are defined in a gamespace.

## Tip

If a game's object is defined in a gamespace, that object should contain a `sprite` object. It should not inherit from the `sprite` class.

Another difference between previous game objects and the `cannonball` class is that it defines its own versions of many of the functions in the `sprite` class. Avoiding inheritance means that programs can't use a `cannonball` object to call `sprite` functions. Therefore, the `cannonball` class must define its own versions of them, as shown in [Listing 9.7](#).

## Listing 9.7. The inline functions for the `cannonball` class

```
1  inline bool
2  cannonball::LoadImage(
3      std::string fileName,
4      image_file_base::image_file_format fileFormat)
5  {
6      return (theBall.LoadImage(fileName, fileFormat));
7  }
8
9  inline bool
10 cannonball::Render()
11 {
12     return (theBall.Render());
13 }
14
15 inline void
16 cannonball::X(
17     float upLeftX)
18 {
19     x = upLeftX;
20 }
21
```

```
22 inline float
23 cannonball::X()
24 {
25     return (x);
26 }
27
28 inline void
29 cannonball::Y(
30     float upLeftY)
31 {
32     y = upLeftY;
33 }
34
35 inline float
36 cannonball::Y()
37 {
38     return (y);
39 }
40
41 inline int
42 cannonball::BitmapHeight()
43 {
44     return (theBall.BitmapHeight());
45 }
46
47 inline int
48 cannonball::BitmapWidth()
49 {
50     return (theBall.BitmapWidth());
51 }
52
53 inline void
54 cannonball::BoundingBox(
55     bitmap_region rectangle)
56 {
57     theBall.BoundingBox(rectangle);
58 }
59
60 inline bitmap_region
61 cannonball::BoundingBox()
62 {
63     return (theBall.BoundingBox());
64 }
65
66 inline void
67 cannonball::BitmapTransparentColor(
68     color_rgb theColor)
69 {
70     theBall.BitmapTransparentColor(theColor);
71 }
72
73 inline color_rgb
74 cannonball::BitmapTransparentColor()
75 {
76     return (theBall.BitmapTransparentColor());
77 }
78
79 inline bool
80 cannonball::BallInFlight()
81 {
82     return (ballInFlight);
```

[Listing 9.7](#) shows only the inline functions for the `cannonball` class. All of these functions except the overloaded `x()` and `y()` functions call the equivalent functions from the `sprite` class. The `x()` and `y()` functions perform their operations on data members that are defined in the `cannonball` class. If you look back at [Listing 9.6](#), you'll see that the `x` and `y` members defined on line 45 are of type `float`. The `cannonball` class uses these data members to store the cannonball's position in the gamespace. It uses the `x` and `y` members defined in the `sprite` class to keep track of where the cannonball's image is in screen space.

Before we move on I want to point out one more feature of the `cannonball` class. In [Listing 9.6](#), you can see that not only does the `cannonball` class define its own `x` and `y` members, it also defines its own velocity vector. As you can see from line 44 of [Listing 9.6](#), the velocity vector is of type `vectorf`, which is a vector that contains floating-point numbers. The `cannonball` class uses this data member to keep track of the cannonball's movement in the gamespace.

## Simulating a Cannonball in Flight

It's time for a bit of floating-point math and physics. Specifically, the `cannonball` class must simulate how a cannonball moves when it's shot through the air. The trajectory of a cannonball is a parabola. [Figure 9.8](#) displays what this looks like.

**Figure 9.8. A cannonball moves in a parabolic trajectory.**



[Figure 9.8](#) demonstrates that the effect of gravity on the cannonball in flight is to bend the ball's trajectory from a straight line into a parabola. So the question is, do you know how to solve parabolic equations?

Actually, you don't need to know math that advanced for this simulation. Parabolic equations aren't necessary. Instead, you can use simple vector



addition. [Figure 9.9](#) shows how.

## Figure 9.9. Using vector addition to calculate the cannonball's path.



[Figure 9.9](#) shows the cannonball in four different positions along its trajectory. The velocity vector always points straight ahead in the direction the ball is moving. This is the upper vector shown for each position of the ball. Gravity points down. To simulate the effect of gravity, all your game has to do is add a gravity vector to the ball's velocity vector at each position along the trajectory. The result is a new vector that becomes the cannonball's actual course. This is the lower vector shown for each position of the ball.

Repeatedly adding the gravity vector causes the course of the ball to change. To be specific, it bends the trajectory into a parabola. You can use this technique to simulate the effect of gravity on any projectile close to the earth's surface. It works for cannonballs, arrows, bullets, baseballs, or whatever else you want to simulate. The exception to this is if the object is powered, as rockets are. That calls for more complex math and physics. But for the simple objects that many games simulate, you can use vector addition to calculate their courses. [Listing 9.8](#) demonstrates how the CannonShoot program uses this technique to simulate the movement of the cannonball.

### Listing 9.8. The `cannonball::Move()` function

```
1  bool
2  cannonball::Move()
3  {
4      static vectorf gravity(0.0f, 32.0f);
5
6      if (ballInFlight)
7      {
8          velocity.Y(velocity.Y() + gravity.Y());
9
10         x += velocity.X();
11         y += velocity.Y();
```

```

12     vector ballPosition =
13         theApp.Game()->GameCoordsToScreenCoords(vecto rf(x,y));
14
15
16     theBall.X(ballPosition.X());
17     theBall.Y(ballPosition.Y());
18 }
19 }

```

The function in [Listing 9.8](#) is called each time the cannon is updated. It defines a variable of type `vector` on line 4. The definition uses the C++ keyword `static` to keep this variable around for as long as the program runs. Recall that variables in functions go away as soon as the function ends. If the function gets called again, its variables are re-created. When you add the keyword `static` to a variable declaration, it tells the program that this variable should continue to exist after the function ends. However, no other function can access it.

The first time the `Move()` function in [Listing 9.8](#) is called, it initializes the variable `gravity` by setting its x component to 0.0 and its y component to 32.0. This creates a floating-point vector that points straight down at 32 feet/second/second (32 ft/s<sup>2</sup>). When this function is called again, it does not have to do the initialization again because the variable `gravity` is declared `static`.

If the cannonball has been fired from the cannon, the statement on line 8 adds the gravity vector to the velocity vector. Lines 10-11 then use the velocity vector to calculate the next position of the cannonball in the gamespace. On lines 13-14, the `Move()` function calls the `cannonshoot::GameCoordsToScreenCoords()` function to convert the ball's position into coordinates. Lines 16-17 call the `sprite::X()` and `sprite::Y()` functions to set the position of the cannonball's image on the screen.

## Tip

If you're using the metric system (a good idea) rather than English units, the downward acceleration of gravity is 9.8 meters/second/second (9.8 m/s<sup>2</sup>). Therefore, you set the y component of your gravity vector to 9.8.

The `cannonball` class's `ballInFlight` member, which is tested on line 6, is set in one of the `cannonball` class's two overloaded `Velocity()` functions. [Listing 9.9](#) gives the code for both of them.

## Listing 9.9. The `cannonball` class's `Velocity()` functions

```
1 void
2 cannonball::Velocity(
3     vectorf direction)
4 {
5     ASSERT(velocity.MagnitudeSquared() >= 0.0f);
6
7     velocity = direction;
8     if (velocity.MagnitudeSquared() > 0.0f)
9     {
10        ballInFlight = true;
11    }
12 }
13
14 vectorf
15 cannonball::Velocity()
16 {
17     return (velocity);
18 }
19 }
```

The CannonShoot program calls the first `Velocity()` function shown in [Listing 9.9](#) to set the cannonball's velocity. In addition to setting the velocity vector, which is named `velocity`, the `Velocity()` function uses an `if` statement beginning on line 8. The `if` statement checks to see if the velocity is nonzero. It does this by calling the `vectorf` class's `MagnitudeSquared()` function rather than its `Magnitude()` function. The `MagnitudeSquared()` function is more efficient because it doesn't have to find any square roots. This doesn't affect the accuracy of the test. If the velocity is not zero, then the square of the velocity is also not zero. So if the test on line 8 evaluates to `TRue`, then the statement on line 10 sets the variable `ballInFlight` to `TRue`. In other words, if the cannonball is moving, it's in flight. If it's sitting still, it's not.

If the cannonball hits something, such as the ground, it should react to that event. That reaction is shown in [Listing 9.10](#).

## Listing 9.10. The `cannonball::BallHit()` function

```
1 void
2 cannonball::BallHit()
3 {
4     velocity.X(0.0f);
5     velocity.Y(0.0f);
6     ballHitSound.Play();
```

```
7     ballInFlight = false;
8 }
```

In this simulation, the cannonball stops moving if it hits something. Therefore, the `BallHit()` function sets its `velocity` vector to zero. It also plays a thumping sound and sets the `ballInFlight` member to `false`.

That's really all there is to the workings of the `cannonball` class. Let's move on and examine how the `cannon` class operates.

## Creating a Cannon

Like the cannonball, the cannon is defined in the gamespace. As a result, it does not inherit from the `sprite` class. Also like the `cannonball` class, the `cannon` class contains members called `x` and `y` that keep track of the cannon's position in the gamespace. Here's the definition of the `cannon` class.

### Listing 9.11. The `cannon` class

```
1  class cannon
2  {
3  public:
4      cannon();
5
6      cannonball &CannonBall();
7
8      bool LoadImage();
9      bool LoadSounds(
10         std::string fireSoundFileName,
11         std::string bounceSoundFileName);
12
13     void X(float xLocation);
14     float X();
15
16     void Y(float yLocation);
17     float Y();
18
19     void ArmCannon(float powderCharge);
20
21     void FireCannon();
22
23     bool Update();
24     bool Render();
25
26 private:
27     float BallMaxVelocity();
28     pointf BallStartPosition();
```

```
29
30 private:
31     sprite theCannon;
32     float x,y;
33     vectorf aimDirection;
34     float charge;
35     cannonball theBall;
36     sound cannonFireSound;
37     bool cannonArmed;
38 };
```

Now that you've seen the `cannonshoot` and `cannonball` classes, the `cannon` class should be straightforward. It provides functions for loading the images of the cannon and cannonball, as well as loading the sound associated with the cannon. In addition, it has overloaded `x()` and `y()` functions that get and set the x and y components of the cannon's position in the gamespace.

The `cannon::ArmCannon()` function was discussed earlier, when I presented the `cannonshoot::OnKeyDown()` function. It sets the `cannon` class's `charge` member, which represents the amount of gunpowder in the cannon. The `charge` member must contain a value greater than 0.0 and less than or equal to 1.0. It is a percentage of the maximum velocity the cannonball can attain. To get the maximum velocity, the program calls the `cannon::BallMaxVelocity()` function. [Listing 9.12](#) presents the code for this function.

### Listing 9.12. The `cannon::BallMaxVelocity()` function

```
1 inline float
2 cannon::BallMaxVelocity()
3 {
4     return (2500.0f);
5 }
```

The only thing the `BallMaxVelocity()` function does is return the value 2500.0. This is a technique for defining constants that go with classes. Recall that previous chapters demonstrated that you can also use statements such as `enum` or `const` to define constants. However, the `enum` statement is for groups of related constants and the `const` statement can't be used inside a class definition. An easy way to define constants that have nothing to do with each other is to make them return values of inline functions. Because the compiler performs code substitutions for inline functions, it adds no overhead to define constants this way.

In addition, defining constants as the return value of a function makes it easy to define constants that are objects. For example, the `cannon` class contains a function called `BallStartPosition()`. The code for this function appears in [Listing 9.13](#).

### Listing 9.13. The `cannon::BallStartPosition()` function

```
1 inline pointf
2 cannon::BallStartPosition()
3 {
4     return (pointf(844.8f,0.0f));
5 }
```

[Listing 9.13](#) shows that the `BallStartPosition()` function always returns the same value. The value it returns is of type `pointf`. The `pointf` class is provided by `LlamaWorks2D` and represents a point in any 2D space. It contains floating-point numbers. So each time that the program calls the `BallStartPosition()` function, the `BallStartPosition()` function returns the starting position of the cannonball. The starting position is relative to the upper-left corner of the cannon in the gamespace.

### Arming and Firing the Cannon

Before the cannon can fire a cannonball, the `CannonShoot` program must call the `cannon::ArmCannon()` function. This sets the initial velocity of the cannonball.

After the cannon is armed, it can be fired by calling the `cannon::FireCannon()` function. [Listing 9.14](#) provides the code for both of these functions.

### Listing 9.14. The functions for arming and firing the cannon

```
1 void cannon::ArmCannon(
2     float powderCharge)
3 {
4     ASSERT(powderCharge > 0.0f);
5     ASSERT(powderCharge <= 1.0f);
6
7     charge = powderCharge;
8
9     cannonArmed = true;
10 }
11
```

```

12
13 void cannon::FireCannon()
14 {
15     if (cannonArmed)
16     {
17         pointf ballStart = BallStartPosition();
18
19         ballStart.x += x;
20         ballStart.y += y;
21
22         theBall.X(ballStart.x);
23         theBall.Y(ballStart.y);
24
25         vectorf ballVelocity = aimDirection.Normalize();
26         ballVelocity *= BallMaxVelocity() * charge;
27         theBall.Velocity(ballVelocity);
28
29         if (!cannonFireSound.Play())
30         {
31             lw_error cantPlaysound(LWES_FATAL_FRAME_ERROR);
32             throw cantPlaysound;
33         }
34     }
35
36     cannonArmed = false;
37     charge = 0.0f;
38 }

```

The `ArmCannon()` function uses assertions on lines 45 of [Listing 9.14](#). The assertions crash the program if the value of the `powderCharge` parameter is not correct. The `powderCharge` parameter must contain a value that is greater than 0.0 and less than or equal to 1.0. If it does not, it is a programmer error that should be caught before the game is released. Assertions make it easy to catch the error because a program crash is hard to miss.

If the assertions indicate that the value of the `powderCharge` parameter is correct, `ArmCannon()` sets the charge member. It also sets the `cannonArmed` member to `true` before it exits.

The `FireCannon()` function starts on line 13 of [Listing 9.14](#). If the cannon is armed, it sets the starting position of the cannonball on lines 17-20. Recall that the starting position of the ball is relative to the upper-left corner of the cannon. Therefore, it must be added to the cannon's current position. That's done on lines 19-20.

On line 25 of [Listing 9.14](#), the `FireCannon()` function gets a unit vector that points in the direction that the cannon is aimed. You may remember from [chapter 5](#), "Function and Operator Overloading," that a unit vector is a vector of length 1. The cannonball's maximum velocity is a floating-point number (also called a

scalar). By multiplying the maximum velocity by the unit vector (see line 26), the `FireCannon()` function gets a vector that points in the direction the cannon is aimed and is set to the cannonball's maximum velocity. Line 26 also multiplies the result by the value in `charge`. Because `charge` contains a percentage, this multiplication scales the vector's magnitude to a percentage of the maximum velocity. On line 27, the `FireCannon()` function sets the cannonball's resulting velocity.

Next, the `FireCannon()` function plays the sound of the cannon firing on line 29. If there is an error when the sound is played, `FireCannon()` throws an exception. This is a way of indicating an error.

Before exiting, the `FireCannon()` function sets `cannonArmed` to `false` so the cannon cannot be fired again until it is rearmed. It also sets `charge` to 0.0 for the same reason.

## Updating and Rendering a Cannon

Because the cannonball class does so much of the work during a frame, updating and rendering the cannon is fairly simple. [Listing 9.15](#) gives the code for the `cannon::Update()` and `cannon::Render()` functions.

### Listing 9.15. The functions for updating and rendering a cannon

```
1  bool cannon::Update()
2  {
3      bool updateOK = true;
4
5      vector cannonPosition =
6          theApp.Game()->GameCoordsToScreenCoords(vectorf(x,y));
7      theCannon.X(cannonPosition.X());
8      theCannon.Y(cannonPosition.Y());
9
10     if (theBall.BallInFlight())
11     {
12         updateOK = theBall.Move();
13     }
14     return (updateOK);
15 }
16
17
18 bool cannon::Render()
19 {
20     bool renderOK = theCannon.Render();
21
22     if ((renderOK) && (theBall.BallInFlight()))
23     {
```



```
24     renderOK = theBall.Render();
25     }
26     return (renderOK);
27 }
```

[Listing 9.15](#) shows that each time the cannon is updated, it sets the position of its image in screen space. If the cannonball has been fired, the cannon also moves the cannonball.

When it is time to render the cannon, the cannon first renders its own image. If the cannonball is in flight, the cannon also renders the cannonball.

# Summary

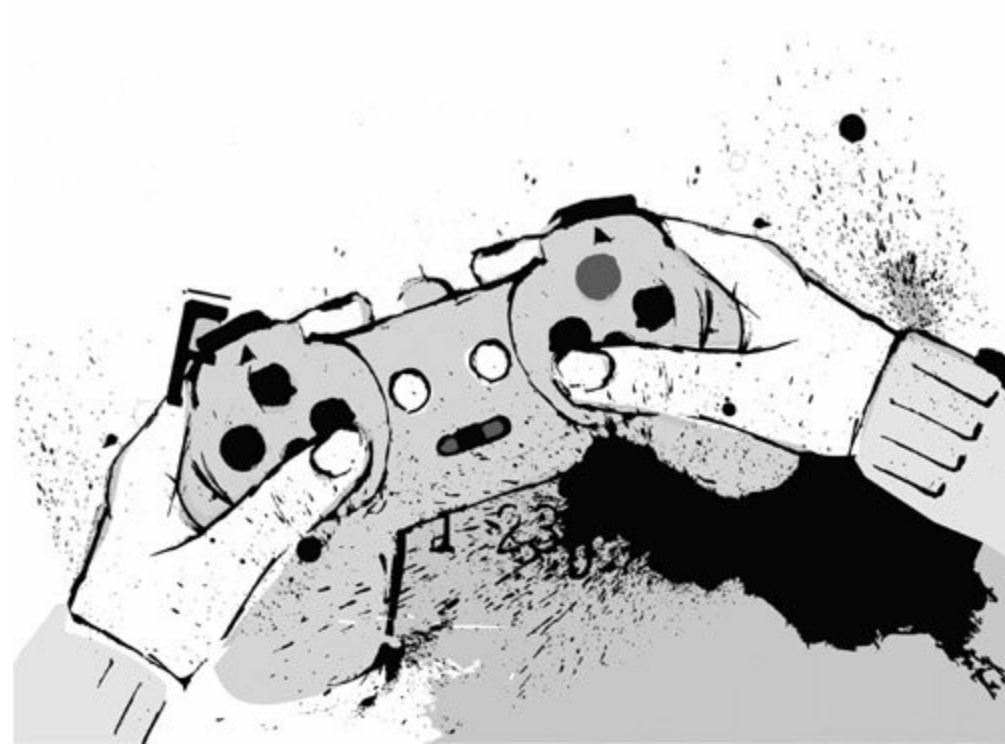
This rather long chapter provided the details of floating-point numbers. You saw how C++ defines and stores floating-point numbers, as well as some potential problems with using them.

In addition, this chapter introduced the concept of gamespaces. It showed how to use floating-point numbers to define objects in gamespaces. This is one of the most common uses of floating-point numbers in games.

In [Part 5](#) of this book, we'll expand the capabilities of gamespaces to include gamespaces that are larger than the screen. That will give us plenty of room to populate the gamespace with lots of interesting characters. Games usually store groups of characters in arrays, which is the subject of the next chapter.

# Chapter 10. Arrays

When you play most games, you're typically attacked by hordes of hostile aliens, trolls, pushy insurance salespeople, or whatever. For each individual character, your game must declare a variable. That can add up to a lot of variables. Wouldn't it be nice if C++ offered an easy way to declare a group of variables all of the same type? Well, as a matter of fact, C++ provides arrays for just that purpose.



# What Are Arrays?

An *array* is a block of memory locations that all hold the same type of information. There are lots of uses for arrays, but one of the most common in games is to create a group of enemies.

To get an idea of what an array is, imagine a group of mailboxes at a post office. Each individual mailbox has its own address, and can hold different information.

## Note

This chapter will give you all the basics you need to understand and code arrays. For the practical application, hold tight: [Part 5](#) is called "The Big Payoff" for a reason we'll use all the knowledge you've been building to actually create a game.

Arrays work very much like a group of mailboxes that you might see at an apartment building or post office. Each "mailbox" is called an [array element](#). Every array element has its own address and holds one piece of data. Every array element is the same data type and the same size as every other array element.

For an example, take a look at [Figure 10.1](#), which provides a graphical representation of an array of integers. All elements in this particular array must contain integers. None of the elements can be used for storing other types of data, such as floating-point numbers, because this array is specifically declared as an array of integers. However, each element in the array can hold a different integer value.

**Figure 10.1. An array of 10 integer elements.**

thisArray 

123	-15	87	0	987	42	0	12	13	14
-----	-----	----	---	-----	----	---	----	----	----

Arrays can hold any data type. For instance, you can declare arrays that hold

floating-point numbers and others that hold sprites.

# Declaring and Using Arrays

An array is a variable. Just like other kinds of variables, your programs must declare arrays to use them. Array declarations must have a data type, a variable name, and a pair of square brackets that usually contain the size of the array. For example, if you want to declare an array of integers, the declaration would be similar to this:

```
int thisArray[10];
```

The statement `int` specifies the data type. The name of the array is `thisArray`, and its size is 10, which means it holds 10 integers. Most programmers call the number in the square brackets of the declaration the size specifier. [Figure 10.2](#) illustrates this array.

**Figure 10.2. An array of 10 elements.**

	0	1	2	3	4	5	6	7	8	9
thisArray	123	-15	87	0	987	42	0	12	13	14

## Note

Programmers call the array element number its *subscript*. When they read statements such as `thisArray[0]`, `thisArray[1]`, and so forth, they read them as "this array sub zero, this array sub one," and so on. Somewhat confusingly, array subscript numbers are also referred to as array *index* numbers.

Every element in an array is numbered. To continue the mailbox analogy, the array element number is essentially the address of the element. Array element numbering always starts with 0 (zero). To access an element in an array, your program specifies the array name followed by the element number in square brackets. If your program needs to access the first element in the array, it

specifies array location 0, like this:

```
thisArray[0] = 5;
```

This statement stores a 5 in the first location in the array, which is location number 0. To store a value in the tenth array element, you store it in array location number 9:

```
thisArray[9] = -20;
```

Pop quiz. If you want to store a value in the fifth element, what location is that?

Answer: The fifth element in the array is in location number 4. This is how you would write it in your code.

```
thisArray[4] = 10;
```

It may be confusing at first that the fifth element in an array is numbered 4 rather than 5. However, as you work with arrays, you'll quickly get used to it.

To retrieve a value from an array, you just put the array to the right of an equal sign, like so:

```
int thisInt = thisArray[4];
```

When you first start working with arrays, it's easy to get the terminology mixed up. For instance, the two statements below both contain numbers in square brackets.

```
int thisArray[10];  
thisArray[9] = -20;
```

The first statement shown here is the declaration of the array. The 10 in the

square brackets is the size specifier for the array. It tells how many integers the array can hold. However, the 9 in the second statement is not a size specifier. It is a subscript. It tells which array location the -20 is to be stored in.





## Why Does Array Numbering Start with 0?

When I was teaching college-level C++ programming classes, my students would often wonder why C++ array numbering starts with 0 rather than 1. There is actually a very good reason.

C++ array numbering starts with 0 because the array name actually holds the address of the start of the array. The starting address of the array is called the array's [base address](#).

To access a particular array element, the executable version of your program contains code that adds the element number to the array's starting address. The compiler generates this code. You never see it in your source code, but that's what's going on behind the scenes.

When your program needs to access the first array element, it contains a statement such as the following:

```
thisArray[0] = 5;
```

The compiler generates executable code that adds the element number, which is 0, to the address of the array. It then stores the 5 at that location.

To access the second, third, or fourth elements, the compiler generates executable code that adds 1, 2, or 3 to the base address.

Because the element or subscript numbers are added to the base address of an array, they can also be referred to as [offset](#) numbers. The technique of using a base address and offset number to access array elements is extremely efficient. It is one of the features that make C++ programs so fast.

## Warning

Even though array size specifiers and subscript numbers both appear in square brackets, they are two different things that serve two different purposes.

You can use an array of integers anywhere you would use an integer. Let's demonstrate with a small sample program.

### Listing 10.1. Using an array in a program

```
1 #include <cstdlib>
2 #include <iostream>
3
```

```

4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      int thisArray[10];
9      int i;
10
11     i = 0;
12     while (i<10)
13     {
14         thisArray[i] = 10-i;
15         i++;
16     }
17
18     i=0;
19     while (i<10)
20     {
21         cout << thisArray[i];
22         cout << endl;
23         i++;
24     }
25
26     system("PAUSE");
27
28     i=9;
29     while (i>=0)
30     {
31         cout << thisArray[i];
32         cout << endl;
33         i--;
34     }
35
36     system("PAUSE");
37     return EXIT_SUCCESS;
38 }

```

This is a short console program, similar to the ones you wrote in the first few chapters of this book. It declares an array on line 8. The program next declares a loop counter variable called `i`. On line 11, the program sets `i` to 0.

## Tip

If you can't remember how to compile a console program, you can access instructions for compiling by inserting the CD that comes with this book into your CD-ROM drive. Your computer will automatically display the Welcome page for the CD. On the Welcome page is a link to instructions for compiling all of the sample programs.

Next, the program enters a loop on line 12. In the loop, the program stores values in `thisArray`. The first time through the loop, `i` is 0, so the statement on line 14 stores the value 10 (the result of `10 - 0`) in `thisArray[0]`. The second time through the loop, `i` is 1, so the statement on line 14 stores the value 9 (the result of `10 - 1`) in `thisArray[1]`. It continues in this manner until `i` is 10. [Figure 10.3](#) shows the result of the loop on lines 12-16.

**Figure 10.3. Storing numbers in an array.**

	0	1	2	3	4	5	6	7	8	9
thisArray	10	9	8	7	6	5	4	3	2	1

After the program fills the array, it pauses until the user presses a key. It then enters another loop on line 19 that prints each array element one by one. The first time through this loop, it prints the 10 stored in `thisArray[0]`. The next time through the loop, the program prints the 9 stored in `thisArray[1]`, and so on. When the loop exits, the program pauses again.

After the user presses a key, the program sets `i` to 9 and starts another loop on line 29. In this loop, `i` starts at 9 and gets decremented each time through the loop until it gets to -1, and then the loop quits. The statements on lines 31-32 print the contents of the current array location followed by an endline. Using this loop, the program prints the values 1 through 10.

## 2D Arrays

The array you saw in [Listing 10.1](#) was one-dimensional. When you use arrays, you can expand them into higher dimensions. This is not as odd as it sounds at first. If you think of a one-dimensional array as a row of data, as shown back in [Figure 10.3](#), then creating a two-dimensional array is just a matter of adding more rows. In fact, you can think of a 2D array as a table, as [Figure 10.4](#) illustrates.

**Figure 10.4. A 2D array.**

	0	1	2	3	4	5	6	7	8	9
0	10	9	8	7	6	5	4	3	2	1
1	100	-57	1	22	9	42	31	-91	0	0
2	-5	61	0	6	8	10	11	1	0	1

The 2D array of integers in [Figure 10.4](#) is shown as a table with 3 rows and 10 columns. Many programmers like to think of a 2D array as having a height and width. The height is the number of rows, and the width is the number of columns.

Your program declares a 2D array by simply adding the extra dimension to an array declaration. Here's how to declare the array shown in [Figure 10.4](#):

```
int thisArray[3][10];
```

The integer array `thisArray` contains 3 rows. Each row holds 10 integers. Another way to say that is that the array has 3 rows and 10 columns.

## Note

Two-dimensional arrays in C++ do not really have rows and columns. It's just convenient to think of them as if they do.

Accessing elements in a 2D array is similar using a one-dimensional array. For example, if I wanted to store the number 5 in the second location of the third row, I would use a statement like this:

```
thisArray[2][1] = 5;
```

Of course the third row in the array is numbered 2 because array numbering starts with 0. For the same reason, the second location in row number 2 (the third row) is numbered 1.

Let's look at a quick example of using 2D arrays ([Listing 10.2](#)). I'll revise the

program from [Listing 10.1](#) so that it uses a 2D array rather than a one-dimensional array.

## Listing 10.2. Using a 2D array in a program

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      int thisArray[3][10];
9      int i,j;
10
11     i = 0;
12     while (i<3)
13     {
14         j = 0;
15         while (j<10)
16         {
17             thisArray[i][j] = (i*j) + (i+j);
18             j++;
19         }
20         i++;
21     }
22
23     i = 0;
24     while (i<3)
25     {
26         j = 0;
27         while (j<10)
28         {
29             cout << thisArray[i][j];
30             if (thisArray[i][j]<10)
31             {
32                 cout << " ";
33             }
34             else
35             {
36                 cout << " ";
37             }
38             j++;
39         }
40         cout << endl;
41         i++;
42     }
43
44     system("PAUSE");
45
46     return EXIT_SUCCESS;
47 }
```

The program in [Listing 10.2](#) declares a 2D array on line 8. It also declares two integer variables it uses as loop counters. On line 12 it enters its first loop. This loop iterates through each row. It contains another loop on lines 15-19 that iterates through each column.

The first time through this pair of loops, both `i` and `j` are 0, so the program stores a 0 in row 0, column 0. Next, line 18 increments `j` to 1. The inner loop repeats because `j` is less than 10. On line 17, the program stores a 1 in row 0, column 1. The inner loop continues iterating until it has executed once for each column in row 0. At that point, `j` equals 10, so the inner loop ends. Line 20 increments `i` to 1 and the outer loop repeats. Line 14 sets `j` back to 0 and the inner loop executes 10 more times, once for each column in row 1. The process repeats again for row 2.

The statements on lines 23-42 use the same loop structure to enable the program to iterate through the 2D array again. As it does, it prints the value in each array location, followed by some spaces. The `if-else` statement on lines 30-37 put either one or two spaces between the printed numbers. This enables the program to line up the output nicely. [Figure 10.5](#) shows the output from this program.

**Figure 10.5. The output from the program in [Listing 10.2](#).**

```
0 1 2 3 4 5 6 7 8 9
1 3 5 7 9 11 13 15 17 19
2 5 8 11 14 17 20 23 26 29
Press any key to continue . . .
```

## 3D Arrays

You can also extend array declarations into three dimensions. Programmers often say that 3D arrays have height, width, and depth. It's rather like layering 2D arrays one on top of another—you can think of it like multiple tables (or spreadsheets from a spreadsheet program) laid one on top of another. Here's the declaration of a 3D array:

```
int thisArray[3][10][5];
```

In this declaration, the array `thisArray` can be thought of as a group of 10x5 tables. The stack of tables is three deep. [Listing 10.3](#) gives a version of the program from [Listing 10.2](#) that uses a 3D array.

## Listing 10.3. Using a 2D array in a program

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8  int thisArray[3][10][5];
9  int i,j,k;
10
11 i = 0;
12 while (i<3)
13 {
14 j = 0;
15 while (j<10)
16 {
17 k = 0;
18 while (k<5)
19 {
20     thisArray[i][j][k] = (i*j*k) + (i+j+k);
21     k++;
22 }
23 j++;
24 }
25 i++;
26 }
27
28 i = 0;
29 while (i<3)
30 {
31 j = 0;
32 while (j<10)
33 {
34 k = 0;
35 while (k<5)
36 {
37     cout << thisArray[i][j][k];
38     if (thisArray[i][j][k]<10)
39     {
40         cout << " ";
41     }
42     else
43     {
44         cout << " ";
45     }
46     k++;
47 }
48 j++;
49 cout << endl;
50 }
```

```
51     i++;
52 }
53
54     system("PAUSE");
55
56     return EXIT_SUCCESS;
57 }
```

The program in [Listing 10.3](#) uses three nested loops on lines 1226 to iterate through the 3D array. These loops store values in the array. Another group of three nested loops on lines 2952 print the values in the array.



## Can I Make Arrays with More than Three Dimensions?

The C++ language supports arrays in more dimensions than three. The maximum number of array dimensions is determined by whoever writes the compiler you're using. Most programmers never use more than three dimensions in their arrays. In fact, the vast majority of programmers seldom use more than two array dimensions. Therefore, if you become comfortable using arrays of one, two, and three dimensions, you will know what you need to for game programming.

A few high-end methods of graphics programming use 4D arrays. Four-dimensional arrays are typically only used for very accurate collision detection or extremely high-end rendering (a technique called volumetric rendering). However, programming techniques involving 4D arrays are not usually needed in games. Instead, they are implemented in applications such as professional flight simulators that are used to train pilots. Such applications run on specialized hardware that generally costs millions of dollars.

# Initializing Arrays

Like other types of variables, arrays can be initialized when they are created. Array initializations require a value for each location that you want to initialize. For example, if you want to initialize an integer array with five elements, you would write a statement similar to the following:

```
int thisArray[5] = {42,11,6,0,1};
```

This initialization starts at the beginning of the array and copies values into the array locations. As a result, `thisArray[0]` gets the value 42, `thisArray[1]` gets set to 11, and so on.

An interesting thing about array initializations is that they enable you to leave off the size specifier. Here's an example:

```
int thisArray[] = {42,11,6,0,1};
```

Notice that there is no size specifier in the array's square brackets. This is not an error. The compiler counts the number of initial values in the initialization and allocates a memory location in the array for each one. Because there are five values in the initialization, the compiler makes `thisArray` big enough to hold five integers.

Arrays can have both size specifiers and initializations. You can leave off one or the other, but not both. You cannot, for instance, create an array with a statement like this:

```
int thisArray[];
```

In this case, `thisArray` has no size specifier and no initialization values. Therefore, the compiler will not know how many integer elements to allocate for `thisArray`. It expresses its displeasure by displaying an error message. To fix this statement, you must use a size specifier, a group of initialization values, or both.

## Tip

Always remember that arrays must have either a size specifier, a group of initialization values, or both.

To initialize arrays with more than one dimension, you separate the initialization values with braces. The initialization of a 2D array would resemble the following statement:

```
int thisArray[2][3] = { {1,2,3}, {2,3,4} };
```

The array `thisArray` contains two rows, with three integers per row. The initialization shown here has initializations for both rows. They are contained in the outer set of braces (the leftmost and rightmost ones). The initialization values for the two rows are also each in a set of braces. The two sets of initialization values are separated by a comma.

If multidimensional initializations seem confusing, you can arrange them into something that resembles a table. Here's how it's done:

```
int thisArray[2][3] =  
{  
    {1,2,3},  
    {2,3,4}  
};
```

This initialization does the same thing as the previous one. The braces that contain the entire initialization have been put on lines by themselves. The initializations for each row are each on their own lines. The row initialization values are still separated by a comma. The nice thing about this style of initialization is that it looks like a 2D table.

If you're wondering whether you can initialize a 3D array, the answer is yes. Here's an example of how it's done:

```
int thisArray[2][3][2] =  
{  
    {{1,2},{3,4},{5,6}},  
};
```

```
    {{2,3},{4,5},{6,7}}  
};
```

As you can see, this style of initialization gets a bit more complex with each dimension you add.

# Problems with Array Boundaries

I want to leave you with a final cautionary note before we end our discussion of arrays. The misuse of arrays is an excellent way to crash your program. Here's an example of what I mean:

```
int thisArray[10];
int i = 0;
while (i<50)
{
    thisArray[i] = i;
    i++;
}
```

You've probably spotted the problem with this code fragment. The array only holds 10 integers. However, the loop tries to store 50 integers into the array.

What happens when you execute code like this?

I think the result of code like this is best summed up with the phrase "crash, burn, and die."

To be more specific, the loop actually executes. It stores 50 integer values starting at the beginning of `thisArray`. Of course, the array only has enough memory for 10 integers. Even so, the program keeps going right past the end of `thisArray`. It happily writes integers as it frolics its way through your computer's memory. Whatever else might be at those 40 locations beyond the end of `thisArray` gets completely overwritten. It is not uncommon for the contents of the overwritten locations to contain information that is critical to the operation of your program. As a result, your program crashes, losing whatever data was not saved. Users don't take kindly to that.

When your program accesses memory before or after the array, programmers say that it is "going beyond the array boundaries." If you try to use negative values as array index numbers, you are outside the array's boundaries.

C++ does not prevent you from going outside an array's boundaries. Depending on which compiler you use, your compiler may or may not be able to figure out the problem and give you an error. Except for the most simple and obvious cases, most compilers won't catch the problem.

## Tip

Array boundary problems are very common in programs, so you should double-check for them in every loop that uses an array.

How do you detect this problem? Simple: your program crashes. Use your debugger to check whether your program crashes when it's in a loop that uses an array. If it does, the first thing you should examine is whether the index numbers are outside the boundaries of the array.

# Summary

The subject of this chapter is probably less exciting than something like writing a cannon simulation. However, arrays are so massively useful in programs you absolutely can't write anything significant without them.

Arrays are blocks of memory locations that all hold the same type of information. They can have one, two, three, or more dimensions. The numbering for each dimension always starts with zero. Programs access arrays using the array name and a subscript number. Your program should never set the subscript number to a value that is outside the boundaries of an array.

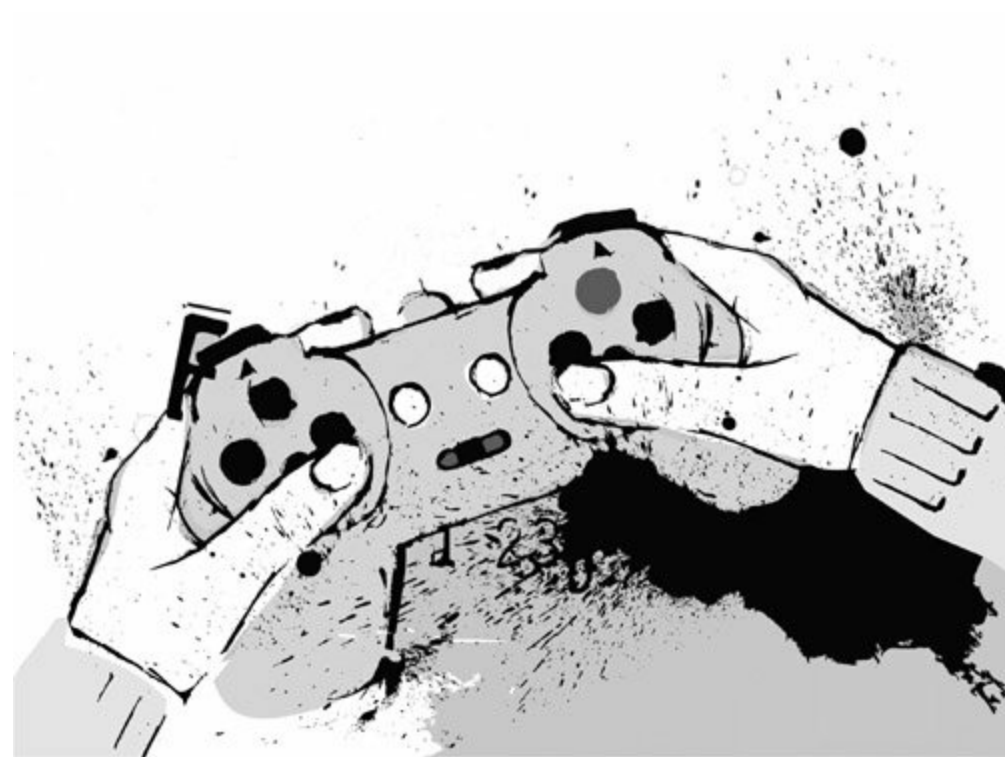
Arrays are a powerful tool for handling groups of variables in your programs. The C++ programming language provides another tool that is similar to arrays but even more powerful and flexible and more confusing. That tool is called a pointer, and pointers are the subject of the next chapter.

# Chapter 11. Pointers

This chapter introduces a subject that is closely related to arrays. As you learned in the previous chapter, an array is a method for creating a block of memory that contains data. A *pointer* is another, more powerful way of doing the same thing.

When your program declares a variable, it asks the operating system for some memory. The program uses that memory to store data. For instance, if your program declares an integer variable, it gets enough memory from the operating system to store an integer. Programmers call this process *memory allocation*.

In addition to allocating memory for data, your programs can allocate memory for pointers to data. A pointer does not contain data; instead, it contains the address of data. Every location in memory has a unique address. When your program creates a pointer, it creates a variable that contains an address of a location in memory.



As you'll see throughout the remaining chapters of this book, games use pointers a lot. The ability to create a pointer is an extremely powerful programming tool. It will also be the source of most of your programming errors. If you want to write games, you *must* learn to use pointers and use them well.



# Why Are Pointers Important to Games?

Games use pointers more than most types of programs. LlamaWorks2D hides most of that from you to make it easier for you to get started with game programming. However, even with LlamaWorks2D you'll need to understand the basics of pointers to write professional-quality games. In this respect, LlamaWorks2D is like every other game engine on the market. They all use pointers.

In addition, many important graphics libraries depend heavily on pointers. This is especially true of Microsoft's DirectX. You cannot use DirectX without pointers. OpenGL is the exception; OpenGL uses pointers, but it hides that completely from you. For that reason, beginners usually find it easier to get started with OpenGL than with DirectX.

Sound libraries also rely heavily on pointers. As you might expect, DirectX Audio is a prime example. OpenAL also hides the use of pointers from you. As with OpenGL, beginners find OpenAL easier to learn than DirectX Audio. However, DirectX Audio is much more powerful and flexible than OpenAL.

The long and short of this is that if you want to become a professional game programmer, you must learn to use pointers.

## **Note**

Don't forget knowing about pointers will help you in [Part 5](#), when we'll really get into the nitty-gritty and create a game from the ground up.

# Declaring and Using Pointers

You declare a pointer variable by stating the type of the variable, the fact that it's a pointer, and the name of the variable. Here's an example:

```
int *anIntPtr;  
int anInt = 100;
```

The variable `anIntPtr` points at integers, as the type name `int` indicates. The asterisk indicates that the variable is a pointer to integers rather than a variable that contains integers. The second variable declaration does not have the asterisk, so `anInt` is a variable that contains an integer.

You store an address in a variable with the ampersand operator, like this:

```
int *anIntPtr;  
int anInt = 100;  
anIntPtr = &anInt;
```

The third of these three statements stores the address of `anInt` into the pointer variable `anIntPtr`. As a result, the variable `anIntPtr` points to the location in memory where `anInt` resides. [Figure 11.1](#) represents this graphically.

**Figure 11.1. Storing an address in a pointer variable.**



In [Figure 11.1](#), the pointer is represented by an arrow. The arrow is a graphical way of depicting the fact that the pointer variable `anIntPtr` stores the address of `anInt`. Your program can access the data in `anInt` through the pointer. To do so, you use the asterisk operator. [Listing 11.1](#) gives a short example of how this is done.

## Listing 11.1. A program with a pointer

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      int anInt = 5;
9      int *anIntPtr = &anInt;
10
11     cout << "anInt = " << anInt << endl;
12     cout << "*anIntPtr = " << *anIntPtr << endl;
13     cout << endl;
14
15     *anIntPtr = 100;
16
17     cout << "anInt = " << anInt << endl;
18     cout << "*anIntPtr = " << *anIntPtr << endl;
19     cout << endl;
20
21     anInt = 50;
22
23     cout << "anInt = " << anInt << endl;
24     cout << "*anIntPtr = " << *anIntPtr << endl;
25     cout << endl;
26
27     system("PAUSE");
28     return EXIT_SUCCESS;
29 }
```

Line 8 of [Listing 11.1](#) declares an integer variable and stores a 5 in it. Line 9 creates an integer pointer (a pointer to an integer) and sets it to point to the variable `anInt`. Line 11 prints the value in `anInt`, which is 5. On line 12, the program prints the value in the location that `anIntPtr` points to. Because `anIntPtr` points to `anInt`, the statement on line 12 prints the 5 again.

## Knowing the Pointer Lingo

Programmers have specific ways of talking about pointers that help everyone understand what they're talking about. For example, programmers often read a statement like `*anIntPtr` as "the contents of `anIntPtr`." However, that's not accurate. A more accurate way to read it is "the contents of the location that `*anIntPtr` points to." However, most programmers find this too wordy, so they use the shorter and less accurate reading.

There's a more technical term for statements that access the contents of the location a pointer points at. Programmers call it a [dereference](#). Anytime programmers say anything about dereferencing a pointer, they're talking about accessing the memory location where the pointer points.

Line 15 of [Listing 11.1](#) stores the value 100 in the location that `anIntPtr` points to. The statements on lines 17-18 prove that doing so changes the value in `anInt` because `anInt` and `anIntPtr` both refer to the same memory location. This is proven again on lines 21-25. The assignment statement on line 21 stores a 50 in `anInt`. The statements in lines 23-24 show that this changes both `anInt` and `*anIntPtr` because both `anInt` and `*anIntPtr` refer to the same memory location.

Is this confusing? If it is, you're having a very normal experience. Pointers are hard to learn. We all struggle with them when we first encounter them. You'll become proficient at them through practice, practice, and more practice. Anyone can master pointers and everyone feels frustrated with them at first.

If you're wondering whether learning pointers is worthwhile, the answer is a big YES. I'll show you why shortly. However, before diving into that, I just want to quickly review what I've presented so far.

Declare a pointer variable by adding an asterisk to the declaration just before the variable name, like this:

```
int *anIntPtr;
```

Take the address of a variable with the ampersand, like this:

```
&anInt
```

Store addresses in pointer variables, like this:

```
int anInt;  
int *anIntPtr;  
anInputPointer = &anInt;
```

Access the contents of the memory location a pointer points at with the asterisk, like this:

`*anIntPtr`

These four points are what you need to know to declare and use pointers.

# Pointers and Dynamic Memory Allocation

What are pointers good for?

To answer that, I need to talk about how programs allocate memory.

The memory allocations done in the sample programs so far have all been *static memory allocation*. Statically allocated memory does not change its size for as long as it exists. Consider, for example, the following statement:

```
int thisInteger;
```

This statement allocates enough memory to hold one integer. For as long as the variable exists, it can hold only one integer.

What if the program declares an array instead? Is that still static?

The answer is yes. Let's examine an array declaration to see why:

```
int theseIntegers[10];
```

The array in the preceding statement allocates space for 10 integers. For as long as this variable exists, it can never hold more than 10 integers. The amount of memory allocated to this array does not change; it is static.

What if a program could change the amount of memory allocated for data as it runs?

It can. We call that *dynamic memory allocation*. Dynamic memory allocation is a tremendously important tool for programs in general and games in particular.

To understand why dynamic memory allocation is so critical to games, think about what happens in games. Imagine you're playing a game called Invasion of the Obnoxious Chicken Men. You're playing through level 1. You successfully save Lunar Colony 5 from the evil pecking horde of chicken men. You move on to level 2. Do you expect to see all of the same monsters that you saw on level 1? Probably not. Most games introduce new monsters when you move onto a higher level.

If you're writing *Invasion of the Obnoxious Chicken Men*, you know what kind of monsters there are in the game. In fact, you have to create a class for each type of monster. For instance, you may program a type of chicken man that you call a Mega-Scratcher. Are there MegaScratchers on level 1? Or are they introduced on level 2? That's up to you, but either way *you don't write that into your program!*

## Warning

Do not confuse the term "statically allocated memory" with the C++ keyword `static`, which you've seen in previous chapters. They refer to two completely unrelated ideas. Confusing, I know, but that's the way it is.

It's true. You don't program your game to know what kind of monsters appear on level 1, what kind appear on level 2, and so forth. Instead, you put that into a special file that you create called a *level file*. The level file contains information about what kind of monsters appear on each level. The game reads the level file and dynamically allocates whatever monsters it specifies. It then starts the level using the monsters the level file told it to.

Games use dynamic allocation for virtually everything contained in a level. This includes the scenery, opponents, allies, powerups, and so on. The level file tells the game what objects the level contains and the game allocates those objects dynamically. As a result, the game doesn't have to contain code for each individual level. If that were necessary, game programs would be *much* bigger than they are now. They would be more difficult and expensive to make. Dynamic allocation saves development time and money as well as memory and disk space. In short, you can't write games without dynamic memory allocation.

## Allocating Memory

The C++ language contains the `new` keyword, which is specifically for dynamic memory allocation. You can use it to allocate a single item. For instance, your program can allocate one integer or one MegaScratcher chicken man. The `new` keyword can allocate any type the compiler recognizes.

Programs can also use the `new` keyword to allocate groups of items. For example, your program can allocate a group of 100 floating-point numbers, or 25 sprites, or 30 MegaScratchers.

C++ does not limit the size of the block that your program allocates. However, a limit is imposed by the amount of memory available. If the requested amount of memory cannot be allocated, the `new` keyword returns a value of `NULL`. The value `NULL` is a special value defined in the C++ standard libraries.

Every time your program performs a dynamic memory allocation, it should test to ensure that the allocation was successful. [Listing 11.2](#) demonstrates dynamic memory allocation.

## Warning

Although some operating systems impose a size limit on blocks of allocated memory, most modern operating systems do not. However, if you ever write games for consoles or handheld devices, it is likely that the device's operating system will impose such a limit. One of the most common size limits is 64K. That was the size limit for an allocated memory block in DOS, the most common operating system before Windows.

## Listing 11.2. Allocating memory with `new`

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      int *someInts = NULL;
9      int *temp = NULL;
10     int i;
11
12     someInts = new int[10];
13     if (someInts != NULL)
14     {
15         i = 0;
16         temp = someInts;
17         while (i < 10)
18         {
```



```

19     *temp = i;
20     i++;
21     temp++;
22 }
23
24     i = 0;
25     temp = someInts;
26     while (i < 10)
27     {
28         cout << *temp << endl;
29         i++;
30         temp++;
31     }
32 }
33 else
34 {
35     cout << "Could not allocate memory." << endl;
36 }
37
38     system("PAUSE");
39     return EXIT_SUCCESS;
40 }

```

The short program in [Listing 11.2](#) begins by declaring three variables. The first two are integer pointers (pointers that point to integers). The third variable is an integer that the program uses as a loop counter.

Line 12 performs a dynamic memory allocation. It allocates enough memory for 10 integers. The syntax you see on line 12 is what you should use whenever you allocate a group of data items. That is, you use the keyword `new`, followed by the type and then square brackets containing the number of items to allocate. When the allocation is performed, the address of the first byte of the allocated memory block is stored in the integer pointer `someInts`.

Suppose you want to allocate just one integer. In that case, you would use a statement similar to the following:

```
int *anInt = new int;
```

This statement allocates space for one integer and saves the address of that integer in the pointer `anInt`.

Getting back to [Listing 11.2](#), imagine that not enough memory remains for 10 integers. In that case, the `new` keyword returns the value `NULL`. The `if` statement on line 13 tests whether the address in `someInts` is not equal to `NULL`. If it is not equal to `NULL`, the allocation was successful. Therefore, the program is free to

use the memory pointed to by the pointer.

On line 15, the program sets `i` to 0 so it can use `i` as a loop counter. Line 16 shows that the program sets the integer pointer `temp` to point to the same block of memory as `someInts`. When this statement executes, the program copies the address in `someInts` into `temp`.

Next, the program enters a loop on line 17. The loop iterates while `i` is less than 10. Inside the loop, the program stores the value in `i` into the memory location pointed to by `temp`. Notice the difference between the statements on lines 16 and 19. Line 16 copies an address into the pointer; it changes where the pointer points. Line 19 copies an integer into the memory that `temp` points at. These are two very different operations.

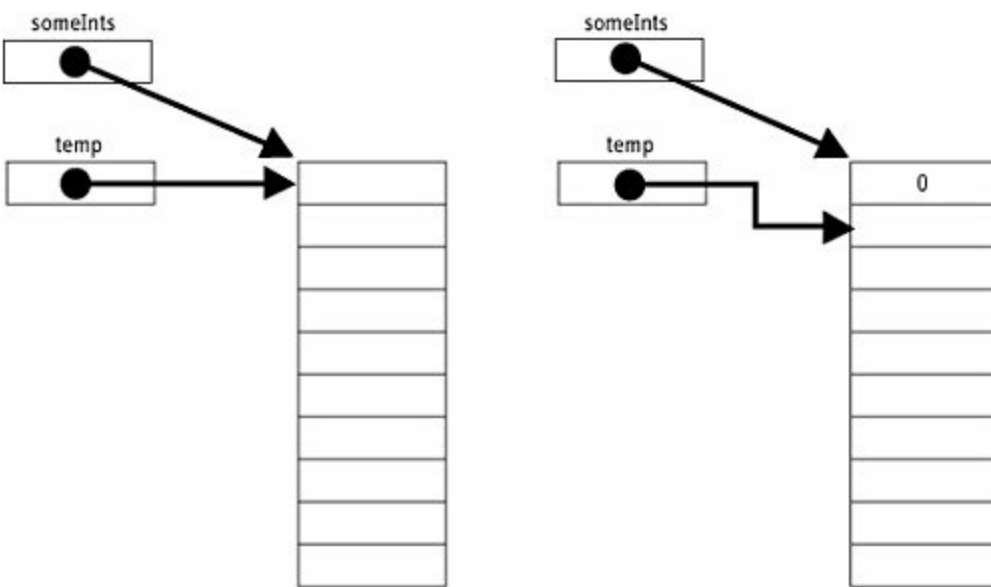
On line 20, the program increments `i`. On line 21, it increments `temp`. The question is, what does it mean to increment a pointer?

## Tip

Always check your memory allocations to ensure that they are successful. If the `new` keyword returns the value `NULL`, the allocation failed. Otherwise, it was successful.

Remember that a pointer contains the address of the location in memory it points at. Memory addresses are simply unsigned integers. If your program increments a pointer, it increments the address the pointer contains. In other words, each time you increment the pointer, it points to the next location in memory. [Figure 11.2](#) illustrates what's going on here.

## Figure 11.2. Incrementing a pointer.



[Figure 11.2](#) displays two views of the dynamically allocated memory from [Listing 11.2](#). On the left, it shows how things look at the beginning of the `while` loop that begins on line 17. The variables `someInts` and `temp` both point to the beginning of the block of integers that were allocated. Line 19 copies the value in `i` to the memory location that `temp` points at. Line 21 increments `temp`. When it does, `temp` points at the next location in memory. This is shown in the right-hand view in [Figure 11.2](#). Each time through the loop, line 21 increments `temp`. This moves `temp` through the entire block of memory one integer at a time.

Here's a question for you: Why does the program set `temp` to point to the same location as `someInts` on line 16?

Another look at [Listing 11.2](#) provides the answer. By the time the loop beginning on line 17 ends, `temp` points one integer location beyond the end of the block of memory. In other words, without `someInts`, the program loses the starting address of the block of memory. The program could solve this by decrementing the address in `temp` again. However, that's a waste of CPU time. It's much faster and more efficient to just keep a pointer to the beginning of the block.

On line 25 of [Listing 11.2](#), you can see the usefulness of using the `temp` pointer instead of incrementing `someInts`. Because the starting address of the memory block is still in `someInts`, the program can set `temp` to point to the beginning of the block again. The program can use another loop to increment it through memory once more. This time, it prints the contents of the memory locations that `temp` points at one at a time.

As you can see, this pointer stuff is not particularly straightforward. You can see that a program could have real problems if it lost the starting address of a

block of memory it allocated. In fact, this is one of the most common types of errors that occur in programs on the commercial markets. It's extremely easy to lose the address of a block of memory completely, making the block unusable and unrecoverable. The only way to recover a lost block of memory is to exit the program. The operating system automatically cleans up all memory allocated to the program.

Programmers have a special name for the process of losing the starting address of a dynamically allocated block of memory. The technical term for it is *memory orphaning*. However, the less formal and more common name of this is *leaking memory*.

If your program has a memory leak, it will typically continue to request memory from the operating system. However, the leaked memory won't be returned to the operating system while the program runs. Gradually, the amount of free memory that the operating system can give to the program gets smaller. Eventually, it will run out of memory completely. This is very likely to cause your program to crash. In the days of DOS, running out of memory usually also caused your entire computer to crash. With modern operating systems, it is much less likely that the program will crash the computer. Nevertheless, I have seen memory leaks crash Windows, Linux, and most other operating systems in use today.

When doing dynamic memory allocation, your biggest problem will be memory leaks. How do I know? Because it's the biggest problem all C++ programmers face when they do dynamic memory allocation. The solution to this problem is to get into the habit of cleaning up all memory you allocate.

## **Tip**

Always make sure your program saves the starting address of dynamically allocated memory. This enables the memory block to be used repeatedly.

## **Freeing Memory**

To keep yourself from having problems when you use dynamically allocated memory, you must remember to free all memory you allocate. You do this by

using the C++ keyword `delete`. Look again at the program in [Listing 11.2](#) to see where you find the keyword `delete`.

What's that? It's not there? You found a memory leak.

The block of memory allocated on line 12 is never freed. Therefore, it is lost. However, it's only lost temporarily. Whenever a program ends, the operating system automatically reclaims all memory given to a program.

You should never depend on the operating system's ability to reclaim leaked memory in your programs. People can play games for hours on end. Because a memory leak gradually decreases the amount of remaining memory, your game could crash after the player has been at it for hours. The result is lost progress and much frustration with your game. In fact, it can be enough to make people stop playing an otherwise good game.

The long and short of this is that you *must* get in the habit of freeing all memory you allocate. Yes, I know I've said that before. However, it's so important that I'm willing to risk your ire by repeating it.

Let's jump into another sample program to see how to free memory. [Listing 11.3](#) contains a program that demonstrates how to free a single integer. It also demonstrates how to free a group of integers. The same technique demonstrated here can be used for freeing any data type.

## Warning

If you ever get the opportunity to write games for a console, you'll find that the operating systems they use are much simpler and more primitive (it makes them faster). As a result, they often *do not* automatically reclaim all memory allocated to a program. It is extremely likely that a large memory leak will crash a console and force the player to completely reboot.

## Listing 11.3. Freeing dynamically allocated memory

```
1 #include <cstdlib>
2 #include <iostream>
3
```

```
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     int *intPointer = NULL;
9
10    cout << "Allocating an integer...";
11    intPointer = new int;
12    cout << "Done." << endl;
13    if (intPointer == NULL)
14    {
15        cout << "Error: Could not allocate the integer." << endl;
16    }
17    else
18    {
19        cout << "Success: Integer was allocated" << endl;
20        cout << "Storing a 5 in the integer...";
21        *intPointer = 5;
22        cout << "Done." << endl;
23        cout << "The integer = " << *intPointer << endl;
24
25        cout << "Freeing the allocated memory...";
26        delete intPointer;
27        cout << "Done." << endl;
28
29        cout << "Allocating 10 integers...";
30        intPointer = new int [10];
31        cout << "Done." << endl;
32    }
33
34    if (intPointer == NULL)
35    {
36        cout << "Error: Could not allocate 10 integers." << endl;
37    }
38    else
39    {
40        cout << "Storing 10 integers into the allocated memory";
41        int *temp = intPointer;
42        int i = 0;
43        while (i < 10)
44        {
45            *temp = 10-i;
46            i++;
47            temp++;
48        }
49
50        cout << "Done." << endl;
51
52        cout << "Here are the 10 integers." << endl;
53        temp = intPointer;
54        i = 0;
55        while (i < 10)
56        {
57            cout << *temp << endl;;
58            i++;
59            temp++;
60        }
61
62        cout << "Freeing the allocated memory...";
63        delete [] intPointer;
64        cout << "Done." << endl;
```

```
65     }  
66  
67     system("PAUSE");  
68     return EXIT_SUCCESS;  
69 }
```

The program in [Listing 11.3](#) declares an integer pointer on line 8 and initializes it to `NULL`. This is a way of saying that the pointer doesn't point to anything.

Next, the program allocates a single integer on line 11. On line 13, the program tests to see if the allocation was successful. If there isn't enough memory to allocate an integer, the program prints an error message on line 15. If the memory was allocated, the program executes the `else` statement that begins on line 17.

## Warning

In some cases, the `new` keyword does not return the value `NULL` when it can't allocate memory. Instead, it can throw an exception. Exceptions are an advanced C++ programming topic that we will not discuss in this book. By default, most compilers are configured to return `NULL` rather than throw exceptions when allocations fail. If your program crashes and displays an error message about an unhandled memory exception, you may need to adjust the compiler's settings. See your compiler documentation for details.

On line 21, the program stores a value in the allocated memory. For emphasis, I want to draw your attention to the fact that when the program stores the address of the newly allocated memory on line 11, it does not use the asterisk operator. However, when it stores a value in the dynamically allocated memory, it must use the asterisk. On line 23, the program prints the value stored in the dynamically allocated memory.

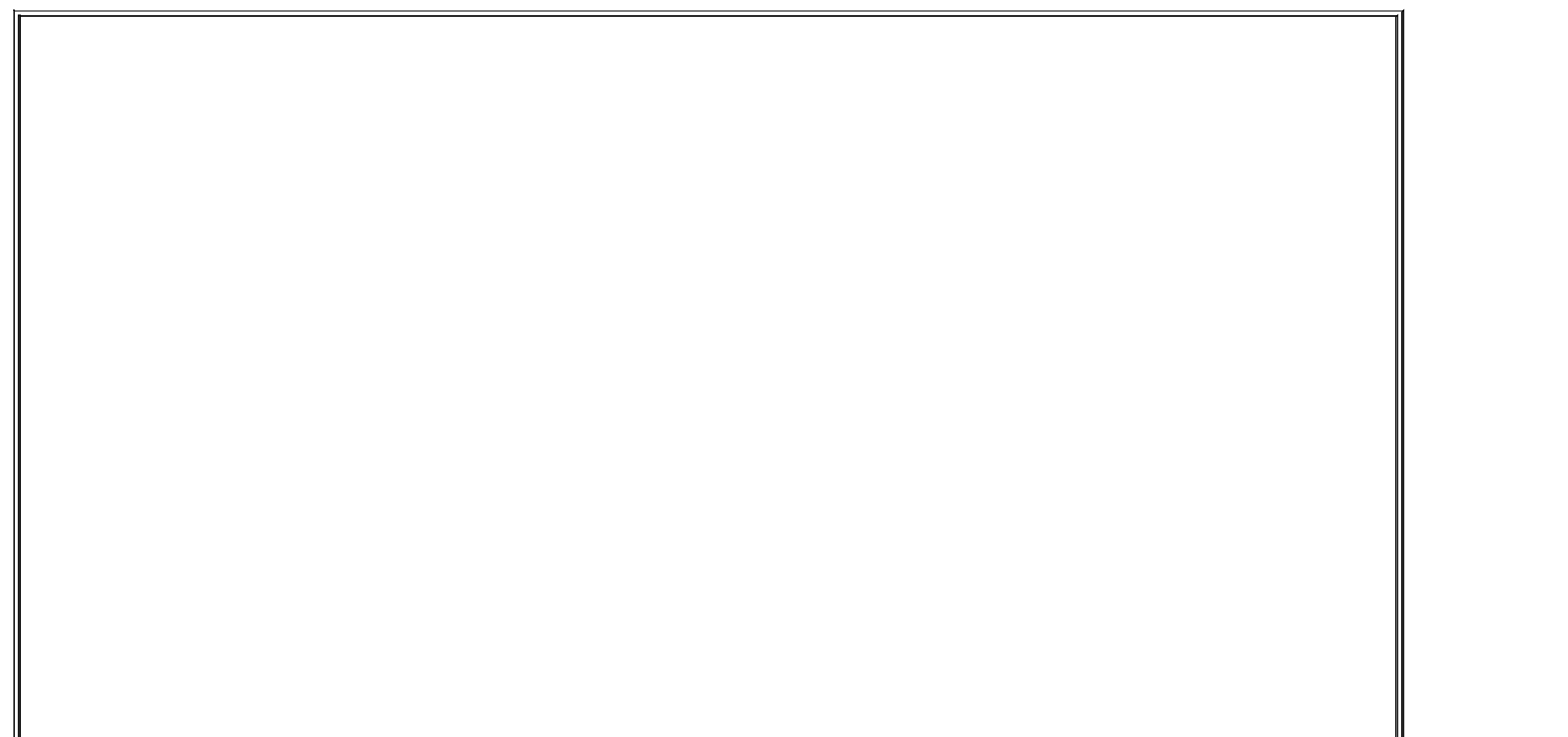
On line 26, the program uses the `delete` operator to free the memory it allocated. Because this statement frees memory for just one integer, it simply contains the keyword `delete` followed by the variable that points to the memory to be freed.

Allocating and freeing a group of memory locations works slightly differently. The statement on line 30 allocates a group of 10 integers. If the allocation is not successful, the `if` statement beginning on line 34 is executed. Otherwise, the program executes the `else` statement that starts on line 38. Within the `else` statement, the program uses the statements on lines 41-48 to store integer values into the allocated memory. On lines 53-60, it prints the contents of the allocated memory.

The big finale is on line 63. This statement deletes the group of integers allocated on line 30. The statement on line 63 is very similar to the one on line 26. The difference between the two is that the statement on line 63 contains the square brackets between the keyword `delete` and the variable. Those square brackets indicate that a group of memory locations are being freed, rather than just one single location.

You may wonder what would happen if you were to leave the square brackets off the statement on line 63. The answer is that it depends on the compiler. Some compilers may be "smart enough" to understand that you're really freeing a group of integers. However, the vast majority of compilers aren't that clever. They will happily compile your program and execute the statement without its square brackets. When they do, they free just one memory location; the rest are orphaned. In other words, you create a memory leak.

The moral from this is simple: You have to make *sure* you free *all* memory that you allocate. Make this your mantra. Say it 10 times every night before you go to bed. Well, maybe not. But do make it a priority when you're doing dynamic allocation in your programs.





## Where Does Dynamically Allocated Memory Really Come From?

In talking about allocating and freeing memory, I've said that when a program allocates memory, the operating system gives the memory to the program. I told you that when memory is freed, it is given back to the operating system. In saying these things, I've generalized a bit.

What specifically happens is that when your program starts, the operating system actually allocates a chunk of memory for your program's dynamically allocated memory. This chunk of memory is called the *heap*. Operating systems often impose a maximum size to the heap. When your program allocates the entire heap, there is no more dynamic memory to allocate.

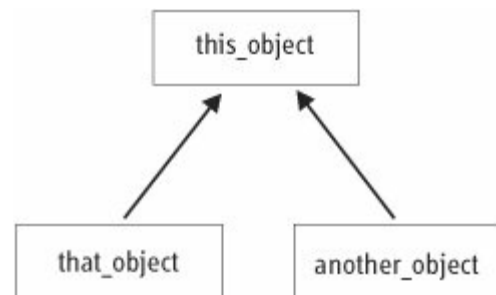
Dealing with the heap is a rather advanced topic and not essential to our goal in this book, which is programming a game in C++. However, you do at least need to know what the heap is.

# Pointers and Inheritance

Although it may not seem like it from the information I've presented so far, C++ pointers contain a surprising amount of smarts. They enable a neat trick when you use them together with inheritance. Here's how it works.

Suppose you create a class called `this_object`. Your program uses the `this_object` class as a base class. Imagine that the program also contains two more classes. One is called `that_object` and the other is named `another_object`. Both `that_object` and `another_object` inherit from `this_object`. [Figure 11.3](#) illustrates the relationship of these three classes.

**Figure 11.3. The `that_object` and `another_object` classes both inherit from `this_object`.**



The tricky thing about pointers is that pointers to base objects can point to derived objects. Imagine again that you're writing a program with the three classes in the diagram in [Figure 11.3](#). If your program declares a pointer to the `this_object` class, it can use that pointer to point to objects of type `that_object` and `another_object`. To see why this is such an important feature of pointers, let's examine the program in [Listing 11.4](#)

## Listing 11.4. Calling functions with base class pointers

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 class this_object
7 {
8 public:
9     virtual std::string Type()
```

```

10     {
11         return ("this_object");
12     }
13 };
14
15 class that_object : public this_object
16 {
17 public:
18     std::string Type()
19     {
20         return ("that_object");
21     }
22
23 };
24
25 class another_object : public this_object
26 {
27 public:
28     std::string Type()
29     {
30         return ("another_object");
31     }
32
33 };
34
35 int main(int argc, char *argv[])
36 {
37     this_object *thePointer;
38
39     thePointer = new that_object;
40
41     if (thePointer == NULL)
42     {
43         return (-1);
44     }
45
46     cout << thePointer->Type() << endl;
47
48     delete thePointer;
49
50     thePointer = new another_object;
51
52     if (thePointer == NULL)
53     {
54         return (-1);
55     }
56
57     cout << thePointer->Type() << endl;
58
59     delete thePointer;
60
61     system("PAUSE");
62     return EXIT_SUCCESS;
63 }

```

The program in [Listing 11.4](#) implements the three classes I've been discussing. As you can see from lines 15 and 25, both `that_object` and `another_object` inherit from

`this_object`. All three classes contain a function called `Type()`. The type functions all return a string that states the type name of their respective classes. The important thing to note about the `Type()` function in the base class is that its definition contains the C++ keyword `virtual`. Whenever a member function starts with the keyword `virtual`, it means that if a derived class contains a function with the same name, return type, and parameter list, then the function in the derived class is called *even if the pointer points to the base class!* In the example in [Listing 11.4](#), the derived class's `Type()` function overrides the `Type()` function in the base class. The pointer is smart enough to know that it's pointing at a derived class object rather than a base class object

So what's the purpose of that?

To discover the answer, read carefully through the `main()` function beginning on line 35. The first thing `main()` does is declare a pointer to a `this_object`, which is the base class for the other two. It then allocates a `that_object` and stores the object's address in the pointer to the base class. This is okay because `that_object` inherits from `this_object`. You can use a base class pointer to point to a derived object.

Nothing too tricky so far, but let's keep going.

If the allocation is not successful, the function returns a -1 to indicate an error. If it was successful, the program keeps going.

Line 46 contains a statement that is definitely tricky. It calls the `Type()` function using the pointer. Notice that it uses the C++ arrow notation (`->`) to call the `Type()` function. If it were to use a period like the programs did in previous chapters, the compiler would output an error. With pointers, you must use the arrow notation to call member functions. The arrow is made by a minus followed by a greater-than sign.

That's actually not the tricky part. It's coming next.

The variable `thePointer` is a pointer to `this_object`. However, it's pointing to a `that_object`.

Question: When the statement on line 46 is executed, which `Type()` function gets called? Will it be the one in the `this_object` class or the one in the `that_object` class?

Answer: The program calls the `Type()` function in the `that_object` class.

If your program uses a base class pointer to point to an object of a derived type, the pointer can be used to call functions in the derived class. This works only if the function in the base class has the keyword `virtual` in front of it, as the `Type()` function does in the `this_object` class. Putting in the keyword `virtual` tells the

program to call the functions in the derived class whenever the pointer points to a derived object.

To prove its point, the program deletes the object on line 48 and allocates an object of type `another_object` on line 50. It calls the `Type()` function again on line 57. This time, the `Type()` function in the `another_object` class gets called.

You might be thinking, "I'll grant you that it's tricky, but what good is it?"

## Note

If a program returns a nonzero value, it usually means there was an error. Typically, the error values are negative, but they do not have to be.

Imagine that the base class is named `screen_object`. Imagine that everything your program displays on the screen is derived from `screen_object`. Now suppose that the `screen_object` class has a virtual function in it called `Render()`. In that case, your program can use a pointer to a `screen_object` to call the `Render()` function for anything you want to display on the screen. You can put all the objects you want to draw into one list and move through the list, pointing to each object in turn with a pointer of type `screen_object`. For each object in the list, your program can call the `Render()` function and everything will draw itself to the screen.

In other words, you can create objects of type `bomb`, `gun`, `flower`, `skunk`, `dragon`, and so forth. All of these classes must be derived from `screen_object`. If they are, they all must have a `Render()` function. They can all go into the same list. You can use a loop that moves a pointer to a `screen_object` through the list. On each pass through the loop, you can call the `Render()` function for that object and the correct `Render()` function will be called.

The ability to call functions in derived classes using pointers to base classes is crucial to games. In fact, LlamaWorks2D uses exactly this technique. LlamaWorks2D actually provides a base class called `screen_object`. Your game can put any object derived from LlamaWorks2D's `screen_object` class into a level. All classes derived from `screen_object` must have a `Render()` function. LlamaWorks2D uses a `screen_object` pointer to call the `Render()` functions of all objects in a level. I'll demonstrate this technique in [Part 5](#), so you'll get practice with it and can learn to use it in your games. Once you get the hang of it, you'll find it very straightforward to do and you'll see that it's absolutely essential for your

games.

# Arrays Are Pointers in Disguise

There is a strong connection between arrays and pointers. In fact, arrays *are* pointers. I like to say that they're pointers in disguise because they don't look like pointers. Unlike pointers, arrays must always point to the same block of memory. Also, the size of that memory block never changes. It's statically allocated.

Because pointers and arrays are related, it's possible to allocate a block of memory with a pointer and then treat it like it's an array. That's very convenient.

In addition, the close relation of pointers and arrays means that it is possible to create dynamically allocated arrays that change size. This is a somewhat advanced technique that I won't demonstrate in this book. However, I do want you to be aware that the possibility exists. For now, I'll just present a program in [Listing 11.5](#) that shows how to allocate a block of memory and treat it as an array.

## Listing 11.5. The equivalence of pointers and arrays

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[])
7  {
8      int *someInts = new int [10];
9      if (someInts==NULL)
10     {
11         return (-1);
12     }
13
14     int i=0;
15     while (i<10)
16     {
17         someInts[i] = 10-i;
18         i++;
19     }
20
21     i=0;
22     while (i<10)
23     {
24         cout << someInts[i] << endl;
25         i++;
26     }
27
28     delete [] someInts;
```

```
29
30     system("PAUSE");
31     return EXIT_SUCCESS;
32 }
```

The short program declares a variable that is a pointer to integers on line 8 of [Listing 11.5](#). It also allocates memory for 10 integers. On lines 14-19, the program uses a loop to store integers into the block of memory it allocated. Notice in particular line 17. Even though `someInts` is declared as a pointer to integers rather than an array, the program can still use array notation with the pointer. It is perfectly acceptable to use array notation with the variable `someInts` because there's a fundamental equivalence between pointers and arrays.

Most programmers find it much easier to deal with array notation rather than pointer notation. The loop on lines 21-26 iterates through the block of memory and prints the integers the memory contains. Once again, it uses array notation with the pointer variable `someInts`. C++ compilers have no problem with this.

It is very common for C++ programmers to use array notation with pointers in all types of programs. This technique is not limited to games.



# Summary

In this chapter, you learned what pointers are and how to use them. The chapter demonstrated that your programs can use pointers to dynamically allocate memory. Whenever a program is through with dynamically allocated memory, it can free the memory for other uses.

Dynamic memory allocation can cause memory leaks. As a result, you should get into the habit of freeing all memory you allocate.

Pointers can be used with inheritance. It enables programs to use base class pointers to call derived class member functions. This is a technique that games rely on quite heavily.

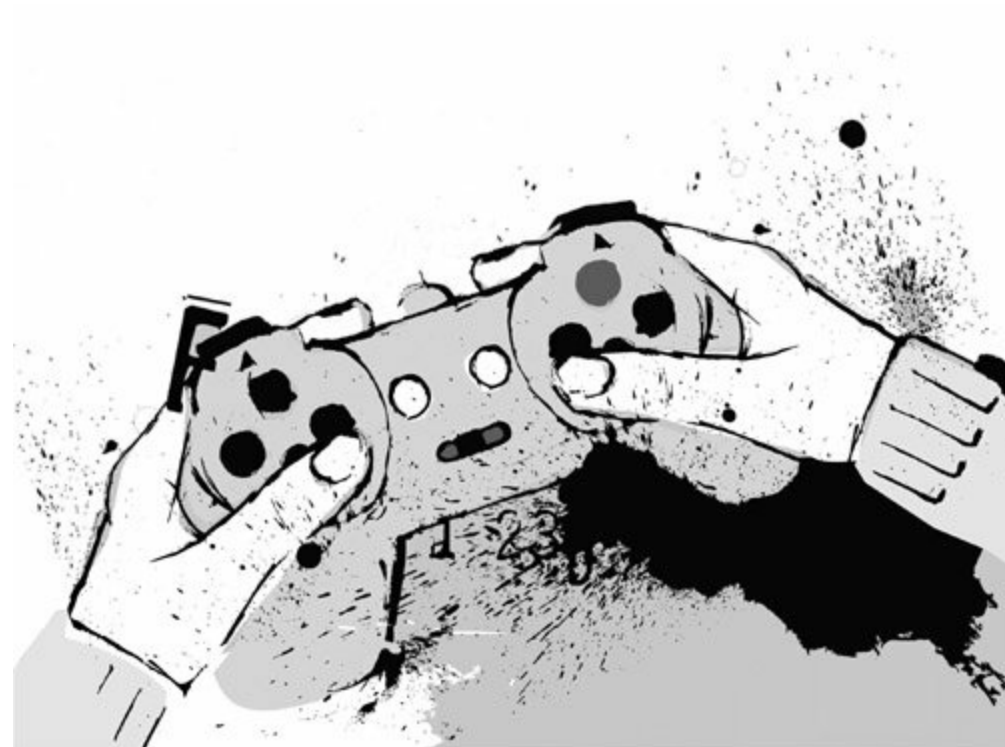
Pointers and arrays are closely related. You can use array notation with pointers. Doing so often makes pointers easier to manage.

# Chapter 12. File Input and Output

This chapter provides an overview of how to read information from files into your programs. It also explains how to write information out to files. Programmers call this file input and output, or just file I/O.

Nearly all programs perform file I/O, even if it's only a small amount. Games are no exception. In fact, games often do large amounts of file I/O. As a result, reading from and writing to files is an essential skill for every game programmer.

In this chapter, you'll learn the basics of file I/O. First, the chapter discusses what games use file I/O for. Next, it describes the types of files you can read and write. Finally, it demonstrates basic I/O techniques. These techniques prepare you for the file I/O tasks you'll perform in [Part 5](#), "The Big Payoff." In [Part 5](#), you'll see how to do such tasks as reading level files and saving a player's progress to a file.



# Games and File I/O

Other than a game's program file, all types of files that a game uses must either be read from or written to. For example, games often need configuration files. Reading the configuration file is done by performing file I/O.

Games process a huge amount of information that they get from reading level files, which you learned about in [chapter 11](#), "Pointers." Each level has its own monsters, enemies, bad guys, or other type of opponents. Level files specify exactly what opponents occur in a level and where they are positioned when they appear. They also usually tell the program which files to read to load the bitmap for the opponent's sprite. In addition, they specify the sound files to play when the opponent makes any kind of noise. Level files can contain information about which music files to play during the level. Your game must read the level file, the bitmap files, and the sound files into memory.

Games must save the player's progress. To do so, they write the player's current progress information to a file. Game programmers often call this file the [save file](#).

If you use a game engine, it will usually handle a lot of the file I/O tasks that your game needs to perform. For example, LlamaWorks2D reads bitmap images for sprites when your game calls the `sprite::LoadImage()` function. It also performs file I/O when your program invokes the `sound::LoadWAV()` function.

As helpful as they are, game engines cannot do all of your file I/O for you. Only you know exactly what you want in your level files. You are also the only one who can determine what goes into a save file. Therefore, you must become proficient with file I/O. Fortunately, it isn't difficult to grasp.

There is one thing you need to keep in mind relative to file I/O: file I/O is always slow. That's because disk drives, such as CD-ROMs, DVD-ROMs, and even hard drives operate *much* slower than a computer's microprocessor. Because file I/O is so slow, it may cause the game to actually pause for a moment. Of course, you don't want that to happen while a level is running; it gets players quite irritated. Therefore, your game must perform its file I/O between levels. Players expect a natural pause before and after each level, so your game can do its file I/O between levels without making them upset.

# Types of Files

Computers only provide two types of files: text files and binary files. Programs use text files to store text, HTML, XML, C++ source code, and many other types of information. They use binary files to store word processing documents, spreadsheets, images, sounds, and so forth.

Games use both text and binary files. Therefore, the remainder of this chapter demonstrates how to read and write information using both text and binary files.

## Text Files

A text file is just what it sounds like. It's a file that can only hold text. These files can't hold just any character; the text must be from the set of characters specified by the [ASCII character set](#). The ASCII character set is a standard set of characters used by all computer manufacturers. ASCII stands for the American Standard Code for Information Exchange. Because it's one of the oldest standardized character sets, all computers recognize the ASCII character set, which you'll find on the CD on the main HTML page that is displayed when you insert it into your CD-ROM drive.

The ASCII character set contains characters for all the letters of the English alphabet. In addition, it has characters for the numeric digits 09, some punctuation marks, and other common characters.

### Note

The mainframe computers made by IBM do not use the ASCII character set. Mainframes are huge computers that are used for business computing, not games. Therefore, we won't deal with them in this book.

## Writing to Text Files

Back in [chapter 2](#), "Writing C++ Programs," I introduced the notion of streams. Recall that the C++ Standard Library defines the streams `cin` and `cout` for you. The C++ Standard Library also provides stream classes for file input and output. However, unlike `cin` and `cout`, streams for files are not created automatically. Your program must create them itself.

To create a stream for text file output, your program must have access to the class `ofstream`. It can get access by including the C++ Standard Library header file `fstream`. Your program then declares a variable of type `ofstream`, which stands for output file stream. After the stream has been created, your program can write to it using the insertion operator. In that respect, it's just like writing to `cout`. [Listing 12.1](#) demonstrates how to write to a text file.

## Listing 12.1. Using streams to output text

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4
5  using namespace std;
6
7  int main(int argc, char *argv[])
8  {
9      std::string textArray[] =
10     {
11         "Jeeves--my man, you know--is really a most extraordinary ",
12         "chap. So capable. Honestly, I shouldn't know
13         what to do ",
14         "without him. On broader lines he's like those chappies who",
15         " sit peering sadly over the marble battlements at the ",
16         "Pennsylvania Station in the place marked \"Inquiries.\" ",
17         "You ,know the Johnnies I mean. You go up to them and say:",
18         " \"When's the next train for Melonsquashville, ",
19         "Tennessee?\" and they reply, without stopping to think,",
20         " \"Two-forty-three, track ten, change at San Francisco.\" ",
21         "And they're right every time. Well, Jeeves gives you just ",
22         "the same impression of omniscience.",
23         "My Man Jeeves, P.G. Wodehouse, 1919"
24     };
25
26     cout << "Opening file...";
27     std::ofstream outputFile("Jeeves.txt");
28     if (outputFile)
29     {
30         cout << "Done" << endl;
31
32         cout << endl << "Writing text to the file Jeeves. txt...";
33         int i = 0;
34         while (i<12)
35         {
36             outputFile << textArray[i] << endl;
37             i++;
38         }
39     }
40 }
```

```
37     };
38     cout << "Done" << endl;
39
40     cout << endl << "Closing Jeeves.txt...";
41     outputFile.close();
42
43     cout << "Done" << endl << endl;
44 }
45 else
46 {
47     cout << "Could not open file." << endl;
48 }
49
50 system("PAUSE");
51 return EXIT_SUCCESS;
52 }
```

Near the beginning of the program in [Listing 12.1](#) (see line 3), the program includes `fstream`, which is provided by the C++ Standard Libraries. On lines 923, the program declares and initializes an array of strings named `textArray`.

After the program declares the string array, it outputs a message to the screen on line 25. It then declares a variable of type `ofstream` on line 26. In the declaration, it passes the name of the file to open to the `ofstream` constructor. This program opens a file called `Jeeves.txt`. When you run this program, it creates the file on your hard drive in the same folder that the executable version of this program (`Prog_12_01.exe`) resides in.

## Factoid

Most text files have the filename extension `.txt`. However, nothing forces them to have that extension. They can also have extensions such as `.xml`, `.html`, `.cpp`, `.h`, `.ini`, `.cfg`, or anything else you think is appropriate; a text editor can open them all.

It is possible for the file to fail to be created and opened. The typical reasons for this failure are that the disk is either full or damaged, or that you do not have write permissions. If you're running this program from a hard drive, it is not likely that the disk is full. It is either damaged or you do not have write permissions.

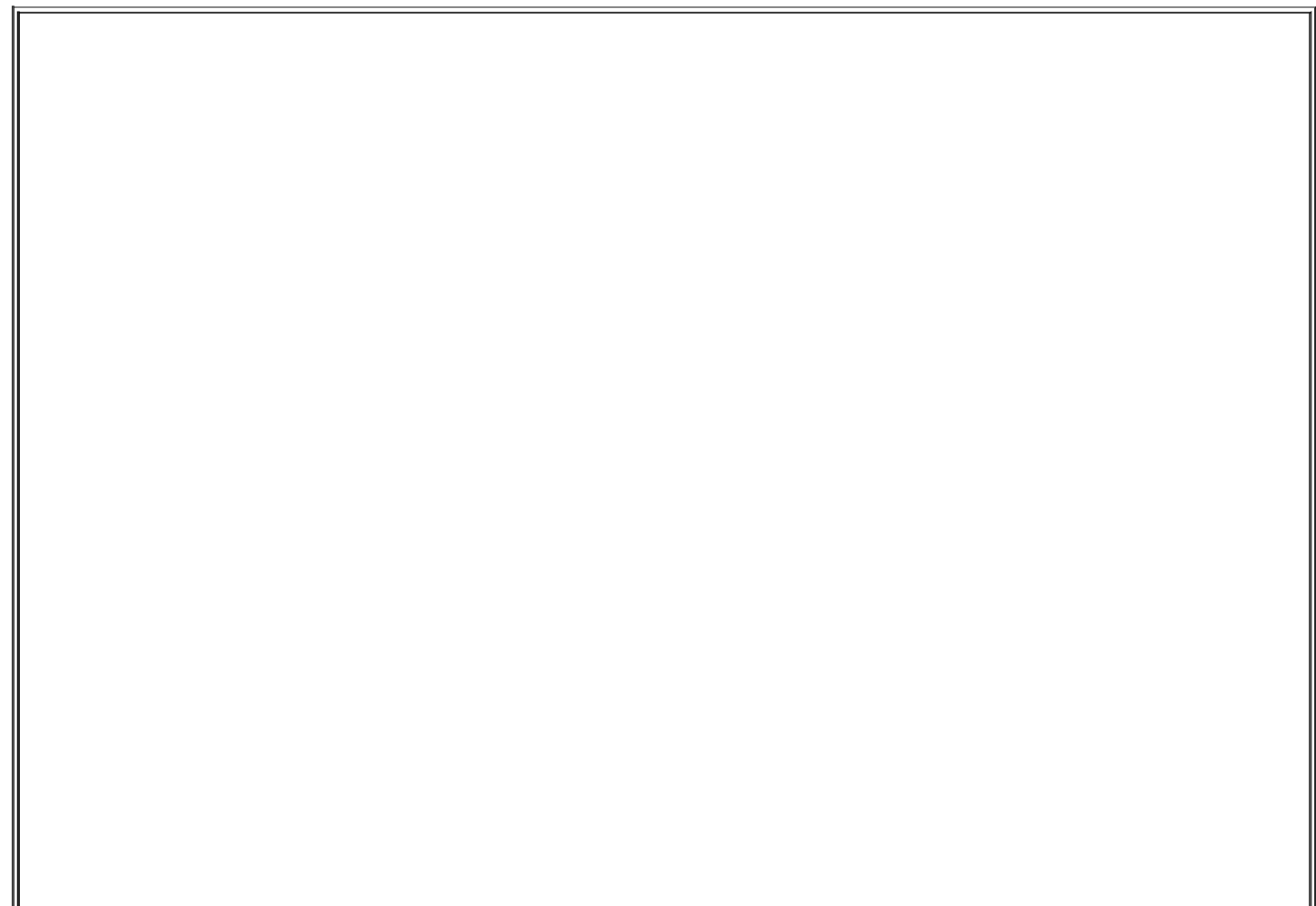
It is also quite unlikely for the creation of this file to fail due to a hard drive failure. More typically, it is because you don't have write permissions. The

most common reason for that is trying to write to a CD or DVD. CDs and DVDs are read-only disks. That means you can't create files on them after they have been burned.

For example, if you run this program from the CD that comes with this book, the program will try to create Jeeves.txt on the CD. That will fail. When it does, the variable `outputFile` declared on line 26 will evaluate to `NULL`. Therefore, you can use it as the condition in an `if` statement. That's exactly what happens on line 27. If the value of `outputFile` is not `NULL`, the `if` statement evaluates to `TRue`. In that case, it executes the body of the `if` statement on lines 28-44.

The body of the `if` statement begins by outputting status messages on lines 29-31. On lines 32-37, it uses a `while` loop to write each string in the array to a text file. The statement that does the actual writing appears on line 35. Notice that it is very similar to writing text to the screen. You simply use the insertion operator. The only real difference is that you don't write to `cout`. Instead, you write to the `ofstream` variable `outputFile`.

When the loop on lines 32-37 is finished, the program outputs some status messages to the screen on lines 38-40. It then uses the `close()` function, which is a member of the `ofstream` class, to close the stream. After that, your program can't write to the file any more without opening the stream again.



## Beyond English Characters

It is possible to create text files that use additional characters that are not defined in the ASCII character set. This capability is absolutely essential for games produced in languages other than English. Note that the A in ASCII stands for American. The first computer manufacturers were all located in the United States. When they got together and defined the ASCII set, they included only characters for English.

These days, computers handle a much broader set of characters than just the ASCII characters. For example, there are kinds of text files that can hold characters for virtually any kind of writing system. This includes alphabets that use Roman characters, such as English, French, German, or Spanish, and characters that are completely unlike Western writing, such as Chinese or Japanese characters.

Having said all that, the most basic kind of text file holds only characters in the ASCII set. Because this is an introductory book, that's all I'll cover. After you've mastered the essentials of file I/O, learning to use a broader range of character sets is not difficult. The information on how to do it is available on the Web for free. The best place to start looking for information on standard international character sets is at [www.wikipedia.org](http://www.wikipedia.org). Do a search on the keyword "Unicode" and you'll get a good introduction.

If the file could not be created, the program executes the `else` statement beginning on line 45. The `else` statement contains a statement that outputs an error message to the screen.

You may ask whether you can only write strings to text files. The answer is no. You can write any type of data that can be converted to strings. For example, you could write the contents of an `int` or `float` variable to a text file. The insertion operator knows how to convert integers and floating-point numbers to text. That's what happens when you write integers and floating-point numbers to the screen. Anything you can write to the screen with the insertion operator, you can write to a text file with the insertion operator.

### Warning

Always close all streams that you open. If you don't, it is possible to lose data on some devices. This won't happen on personal computers, but it can happen on some smaller game machines such as consoles or handheld devices. It's safest to just get into the habit of closing all streams you open.



## Reading from Text Files

As with writing to text files, your programs use streams to read from text files. They create input streams by declaring variables of type `ifstream`, which stands for input file stream. [Listing 12.2](#) shows how to declare and use an input stream.

### Listing 12.2. Using streams to input text

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4
5  using namespace std;
6
7  int main(int argc, char *argv[])
8  {
9      ifstream inputFile;
10
11     cout << "Opening input file...";
12     inputFile.open("Input.txt");
13
14     if (inputFile.is_open())
15     {
16         cout << "Done" << endl << endl;
17
18         char inputString[80];
19
20         cout << "Reading input file..." << endl;
21         cout << endl;
22         int i=0;
23         while (!inputFile.eof())
24         {
25             inputFile.getline(inputString,80,'\n');
26             cout << inputString << endl;
27             i++;
28         }
29         cout << endl;
30     }
31     else
32     {
33         cout << "Could not open input file." << endl;
34     }
35     system("PAUSE");
36     return EXIT_SUCCESS;
37 }
```

Like the program in [Listing 12.1](#), the program in [Listing 12.2](#) declares a stream variable. The `ifstream` variable `inputFile` enables the program to open and read a text

file.

To demonstrate an alternate method of opening the file, I did not pass the file name to the `ifstream` constructor. Instead, the program calls the `open()` function on line 12 of [Listing 12.2](#). This function exists in both the `ifstream` and `ofstream` classes, so you can call it to open either input or output files. When your program calls the `open()` function, it passes the filename in the parameter list.

## Note

Some compilers require an explicit conversion from `char *` to `string`. But that's not something I want to get into.

In [Listing 12.1](#), the program tested the stream to see if it was opened by testing the variable name. The C++ I/O classes provide another way to accomplish the same thing. On line 14 of [Listing 12.2](#), the program calls the `is_open()` function on the input stream. Output streams also provide this function. So whether your program is testing input or output streams, it can see if they're open by calling `is_open()`. If the program was able to open the stream, `is_open()` returns `true`. If not, it returns `false`.

If the file was opened, the program prints a status message on line 16 and then declares an array of characters. C++ uses character arrays as strings. This is a hold-over from the C language.

In most cases, using the C++ Standard Library's `string` class is preferable to using character arrays. However, in some cases, you must use character arrays. This happens to be one of them. A look at line 25 shows why.

On line 25, you can see that the program does not use the extraction operator to read input from the file. Instead, it uses an `ifstream` member function named `getline()`. The `getline()` function reads characters into the character array specified in its first parameter. Because this parameter requires a character array, you cannot use variables of type `string` with the `getline()` function.

## Factoid

The C++ programming language is derived from the older C programming language.

## Note

There is a way to make the `getline()` function work with variables of type string, but explaining it would take us too far off our current subject. You can find information on it in the C++ programming books in the list of suggested reading on the CD.

Character arrays have a specific size. In the program in [Listing 12.2](#), the array is 80 characters long. Because you use character arrays with `getline()`, you must specify the maximum number characters to read for each line. Otherwise, `getline()` could potentially read more characters than will fit in the array. That is likely to cause your program to crash. The 80 on line 25 ensures that `getline()` reads no more than 80 characters. The last one is always a null character, which is represented in C++ by the `'\0'` character. The null character is considered one character even though you use both a backslash and a zero to represent it. Because `getline()` always puts a `'\0'` character at the end of the input string, it always reads one less input character than you tell it to. Specifying 80 as the second parameter to `getline()` makes `getline()` read 79 characters and append a `'\0'` at the end.

## Note

In the C programming language, all strings had to have the `'\0'` character at the end to mark the end of the string. That convention is often followed in C++ as well.

The third parameter to the `getline()` function is a delimiter character. When `getline()` finds the delimiter character in the text it's reading, it stops reading input and returns. The character `'\n'` is actually one character, not two. When you're specifying one literal character, like an `'A'`, you always use single quotes rather than double quotes (double quotes are for strings). In any case, the character `'\n'` stands for a newline, which is the same as an endline. Using the `'\n'` character as the delimiter means that the `getline()` function reads until it encounters the end of the line. Every line of text in a text file contains a `'\n'`

character at the end. The combination of the second and third parameters to the `getline()` function means that `getline()` will read text from the text file until it either reads 80 characters or encounters the end of the line. If either one of those conditions occurs, it stops reading and returns.

If you wanted, you could change line 25 to read as follows:

```
inputFile >> inputString;
```

If you did this, the program would execute very differently. When your program uses the extraction operator to read from text files, it stops any time it encounters white space. White space includes endlines, spaces, and tabs. If the words in your text file are separated by spaces, the extraction operator will read one word at a time rather than one line of text at a time. This is a very important difference to most games. It's much more typical for games to read one line at a time rather than one word at a time.

## Factoid

The character `'\n'`, which stands for newline, is as another name for an endline. It's a holdover from the C language.

## Binary Files

The other type of file is called a *binary file*. Binary files can contain any type of data in binary format. As you may recall, I mentioned binary numbers briefly in [chapter 1](#), "What It Takes to Be a Game Programmer." I explained that you do not have to learn the binary number system to write games (although it does help if you do). The same is true of using binary files. You do not have to learn binary to read from or write to binary files.

Unlike text files, binary files can contain any kind of data. Any information you can store in memory can be stored in a binary file. This includes simple data types such as integers and floating-point numbers, or complex types such as objects. That's right you can save entire objects to binary files.

Because binary files can hold any type of data, the information they store tends to be more complex than the information stored in text files. Games typically use binary files for their level and save files. They may use text files for their configuration files.

## Writing to Binary Files

When your program writes data to binary files, it treats the data it's writing as a group of bytes. To write a group of bytes to a file, your program must call the `write()` function, which is a member of the `ofstream` class, and pass it a pointer to the data to be written. It must also tell the `write()` function how many bytes to write. [Listing 12.3](#) gives a short example of how to do this.

### Listing 12.3. Writing a binary save game file

```
1 #include <cstdlib>
2 #include <iostream>
3 #include <fstream>
4
5 using namespace std;
6
7 class game_save_info
8 {
9 public:
10     game_save_info();
11
12     void CurrentLevel(int levelNumber);
13     int CurrentLevel();
14
15     void CurrentScore(int score);
16     int CurrentScore();
17
18     void PlayerName(char name[80]);
19     std::string PlayerName();
20
21 private:
22     int currentLevel;
23     int currentScore;
24     char playerName[80];
25 };
26
27
28 game_save_info::game_save_info()
29 {
30     currentLevel = 0;
31     currentScore = 0;
32 }
33
34 void game_save_info::CurrentLevel(int levelNumber)
35 {
```

```

36     currentLevel = levelNumber;
37 }
38
39 int game_save_info::CurrentLevel()
40 {
41     return (currentLevel);
42 }
43
44 void game_save_info::CurrentScore(int score)
45 {
46     currentScore = score;
47 }
48
49 int game_save_info::CurrentScore()
50 {
51     return (currentScore);
52 }
53
54 void game_save_info::PlayerName(char name[80])
55 {
56     strcpy(playerName, name);
57 }
58
59 std::string game_save_info::PlayerName()
60 {
61     return (playerName);
62 }
63
64 int main(int argc, char *argv[])
65 {
66     game_save_info saveGame;
67
68     saveGame.CurrentLevel(3);
69     saveGame.CurrentScore(11042);
70     saveGame.PlayerName("Ozymandius Jones");
71
72     cout << "Opening save game file...";
73     ofstream saveFile;
74     saveFile.open("save.dat", ios::binary);
75     if (saveFile.is_open())
76     {
77         cout << "Done." << endl;
78
79         cout << "Writing save game info to file...";
80         saveFile.write(
81             (char *)&saveGame,
82             sizeof(game_save_info));
83         cout << "Done." << endl;
84     }
85
86     system("PAUSE");
87     return EXIT_SUCCESS;
88 }

```

The first thing that the program in [Listing 12.3](#) does is define a class called `game_save_info`. This is a simplified version of a class you might create to write

information to a file that enables players to save their progress in a game. This simple class has only a few items of member data, a constructor, and some functions for accessing the member data.

Notice that the player name is a character array rather than an object of type `string`. When you write data to files, it is a much simpler process if you write things that have a fixed size. A `string` object does not. The size of a `string` object depends on how many characters it holds. The character array defined on line 24 is always exactly 80 characters long. Even if the string it holds is shorter, the array is still 80 characters long. Because it's a known, fixed size, it's easier to write to a file than a `string` object would be.

## Tip

It is not uncommon for C++ programs to require the use of functions from the old C Standard Library. As you become more proficient in C++ programming, you may want to get a good reference book on C programming. Alternatively, you could use the C/C++ Language Reference on the Web at [www.msdn.microsoft.com](http://www.msdn.microsoft.com).

Because the `game_save_info` object contains a character array, it must use the old C Standard Library `strcpy()` function in the `PlayerName()` function on lines 59-62. The name "strcpy" stands for "string copy." The `strcpy()` function, which was heavily used in C programs, copies the contents of one character array to another. In the case of line 56, it copies the characters in the parameter `name` to the data member `playerName`. Both of these are arrays of no more than 80 characters.

In the `main()` function, this program declares an object of type `save_game_info` on line 66. It then stores some information in the object. It will write that information to a file shortly.

After outputting a status message, the program declares a variable of type `ofstream` on line 73. In the program back in [Listing 12.1](#), I passed the output file name to the `ofstream` constructor. Just to demonstrate that you can also use the `open()` function with output streams, this program calls `open()` on line 74. Notice that it passes two parameters to `open()` this time. The first is a string containing the file name. The second is a constant that is defined in the C++ Standard Library. The name of the constant is `ios::binary`. Passing the constant `ios::binary` to the `open()` function tells `open()` to create a binary file rather than a text

file.

On line 75, the program tests whether the file was successfully opened by calling the `is_open()` function. If it was, the program outputs some status messages to the screen.

Next, the program calls the `ofstream` class's `write()` function on line 80 to write the contents of the variable `saveGame` to the binary file. The first parameter to the `write()` function is always a pointer to characters. C++ treats characters as bytes, so a pointer to characters is another way of saying a pointer to bytes. The variable `saveGame` is not a pointer to characters. It is an object of type `save_game_info`. To pass the information in `saveGame` to the `write()` function, the program must convert `saveGame` to a pointer to characters. It does that in specific steps. First, it puts the variable name in parentheses and puts the ampersand in front of it. Recall from our discussion of pointers in [chapter 11](#) that the ampersand operator takes the address of a variable. In other words, the statement `&saveGame` obtains a pointer to the variable `saveGame`.

Whenever you pass `write()` a variable, rather than a pointer, you *must* use the ampersand operator to convert the variable to a pointer. The type of the variable does not matter. This rule holds true for simple types such as `int` or classes such as `save_game_info`.

## Note

The formal name for converting one type to another by inserting the destination type in parentheses (as shown on line 81 of [Listing 12.3](#)) is *type casting*.

Using the ampersand with the variable name obtains the address of the variable. Another way of saying that is that it gets a pointer to the variable. In order to pass the pointer to the `write()` function, the program must convert the pointer to a character pointer. To do so, it uses the statement `(char *)` at the beginning of line 81. So the `(&saveGame)` gets a pointer to the type `game_save_info` and the `(char *)` converts it to a character pointer for that statement only. Therefore, `write()` receives a pointer to a group of characters (which it treats as a pointer to a group of bytes) as its first parameter. That is exactly what it needs.

The second parameter to the `write()` function appears on line 82. The `write()`



function requires an integer as its second parameter that specifies how many bytes to write to the file. To figure that out, the program uses the C++ `sizeof()` operator. The `sizeof()` operator takes a type name in its parentheses. It calculates how many bytes a variable of that type occupies in memory. In the case of line 82, it calculates the number of bytes required for a variable of type `game_save_info`. When you write an object to a file, only the data gets written, not the member functions. So the `sizeof()` operator simply adds up the sizes of the member data to get the final total.

This program doesn't provide a way to check that everything was written to the file properly. The only way we can really verify that is to read the contents of the file back into memory and print them to the screen. That's the subject of the next section.

## Note

Did you spot the mistake in the program in [Listing 12.3](#)? It's a small one, but I put it in as a reminder. The mistake is that the program never closes the file it opens. This is poor programming practice and should never be done in a professional program.

## Reading from Binary Files

Programs reading data from binary files by calling the `ifstream::read()` function. The `read()` function works nearly identically to the `ofstream::write()` function except that the data flows into the program instead of out to a file. [Listing 12.4](#) shows how to use it.

### Listing 12.4. Using the `read()` function to read binary data

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <fstream>
4
5  using namespace std;
6
7  class game_save_info
8  {
9  public:
```

```
10  game_save_info();
11
12  void CurrentLevel(int levelNumber);
13  int CurrentLevel();
14
15  void CurrentScore(int score);
16  int CurrentScore();
17
18  void PlayerName(char name[80]);
19  std::string PlayerName();
20
21  private:
22      int currentLevel;
23      int currentScore;
24      char playerName[80];
25  };
26
27
28  game_save_info::game_save_info()
29  {
30      currentLevel = 0;
31      currentScore = 0;
32  }
33
34  void game_save_info::CurrentLevel(int levelNumber)
35  {
36      currentLevel = levelNumber;
37  }
38
39  int game_save_info::CurrentLevel()
40  {
41      return (currentLevel);
42  }
43
44  void game_save_info::CurrentScore(int score)
45  {
46      currentScore = score;
47  }
48
49  int game_save_info::CurrentScore()
50  {
51      return (currentScore);
52  }
53
54  void game_save_info::PlayerName(char name[80])
55  {
56      strcpy(playerName, name);
57  }
58
59  std::string game_save_info::PlayerName()
60  {
61      return (playerName);
62  }
63
64  int main(int argc, char *argv[])
65  {
66      game_save_info saveGame;
67
68      cout << "Opening save game file...";
69      ifstream saveFile;
70      saveFile.open("save.dat", ios::binary);
```

```

71  if (saveFile.is_open())
72  {
73      cout << "Done." << endl;
74
75      cout << "Reading save game info from file...";
76      saveFile.read(
77          (char *)&saveGame,
78          sizeof(game_save_info));
79      cout << "Done." << endl;
80
81      cout << endl;
82      cout << "Player Name: " << saveGame.PlayerName();
83      cout << endl;
84      cout << "Player Score: " << saveGame.CurrentScore();
85      cout << endl;
86      cout << "Current Level: " << saveGame.CurrentLevel();
87      cout << endl;
88
89      saveFile.close();
90  }
91
92
93  system("PAUSE");
94  return EXIT_SUCCESS;
95  }

```

This program is very similar to the one in [Listing 12.3](#). The difference is in the `main()` function. After declaring a variable to hold the data that will come in from the file, the `main()` function opens the data file written by the program in [Listing 12.3](#). If the file is opened successfully, the program uses the `ifstream::read()` function on lines 76-78.

As with the `ofstream::write()` function, the first parameter to `ifstream::read()` is a character pointer. The pointer is the address of the memory that will receive the data to be read from the file. The second parameter to the `read()` function is the number of bytes to read. Because the program is reading an object of type `game_save_info`, it uses the statement `sizeof(game_save_info)`. This tells the program the number of bytes that a `game_save_info` object requires.

That is exactly the number of bytes the program should read from the file. When the `read()` function executes, it reads the number of bytes specified in its second parameter into the memory location contained in its first parameter.

After the program reads the data back into memory, it prints the data to the screen on lines 81-87. It then closes the data file before it exits. [Figure 12.1](#) shows what the output looks like.

**Figure 12.1. The contents of the object read in from the file**

# Save.dat.

```
Opening save game file...Done.  
Reading save game info from file...Done.  
Player Name: Ozymandius Jones  
Player Score: 11042  
Current Level: 3  
Press any key to continue . . . _
```

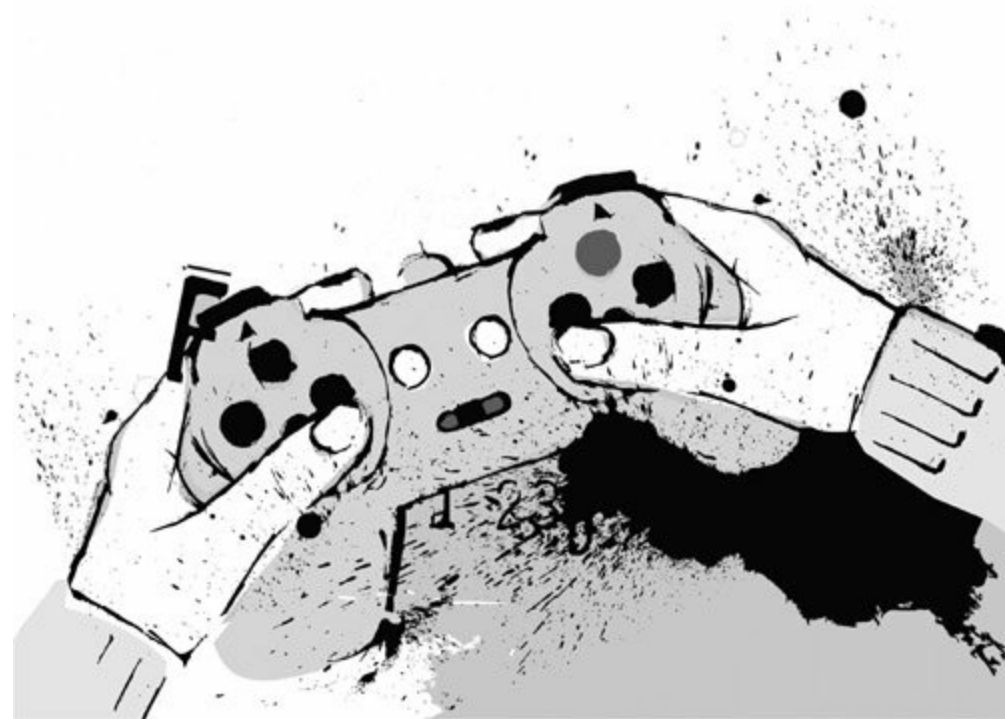
# Summary

File I/O is a crucial skill for game programmers. In this chapter, we provided an overview of the most important aspects of file I/O. You saw that files can hold either text or binary data. C++ provides a variety of operators and functions for file I/O, and it is important that you get as much practice with them as you can.

# Chapter 13. Moving into Real Game Development

You've covered a lot of ground since you started this book. As you've read the chapters and examined the sample programs, you've acquired most of the essential game programming skills you need. Yes, there are still a lot of fancy graphics and sound techniques you could learn. However, you have most of the basics. There are really only two more things you must learn to start producing professional-quality games: how to make animated sprites and how to get player input quickly.

This chapter introduces techniques for displaying and using animated sprites. It also covers a method of performing the type of high-speed player input that most games require.



# Sprites that Come Alive

Sprites move around the screen, and as they do they typically change appearance. Imagine you're making a game called Mole Thumper. The object of the game is to guide a farmer around the screen and have him thump moles on the head so they don't eat his corn. As the farmer moves, he must appear to walk. When he thumps the moles, he must swing his mallet down on their pesky little heads. The moles must react to being thumped. All of this means that the farmer and the moles must change appearance as the game progresses. That's precisely the purpose of animated sprites.

## Sprites and Animation Frames

To create an animated sprite, you must imagine a sprite as a rectangular movie screen that moves around the computer's display. Because the sprite is like a miniature movie screen, your game can display successive images on the sprite's "screen" to give the appearance of movement and change.

For example, suppose you want to display a farmer walking across the screen. To do so, you display a succession of images of the farmer with his legs and arms in a slightly different position. Each successive image is one frame of the sprite's animation. Every frame of sprite animation has to fit into the animation frames of the screen as a whole. In other words, to get the farmer to walk across the screen, your animated sprite must display the first frame of its animation in a frame of screen animation. The sprite then draws the next frame of its animation into the next screen frame. The sprite continues in this way until it reaches the end of its animation frames. It then repeats the animation to continue the appearance of the farmer walking.

In professional games, there are typically many sprites displayed on the screen at once. During each frame of screen animation, each sprite busily displays one frame of its animation. The result is that, during each frame of screen animation, all of the sprites change appearance. The movement and the variation is what helps make the game come alive.

## Animated Sprites in LlamaWorks2D

LlamaWorks2D provides two classes that enable you to easily add animated sprites to your games. The first is the `animated_sprite` class. The `animated_sprite` class acts like a regular sprite in that it moves around the screen. In addition, each

`animated_sprite` object can contain one or more animation. An animation is implemented in a class called `animation`. An `animation` object holds the bitmap images that it uses as the frames of its animation.

Objects of type `animated_sprite` can have multiple animations. In our mole thumping game, the `animated_sprite` object the game uses to represent the farmer needs animations of the farmer walking left and right. It also needs animations of the farmer swinging his mallet. It may have an animation of the farmer crying when the moles get all his corn, or one of him dancing when he thumps the last mole on the level. The ability to hold multiple animations in an `animated_sprite` object enables you to create a wide range of behaviors for your game's characters.

To build an animated sprite, your game begins by creating at least one `animation` object. It loads frames into the animation. Next, your program creates an `animated_sprite` object and adds the `animation` object to it. It can add more animations as necessary. Only one animation can be displayed at a time. Therefore, one of the animations in the animated sprite is always considered the current animation. By default, the last animation you add to the sprite is the current animation. However, you can select any animation that the `animated_sprite` object contains to be the current animation.

Animations in LlamaWorks2D can loop. This enables your game to display motion that looks continuous. The walking farmer is an example. As the farmer walks across the screen, his animation loops over and over until he changes direction or thumps a mole. Then a different animation plays.

## Note

Looping frames in forward and then reverse order is very common. When your game does this, it displays each frame of the sprite from first to last. It then displays them from last to first. This style of looping is usually what is used to make characters walk.

Your game can display animations that loop forward, reverse, or forward and then reverse. Forward looping means that the animation is displayed from the first frame to the last frame. The animation then starts over with the first frame. Reverse animation is just the opposite. The animation starts with the last frame and then displays each frame in reverse order until it gets to the first frame. It then starts over. This enables your game to do some nice effects



very simply.

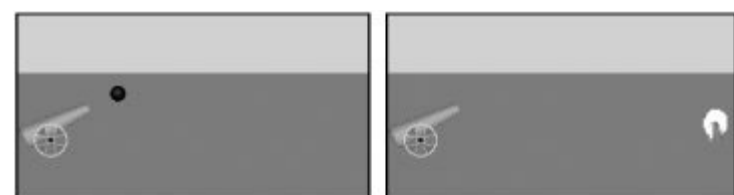
## Cannonshoot Revisited

Let's dive into a sample program to see how animated sprites work. We'll use a revised version of the cannon simulator program from [chapter 9](#). In this revised version, the program displays an animation of an explosion when the cannonball hits the ground.

If you look at the files that make up the CannonShoot program, you'll see that in addition to the LlamaWorks2D files, CannonShoot includes the files CannonShoot.h and CannonShoot.cpp. These files contain the logic of how the game works. However, CannonShoot.h and CannonShoot.cpp do not need to be changed in order to add animated sprites. Instead, the animated sprite is added to the `cannonball` object, which is in the files Cannonball.h and Cannonball.cpp. Let's begin by looking at the new version of Cannonball.h in [Listing 13.1](#).

[Listing 13.1](#) presents part of the file Cannonball.h. To save space, I've omitted the parts that have not changed since [chapter 9](#). On line 48, you can see that there's a new data member of type `animated_sprite`. This is in addition to the `sprite` object declared on line 47. While the cannonball is in flight, the `cannonball` object displays the `sprite` object that contains the image of the ball. When the cannonball hits the ground, the `cannonball` object displays the `animated_sprite` object. This causes the animation of the explosion to be played, as shown in [Figure 13.1](#).

**Figure 13.1. When the cannonball is in flight (left frame), the program uses a sprite object because it doesn't change appearance. After the cannonball hits the ground (right frame), the program uses an animated sprite to display the explosion.**



## NOTE

You can find Cannonball.h on the CD in the folder Source\Chapter13\Prog\_13\_01.

### Listing 13.1. Adding an animated sprite to the **cannonball** class

```
1 class cannonball
2 {
3 public:
4     cannonball();
5
6     bool LoadImage(
7         std::string fileName,
8         image_file_base::image_file_format fileFormat);
9
10    bool LoadExplosionAnimation();
11
12    bool LoadHitSound(
13        std::string fileName);
14
15    void BallHit();
16    bool DidBallHit();
17
18    bool Move();
19    bool Render();
20
21    void X(
22        float upLeftX);
23    float X();
24
25    void Y(
26        float upLeftY);
27    float Y();
28
29    int BitmapHeight();
30    int BitmapWidth();
31
32    void BoundingRectangle(
33        bitmap_region rectangle);
34    bitmap_region BoundingRectangle();
35
36    void BitmapTransparentColor(
37        color_rgb theColor);
38    color_rgb BitmapTransparentColor();
39
40    void Velocity(
41        vectorf direction);
42    vectorf Velocity();
43
44    bool BallInFlight();
```

```

45
46 private:
47     sprite theBall;
48     animated_sprite ballExplosion;
49     vectorf velocity;
50     float x, y;
51     bool ballInFlight;
52     bool ballHit;
53     sound ballHitSound;
54 };

```

In addition, there's a new data member on line 52 that keeps track of when the cannonball hits something. There is also a new member function in the `cannonball` class on line 16 of [Listing 13.1](#). It tells other classes when the cannonball has hit something. The `cannonball` class contains a new member function for loading the explosion animation on line 10.

Most of the member functions of the `cannonball` do not need to change. However, those that do, along with the new member functions, are shown in [Listing 13.2](#).

## Listing 13.2. New and updated member functions for the `cannonball` class

```

1  cannonball::cannonball() :
2      velocity(0.0f,0.0f)
3  {
4      ballInFlight = false;
5      ballHit = false;
6  }
7
8
9  bool
10 cannonball::LoadExplosionAnimation()
11 {
12     bool loadOK = true;
13     animation *theAnimation = new animation;
14
15     if (theAnimation!=NULL)
16     {
17         theAnimation>BitmapTransparentColor(
18             color_rgb(0.0f,1.0f,0.0f));
19
20         loadOK = theAnimation>LoadImage(
21             "Expl1.bmp",
22             image_file_base::LWIFF_WINDOWS_BMP);
23     }
24     else
25     {
26         loadOK = false;

```

```
27     theApp.AppError(LWES_OUT_OF_MEMORY);
28 }
29
30 if (loadOK)
31 {
32     loadOK = theAnimation>LoadImage(
33         "Exp12.bmp",
34         image_file_base::LWIFF_WINDOWS_BMP);
35 }
36
37 if (loadOK)
38 {
39     loadOK = theAnimation>LoadImage(
40         "Exp13.bmp",
41         image_file_base::LWIFF_WINDOWS_BMP);
42 }
43
44 if (loadOK)
45 {
46     loadOK = theAnimation>LoadImage(
47         "Exp14.bmp",
48         image_file_base::LWIFF_WINDOWS_BMP);
49 }
50
51 if (loadOK)
52 {
53     loadOK = theAnimation>LoadImage(
54         "Exp15.bmp",
55         image_file_base::LWIFF_WINDOWS_BMP);
56 }
57
58 if (loadOK)
59 {
60     loadOK = theAnimation>LoadImage(
61         "Exp16.bmp",
62         image_file_base::LWIFF_WINDOWS_BMP);
63 }
64
65 if (loadOK)
66 {
67     loadOK = theAnimation>LoadImage(
68         "Exp17.bmp",
69         image_file_base::LWIFF_WINDOWS_BMP);
70 }
71
72 if (loadOK)
73 {
74     theAnimation>LoopAnimation(
75         true,
76         animation::FORWARD_THEN_REVERSE);
77 }
78
79 if (loadOK)
80 {
81     // Add the animation to the animated sprite.
82     ballExplosion.Animation(theAnimation);
83 }
84
85 return (loadOK);
86 }
87
```

```

88
89
90 void
91 cannonball::BallHit()
92 {
93     velocity.X(0.0f);
94     velocity.Y(0.0f);
95     ballHitSound.Play();
96     ballInFlight = false;
97     ballHit = true;
98 }
99
100
101 bool
102 cannonball::DidBallHit()
103 {
104     return (ballHit);
105 }
106
107
108 bool
109 cannonball::Render()
110 {
111     bool renderOK = true;
112
113     if (ballInFlight)
114     {
115         renderOK = theBall.Render();
116     }
117     else if (ballHit)
118     {
119         animation::animation_status explosionStatus =
120             ballExplosion.AnimationStatus();
121         if ((explosionStatus != animation::ANIMATION_ERROR) &&
122             (explosionStatus != animation::AT_END_OF_LOOP))
123         {
124             ballExplosion.X(theBall.X());
125             ballExplosion.Y(theBall.Y());
126             renderOK = ballExplosion.Render();
127         }
128         else if (explosionStatus == animation::AT_END_OF_LOOP)
129         {
130             ballHit = false;
131             ballExplosion.ResetAnimation();
132         }
133         else if (explosionStatus == animation::ANIMATION_ERROR)
134         {
135             renderOK = false;
136         }
137     }
138
139     return (renderOK);
140 }

```

## Note

These member functions are from Cannonball.cpp on the CD in the folder Source\Chapter13\Prog\_13\_01.

[Listing 13.2](#) starts with the updated constructor from the `cannonball` class. The difference between this version and the one in [chapter 9](#) is that this one initializes the new `ballHit` data member. The constructor does not need to initialize the `cannonball` class's `sprite` or `animated_sprite` members. Their constructors are automatically called whenever the `cannonball` class constructor is called. As a result, the `sprite` and `animated_sprite` members are automatically initialized whenever the program creates a `cannonball` object.

The `cannonball` class's new `LoadExplosionAnimation()` function begins on line 9 of [Listing 13.2](#). On line 13, `LoadExplosionAnimation()` uses the C++ `new` keyword to dynamically allocate an `animation` object. Like the `animated_sprite` class, the `animation` class is provided by LlamaWorks2D.

If the `animation` object is allocated successfully, the `LoadExplosionAnimation()` function sets its transparent color. Setting the transparent color for the animation sets the transparent color for all of its frames.

Next, by calling the `animation` class's `LoadImage()` function repeatedly, the `LoadExplosionAnimation()` function loads the individual frames of the animation. The `animation` class lets a program load as many frames as needed into an `animation` object. The program just keeps calling `LoadImage()` until all of the frames are loaded. The frames of the animation each have an index number. Numbering starts with 0, so the seven frames of the animation in [Listing 13.2](#) are numbered 06.

On lines 7476, the `LoadExplosionAnimation()` function tells the `animation` object that its animation should loop. Because the loop direction is set to `animation::FORWARD_THEN_REVERSE`, the `animation` object first plays frames 06 of the animation. It then plays frames 5, 4, 3, 2, 1, and 0 to complete the loop. If the animation is allowed to continue looping, the `animation` object plays frames 1, 2, 3, 4, 5, and 6 on the next loop cycle.

The `LoadExplosionAnimation()` function ends by storing the `animation` object in the `cannonball` class's `animated_sprite` object on line 82.

The next function in [Listing 13.2](#) is called `BallHit()`. It is almost the same as the version in [chapter 9](#). However, the version shown here sets the `ballHit` data

member to `true`. This is important because of the way animations work in LlamaWorks2D. When the explosion animation begins playing, it displays one of its animation frames during each screen animation frame. While the cannonball's explosion animation is playing, the program should not do anything else except play the explosion sound. That's exactly how this new version of CannonHit works. The `ballHit` member becomes `false` when the explosion animation is done. At that point, the program lets the player shoot the cannon again. The CannonShoot program calls the `DidBallHit()` function, which starts on line 102, to determine if the `cannonball` class's `ballHit` member is `true` or `false`.

## Note

As long as the `cannonball` class's `ballHit` member is `true`, the program doesn't do anything but display successive frames of animation and play an explosion sound.

The final function in [Listing 13.2](#) is the `cannonball` class's `Render()` function, which starts on line 108. This version of the `Render()` function renders the `cannonball` class's `sprite` object if the cannonball is in flight. That's shown on lines 113-116. If the `ballHit` data member is `true`, the `Render()` function renders the `cannonball` class's `animated_sprite` object instead. It does this by first getting the status of the `animated_sprite` object's animation. If the animation is currently playing or has not yet started, it sets the x and y position of the `animated_sprite`. It then renders one frame of the cannonball's explosion animation.

In the CannonHit program, the idea is to have the `animated_sprite` object play its animation once, and then let the program continue. Therefore, the `cannonball` class's `Render()` function checks to see whether the `animated_sprite` object's animation is at the end of its loop on line 128. If it is, the `Render()` function sets `ballHit` to `false`. This tells the CannonHit program to stop rendering the explosion animation and continue. The `Render()` function also calls the `animated_sprite` class's `Reset()` function to reset the animation. This ensures that the animation will start playing from the first frame (frame 0) the next time the cannonball hits something.

So that's how the `cannonball` class works. Now let's examine how the `cannon` class uses the new version of the `cannonball` class. [Listing 13.3](#) shows the `LoadImage()` and `Render()` functions from the `cannon` class, which were the only two that changed.

## Listing 13.3. The updated functions from the cannon class

```

1  bool cannon::LoadImage()
2  {
3      bool loadOK=true;
4      bitmap_region boundingRect;
5
6
7      theCannon.BitmapTransparentColor(color_rgb (0.0f,1.0f,0.0f));
8      loadOK =
9          theCannon.LoadImage(
10             "cannon.bmp",
11             image_file_base::LWIFF_WINDOWS_BMP);
12
13     if (loadOK)
14     {
15         theBall.BitmapTransparentColor(color_rgb (0.0f,1.0f,0.0f));
16         loadOK =
17             theBall.LoadImage(
18                 "cannonball.bmp",
19                 image_file_base::LWIFF_WINDOWS_BMP);
20     }
21
22     if (loadOK)
23     {
24         loadOK = theBall.LoadExplosionAnimation();
25     }
26
27     if (loadOK)
28     {
29         boundingRect.top = 0;
30         boundingRect.bottom = theBall.BitmapHeight();
31         boundingRect.left = 0;
32         boundingRect.right = theBall.BitmapWidth();
33         theBall.BoundingRectangle(boundingRect);
34     }
35
36     return (loadOK);
37 }
38
39
40 bool cannon::Render()
41 {
42     bool renderOK = theCannon.Render();
43
44     if (renderOK)
45     {
46         if ((theBall.DidBallHit()) || (theBall. BallInFlight()))
47         {
48             renderOK = theBall.Render();
49         }
50     }
51
52     return (renderOK);
53 }

```

The `cannon` class's `LoadImage()` function starts on line 1 of [Listing 13.3](#). It is extremely similar to the version presented back in [chapter 9](#). The difference is



on lines 2225, where it calls the `cannonball` class's `LoadExplosionAnimation()`. This enables all of the frames of the explosion animation to be loaded into the `cannonball` object.

## Note

These member functions are from `Cannon.cpp` on the CD in the folder `Source\Chapter13\Prog_13_01`.

The `Render()` function for the `cannon` class renders the cannon as in the version from [chapter 9](#). However, unlike the previous version, this version renders the `cannonball` object both if it is in flight and if its explosion animation is playing.

As you can see, the impact of using animated sprites is small on the `cannon` class. The `cannonshoot` class requires no changes to use animated sprites.

In general, you should use animated sprites in your games whenever you can. They add a greater feeling of liveliness and energy to your games. As a result, players typically feel that the game is more engaging than games that do not use animated sprites.

# High-Speed Input

As you played the Ping game introduced in earlier chapters, you may have noticed that the keyboard input was somewhat slow. In fact, it was possible for one player to keep the other player from moving her paddle by simply holding down a key and not releasing it. Problems like these are caused by the way in which Windows gets input from the keyboard. On its own, Windows is not made for high-speed input.

Before I launch into a fascinating (I hope) discussion of how to do high-speed input, it's probably best to define exactly what I mean by the term "high-speed input." In games, high-speed input is generally defined as the ability to get the state of an input device during each frame of animation. In other words, your game can do high-speed input if it can look at the keyboard during each animation frame and see which keys are pressed. That's not how Windows works. Instead, Windows sends your program messages that the program can respond to. When you're using LlamaWorks2D, you use a message map to call functions that respond to messages.

## Factoid

High-speed input is also called [\*real-time input\*](#).

Windows messages do not arrive at the program immediately after the player presses a key on the keyboard. They can take a while to get there. When you're doing something slow like typing in your name, it doesn't matter. The program will still be fast enough to get all of the letters you type and display them on the screen. LlamaWorks2D provides message maps so that you can handle slow tasks easily.

Using messages and message maps is all well and good when the player is doing something slow. However, it's not fast enough when he's trying to shoot a SnarfBeast that's running at his character with all four of its jaws open wide. In that situation, waiting for Windows messages to arrive at your message map will result in the SnarfBeast getting a nice meal with the player's character as the main course.

LlamaWorks2D provides a technique that enables you to get around having to

use Windows messages. That technique is called immediate mode input.

## Introducing Immediate Mode Input

Immediate mode input is something that I invented for LlamaWorks2D. Other game engines use similar techniques for getting high-speed input. In LlamaWorks2D, immediate mode input means that a program can examine the keyboard during each frame and determine immediately which keys are pressed.

Recall that LlamaWorks2D creates an application object that represents the program itself. The application object provides services that have to do with input, output, and the overall state of the program.

The LlamaWorks2D application class, which is called `lw2d_app`, contains a member function called `IsKeyPressed()`. The `IsKeyPressed()` function immediately checks the keyboard to determine whether the specified key is currently pressed down. If it is, `IsKeyPressed()` returns `true`. If it isn't, `IsKeyPressed()` returns `false`. You specify which key you're interested in by passing the key code to the `IsKeyPressed()` function. The key codes are defined in a list in the file `LW2DInputDevice.h`. The constants in the list all start with the letters `KC`, which stands for key code.

So, for instance, if you want to determine whether the right Shift key is pressed, you pass the value `KC_RIGHT_SHIFT` as the parameter to `IsKeyPressed()`. Or if you just want to know if any Shift key is pressed (right or left), you can pass the value `KC_SHIFT`. Likewise, if your game uses the spacebar to fire a weapon (lots of games do), you can pass the value `KC_SPACEBAR` to `IsKeyPressed()`.

That's all there is to high-speed input when you're using LlamaWorks2D. You just call one function to determine whether any particular key on the keyboard is pressed. So now, let's see immediate mode input in action by revising the Ping program to use it.

## Ping and Immediate Mode Input

Making a program use immediate mode input is very straightforward. All you need to do is call the `IsKeyPressed()` function from your `UpdateFrame()` function, or from a function that is called from `UpdateFrame()`. To add immediate mode input to Ping, I'll create a function in the `ping` class called `GetKeyboardInput()`. The `GetKeyboardInput()` function is called from the `ping` class's `UpdateFrame()` function. `GetKeyboardInput()` checks to see if particular keys are pressed. If they are, it responds by updating the

movement of the paddle. If they aren't, it just returns to `UpdateFrame()`. [Listing 13.4](#) gives the code for `GetKeyboardInput()` and the new version of the `UpdateFrame()` function.

## Listing 13.4. Modifications to make Ping do immediate mode input

```
1  bool ping::UpdateFrame()
2  {
3      bool updateOK = true;
4
5      if (!GameOver())
6      {
7          GetKeyboardInput();
8          // The rest of this function is unchanged from
9          // Chapter 8. It has been omitted to save space.
10     }
11
12     return (updateOK);
13 }
14
15
16
17 bool ping::OnKeyDown(
18     keyboard_input_message &theMessage)
19 {
20     switch (theMessage.keyCode)
21     {
22         case KC_ESCAPE:
23         case KC_Q:
24             GameOver(true);
25             break;
26     }
27     return (false);
28 }
29
30
31 void ping::GetKeyboardInput()
32 {
33     vector paddleDirection;
34
35     if (theApp.IsKeyPressed(KC_UP_ARROW))
36     {
37         paddleDirection.Y(15);
38         rightPaddle.Movement(paddleDirection);
39     }
40     else if (theApp.IsKeyPressed(KC_DOWN_ARROW))
41     {
42         paddleDirection.Y(-15);
43         rightPaddle.Movement(paddleDirection);
44     }
45
46     if (theApp.IsKeyPressed(KC_A))
47     {
48         paddleDirection.Y(15);
```

```

49     leftPaddle.Movement(paddleDirection);
50     }
51     else if (theApp.IsKeyPressed(KC_Z))
52     {
53         paddleDirection.Y(15);
54         leftPaddle.Movement(paddleDirection);
55     }
56 }

```

As you examine [Listing 13.4](#), you'll notice that most of the `UpdateFrame()` function is not there. I left it out to save space. You can see it by looking on the CD in the file `Ping.cpp` in the folder `Source\Chapter13\Prog_13_02`. The stuff that I left out is everything that hasn't changed since [chapter 8](#). The only thing that's really new in the `UpdateFrame()` function is the call to `GetKeyboardInput()`, which appears on line 7 of [Listing 13.4](#).

The code for the `GetKeyboardInput()` function starts on line 31. After declaring a variable on line 33, it calls the `IsKeyPressed()` function on line 35. Notice that the call to the `IsKeyPressed()` function is inside the parentheses of an `if` statement. The `if` statement tests the return value of the `IsKeyPressed()` function. If `IsKeyPressed()` returns `TRue`, it means the up arrow key is pressed. Therefore, the `GetKeyboardInput()` function executes the statements on lines 37-38. If the up arrow is not being pressed, the `IsKeyPressed()` function returns `false`. In that case, `GetKeyboardInput()` calls `IsKeyPressed()` again on line 40 to see if the down arrow is currently pressed. If so, it executes the statements on lines 42-43.

## Tip

Use immediate mode input for player input that requires a fast response, such as fighting, shooting, dodging danger, or bravely running away. Use the LlamaWorks2D `ON_WMKEYDOWN()` macro in a message map to respond to program commands such as quit or save game. Use the `ON_WMCHAR()` macro in a message map when the player is typing in a string.

The `GetKeyboardInput()` function follows essentially the same pattern for the A and Z keys. It calls `IsKeyPressed()` on line 46 to see if the A key is pressed and executes the statements on lines 48-49 if it is. If not, it tests to see if the player is pressing the Z key. If Z is pressed, `GetKeyboardInput()` executes the statements on lines 53-54.

A final thing to note about [Listing 13.4](#) is that it also contains the code for the `ping` class's `OnKeyDown()` function. You may recall from previous chapters that this function responds to the `WM_KEYDOWN` message that Windows sends to programs. The `WM_KEYDOWN` message comes to LlamaWorks2D programs through their message maps. Using immediate mode input in your game does not prevent you from also using message maps. The `OnKeyDown()` function in [Listing 13.4](#) tells the game that it's done when the player presses the Escape or Q keys. Because this type of input does not require an immediate response, it is just the right kind of input for message maps.

# Summary

So there you have it the last two important programming techniques you need to know to write real, professional games. This chapter showed how to create animated sprites to make your game more lifelike and engaging. It also demonstrated a simple technique for getting high-speed player input. Now you're ready to write a real game. And that's exactly what you're going to start doing in [Part 5](#), "The Big Payoff." You'll be putting your skills to the test. So don't hesitate to refer back to these skills chapters if necessary.

# Part 5: The Big Payoff

[Chapter 14. No Slime Allowed: Invasion of the Slugwroths](#)

[Chapter 15. Captain Chloride Gets Going](#)

[Chapter 16. The World of Captain Chloride](#)

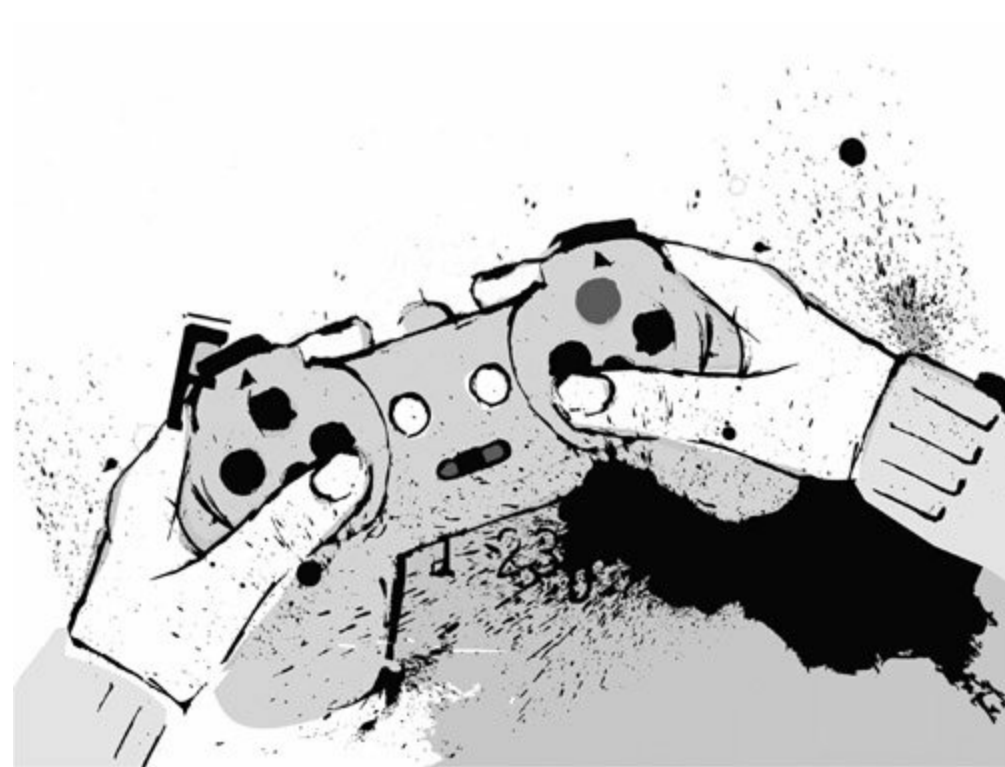
[Chapter 17. Captain Chloride Encounters Solid Objects](#)

[Chapter 18. That's a Wrap!](#)



# Chapter 14. No Slime Allowed: Invasion of the Slugwroths

Making a professional-quality game isn't as easy as it used to be. People generally aren't satisfied with simple games like Pacman or Space Invaders any more. It takes a lot more creative thinking to come up with something that hasn't already been done. In this chapter, I'll explain the process of designing a game. Then I'll walk you through it by designing the basics of a game I made up called Invasion of the Slugwroths.



# What It Takes to Make a Real Game

To make a game that looks professional and has a chance of selling well, you must combine the essential elements of a successful game in unique and creative ways.

First, your game must have decent graphics. That does not mean it requires photorealistic 3D images with all sorts of special effects. In fact, such elaborate graphics can actually detract from the game. For example, if your game is focused around solving logic problems or puzzles, 2D graphics will be fine. However, they need to be done well.

An example of innovative 2D graphics is the game Yoshi's Story, which was produced for the Nintendo 64. Although this is a 2D game, the characters are all created in 3D and rendered into 2D. The backgrounds and scenery display a wide variety of unique looks. On the first level, for instance, the sky, clouds, and other scenery look as if they're sewn from denim. It looks surprisingly good and is completely original.

When you do the graphics for your game, you need to make it visually interesting. The denim look to parts of Yoshi's Story gives the scenery greater texture. I have found in my own artwork that people consider texture more visually engaging than color or even exact shape. The artwork in older games used lots of solid colors; it looks primitive by today's standards. Adding textures to graphics makes them look much more engaging and professional.

The second essential game element you must use is sound. The most important thing to keep in mind is that players expect the sonic experience of a game to be as sophisticated as the graphics. The dinkydink sound of the music in games of yesteryear is no longer sufficient.

As I mentioned in [chapter 8](#), "Sound Effects and Music," you can get decent-sounding music out of band-in-a-box style programs. Nevertheless, you must select your music carefully. Music must be appropriate to the action. A slow, mellow jazz tune should not be used for a level in a high-speed arcade game. Likewise, you would not expect high-octane chase music in a game like SimCity.

A game's music must also be appropriate to the audience. A game for preschoolers is not the place for your favorite industrial death grunge tune. By the same token, a Bomberman-style arcade game is probably not the best place for your favorite Mozart sonata.

The third essential element of a great game is the gameplay. For fast-paced

games (I call them "twitch games" because they require fast eye-hand coordination), the player must feel that the game responds immediately to input. There is nothing more frustrating than hitting the button to fire at an enemy and experiencing a delay.

Gameplay also includes the difficulty of the game. Can the average person play the game with minimal training? Or does the player have to be really experienced with the kind of game you're making? Players become upset if they die fast when they start the game; they must feel that they're making some progress at first.

The final essential element of a hit game is superb design. This does not require the game to be complex. For instance, a Rubik's Cube is a superbly designed (noncomputer) game. However, it's also very simple. Likewise, the game Tetris is an outstanding piece of game design. It's simple in concept and takes almost no time to learn. However, people have found it addictive for nearly 20 years.

An example of a more complex game that has outstanding design is Myst. This is an older game, but one that I think every game developer should play through. Myst has a complex story that is very freeform. You can play the game in any order you want. You can wander around, examine everything you find, and learn the object of the game as you go. Myst uses a very nonlinear style to tell a story.

Other examples of excellent game design are Microsoft's Halo, Oblivion's The Elder Scrolls, Blizzard's Warcraft, and Electronic Arts' The Sims. They all provide a rich game experience. None of them require the player to be a hardcore gamer to be successful. Each has its own innovative design elements that map well to its style. For example, Halo puts the player in a world of constantly shifting alliances. The Elder Scrolls provides one of the most freeform universes you'll ever see. Warcraft combines fast-paced fighting with strategic planning and allocation of resources. The Sims gives the player surprisingly engaging social simulations.

The point is that when you design a game, your game design must be innovative and distinct. You can't just crank out a slightly better version of someone else's game.

Saying that you've got to be innovative is all well and good, but how do you make that happen? To find out, the next section of this chapter examines the design process in more detail.

## **Note**

I've mentioned this before: a big clue is how you describe your game. If you say, "It's like Quake only better," don't bother making it. You need to be more innovative than that. If you can describe your game as, "It's like a cross between The Sims and Atomic Bomberman," you might have something worthwhile. Those are such radically different games that combining their gameplay styles might just give you something unique.

# Essential Game Design

Designing games requires understanding the elements of good game design. You also have to know the process most professionals use to design games. Creating a good game begins with using the elements of good game design within the game design process.

## Elements of Good Game Design

To design good games, there are certain techniques that you need to use:

- Training the player during gameplay
- Keeping it fresh with new challenges
- Keeping challenges appropriate
- Incorporating powerups for player relief
- Deciding when to use a boss

## Training the Player During Gameplay

Whatever else you do in your game, don't just throw a new player into the middle of an intense battle or challenge without some initial instructions. In general, players will not read the game's instructions. So typing up a nice help file about how to play the game is usually not sufficient. You typically have to train the player in the first level or so of the game. There are a number of ways to accomplish this.

First, you can have a level that is for nothing but training. The setting of the level is actually a training school where you have to pass all available courses before moving into the game itself. Several of the firstperson shooters do this when you begin playing them. It's a bit tedious, but if there are a lot of weapons, movement techniques, and so forth that the player needs to know before they get into the game, players will generally put up with a training level. Often, the game features some sort of narrator's voice giving instructions.

Another technique is to have a level that is simple but that is actually part of the overall plot or mission of the game. In this simple level, the player can become familiar with moving the character or using the weapons, or begin learning the rules of the game. There may be a narrator for this type of level as well.

Instead of having levels designed for training, some games use popup dialog boxes with instructions. When the player encounters a certain object, instructions pop up explaining how to use it. Instructions might cover such topics as running, jumping, using weapons, or driving vehicles.

However you decide to approach this task, be sure to provide players with an interactive method of getting training.

## **Warning**

If you use popup instructions, you must either disable the instructions after the player has seen them or provide the player with a manual way to bypass them. Players do not want to encounter a set of instructions about how to use a pistol every time they pick up a pistol.

## **Keeping it Fresh with New Challenges**

Monotony is the death of any game. It is important that the game gradually gets harder as the player moves to higher levels. The more the player becomes proficient at the game, the harder the game should get. Each level should present slightly different and slightly more unusual challenges. Also, the training that you started at the beginning of the game should continue as the challenges get harder.

## **Keeping Challenges Appropriate**

The challenges that you present a player with must always be appropriate to the player's skill. As a result, it makes sense that your game should be easier at first and grow more difficult later. However, that's not always true.

For example, if you give the player an especially challenging level to overcome, the next level should be slightly easier. That provides the player with some relief and lowers his or her frustration level. Also, it provides you with an opportunity to give the player new abilities, weapons, or other tools within the game. Players can spend time in a slightly easier level learning the new tools or techniques. They can then move onto more difficult levels after becoming proficient with the new stuff.

To keep challenges appropriate, have a specific goal in mind for each level. The goal can be nothing more than you want the player to kill lots of monsters. Or it might be that you want players to learn a new strategy that they'll need later. Either way, focus the challenges on the goal.

## **Tip**

You play your game a lot while you're developing it. As you might expect, you get very good at it. Your own skill at playing the game makes it difficult for you to evaluate how challenging each level is for the player. To more accurately know how challenging each level is, have a person play your game during development who has never played it before. In the game industry, this is called focus testing. Ask them what they think. Write down what they say. Give them some pizza!

## **Incorporating Power-Ups for Player Relief**

Players must have some sort of relief mechanism. These usually take the form of powerups. Even the oldest and simplest games used this technique. For instance, one of the first games after Pong was Space Invaders. In this game, the player was expected to shoot rows of invading aliens. The player started the game with three lives. After attaining a certain score, the player was given another life. Every now and then, an alien spacecraft would drift across the top of the screen. If the player shot it, the game awarded extra points. This helped the player get a new life faster.

Although you have to be careful not to provide too many powerups, it is important to offer the player some method of relief. This helps keep people's frustration levels down and enjoyment up. Also, getting a powerup as a reward

helps bring a sense of accomplishment as the game progresses.

## Deciding When to Use a Boss

Many games have a big monster, or some other kind of major challenge, at the end of each level. This monster or challenge is usually referred to as the *boss*. You must decide whether your levels will have bosses or not. They are not always appropriate.

If you use a boss, don't do it just to have a boss. Put in the boss for a purpose. In the best games, fighting the boss trains the player for the game's final challenge. Putting in a boss is, in a way, the opposite of putting in powerups. You add bosses to levels to increase the challenge and difficulty. Used appropriately, this helps give the player a sense of accomplishment.

## The Game Design Process

Now you're ready to learn the game design process. Keep in mind that the process you see here is a generic one. It differs for different types of games. For games that are strongly driven by a storyline, the game design process can be very much like making a movie. More freeform games use a process similar to the one shown here.

The basic steps for designing a game are as follows:

1. Start with a brainstorming session.
2. Create the goal for the game.
3. Select the emotional experience for the game.
4. Choose the right look.
5. Set the hook.
6. Create a design document.
7. Let it simmer a while.



8. Revise the design document.
9. Create a prototype and play it.
10. Iterate as necessary.

The first nine steps need to be completed before you start writing any C++ code. It's important to finish each step. If you skip any, you're likely to undermine the quality of your game.

## **Start with a Brainstorming Session**

The goal of this step is to come up with a unique and unusual idea for a game. It's best to have multiple brainstorming sessions over a period of time.

For each game idea you come up with, write a brief summary of what the game is about. The summary should not be longer than about 35 sentences.

When you have a list of good ideas, go through the list and rate them on a scale of 0 to 100. Assign the scores based on originality, potential audience, how fun you think it is, and how hard it is to implement. Give up to 25 points for each category.

For instance, you might think your idea is extremely unique and assign it a score of 25 points for originality. However, if only males between the ages of 14 and 24 are going to like your game, its potential audience is smaller than it might otherwise be. Therefore, it should get a low score for that category.

When you assign the score for how fun your idea is, be careful. You might think your game is fun because you're in the target audience. But how fun is it for everyone else? You can assign your game a good fun rating if it hits its target audience well. However, if there's a way to tweak the idea so that it appeals to a broader audience, you can give it both a higher audience score and higher fun score.

The last category you should assign points for is difficulty of implementation. Elaborate games may be very cool, but if your idea takes you five or six years to program up, you may want to think twice before diving into it. Beginning game programmers should start with games that are easier to implement. There are a lot of good games that fit into this category. For example, many puzzle and logic games are not difficult to write. This is especially true if you

use LlamaWorks2D.

## **Tip**

It's better to stay away from elaborate 3D firstperson shooters until you feel more proficient as a programmer.

## **Create the Goal for the Game**

Every game has a goal. For most firstperson shooters, the goal is to maximize carnage and violence. The goal of most puzzle and logic games is to present brainteasers. Other goals for games might be telling a story, building civilizations, forming relationships between game characters, or saving kidnapped dragons from bloodthirsty princesses. The goal can be anything you want. However, everything in the game must be focused on the goal. All aspects of the game should move the player toward it.

Given that, it is sometimes desirable to put things into a game that are deliberately unrelated to the game's goal. These are usually bonus diversions and surprises that you include just for fun. In the game industry, these are called Easter eggs.

Suppose, for example, that you're writing a roleplaying game called Arnold the Anteater Saves the Universe. It's perfectly possible for Arnold to find a game on the computer aboard the spaceship of the Vampire Antoids. That game might be a parody of Space Invaders in which a Vampire Antoid protects the planet from rows of descending humans. That's a fun little Easter egg that players might enjoy.

## **Select the Emotional Experience for the Game**

Closely related to choosing the goal for your game is selecting the experience the player has. While the goal defines the point of the game, the experience defines the feelings and emotions people have while playing the game.

If the goal for a firstperson shooter is to maximize carnage, then the experience should be fastpaced. The player should experience a series of

intense adrenaline rushes throughout the entire game.

On the other hand, if the goal for a game is to develop relationships between game characters, then the experience should be relaxed, lighthearted, and romantic.

Whatever your game's goal is, you should carefully think about how the game's emotional experience helps you attain that goal. If the emotional experience does not match the game's goal, the player won't like the game. It's really that simple.

## **Choose the Right Look**

Before you start writing a game, you need to decide how you want the game to look. Because the experience of the game is primarily a visual experience, the look of your game is critical to its success.

Don't think, however, that your game must use photorealistic graphics to be enjoyable. To understand why, take cartoons as an example. Most Disney cartoons use excellent graphics. Likewise, the old Looney Toons cartoons were elaborately drawn for their time. However, people also enjoy highly stylized looks such as Rugrats cartoons. Heck, people even seem to like the simple, ugly graphics of Sponge Bob Squarepants. I can't imagine why.

The point here is that there are a wide range of visual styles you can choose from. Your game can use elaborate or simple graphics. Either way, the look of the game should be focused on the goal and the emotional experience. That look should also include a color scheme.

For instance, firstperson shooters generally have settings that are dark, foreboding, and brooding. In some places, they use unearthly neon greens or raging oranges and reds. You generally would not see pinks or pastels in a firstperson shooter.

On the other hand, games oriented toward relationships might make good use of pastels to invoke feelings of softness and gentleness. They might also use bright primary colors to incorporate a feeling of liveliness and fun. Relationship games would probably stay away from cool colors and use warmer colors.

Never underestimate the impact of the overall look of your game. It has a huge effect on the emotions your game evokes in the player.

# Set the Hook

Writing a game is like fishing. You want to draw players into a game with a pretty lure. Usually, that's something like cool graphics, fast action, stimulating puzzles, or complex strategies. However, just like getting a fish on the line, you need to set the hook so that players stay addicted. The question is, how do you get them hooked?

A lot of different hooks have been used in the game industry. Here are a few of the most common:

- **High score.** This hook was common in early arcade games. It's not used as much any more because players expect more from a game experience than just a score. However, some types of games use this well. Logic games and sports games are good examples.
- **Finishing the game.** The hook of finishing the game was commonly used in early arcade games. My experience is that it's not as effective as many other hooks. It's usually not the end of the journey that keeps players going; it's the experience along the way.
- **Mastery.** This is a huge hook for sports games and vehicle racing games. Controlling virtual athletes or vehicles in a competition requires a steady, practiced hand. This hook is also used for fighting games such as Mortal Kombat or Virtua Fighter. To be the best, players must master tricky power moves that devastate opponents.
- **Exploration.** Exploring new and interesting worlds is by far one of the most common and powerful hooks in games. It doesn't necessarily take fancy 3D graphics to utilize the exploration hook. The old Super Mario Brothers for the Nintendo Entertainment System (NES) is an excellent example of a 2D sidescrolling game that made extremely effective use of exploration as a way to get players hooked on a game.

Now that you know what the most common hooks are, how and when do you use them?

The high score and mastery hooks are for people who want bragging rights. Put them into fighting, racing, sports, and other competition-oriented games.

Exploration works in almost any type of game. One of the best ways to

encourage exploration is to hide some of the game's levels. I've heard it said by top game designers that as many as 40% of a game's levels should be hidden. As the player moves through a level, he or she finds ways to unlock or open hidden areas. The first few should be extremely easy to find so that virtually everyone will see and play them.

The levels that you hide in your game can be almost anything. For example, they can be arcadestyle minigames (games within games). They can be additional tennis courts for tennis games or more golf courses for golfing games. A hidden level might be a dungeon, a castle in the sky, or a kingdom beneath the sea.

Alternately, you can encourage exploration by hiding powerups, weapons, or vehicles. A racing game can have additional cars that are awarded when the player achieves a specific number of firstplace trophies. A boxing game can have a special trainer that visits only if the player gets a TKO within the first three rounds in a particular match. The special trainer teaches the player some new moves that can be used in upcoming fights.

You can incorporate multiple hooks into a game. It actually makes your game more addictive if you do. For instance, a racing game can use the exploration, mastery, and high score hooks.

Whichever hooks you decide are important to your game, take the time to put careful thought into using them. If you wait to make these decisions until after you've started programming, there's a good chance that you'll have to go back and rewrite large portions of your game.

## **Tip**

As the player moves through the game, the hidden levels should get harder to find.

## **Create a Design Document**

The single most important step in designing a game is to create a design document. Even if you are writing the game all by yourself, you must take time to do this if you want an effective game enjoyed by a wide audience.

Your design document should contain all of the decisions you've made about your game. It should describe in one paragraph the basic idea for the game. It should also provide a longer, detailed description of the idea. The short description is what you keep in mind as you write the game. It's your destination and tells you when you've finished writing the game. The detailed description forces you to think through the idea thoroughly. This can save you from a lot of wasted time if your idea stinks. And all of us come up with game ideas that stink. In fact, probably 9 out of 10 of the ideas a game designer comes up with should never be produced. Writing a detailed description of the game helps weed out the bad ideas from the good ones.

The design document also needs statements describing the game's goal, emotional experience, and look. It should state the hooks you're going to use and how you're going to use them. Lastly, it should contain the design of every level in the game.

## **Let it Simmer a While**

After you've written the design document, stop and go do something else for a while. Don't skip this step. Let the design document sit in a drawer or file cabinet for at least a month.

There are often problems in the game's design that you don't see while you're immersed in the process of designing it. By letting it sit for a month or so, you'll come back to it with a fresh pair of eyes. It will be much easier to spot design problems.

## **Revise the Design Document**

Okay, you've got your finished design document. Now start over and revise it. Eliminate everything that doesn't add to the emotional experience you're going for. Get rid of anything that detracts from the game's goal.

Good; now revise your design document again. Add any hooks that you think will improve the effectiveness of your game. Add secret levels, hidden bonuses, Easter eggs, and other goodies for the player to find.

Now you're ready to start programming.

## **Create a Prototype and Play It**

You can start programming, but don't write the entire game. Get one level up and running. This is your prototype. It demonstrates what your game is like and what it's about. It also shows you what the game really looks like. All too often, those cool stylized graphics we come up with look completely lame when we get them into a game. Your prototype demonstrates that the look of the game is actually as attentiongrabbing as you thought it would be. If it isn't, you must redesign the look of the game.

The prototype also enables you to play the game. That tells you whether the game design really works. It's often the case that the level you designed is boring or too hard to play. That isn't an insult. That's a normal experience that every game developer has. The only way to deal with it is to understand that this is going to occur and plan for it. Creating a prototype and playing a level (or a few levels) goes a long way toward telling you how good your game is and where its faults lie.

## **Iterate as Necessary**

After you are satisfied with your prototype, continue writing the game. You'll add levels, program new powerups and weapons, and so on. As you do, you'll need to play your game over and over. Let others play it as well so you can watch. Anything that they have trouble with is something that might need to be redesigned.

In fact, you'll be redesigning constantly as you go along. Good game designers repeat the game design process as often as possible through the development of the game. When you are done, you'll have a game that is finely honed into a compelling experience.

# Designing Invasion of the Slugwroths

To put the ideas from this chapter into practice, we'll walk through the process of designing a game. Because this game is a teaching tool, it will be much simpler than a real game. Nevertheless, it will have all of the basic elements of a real game. Also, the game will be a full enough implementation to demonstrate the design process.

## The Brainstorm

The first step in designing a game is to come up with the idea for it. The idea must fit within the limitations of this book. Therefore, it must meet the following requirements:

1. The game must first and foremost be a teaching tool.  
The game must be simple enough so that someone who has never written a
2. game can implement it.
3. The game must be real enough to be worthwhile for you, the reader.
4. The graphics must be in 2D so as to make use of LlamaWorks2D.

To come up with the idea for this game, I spent time thinking about the 2D games I played in the 1980s and '90s. I was able to dig out some old disks containing these games, and I installed and played them. In addition, I made a list of the old games I liked best and why. Eventually, I decided on a sidescrolling platform game. Sidescrollers are games in which the character moves continuously to the left or right as the level progresses. Platform games are games in which the character spends the majority of the level hopping, swinging, or jumping from one platform on the screen to another. Most old sidescrollers were also platform games to one degree or another.

After a lot of ruminating, I finally settled on a game called Invasion of the Slugwroths. In this game, the world is invaded by a nasty, slimy race of sluglike creatures that are here to steal our entire supply of radishes (gasp!). Our erstwhile hero is Captain Chloride, defender of the defenseless, helper of the helpless, and generally regarded as the gum on the shoe of evil.

The Slugwroths, which is what the invaders are called, have long,



trumpetshaped snoots that they can shoot boogers from (this game was designed with my 10yearold son in mind). The player starts with three Captain Chlorides. When Captain Chloride gets hit by boogers, he gets stuck in goo and looks out at the player with a sad expression. He doesn't exactly die, but the player loses one Captain Chloride. Also, the Slugwroths leave a trail of slime behind them when they move. If Captain Chloride touches it, he looks sad and the player loses a Captain Chloride.

The Slugwroths have set up many bases around the world. Captain Chloride must move from base to base, cleaning out the Slugwroths as he goes. He can increase his number of lives by eating the perfect food (pizza) whenever he finds it. Inexplicably, there are many slices of pizza scattered around the world. When he eats enough pizza, Captain Chloride gets another life.

Also scattered around the world are large pellets of salt. No one knows how they got there, but it's lucky they didfor, you see, Captain Chloride carries the most devastating weapon the world has ever known. It's the one weapon feared and dreaded by all Slugwroths. Yes, I'm speaking of the worldfamous salt shooter. When a Slugwroth gets hit by Captain Chloride's salt shooter, he shrivels up and gets a sad look on his face. When he is scrolled off the screen because Captain Chloride has moved away, the Slugwroth disappears forever.

That's the basic idea for the game. We'll write as much of it as space in this book allows.

## **The Goal**

The goal of Invasion of the Slugwroths is to teach. If I was implementing this as a real game, the goal would be for the player to work through all of the Slugwroth bases and clean them out. The player would have to learn the techniques and acquire the tools needed to eliminate Slimeordeous, the leader of the Slugwroth forces on Earth.

## **The Experience**

If this were a real commercial game, the experience of Invasion of the Slugwroths would be fastpaced and fun. There would be a lot of corny, campy humor and fun surprises built into the game.

Because this is designed to be a teaching tool, the experience you'll hopefully take away from writing a portion of Invasion of the Slugwroths is that it is

really easy to get started in game programming.

## **The Look**

The look and style of Invasion of the Slugwroths will be similar to the arcade games of the late '80s and early '90s. It will not use 3D rendered into 2D as many games did in the mid and late '90s. It will be strictly 2D and cartoony.

## **The Hooks**

If this were to be a commercial product, Invasion of the Slugwroths would use several hooks. The most important would be exploration. The player must explore the world to find hidden bases, powerups, and weapons.

In addition, Invasion of the Slugwroths would use the mastery hook. The player must get proficient at running, jumping, dodging, and strategic thinking in order to beat the levels. Because some players are concerned with scores, this game would also use the high score hook.

Another hook that can make a game distinctive is its humor. This tends to be a less compelling hook. But in the case of Invasion of the Slugwroths, humor can add a lot to the experience of the game. If I wanted to sell this game, I would put in as much goofy (or even stale) humor as possible.

## **The Design Document**

For a real game, I would write an extensive design document. It would contain the basic game description presented previously, the statement of the goal, the description of the experience, some sample drawings to indicate the look, and the statement of the hooks.

In addition, the design document would have a much more detailed description of the game. It would have drawings of each level with descriptions of where the traps were and how they all work, where the powerups were and how they work, and the locations of hidden levels with a statement of how each one is accessed or activated. If this game were going to have bosses, the design document would describe them, state their locations, and tell how they are defeated.

Although this sounds like a lot of information for the design document, it is actually rather sparse compared to the design documents in professional game companies. Because game design is so involved, it is often split into a different job than game programming. However, it's something of a truism that the more game programmers know about game design, the better they can do their jobs.

# Summary

In this chapter you received an introduction to professional game design. Keep in mind that it is only an introduction. Many books have been written on game design. It is a large and complex subject. However, all game programmers should at least be familiar with the basics presented in this chapter. Specifically, you should know that your game must be innovative, use attractive or stylized graphics, and have the best sound you can give it.

To keep players engaged, you must train them as they play. You must regularly present players with fresh and appropriate challenges. At intervals, give the player powerups or some other form of relief.

In addition, it's important to use the game design process presented in this chapter. Don't take shortcuts; they won't help you. The best games are always carefully designed and crafted to be a compelling experience.

Okay, we have a design for Invasion of the Slugwroths. Beginning in the next chapter, we'll use the programming techniques presented in previous chapters to implement it.

# Chapter 15. Captain Chloride Gets Going

In this chapter, you'll see how to start implementing the sample game Invasion of the Slugwroths, which was described in [chapter 14](#). I've learned by experience that one of the best ways to develop a computer program is to write a little of it at a time. If I can get a bit of it debugged and running, it's easy to add some more and debug that. I'll use this method to build the sample game through the remainder of this book.

This chapter illustrates the process of getting Captain Chloride on the screen and walking around.



# Introducing Captain Chloride

Captain Chloride, erstwhile defender of the planet Earth, requires the use of animated sprites to seem lifelike. His most common movement is walking, so that's the first thing I'll implement.

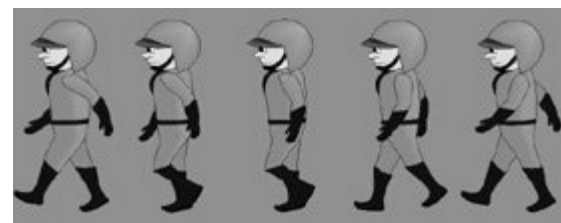
Because Invasion of the Slugwroths is a side-scrolling game, Captain Chloride must be able to walk either left or right. Each of these movements requires its own animation. Also, the game must display an image when he is not moving. I like to start simply, so I'll demonstrate walking Captain Chloride using an animated sprite for all of Captain Chloride's walking movements. Although it doesn't look all that professional, I'll have Captain Chloride just freeze in place when he stops walking.

## Note

When the main character is not moving, many games do more than just display an image of it frozen in a walking position. They typically show something like an animation of the character looking out at the player and impatiently tapping its foot. It adds both life and humor to the game.

Making Captain Chloride walk in a particular direction requires several animation frames. [Figure 15.1](#) displays the frames for walking left.

**Figure 15.1. Each frame shows Captain Chloride in a slightly different position.**

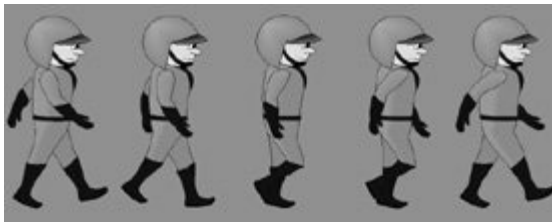


## Note

The frames for walking left are mirror images of those used for walking right. Graphics programs let you flip bitmaps. I actually only drew the frames for walking right. I then flipped each frame so that the Captain faced left and saved the flipped frames under different file names. This is a technique you should use whenever possible.

To display Captain Chloride walking right, the game needs a set of animations that are mirror images of the ones shown in [Figure 15.1](#). These are given in [Figure 15.2](#).

**Figure 15.2. This series of frames is a mirror image of those used to walk left.**



Both sets of frames are included with the sample programs on the CD. You can find them in the folder Source\Chapter15\Prog\_15\_01. I drew these animation frames using CorelDraw, but they could have been done just as easily with GIMP, which I've provided for free on the CD. When I drew each frame, I saved it as a Windows BMP file.

## Designing the Chloride Class

It's often the case in your games that the main character will be implemented in a class especially designed for that character. Therefore, that's the way I'll write Invasion of the Slugwroths. [Listing 15.1](#) shows the `chloride` class.

**Listing 15.1. The class that implements Captain Chloride**

```

1 class chloride
2 {
3 // Public types.
4 public:
5     enum chloride_state
6     {
7         STAND,
8         WALK_LEFT,
9         WALK_RIGHT
10    };
11
12 // Public member functions
13 public:
14     chloride();
15
16     bool LoadResources();
17
18     bool Update();
19     bool Render();
20
21     void X(
22         float upLeftX);
23     float X();
24
25     void Y(
26         float upLeftY);
27     float Y();
28
29     void GetKeyboardInput();
30
31     void Movement(
32         vectorf new_velocity);
33
34     int BitmapHeight();
35     int BitmapWidth();
36
37 private:
38     animated_sprite leftWalk;
39     animated_sprite rightWalk;
40     animated_sprite *standingSprite;
41     sound leftWalkSound;
42     sound rightWalkSound;
43     vectorf velocity;
44     float x,y;
45     chloride_state state;
46     chloride_state lastWalkState;
47     int lastStepIndex;
48 };

```

The `chloride` class defines a set of constants on lines 5-10. The class uses these constants to keep track of what Captain Chloride is doing during any given frame of screen animation.

## Factoid



Many Windows programmers use function names like `LoadResources()` in their classes because sounds and images are referred to as resources in Windows.

Next, the `chlорide` class contains the prototypes for its member functions. There are a few noteworthy items here. First, the `chlорide` class has a member function called `LoadResources()`. This function, as we'll see shortly, loads all of the images and sounds that the `chlорide` class uses.

The second important item in the `chlорide` class member functions is the fact that the `chlорide` class contains its own `X()` and `Y()` functions. These versions use floating-point numbers rather than integers. Although this is not necessary for this first example program, implementing the `chlорide` class this way makes it easier to put Captain Chloride into a custom gamespace later.

Notice also that there is a member function in the `chlорide` class called `GetKeyboardInput()`. This function gets the player's input and tells Captain Chloride to move left or right accordingly. It is in this class rather than the game class because it's an easy way to control Captain Chloride. You don't have to do it this way. You could put this into your game class instead.

Lines 3840 show that I'm using three different animated sprites in this program to store Captain Chloride's animations. In the next chapter, I'll show how to put all of them into one animated sprite.

Lastly, notice that lines 4546 introduce two data members of type `chlорide_state`. These private members keep track of what the good Captain is currently doing and what he was doing in the last frame. If he was walking, the data member on line 46 stores the animation frame that was current at that time.

## Note

A blue background is the default background in LlamaWorks2D.

## Implementing the Chloride Class

**Listing 15.2** shows the implementations of several of the most important member functions. To save space, I won't show all of the member functions. However, you can find them on the CD in the folder Source\Chapter15\Prog\_15\_01. The file that contains the `chloride` class is Chloride.cpp.

## Listing 15.2. Some member functions of the `chloride` class

```
1 void chloride::GetKeyboardInput()
2 {
3     vectorf captainDirection;
4
5     if (theApp.IsKeyPressed(KC_UP_ARROW))
6     {
7         captainDirection.Y(-1);
8     }
9     else if (theApp.IsKeyPressed(KC_DOWN_ARROW))
10    {
11        captainDirection.Y( 1);
12    }
13    if (theApp.IsKeyPressed(KC_LEFT_ARROW))
14    {
15        captainDirection.X(-1);
16    }
17    else if (theApp.IsKeyPressed(KC_RIGHT_ARROW))
18    {
19        captainDirection.X( 1);
20    }
21
22    if (captainDirection.MagnitudeSquared() == 0.0)
23    {
24        state = STAND;
25    }
26    else
27    {
28        // Make the direction be a unit vector, in
29        // case he is moving diagonally, then scale
30        // it by the speed.
31        captainDirection = captainDirection.Normalize();
32        captainDirection *= 15.0;
33
34        if (captainDirection.X() < 0.0)
35        {
36            state = WALK_LEFT;
37        }
38        else if (captainDirection.X() > 0.0)
39        {
40            state = WALK_RIGHT;
41        }
42        else
43        {
44            state = lastWalkState;
45        }
46    }
47 }
```

```

48     Movement(captainDirection);
49 }
50
51
52 bool
53 chloride::Update()
54 {
55     x += velocity.X();
56     y += velocity.Y();
57
58     if (x < 0.0)
59     {
60         x = 0.0;
61     }
62     else if (x > theApp.ScreenWidth() - BitmapWidth())
63     {
64         x = theApp.ScreenWidth() - BitmapWidth();
65     }
66
67     if (y < 0.0)
68     {
69         y = 0.0;
70     }
71     else if (y > theApp.ScreenHeight() - BitmapHeight())
72     {
73         y = theApp.ScreenHeight() - BitmapHeight();
74     }
75
76     return true;
77 }
78
79
80 bool
81 chloride::Render()
82 {
83     bool renderOK = true;
84     animated_sprite *currentSprite;
85
86     switch( state)
87     {
88         case STAND:
89             // pause the animation
90             standingSprite->Animation(0)->CurrentFrame
91                 (lastStepIndex);
92             currentSprite = standingSprite;
93             break;
94
95         case WALK_LEFT:
96             lastWalkState = WALK_LEFT;
97             standingSprite = &leftWalk;
98
99             lastStepIndex =
100                 leftWalk.Animation(0)->CurrentFrame();
101             if (lastStepIndex == 3)
102             {
103                 leftWalkSound.Play();
104             }
105
106             currentSprite = &leftWalk;
107             break;
108

```

```

109     case WALK_RIGHT:
110         lastWalkState = WALK_RIGHT;
111         standingSprite = &rightWalk;
112
113         lastStepIndex =
114             rightWalk.Animation(0)->CurrentFrame();
115         if (lastStepIndex == 3)
116         {
117             rightWalkSound.Play();
118         }
119
120         currentSprite = &rightWalk;
121     break;
122
123     default:
124         ASSERT(0); //Unhandled state!
125     break;
126 }
127
128 currentSprite->X((int)x);
129 currentSprite->Y((int)y);
130 currentSprite->Render();
131
132 return (renderOK);
133 }

```

The first member function in [Listing 15.2](#) is the `GetKeyboardInput()` function. It begins by declaring a vector that is used to store the direction Captain Chloride moves. This variable is of type `vectorf` rather than `vector`. Recall from [chapter 5](#), "Function and Operator Overloading," that the difference between `vectorf` and `vector` is that `vectorf` uses floating-point numbers for its x and y components rather than integers. The variable `captainDirection` is of type `vectorf` because Captain Chloride will be in a gamespace defined with floating-point numbers in the final game.

On lines 520, the `GetKeyboardInput()` function contains statements that test the direction arrow that the player presses and sets Captain Chloride's direction accordingly. Notice in particular that, in addition to the left and right arrows, this function handles moving Captain Chloride up and down. Try running this program, which is called `Prog_15_01.exe` on the CD. You'll see that Captain Chloride moves up and down when you press the up or down arrow. If he's facing left, he looks like he's walking left, but he just moves up. We'll fix that later. For now, try pressing both the left and up arrows. When you do, you'll see the reason I put in the code on lines 512 of [Listing 15.2](#); Captain Chloride walks diagonally up and left across the screen. If you want your character to walk diagonally, this is how you do it.

Next, line 22 of the `GetKeyboardInput()` function tests to see whether Captain Chloride's velocity is zero. If it is, line 24 sets his current state to indicate that

he's standing still.

## Tip

I've pointed this out before in previous chapters but it bears repeating. Whenever you can, test the magnitude squared of a vector rather than the magnitude itself. Calculating the magnitude squared is faster for your game than calculating the magnitude.

If Captain Chloride is moving, line 31 of [Listing 15.2](#) sets his direction vector to a unit vector. Recall that unit vectors can point in any direction, but they always have a magnitude of 1. Line 32 multiplies the unit vector by 15 to make Captain Chloride move by 15.

The `GetKeyboardInput()` function sets Captain Chloride's current state on lines 3445. This enables the object that represents Captain Chloride to keep track of what he's doing. At any given time, this object must "know" whether Captain Chloride is walking, standing, and so forth.

Just before the `GetKeyboardInput()` function ends, it sets Captain Chloride's movement vector.

During each frame of screen animation, the program calls the `chloride::Update()` function. If you look in the file `Invasion.cpp` on the CD, you'll see that the game class's `Update()` function gets the keyboard input and then calls the `chloride::Update()` function.

## Note

You'll find the file `Invasion.cpp` in the folder `\Source\Chapter15\Prog_15_01`.

The `chloride::Update()` function moves Captain Chloride by setting his x and y position on lines 5556. If he hits the edge of the screen (any edge), lines 5874 adjust his x and y position to keep him on the screen.

The `chloride::Render()` function begins on line 81 of [Listing 15.2](#). It declares a pointer

to an `animated_sprite` object as a temporary variable on line 84. On lines 72-108 the `chloride::Render()` function figures out what Captain Chloride is doing and points the variable `currentSprite` at the appropriate `animated_sprite` object. For instance, if he's walking left, the `switch` statement executes the `case` statement that begins on line 95. This results in setting `currentSprite` to point at the animated sprite containing the animation of Captain Chloride walking left on line 106. When the `Render()` function actually renders Captain Chloride, it renders the animated sprite that `currentSprite` points at (see line 130).

Notice also that the case statements in the `Render()` function set the private data member `standingSprite`. This is necessary to correctly freeze Captain Chloride in place when he's standing still. Whenever Captain Chloride takes a step, the `Render()` function sets `standingSprite` so that it points either at the animated sprite of him walking left or the one of him walking right. If, during the next frame of screen animation, Captain Chloride is standing still, the `Render()` function displays Captain Chloride frozen in the last position he was in. It does this by setting the `currentSprite` variable to point at the same animation as the `standingSprite` data member on line 78. In addition, it uses the data member `lastStepIndex` to set the current animation frame. In this way, Captain Chloride freezes when the player stops walking him.

# Pulling It Together In The Game Class

This version of Invasion of the Slugwroths is Program 15.1 on the CD. Its game class, called `invasion`, is surprisingly brief. It makes everything work by initializing the program, processing input, and rendering the Captain on the screen. [Listing 15.3](#) provides the code for the `invasion` class.

## Listing 15.3. The `invasion` class

```
1  class invasion : public game
2  {
3  public:
4      bool OnAppLoad();
5      bool InitGame();
6      bool UpdateFrame();
7      bool RenderFrame();
8
9      ATTACH_MESSAGE_MAP;
10
11     bool OnKeyDown(
12         keyboard_input_message &theMessage);
13
14     void GetKeyboardInput();
15
16 private:
17     chloride theCaptain;
18 };
19
20 CREATE_GAME_OBJECT(invasion);
21
22 START_MESSAGE_MAP(invasion)
23     ON_WMKEYDOWN(OnKeyDown)
24 END_MESSAGE_MAP(invasion)
25
26 bool invasion::OnAppLoad()
27 {
28     bool initOK = true;
29
30     //
31     // Initialize the window parameters.
32     //
33     init_params lw2dInitParams;
34     lw2dInitParams.openParams.fullScreenWindow = true;
35     lw2dInitParams.winParams.screenMode. AddSupportedResolution(
36         LWSR_800X600X32);
37     lw2dInitParams.winParams.screenMode. AddSupportedResolution(
38         LWSR_800X600X24);
39     lw2dInitParams.openParams.millisecondsBetweenFrames = 50;
40
41     // This call MUST appear in this function.
42     theApp.InitApp(lw2dInitParams);
43
44     return (initOK);
45 }
```

```

46
47
48 bool invasion::InitGame()
49 {
50
51     bool initOK = true;
52
53
54     initOK = theCaptain.LoadResources();
55
56     if (initOK==true)
57     {
58         theCaptain.X(400.0);
59         theCaptain.Y(300.0);
60     }
61
62     return (initOK);
63 }
64
65
66 bool invasion::UpdateFrame()
67 {
68     GetKeyboardInput();
69     bool updateOK = theCaptain.Update();
70
71     return (updateOK);
72 }
73
74
75 bool invasion::RenderFrame()
76 {
77     bool renderOK = theCaptain.Render();
78
79     return (renderOK);
80 }
81
82
83 bool invasion::OnKeyDown(
84     keyboard_input_message &theMessage)
85 {
86     switch (theMessage.keyCode)
87     {
88         case KC_ESCAPE:
89         case KC_Q:
90             GameOver(true);
91             break;
92     }
93     return (false);
94 }
95
96
97 void invasion::GetKeyboardInput()
98 {
99     theCaptain.GetKeyboardInput();
100 }

```

The `invasion` class contains an object of type `chloride` as its only private data



member. As with all game classes in LlamaWorks2D, the `invasion` class attaches its message map, creates its game object, and then defines its message map.

## Tip

Most game programmers buy a video card that supports multiple monitors. That way they can run the game on one and the debugger on the other. I highly recommend this approach if you're serious about writing games.

When this program initializes itself, it runs in full-screen mode. This is how you generally want things when your game runs. However, when you're debugging, it's a good idea to run the program in a window. That way you can switch back and forth between the game and Dev-C++. This is somewhat tedious, but it works reasonably well. To get this program to run in windowed mode, change the value `true` on line 34 of [Listing 15.3](#) to `false`.

The game initialization, starting on line 48, calls the `chloride::LoadResources()` function, which loads all of the bitmaps for Captain Chloride's animations. This function is not shown in this chapter because it's both long and straightforward. To view the code, please see the file `Chloride.cpp` on the CD in the folder `Source\Chapter15\Prog_15_01`. After the program loads the Captain's bitmaps, it sets his initial position on the screen on lines 5859.

During each frame, the program calls the `invasion::GetKeyboardInput()` function on line 68. If you look at the code on lines 97100, you can see that the `invasion::GetKeyboardInput()` function simply calls the `chloride::GetKeyboardInput()` function. If this were a multiplayer program, it would also get keyboard input for any additional players.

After the program gets the current keyboard input, it reacts to the input on line 69. As shown previously, the `chloride::Update()` function moves the Captain based on the player's input.

Because Captain Chloride is the only thing on the screen, the `invasion` class's `Render()` function just calls the `chloride::Render()` function during each frame. The `invasion` class's `OnKeyDown()` function, which is used for slower keyboard input, exits the game if the player presses the Escape or Q key.

That's all it takes to get the Captain walking around the screen.



# Summary

In this chapter you saw how to get an animated character up and running. This is the first step in building Invasion of the Slugwroths.

This chapter demonstrated how to load a character's animation frames and render them on the screen. It also illustrated the process of getting and responding to player input using LlamaWorks2D's immediate mode input.

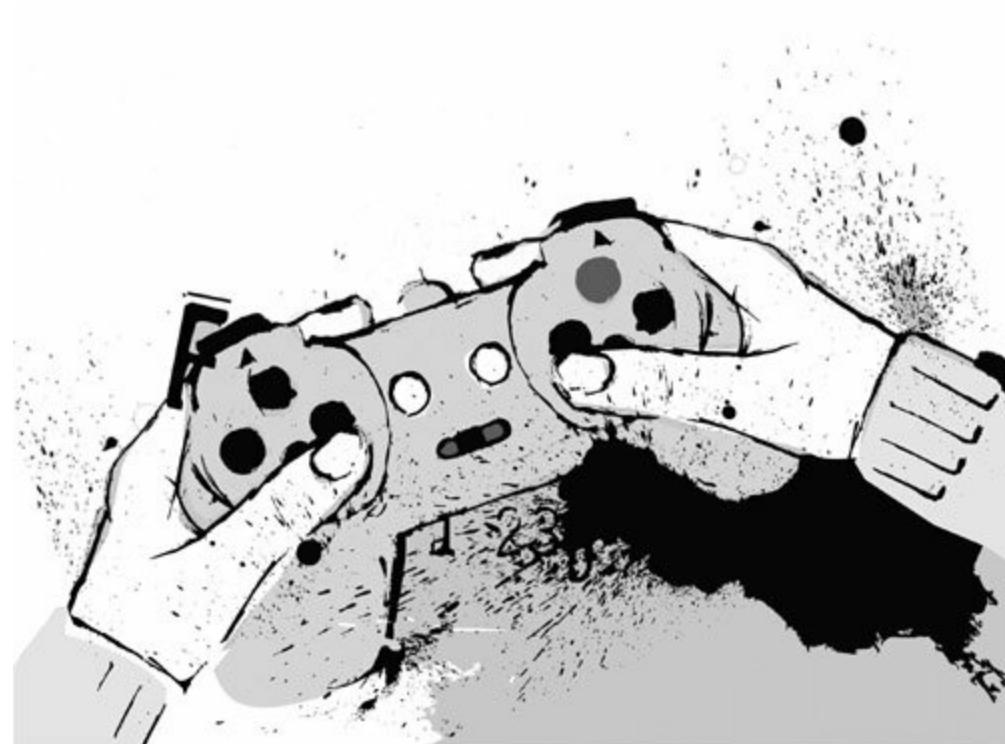
Walking Captain Chloride around on a blue background is a good first step. In the next chapter, we'll give him a world to explore.

# Chapter 16. The World of Captain Chloride

To get Captain Chloride to do anything interesting, we've got to put him into a world. In games, worlds are generally implemented as levels. And making a level requires creating a file that specifies what objects the level contains. The game must read the file and create all of the objects it specifies. Although this can be a challenging task, LlamaWorks2D provides a class that makes adding levels to your game fairly easy.

This chapter demonstrates the essentials of creating a level. It shows you how to put a character onto a background and scroll the background as the character moves.

Before Captain Chloride can go into a level, we need to make some improvements to him.



# The New Captain Chloride

The version of the chloride class in [chapter 15](#), "Captain Chloride Gets Going," used multiple animated sprites for clarity. In a real game, you do not need to do that. The LlamaWorks2D `animated_sprite` class allows you to put multiple animations into a single `animated_sprite` object. This enables you to create one `animated_sprite` object for a character and assign as many animations to it as you need.

[Listing 16.1](#) gives the new version of the `chloride` class. As you can see, it uses only one animated sprite. As the Captain moves, the program plays different animations depending on which direction he's moving.

## Listing 16.1. The new `chloride` class

```
1 class chloride
2 {
3 public:
4     // The Captain will go from one state to another
5     // depending on user input and other events.
6     enum state
7     {
8         STATE_STAND,
9         STATE_WALK_LEFT,
10        STATE_WALK_RIGHT
11    };
12
13    // These are the indices into 'anim' member.
14    enum anim_index
15    {
16        ANIM_WALK_LEFT=0,
17        ANIM_WALK_RIGHT=1
18    };
19
20    chloride();
21
22    bool LoadResources();
23
24    bool Update(
25        invasion *theGame);
26
27    bool Render(
28        invasion *theGame);
29
30    void X(
31        float upLeftX);
32    float X();
33
34    void Y(
35        float upLeftY);
36    float Y();
```

```

37
38 void GetKeyboardInput();
39
40 void Movement(
41     vectorf new_velocity);
42
43 int BitmapHeight();
44 int BitmapWidth();
45
46 private:
47     //indexed into by chloride::anim_index enum.
48     animated_sprite theCaptain;
49
50     sound leftWalkSound;
51     sound rightWalkSound;
52     vectorf velocity;
53     vectorf worldPos;
54     chloride::state currState;
55     chloride::state lastWalkState;
56     int lastStepIndex;
57 };

```

This version of the `chloride` class has a new enumeration on lines 1418. As I mentioned in [chapter 7](#), "Game Program Structure," enumerations are a way of defining constants. This enumeration defines a constant for walking left and another for walking right. Whenever the program needs to select a particular animation, it passes one of these two constants to the `animated_sprite` class's `CurrentAnimation()` function. It could pass numbers instead. However, these constants are more descriptive. If you see code like the following, you know that it makes Captain Chloride walk left:

```
theCaptain.CurrentAnimation(ANIM_WALK_LEFT);
```

However, if I were to write this without constants, it would be less obvious what is going on:

```
theCaptain.CurrentAnimation(0);
```

## Tip

Other programmers find it easier to read your code if you use constants rather than just numbers.

On line 53 of [Listing 16.1](#), the `chloride` class now has a floating-point vector that it did not have before. I put this in for a specific reason. Game programmers often use vectors to indicate the position of something on the screen. This is perfectly reasonable according to the rules of vectors. The vector on line 53 stores the location of Captain Chloride in the gamespace. Recall that coordinates in gamespaces are also called world coordinates. So we can say that this data member stores the Captain's location in world coordinates.

## Factoid

A vector can be used to describe the location of a point.

The member function in the new version of the `chloride` class works differently because of the differences in the member data. The majority of the changes occur in the `Update()` and `Render()` functions. These are shown in [Listing 16.2](#)

## Listing 16.2. The workhorse functions of the `chloride` class

```
1 bool
2 chloride::Update(
3     invasion *theGame)
4 {
5     // update the world position.
6     worldPos += velocity;
7
8     // correct the world position if necessary.
9     theGame->CollisionCheck(
10        worldPos, BitmapWidth(), BitmapHeight());
11
12    // update the "camera".
13    // this is the world position that we would like
14    // to be in the center of the screen if the world
15    // had no boundaries.
16    float cameraWorldX =
17        X() + 0.5 * (float)BitmapWidth();
18    theGame->SetCamera(cameraWorldX);
19
20    return (true);
21 }
22
23
24 bool
25 chloride::Render(
```

```

26  invasion *theGame)
27  {
28      bool renderOK = true;
29      int currentAnimation;
30
31  switch( currState)
32  {
33      case STATE_STAND:
34          // pause the animation
35          currentAnimation =
36              theCaptain.CurrentAnimation();
37          theCaptain.Animation(
38              currentAnimation)->CurrentFrame(
39              lastStepIndex);
40      break;
41
42      case STATE_WALK_LEFT:
43          lastWalkState = STATE_WALK_LEFT;
44
45          theCaptain.CurrentAnimation(ANIM_WALK_LEFT);
46
47          lastStepIndex =
48              theCaptain.Animation(
49                  ANIM_WALK_LEFT)->CurrentFrame();
50
51          // When the foot touches the ground, play sound.
52          if (lastStepIndex == 3)
53          {
54              leftWalkSound.Play();
55          }
56      break;
57
58      case STATE_WALK_RIGHT:
59          lastWalkState = STATE_WALK_RIGHT;
60
61          theCaptain.CurrentAnimation(ANIM_WALK_RIGHT);
62
63          lastStepIndex =
64              theCaptain.Animation(
65                  ANIM_WALK_RIGHT)->CurrentFrame();
66
67          // When the foot touches the ground, play
68          sound.
69          if (lastStepIndex == 3)
70          {
71              rightWalkSound.Play();
72          }
73      break;
74      default:
75          ASSERT( 0); //Unhandled state!
76      break;
77  }
78
79  // Transform from world to screen and render.
80  vector screenPos;
81  theGame->WorldToScreen(worldPos, screenPos);
82  theCaptain.X(screenPos.X());
83  theCaptain.Y(screenPos.Y());
84  theCaptain.Render();
85

```



```
86 return (renderOK);  
87 }
```

During each frame of screen animation, the program calls the `chloride` class's `Update()` function. Updating the Captain when he was walking around the soothing blue background was easy. This version, however, does more work. First, the `Update()` function sets Captain Chloride's position in world coordinates on line 6 of [Listing 16.2](#). Next, it checks to see if the Captain has collided with anything. Because the `chloride` class doesn't "know" what's in the level, it must ask the game object to perform collision checking, as shown on lines 9-10. I'll present this collision-checking function later in this chapter.

## Note

This version of Invasion of the Slugwroths only lets Captain Chloride move left or right. Therefore the camera's y value does not change as the game runs. That's why the `chloride` class only updates the camera's x position.

Before the `chloride` class's `Update()` function ends, it updates the position of an imaginary camera that follows Captain Chloride through the world. In actuality, there is no camera. What's really happening here is that the program keeps the Captain at the center of the screen. As he moves left or right, the background scrolls in the opposite direction. However, it's helpful to imagine that there's a camera that follows him. The statement on lines 16-17 sets the camera's x position to Captain Chloride's x position plus half the width of his bitmap image. This centers it on his image. The statement on line 18 sets the camera's y position.

The `Render()` function for the `chloride` class is similar to the version in [chapter 15](#). Each `case` in the `switch` statement still plays the appropriate animation. The primary difference is that all of the animations are now in one `animated_sprite` object. For example, when the Captain is supposed to walk to the left, the statement on line 45 uses the constants defined in the `chloride` class to select the animation of him walking left.

To freeze him in place when he stops walking, the `Render()` function saves the frame number of the current frame in whichever animation is currently

playing. For instance, if he's walking left, the program executes the statements on lines 4749. What happens here is that the `Render()` function invokes the `animated_sprite` class's `Animation()` function to get the current animation. The `Animation()` function returns a pointer. The statements on lines 4749 use that pointer to call the `animation` class's `CurrentFrame()` function. This retrieves the current frame number.

Just before returning, the `Render()` function changes Captain Chloride's position in world coordinates to coordinates on the screen. It then uses the `animated_sprite` class's `Render()` function to display his image on the screen.

At this point, Invasion of the Slugwroths has a chloride object that works in a level. This means we're ready to examine how levels work.

# Levels in LlamaWorks2D

The LlamaWorks2D game engine contains a class called `level`. The `level` class does most of the work of loading level files, figuring out what they contain and allocating the objects they specify.

In your games, you won't use the `level` class as it is. The LlamaWorks2D `level` class provides the basic functionality of a level, but you need to derive your own class from the `level` class. This enables you to add functionality that is specific to your game.

Before we dive into how to use the `level` class, I want to take a moment to clarify what goes in your custom level class and what you should put into your game class. Until now, all of the game logic for the sample programs has existed in their respective game classes. Once you start using the LlamaWorks2D `level` class, you'll need to divide the game's work between the level and the game classes. Here's the rule of thumb to use: if information needs to be kept from level to level, it should go in your game class. Otherwise, it goes in your level class.

For instance, the sample game contains the character Captain Chloride. The Captain moves from level to level. Therefore, he should exist in the game class. The player's score, power-ups, and weapons should also be stored in the game class because they continue with the player from level to level.

On the other hand, all of the monsters in a level are specified in the level file. The game logic for handling them should go in the level class. The same is true of the background. Its bitmap should be given in the level file. The logic for handling anything the level contains should be in your level class.

Let's see how all of this works by first examining a simple level file.

## Xml Level Files

I mentioned back in [chapter 12](#), "File Input and Output," that games generally store their level files as binary information. They typically save their configuration information in text files containing information in Extensible Markup Language (XML) format. LlamaWorks2D breaks that mold by requiring XML level files. I've found that it's much easier for those new to game programming to fix problems with their levels if the level files are in XML.

XML files always hold text tags. The rules of XML say that the tags usually occur in pairs. In LlamaWorks2D, they *always* occur in pairs. Pairs of XML tags can contain information, other tags, or both.

The XML schema used by LlamaWorks2D is defined in [Table 16.1](#). The table shows all of the tags that LlamaWorks2D recognizes, what they mean, what tags they must be contained in, and what tags they contain.

**Table 16.1. Llamaworks2D XML Tags**

Tag Pair	Description	Contains	Contained By
<AnimatedSprite> </AnimatedSprite>	Marks the start and end of a sprite.	<Animation>, <CurrentAnimation>, <XY>, <Velocity>, <BoundingRectangle>	<Level>
<Animation> </Animation>	Marks the start and end of an animation.	<Frames>, <TransparentColor>, <CurrentFrame>, <LoopStyle>	<AnimatedSprite>
<Blue></Blue>	Specifies a blue value in a color.	A floating-point blue value.	<Transparent-Color>
<Bottom></Bottom>	Contains the lower boundary of a bounding rectangle.	An integer specifying the bottom of the bounding rectangle.	<Bounding-Rectangle>
<Bounding-Rectangle> </Bounding-Rectangle>	Specifies the limits of a bounding rectangle.	<Top>, <Bottom>, <Left>, <Right>	<Sprite>, <Animated-Sprite>
<CurrentFrame>	Specifies the current frame in an animation.	An integer frame number.	<Animation>
<Frames></Frames>	Defines a set of images used as frames in an animation.	<ImageFileName>	<Animation>
<Green></Green>	Specifies a green value in a color.	A floating-point green value.	<Transparent-Color>
<ImageFileName>, </ImageFileName>	Specifies the name of a bitmap file.	A filename.	<Sprite>, <Animated-Sprite>, <Frames>

<Left></Left>	Contains the left boundary of a bounding rectangle.	An integer specifying the left side of the bounding rectangle.	<Bounding-Rectangle>
<Level></Level>	Marks the start and end of a level.	<Sprite>, <AnimatedSprite>, Custom tags	N/A
<LoopStyle>	Selects the looping style of an animation.	One of the following values: FORWARD, REVERSE, FORWARD_THEN_REVERSE	<Animation>
<Red></Red>	Specifies a red value in a color.	A floating-point red value.	<Transparent-Color>
<Right></Right>	Contains the right boundary of a bounding rectangle.	An integer specifying the right side of the bounding rectangle.	<Bounding-Rectangle>
<Sprite></Sprite>	Marks the start and end of a sprite.	<ImageFileName>, <TransparentColor>, <XY>, <Velocity>, <BoundingRectangle>	<Level>
<Top></Top>	Contains the upper boundary of a bounding rectangle.	An integer specifying the top of the bounding rectangle.	<Bounding-Rectangle>
<TransparentColor></TransparentColor>	Specifies the transparent color for a bitmap.	<Red>, <Green>, <Blue>	<Sprite>, <Animation>
<Velocity></Velocity>	Specifies the direction and speed of an object.	<X>, <Y>	<Sprite>, <Animated-Sprite>
<X></X>	Specifies the x component of a point or vector.	An integer x value.	<XY>, <Velocity>
<XY></XY>	Contains the x and y coordinates of an object. Must be integers.	<X>, <Y>	<Sprite>, <Animated-Sprite>

---

The collection of XML tags in [Table 16.1](#) form the schema that LlamaWorks2D uses to create levels. In addition to these tags, you can make your own tags to have custom objects in your levels. For example, you could create a pair of tags called `<Monster>` and `</Monster>` to add a monster to your level. The next section shows how to use the XML and extend it with your own custom tags.

## A Level File for Invasion of the Slugwroths

[Listing 16.3](#) shows the level file for Program 16.1, which you'll find on the CD in the folder `\Source\Chapter16\Prog_16_01`. Because the level is very simple, the level file is quite short.

### Listing 16.3. The XML level file for Program 16.1

```
1 <Level>
2
3   <CaptainChloride>
4     <XY>
5       <X>320</X>
6       <Y>350</Y>
7     </XY>
8   </CaptainChloride>
9
10  <Background>
11    <Sprite>
12      <ImageFileName>background.bmp</ImageFileName>
13    </Sprite>
14  </Background>
15
16 </Level>
```

All level files for LlamaWorks2D must begin with the XML tag `<Level>` and end with `</Level>`. If either one of these tags is missing, LlamaWorks2D generates an error. Your levels can contain three kinds of items. First, they can hold sprites. Sprites are defined with the `<Sprite>` and `</Sprite>` tags. Second, levels can contain animated sprites, which are defined with the `<AnimatedSprite>` and `</AnimatedSprite>` tags. Third, they can contain your custom objects specified with your custom tags.

The level in [Listing 16.3](#) has two custom objects. One represents Captain

Chloride himself. The other is the background. Recall that the object in the game that represents Captain Chloride is kept in the program's game class. However, each time a level starts, Captain Chloride can be in a different position in the level. Therefore, his starting position is specified in the level file.

The background object is actually a customized sprite. In fact, when you use LlamaWorks2D, everything on the screen is a `sprite`, an `animated_sprite`, or something derived from one of those two classes. In this case, the background is just a `sprite` object that the program scrolls back and forth.

When it starts up, the program reads the level file in [Listing 16.3](#). It has to separate each individual tag and value in the level file. It must then figure out what they mean and react accordingly. This process is called [parsing the tokens](#). In a LlamaWorks2D level file, the tokens are XML tags. Next, we'll examine how parsing is done.

## Parsing Level Files

As your game parses a level file, it adds objects to LlamaWorks2D's collection of visible objects. To parse the level file, a game using LlamaWorks2D needs a custom object derived from the `level` class. The custom class that the version of Invasion of the Slugwroths uses is shown in [Listing 16.4](#).

### Listing 16.4. The `chloride_level` class

```
1 class chloride_level : public level
2 {
3 public:
4     chloride_level();
5
6     bool Update(
7         invasion *pInvasion);
8
9     bool Render(
10        invasion *pInvasion);
11
12    bool ObjectFactory(
13        FILE *levelFile,
14        std::string tagToParse);
15
16    float ChlorideStartX();
17    float ChlorideStartY();
18
19    float MinX();
20    float MinY();
```

```

21 float MaxX();
22 float MaxY();
23
24 private:
25 float chlorideStartX, chlorideStartY;
26
27 float maxWorldX, maxWorldY;
28 };

```

The `chloride_level` class contains functions to update and render the level. These functions are overrides of the same functions in the `llamaworks2d::level` class. If you don't provide these functions, the versions in the `level` class are called. The `level::Update()` function doesn't actually do anything. It just returns `true`. The `level::Render()` function automatically renders every object in LlamaWorks2D's collection of visible objects. To get everything displayed on the screen properly, your custom level class should call the `level::Render()` function.

One of the most important functions in the `chloride_level` class is `ObjectFactory()`. If your file contains any custom XML tags, your level class must have a function called `ObjectFactory()`. It's in this function that your level class parses your custom XML. [Listing 16.5](#) gives the `ObjectFactory()` function for the `chloride_level` class.

## Listing 16.5. The `ObjectFactory()` function

```

1 bool chloride_level::ObjectFactory(
2     FILE *levelFile,
3     std::string tagToParse)
4 {
5     bool parseOK = true;
6     std::string temp;
7
8     if (IsTag(tagToParse, "CaptainChloride"))
9     {
10        EliminateWhiteSpace(levelFile);
11        temp = ReadTagOrValue(levelFile);
12        int startX, startY;
13        parseOK = ParseXY(levelFile,
14            startX, startY);
15        if (parseOK)
16        {
17            chlorideStartX = (float)startX;
18            chlorideStartY = (float)startY;
19
20            EliminateWhiteSpace(levelFile);
21            temp = ReadTagOrValue(levelFile);
22
23            if (!IsTag(temp, "/CaptainChloride"))
24            {
25                parseOK = false;

```



```

26     }
27     }
28 }
29 else if (IsTag(tagToParse,"Background"))
30 {
31     sprite *theBackground = new sprite;
32     if (theBackground != NULL)
33     {
34         EliminateWhiteSpace(levelFile);
35         temp = ReadTagOrValue(levelFile);
36         if (IsTag(temp,"Sprite"))
37         {
38             parseOK = ParseSprite(levelFile, theBackground);
39             if (parseOK)
40             {
41                 allObjects.push_back(theBackground);
42             }
43         }
44
45         EliminateWhiteSpace(levelFile);
46         temp = ReadTagOrValue(levelFile);
47
48         if (!IsTag(temp,"/Background"))
49         {
50             parseOK = false;
51         }
52         if (parseOK)
53         {
54             maxWorldX = (float)theBackground->BitmapWidth();
55             maxWorldY = (float)theBackground->BitmapHeight();
56         }
57     }
58     else
59     {
60         parseOK = false;
61         theApp.AppError(LWES_OUT_OF_MEMORY);
62     }
63 }
64
65 return (parseOK);
66 }

```

The `chloride_level::ObjectFactory()` function starts by declaring the variables it needs. It then examines its `tagToParse` parameter. When `LlamaWorks2D` calls the `ObjectFactory()` function, it passes the XML tag it doesn't recognize in the `tagToParse` parameter.

## Note

When your `ObjectFactory()` function looks for XML tags, such as `<CaptainChloride>` or `</CaptainChloride>`, it does not need to include the `<` and `>` characters. `LlamaWorks2D` removes them.

For example, line 8 of [Listing 16.5](#) tests to see if the tag was `<CaptainChloride>`. To do so, it calls the `level` class's `IsTag()` function. This is one of several parsing functions that the `level` class provides to your custom level classes. All of the parsing functions in the `level` class are protected, which means they are available to classes derived from the `level` class. You can find the prototypes for these functions in the LlamaWorks2D file called `LW2DLevel.h`.

If the tag is `<CaptainChloride>`, the `ObjectFactory()` function calls the `level::EliminateWhiteSpace()` function. This is another parsing function in the `level` class. It throws away white space such as tabs, spaces, endlines, and so forth.

At this point, the `ObjectFactory()` function reads the next tag or value to parse by calling `level::ReadTagOrValue()`. The next thing the `ObjectFactory()` function should encounter in the XML file is the `<XY>` tag. If it doesn't, the XML file is wrong.

Because the `<XY>` tag is already part of the schema that LlamaWorks2D uses, the `ObjectFactory()` function can call the `level::ParseXY()` function to parse it. To do so, it must pass the variables to receive the x and y values as the second and third parameters in the `ParseXY()` function's parameter list. If the `ParseXY()` function is successful, it returns `TRue` and the variables contain the x and y values it parsed. The `ObjectFactory()` function stores the x and y values for Captain Chloride's starting location in some private data members of the `chloride_level` class. When the level starts, the game can query these data members to place Captain Chloride properly in the level.

## Warning

I did not put error checking into the `ObjectFactory()` function in [Listing 16.5](#) to detect when the `<XY>` tag is missing. I did this to keep the code brief. However, your parsing code should not do this as it will cause your game to crash when the XML file is wrong.

The next XML tag the `ObjectFactory()` function should encounter is `</CaptainChloride>`. To parse it, `ObjectFactory()` throws away any white space and then reads the next tag. If it is not `</CaptainChloride>`, `ObjectFactory()` sets a status variable to return the error.

## Note

The `level` class in `LlamaWorks2D` has a protected data member called `allObjects`. Because this member is protected, it is available to classes derived from `level`. The `allObjects` member is a fancy type of array provided by the C++ Standard Template Library. Its `push_back()` member function adds items to the array. It also has a `size()` function that tells how many items are in the array.

Beginning on line 29 of [Listing 16.5](#), `ObjectFactory()` parses the `<Background>` tag. The background is actually a `sprite` object, so line 31 allocates a `sprite`. It calls the `level::ParseSprite()` function to parse the sprite information. If it does that successfully, it stores the background in the `level` class's collection of screen objects. It does this by passing the background's `sprite` to the `push_back()` function.

The `ObjectFactory()` function looks for the `</Background>` tag on lines 4551. If it isn't next in the XML file, `ObjectFactory()` sets an error status variable. When `ObjectFactory()` successfully parses the information between the `<Background>` and `</Background>` tags, it sets the height and width of the world coordinate system to the height and width of the background bitmap.

With an XML level file and an `ObjectFactory()` function, the program has what it needs to load a level.

## Loading a Level

When the `Invasion of the Slugwroths` program starts, `LlamaWorks2D` calls the `invasion::InitGame()` function. At that time, it initializes the `chloride` object that represents Captain Chloride. It also calls a function to load a level. The code for these two functions appears in [Listing 16.6](#).

### Listing 16.6. The game class's functions for loading a level

```
1  bool invasion::InitGame()
2  {
3      bool initOK = true;
4
5      initOK = theCaptain.LoadResources();
6  }
```

```

7   if (initOK==true)
8   {
9       initOK = InitLevel();
10  }
11
12  return (initOK);
13 }
14
15
16 bool invasion::InitLevel()
17 {
18     bool initOK = true;
19
20     initOK =
21         currentLevel.Load(
22             "Invasion.xml",
23             level_file_base::LWLFF_XML);
24
25     if (initOK==true)
26     {
27         theCaptain.X(
28             currentLevel.ChlorideStartX());
29         theCaptain.Y(
30             currentLevel.ChlorideStartY());
31     }
32
33     return (initOK);
34 }

```

**Listing 16.6** shows that the `invasion` class's `InitGame()` function invokes the `chloride` class's `LoadResources()` function to load all of the bitmaps for Captain Chloride's animations. It then calls the `invasion::InitLevel()` function, which starts on line 16. `InitLevel()` uses the `level` class's `Load()` function and passes it the name of the XML level file as the first parameter. The second parameter is a constant telling the `Load()` function that the level file is XML.

The `level::Load()` function reads the XML level file and parses it. As mentioned earlier in this chapter, `level::Load()` is able to parse any sprites or animated sprites in the XML level file. If `Load()` encounters any tags it does not recognize, such as the `<CaptainChloride>` and `<Background>` tags, it automatically calls the `chloride_level::ObjectFactory()` function that was given in [Listing 16.5](#). The `ObjectFactory()` function parses those tags. When the level has been loaded, `InitLevel()` calls a pair of functions from the `chloride_level` class to obtain Captain Chloride's starting position in the world.

## Updating a Level

Earlier in this chapter, in the section called "The New Captain Chloride," I

showed how the `chlbride` object is updated during each frame. Please refer back to [Listing 16.2](#) if you need a refresher on how that works. The program updates the Captain's position, status, and animations from the `invasion` class's `UpdateFrame()` function, as shown in [Listing 16.7](#).

## Listing 16.7. Functions for updating and checking for collisions

```
1  bool invasion::UpdateFrame()
2  {
3      GetKeyboardInput();
4      bool updateOK = theCaptain.Update(this);
5      if (updateOK)
6      {
7          updateOK = currentLevel.Update(this);
8      }
9
10     return (updateOK);
11 }
12
13
14 void invasion::CollisionCheck(
15     vectorf &worldPos, double fWidth, double fHeight)
16 {
17     if (worldPos.X() < currentLevel.MinX())
18     {
19         worldPos.X(currentLevel.MinX());
20     }
21     else if (worldPos.X() > currentLevel.MaxX() - fWidth)
22     {
23         worldPos.X( currentLevel.MaxX() - fWidth);
24     }
25
26     if (worldPos.Y() < currentLevel.MinY())
27     {
28         worldPos.Y( currentLevel.MinY());
29     }
30     else if (worldPos.Y() > currentLevel.MaxY() - fHeight)
31     {
32         worldPos.Y(currentLevel.MaxY() - fHeight);
33     }
34 }
```

The `invasion` class has an object of type `chlbride` called `theCaptain`. During each frame, the `UpdateFrame()` function calls the `chlbride::Update()` function, as shown on line 4 of [Listing 16.7](#). When `chlbride::Update()` executes, it checks for collisions between the Captain and objects in the level. `CollisionCheck()`, the function it calls to perform the check, is in the `invasion` class. You can find its code beginning on line 14. The current version of `CollisionCheck()` is extremely simple. It just uses `if-else` statements

to determine whether the Captain's bitmap has moved off the screen. However, when we add more objects to Captain Chloride's world, this function will grow to check for more collisions.

## Rendering a Level

When Invasion of the Slugwroths renders a frame, it must render both the level and Captain Chloride, as shown in [Listing 16.8](#). Most of the complex work of rendering is done in the `chloride::Render()` function, which you saw back in [Listing 16.2](#).

### Listing 16.8. The `invasion::RenderFrame()` function

```
1 bool invasion::RenderFrame()
2 {
3     bool renderOK = currentLevel.Render(this);
4
5     if (renderOK)
6     {
7         renderOK = theCaptain.Render(this);
8     }
9
10    return (renderOK);
11 }
```

[Listing 16.9](#) demonstrates that rendering the level involves setting the position of the background on the screen and then rendering it.

### Listing 16.9. The `chloride_level::Render()` function

```
1 bool
2 chloride_level::Render(
3     invasion *theGame)
4 {
5     bool renderOK = true;
6     vector screenPos;
7     vectorf worldPos;
8
9     // The background is always at 0,0 in world coordinates.
10    sprite *theBackground = (sprite*)allObjects[BACKGROUND];
11    worldPos.X( 0.0);
12    worldPos.Y( 0.0);
13    theGame->WorldToScreen( worldPos, screenPos);
```

```
14 theBackground->X( screenPos.X());
15 theBackground->Y( screenPos.Y());
16
17 renderOK = level::Render();
18
19 return (renderOK);
20 }
```

## Note

If your game stores custom objects in a level, the custom objects must inherit from the LlamaWorks2D `screen_object` class.

The `chloride_level` class's `Render()` function inherits from the `llamaworks2d::level` class. As a result, it has direct access to the `level` class's collection of screen objects. The function uses this collection, which is called `allObjects`, on line 10. As I mentioned previously, this collection is a fancy array provided by the C++ Standard Template Library. Line 10 demonstrates that when the `chloride_level::Render()` function access this collection, it uses the collection just like any other array. The collection stores generic pointers to screen objects such as sprites and animated sprites. Everything in the collection must inherit from a class in LlamaWorks2D called `screen_object`. Both the `sprite` and `animated_sprite` classes inherit from `screen_object`.

The `chloride_level::Render()` function gets a pointer to the background on line 10. It type casts the pointer variable to be a pointer to `sprite` objects and saves it in the variable `theBackground`. The functions in your level class should use the same technique when they need access to an object in the collection.

On lines 11-15, the `chloride_level::Render()` function sets the position of the bitmap on the screen. In world coordinates, the position of the background is always (0,0). However, because the game updates its imaginary camera, it "knows" how to convert (0,0) in world coordinates to screen coordinates.

# Summary

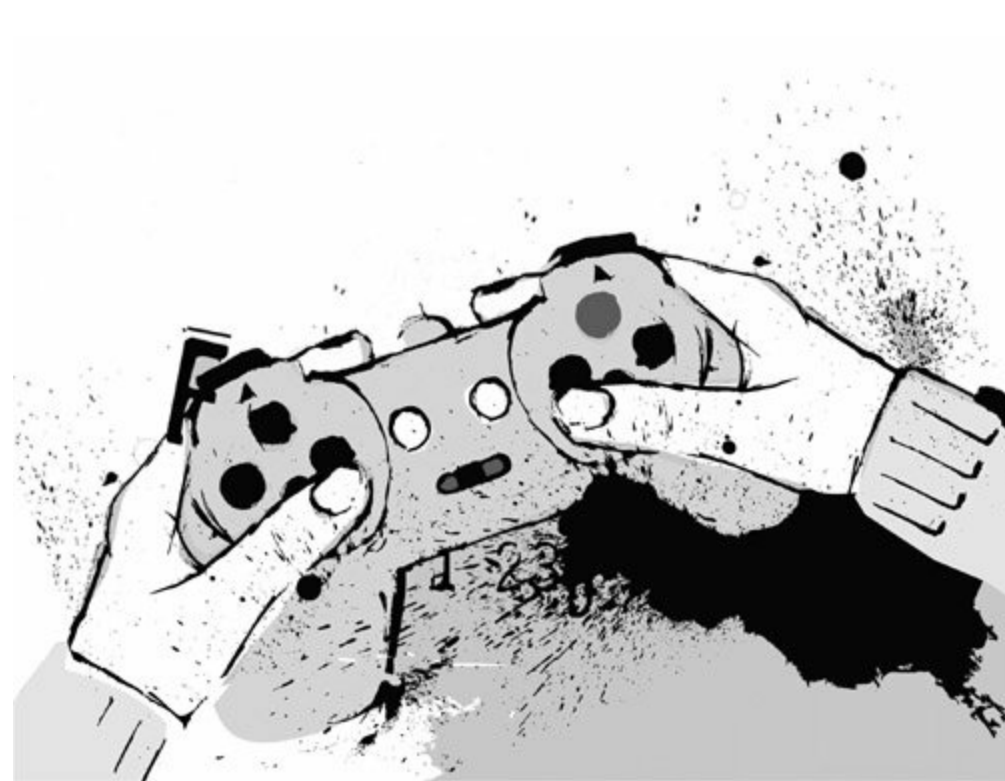
This chapter presented quite a bit of material. You saw how to revise the Captain so that he can function properly in a level. You also learned how to use the support provided by LlamaWorks2D for levels. That included making a level file, creating custom objects, and storing them in the level's collection of screen objects. You now know how to access items in that collection so that you can update and render them.

The next step in learning to write real games is to start putting obstacles in a level. So that is what the next chapter is about.



# Chapter 17. Captain Chloride Encounters Solid Objects

This chapter continues the work of implementing the sample game, Invasion of the Slugwroths. In the previous chapter, you saw how to add Captain Chloride to a level. Because the world he existed in was nothing more than a background, our good Captain could walk right through doors and walls. In this chapter, I'll show you how to fix that. First, I'll explain how to use sprites to make the Captain bump into a door. Next, you'll learn how you can make Captain Chloride pick up keys. Finally, I'll demonstrate how to make the door open and close. If Captain Chloride doesn't have the key, the door won't open. If he does, it slides up to admit him inside.



# Bumping into a Solid Door

In LlamaWorks2D, everything that moves is a sprite or an animated sprite. That includes doors, which open and close. Putting a solid object, such as a door, into Captain Chloride's path is simply a matter of creating the doors as a sprite.

## Tip

Because walls, trees, rocks, and similar objects don't move, you should probably not define them as sprites. Instead, you can create custom objects that just contain an x, y location and a bounding rectangle. Any object that has a bounding rectangle can be tested for collisions.

The game must somehow detect when Captain Chloride bumps into sprites. As you may remember from the Ping program in previous chapters, detecting collisions is exactly what bounding rectangles are for. Invasion of the Slugwroths assigns a bounding rectangle to all sprites and animated sprites that Captain Chloride can have physical contact with. When the Captain's bounding rectangle overlaps the bounding rectangle of another sprite (or animated sprite), the game must react in some way. In the case of a door, the game reacts by playing a bumping sound and preventing the Captain from moving forward.

## Collision Management

Using bounding rectangles to detect collisions is actually just the first step in dealing with colliding onscreen objects. Reacting to collisions requires some sort of collision management system. Game programmers tend to be very particular about how this system is implemented, so I did not make it part of LlamaWorks2D. However, I'll demonstrate a simple collision management system by creating a class called `world_object`. The `world_object` class implements collision handling for all objects in Captain Chloride's world. [Listing 17.1](#) provides the definition for the `world_object` class.

## Note

Virtually all 2D games use bounding rectangles to detect collisions between sprites, animated sprites, and nonmoving objects on the screen. This technique can be easily extended into 3D, so most 3D games use it as well.

## Listing 17.1. A simple collision management class

```
1 class world_object
2 {
3 public:
4     world_object();
5
6     virtual bool Intersects(world_object &other);
7
8     virtual void Hit(world_object &stationary);
9
10    virtual bool Collidable();
11
12    virtual bool Visible();
13
14    void X( float x);
15    float X();
16
17    void Y( float y);
18    float Y();
19
20    void CollisionWidth(float w);
21    float CollisionWidth();
22
23    void CollisionHeight(float h);
24    float CollisionHeight();
25
26    virtual std::string Type() = 0;
27
28 protected:
29     vectorf worldPos;
30
31 private:
32     float collisionWidth;
33     float collisionHeight;
34     bool collidable;
35     bool visible;
36
37 public:
38     world_object *next;
39 };
```

## Note

You'll find the code for the `world_object` class in the files `WorldObject.h` and `WorldObject.cpp`. They are in the folder `Source\Chapter17\Prog_17_01`.

## Warning

The `next` member of the `world_object` class is public. I made this compromise to keep the list management code very simple. However, it's seldom a good idea to have public data members in professional programs. Public data members can be used in incorrect ways and cause objects to contain invalid data.

For each object that the game checks for collisions, it creates a `world_object`. It then adds each `world_object` to a list that it keeps. The list, which is called a *linked list*, is stored in the game class with a pointer to objects of type `world_object`. The pointer points to the first item in the list. When the game moves to the `next` item in the list, it uses a pointer in the `world_object` class called `next`. The `next` pointer points to the `next` item in the list. If the game has reached the end of the list, the `next` pointer contains `NULL`.

## Note

In the `world_object` class, the `Hit()` function doesn't do anything. The derived class implements the `Hit()` function to provide the reaction.

The `world_object` class is intended to be used as a base object for derived classes. It has a member function called `Intersects()` that tests to see whether two objects with bounding rectangles are currently intersecting. Its `Hit()` function, which should be overridden in derived classes, reacts to collisions.

The other member data and functions of the `world_object` class keep track of whether the object associated with the `world_object` can collide with other objects. They also tell whether the object is currently visible on the screen.

## A Basic Door

In this version of the game, the door simply blocks Captain Chloride's way. As you'll see in [chapter 18](#), "That's a Wrap," the door also plays a role in killing the evil Slugworths. For now, I'll demonstrate how to tell the game whether the door is open or closed. I'll also show how to detect collisions between Captain Chloride and the door.

### Note

In the section called "Making the Door Open and Close," which appears later in this chapter, I'll add more functionality to the `door` class.

As you probably expect by now, you create a door in a game by creating a `door` class. The `door` class needs an enumeration to show what state it's in. The door can be open, in the process of opening, closed, or in the process of closing.

[Listing 17.2](#) gives the code for the `door` class.

### Listing 17.2. An initial version of the `door` class

```
1 class door : public world_object
2 {
3 public:
4     // The door will go from one state to another
5     // depending on user input and other events.
6     enum state
7     {
8         STATE_CLOSED,
9         STATE_OPENING,
10        STATE_OPEN,
11        STATE_CLOSING
12    };
13
14    door();
15
16    std::string Type();
```

```
17
18 bool Update(
19     invasion *theGame);
20
21 bool Render(
22     invasion *theGame);
23
24 void Movement(
25     vectorf new_velocity);
26
27 private:
28     vectorf velocity;
29     vectorf worldPos;
30     door::state currState;
31 };
```

The enumeration on lines 612 of [Listing 17.2](#) define all of the states that the door can be in. The `Type()` function returns a string telling what data type this object is. In this version of the class, the `Update()`, `Render()`, and `Movement()` functions do nothing because the image of the door is part of the background. Notice that the `door` class inherits from the `world_object` class. That's where all of the work for the `door` object occurs.

## Note

The code for the `door` class is in the files `Door.h` and `Door.cpp` in the folder `Source\Chapter17\Prog_17_01`.

## The Very Collidable Captain Chloride

Enabling Captain Chloride to collide with objects in his world requires very little modification of the `chloride` class. In fact, the `chloride` class only needs one new function, named `Hit()`, and a modification to its `Update()` function. [Listing 17.3](#) gives the code for both.

## Note

The code for this version of the `chloride` class is in files `Chloride.h` and `Chloride.cpp` in the folder `Source\Chapter17\Prog_17_01`.

## Listing 17.3. The `Update()` and `Hit()` functions for the `chloride` class

```
1 bool
2 chloride::Update(
3     invasion *theGame)
4 {
5     // update the world position.
6     worldPos += velocity;
7
8     // correct the world position if necessary.
9     theGame->CollisionCheck(*this);
10
11     // update the "camera".
12     // this is the world position that we would like
13     // to be in the center of the screen if the world
14     // had no boundaries.
15     float cameraWorldX =
16         X() + 0.5 * CollisionWidth();
17     theGame->SetCamera(cameraWorldX);
18
19     return (true);
20 }
21
22
23 void
24 chloride::Hit(
25     world_object &stationary)
26 {
27     if (stationary.Type() == "door")
28     {
29         door *theDoor = (door*)&stationary;
30
31         // We hit a door. Go back to previous position
32         // and play a sound.
33         worldPos -= velocity;
34
35         // Need a way to know if this is still playing
36         // and to not play it again until the current
37         // instance has finished.
38         bonkSound.Play();
39     }
40 }
```

The version of the `chloride::Update()` function in [Listing 17.3](#) is almost the same as the version from [chapter 16](#), "The World of Captain Chloride." However, there is one key difference. Notice that on line 9, the `Update()` function invokes a function in the game class called `CollisionCheck()`. The `CollisionCheck()` function takes an

object derived from the `world_object` class as its only parameter. In this case, the object being passed in is Captain Chloride himself. The statement `*this`, which appears in the parameter list, refers to the current object. Because `Update()` is a member of the `chloride` class, and because there's only one object of that type in the entire game, the call on line 9 always passes in the object that represents Captain Chloride.

When the `chloride::Update()` function calls `CollisionCheck()`, the `CollisionCheck()` function compares the object that represents Captain Chloride to every other object on the screen. If there's an overlap between their bounding rectangles, a collision has occurred. In that event, the `chloride::Hit()` function is called automatically. The `Hit()` function begins on line 23 of [Listing 17.3](#).

When the `Hit()` function is called, it receives the object that the Captain collided with in its parameter list. The first thing that the `Hit()` function needs to do is find out what kind of object the Captain banged into. It calls the `Type()` function, which every class derived from `world_object` has. The `Type()` function simply returns a string containing the name of the class for that kind of object. If the string is "door" it means that Captain Chloride hit an object of type `door`. If that's the case, the `Hit()` function points a pointer at the door object on line 29. It uses a type cast to clearly state that the object is a `door`. It then positions Captain Chloride so that he can't move any farther. This prevents him from walking through the closed door. The `Hit()` function then plays a bonk sound.

## Captain Chloride Collides with Reality

The last thing needed to implement collision handling in the program is to write the game class's `CollisionCheck()` function. To check for collisions, `CollisionCheck()` tests to see whether the object being checked has gone off the screen. It then scans a list of all of the objects in the world to see if there is overlap between them and the object being tested. [Listing 17.4](#) provides the code for the `CollisionCheck()` function.

### Note

The code for the `CollisionCheck()` function is in the file `Invasion.cpp` in the folder `Source\Chapter17\Prog_17_01`.



## Listing 17.4. The `CollisionCheck()` function for the `invasion` class

```
1 void invasion::CollisionCheck(
2   world_object &movingObject)
3 {
4   // First test against the world limits
5   double width = movingObject.CollisionWidth();
6   double height = movingObject.CollisionHeight();
7   if (movingObject.X() < currentLevel.MinX())
8   {
9     movingObject.X(currentLevel.MinX());
10  }
11  else if (movingObject.X() > currentLevel.MaxX() width)
12  {
13    movingObject.X(currentLevel.MaxX() width);
14  }
15
16  if (movingObject.Y() < currentLevel.MinY())
17  {
18    movingObject.Y(currentLevel.MinY());
19  }
20  else if (movingObject.Y() > currentLevel.MaxY() height)
21  {
22    movingObject.Y(currentLevel.MaxY() height);
23  }
24
25  // Now test against all the objects
26  world_object *testObject = worldObjects;
27  while(testObject)
28  {
29    if (testObject != &movingObject)
30    {
31      if (testObject>Intersects(movingObject))
32      {
33        movingObject.Hit(*testObject);
34      }
35    }
36    testObject = testObject>next;
37  }
38 }
```

Because of the way it's designed, the game can use the `CollisionCheck()` function to check for collisions between virtually any two moving objects on the screen. It begins by obtaining the height and width of the object being tested. Next, it tests to see whether the x coordinate of the object is off the left edge of the screen. If it is, the function sets the object's x coordinate to the screen's left edge on line 9 of [Listing 17.4](#).

If the moving object has not gone off the left edge of the screen, the `CollisionCheck()` function tests to see whether it went off the right edge on line 11. If so, it adjusts the object's position so that it stays on the screen.

On lines 1623, the `CollisionCheck()` function checks whether the moving object went off the top or bottom of the screen. If so, it sets the y position to keep the object on the screen.

It's at this point that the `CollisionCheck()` function uses the linked list presented earlier in this chapter. The start of the list is a pointer to objects of type `world_object`. The `CollisionCheck()` function enters a loop on line 27. In this loop, it uses the `Intersect()` function to see if there's any overlap between the moving object and the current object in the linked list. If there is, `CollisionCheck()` calls the `Hit()` function for the moving object. It passes the current object in the list as the parameter for the `Hit()` function.

## Note

Pointers of type `world_object` can point at any object derived from the `world_object` class. Therefore, every object in the list must be derived from `world_object`.

If there is no overlap between the bounding rectangles of the moving object and the current object in the list, the loop executes the statement on line 36. This statement sets the pointer variable `testObject` to point at the next object in the list. The `CollisionCheck()` function then performs the loop test on line 27. This test checks to see whether the variable `testObject` points at anything. Another way to write this test is as follows:

```
while (testObject != NULL)
```

This statement and the statement on line 27 say exactly the same thing. They are both `true` while `testObject` does not contain the value `NULL`. When this loop gets to the last object in the linked list, the `next` pointer on line 36 is `NULL`. As a result, the variable `testObject` receives the value `NULL`. This causes the loop test on line 27 to be `false`, which makes the loop end.

Invasion of the Slugwroths now has a system that can handle collisions between any two moving objects on the screen. The code in Program 17.1 on the CD demonstrates how to bump Captain Chloride into solid objects such as doors. However, the game can use pretty much the same code to enable him to pick objects as well. That's the subject of the next section.



## Picking Up a Key

It's common in games for characters to pick up powerups, keys for doors, and so on. It's not hard to add that ability to Invasion of the Slugwroths. I'll present an example program, Program 17.2 on the CD, that enables Captain Chloride to pick up a key card. If he has the red key, he can open the red door. If not, the door won't open.

## Objects that Can be Picked Up

The first step is to add a class that represents the object to be picked up. The class, as you might expect, is called `key`. [Listing 17.5](#) presents its class definition.

### Note

This code is from the file `Key.h` in the folder `Source\Chapter17\Prog_17_02` on the CD.

### Listing 17.5. The definition of the `key` class

```
1 class key : public world_object
2 {
3 public:
4     key();
5
6     std::string Type();
7
8     bool Render(
9         invasion *theGame);
10
11 private:
12     vectorf velocity;
13
14 public:
15     sprite mySprite;
16 };
```

The `key` class has very little code because a key doesn't do much. It just lies there until Captain Chloride picks it up. Once he does, it's no longer visible on the screen. This isn't handled directly by the `key` class. Instead, the `world_object` class contains a private data member named `visible`. When `visible` is set to `false`, the object becomes essentially invisible. Because the `key` class inherits from `world_object`, it too can become invisible when needed.

The `key` class contains a data member of type `sprite`. The class uses this member to display the key's image on the screen.

## Warning

The `mySprite` member of the `key` class is another example of me making a data member public rather than private for simplicity. In a professional program, I would never do this because it can lead to accidental data corruption.

The member functions for the `key` class are equally simple. The only one I'll present here is the `Render()` function. If you want to see the others, please look at the source code on the CD. The `key` class's `Render()` function appears in [Listing 17.6](#).

## Listing 17.6. The `Render()` function from the `key` class

```
1 bool key::Render(invasion *theGame)
2 {
3     bool renderOK = true;
4     vector screenPos;
5     theGame>WorldToScreen(worldPos, screenPos);
6     mySprite.X(screenPos.X());
7     mySprite.Y(screenPos.Y());
8     renderOK = mySprite.Render();
9
10    return (renderOK);
11 }
```

[Listing 17.6](#) shows that the `Render()` function sets the position of the key in the level and uses the `sprite::Render()` function to draw the key to the screen.

To get the key into the level, you create XML tags for it in the level file. You also need to add some code to the `chloride_level` class to parse the XML for the key, dynamically allocate a key object, and initialize it with the information from the XML tags. This is the same process you saw in [chapter 16](#). Whether you're adding doors, keys, monsters, or whatever, you follow this same process. If you do, the game adds the key to the linked list of objects in the level. The game draws everything in the linked list every frame.

## Note

You can see the XML tags for the key in the `Invasion.xml` file in the folder `Source\Chapter17\Prog_17_02` on the CD. To see the code that parses the key's XML tags, please see the `ObjectFactory()` function in the file `ChlorideLevel.cpp` (which is in the same folder).

## Making Captain Chloride Pick Things Up

Getting Captain Chloride to pick up a key requires only minor modifications to the `chloride` class. You just need to add a data member that is `true` when he has the key and `false` if not. If you want to play a sound whenever the Captain picks something up, which is a good idea, simply add a member of type `sound` to the `chloride` class and load the "pickup" sound into it. I added both of these members to the `chloride` class in Program 17.2 on the CD. To view them, please see the `Chloride.h` file in the folder `Source\Chapter17\Prog_17_02`.

Recall that the `chloride` class has a member function named `Hit()`. Whenever the Captain encounters an object with a bounding rectangle, the game calls the `chloride::Hit()` function. Therefore, that's where the code goes that controls what happens when Captain Chloride encounters a key. [Listing 17.7](#) provides the code for the `chloride` class's `Hit()` function.

### Listing 17.7. Enabling Captain Chloride to pick up a key

```
1 void chloride::Hit(world_object &stationary)
2 {
3     if (stationary.Type() == "door")
4     {
5         door *theDoor = (door*)&stationary;
```

```

6
7     if (haveKey)
8     {
9         theDoor>Open( this);
10    }
11
12    worldPos = velocity;
13
14    bonkSound.Play();
15 }
16 else if (stationary.Type() == "key")
17 {
18     key *theKey = (key*)&stationary;
19     theKey>Visible(false);
20     theKey>Collidable(false);
21     pickupSound.Play();
22     haveKey = true;
23 }
24 }

```

The code on line 19 of [Listing 17.7](#) marks the key as invisible when the Captain collides with it. It also turns off the key's collision checking on line 20. The `Hit()` function plays a blip sound to notify the player that the Captain picked up the key. It also sets the `chlride` class's `haveKey` member to `true`.

As the Captain continues to walk around his world, he eventually collides with the door. When he does, the game calls the `Hit()` function again. This time, it executes the code on lines 514. If Captain Chloride has the key, the `Hit()` function calls the door's `Open()` function. In the `Open()` function, the door opens by sliding upward. As it does, you'll hear a nice door opening sound. When the Captain moves away from the door, it closes. The next section explains how all of that happens.

# Making the Door Open and Close

Creating a door that opens and closes is somewhat harder than picking up keys. You might ask why. Go ahead, ask.

That's a good question. The answer is that it's harder because it involves working in more than two dimensions.

Is this the part where I tell you all about 3D programming? In a word, no. Although it takes more than two dimensions to get Captain Chloride to walk through a door, it takes fewer than three.

Huh?

For years, game programmers have used a technique they call 2.5D programming. It's not 2D programming, but it's also not 3D programming. If you run Program 17.2 on the CD, you'll see 2.5D programming in action. When Captain Chloride, who is twodimensional, walks through the open door, he walks in front of the back part of the wall and behind the front part of the wall. All objects in the scene are 2D; however, they are arranged in layers.

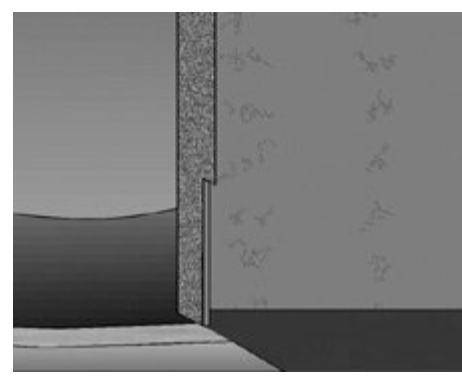
## Adventures in Two and a Half Dimensions

In 2D programming, the entire scene is painted on the background. This is how things worked in Program 17.1. The image of the door was part of the background. The program just puts a bounding rectangle at the door's position so that Captain Chloride has something to bump into.

To get Captain Chloride to walk through the doorway in a way that looks real, the game has to layer parts of the scene on top of each other. [Figures 17.1, 17.2, and 17.3](#) illustrate what I mean.

### **Figure 17.1. The Background**





[Figure 17.1](#) shows part of the background for Program 17.2. Notice that part of the wall has been cut away. All parts of the scenery remaining in the background are things that Captain Chloride walks in front of. This background goes in a layer underneath the image of the Captain. In other words, the program first renders the background and then renders the Captain. Next it draws the door's image, which is a sprite. [Figure 17.2](#) shows the door's image.

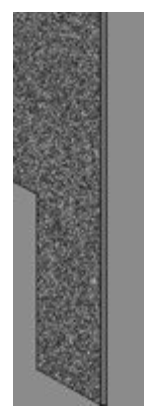
### **Figure 17.2. The door**



[Figure 17.2](#) shows the door with its transparent color. As I mentioned, the door is rendered as a sprite in the same layer as Captain Chloride. Making the door open is just a matter of sliding the sprite upward.

As the Captain walks through the open door, he walks behind the front part of the wall, which you can see in [Figure 17.3](#).

### **Figure 17.3. The front piece of the wall.**



The game draws the front part of the wall in a layer that's in front of Captain Chloride. As a result, the Captain appears to walk through the hole in the wall left when the door opens.

Tricky, don't you think?

Layering 2D images on top of each other in this manner extends 2D programming so that things appear more realistic. It's called 2.5D programming because it involves more than two dimensions and fewer than three.

## Two and a Half Dimensional Programming in Action

Programmers have many ways of adding two and a half dimensions to their games. Because there are many approaches, I did not add direct support for 2.5D programming in LlamaWorks2D. I didn't want to limit you to one approach.

There are two techniques for 2.5D programming that are particularly common. Some programmers just make everything a generic sprite and order things in their scenes so that everything is drawn correctly from back to front. Others divide their scenes into distinct layers and assign sprites into the layers.

For beginners, the first technique is easiest. I demonstrate it in Program 17.1. As you move into more advanced programming and your games become more complex, I suggest you try the second technique, which I demonstrate in Program 17.2.

As a first step in implementing layered 2.5D programming, I created an object for the game called `scene_sprite`. This class is shown in [Listing 17.8](#).

## Listing 17.8. The `scene_sprite` class

```
1 class scene_sprite : sprite
2 {
3 public:
4     bool scene_sprite::Render(
5         invasion *theGame);
6
7     void WorldX(float x);
8     float WorldX();
9
10    void WorldY(float y);
11    float WorldY();
12
13 private:
14     vector<float> worldPos;
15 };
```

The `scene_sprite` class is a special class for sprites that Captain Chloride can't interact with in any way. The only thing you can do with a `scene_sprite` object is render it. The XML level file for Program 17.2 contains some XML tags that allocate the front piece of the wall as a `scene_sprite` object and position it correctly in the scene. It covers the back part of the wall, which is part of the background. It also covers the door when the door slides up.

With the `scene_sprite` class implemented, you can now divide the scene into distinct layers. Instead of having one `Render()` function like most visible objects in the game, the `chloride_level` class can now use two. [Listing 17.9](#) gives the code for them.

## Listing 17.9. The `chloride_level` class's rendering functions

```
1 bool chloride_level::RenderBackground(
2     invasion *theGame)
3 {
4     bool renderOK = true;
5
6     scene_sprite *theBackground =
7         (scene_sprite*)allObjects[BACKGROUND];
8     renderOK = theBackground->Render( theGame);
9
10    return (renderOK);
11 }
12
13
14 bool chloride_level::RenderForeground(
15     invasion *theGame)
```

```

16 {
17     bool renderOK = true;
18
19     scene_sprite *theWallFront =
20         (scene_sprite*)allObjects[WALLFRONT];
21     renderOK = theWallFront>Render( theGame);
22
23     return (renderOK);
24 }

```

Recall that under the rules of LlamaWorks2D, the game class normally calls the level class's rendering function. Because Program 17.2 implements layered 2.5D scenery, the `invasion::RenderFrame()` function first calls the `chloride_level::RenderBackground()` function. As the function name implies, this renders everything in the background behind the moving objects in the level.

The next step is for the `invasion::RenderFrame()` function to render everything between the background and foreground. I'll show how this is done shortly.

Finally, `invasion::RenderFrame()` must call the `chloride_level::RenderForeground()` function. As [Listing 17.9](#) shows, `RenderForeground()` renders the front piece of the wall.

The code for the `invasion::RenderFrame()` function appears in [Listing 17.10](#). As you can see, it performs its rendering in the three distinct layers that I've described.

## Listing 17.10. The `invasion::RenderFrame()` function

```

1 bool invasion::RenderFrame()
2 {
3     bool renderOK = currentLevel.RenderBackground(this);
4
5     world_object *oneObject = worldObjects;
6     while( oneObject)
7     {
8         if (renderOK)
9         {
10            if (oneObject>Visible())
11            {
12                renderOK = oneObject>Render(this);
13            }
14        }
15        oneObject = oneObject>next;
16    }
17
18    if (renderOK)
19    {
20        renderOK = currentLevel.RenderForeground(this);

```

```
21 }
22
23 return (renderOK);
24 }
```

As you read [Listing 17.10](#), the `RenderFrame()` function uses the fact that all of the objects between the background and the foreground are in the linked list. That's because the linked list contains all of the objects that interact with one another. It's precisely those objects that occur between the background and the foreground. Therefore, the `RenderFrame()` function moves through the linked list and renders each object it finds there.

By calling `chloride_level::RenderBackground()`, rendering everything in the linked list, and then calling `chloride_level::RenderForeground()`, the `invasion::RenderFrame()` function divides everything in the scene into distinct layers.

# Summary

When you write a game with a character moving around in a world, the character must have contact with solid objects. This includes objects that the character bumps into and picks up.

If your character must encounter an object that doesn't move but blocks his movement in the scene, the simplest way to program that is to have the object's image be part of the background. You simply position a bounding rectangle at that location in the scene. When the character hits the bounding rectangle, his progress stops.

For more complex objects that the character bumps into, you can create the object as a sprite or as a collection of layered sprites. Game programmers call layering sprites in a scene *2.5D programming*. The easiest way to do 2.5D programming is to simply load the back sprites into the scene before the front sprites. That way, the back sprites are drawn first. When the game draws the front sprites, they cover the parts of the back sprites that need to be covered.

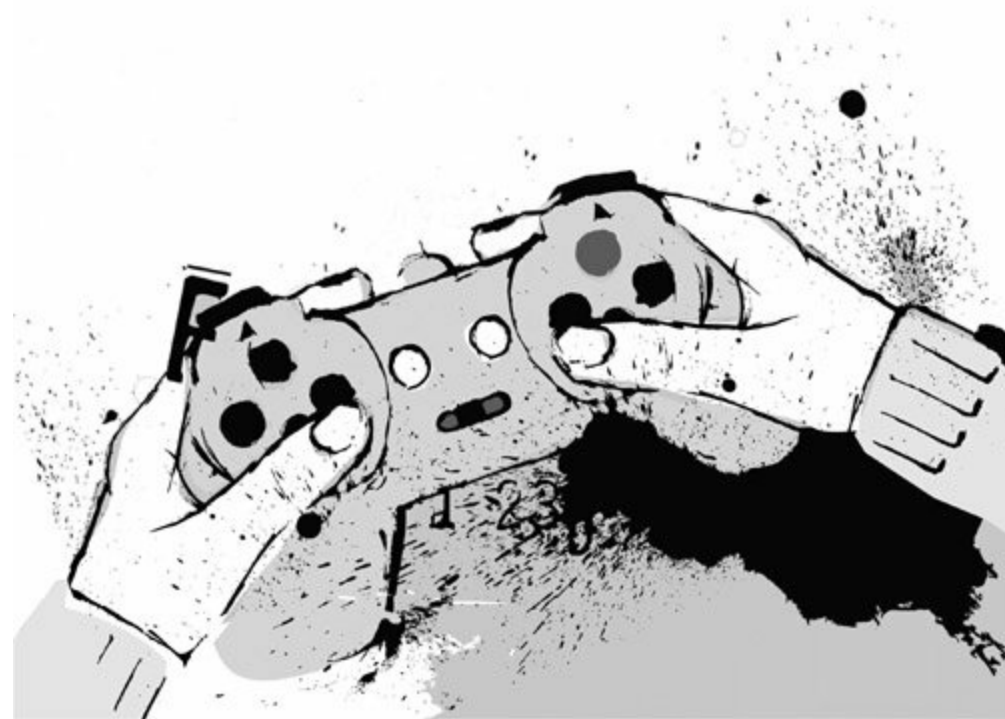
Game characters usually pick up objects by walking over them. Games draw such objects in the scene as sprites that turn invisible after they're picked up.

In the next chapter, we'll wrap things up by demonstrating how to add Slugwroths to the level.

# Chapter 18. That's a Wrap!

With a lot of perseverance, you've reached the last chapter of this book. You've followed along as a game has been developed and programmed. By the end of this chapter, you'll know enough of the basics to write games of your own; there are just a few loose ends to tie up.

In this chapter, you'll see how to simplify your games a bit by turning over more of the work to LlamaWorks2D. I'll then demonstrate the simplified programming style by putting the evil and foul-smelling Slugwroths into the game. Alas, one book is too short to present everything there is to know about game programming in C++. At the end of the chapter, I'll present a brief review of some of the skills that will take you to the next level of game programming.



# Time for Consolidation

In [chapter 17](#), I created a class called `world_object`. The `world_object` class represented things in the world for Captain Chloride and other characters to collide with. I used it to implement a straightforward system of collision detection and response. In this chapter, I am going to simplify the management of bitmaps that are loaded when the game reads its level file. I'll do this by making the `world_object` class use inheritance to derive from LlamaWorks2D's `screen_object` class.

## Note

You'll find the code for the `world_object` class on the CD in the folder `Source\Chapter18\Prog_18_01`.

Making the `world_object` class derive from the `screen_object` class gives your game the ability to store all objects of type `world_object` into a list that is automatically built and maintained by LlamaWorks2D's `level` class. As a result, your code does not have to delete objects of type `world_object` when a level finishes. That's done automatically by the `level` class.

## Factoid

If a class keeps a collection of dynamically allocated objects and it takes ownership of those objects, it must delete them when they are no longer used.

In [chapter 17](#), the `chloride_level::ObjectFactor()` function stored pointers to object of type `world_object` into the list kept by the `level` class. In this version, it stores the objects themselves. This enables the level class to delete them automatically when the level ends. Programmers say that the `level` class is talking [ownership](#) of all of the level's objects.

Ownership is a concept critical to games. Because games allocate so many of their objects dynamically, you must always keep track of which objects own other objects. To keep things simple, your games should always include a level object that owns all objects allocated for the level. When the level ends, the



level object should delete the objects it owns. This may sound obvious. However, if you don't use this technique, it's easy to leak memory at the end of a level because objects don't get deleted.

## Tip

Always be sure that you know which objects in your game take ownership for all the others.

In LlamaWorks2D, many objects assume ownership of other objects. The application object owns the game objects. The game object owns the level object. It can also own other objects. For example, the `invasion` class, which is the game class for Invasion of the Slugwroths, owns both the level object and the object that represents Captain Chloride. When the application object deletes the `invasion` object, the `invasion` object must make sure it deletes the `chloride` and `chloride_level` objects. The `chloride_level` object in turn deletes all objects in the level. Everything gets cleaned up nicely.

# Enter villains, Stage Left

Nearly every computer game has villains. For some reason, Earth is the one planet in the entire universe that every alien wants. There doesn't seem to be any shortage of life forms that want to have this world as their own. Invasion of the Slugwroths is no exception. The Slugwroths have established bases on Earth that must be cleaned out by Captain Chloride. How they managed to establish their bases in the first place is a mystery, but there you have it. What can the Captain do but load his salt shooter and embark on his solitary quest to free Earth?

## Designing a Simple Slugwroth

We won't implement Captain Chloride's salt shooter; I'll keep the elimination of the Slugwroths basic for the purposes of this book. Later in this chapter I'll explain how you can set the salt shooter up yourself. To kill Slugwroths, I made them stationary until the Captain bumps them. When he does, they chase him out of the base. He kills them by making them collide with the door. This is a simple way of demonstrating the same techniques needed to create a salt shooter that kills Slugwroths.

You may be surprised to find that you already know how to implement a class for Slugwroths. Making such a class is just a variation on the techniques I demonstrated for creating Captain Chloride himself. This is very evident in the class definition, which is shown in [listing 18.1](#).

### Listing 18.1. The class that represents Slugwroths

```
1 class slug : public world_object
2 {
3 public:
4     // This Slugwroth will go from one state to another
5     // depending on Captain Chloride's actions.
6     enum state
7     {
8         STATE_ASLEEP,
9         STATE_CHASING
10    };
11
12    slug();
13
14    std::string Type();
15
16    bool Update(
```

```

17     invasion *theGame);
18
19     bool Render(
20         invasion *theGame);
21
22     // If something wakes me up, I will chase it!
23     void WakeUp(
24         world_object *waker);
25
26     void Kill();
27
28     // If I walk into something, I will stop.
29     void Hit(
30         world_object &stationary);
31
32 private:
33     slug::state currState;
34     vectorf velocity;
35
36 public:
37     animated_sprite myAnims;
38 };

```

Like Captain Chloride, Slugwroths can have more than one state. They are either stationary, which I refer to as being asleep, or they are chasing the Captain. The `slug` class defines these two states in an enumeration on lines 610.

The `slug` class contains the functions you probably expect by now. Specifically, it has the `Type()`, `Update()`, and `Render()` functions. In addition, the class has a function to tell the Slugwroth to wake up and chase the Captain. It also contains a `Kill()` function that the program uses to kill the Slugwroth. The Slugwroth's animations are stored in its `myAnims` data member.

## Warning

As with classes in [chapter 17](#), the `slug` class contains a public data member. This is not how professional programs are written. I did this to simplify the code. If you do this in a game company, your coworkers may wonder how well you understand the concepts behind object-oriented programming. Seriously.

## Implementing the Slugwroth Class

The functions for the `slug` class, which are shown in [Listing 18.2](#), are fairly straightforward. You'll find them in the file `Slug.cpp`, which is in the folder `Source\Chapter18\Prog_18_01`.

## Listing 18.2. The member functions of the `slug` class

```
1  const float SPEED = 10.0;
2
3  const int ANIM_WALK_LEFT = 0;
4  const int ANIM_WALK_RIGHT = 1;
5
6  slug::slug() :
7      currState(STATE_ASLEEP)
8  {
9  }
10
11 bool
12 slug::Update(
13     invasion *theGame)
14 {
15     switch(currState)
16     {
17     case STATE_ASLEEP:
18         // Slugwroths are strange creatures and
19         // prefer to sleep facing to the right!
20         myAnims.CurrentAnimation(ANIM_WALK_RIGHT);
21         break;
22
23     case STATE_CHASING:
24         myAnims.CurrentAnimation(ANIM_WALK_LEFT);
25         worldPos += velocity;
26         theGame->CollisionCheck(*this);
27         break;
28
29     default:
30         ASSERT(0); //Unhandled state!
31         break;
32     }
33
34     return (true);
35 }
36
37 bool
38 slug::Render(
39     invasion *theGame)
40 {
41     bool renderOK = true;
42
43     vector screenPos;
44     theGame->WorldToScreen(worldPos, screenPos);
45     myAnims.X(screenPos.X());
46     myAnims.Y(screenPos.Y());
47     renderOK = myAnims.Render();
48
49     return (renderOK);
50 }
```

```

51 void
52 slug::WakeUp(
53     world_object *waker)
54 {
55     // We wake up and follow whatever hit us.
56     currState = STATE_CHASING;
57     if (waker>WorldX() < WorldX())
58     {
59         // Run to the left.
60         velocity.X(SPEED);
61     }
62     else
63     {
64         // Run to the right.
65         velocity.X(SPEED);
66     }
67 }
68 }
69
70 void
71 slug::Kill()
72 {
73     Collidable(false);
74     Visible(false);
75 }
76
77 void
78 slug::Hit(
79     world_object &stationary)
80 {
81     // We hit something. Go back to previous position.
82     worldPos = velocity;
83 }

```

There are some interesting things about the way I've written the `slug` class. The first thing to notice is that it uses a lot of constants instead of numbers. This is always a good idea because it helps make your code more readable to others on your project.

## Tip

In professional programs, it's advisable to use as many constants as you can rather than literals like numbers or strings in quotes. This makes your code more readable to other programmers who are working on the game with you. It also helps you remember why you did things the way you did. You'd be surprised how fast you can forget what literals stand for.

The constants defined on lines 14 of [Listing 18.2](#) are not defined in a function. So which functions can access them?

If you define a constant in a .cpp file using the C++ keyword `const` and you define the constant outside a function, all functions in that file can see and use the constant. The constant is visible from the point it's declared to the end of the file. Functions in other files cannot access the constant.

The constructor on lines 69 of [Listing 18.2](#) initializes the `currState` data member to `STATE_ASLEEP`. This makes the Slugwroth stay in one position until it's bumped. The change of state occurs in the `Update()` function, which appears on lines 1135. `Update()` begins by checking the current state of the Slugwroth. If it's asleep, `Update()` displays the Slugwroth in a frozen position facing right. If the Slugwroth is in hot pursuit of Captain Chloride, the `Update()` function sets the Slugwroth's velocity and performs a collision check.

Recall that the `invasion::CollisionCheck()` function automatically calls the `Hit()` function for the object it's checking. The code for the `slug::Hit()` function begins on line 77. All it does is prevent the Slugwroth from walking through things.

You may wonder how the Slugwroth makes the transition from being asleep to chasing the Captain. That occurs when the Captain bumps into the Slugwroth. The `chloride::Hit()` function, which is given in [listing 18.3](#), calls the `slug::WakeUp()` function. During each frame, `slug::Update()` moves the Slugwroth and checks for collisions.

### Listing 18.3. The `chloride::Hit()` function

```
1 void chloride::Hit( world_object &stationary)
2 {
3     if (stationary.Type() == "door")
4     {
5         door *theDoor = (door*)&stationary;
6
7         if (haveKey)
8         {
9             theDoor>Open( this);
10            // Hold on to the key; the door will close
11            // automatically and we might still need it.
12        }
13
14        // We hit a door. Go back to previous position
15        // and play a sound.
16        worldPos = velocity;
17
18        // Need a way to know if this is still playing
19        // and to not play it again until the current
20        // instance has finished.
```

```

21     bonkSound.Play();
22 }
23 else if (stationary.Type() == "key")
24 {
25     key *theKey = (key*)&stationary;
26     theKey>Visible(false);
27     theKey>Collidable(false);
28     pickupSound.Play();
29     haveKey = true;
30 }
31 else if (stationary.Type() == "slug")
32 {
33     // We hit a slug. Go back to previous position.
34     worldPos = velocity;
35
36     slug *theSlug = (slug*)&stationary;
37     theSlug>WakeUp( this);
38 }
39 }

```

The `chloride::Hit()` function wakes the Slugwroth so that it can start chasing the Captain. New game programmers often feel that this is an odd approach to things. Shouldn't it be the `slug` class's job to wake up a Slugwroth when something bumps it?

While that is a logical viewpoint, it's not practical. It's hard to make objects in a game detect when they've been bumped. It's much easier to detect when they bump something. Therefore, when the Captain bumps a Slugwroth, he essentially says to the Slugwroth, "Wake up and chase me." I'll admit it's odd, but it works.

I mentioned at the beginning of this chapter that I'd make the Slugwroth die by running into the door. The door does this in its `Hit()` function, as shown in [listing 18.4](#).

## Listing 18.4. The `door::Hit()` function

```

1 void door::Hit(world_object &stationary)
2 {
3     Open( &stationary);
4
5     // Slugwroths are weak!
6     if (stationary.Type() == "slug")
7     {
8         slug *theSlug = (slug*)&stationary;
9         theSlug>Kill();
10    }
11 }

```

The door can only hit something when the door itself is moving. That means the door is only partially open. Therefore, the door opens itself completely by calling its own `Open()` function on line 3. If the thing it hit is a Slugwroth, then the door tells the Slugwroth it's dead by calling the `slug::Kill()` function.

At this point, we're done adding features to Invasion of the Slugwroths. But before I end this book, let's examine how you would move forward with the game if you were actually developing it.



# Additions to the Game

If I were developing Invasion of the Slugwroths for commercial release, the next thing I'd add would be Captain Chloride's salt shooter so he can kill the Slugwroths himself instead of running them into doors. Of course, this would require a `salt_shooter` class. I would add a data member of type `salt_shooter` to the `chloride` class.

The `salt_shooter` class would need to contain member data for storing how many shots were left. It could have a variable indicating its maximum range. That way, I could add SuperPower SaltTabs in later levels. If Captain Chloride finds and picks up SuperPower SaltTabs, his salt shooter fires a greater distance for a specific number of shots.

To enable the Captain to carry his salt shooter as he walks, I would need to add two animations of him carrying it while walking one animation for walking left and the other for walking right. Firing the salt shooter requires two more animations: firing left and firing right.

Adding all of these animations requires techniques that you know already the same techniques I used to make him walk. Selecting the correct animation means adding more states to the `chloride` class. Recall that the `chloride` class contains an enumeration called `state`. The constants in that enumeration tell what state the Captain is in. If you look in the file `Chloride.h` in the folder `Source\Chapter18\Prog_18_01` on the CD, you'll see that there are constants for walking left and right. I would need to add constants for walking left with the salt shooter out, walking right with the salt shooter out, firing to the left, and firing to the right.

## Tip

When you simulate a projectile being fired over a short distance, like bullets or the salt blast from the salt shooter, you usually do not add the effects of gravity. Gravity does not change the course much of bullets and similar projectiles over short distances. Therefore, the effect can be ignored.

When the salt shooter fires, it works very much like the cannon example you first saw in [chapter 9](#), "Floating Point Math in C++." Instead of a cannonball, I

would create a class to represent a blast of salt. Unlike the `cannonball` class, the `salt_blast` class would not model the effects of gravity. The blast of salt would always continue in a straight line until it hit something. When it did, it would explode in a white poof. If it hit a Slugwroth, the `salt_blast` object would call the `slug::Kill()` function.

As you can see, adding the salt shooter is just a matter of applying the techniques you've already seen in the example programs in this book.

What about Captain Chloride's powerups? How would those work?

## Factoid

When a character in a game jumps, he behaves very much like a projectile that has just been launched in a gravity field. Therefore, the code for plotting his course is very similar to the code used for the cannonball in the cannon examples.

I think you've probably guessed that adding powerups is just a variation on the techniques we used to implement the door key. Your game characters can pick up virtually anything that way.

How about jumping? Wouldn't it be great if the Captain could jump?

Not a problem: That's just a matter of adding another pair of animations to his `animated_sprite`, adding a state constant for jumping left and one for jumping right, and writing a function to make him jump. Interestingly, the code in the Captain's `Jump()` function would be extremely similar to the code that made the cannonball move in the cannon examples. Any time you're modeling a projectile in a gravity field, it moves the way the cannonball did.

Another feature that might be nice is to have a hole for Captain Chloride to fall into. To do so, I would add the hole as a special type of nonmoving sprite. When the Captain's bounding rectangle bumps the bounding rectangle of the hole, the game plays an animation of him falling in.

Likewise, many games contain elevators and moving platforms. I could add that to Invasion of the Slugwroths by creating the elevator or platform as a type of sprite or animated sprite. As long as Captain Chloride is standing on top of the elevator or platform, he moves with it. How can the game tell? It

uses the bounding rectangles.

As you can see, now that you have the programming basics down there are a number of additions that you can make to this game or to a new game that you develop yourself. Of course, there are still aspects of game programming left to learn. And that's what we'll discuss next.

# Epilogue: Not the End

Although this is the end of the book, it's the beginning for you. You still have many skills to acquire as you grow into a professional game programmer. Now before I start into describing what those skills are, I want you to know that I've provided a list of books for recommended reading on the CD. Just put the CD into your CD/DVDROM drive and an HTML page pops up. On that page is a list of items. One of them discusses how to move forward by getting more information. If you click that link, it will take you to the list of suggested reading.

## Note

If the HTML page does not pop up when you insert the CD into your CD/DVDROM drive, click your Start button. Next, select Run and then type in `<d>:\AutorunPro.EXE`, where `<d>` is the letter of your CD/DVDROM drive.

First, you need to get a book on advanced C++ programming. You've received a good introduction to it in this book. However, there's always more to learn. C++ is your primary tool for developing games. A lot of your success as a game programmer is based on how well you know C++.

Next, learn more about Windows programming. Specifically, I strongly advise you to study both DirectX and OpenGL. Some game companies use DirectX and others use OpenGL. If you really want to be a pro, learn both.

Another skill you might want to study is the details of sound programming. Many professional game programmers know far more about graphics programming than sound programming. As a result, it's something of a sought-after specialty in the game programming world. Advanced sound programmers can blend and alter sounds in ways that boggle the mind.

In addition, you need to know about other methods of collision detection. It is possible to use bounding circles or ellipses instead of bounding rectangles. This is not a common technique. However, when you get to 3D programming, you'll have to learn how to use bounding cylinders. So if you start with bounding circles, you'll make things easier on yourself.

Also, you should examine the way that LlamaWorks2D blends bitmaps into the

background image while eliminating the transparent color. The formal name for this is [\*texture blending\*](#). The code that performs LlamaWorks2D's texture blending is in the files LW2DTexture.h and LW2DTexture.cpp.

There are many good books that talk about texture blending, some of which are in the list of suggested reading. Texture blending is a vital technique for game programmers. You use it to do sprite animation. It's also necessary for adding light areas and shadows to scenery. For example, if your character is walking through a dark area of a level and he turns on a light, you lighten that area with texture blending. Likewise, you can use texture blending to add darkness and shadows where appropriate. Texture blending accounts for a huge amount of the work involved in making the graphics in games look real.

As you're studying these topics, you can also study 3D programming. Programming in 3D is a huge and complex task. There are many, many books written about it. Virtually all of them assume that you're already a good C++ and Windows programmer. However, after reading this book, you have the skills you need to read one of those books and do well with it.

On a personal note, I hope you've enjoyed reading this book. Although it may sound corny, my goal in writing it was to help bright and enthusiastic minds get into a creative profession that they might not otherwise enter. Let's face it: the challenges to learning game programming are tough when you're just starting out. I hope you feel that game programming is something you can do for fun, and maybe for a career. I also hope you see that, for you, this is not...

...THE END

# Appendix. Glossary

## **alpha channel**

An extra 8 bits per pixel in a bitmap of extra information that specifies how transparent or opaque the pixel is.

## **array element**

One item in an array.

## **ASCII character set**

The set of characters specified for computer use by the American Standard Code for Information Interchange.

## **attribute**

Information in a software object that describes the thing the object represents.

## **back buffer**

The buffer containing the next image to display on the screen.

## **base address**

The starting address of an array or a block of memory.

## **base class**

A class that other classes inherit from.

## **binary**

Numbers in base 2. Binary numbers only use the digits 0 and 1.

## **bit**

A binary 0 or 1.

## **bitmap**

A collection of pixel values, usually stored in a file on a disk, that form a picture.

## **buffer**

A section of a computer's memory that is used for storing data.

## **bug**

A mistake in a program.

## **byte**

A group of 8 bits.

## **C++**

A programming language that resembles English. Most games are written in C++.

## **central processing unit (cpu)**

See [\*microprocessor\*](#).

## **child class**

See [\*derived class\*](#)

## **closing brace**

Another term for the right brace (}).

## **collision detection**

The process of testing to see if a sprite has hit something as it moves.

## **color depth**

The number of bits per pixel.



## **compiler**

A program that translates statements in programming languages such as C++ into binary.

## **compound logical expressions**

A logical expression that contains more than one condition.

## **condition**

A comparison in a logical expression.

## **constructor**

A special class member function that initializes an instance of the class to a known state.

## **debugger**

A software tool that helps find bugs.

## **default constructor**

A constructor with no parameters.

## **default parameter**

A function parameter that is set to a default value.

## **dereference**

Accessing the contents of the location a pointer points to.

## **derived class**

A class that inherits from another class.

## **destructor**

A special class member function that performs cleanup tasks on an instance of the class.

## **digitized sound**

Sound or music recorded in a digital format.

## **dot product**

The multiplying of two vectors to get a scalar.

## **dynamic music generation**

The technique of generating music as a game runs that is synchronized

with the action of the game.

## **exponent**

A power by which a number is raised.

## **frame**

One picture drawn on a screen in a series of pictures. The rapid display of frames produces the illusion of animation.

## **friend function**

A function that is not a member of the class in question but that has access to its private data.

## **front buffer**

The buffer containing the image that is currently displayed on the screen.

## **function body**

The statements or commands contained in a function between the opening and closing braces (the { and } symbols).

## **gain**

The volume of the sound.

## **game class**

A class that LlamaWorks2D uses to represent the game itself.

## **game engine**

Code that does the most common tasks performed by particular types of games. Each type of game requires its own game engine.

## **gigabyte**

A group of 1,000 megabytes.

## **graphics library**

A collection of functions that perform the most common and essential tasks all games need to do.

## **include statements**

C++ statements that begin with the `#include` directive, followed by the name of the include file. Use them to read .h files into .h or .cpp files.

## **index**

A subscript number of an array.

## **inline member functions**

Member functions whose code is defined inside the class itself.

## **kilobyte**

A group of 1,024 bytes.

## **linked list**

A list of objects in which each item in the list contains a pointer to the next item.

## **linker**

A program that converts object code into executable code.

## **literal string**

A string that is typed directly into program code and contained in quote marks.

## **logical expression**

A comparison that uses a logical operator and evaluates to the values `true` or `false`.

## **macro**

A special C++ marker you can define that enables the compiler to insert C++ statements into source code.

## **main() function**

The program entry point, which is another way of saying the point at which the program starts running.

## **megabyte**

A group of 1,000 kilobytes.

## **member data**

Data items in a class definition.

## **message map**

A technique for assigning specific functions to handle user input.

## **microprocessor**

The "brain" of a computer. Microprocessors really don't "understand" anything; they just execute binary instructions.

## **nameless temporary variables**

Variables that have no name and are created by calling a class's constructor that are immediately thrown away.

## **namespace**

A name given to a group of related types, functions, and so forth.

## **object code**

An intermediate form between source code and executable code. Object code is in binary.

## **object-oriented programming**

The definition of custom types that represent real or imaginary things in software.

## **offset**

A subscript number of an array.

## **opening brace**

Another term for the left brace (`{`).

## **out-of-line member functions**

Member functions whose code appears outside of a class definition.

## **ownership**

The responsibility for deleting dynamically allocated objects.

## **parent class**

See [base class](#).

## **parse**

The process of reading input and dividing it into meaningful tokens.

## **patches**

MIDI instrument sound definitions.

## **pixel**

A group of three phosphorus dots on a computer monitor. The group consists of one red dot, one green dot, and one blue dot.

## **pixel map**

See [bitmap](#)



## **pixmap**

See [bitmap](#)

## **postincrement**

An increment that occurs after the value in the variable being incremented has been used.

## **posttest loop**

A C++ looping statement, such as `do-while`, that performs its test at the end of the loop.

## **precision error**

An error caused by a floating-point variable not having enough significant digits to provide the required precision.

## **pretest loop**

A C++ looping statement, such as `while` or `for`, that performs its test at the beginning of the loop.

## **program**

A set of binary instructions that tell a computer how to do a task.

## **program entry point**

The starting point of a program. In C++, the `main()` function is the program entry point.

## **program stack**

A special area of memory programs used to store temporary variables in a function and values that are returned from a function.

## **programming language**

A language used to write computer programs. Most programming languages are easier to understand than binary.

## **prototype**

A statement that describes how to use a function. Specifically, it enables parameter and return type checking.

## **real-time input**

Input that reflects the current state of the input devices and is responded to faster than humans can respond.

## **Redbook audio**

CD-quality sound and music.

## **reference**

Passing or returning an actual variable, rather than a copy of the variable.

## **refresh rate**

The amount of time it takes for a color screen's group of three electron guns to hit every pixel on the screen, one after the other.

## **rendering**

The drawing or display of graphics on a computer screen.

## **resolution**

The number of pixels on a monitor.

## **save file**

A file containing the player's current progress in the game.

## **scalar**

An integer or floating-point number.

## **scan line**

A row of pixels on a screen.

## **scope**

The portion of a program over which a program element, such as a variable or function, can be seen.

## **significant digits**

The most important digits in a floating-point number. The digits in a floating-point number that most significantly affect the number's value.

## **software object**

A programmer-defined data type that represents something imaginary or real and that defines a set of operations programs can perform on variables of that type.

## **source code**

Statements written in a programming language such as C++.

## **source files**

Text files containing statements in a programming language such as C++.

## **stream**

A flow of data into or out of a program.

**subscript**

An array element number.

**terabyte**

A group of 1,000 gigabytes.

**text file**

A data file containing text.

**texture blending**

Blending two or more bitmaps together, possibly with transparency.

**token**

A meaningful symbol or value, such as an XML tag.

**variable**

A named spot in memory that holds data.

**video adapter**

A circuit card to which a computer's monitor is connected.

## **video mode**

The resolution and bit depth of a computer monitor.



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]





# Index

[**SYMBOL**] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

& (ampersand operator) 2nd 3rd 4th 5th

( ) parentheses 2nd 3rd

\* (asterisk) 2nd 3rd 4th

2.5D programming

2D arrays

2D games 2nd

2D graphics 2nd

2D programming 2nd

2D/3D points

2D/3D spaces

3D arrays

3D games

3D graphics 2nd 3rd

3D modeling program

3D programming 2nd

3DO game console

4D arrays

:(colon) 2nd 3rd

:: (C++ scope resolution operator) 2nd 3rd 4th

:(semicolon) 2nd

= (assignment operator)

[ ] square brackets 2nd 3rd 4th

\_ (underscore character)

{ } braces 2nd 3rd 4th

~ (tilde)



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

actions 2nd 3rd [See also [character actions](#); [object actions](#).]

addition operator

alpha channels 2nd

ampersand operator (&) 2nd 3rd 4th 5th

animated sprites 2nd 3rd 4th

animation [See also [actions](#); [frames](#).]

in games

objects

screen 2nd 3rd

AnInt ( ) functions

application object

approximation

arguments 2nd [See also [parameters](#).]

ArmCannon ( ) function

array elements 2nd

array numbering

arrays

2D

3D

4D

boundaries of

character 2nd 3rd

data types in

declaring

described

dynamically allocated

initializing

pointers and

program crashes and

string

using

art, placeholder

art skills [See also [graphics libraries.](#)]

ASCII character set 2nd 3rd

assertions

assignment operator (=) 2nd

asterisk (\*) 2nd 3rd 4th

attributes 2nd 3rd 4th

Audacity audio editor

audio [See [sound.](#) ]

audio editors 2nd



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

back buffer 2nd 3rd 4th 5th

background music

balls

bouncing

CannonShoot game

hitting 2nd

movement/direction 2nd

Ping game

stationary

velocity

band-in-a-box programs 2nd 3rd 4th

base address 2nd

base class functions

base class pointers

base classes 2nd 3rd 4th 5th

Bin folder 2nd

binary 2nd 3rd 4th

binary files

binary instructions 2nd 3rd

binary numbers 2nd 3rd

binary operators

bitmap (BMP) files 2nd 3rd 4th

bitmap (BMP) format 2nd

bitmap\_region type

bitmaps

file I/O and

inverting

loading 2nd 3rd 4th

overview

sharing

transparent 2nd

bits 2nd

**BMP** [See [bitmaps.](#) ]

bool data type

Boolean value

bosses 2nd

bounding circles/cylinders

bounding rectangles 2nd 3rd 4th 5th

braces { } 2nd 3rd 4th

brackets [ ] 2nd 3rd 4th

buffers 2nd 3rd 4th

bugs 2nd

bytes 2nd





# Index

[SYMBOL] [A] [B] [**C**] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

C++ instructions

C++ language 2nd 3rd 4th 5th 6th

C++ programming [See [programming.](#) ]

C++ programs [See [programs.](#) ]

C++ scope resolution operator (: :) 2nd 3rd 4th

C++ Standard Libraries 2nd 3rd 4th [See also [libraries.](#)]

camel notation

CannonShoot program

adding animated sprites to

cannon creation

cannonball creation

cannonshoot class

overview

source files for

Captain Chloride 2nd 3rd [See also [characters](#); [Invasion of the Slugwroths.](#)]

Cartesian coordinate system 2nd

case statement

CD-ROM drives 2nd 3rd 4th

CDs [See also [DVDs.](#)]

hard drive failure and

music from

programs on

resources on

central processing unit (CPU)

char data type

character actions [See also [objects.](#)]

carrying items

chasing/being chased 2nd 3rd 4th

falling

firing weapons 2nd 3rd 4th

jumping

killing 2nd 3rd 4th 5th

opening/closing doors 2nd

picking up items

character arrays 2nd 3rd

characters [See also [objects](#).]

actions [See [character actions](#). ]

bosses 2nd

Captain Chloride 2nd 3rd

Slugwroths 2nd

states 2nd

villains

characters, text 2nd

child classes [See also [derived classes](#).]

child objects

cin stream

class keyword

class members

class type

classes [See also [structures](#).]

base [See [base classes](#). ]

child

constructors

defining

derived [See [derived classes](#). ]

destructors

inheritance and

inline member functions

names 2nd

namespaces

parent

point2d

point3d

cleanup

games 2nd 3rd

levels 2nd 3rd

memory

close( ) function

code [See also [programs](#); [source code](#).]

compiling [See [compilers](#); [compiling programs](#). ]

continuation arrows

exceeding character limit

executable

game 2nd 3rd 4th 5th 6th

line length

object 2nd 3rd

reusing [See [inheritance](#). ]

collision detection 2nd 3rd 4th

collision management

collisions 2nd

colon (:) 2nd 3rd

color

background 2nd

in games 2nd 3rd 4th

monitor 2nd 3rd

transparent 2nd

color depth 2nd

color values 2nd

Comdex

comments

compilers

commercial

compatibility

Dev-C++ 2nd 3rd 4th 5th 6th

errors 2nd

member functions and

overview 2nd

compiling programs 2nd 3rd

compound logical expressions

compression 2nd

computer

monitor [See [monitor.](#) ]

requirements for

conditional operator

conditions

described

logical operators and 2nd

testing

true/false

configuration files 2nd

console programs 2nd

const statement

constants 2nd 3rd 4th

constructors 2nd 3rd [See also *specific constructors.*]

coordinate systems 2nd

coordinates

gamespace 2nd

integer screen

screen space 2nd

world 2nd

x/y

cout stream 2nd 3rd

.cpp files 2nd 3rd 4th

CPU (central processing unit)

Creative WaveStudio

CRT screen



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

data [See also [data members](#); [member data](#).]

controlling access to

private 2nd

public 2nd 3rd 4th

in software objects

valid state

data members 2nd 3rd 4th 5th [See also [member data](#).]

data types

in arrays

common

considerations 2nd

creation of

defining

described 2nd

operation performed

variables and

debuggers 2nd 3rd 4th 5th

debugging 2nd

decimal points 2nd 3rd 4th

decrement operator

#define statement

definitions 2nd 3rd

delete keyword

delete operator

delimiter character

dereference 2nd

derived classes



calling functions in  
inheritance and 2nd  
overriding base class functions in  
overview 2nd  
protected member data access  
specializations

derived objects

Descent

designing games [See [game design/creation.](#) ]

destructors 2nd

Dev-C++ 2nd 3rd 4th

Dev-C++ compiler 2nd 3rd 4th 5th 6th

digitized music 2nd 3rd

digitized sound 2nd

Direct3D library 2nd

DirectMusic

DirectSound

DirectX Audio 2nd 3rd

DirectX Graphics library [See [Direct3D library.](#) ]

DirectX Software Development Kit(SDK)

display [See [monitor.](#) ]

division operator

DoGameOver ( ) function

DoLevelDone ( ) function 2nd

Doom

doors, opening/closing 2nd

dot product 2nd

double data type

do-while loops

drawing objects

DVD-ROM drives 2nd 3rd

DVDs 2nd 3rd 4th [See also [CDs.](#)]

dynamic memory allocation

dynamic music generation



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Easter eggs

Elder Scrolls, The

electron guns 2nd

electrons 2nd

endline 2nd 3rd 4th

enum statement

enumerated type 2nd

enumerations

equations 2nd 3rd 4th

error messages 2nd 3rd 4th

error values

errors [See also [debugging](#); [troubleshooting](#).]

    compiler 2nd

    lost block of memory

    precision

    rounding

exceptions 2nd

executable code

executable files

EXIT\_SUCCESS value

exponent 2nd 3rd

expressions 2nd 3rd

extensions 2nd

extraction operator 2nd



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

[false condition](#)

[file extensions 2nd](#)

[file formats 2nd](#)

[file I/O](#)

[file structure](#)

[files](#)

[binary](#)

[bitmap \[See \[bitmap \\(BMP\\) files.\]\(#\) \]](#)

[configuration 2nd](#)

[.cpp 2nd 3rd 4th](#)

[executable](#)

[level 2nd 3rd 4th](#)

[MIDI 2nd](#)

[MP3 2nd 3rd 4th 5th](#)

[music](#)

[.obj](#)

[project 2nd](#)

[save 2nd 3rd](#)

[sound 2nd 3rd 4th](#)

[text](#)

[TGA](#)

[XML 2nd 3rd](#)

[FireCannon \( \) function](#)

[first-person shooters 2nd 3rd 4th](#)

[float data type](#)

[floating-point numbers](#)

[case study](#)

color values

data in

described

fractional parts 2nd 3rd

gamespaces

overview

precision errors

rounding errors

vector classes for

focus testing

formats,file 2nd

fractions

frames [See also [animation.](#)]

drawing 2nd

overview 2nd

rendering 2nd 3rd

sprites and

updating 2nd 3rd

friend functions 2nd 3rd

front buffer 2nd 3rd

function body

functions [See also *specific functions.*]

body of

case 2nd

creating

member [See [member functions.](#) ]

names 2nd 3rd 4th

operator-equals

overloading

parameter list 2nd

passing values to

protected

prototypes

public

required elements

returning values from

scope in





# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

gain 2nd

Gain ( ) function

game classes

definition of

described 2nd 3rd

inheriting

name of

OnAppLoad( ) function

writing

game code 2nd 3rd 4th 5th 6th

game design skills

game design/creation

artistic aspect of

books on

final steps for

game tasks

plausible situations

playability

program structure

sound/music

uniqueness

game engines 2nd

game initialization 2nd 3rd

game players [See [players.](#) ]

game programmers

game programming [See [programming.](#) ]

game tasks

GameOver ( ) function 2nd  
gameplay

importance of

training player during

games

2D 2nd

3D

animating

characters in [See [characters.](#) ]

cleaning up 2nd 3rd

crashes 2nd

creating [See [game design/creation.](#) ]

described

emotion and

exploration 2nd

finishing

goal of

hooks 2nd 3rd 4th

initializing 2nd 3rd

marketability of

mastery of

objects in [See [objects.](#) ]

platform

problem-solving in

scores 2nd 3rd 4th 5th

side-scrolling 2nd 3rd

tips for

visual style

gamespace coordinates 2nd 3rd

gamespaces

GarageBand.com

GCC (GNU Compiler Collection)

Get F ( ) function

[Get I \( \) function](#)

[GetKeyboardInput \( \) function 2nd 3rd 4th](#)

[getline \( \) function](#)

[GIF format](#)

[gigabytes 2nd](#)

[GIMP 2nd](#)

[global namespace](#)

[GNU Compiler Collection \(GCC\)](#)

[Gnu Image Manipulation Program\(GIMP\)](#)

[graphics 2nd 3rd 4th](#)

[graphics cards](#)

[graphics libraries 2nd 3rd](#)

[graphics modes](#) [See also [video modes.](#)]

[gravity](#)

[Gun Metal](#)

[guns](#) [See [weapons.](#) ]



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

.h file 2nd

Halo 2nd

heap

hexadecimal address numbers

Hit ( ) function

hooks 2nd 3rd 4th

Hungarian notation



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

IDEs (integrated development environments) 2nd

if-else statements

if statements 2nd 3rd

if-else statements 2nd

ifstream constructor

implementations

include statements 2nd 3rd 4th

increment operator 2nd

index numbers 2nd 3rd 4th

inheritance

advantages of

cannonball class

customizing games with

derived classes 2nd

in LlamaWorks2D

overriding base class functions

overview

pointers and

protected members

inheritance diagrams

InitApp ( ) function

InitGame ( ) function 2nd 3rd 4th 5th

initialization

arrays

games 2nd 3rd

levels 2nd

object

programs 2nd

InitLevel ( ) function

inline member functions 2nd 3rd

input

high-speed

immediate mode

keyboard 2nd 3rd

real-time

user 2nd 3rd 4th 5th

insertion operator 2nd 3rd

instructions

binary 2nd 3rd

C++

program

int data type

int statement

integer screen coordinates

integer variables 2nd

integers

color values

decimal points in 2nd 3rd

described 2nd

operations performed on

pointers to 2nd

integrated development environments (IDEs) 2nd

Intersects ( ) function

Invasion of the Slugwroths

Captain Chloride 2nd 3rd

design of

implementation of

Slugwroths 2nd

solid objects

iostream file

IsKeyPressed ( ) function

is\_open ( ) function







# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

JPG (JPEG) format 2nd  
jump actions



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

key codes 2nd

keyboard input 2nd 3rd

keywords 2nd [See also *specific keywords.*]

killing characters 2nd 3rd 4th 5th

kilobytes 2nd



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

level files 2nd 3rd 4th

LevelDone ( ) function

levels

cleaning up 2nd 3rd

hidden

initialization 2nd

LlamaWorks2D

loading

rendering

updating

worlds

libraries

C++ Standard Libraries 2nd 3rd 4th

graphics 2nd 3rd 4th

linking to

sound 2nd 3rd

light

line numbers

linked list 2nd

linkers 2nd

linking programs 2nd

literal strings 2nd

LlamaWorks2D

animating objects

compiling programs

components of

drawing objects

formats supported

high-speed input

immediate mode input

levels

macros in

namespace

obtaining

OpenGL

overview

pointers and

project configuration

project creation

running programs 2nd

sound effects in

sprites in

structures

texture blending 2nd

Windows and 2nd

Windows messages

writing game class

LlamaWorks2D.h file

LoadWAV ( ) function 2nd

logical expressions 2nd

logical operators

long data type

loop counters 2nd

looping animation

looping music 2nd

loops

do-while

message

overview

pretest/posttest



while

LW2DApp.cpp file

LW2DLevel.h file



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

macros

described 2nd

in LlamaWorks2D 2nd

magnitude 2nd 3rd

Magnitude ( ) function

MagnitudeSquared ( ) function

main ( ) function

modifying

overview 2nd

in Windows programs

mastery hooks 2nd

math operators

megabytes 2nd

member data [See also [data members.](#)]

controlling access to

outputting

overview 2nd

private 2nd

public 2nd

valid state

vs. data members

member functions

calling

creating

defining

inline 2nd

out-of-line 2nd

overview

prototypes 2nd

scope resolution operator

using

memory

allocating 2nd

array boundaries and

buffers 2nd 3rd 4th

cleanup

dynamic

freeing

heap

high-resolution modes and

leaks

orphaning

static

memory addresses 2nd 3rd

memory blocks 2nd 3rd

message handling 2nd

message loop

message maps 2nd 3rd

message processing

MessageBox ( ) function

messages, error 2nd 3rd 4th

microprocessor 2nd 3rd

**Microsoft DirectX [See DirectX. ]**

Microsoft Hungarian notation 2nd

Microsoft Visual C++

MIDI audio

MIDI commands

MIDI files 2nd

MIDI instruments

monitor [See also [screen.](#)]

color displayed on 2nd 3rd

components of  
displaying pictures on  
plasma  
resolution 2nd 3rd  
video modes

Move ( ) function

Movement ( ) function

MP3 files 2nd 3rd 4th 5th

multiplication operator

music [See also [sound.](#)]

background

band-in-a-box programs 2nd 3rd 4th

creating

digitized 2nd 3rd

dynamic generation

finding

from CD/DVD 2nd

importance of 2nd 3rd

loading

looping 2nd

playing when player wins

recording

rewinding

skills required

stopping

volume control

music files

music generation program

music/sound skills

Myst



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

nameless temporary variables

namespaces

classes and

global

LlamaWorks2D 2nd

scope resolution and

standard 2nd

new keyword

new keyword

newline character (\n)

Normalize ( ) function

NULL value 2nd 3rd

numbers

binary 2nd 3rd

calculating square of

decimal points in 2nd 3rd 4th

exponent 2nd 3rd

floating-point [See [floating-point numbers.](#) ]

fractional

index 2nd 3rd 4th

line

metric system vs. English units

offset 2nd

random 2nd

rounding

significant digits 2nd 3rd

vs. constants







# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

.o files

.obj files

object actions

bumping/being bumped

elevators/moving platforms

opening/closing doors 2nd

object code 2nd 3rd

object code modules

object-oriented programming [See [OOP \(object-oriented programming\)](#). ]

objects [See also [characters](#).]

animating

balls [See [balls](#). ]

child

creating 2nd

custom

declaring same type

defining

derived

designing

drawing

initializing

operations performed on 2nd

ownership

parent

projectiles 2nd

rendering

reusable

software 2nd 3rd

solid

sprite 2nd

string

weapons [See [weapons.](#) ]

offset numbers 2nd

ofstream constructor

OnAppLoad ( ) function 2nd

OnKeyDown ( ) function 2nd

OOP (object-oriented programming)

classes

constructors

described 2nd

destructors

if-else statement

inline member functions

logical operators

member data [See [member data.](#) ]

member functions [See [member functions.](#) ]

namespaces

scope resolution

software objects

structures

open ( ) function

OpenAL (Open Audio Library) 2nd 3rd

OpenGL (Open Graphics Library) 2nd 3rd

operations [See also [actions.](#)]

operator-equals operators

operators [See also *specific operators.*]

binary

decrement

increment

logical

math

object-oriented [See [OOP \(object-oriented programming\)](#). ]

overloading 2nd 3rd

overloading

constructors 2nd

functions

operators 2nd 3rd

purpose of 2nd

vector classes 2nd

overview

ownership 2nd



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

paint programs

parabolic equations 2nd

parameter list 2nd 3rd

parameters [See also [arguments.](#)]

default 2nd

described

inheritance and

multiple

names

overloaded constructors

passing

type

vs. variables

parent class objects

parent classes [See also [base classes.](#)]

parentheses ( ) 2nd 3rd

parsing 2nd

patches

PAUSE command

pauses, in programs 2nd 3rd 4th 5th 6th

Persistence of Vision (POV) Paytracer

phosphorus

Ping game

adding music/sound

balls 2nd

cleanup

described

files for

game over

handling messages 2nd

initialization 2nd

keyboard input

object creation

paddles

playing again

rendering frames

score markers

updating frames

Windows messages

writing

pixel maps [See also [bitmaps.](#)]

pixels 2nd 3rd 4th

pixmap [See also [bitmaps;](#) .]

placeholder art

plasma displays

platform games

Play ( ) function 2nd

players

game performance and 2nd

holding interest of 2nd 3rd

hooks 2nd 3rd 4th

power-ups

scores 2nd 3rd 4th 5th

sound/music and 2nd

training

winning game

point2d class

point3d class

pointer variables 2nd

pointers

arrays and  
base classes  
bitmaps and  
declaring  
dereferencing  
described 2nd 3rd  
Direct X and  
importance of 2nd  
incrementing  
inheritance and  
learning about  
LlamaWorks2D and  
memory allocation and 2nd  
OpenGL and  
sound libraries and  
temp  
to characters  
using

postincrement  
postincrement operator  
posttest loop 2nd  
power-ups 2nd  
precision error  
pretest loop 2nd  
Print ( ) function  
Pro tools program  
program entry point 2nd  
program stack 2nd  
program structure  
programmers  
programming  
2.5D  
2D 2nd  
3D 2nd



game

learning about

overview

project file creation

resources for

skills required for

sound

styles

Windows 2nd

programming languages 2nd 3rd 4th 5th

programs [See also [code](#); [programming](#).]

analyzing line by line

camel notation

case considerations

on CD

comments in 2nd

compiling [See [compilers](#); [compiling programs](#). ]

console 2nd

crashes 2nd

described 2nd 3rd

initializing 2nd

instructions in

linking 2nd

names

pauses in 2nd 3rd 4th 5th 6th

problems with [See [troubleshooting](#). ]

quitting 2nd 3rd 4th 5th

robustness of code

running 2nd 3rd 4th 5th

project files 2nd

projectiles 2nd

projects

configuring

creating 2nd 3rd

protected keyword

prototypes

creating/playing

member functions 2nd

overloading operators 2nd

overview 2nd

public data members 2nd

public keyword



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

quitting programs 2nd 3rd 4th 5th  
quote marks



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

[rand \( \) function](#) 2nd

[read \( \) function](#)

[real-time input](#)

[Redbook audio](#) 2nd

[reference](#) 2nd 3rd 4th

[refresh rate](#) 2nd 3rd

[Render \( \) function](#) 2nd 3rd 4th 5th

[RenderFrame \( \) function](#) 2nd 3rd 4th

[rendering](#)

[described](#) 2nd

[frames](#) 2nd 3rd

[high-end \(volumetric\)](#)

[levels](#)

[objects in LlamaWorks2D](#)

[resolution](#)

[described](#)

[high-resolution mode](#)

[memory and](#)

[monitor](#) 2nd 3rd

[refresh rate](#)

[screen](#) 2nd 3rd

[return statement](#)

[return type](#) 2nd

[rounding errors](#)

[running programs](#) 2nd 3rd 4th 5th



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

salt shooter 2nd 3rd

save files 2nd 3rd

scalars 2nd

scan lines 2nd 3rd

scope 2nd

scope resolution operator (: :) 2nd 3rd 4th

score markers 2nd

score, player 2nd 3rd 4th 5th

screen [See also [monitor](#).]

outputting to

refresh rate

resolution 2nd 3rd

screen animation 2nd 3rd

screen space coordinates 2nd

screen\_object pointer

semicolon (;) 2nd

SetI ( ) function

shadows

shooters 2nd 3rd 4th 5th

shooting weapons 2nd 3rd 4th

short data type

side-scrolling games 2nd 3rd

significant digits 2nd 3rd

Sims, The

sizeof ( ) operator

slash marks (/ /)

Sleep ( ) function

Slugwroths 2nd [See also [characters](#); [Invasion of the Slugwroths](#).]

software objects 2nd 3rd

sound [See also [music](#); [sound effects](#).]



audio editors 2nd

compression

digitized 2nd 3rd

from CD/DVD 2nd

gain 2nd

importance of

loading

MIDI

mixing with OpenAL

MP3 files 2nd 3rd 4th 5th

overview

playing

recording

sound cards 2nd 3rd 4th

sound class 2nd

sound effect collections

sound effect generator programs

sound effects [See also [music.](#)]

creating

DirectX Audio 2nd 3rd

finding

importance of

in LlamaWorks2D

loading sounds

OpenAL 2nd 3rd

playing sounds

software/tools for

volume control

WAV files 2nd

sound files 2nd 3rd 4th

sound libraries 2nd 3rd

sound programming

sound/music skills

source code [See also [code](#); [object code](#).]

compatibility of

compiling [See [compilers](#); [compiling programs](#). ]

described 2nd

line numbers in

LlamaWorks2D

macros for 2nd

translating into binary

source files 2nd

space

variables and

white 2nd 3rd

sprite animation 2nd 3rd 4th

sprite class 2nd 3rd

sprite objects 2nd

sprites

animation frames and

bouncing

collision detection

described

in LlamaWorks2D

setting direction/speed of

sqrt ( ) function

square brackets [ ] 2nd 3rd 4th

squared ( ) function

rand ( ) function

statements [See also *specific statements*.]

described 2nd

executing 2nd 3rd

translating into binary commands

states 2nd 3rd

static keyword

static memory allocation

std namespace 2nd

stdlib (standard) libraries 2nd

stdlib.h file

Stop ( ) function

strcpy ( ) function

streams

cin

cout 2nd 3rd

described

for files

include statements and

string arrays

string object

string type

strings

data in

described

literal 2nd

structures [See also [classes.](#)]

subscript 2nd

subtraction operator

Super Mario games 2nd 3rd

switch statement

system ( ) function



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Tagged Image File (TIF) format 2nd

temp pointer

terabytes 2nd

Tetris

text files 2nd 3rd 4th

texture blending 2nd

TGA files

theApp. object

theAPP.InitApp ( ) function

TIF (Tagged Image File) format 2nd

tilde (~)

tokens 2nd

transparent color 2nd

troubleshooting [See also [debugging](#); [rounding errors](#).]

crashes 2nd

jerky movements

memory problems

slowness of programs

true condition 2nd

Type ( ) function

type casting

type functions

type, declaring objects of same



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

uncomments

underscore character ( \_ )

unit vectors

unsigned data type

unsigned integer

UpdateFrame ( ) function 2nd

user input 2nd 3rd 4th 5th

using keyword





# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

## values

color 2nd

error

passing

returning 0 value

returning from functions

## variables

data types and

declaring 2nd 3rd

decrementing

described

incrementing

keywords and

nameless temporary 2nd

names 2nd

"out of scope,"

overview

pointer 2nd

scope of

storing addresses in

vs. parameters

## vector classes

floating-point

overloading 2nd

## vector constructors

## vector variable

vectors

adding

direction

division of

dot product of

magnitude 2nd 3rd

multiplying

point location

subtracting

unit

velocity 2nd 3rd

video adapters 2nd 3rd 4th

video modes 2nd 3rd [See also [graphics modes.](#)]

villains

virtual key code

virtual keyword

Visual C++

volumetric rendering



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

WarCraft

WAV files

weapons

cannons [See [CannonShoot program.](#) ]

electron guns 2nd

firing 2nd 3rd 4th

salt shooter 2nd 3rd

while loops

white space 2nd 3rd

window class

Windows messages 2nd 3rd

Windows Paint

Windows programming 2nd

Windows Recorder 2nd 3rd

Windows version of GCC

Windows-style coordinate system

WinMain () function

WMF (Windows Metafile) format

WM\_KEYDOWN message

world coordinates 2nd

worlds 2nd

write ( ) function



# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

X ( ) function 2nd 3rd

XML level files 2nd 3rd

XML tags 2nd 3rd 4th 5th

x/y components

x/y coordinates



# Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#)  
[\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Y\]](#) [\[Z\]](#)

[Y \( \) function 2nd](#)  
[Yoshi's Story](#)





# Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q]  
[R] [S] [T] [U] [V] [W] [X] [Y] [Z]

z coordinate