C++游戏编程入门
(第3版)

[英] 约翰·霍顿(John Horton) 著
王志强 王远鹏 译

清華大學出版社

北 京

北京市版权局著作权合同登记号　图字：01-2024-5232

图书在版编目(CIP)数据

C++游戏编程入门：第3版 /(英) 约翰·霍顿(John Horton) 著；王志强, 王远鹏译.
北京：清华大学出版社, 2025. 6. -- ISBN 978-7-302-69399-4

I. TP317.6

中国国家版本馆 CIP 数据核字第 2025C507C1 号

责任编辑：王　军
封面设计：高娟妮
版式设计：恒复文化
责任校对：成凤进
责任印制：宋　林

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦A座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：小森印刷（天津）有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：28　　**字　　数**：753千字
版　　次：2025年7月第1版　　**印　　次**：2025年7月第1次印刷
定　　价：128.00元

产品编号：109389-01

本书贡献者

关于作者

John Horton 是英国的一位程序及游戏发烧友。

谨以此书献给 Ray 与 Barry 两位兄弟，感谢他们的指引、示范与支持。

——John Horton

关于审阅者

Yoan Rock 虽然只有 26 岁，但已拥有 4 年游戏行业的从业经验。Yoan 具有 C++软件工程方面的背景，对 C++游戏产业有独到的见解，尤其是在 Unreal Engine 的使用以及通过设计图(blueprint)创建沉浸式体验方面。

Yoan 在 Limbic Studio 工作期间主要参与 Park Beyond 的研发工作，这是一款 AAA 级游戏，玩家可以在其中创建并管理自己的主题公园。他专精于玩法开发与缺陷修复，以及提升团队成员间的交流体验。

Yoan 随后与 Chillchat 工作室就 Primorden 达成合作，后者是基于 Unreal Engine 5 和 Gameplay Ability System 的一个多玩家项目。在此项目中，Yoan 在实现游戏机制、怪兽能力及 AI 行为树等方面做出了重要贡献。

Yoan 在 Game Atelier 的一个内部项目中领导 UI 开发，这充分体现出他在使用 Unreal Engine 5.3、Common UI 与 UMG 等工具创建沉浸式玩家体验的丰富经验。

目前，Yoan 在 Blacksheep 参与开发一个令人激动的大型项目。他勇于创新，时刻把握着产业的趋势，并探索利用 Unreal Engine 5.3 开发个人项目的方法。

前　言

你是否一直梦想着去创建自己的游戏？在第 3 版《C++游戏编程入门》的帮助下，你便能够实现这个梦想！这本初学者教程经过了修订，展示最新的 **VS 2022**、**SFML** 以及现代 **C++20** 编程技术，其中我们将构建 Timber!!!、Pong、Zombie Arena 与 Run 这 4 个复杂度递增的游戏，从而引导你踏上妙趣横生的游戏编程入门之旅。

本书始于对编程基础的讨论，你将学习 C++的若干重点主题，如 **OOP**(Object-Oriented Programming，**面向对象编程**)与 C++指针等，并逐渐熟悉 **STL**(Standard Template Library，**标准模板库**)的用法。本书随后将通过构建 Pong 游戏来介绍碰撞检测技巧等游戏物理学知识。在构建游戏的过程中，你同样会学到顶点数组、方向性(空间化)音效、OpenGL 可编程着色器、对象创建等技术，这些都是非常有用的游戏编程概念。同时，你还将能够深入挖掘游戏机制，并实现输入处理、角色升级等过程与简单的敌方 AI。最后，你将探索一些游戏设计模式来强化游戏编程技巧。

读罢本书，你将能掌握自主创建炫酷游戏所需要的全部知识。

本书读者对象

如果你没有 C++编程经验，并因此需要一本针对初学者的启蒙教程，或者希望学习如何构建游戏，或者仅仅将游戏编程视为一种学习 C++的手段，那么本书就非常适合你。

无论是有意发布一款游戏(例如，在 Steam 上)，还是仅仅希望用自己偷偷努力的成果惊艳友人，你都能从本书中获益。

本书各章的主要内容

第 1 章，“欢迎阅读《C++游戏编程入门》(第 3 版)”：本章概述我们将要踏上的、使用 OpenGL 驱动的 **SFML** 库和 C++为 PC 编写精彩游戏的旅程。本书已全面升级至第 3 版，在深度和广度上进行了显著提升与扩展。新版内容丰富，涵盖了从 C++基础(如变量、循环)到面向对象编程、**标准模板库**、SFML 特性，乃至 C++的新功能等知识。完成这段学习之旅后，你不仅将掌握 4 款可玩性高的游戏的制作技巧，还将奠定深厚而坚实的 C++基础。

第 2 章，“变量、运算符与决策——让精灵动起来”：本章将完成许多绘制任务，为背景图上的云朵以及前景中的蜜蜂分别赋予随机移动的能力(包括高度随机与速度随机)，而这需要用到更多 C++知识。我们将学习如何利用变量来存储数据，也会学习如何通过运算符来操作这些变量，以及如何根据变量的值来制定决策，有选择地执行若干分支路径中的一种。所有这些知识与本章所介绍的 SFML `Sprite` 与 `Texture` 这两个类的信息相结合，便能让我们实现云朵和蜜蜂的动画效果。

第 3 章，“C++字符串、SFML 时间、玩家输入与 HUD”：本章将用一半的篇幅来介绍文本操

作以及在屏幕上显示文本的方法，另一半篇幅则针对计时功能，学习通过使用更形象的时间棒来向玩家提醒剩余时间并制造紧迫感。

第 4 章，“循环、数组、switch、枚举与函数——实现游戏机制”：与本书其他各章相比，本章所含的 C++信息应该是最多的。这些信息被组织为一些基础的概念，并进而大大拓展我们对这门语言的理解。此外，本章同样将详述函数、游戏循环以及循环结构等之前被有意略去的一些模糊内容。

第 5 章，“碰撞、音效及终止条件：让游戏能玩起来”：这是我们首个游戏项目的最后一章，结束后便将得到自己的第一个完整游戏 Timber!!!。而真正玩起来之后，别忘了阅读本章最后一节，该节提供了对游戏的一些改进意见。具体而言，本章涵盖以下内容：添加剩余精灵(即 Sprite 对象)、处理玩家输入、让木料飞起来、处理角色之死、增加音效与其他功能、改进 Timber!!!。

第 6 章，“面向对象编程——开启 Pong 游戏”：本章将简要介绍一些有关 OOP(即面向对象编程)的理论知识，这些理论是我们开始应用 OOP 的基石。OOP 有助于我们将代码组织成人类可识别的结构，并有效控制代码的复杂性。我们不会让理论束之高阁，而是会立即将其应用于实践，通过开发一个 Pong 游戏来展示 OOP 的作用。我们将深入探索如何在 C++中创建可用作对象的新类型，并通过编写我们的第一个类来实现这一过程。本章首先将介绍一个简化的 Pong 游戏场景，以学习类的基础知识，随后将运用所学的知识，从头开始编写一个真正的 Pong 游戏，将理论转化为实践。

第 7 章，“AABB 碰撞检测与物理学——完成 Pong 游戏”：本章将编写第二个类。在这个过程中，我们体会到，虽然球明显异于球拍，但可以使用相同的技术将球的外形以及功能封装在 `Ball` 类中，这正是球与 `Bat` 类之间的关系。随后我们为碰撞检测与记分功能编程，从而完成 Pong 游戏的收尾工作。虽然这两种功能听起来有些复杂，但按照之前的趋势，使用 SFML 将大大简化其实现过程。

第 8 章，“SFML View 类——开启僵尸射手游戏”：这个项目会令我们更频繁地践行 **OOP** 思想，初步体会其强大效果。我们也将探索 SFML 中的一个多用途的 `View` 类，该类允许我们将游戏按照不同的视角分层。在 Zombie Arena 项目中，我们会把 HUD 与主游戏各划为一层，而之所以这样，是因为玩家每清空一批僵尸后都会拓展游戏世界，最终会让游戏世界远大于屏幕，玩家因而需要滚动摄像头。借助于 **`View`** 类，我们能让 HUD 文本不与背景一同滚动。

第 9 章，“C++引用、精灵表单与顶点数组”：我们曾在第 4 章介绍过作用域的概念，如果变量定义在函数中或某内层区块中，则此变量的作用域仅限于该函数或区块内部(换句话说，仅在该函数或区块内可见/使用)。在目前所学的 C++知识范围内，这可能带来问题，例如，我们可能无法处理 `main` 函数所需要的复杂对象，毕竟强行实现意味着全部代码均应位于 `main` 函数中。

本章将探索的 C++**引用**允许在变量或对象的作用域之外对其进行操作。此外，引用同样有助于避免在函数间直接传递大对象，由于这种传递每次均需要创建变量或对象的副本，因此非常缓慢。

掌握了引用这项新技能之后，我们会学习 SFML 中的 **`VertexArray`** 类，该类允许使用图像文件内的多个图片单元来高效地构建大型图像。本章结束时，通过引用机制以及一个 `VertexArray` 对象便可构建可缩放、可滚动的随机背景图片。

第 10 章，“指针、标准模板库与纹理管理初探”：本章将介绍许多知识，并完成游戏中的大量内容。首先，我们会学习**指针**这一基本 C++主题，这是保存内存地址的一种变量，且通常会保存另一变量的内存地址。虽然这听起来与引用类似，但后文将说明指针的功能更加强大，并会实际

使用指针来处理规模持续扩张的僵尸群。

我们还将学习**标准模板库**，其中整合了许多类，能够用来轻松实现一些常见的数据管理技术。

第 11 章，“编写 TextureHolder 类并构建僵尸群”：至此，我们所理解的 STL 基础知识足以管理游戏所需的一切纹理资源，毕竟不必为上千僵尸而反复给 GPU 加载图片。

随后，我们将进一步钻研 OOP 思想并使用静态函数。静态函数虽然也属于某个类，但在调用时不需要借助于该类的具体实例。同时，我们还会学习如何设计类，以令其仅存在一个实例，这种技巧非常适合确保某实例在程序不同位置上使用相同的内部数据。

第 12 章，“碰撞检测、拾取包与子弹”：现在，我们已经实现了游戏的主要视觉内容，让玩家能够控制角色在竞技场中跑动，其中充斥着正在追逐他的僵尸。但现在的问题是，其中没有任何交互，玩家能够直接穿越僵尸而毫发无损。为此，我们需要在僵尸与玩家之间进行碰撞检测。

另一方面，如果僵尸能够伤害并最终杀死玩家，那么为保持公平性，我们需要为玩家手中的枪械提供子弹，并保证子弹能够击中并杀死僵尸。

此外，由于本章需要实现子弹、僵尸与玩家三者之间的碰撞检测，因此同样需要将医疗包与弹药包抽象为类。

以上都是本章的任务，具体而言，包括子弹射击、增加准星、隐藏鼠标指针、创建拾取包以及碰撞检测。

第 13 章，“借助分层视图实现 HUD”：本章将揭示 SFML `View` 类的实际效果。我们会增加一组 SFML `Text` 对象，并参照 Timber!!!与 Pong 这两个项目来操作它们。此外，本章还将引入第二个 `View` 实例来绘制 HUD，这样，无论背景、玩家、僵尸或其他游戏对象如何行止，视角如何移动，HUD 均将作为所有游戏动作的最顶层而出现。

第 14 章，“音效、文件 I/O 操作与完成游戏”：行文至此，本游戏项目即将完成。当前这个短章将演示如何使用 C++标准库来简单地操作硬盘上的文件，也会介绍为游戏添加音效的方法。当然，我们知道如何添加音效，但本章将详细介绍 `play` 函数在代码中的具体位置。随后在为游戏完成一些辅助性功能后，本游戏便大功告成。具体而言，本章将介绍通过文件输入与文件输出操作来加载并保存高分纪录、添加音效、允许玩家升级、创建下一波僵尸等操作。

第 15 章，“Run!”：欢迎来到最终的项目。Run 是一个无限跑酷游戏，其中玩家脚下的平台会从后向前逐一消失，玩家需要持续向前跑动以避免被追上。我们将学习更多游戏编程技术，而这需要我们进一步学习更多 C++知识才能逐一实现。相比于之前三个项目，也许这个游戏最显著的特点在于其大大强化了面向对象理念。该游戏将使用的类远多于之前的游戏，只是其中大多数类并不复杂，代码也不长。此外，我们将把游戏内部所有对象的功能与外观封装为类，从而在改动对象时维持游戏循环本体不发生变化。我们很快便能意识到这种设计的强大之处：只需设计出描述所需游戏实体的行为与外观的独立组件(类)，便能创建出迥然不同的游戏。这也意味着你在自主设计游戏时完全可以采用这种代码结构。即便如此，这也不是这种设计思路的全部优势，还有更多的细节有待探索。

第 16 章，“声音、游戏逻辑、对象间通信与玩家”：本章将快速实现本游戏的声音效果。之前已经做过类似的工作，所以这不算难，而且通过仅仅几行代码便能为项目添加音乐背景。本项目后面将添加方向性(空间化)音效。

本章负责把与声音有关的全部代码封装为 `SoundEngine` 类。我们在实现声音效果后便转而实现玩家，而且只需要分别扩展 `Update` 类与 `Graphics` 类得到两个新的类结构，便能实现整个玩家角色的功能。我们之后为完成本游戏的全部工作，也基本上是通过扩展既有类来创建新的

游戏对象。此外，我们还将介绍通过指针来进行对象间通信的一种简单方法。

第 17 章，“图像、摄像机与动作”：我们有必要深入讨论本项目的图像机制。本章将编写负责绘制工作的摄像机类，所以同样适合讨论图像。打开 `graphics` 文件夹便可发现，其中只有一个图片文件。此外，我们目前完全没有调用过 `window.draw` 函数。这里我们将讨论为什么需要尽量避免调用它，并转而实现代替我们完成这项工作的 `Camera` 结构。本章结束时，我们将能运行游戏并亲身体会摄像机的效果，其中包括主视图、雷达视图与计时器文本。

第 18 章，“编写平台、玩家动画与控制机制”：本章将编写平台、玩家角色动画及其控制操作。在我看来，我们早已完成了其中的困难部分，所以本章大部分工作的投入产出比很高。而且，本章的趣味性很强，将介绍平台如何支撑玩家角色并令其能够跑动，还将演示如何通过循环播放动画帧来实现玩家角色平滑跑动的效果。具体而言，本章将完成编写平台结构、为玩家角色结构添加新功能、实现 `Animator` 类、实现动画效果、添加玩家角色平滑跑动的动画等工作。

第 19 章，“创建菜单与实现下雨效果”：本章将实现两大重要功能：其一是能够为玩家提供开始、暂停、重新开始与退出游戏等功能选项的游戏菜单界面，其二是营造出简单的下雨效果。可能你会认为下雨效果没有必要，甚至可能不适合 Run 游戏，但这个技巧简单又有趣，很适合学习并掌握。这里更值得期待的是我们如何通过再次编写 `Graphics` 与 `Update` 的派生类，并将其组合为 `GameObject` 实例来完成这两个目标，同时需要保证这两个派生类能与游戏中的其他实体协同工作。

第 20 章，“火球与空间化”：本章将添加所有的音效与 HUD。虽然前几个项目也实现了音效，但这一次稍有不同，因为我们将探索声音**空间化**(spatialization)这个复杂的概念，并学习 SFML 让它变得简单而优雅的方法。

第 21 章，“视差背景与着色器”：本章是 Run 游戏编写过程的最后一章，在添加所有功能后，它便能完整地玩起来了。在整款游戏的收尾过程中，我们将初步学习 **OpenGL**、着色器与**图形库着色语言**(Graphics Library Shading Language，**GLSL**)，并实现可滚动的背景与着色器，以最终完成 `CameraGraphics` 类，还将使用他人的代码在游戏中使用着色器。最后，我们会运行整个游戏。

如何最大化本书的阅读效果

阅读本书没有任何前置知识要求，不需要知晓任何编程知识，因为本书将带领你从零学起，并最终得到 4 个可玩的游戏。此外，拥有几种电脑游戏的体验并有学下去的决心对阅读本书会有所帮助。

下载示例代码文件与彩图

读者可通过扫描本书封底的二维码下载本书的源代码。我们还提供了一个 PDF 文件，其中含有本书所用的全部屏幕截图与图表素材，读者可以通过网址 https://packt.link/gbp/9781835081747 下载，也可以通过扫描本书封底的二维码下载。

目　录

第1章

欢迎阅读《C++游戏编程入门》(第3版)

从本章开始，我们将踏上使用C++和OpenGL驱动的**SFML**库为PC编写精彩游戏的旅程！本书已全面升级至第3版，在深度和广度上进行了显著提升与扩展。新版内容丰富，涵盖了从C++基础(如变量、循环)到面向对象编程、**标准模板库**(Standard Template Library)、**SFML**特性，乃至C++的新功能等知识。完成这段学习之旅后，你不仅将掌握4款好玩的游戏，还将奠定深厚而坚实的C++基础。

本章将涵盖以下主题：

- 首先，我们将介绍本书中编写的4个游戏。第一个游戏与第2版中的相同，旨在帮助你掌握C++基础，如**变量**(variable)、**循环**(loop)和决策制定。第二、三个游戏在第2版的基础上进行了增强、修改和优化，而第四个游戏是新增的，在我看来，这个游戏在可玩性和所提供的学习价值方面远远超过了第2版中最后两款游戏的总和。
- 接下来的内容至关重要。我们将深入探讨为何应该使用C++来学习游戏编程，或者其他类型的编程。使用C++来学习游戏开发可能是最佳选择，原因有很多。
- 最后，我们将探索**SFML**库及其与C++的密切关系。
- 没有人喜欢针对企业的宣传，本书也不会涉及这类宣传，但了解**Microsoft Visual Studio**以及我们为什么在本书中使用它的确有充分的理由。
- 接下来，我们可以开始搭建开发环境了。诚然，这是一项稍显乏味的工作，但我们会迅速而有序地一步一步完成它。好在这项工作对于每个项目而言是一次性的，一旦完成便无需再次进行。
- 随后，我们将规划与筹备第一个游戏项目：Timber!!!。
- 紧接着，我们将编写本书的第一段C++代码，并制作出此游戏的第一个可运行版本，即绘制出游戏的漂亮背景。在下一章中，我们将进一步推进该游戏，让图形动起来。本章所学的知识将为我们的第一个游戏项目取得更快的进展奠定坚实的基础。
- 最后，我们将探讨在学习C++和游戏编程过程中可能遇到的问题及解决方法，包括配置错误、编译错误、链接错误和缺陷等。

当然，你最想知道的是，在读完这本厚重的书籍后，你会收获哪些知识。下面我们将更深入

地了解即将构建的游戏。

本章的源代码可以通过扫描本书封底的二维码下载。

1.1 我们将构建的游戏

这次学习之旅将非常顺畅，因为我们会循序渐进地介绍 C++这一高效编程语言的基础知识，然后通过为即将构建的 4 个游戏添加酷炫功能来运用这些新知识。

下面介绍本书中的 4 个游戏项目。

1.1.1 Timber!!!

Timberman 是一款令人上瘾、节奏紧凑的热门游戏，而我们的第一款游戏是它的翻版。在 Timber!!!这款真正好玩的游戏的构建过程中，我们将了解 C++的所有基础知识。完成该游戏并添加一些增强功能后，我们的游戏版本将如图 1.1 所示。

图 1.1 Timber!!!游戏

Timberman 可以在 http://store.steampowered.com/app/398710/找到。

1.1.2 Pong

Pong 是最早的几款电子游戏之一，非常适合演示游戏对象动画、玩家输入、碰撞检测等常见游戏机制的基本原理。本书将在第 6 章与第 7 章中制作一个简化版的 Pong 游戏，同时探索类与面向对象编程的概念，其最终效果如图 1.2 所示。

图 1.2 Pong 游戏

在该游戏中，玩家需要控制屏幕底端的球拍，把球重新打回屏幕上端。另外，如果有兴趣，可以访问 https://en.wikipedia.org/wiki/Pong 了解 Pong 的历史。

1.1.3　Zombie Arena

接下来，我们会构建一款疯狂的射击游戏，它很接近于 Steam 上的一款僵尸射击游戏 Over 9000 Zombies!，其地址为 http://store.steampowered.com/app/273500/。在我们的游戏中，玩家需要用一顶机枪，在一个随机生成的游戏世界中击退逐批次增加的僵尸。这个游戏的最终效果如图 1.3 所示(可惜静态图并未演示出我们的滚屏效果)。

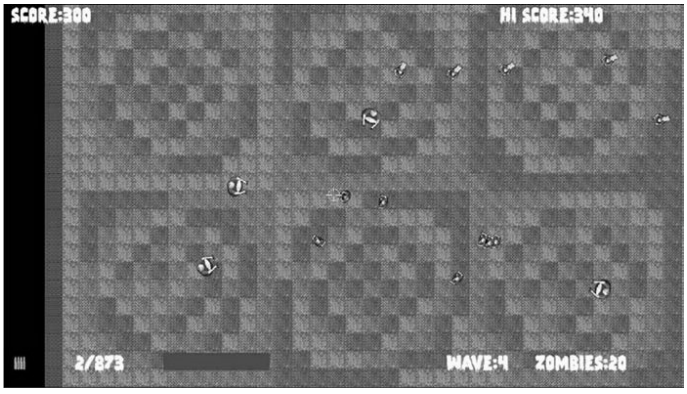

图 1.3　Zombie Arena 游戏

在实现期间，我们会进一步接触面向对象编程思想，学习将代码整理为易于编写和维护的大型**代码基**(code base，意指大量代码)的方法，并实现海量敌人、速射武器、拾取包等经典功能，再令游戏角色在清空每波敌军后升级。

1.1.4　Run

最后的游戏属于**平台游戏**(platform game)，叫作 **Run**。这款游戏的玩法丰富多样，是完全凭借我们自己的 C++技能，在强大的 SFML 库的辅助下逐步实现的，其最终效果图如图 1.4 所示。

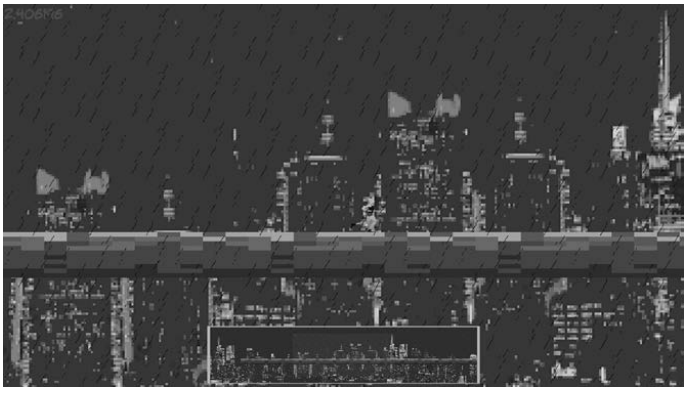

图 1.4　平台游戏

这款游戏具备仿真着色器背景、平行滚动的城市风光、空间化(指向性)音效、小地图、动画化游戏角色、天气(下雨)效果、音乐、弹出菜单等众多功能。同时，Run 堪称 4 个游戏之最，其中的很多代码具备优秀的重用价值，完全可以供你自行实践游戏编程。

以上是我们将制作的 4 款游戏。接下来探讨第二个话题，即 C++游戏编程。

1.2 为什么要学习 C++游戏编程

这个标题也可以理解为“为什么要通过游戏编程来学习 C++”。这是因为(在我[1]看来)，C++、游戏编程和初学者是一个完美的组合。让我们更深入地探讨 C++，同时依然专注于游戏和初学者。

- **运行速度**：C++因其高性能、高效率而著称。在游戏开发过程中，性能是一个重要因素，而 C++代码的运行效率几乎能与 CPU 或 GPU 内部所用的语言相媲美，因此 C++对高度依赖于性能的程序非常有吸引力，而游戏正属此列。而且，C++可以被转化为内部执行的机器指令[2]，这成就了其高效率，也满足了游戏编程的需求：很多游戏动辄带有千百甚至数十万实体内容，因此必须顾及运行效率。我们在第 21 章会看到 C++通过使用着色器程序与 GPU 直接交互的方法。
- **跨平台开发**：C++不限定平台环境，无需伤筋动骨地修改代码，便能在多种平台上编译和运行。虽然本书基于 Windows 编写，但其中的代码只需要稍加改动，即可运行在 MacOS 或 Linux 上。此外，下一代的控制台游戏乃至移动端游戏的开发过程中更是在大量使用 C++。这里的“编译”是一个过程，负责为 CPU 将 C++代码转化为机器指令。
- **丰富的游戏引擎和游戏库**：很多游戏引擎与游戏库要么用 C++编写，要么提供 C++ API，所以具备坚实 C++基础的开发人员可以利用大量的游戏开发工具，例如，大名鼎鼎的 **Unreal Engine** 便主要使用 C++进行开发。再如，**Vulcan**、**OpenGL**、**DirectX**、**Metal** 等都是性能优异的 C++图形库，物理库 **Box2D**、UI 工具 **IMGUI** 同属 C++资源。此外，还有 **RakNet**、**Enet** 等用于支持协同操作和多玩家的网络库资源，SFML 同样具有这些网络功能。
- **底层控制**：C++提供对硬件的底层控制，这对于优化游戏性能至关重要。在游戏开发中，程序员可能需要管理内存、优化渲染流水线(又称为“渲染管线”)，还需要临时管理运行游戏的系统，而功能强大且灵活的 C++完全能够胜任这些任务。如果“内存管理”与“渲染流水线”显得有些高深莫测，请放心，本书将透彻介绍这两个概念，让初学者也能理解，本书第 10 章和第 21 章将分别详述它们。此外，在面对它们的时候，你不必犹豫不前，因为这些机制无论多么复杂，都是可控的，掌握它们不仅可以让你亲身感受到这门语言的强大，也能增进对编程生涯的认同感。
- **文档与支持**：围绕着 C++游戏开发有着非常活跃的社区，有丰富的教程与论坛资源可供答疑解惑。如果你遇到了某个 C++问题，我可以保证你不是第一个有此疑问的人，所以一次轻松的网络搜索，基本上就能够找到解决方法。ChatGPT 也是解决 C++问题的顶尖高手。

---

1 本书中的“我”代指原作者而非译者。另外，除非有明确的标注，否则脚注来自译者。

2 C++代码本身不能直接运行，能够直接运行的是机器指令，而可执行程序可以理解为一系列 CPU 机器指令的组合。但 C++代码可以被转换为机器指令，并进一步组成可执行程序来运行，这一步骤也是 C++高效率的原因。这个转换操作涉及很多步骤，编译便是其中之一，由编译器完成，后文还会提到其余步骤。

- 学习 C++确实会遇到一些挑战，但只要脚踏实地，掌握它并不难。不要畏惧，请大胆迎接挑战，因为它最终能够解决并成为游戏内的一抹亮色，这显然是非常令人欣慰的。虽然游戏开发经常需要使用一些复杂的算法以及数据结构，并需要遵循一定的经验法则，但 C++有**标准模板库**(Standard Template Library，**STL**)，也有各种各样的类工具，这是**面向对象编程**(Object-Oriented Programming，**OOP**)思想在 C++中的直接体现，它把那些复杂的技术分解为若干可复用的结构，从而降低了学习压力。本段中提到的 OOP 将在本书中的第 6 章介绍，而 STL 则会在第 10 章介绍。

关于 OOP 还需要多提一点。现代 C++编程离不开 OOP 思想，这也许是它最大的优势。人们所接触的每份 C++新手教程都会传授 OOP 思想并加以实践，而且 OOP 是现代编程的主流思想，基本上也是编程语言的未来趋势。所以，如果希望从头开始学习 C++，为什么要绕开 OOP 呢？

- **C++行业标准**：正因为前面所讨论的各项优势，C++得以在目前的游戏开发界广泛使用，所以熟练的 C++技巧有利于与其他开发者合作，上手既有代码基也比较容易，甚至允许在不同工作中使用不同的游戏引擎，从而在业内维持高薪。

但是，持批评意见的人会说，相比于其他语言，C++的学习曲线相对陡峭，而且对于那些从未接触过编程或游戏开发的人而言，与其从 C++上手，不如先学习一些对初学者更友好的语言，如 **C#**或 **Python**(C#是 Unity 开发所使用的语言，Python 则针对简单游戏项目)。这些意见原本不无道理，但 C++是一门活跃的编程语言，从未停下演进的脚步，近年来更是引入了很多改进与优化以降低学习成本，并提升开发效率。**auto** 等关键字[1]、**spaceship** 运算符(宇宙飞船运算符)以及 lambda 表达式、协程、智能指针等 C++新技术便是在过去十余年间引入的，这让那些批评意见不再无懈可击。

总而言之，我认为把 C++作为首门编程语言不是一种错误。如果希望提升学习过程的趣味程度，并希望尽快看到效果，通过游戏来学习编程更是当仁不让的选择。最后，如果有意成为一名独立开发者，或就职于顶级游戏工作室，除非有更明确、更直接的办法，否则 C++自然是首选。

既然 C++非常优秀，也有很多资源库，为什么还要选择使用 SFML？

1.2.1　SFML

SFML 全称“**Simple Fast Media Library**”，它不是为游戏或多媒体而开发的唯一 C++库，还有其他选项，但我总会忍不住想到它。首先，SFML 是用面向对象的 C++编写的，它的诸般优势会在我们推进项目的过程中慢慢体现出来。

同时，SFML 还易于上手，初学者可以放心选用，而行家里手也能利用 SFML 构建出高水平的 2D 游戏，所以通过 SFML 入门，便不用担心其跟不上自身知识储备的速度。同时，2D 游戏是 SFML 主要的贡献领域(本书即为了制作 2D 游戏而使用 SFML)，而 Unreal Engine 则更适合构建 3D 游戏，尝试由 SFML 入门后再转向此 3D 引擎，可能更加顺利。

SFML 库基本涵盖开发 2D 游戏所需要的全部功能。另外，考虑到 SFML 底层是由 OpenGL 实现的，所以使用 SFML 也在使用 OpenGL，但 OpenGL 还能用于构建 3D 游戏和跨平台游戏。

1　关键字(keyword)，又称关键词，指 C++等编程语言内部事先定义的、有特殊意义的标识符。

SFML 提供了以下功能：

- 2D 图形及 2D 动画，并能制作出大型游戏世界(指大于一个屏幕、需要移动视野来观察的游戏世界)。
- 音效与音乐(包括高品质的指向性声音)。
- 处理键盘、鼠标与游戏手柄等输入信号。
- 支持在线多玩家模式。
- 不需要改动，代码便可在所有主流桌面操作系统中编译和运行，甚至包括移动端。

经验证明，即使是资深开发者也没有比使用 SFML 更合适的方法来使用 C++构建 PC 端 2D 游戏，初学者更能从中感受到利用妙趣横生的游戏开发过程来学习 C++的魅力。至此，我们已介绍了 C++，也介绍了 SFML，接下来则需要介绍开发工具了。

1.2.2 Microsoft Visual Studio

Visual Studio 是一种界面清晰而功能强大的**集成开发环境**(Integrated Development Environment，**IDE**)，它既能简化游戏开发过程，又不会让人们忽略掉一些高级编程功能，例如，其代码补全与语法高亮等功能便十分有助于 C++学习。在诸多 IDE 中，Visual Studio 几乎被公认为是最先进的免费 IDE，而 Microsoft 公司之所以放弃了这份收益，不是在弥补其过往的过失，而是在放长线钓大鱼，希望吸引人们将来转用付费的版本。所以，目前还是暂且享受这份免费的乐趣吧。

Visual Studio 提供了强大的调试器[1]，它支持断点设置与调用栈等功能。在 Visual Studio 中运行游戏时，可以让它停在自定义的断点处，以便审查代码中的具体数值，并可以逐行运行代码。逐行运行非常有助于让初学者理解代码的工作方式，甚至可以尝试自行解决代码中的一些错误。

IntelliSense 是 Visual Studio 的代码提示器，也具有实时勘误功能。此外，其所提供的即时高亮错误与自动补全等功能非常有利于提升学习效率。这不仅仅是初学者的一个绝佳的学习工具，也能极大提升编程老手的工作效率。

Visual Studio 拥有庞大而活跃的社区，具备丰富的教程资源以及能够答疑解惑的论坛，可以让初学者尽快掌握在 Visual Studio 中使用 SFML 来完成 C++项目的方法。

Visual Studio 还提供了很多高级功能。例如，Visual Studio 整合了常见的**版本控制系统**(Version Control System，VCS)，如 **Git**，以便管理由多名程序员参与的大型项目，提升团队效率。此 IDE 同样提供性能监视功能，以便监测游戏的内存与 CPU 使用率，进而优化游戏。而且，Visual Studio 没有故步自封，仍在持续开发，正如你那日益增长的知识技能。

Visual Studio 几乎形成了行业标准。作为 C++开发最常用的 IDE，Visual Studio 的用户数量非常庞大，所以初学者在遇到困难时可以找到针对性的在线指导。此外，如果面对某困境确实束手无策，还可以求助于 Microsoft 公司。不要畏惧麻烦，因为对 Visual Studio 了如指掌也是一种价值。

Visual Studio 把预处理、编译与链接[2]的复杂操作封装在一键式操作之下，并提供了华丽的用户界面供用户输入代码，而且无论项目中的代码文件以及其他资源有多少，Visual Studio 都能够轻松管理它们。

---

1 调试器是用来调试的；关于调试及其意义，请参见本章最后的 1.12 节。

2 除编译外，预处理和链接同样参与把 C++代码转化为可执行程序的过程，参见后文。

虽然 Visual Studio 拥有诸多优势，但使用它创建的游戏项目同样可以由其他开源工具代为实现。只是即便如此，以 Visual Studio 入门非常简单，而如果需要使用其他开发工具，从 Visual Studio 迁移过去也往往比直接使用那种工具更加顺畅。

尽管 Visual Studio 也提供了需要花费上百美元的高级版本，但我们仍可使用免费的 Visual Studio 2022 Community 版本来构建游戏。之所以使用 2022 版本，是因为在撰写本书时，这已经是最新版本；但如果有更新，建议使用新版本，因为 Visual Studio 努力让自身成为**向后兼容**(backward compatible)的软件，多年间又尽量不去大幅改动用户界面，所以换用新版既可以享受到新的功能，用起来也更顺畅，本书的示例还能正常运行。

接下来，我们会首先讨论在 Mac 与 Linux 操作系统下使用本书的方式，随后便开始实际搭建开发环境。

1.2.3　在 Mac 或 Linux 操作系统下使用本书的方法

本书中的 4 款游戏在 Windows、MacOS 和 Linux 系统上均可运行。准确地说，运行在不同系统上的游戏具有相同的代码，只是需要各自进行编译与链接操作，而本书针对编译和链接操作的说明仅适合 Windows 系统。

尽管本书并不完全适合 Mac 与 Linux 系统，对初学者尤其如此，但如果你是这两个系统的拥趸，也不希望换用 Windows，那么别担心，这 4 个项目仍旧可以顺利完成。虽然本书针对 Windows，但更换平台所涉及的附加挑战主要出现在搭建开发环境、配置 SFML 以及第一个项目等过程中，此后则将一马平川。

接下来将介绍为 Windows 系统搭建开发环境的方法，直到 1.5 节为止。非 Windows 系统的用户请转用以下教程：

- Linux 用户请遵照 https://www.sfml-dev.org/tutorials/2.6/start-linux.php。
- Mac 用户请遵照 https://www.sfml-dev.org/tutorials/2.6/start-osx.php。

1.2.4　安装 Visual Studio 2022

本书创建游戏的第一步是安装 Visual Studio 2022。此 IDE 的安装过程非常简单：下载一个文件，单击几个按钮，完成。至于其中的难点，最多是在选择下载的软件时需要稍加注意。本小节将引导你下载正确的版本。

虽然多年来，Microsoft 经常改动 Visual Studio 产品的具体名称、外观及下载地址，导致接下来介绍的用户界面的布局以及相应的安装方式存在过期的可能，但据我所知，Microsoft 仍然尽力让不同版本保持一致。此外，为每个项目配置 C++及 SFML 非常重要，但版本更迭并不一定会让配置教程彻底失效，只要经过审慎编辑，那么即便新版的 Visual Studio 已经经过了重大升级，为旧版 IDE 编辑的教程同样可能适用于新版产品。

接下来，我们开始安装 Visual Studio。

(1) 首先需要一个 Microsoft 账户的登录信息。Hotmail、Windows、Xbox 或 MSN 等账户都可以用作 Microsoft 账户；如果都没有，可以在 https://login.live.com/中免费注册。

(2) 在撰写本书时(2024 年 5 月)，Visual Studio 2022 是最新版本，也是这本教程所针对的版本。请访问 https://visualstudio.microsoft.com/，向下滚动，找到 Visual Studio 的下载链接。图 1.5 是我在访问这个网址时它的状态。

图 1.5　下载 Visual Studio

(3) 在 Visual Studio 的下载按钮旁边的下拉菜单中选择 **Community 2022**。注意，此菜单中另有两个版本，它们都是需要付费的共享软件，而右边的 Visual Studio Code 则与本书完全无关。单击“**保存**”(Save)按钮，开始下载[1]。

(4) 下载完毕后，双击运行所下载的文件。此时会请求以管理员身份运行，以便对计算机进行修改。请授予 Visual Studio 此权限，等待安装程序自动下载一些额外的文件，随后自动进入安装过程的下一阶段。

(5) 很快，安装程序会询问 Visual Studio 的安装位置，此时需要选择一块至少有 50 GB 可用空间的硬盘分区。虽然大多数教程的建议远小于 50 GB，但一旦开始构建项目，50 GB 一般能保证未来的开发过程不会受限于硬盘空间。选定分区后，找到“**使用 C++的桌面开发**”(Desktop Development with C++)选项并选中它，随即单击“**安装**”(Install)按钮，这一步骤可能需要花费不少时间。

接下来，我们便可以把注意力转移到 SFML 的配置上，并准备开始第一个项目了。

1.3　搭建 SFML 环境

本节这份简明教程将带领读者下载我们需要的 SFML 库。此外，我们还将学习让 SFML 的 DLL 文件与我们的代码协作的方法。以下是搭建 SFML 环境的详细步骤。

(1) 访问 SFML 网站内的 http://www.sfml-dev.org/download.php，单击“**最新稳定版本**”(Latest stable version)按钮，如图 1.6 所示。

1　有时，选定“Community 2022”后，不需要单击“保存”按钮即可开始下载。

图 1.6　下载 SFML 2.6

(2) 在阅读本书时，2.6 可能不再是最新版本，但这不影响接下来的操作。这里我们需要下载 32 位的 SFML 库——32 位稍稍有违直觉，毕竟你更可能使用 64 位的电脑(64 位也更常见)，但 32 位的 SFML 库在 32 位与 64 位设备上都能运行，64 位库则不然。还需要选择与 Visual Studio 2022 适配的版本，并单击 Download 按钮，见图 1.7。

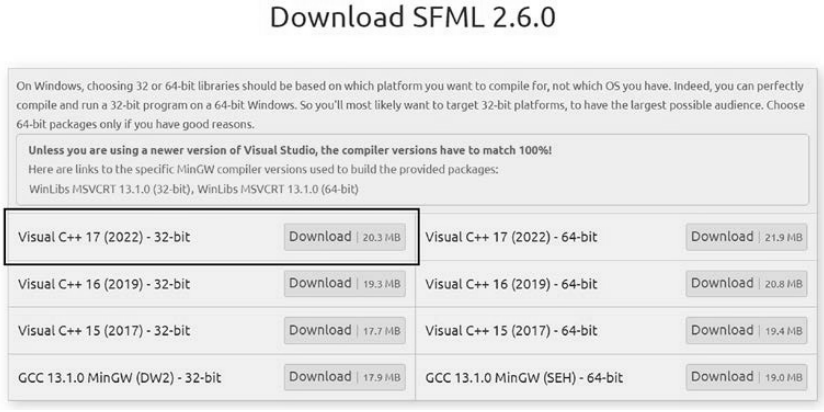

图 1.7　下载 SFML 17_22

(3) 下载完毕后，在安装有 Visual Studio 的分区根目录上新建 `SFML` 与 `VS Projects` 这两个文件夹。

(4) 最后，在桌面上解压缩所下载的 SFML 文件。我的文件是 `SFML-2.6.0-windows-vs17-32-bit.zip`，对应 2.6 版，如果 SFML 有更新，其名称也会相应变化。解压缩完毕后，可以删掉解压缩所得的`.zip` 文件夹，保留此时桌面上的那个新文件夹，其名称将反映出所下载 SFML 的版本。双击打开此文件夹，从中可以找到子文件夹 `SFML-2.6.0`，再次双击打开它[1]。

图 1.8 展示了我的文件夹中的内容，你的文件夹内容应该与此类似。

请把此文件夹内的全部内容复制到第(3)步创建的 `SFML` 文件夹中。本书此后将把这个文件夹称作“你(的)SFML 文件夹”。

现在，我们可以开始在 Visual Studio 中利用 SFML 库来进行 C++编程了。

1　有时解压缩得到的文件夹可能与本段的介绍不符，这与解压方式有关，例如，可能没有这个`.zip` 文件夹(那时自然不需要再删除)，或者解压缩后将直接得到原文中的子文件夹，而没有外层文件夹。无论如何，最终需要的文件均位于此“SFML-<版本号>”文件夹内。

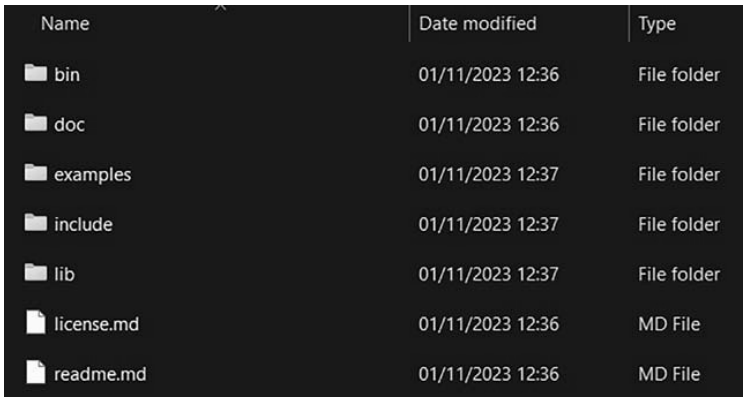

图 1.8　SFML 文件夹的内容

1.4　新建 Visual Studio 项目

本节会用尽可能简洁的语言逐步介绍新建项目这个容易出错的过程，希望读者能够尽快适应。

(1) 作为一个软件，打开 Visual Studio 的方法同样是在 Windows 开始菜单中单击其图标，它是由默认安装过程添加的。打开此软件后，你将看到图 1.9 所示的窗口。

图 1.9　在 Visual Studio 2022 中新建一个项目

(2) 图 1.9 中的方框内是“**创建新项目**”(Create a new project)按钮，单击后会弹出“**创建新项目**”(Create a new project)窗口，见图 1.10。

图 1.10　“创建新项目”窗口

(3) 在“**创建新项目**”窗口中，我们需要选择所创建项目的类型。我们的第一个项目属于控制台应用，它不涉及菜单、复选框等与窗口相关的内容，也没有使用其他 Windows 工具，所以请选择“**控制台应用**”(Console App)，并单击“**下一步**”(Next)按钮，见图 1.10。这会打开如图 1.11 所示的“**配置新项目**”(configure your new project)窗口，该图显示了步骤(6)结束后此窗口的状态。

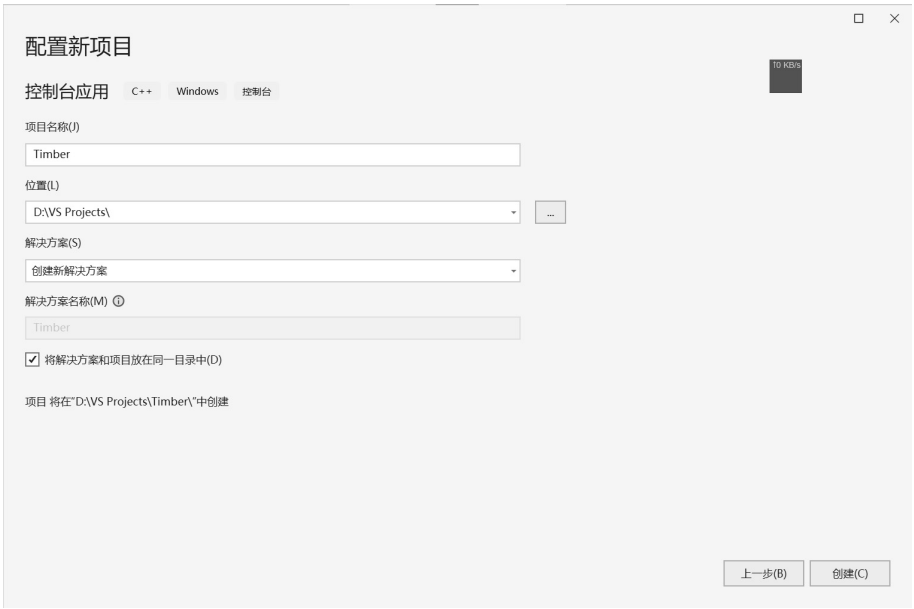

图 1.11　配置新项目

(4) 在“**配置新项目**”窗口内的“**项目名称**”(Project name)一栏填入 `Timber`。注意，这会让 Visual Studio 把“**解决方案名称**”(Solution name)自动配置为相同名称。

(5) 在“**位置**”(Location)栏，选定上一节创建的 VS Projects 文件夹，这是我们保存所有项目的路径。

(6) 选中“**将解决方案与项目放在同一目录中**”(Place solution and project in the same directory)前的复选框。

(7) 此时，“配置新项目”窗口应呈现出图 1.11 所示的状态，单击“**新建**”(Create)按钮便可创建 Timber!!!项目，其中附有少量 C++代码，见图 1.12，这也是我们的主要编程环境。

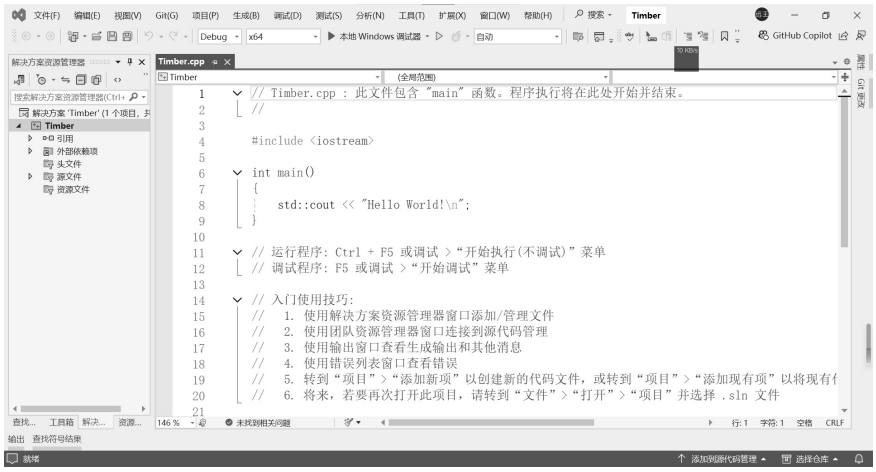

图 1.12　Visual Studio 代码编辑器

(8) 下面需要配置项目，以使用 `SFML` 文件夹内的 SFML 库。从 Visual Studio 主菜单中选择“项目”(Project) -> “**Timber 属性…**”(Timber Properties...)，这将弹出图 1.13 所示的窗口。

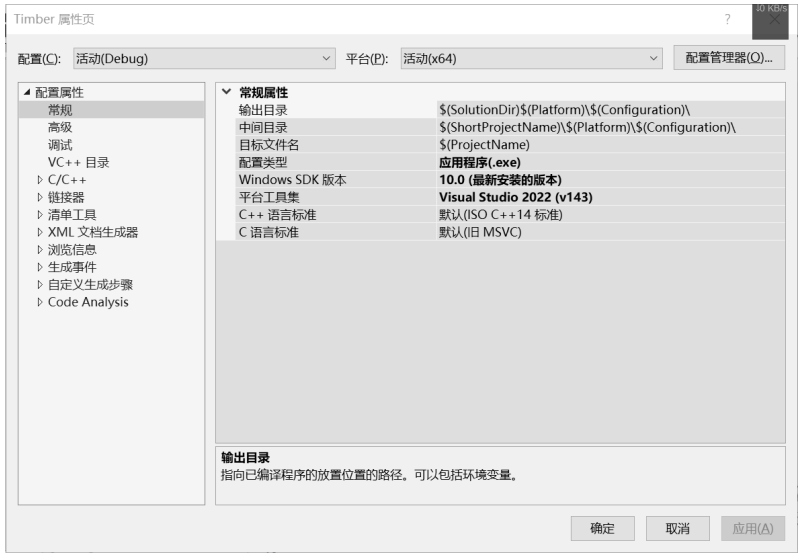

图 1.13　Timber 属性页

说明：此图内的“**确定**”“**取消**”“**应用**”三个按钮并不完整，这可能是 Visual Studio 的一个小故障，即其未能正确适配我的屏幕。虽然你应该不会遇到同样的问题，但即便遇到，也并不影响接下来的操作。

下面，我们将开始配置项目属性。鉴于这些步骤比较琐碎，这里会专门用一小节的篇幅来介绍。

配置项目属性

所有的配置工作都是在上一小节末尾所展示的“**Timber 属性页**”(Timber Property Pages)中进行的，所以在介绍本小节之前，请不要关掉它。由于配置过程涉及很多琐碎的细节，为描述清晰起见，我添加了一些数字标注，见图 1.14。

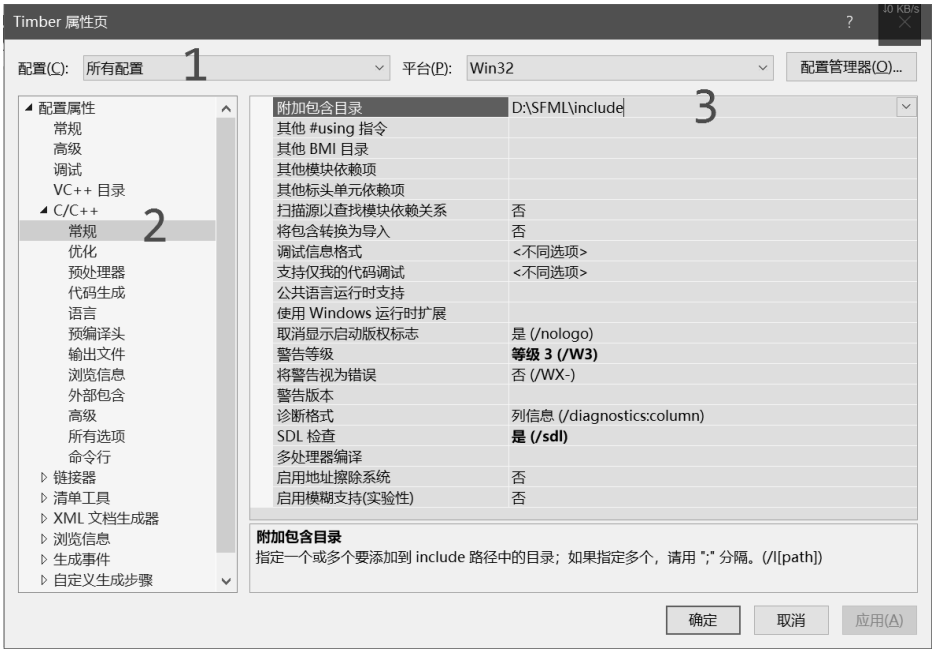

图 1.14　标注后的项目属性配置页面

再次提醒大家，本小节将进行的项目配置虽然很琐碎，却是每个项目能够正常运行的必要条件。所幸每个项目仅需配置一次，而且熟悉后，此配置过程会越做越快，越做越简单。对于我们的项目而言，Visual Studio 需要知道获取第三方 SFML 库的位置，而这些文件可分为两种特殊的类型。其一为头文件，与前面主要的源代码文件不同的是，它们主要包含在扩展名为.hpp 的文件中(虽然.h 这个扩展名更常见)，主要负责指导编译器在我们使用 SFML 代码时如何处理那些代码。我们在构建下一个项目时会创建自己的头文件，那时便能更清晰地演示这一点。其二为 SFML 库文件，其位置同样需要告知 Visual Studio。为此，我们需要在“**Timber 属性页**”中加以配置，具体可分为以下三个主要步骤：

(1) 在标注“**1**”处，在“**配置**”(Configuration)下拉菜单中选择“**所有配置**”(All Configurations)，并保证在右边的下拉菜单中已选中“**Win32**”项。

(2) 在标注“**2**”处，从左侧菜单中依次选择“**C/C++**”->“**常规**”(General)。

(3) 在标注“**3**”处，单击“**附加包含目录**”(Additional Include Directories)编辑框，进入编辑状态，此时填入SFML文件夹所在的分区号并继续键入\SFML\include。举例来说，如果SFML文件夹位于D:盘，那么需要键入的完整路径为D:\SFML\include，见图1.14；而如果SFML位于其他分区，请相应修改路径。

(4) 单击“**应用**”(Apply)按钮保存目前已完成的修改。

(5) 至此，附加包含目录已编辑完毕。接下来，仍在此窗口内，请参照图1.15继续编辑。首先，在标记“**1**”附近，依次选择“**链接器**”(Linker)->“**常规**”(General)[1]。

(6) 其次，找到“**附加库目录**”(Additional Library Directories)编辑框(即“**2**”标记点)，并向其中键入SFML文件夹所在的分区号，再键入\SFML\lib。例如，如果你的SFML文件夹位于D盘，那么所需键入的完整路径是D:\SFML\lib，见图1.15；如果SFML位于其他分区，请相应修改路径。

图1.15 附加库目录

(7) 单击“**应用**”(Apply)按钮保存目前所完成的修改。

(8) 接下来完成此窗口内最后的配置任务。请在此窗口内标记“**1**”处的“**配置**”(Configuration)下拉菜单中选择“**Debug**”，见图1.16。

(9) 在标记“**2**”的区域中依次选择“**链接器**”(Linker)与“**输入**”(Input)。

(10) 定位到“**附加依赖项**”(Additional Dependencies)编辑框(“**3**”标记处)，单击此框最左侧，进入编辑状态，并完整添加以下各项：

```
    sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml
-audio-d.lib;
```

1 这里我们在“链接器”中再次看到了“链接”这个概念。简单而言，与链接操作相关的正是前面提到的库文件，这些库文件与我们的源代码协作后才能得到可以执行的程序。

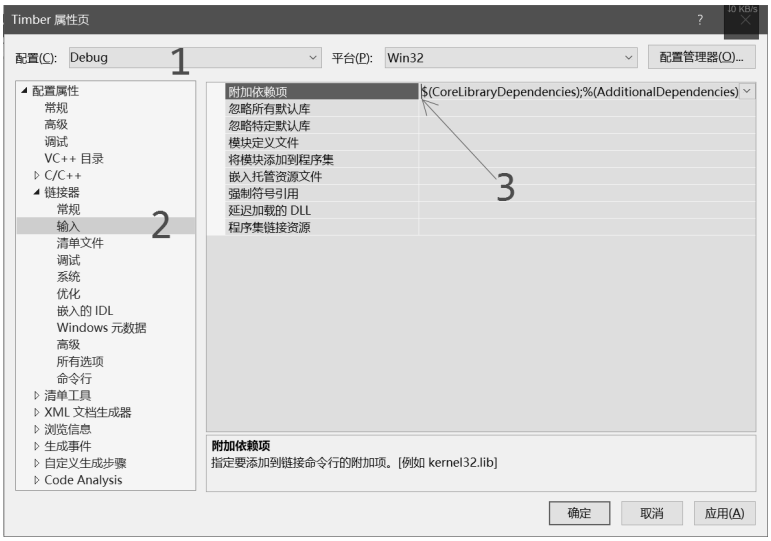

图 1.16　链接输入配置

注意，一定要在最左侧输入这些内容，不要多字少字，最后的分号也是不可或缺的。

(11) 单击“**确定**”(OK)按钮。

(12) 先后单击“**应用**”(Apply)与“**确定**”(OK)按钮。

至此，Visual Studio 的配置工作已全部完成，下面将开始规划 Timber!!!项目。

1.5　规划 Timber!!!项目

每个游戏的创建过程均应始于纸笔——如果不能精确地设计出游戏在屏幕上的呈现效果，何谈用代码实现？

进行规划时，如果完全无从下笔，不妨先一探 Timberman 的运行效果，让自己心中有数。这款游戏在 Steam 的地址是 http://store.steampowered.com/app/398710/Timberman/，而且往往有折扣，一般不到 1 美元，如果感觉自己的预算还撑得住，可以买下并亲身体验一番。

一款游戏的具体玩法由其提供的各种功能和游戏对象所定义，这就是所谓的**游戏机制**(mechanics)。我们游戏的基本机制包括以下几点：

- 时间永远在流逝。
- 砍树可以延长时间。
- 砍树也会导致枝杈掉落。
- 玩家必须躲避下坠的枝杈。
- 时间耗尽或玩家角色被枝杈压扁则游戏结束。

应该说，在这份 C++新手教程第 1 章的这个位置便期望你立刻开始编写 C++代码，并不是什么明智的要求，所以我们首先会展示此游戏最终的功能效果，再浏览其所用的诸般素材，最后才会开始编程。

图 1.17 是我们游戏的截图，其中带有注解。

图 1.17　Timber!!!游戏场景

从中可见，我们的游戏具有以下功能。

- **玩家得分**：玩家每砍一棵树，都会得 1 分。另外，在游戏中，砍树是用左右方向键来完成的。
- **玩家角色**：玩家每次砍树时所使用的方向键也会决定他与树的左右位置关系，所以在砍树后，他可能停在原地，也可能跳到对侧。由于枝杈持续掉落，玩家必须小心选择砍树的位置，以免被压扁。
- 当玩家砍树时，他的手掌上会播放一段简单的斧头动画。
- **持续缩短的时间棒**：玩家每砍一次树，都会让时间棒少量增长。
- **致命的枝杈**：玩家砍得越快，赢得的时间便越多，但也会让枝杈从树上坠落得越快，更容易把玩家压扁。另外，枝杈是在树顶随机生长出来的，并伴随着每次砍树而下落。
- 当玩家角色被压扁时，会播放一段墓碑动画。经验表明，这段动画会经常出现。
- **已砍下的树**：玩家每次砍树，都会砍掉一段木料且木料会飞走。
- **装饰**：界面内有三片云朵，其运动速度与高度均是随机的；同时，这里还有一只肆意飞动的蜜蜂。
- **背景**：我们的游戏有个漂亮的背景，全部操作均位于这个背景之上。

通过简单分析可知，玩家为获取高分，必须疯狂地砍树，这虽然也能避免耗尽时间，但砍得越快，也越容易被压扁。

现在我们初步了解了游戏的场景、玩法及其背后的动机，接下来便准备实现它。为此，请遵循以下步骤。

(1) 把 SFML 的 `.dll` 文件复制到项目的主文件夹内。我的项目主文件夹是 `D:\VS Projects\Timber`，这是 Visual Studio 在 `VS Projects` 文件夹内自动创建的，也是这里的目标文件夹。如果你的 `VS Projects` 文件夹位于其他分区，操作是类似的。我们需要的文件位于 `SFML\bin` 文件夹中，也即源文件夹。请在两个窗口中分别打开源文件夹与目标文件夹，并参照图 1.18 选中源文件夹内的全部文件。

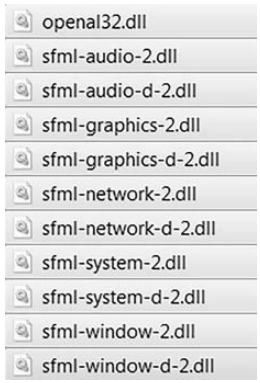

图 1.18　选中所需的全部文件

(2) 现在，请复制这些文件，并把它们粘贴到目标文件夹 `D:\VS Projects\Timber` 中。

(3) 至此，我们可以准备开始编写这个项目。此时，你的屏幕应该如图 1.19 所示，其中包括我的注解，以便你尽快熟悉 Visual Studio 的界面。我们很快会回到这里，正式开始编程。

图 1.19　编写代码的位置

与大部分软件相同，Visual Studio 的窗口也是可定制的，所以你的窗口的布局可能与图 1.19 有所不同。但相比之下，这种布局中**解决方案资源管理器**(Solution Explorer)的位置与大小能让其内容更加清晰，从而提升工作效率，所以值得花些时间来主动调整窗口布局。

接下来，我们首先会遍览本项目所用的各种资源，随后开始实际编程。

1.6　项目资源

项目资源是用于制作游戏项目的全部资源的统称。我们的项目中包含以下资源：

- 用于在屏幕中绘制文字的一个**字体**(font)文件。
- 针对砍树、死亡、时间耗尽等事件的**音频音效**(sound effect)文件。
- 代表人物形象、背景、枝杈等游戏对象的**图片**(graphics)，这些图片又称**纹理**或**纹理图**(texture)。

本游戏所需的图像资源全部位于随书下载的代码包中的 `Chapter01\graphics` 文件夹内，所有音频文件则位于 `Chapter01\sound` 中。

为避免版权纠纷，我没有把那份字体文件也放在其中[1]。但这不是问题，因为本章随后会仔细介绍自主获取字体的方法。

1.6.1 定制音效

从 **Freesound**(www.freesound.org)等网站中可以免费下载各种音效(Sound Effects，FX)，但来自免费网站的音效往往不能在收费游戏中使用。除了这些免费素材，开源软件 **BFXR** 可用于自主创建音效文件，而这些文件完全由其制作者支配。**BFXR** 可以从 www.bfxr.net 获取。

1.6.2 在项目中添加资源

一旦确定项目所需要的资源，下一步是将其添入项目内。本小节假定所用的资源全部来自本书代码包，但如果需要使用自己的资源，可以在完成这些操作后，将自定义资源重命名为对应代码包内的资源名，并复制至此以完成替换。

(1) 定位到项目文件夹中，例如，`D:\VS Projects\Timber`。

(2) 在其中新建三个文件夹，分别命名为 `graphics`、`sound` 和 `fonts`。

(3) 从下载的代码包中，把 `Chapter01\graphics` 文件夹下的全部内容复制到 `D:\VS Projects\Timber\graphics` 文件夹内。

(4) 从下载的代码包中，把 `Chapter01\sound` 文件夹下的全部内容复制到 `D:\VS Projects\Timber\sound` 文件夹内。

(5) 在浏览器中访问 http://www.1001freefonts.com/komika_poster.font，并下载 **Komika Poster** 字体。

(6) 解压缩所下载的 ZIP 包，并将其中的 `KOMIKAP_.ttf` 文件复制到 `D:\VS Projects\Timber\fonts` 文件夹内。

至此，全部资源已就绪，但我们对其内容以及在代码中的大致用途仍旧一无所知。接下来，让我们看看这些资源，特别是图形资源，以便在 C++代码中使用它们时，能够直观地了解所发生的事情。

1.6.3 浏览项目资源

本项目中一共包含三种资源，其中图像资源是构造游戏场景的组块，而游戏场景基本等同于游戏本身。在此，即便不知道这些图片资源的具体用途，但观其内容便可略知一二，如图 1.20 所示。

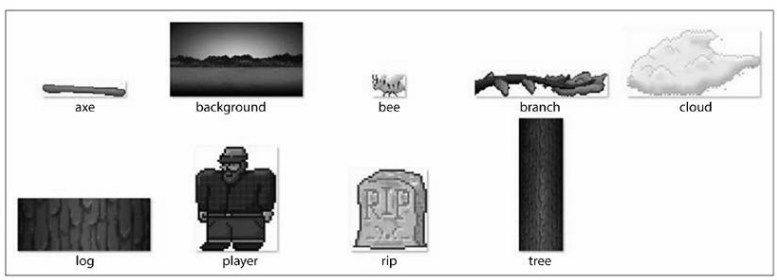

图 1.20 图像资源

1 有时，所下载的资源包中也可能包含这份字体文件，这样就不需要重新下载。

第二种资源是音频文件。本项目的音频文件全部采用.wav 格式，由 **BFXR** 创建，分别对应着游戏中出现特定事件时需要播放的音效，罗列如下：

- chop.wav：模仿用斧头砍树的声音。
- death.wav：接近于复古掌机游戏中失败的声音；在玩家被压扁时播放。
- out_of_time.wav：玩家因时间耗尽而导致游戏失败时播放的音效。

另有字体文件，其效果可见于前面的游戏截图。以上是本项目的全部资源，而接下来，我们会简要讨论屏幕分辨率的话题，其中还会谈到在屏幕中放置图片的方法。

1.7　理解屏幕及内部坐标

在实际开始编写 C++代码之前，还需要谈一谈坐标的话题。一般而言，显示在屏幕上的图像是由像素构成的，像素则相当于一个微小的可着色光点，大量光点按照预定方式组合即可构成屏幕上的图像。

屏幕的分辨率是构成屏幕的像素数。虽然具体的分辨率多种多样，但典型的显示器在横向有 1920 像素，在纵向有 1080 像素。屏幕上的像素会被一一编号，而编号会从屏幕的左上角开始。1920 像素×1080 像素屏幕的横轴(*x* 轴)坐标的范围为 0~1919，纵轴(*y* 轴)坐标则从 0~1079，见图 1.21。

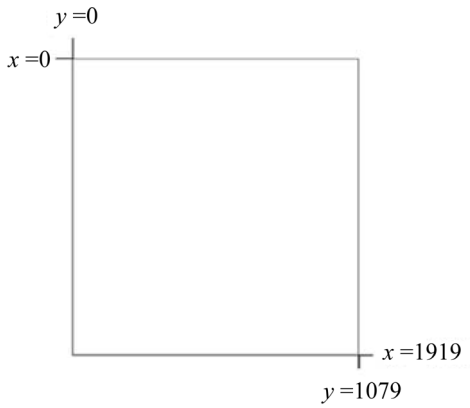

图 1.21　屏幕及其内部坐标

因此，通过 *x* 与 *y* 坐标便可实现精确的屏幕定位。在此基础上，如果把背景、游戏形象、子弹及文本等游戏对象按照游戏的逻辑各自摆放在屏幕上，即构成游戏的一个场景。

我们需要通过像素坐标来指定摆放每个游戏对象的具体位置。图 1.22 演示的便是在屏幕中心附近进行绘制的方法。对于 1920 像素×1080 像素分辨率而言，这个中心位置的坐标是(960, 540)。

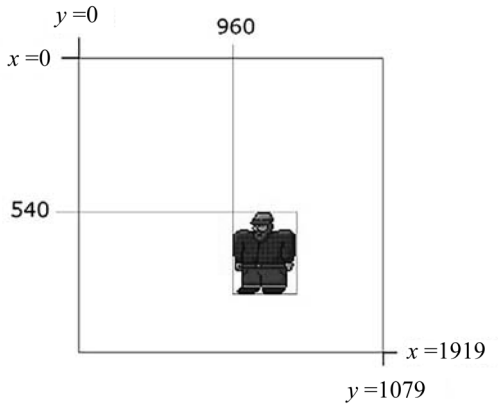

图 1.22　在中央绘制游戏人物

除了这种屏幕坐标，每个游戏对象同样具备属于自身的坐标系统，称为**内部**(internal)坐标或**局部**(local)坐标。这种坐标与屏幕坐标相互独立，但形式类似，而且二者均始于(0, 0)，该点同样位于左上角。

在图 1.22 中，我们把人物形象的(0, 0)内部坐标置于屏幕的(960, 540)处，从而让它呈现在屏幕上。在游戏开发过程中，这种形象以及将来的僵尸等可见的 2D 游戏对象称为**精灵**(sprite)，它们往往由一张图片构成，且各自带有原点。所谓原点，默认情况下其坐标为(0, 0)，相当于精灵的定位点。如果我们把一个精灵绘制于屏幕的特定坐标上，那么出现在该坐标上的正是该精灵的原点，见图 1.23。

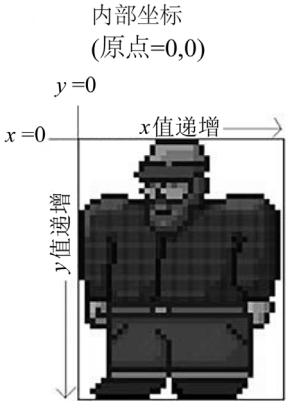

图 1.23　带有原点的精灵

因此，在前面展示的向屏幕中绘制形象的示意图中，虽然我们把图片画在了中心区域(960, 540)，它实际上却向右、向下偏了些许。

了解这些知识很重要，因为这能帮助我们理解用于绘制图片的坐标。

此外，不同玩家实际所用的屏幕分辨率五花八门，所以我们的游戏需要尽可能适应各种分辨率。在本书第三个项目中，我们将学习如何让游戏动态适配任意分辨率，但在这第一个项目中，我们固定使用 1920 像素×1080 像素的分辨率，并假定玩家的分辨率不低于此。

现在，我们开始编写第一段 C++代码并让它运行起来。

1.8　开始编写游戏

如果尚未打开 Visual Studio，请现在打开它。接下来，请在其主界面内的“**打开最近使用的内容**”(Recent)列表中单击“Timber”以打开此项目。

在窗口左侧的**解决方案资源管理器**中，定位到“**源文件**”(Source Files)文件夹内的 `Timber.cpp` 文件，其扩展名`.cpp`代表“C plus plus”(即 C++)，双击打开它。

接下来，我们需要编辑代码文本。虽然这些操作也能用任何文本编辑器或文档处理器来完成，但仍推荐在 Visual Studio 的代码编辑框中进行。这里，我们需要删除此 `Timber.cpp` 中的全部内容，并键入以下代码，我们随后会给出解释。虽然可以直接把本书的代码复制过去，但手动键入更令人印象深刻。

```
// 这是游戏的起点
int main()
{
  return 0;
}
```

这段简单的 C++程序是一个不错的开端，让我们逐行加以分析。

1.8.1　注释让代码变得更清晰

代码的第一行是这样的：

```
// 这是游戏的起点
```

每行以两个正斜杠开始的代码均为注释，编译器会忽略它们，所以这些内容完全不参与编译，仅为程序员留下了一些辅助消息。注释会因换行而终止，所以下一行的内容无论为何，均与此注释行无关。此外，还有另一种类型的注释，称为**多行**(multi-line)注释或 **C 风格**(C-style)注释，这种风格可以给出多于一行的注释内容，本章随后会给出例子。本书全部的注释有上百行之多，主要用于解释代码。

1.8.2　main 函数

下一行代码是这样的：

```
int main()
```

首先，这个 `int` 是一种**类型**(type)。C++有很多类型，对应着不同类型的数据。这里，`int` 表示一个**整数**(integer)，或称整型数字。请记住这个概念，我们还会遇到它。

随后是 `main()`，它后面还会有一段代码，且这段代码由一对大括号{}界定，二者分别称为起始大括号({)和终止大括号(})，而 `main` 可称为此段代码的名称。

换言之，这对大括号内的一切代码均属于 `main`。我们把这样的代码块称为**函数**(function)。

每个 C++程序都有一个 main 函数，它是整个程序**执行**(execution；又可称为**运行**，running)的开始。在后文中，我们将看到，游戏可以拥有多个代码文件，但其中只能有一个 `main` 函数，而且无论我们编写什么代码，程序永远会从 `main` 函数内的第一行代码开始运行。

`main` 后面的那一对小括号“()”看起来有点奇怪，但目前不必纠结于此，只需知道它代表函

数即可，第 4 章将从一个全新的角度来介绍函数的概念。

接下来，让我们仔细看一看 main 函数内部那唯一的一行代码。

1.8.3 代码的形式与语法概览

下面再整体看一看这个 main 函数：

```
int main()
{
     return 0;
}
```

可以看出，在 main 内部只有 return 0;这一行代码。这行代码与其他内容稍有不同，所以在讲述其具体功能之前，可以先研究其形式。经验证明，这能让我们写出的代码更清晰，也更易读懂。

首先注意，此语句向右缩进了一个 *Tab* 键，显然它与 main 并不是并列存在，而位于其内部。虽然目前这段代码读起来一目了然，但随着代码长度的增加，我们只能依赖于合适的代码缩进与留白来避免增加阅读代码的难度。

目前，这里只有一行代码，但任务一般需要由多行代码协作才得以完成。为便于管理，人们会把这些代码合称为一个**代码块**(block)，并通过缩进加以分隔。这里虽然只有一行代码，也可称为块。

接下来还请注意行尾的标点，即半角分号“;”。它用于通知编译器，这是当前代码指令的结束点，其后无论什么内容都属于下一条指令。我们把一个以分号结尾的指令称为**语句**(statement)。

这里需要提醒一点：编译器并不关心某语句末尾的分号与下一语句之间的分隔符，换行、空格或无分隔均可接受；但依然有必要换行，因为忽视分隔符会为阅读代码添加不必要的困难，甚至可能导致忘记添加分号。虽然阅读困难不影响编译，但缺失分号的语句属于语法错误，会让代码无法编译，让游戏无法运行。

至此，我们介绍了 main 函数的概念，并建议通过合适的缩进来对齐代码，也知道了每条语句的末尾要有一个分号。接下来便可以介绍这条 return 语句的真正功能了。

1.8.4 从函数返回一个值

从实际效果来看，这条 return 0;语句基本上没有完成任何工作，然而其背后的理念却非常重要。return 是 C++中的一个关键字，既可以后跟某数值，又能直接以分号结束，但无论有无数值，其皆为针对程序执行流程的一条跳转指令，用于令执行流程跳至(或者说回到)原先启动这个函数的代码。

一般而言，启动一个函数的代码属于另一个函数，被启动的函数在执行完毕后将回归执行那个启动它的函数。只是这里 main 函数是由操作系统启动的，所以在此 return 语句执行完毕后，main 函数退出，并返回操作系统，从而结束程序。

此外，return 后所带有的 0 同样会传给操作系统，其他数字亦然。

在编程语言中，我们把启动某函数的行为称作**调用**(call)该函数，而该函数结束后传回数值的操作称为**返回**(return)某个值。

本小节仅在于介绍函数的概念，不需要掌握函数的全部信息。在实现本书第一个项目的过程中，我们会详细探索这个概念。但在进入下一小节前，我们还需要最后讨论一下关于函数的知识：

还记得前面 `int main()` 中的 `int` 吗？其作用是告知编译器，`main` 函数所返回的值一定属于整型(整数或整型数字)，我们因此可以返回任意整数，如 0、1、999、6358 等，但如果尝试返回某个非整型数字，如 12.76，则将无法编译，游戏自然无法运行[1]。

函数所能返回的数据类型多种多样，而且 C++允许我们自己定义新的类型，只要编译器能够知晓其全部细节，便可由函数返回。

以上我们介绍了函数的一些背景知识，这有助于增进你对于本书后文内容的理解。

1.8.5　运行游戏

我们已经编写了一些代码，现在可以尝试运行游戏了，这可以通过单击 Visual Studio 快速启动栏上的“**本地 Windows 调试器**”(Local Windows Debugger)按钮来完成，见图 1.24，或者按下快捷键 F5。

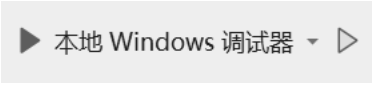

图 1.24　本地 Windows 调试器

运行前，需要把“**本地 Windows 调试器**”旁边的按钮设为“**x86**”，见图 1.25，这意味着我们的程序将是 32 位的，这也与我们下载的 SFML 相配。

图 1.25　确保你正运行在 x86 之下

运行后，你将看到一个黑色窗口一闪而过，而若此窗口停留于屏幕中，按下任意键即可关闭。这个窗口即为 C++控制台，会在运行每个控制台应用时打开，也是我们用于调试程序的工具，虽然暂时无此需要。目前我们的程序之所以一闪而过，是因为它在启动后会从 `main` 函数的第一行(也即 `return 0;`)语句开始运行，此后 `main` 退出，程序结束，返回操作系统，而这些操作都是非常迅速的。

至此，这个最简单的程序就正式完成了。接下来，我们会添加更多代码，启动游戏窗口。

1.9　使用 SFML 启动一个窗口

现在，让我们再添加一些代码。下面的代码将利用 SFML 库启动一个新的窗口，让 Timber!!! 游戏在其中运行。具体而言，此窗口宽 1920 像素、高 1080 像素，并处于全屏运行状态(没有边框，亦无标题)。

以下代码中高亮显示的部分是需要添加的新代码，请按照这里所展示的顺序，通过手动键入或复制粘贴的方式将其添加到你的代码中，同时试着想一想这些代码都做了什么，我们随后会逐行解释。

1　事实上，即使传回的不是整数，只要它能被转化为整数，代码便可编译，但一般会给出警告，这在第 2 章中将有所介绍。只有当代码传回的内容如果完全不能处理为整数(如传回一个汉字等)时，才无法编译。

```
// 在这里包含重要库
#include <SFML/Graphics.hpp>

// 通过此 using 指令让代码的键入更简单些
using namespace sf;

//这是游戏的起点
int main()
{
  // 创建 VideoMode 对象
  VideoMode vm(1920, 1080);

  // 为游戏创建窗口
  RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

  return 0;
}
```

接下来，我们将逐行解释这些代码。

1.9.1 引入 SFML 功能

我们添加的第一行代码是一条#include **指令**(directive)。

这条#include 指令引导 Visual Studio 在编译前，把另一个文件中的全部代码完整地包含(或称添加)到当前位置，从而让其他代码成为程序的一部分。这些代码一般由他人编写，我们实际上是在复用那些代码，避免重造轮子。这种把其他文件中的代码添加到我们代码中的过程称为**预处理**(preprocess)过程，是由所谓的**预处理器**(preprocessor)完成的。.hpp 扩展名意味着所包含的是个**头文件**(header file)。

因此，这条#include <SFML/Graphics.hpp>指令让预处理器寻找 SFML 文件夹内的 Graphics.hpp 文件，并将其全部内容填于此处，从而让我们得以使用一些 SFML 功能。后面我们会自主编写头文件，并通过#include 指令使用它，那时将进一步了解这个过程。

另外，Graphics.hpp 是 SFML 众多头文件中的一个，而通过在代码中用 include 指令引入 SFML 头文件，我们便能使用 SFML 库所提供的全部酷炫游戏功能，这也是本书中最常见的 include 指令。除了 SFML 头文件，我们还会使用#include 访问 **C++标准库**的头文件，这些头文件为我们提供了 C++语言自带的一些核心功能。

目前我们只需要知道，只有在添加这行代码后，我们才能使用 SFML 所提供的功能。

下一条语句是 using namespace sf;。接下来，我们会看一看它的作用。

1.9.2 OOP、类与对象

本书会一直讨论 OOP、类与对象，而本小节则会简要解释目前所遇到的现象。

正如前述，OOP 是面向对象编程的缩写。稍加展开地说，OOP 是一种编程范式，或是一种编程思想。在现今编程界中，OOP 在大多数情况下被认为是最好的一种专业编程思想，有些人甚至排斥其他编程思想——注意我说的是大多数，不乏有例外存在。

虽然 OOP 引入了许多编程概念，但其中最根本的概念在于**类**(class)与**对象**(object)。在编程时，我们总会尽力让代码具备可复用、易于维护、不易出错等特点，而把代码组织为类，一般能够实

现这些目标。本书第 6 章将介绍把代码组织为类的方法，但目前关于类，我们只需要知道，类的代码不会直接作为程序的一部分而存在，我们需要使用类来创建可复用的对象。例如，如果希望有 100 个僵尸 NPC(non-player character，**非玩家角色**)，那么我们可以先仔细地编写 Zombie 类，随后利用此类便可创建出任意数量的僵尸对象。每个僵尸对象的功能及其内部数据的类型是一模一样的，但每个僵尸对象是各自独立的实体。

接下来，我们会在不涉及 Zombie 类的具体代码的前提下演示僵尸对象的用法。例如，以下代码使用 Zombie 类新建了僵尸对象 z1：

```
Zombie z1;
```

现在，z1 对象便具备 Zombie 类的所有功能，成为可操作的 Zombie 对象。我们随即写下这样的代码[1]：

```
Zombie z2; Zombie z3; Zombie z4; Zombie z5;
```

至此，我们总计得到 5 个独立的 Zombie 实例[2]，但它们均由 Zombie 类创建。在继续介绍我们的代码之前，还需要再补充一点：这些僵尸对象都具有 Zombie 类的行为(类通过函数来定义对象的行为)，也各自具有数据，而这些数据则代表每个对象自身的血量、速度、位置、移动方向等信息。例如，假定 Zombie 的设计支持以下功能[3]，那么我们便能这样使用这些对象：

```
z1.attack(player); z2.growl(); z3.headExplode();
```

注意，目前这些假想的代码仅供演示，不要把它放入代码文件中，否则只会得到大量的错误。

在设计 Zombie 类的过程中，基本的指导原则是为此类提供控制自身行为以及数据的方法，并使这些方法尽可能服务于游戏的目标。例如，我们可能需要在创建僵尸对象时，便可设定其内部的数据，那么设计时便应该满足此要求。具体而言，假定我们需要在创建僵尸对象时为其命名，并指定运动速度(单位为 m/s)，那么只要仔细设计 Zombie 类，我们便能如此创建僵尸对象：

```
// 戴夫(Dave)在被感染前是奥林匹克百米短跑冠军
// 他每秒能跑 10 米
Zombie z1("Dave", 10);

// 吉尔(Gill)的双腿在被感染前就被吃掉了
// 她只能以 0.01m/s 的速度爬行
Zombie z2("Gill", .01);
```

虽然以上仅限于构思，但其重点在于说明类所具有的那种近乎无限的灵活度，而且一旦设计完毕，我们便能创建对象来使用类的代码。这也是 SFML 的用法。正是通过类与对象，我们才得以享受 SFML 所提供的强大功能。此外，我们也会编写自己的类，包括 Zombie 类。

下面，我们继续解释前面所添加的那些代码。

---

1 这一行代码中其实含有 4 条语句，根据上一节的介绍，这不是一种好习惯，但这些代码仅用于演示，不是实际的代码，因此勉强可以接受。

2 “实例”是面向对象编程中很常见的一个概念，目前可以把它理解为前面出现过的对象，而对象也是面向对象编程中的重要概念。本节随后会简要介绍面向对象编程，而详细说明则将出现在本书第 6 章中。此外，仍需说明，在介绍第 6 章前，“实例”字样还会多次出现。

3 指 Zombie 类设计有 attack、growl 和 headExplode 等函数，且 attack 函数还会接受 player 实例作为参数。本书第 4 章将详细介绍函数与函数的参数。这一行包含三条语句，同样不是什么好习惯。

1.9.3 命名空间与using语句

你可能已经猜到，VideoMode与RenderWindow是SFML提供的两个类，但在具体介绍这两个类之前，我们先要分析那条using语句的功能。

命名空间(namespace)是C++程序中一个可以定义和使用名称的区域，其中的每个名称(包括类名)都具有唯一的含义。无论何时，类均创建于某命名空间内，以便将其与同空间或其他空间中的类进行区别。下面使用VideoMode类进一步解释。

VideoMode类用于定义视频的模式，由宽度、高度与色彩深度(通道)组成，但在Windows等环境中，完全可能存在其他VideoMode结构，如若不加区分，则必然造成名称冲突，威胁编译过程；但只要这些名称属于不同的命名空间，借助于命名空间便可确保这些名称不会相互冲突。

VideoMode类所属的命名空间是sf，此空间由SFML定义，所以使用此类的完整语法如下(注意这里的省略号是在偷懒，不是正确的语法)：

```
sf::VideoMode ...
```

前面的那条using语句使我们能在代码中省掉sf::前缀，否则即便这第一个项目在完成后也将出现百余sf::。同时，这条using语句也有助于降低我们代码的复杂度，使其更加简短。

1.9.4 SFML VideoMode类与RenderWindow类

在main函数内，我们有两行注释和两行可执行代码。其中第一行可执行代码如下所示：

```
VideoMode vm(1920, 1080);
```

这行代码用VideoMode类创建了一个vm对象，并为其内部设定了1920和1080这两个数值[1]。这两个数值实际上代表玩家屏幕的分辨率，不是随意指定的。

下一行代码如下：

```
RenderWindow window(vm, "Timber!!!", Style::Fullscreen);
```

这里，window对象在运行代码后，将对应着屏幕内的一个窗口，而我们使用SFML所提供的RenderWindow类创建了它，并在随后的括号中为其额外提供了一些值，解释如下。

首先，在构建window对象时，需要使用对象vm。乍看起来，这种嵌套可能让人困惑，但别忘了，类具有无限的灵活性，可以任意按照其设计者的想法而行止，所以类完全可以包含其他类的实例。

对此，目前只需了解并接受用类创建可用对象的过程及其中所涉及的概念即可，不必完全理解背后的机制。如若实在无法接受，可以利用建筑师所绘制的设计图进行类比：家具、粮油和狗无法放入设计图内，但根据这张图可以造出一所房子(或更多房子)来容纳它们——类相当于设计图，而对象则相当于房子。

接下来的"Timber!!!"是用来命名窗口的，随后使用预定义的Style::Fullscreen值

1 这里的括号和前面main函数后的括号形式相同，本质上也是一次函数调用，但存在两处不同：第一，这里所调用函数的名称不是括号前面的vm，而是VideoMode类的构造函数，详见第6章(注意，前面假想代码中创建Zombie对象时没有括号，第6章中也会给出解释)；第二，1980与1080这两个数值是在调用构造函数时传入其中的两个参数，关于函数参数，请参见第4章。

让我们的 window 对象(即游戏主窗口)全屏运行。Style::Fullscreen 是 SFML 定义的一个不变的整数值，代表全屏运行的状态。这种固定不变的数值又称**常量**(constant)，而可以改变的量则称为**变量**(variable)。这两个概念会在下一章详细解释。

下面来看一看这段代码的运行效果。

1.9.5　运行游戏

现在又能运行游戏了，这时将能看到一个更大的黑色屏幕一闪而过，而那正是刚才所编写的 1920 像素×1080 像素的全屏窗口。我们的程序在启动后，首先运行 main 函数内第一行代码，创建出一个更大的游戏窗口，随即很快会再次 return 0;，之后立刻退出 main 函数而回到操作系统中。

下面我们将添加一些代码，以建立基本的游戏循环结构。这不仅是本书中每个游戏的基本结构，也几乎是每个游戏的基本结构。

1.10　游戏循环

作为一款游戏，我们希望其能够持续运行下去，直到玩家主动退出。同时，随着 Timber!!!项目的推进，我们希望能够清晰地标注出代码的不同区段，以便降低管理代码的难度。此外，我们不希望程序在中途退出，所以需要提供一种机制，让玩家在需要时能够顺利退出，否则游戏将永远运行。以上均为本节的目标。

为此，请将以下高亮显示的代码添加到既有代码中：

```
int main()
{
  // 创建 VideoMode 对象
  VideoMode vm(1920, 1080);

  // 为游戏创建窗口
  RenderWindow window(vm,"Timber!!!", Style::Fullscreen);

  while (window.isOpen())
  {
    /*
    ****************************************
    处理玩家输入
    ****************************************
    */
    if (Keyboard::isKeyPressed(Keyboard::Escape))
    {
      window.close();
    }

    /*
    ****************************************
    更新场景
    ****************************************
    */
```

```
    /*
    ****************************************
    绘制场景
    ****************************************
    */

    // 清空上一帧内的全部内容
    window.clear();

    // 在这里绘制我们的游戏场景
    // 展示所绘制的全部内容
    window.display();
  }

  return 0;
}
```

下面我们将逐一解释这些新代码。

1.10.1 while 循环

在新代码中，我们首先可以看到以下结构：

```
while (window.isOpen())
{
```

新代码的最后是个终止大括号“}”，这便是所谓的 while 循环结构。这种结构始于那行 while 代码，包括随后的起始大括号“{”，至终止大括号“}”而结束，其效果为反复执行这对大括号内的所有代码，甚至可以无限执行下去。

while 行中有两对小括号，值得一提：

```
while (window.isOpen())
```

本书第 4 章会介绍循环及循环条件，那时才有关于此代码的详细解释，目前我们只需要知道，循环体内的代码会重复执行，而关闭运行游戏所弹出的窗口意味着关闭了 window 对象，这也将终止 while 循环，随即开始执行循环终止大括号后面的语句。我们很快会介绍关闭窗口的方法。

下一条语句回归 return 0;，可结束游戏程序。

现在我们知道，前面那段 while 循环将周而复始地执行其中的所有代码，直到我们关闭 window 对象。

1.10.2 C 风格代码注释

while 循环内首先映入眼帘的是仿佛 ASCII 艺术般的文字：

```
/*
**************************************
处理玩家输入
**************************************
*/
```

ASCII 艺术利用计算机文本来绘图，是一种小众但有趣的工作。更多相关信息请参见 https://en.wikipedia.org/wiki/ASCII_art。

这段代码同样是注释，但属于另一种注释，称为**C 风格**注释。这种注释由“/*”开始，以“*/”结束，其间的一切信息均不参与编译。这种注释风格能够更详尽地说明信息，故可用于详细描述每部分代码的意义，例如，这段注释让你立刻判断出，接下来的代码块将负责处理玩家的输入。

跳过几行代码后即来到另一段 C 风格注释，说明随后将更新场景。

再下一段 C 风格注释标注的是负责绘制图形的部分。

下面让我们分别讨论这些代码段的意义。

1.10.3　输入、更新、绘制、重复

麻雀虽小，五脏俱全。虽然目前我们仅仅使用了最简单的游戏循环，但其中仍具备游戏循环的四大典型阶段。

(1) 获取玩家的输入(即使暂时没有输入，也需要提供处理代码)。

(2) 根据游戏 AI、物理学原理或玩家输入等因素更新场景。

(3) 绘制当前场景。

(4) 用足够高的频率重复执行以上步骤，从而实现动画效果以及流畅的交互式游戏世界。

接下来，我们会依次分析游戏循环的具体代码。

1.10.4　检测按键动作

首先是注释着“处理玩家输入”的代码：

```
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
  window.close();
}
```

这段代码检查当前有没有按下键盘上的 *Esc* 键，而按下此键后，高亮显示的代码将负责关闭 `window` 对象。此后，当 `while` 循环体准备继续执行时，便可发现需要退出循环，从而跳转到 `while` 循环体之后，进而退出游戏。另外，这段代码中还使用了一条 `if` 语句，这会在第 2 章中讨论。

1.10.5　清空并绘制场景

现在，更新场景部分空空如也，所以我们直接跳到绘制场景的部分。这里首先需要通过以下代码来擦除上一帧所绘制的内容：

```
window.clear();
```

接下来原本需要绘制游戏中的每个对象，然而目前无物可绘制。

下一行代码如下：

```
window.display();
```

此行代码背后的机制略显复杂。所有的游戏对象均绘于一个可供呈现的隐藏图层之内，而屏

幕所呈现的是另一个图层。这行代码负责交换这两个图层，从而呈现出原本隐藏但绘有新内容的图层。现在，当看到新图层时，其中已经绘有全部内容，所以玩家永远看不到绘制的过程；这种机制还能保证所绘场景的完备性，有助于避免出现画面**撕裂**(tearing)问题[1]。这便是所谓的**双重缓冲区**(double buffering)技术。

最后请注意，清屏以及诸般绘制工作均由 `window` 对象完成，而 `window` 对象又是通过 SFML `RenderWindow` 类创建的。

1.10.6 运行游戏

此时运行游戏，将得到一个空白的全屏窗口。与前面一闪而过的窗口不同的是，这个窗口永不关闭，除非按下 *Esc* 键。

这是很好的进展：我们得到了一个可执行程序，它会打开一个常亮的窗口，直到按下 *Esc* 键才会退出。现在，我们可以开始绘制游戏的背景图片了。

1.11 绘制游戏背景

接下来，我们将为游戏添加图像，这是通过创建**精灵**(sprite)来实现的。我们首先选择绘制游戏的背景图。根据上一节的介绍，这应该在清屏与显示(指调换图层)之间完成。

1.11.1 让精灵与纹理图协同工作

SFML 的 `RenderWindow` 类让我们得以创建 `window` 对象，并提供游戏窗口所需的众多功能，但绘制精灵显然不在此列。

为此，我们需要求助于 SFML 内负责向屏幕绘制精灵的两个类：其一是名副其实的 `Sprite` 类，其二则是 `Texture` 类，其中“Texture”可译为“纹理”，是保存在显存中的一张图片，而显存则是 **GPU**(Graphics Processing Unit，**图像处理单元**)内部的存储结构。

`Sprite` 对象只有在与 `Texture` 对象配合之后才能显示为屏幕图片，具体操作参见下面高亮显示的代码，请将其添加到你的代码中，同时可以试着猜测其含义。

```
int main()
{
  // 创建 VideoMode 对象
  VideoMode vm(1920, 1080);

  // 为游戏创建窗口
  RenderWindow window(vm,"Timber!!!", Style::Fullscreen);

  // 创建 Texture 对象以便在 GPU 中保存图片
  Texture textureBackground;
  // 向此对象中加载图片
  textureBackground.loadFromFile("graphics/background.png");
```

1 画面撕裂问题是由显卡内部图层更新频率与显示器刷新率不匹配造成的，表现为显示器中的画面是由第一幅图的上半部分与第二幅图的下半部分拼接而成，根据具体情况，还可能会出现更多图片拼接甚至左右拼接的效果。很多游戏提供垂直同步选项来避免画面撕裂问题，这里的双重缓冲区同样是一种不错的辅助方式，方法不一。

```
// 创建 Sprite 对象
Sprite spriteBackground;
// 让 Sprite 对象与 Texture 对象建立关联
spriteBackground.setTexture(textureBackground);
// 让 spriteBackground 占据整个屏幕
spriteBackground.setPosition(0, 0);

while (window.isOpen())
{
```

注意，这段代码位于 while 循环之前，因为它们只需要运行一次，其背后的原因随后会有介绍，目前则继续逐行解释代码。这里的第一步是利用 SFML Texture 类创建对象 textureBackground：

```
Texture textureBackground;
```

其次，我们使用此对象加载 graphics 文件夹内的背景图，代码如下：

```
textureBackground.loadFromFile("graphics/background.png");
```

我们仅需要指定 graphics/background.png，因为这里使用的是相对于 Visual Studio **工作目录**(working directory，又称**工作路径**)的路径(即相对路径)，其中的资源早已就绪。

接着，我们用 SFML Sprite 类创建 spriteBackground 对象：

```
Sprite spriteBackground;
```

随即让 Texture 对象 textureBackground 与 Sprite 对象 spriteBackground 建立关联关系：

```
spriteBackground.setTexture(textureBackground);
```

最后，令 spriteBackground 对象在 window 中的位置坐标为(0, 0)，即左上角：

```
spriteBackground.setPosition(0, 0);
```

不难发现，background.png 图像宽 1920 像素，高 1080 像素，这与 vm 对象的设定相符，所以这张图恰好填满整个窗口。另外仍请注意，此时仍未显示出图片，仅仅设定了将来的显示位置。

至此，spriteBackground 对象可用于绘制背景图片了，但在此之前，让我们再看一看 SFML 选择这种绘制机制背后的逻辑。

诚然，SFML 借助纹理来绘制精灵的过程似乎有些多余，这可能会让你感到困惑，但这是由显卡及 OpenGL 的工作机制决定的。纹理会占据显存，但加载后可以一直存在，直至主动释放或程序退出，而显存是一种有限的资源。同时，将图片加载到显存的操作是非常慢的，虽然不至于慢得让人们能察觉逐渐加载的过程，也不会在加载时让 PC 出现明显的卡顿现象，但也慢得不能在游戏循环的每一帧中均加载一次。所以，有必要将加载纹理的代码(这里指操作 textureBackground 对象的代码)放至游戏循环之外。

后文将介绍，我们是通过精灵来移动图片的。每个 Texture 对象都位于 GPU 内部，虽然不会在加载后立刻绘出，但只要其所关联的 Sprite 对象提供绘制位置，它便可呈现在屏幕上。在后面的项目中，我们将让多个 Sprite 对象使用同一个 Texture 对象，这显然提升了 GPU 内存

的使用效率。

绘制的全过程可以总结如下：

- 纹理图加载到 GPU 内部的过程很慢。
- 访问位于 GPU 内的纹理是非常快的。
- 可以把 Texture 对象关联到 Sprite 对象上。
- Sprite 对象可以改变其位置和朝向角度(这经常出现在更新场景部分)。
- Sprite 对象在与 Texture 对象建立关联后便可绘制出来(这经常出现在绘制场景部分)。

所以，只需在 window 对象所提供的双重缓冲区机制内绘制出此 Sprite 对象(spriteBackground)，我们便能首次实际看到图像。

1.11.2 背景精灵与双重缓冲区

最后，我们需要在游戏循环内合适的位置上绘制精灵及其纹理。

注意，我在给出属于同一代码块的代码时并未严格考虑缩进，否则将增加书中换行的次数，干扰正文文本排版。但代码是暗含缩进的，可以参考本书下载资源包中的代码文件来了解完整的缩进形式。

请添加以下高亮显示的代码：

```
/*
****************************************
绘制场景
****************************************
*/
// 清空上一帧内的全部内容
window.clear();

// 在这里绘制我们的游戏场景
window.draw(spriteBackground);

// 展示所绘制的全部内容
window.display();
```

这里我们仅在清屏后、绘制新场景前添加了一行代码，负责将 spriteBackground 绘制到 window 内。

我们现在知道了精灵是什么，也知道了可以让一个纹理图片与之关联并绘制到屏幕上。现在游戏再次做好了运行准备，让我们来看一看这些新代码的效果。

1.11.3 运行游戏

此刻运行程序后，我们将首次找到正在制作一个真正游戏的感觉，如图 1.26 所示。

图 1.26　运行游戏

虽然这个游戏目前的状态还拿不到年度游戏的荣誉，但至少我们已经迈出了第一步！

接下来，我们会介绍 C++编程时经常遇到的一些问题。

1.12　处理错误

每一个项目都注定会遇到困难，这是无法回避的事实。但另一方面，问题越棘手，在成功解决后的满足感就越强。试想象，编程时突然遇到了一个问题，但在数小时的挣扎之后终于解决，使之变为一个新的游戏功能，这会让你感觉非常愉快。如果不经历烦恼挣扎，功能实现后反而可能感到不够精彩。

读到这里，虽然尚未接触晦涩难懂的 C++技能，但你依旧可能经历过一些挣扎。此时，请保持冷静，坚信自己能渡过难关，然后继续学习下去。

记住，无论遇到什么问题，你都不大可能是为其所困的第一人，这意味着在线搜索往往是解决编程问题的一种实用而高效的手段。为此，可以用一个简单的句子甚至短语来描述你的问题或遇到的错误，并在 **Google** 或 **ChatGPT** 中搜索，一般立刻便能找到解决的方法，因为经常有人先于你解决了问题。

另外，如果仍然未能顺利运行本章的代码，那么可以从以下角度排查原因。

1.12.1　配置错误

本章最可能遇到的问题是**配置错误**(configuration error)。我们在前面搭建了 Visual Studio 开发环境，并为项目进行了诸如 SFML 等配置，其间显然涉及很多精细操作，大批文件名、文件夹名均须准确设置，一个字符之差亦可能引发若干错误，不少错误信息甚至完全没有解释出错的实际原因。

本章 1.9 节中的以纯黑屏为最终效果的项目基本上是空白项目，如果看不到黑屏，那么可以考虑重新开始。请确保所有的文件与文件夹都按照你自己的安装环境配置完成，并尝试运行那十

余行代码(包括注释)。此时，一闪而过的黑色窗口即为配置成功的证明[1]。

1.12.2 编译错误

编译错误(compilation error)应该是最常见的错误，这种错误一般意味着写错了代码。为此，请详加检查，并确保你的代码与本书示例代码严格相同，尤其要保证行尾的分号以及类名与对象名中的大小写格式是正确的。如果无论如何都无法消除错误，可以从下载的代码包中复制代码。

尽管本书中不排除有拼写错误的存在，但下载包中的代码文件来自实际可以运行的项目，绝对能通过编译！

1.12.3 链接错误

链接错误(link error)最可能由缺失 SFML 的`.dll`文件引起。你有没有把全部文件复制到你的项目文件夹中？

1.12.4 缺陷

缺陷(bug)指代码工作得与预期不符。修复缺陷的过程称为**调试**(debug)，它有些枯燥，却也能带来快感：所碾压的缺陷越多，你的游戏质量便会越高，也越会对自己当日的工作感到满意。这里只强调一点：解决缺陷的技巧在于提早发现它们，而且越早越好！缺陷发现得越早，其可能的原因在你脑海中就越清晰。所以，我建议只要添加了新内容，就立刻编译和运行，本书便是在完成每阶段的开发后立刻尝试运行。

1.13 本章小结

本章虽然是 C++入门教程的第 1 章，却也富有挑战性。例如，必须承认，配置 IDE 以使用 C++库的过程很是烦琐，很多初次接触编程的人更是难以理解类与对象的概念。

但既然已经读到这里，不妨进一步学习 C++、SFML 与游戏编程的技能。随着阅读继续，我们实现的游戏功能会越来越强大，学到的 C++知识也会越来越丰富，并将深入探索函数、类与对象等编程理念，围绕着它们的神秘面纱终将为我们所揭去。

而且在本章中，我们同样有所收获，例如，我们逐行分析了一个带有`main`函数的 C++示例程序，学习了其中每行代码的意义。我们还构建了一个简单的游戏循环，其功能在于监听玩家的输入，并在屏幕上绘制一个精灵(及其配套的纹理图片)。

在下一章中，我们将学到更多 C++知识以绘制更多的精灵，并让它们动起来。

---

1 准确地说，本项目在1.8 节结束时仅由5 行代码与注释组成，此时的运行状态也是纯黑色的窗口，但那5 行代码不使用SFML，在进行 1.3、1.4 节的全部配置之前亦可编译和运行，其中甚至可能存在未发现的配置错误(例如，写错了库名)，但这 5 行代码也不失为一种排查的方案。但是，1.8 节中的黑色窗口很少存在意外，而一旦出现意外，甚至可能是 Visual Studio 本身的问题，如安装失败等，此时配置项目的帮助有限。

1.14　常见问题

以下是一些你可能感到困惑的问题：

问题 1：本章的内容已经让我挣扎不已，这是不是意味着我不太适合学习编程？

回答：搭建开发环境以及建立 OOP 的概念可能是本书中难度最大的事情。如果你的游戏能够正常工作(指能够呈现出背景图)，那么完全可以继续读下去。

问题 2：我完全无法承受那些关于 OOP、类和对象的讨论，这甚至可能要毁掉我的学习体验。

回答：不必担心。接下来，我们会不时提及 OOP、类和对象这些概念，这样当你读到第 6 章时，很可能已经不再抵触它们，从而能够在该章中深度探索 OOP 思想。目前，你只需了解 SFML 已编写出一整套实用类，我们会通过这些类创建可复用的对象，以此发挥 SFML 的强大功能。熟知 OOP 后，你甚至会感到自己充满力量。

问题 3：我实在理解不了函数的概念。

回答：没关系。我们会经常接触它，第 4 章更会深入展开研究。目前你只需要知道，调用一个函数便会执行其中的代码，而执行完毕后(执行完一条 `return` 语句)，程序会回到调用此函数(代码)的位置。

第2章

变量、运算符与决策——让精灵动起来

本章将完成许多绘制任务，为背景图上的云朵以及前景中的蜜蜂分别赋予随机移动的能力(包括高度随机与速度随机)，而这需要更多C++知识。我们将学习如何利用变量存储数据，也会学习如何通过运算符操作这些变量，以及根据变量的值制定决策，进而有选择地执行若干分支路径中的一种。只要结合所有这些知识与前面学到的关于**SFML**(Simple and Fast Multimedia Library)内的`Sprite`与`Texture`类的知识，我们便能实现云朵和蜜蜂的动画效果。

本章将涵盖以下主题：

- 系统学习C++变量
- 熟悉操作变量的方法
- 添加云朵和蜜蜂，并添加可供玩家砍伐的树
- 随机数
- 用`if`与`else`制定决策
- 计时功能
- 移动云朵与蜜蜂

2.1 系统学习C++变量

变量(variable)是C++游戏保存并操作数值/数据的途径：为了获取玩家当前的生命值，需要变量；为了获取当前攻势中剩余僵尸的数量，需要变量；为了记录获得特定高分的玩家，需要变量；为了获取当前游戏已经结束或仍在运行的状态，依旧需要变量。

变量是某**内存位置**(location in memory)的命名标识符。例如，我们可以定义一个`numberOfZombies`变量，此变量将指代内存中的某个位置，其中保存了当前这次攻势中所剩余僵尸的数量。

计算机系统访问内存位置的机制很复杂，所以各种编程语言使用了变量这种对人类友好的机制来管理相应内存位置上的数据。事实上，这也正是编程语言的真正任务，即用一种对人类友好的方法来管理复杂的系统。不同语言的效率和友好度有所差异。C++一直以高效著称，并且经过半个世纪的发展，这门语言对用户越来越友好。

C++是由 Bjarne Stroustrup 在 20 世纪 80 年代早期创建的[1]，他为了向原始的 C 语言增加面向对象编程机制而开发了 C++语言，从而让代码更高效且更易于管理。多年来，C++经历了多次修订与改进。

本节伊始蜻蜓点水般地介绍了变量，却已暗示了变量需要存在多种类型。事实上，C++的变量有很多类型，下面我们便会看到本书中用得最频繁的几种。

2.1.1 变量类型

关于 C++变量及其内容的讨论可以轻松地占据一整章的内容，这也是其他众多出版物的一贯做法，但读者选择本书更可能是为了尽快完成游戏制作，所以这里没有墨守成规，而是用表 2.1 来列举本书中最常用的几种类型及其具体用法。

表 2.1 变量的类型

类型	示例值	解释
`int`	`-42`、`0`、`1`、`9826`	整型数字
`float`	`-1.26f`、`5.8999996f`、`10128.3f`	单精度浮点值最多包含 7 位有效数字
`double`	`925.83920655234`、`1859876.94872535`	双精度浮点值最多包含 15 位有效数字
`char`	`a`、`b`、`c`、`1`、`2`、`3`(总计 128 个符号，包括?、~、#等)	即 ASCII 码表中的所有字符(参见关于变量的下一条提示)
`bool`	`true` 或 `false`	`bool` 代表**布尔型**(Boolean)，只有 `true` 与 `false` 两种值(即布尔值)
`string`	`Hello Everyone! I am a String.`	任何文本，可以短至一个字母，长到一整本书的内容

C++属于**强类型**(strongly typed)语言，是一种会严格规定变量类型并限制隐式类型转换的语言系统。对于强类型语言，涉及不同数据类型的操作一般要求显式转换类型，否则可能带来编译错误。这种强制要求有助于降低游戏中的行为不合预期的概率，因为编译器或解释器[2]会保证变量的用法符合其类型。

因此，编译器必须知晓某变量的具体类型，以便为其分配合适的内存空间，且编译器在知晓某变量的类型后，还会检查其使用方式是否存在错误，例如，不能让字符串除以布尔型变量。为每个变量选取最好、最合适的类型是一种良好的编程习惯，但在实践中，人们倾向于过度提升变量的精度，例如，`float` 变量足以存储最多 5 位有效数字的浮点数值，可惜使用 `double` 也不会让编译器给出抱怨。然而，一旦代码将 `float` 或 `double` 值存入 `int` 变量，编译器便需要改变(或称转换)此浮点值，以便存入，这将让实际所存入的值异于原值，编译器可能会因此而给出警告。本书为每个变量所选择的类型基本上是最合适的，但同样不排斥主动进行类型转换。

表 2.1 中还有个细节值得一提，那就是每个 `float` 值的字母后缀 `f`。此后缀负责告知编译器，

1 1979 年，Bjarne Stroustrup 在 C 语言的基础上完成了一门新的编程语言，称为“C with Classes”，随后在 1983 年，这门语言正式更名为“C++”，他也因此被称为“C++之父”。本章后面还有一条相关的趣闻，可以参考。

2 解释器同属执行 C++代码的工具程序，性质上类似于编译器，但与编译器先将代码编译为可执行程序再执行所不同的是，解释器略过编译过程而直接逐行执行 C++程序，有时也称解释执行。解释执行具有巨大的灵活性，其效率却远低于编译执行。有些程序语言(如 MATLAB、Python 等)更习惯于解释执行，C++虽然不排斥这种方式，但仍以编译执行为主。

相应值的类型是 float 而非 double，因为不带有 f 的浮点值默认属于 double 类型。更多相关细节可参见下一条关于变量的提示。

用户定义类型

与前面诸般类型相比，用户定义类型更加高级。第 1 章曾简单讨论过类(与对象)，而 C++中的用户定义类型正是类及枚举量。很快，我们将会用独立的一个或多个文件来组织代码，进而声明、初始化并使用自主设计的类。具体而言，自主定义(或称创建)类的工作将在第 6 章进行，但在那之前，我们首先会在第 4 章学习枚举量，这种机制可用于自定义一些类型，如僵尸、能力增幅、外星飞船的具体种类等，学会这些类型后有助于理解类的概念。但现在，让我们回到 C++内置类型，它们通常称为**基本类型**(fundamental type)，因为这些类型代表着我们前面所见的非常基础的值。

2.1.2　声明并初始化变量

现在，我们知道变量可用来存储游戏赖以运行的数据值，例如，变量可以代表玩家当前还有多少条命，也能代表玩家的名字。我们也知道，变量所代表的数据可以有非常多的类型，如 int、float、bool 乃至自定义类型等。但是，我们暂未介绍变量的具体用法。

创建并准备一个新的变量分为两个阶段，分别是**声明**(declaration)和**初始化**(initialization)。下面将依次讨论。

1. 声明变量

C++可以这样声明变量：

```
// 玩家得分几何?
int playerScore;
// 玩家名的首字母是什么?
char playerInitial;
// 圆周率是多少?
float valuePi;
// 玩家是死是活?
bool isAlive;
```

在上面这段代码中，我们先后声明了 int 型变量 playerScore、char 型变量 playerInitial、float 型变量 valuePi 与 bool 型变量 isAlive。如果尚未完全掌握这些类型，请复习表 2.1。有了这些声明[1]，便意味着已经在内存中为这些变量预留(即分配)出合适的空间，可供保存并操作相应位置上的值，但那些空间上目前暂未保存任何数据(或者说那些数据没有意义)。为此，我们需要继续学习。

2. 初始化变量

声明了意义明确的变量后，便可用合适的数值来初始化它们：

```
playerScore = 0;
playerInitial = 'J';
```

1　准确地说，这里的“声明”应作“定义”，因为只有在定义变量时才会分配内存，而严格意义上的变量声明却不会。事实上，非常多的英文资料并不严格区分这两种行为。

```
valuePi = 3.141f;
isAlive = true;
```

执行上面的代码后，计算机内存中便有了实际的数据。这4个变量所保存的值依次是0、大写字母J、单精度浮点数值3.141与布尔值true。

3. 声明初始化

事实上，变量声明步骤与初始化步骤完全可以合二为一。如果变量初始值已知，则可以这样写：

```
int playerScore = 0;
char playerInitial = 'J';
float valuePi = 3.141f;
bool isAlive = true;
```

如果只能在程序执行过程中才能确定变量的值，那么上一小节中的写法自然更合适。虽然在C++中，这两种写法都是正确的，但总有一种更适合你的游戏。

如果希望查看所有C++类型的完整列表，可以访问http://www.tutorialspoint.com/cplusplus/cpp_data_types.htm。如果希望更深入地研讨float、double及f后缀，可以读一读http://www.cplusplus.com/forum/beginner/24483/。如果希望获取关于ASCII字符码的更多信息，可以访问http://www.cplusplus.com/doc/ascii/。

4. 常量

有时，我们需要保证某个值无法改动，这可以通过使用const关键字声明并初始化一个**常量**(constant)来完成。例如，圆周率永远不变，所以此量更适合以常量形式出现：

```
const float PI = 3.141f;
const int NUMBER_OF_ENEMIES = 2000;
```

这段代码保证常量PI的值不会在程序执行期间发生任何变化，常量NUMBER_OF_ENEMIES亦然。声明常量通常使用与声明变量不同的命名格式，例如，本书选择以下画线分隔的全大写词组，而不使用命名变量的骆驼风格[1]。

需要明确的是，这里声称某常量不被修改，应当理解为其不会被程序执行逻辑改动，但在初始化时，其初始值可以被程序员任意修改，只是此后不能通过写代码来改动常量的值：

```
// const int PLANETS_IN_SOLAR_SYSTEM = 9;
// 啊不对！冥王星在2006年被重新归类为矮行星
const int PLANETS_IN_SOLAR_SYSTEM = 8;
```

在第4章中，我们将看到一些常量在实际中的应用，但关于变量初始化，还需要讲解一个知识点。

5. 一致性初始化

一致性初始化(Uniform Initialization)又称列表初始化，是初始化变量的一种新格式。2011年发

1 骆驼风格(camel case)是一种变量命名规则，其要点是，在多个英文单词组成的名称中，各单词仅首字母大写(名称的首字母则可小写)，且单词间无需连接符，参见本小节前面介绍的各变量名。考虑驼峰的形象便能够理解这种机制得名的原因。

布了一次对 C++语言标准的重大升级，随后将此新标准称为 C++11，而一致性初始化正是 C++11 所引入的。这种机制统一了初始化变量与用户定义类型的对象的格式，允许使用大括号(即“{}”)来初始化变量，而界定 main 函数范围的也是这对大括号。以下代码使用这种形式来初始化前面几个变量：

```
int playerScore{0};
char playerInitial{'J'};
float valuePi{3.141f};
bool isAlive{true};
```

这段代码对每个变量用一致性初始化语法替换了赋值运算符“=”。这种语法是现代 C++中的正式标准，并且在现代商业 API 中最常见。虽然出于一些深层原因，一致性初始化相比于“传统”语法更不易出错，但本书采用的还是后者：

```
int playerScore = 0;
```

这两种语法都是有效的。本书因考虑到“{}”语法可能会在其他 C++资料中出现而介绍一致性初始化，第 6 章在讨论类时则会主动回归这种语法。如果有意，你完全可以主动将本书全部代码改用一致性初始化，这并不复杂。此外，虽然传统语法对初学者更友好，但就职于大型企业的员工很可能会使用一致性初始化。

6. 声明并初始化用户定义类型

我们已经学习过声明并初始化一些 SFML 类型的例子。正是因为创建(或称定义)类型(即 C++类)的方法非常灵活，所以声明并初始化它们的方法同样是多种多样的。上一章中定义并初始化了一些用户定义类型，下面对比给出若干提示。

创建 VideoMode 类型的一个对象 vm，并以两个 int 型值 1920 与 1080 来初始化它：

```
// 创建 VideoMode 对象
VideoMode vm(1920, 1080);
```

创建一个 Texture 对象 textureBackground，但不做任何(显式)初始化操作：

```
// 创建 Texture 对象以在 GPU 中保存图片
Texture textureBackground;
```

注意，尽管我们没有指定任何用于初始化 textureBackground 的数值，但其内部很可能已经设定了若干变量的值。一般而言，某对象是否需要或能否接受初始值，完全取决于对应类的具体代码，所以这其中具备相当的灵活性。这进而意味着自主编写类可能很复杂，但也意味着自定义类型(即 C++类)有足够的设计自由度，所以期望这些类恰好符合我们制作游戏的要求绝非痴人说梦。一旦把 C++这种强大的灵活性与 SFML 类相结合，我们的游戏便具备近乎无限的潜力！

本章随后会演示 SFML 所提供的一些用户定义类型，后续章节中还会展示更多用户定义类型。第 6 章在实现 Pong 游戏时更会设计并编写自定义的类型。

2.2　熟悉操作变量的方法

至此，我们已经知晓变量到底是什么、有哪些主要类型以及声明和初始化变量的方法，但仍

未学习发挥变量真正功能的方法。为此，我们需要操作变量，例如，执行加法、减法、乘除法等，尤其是条件判断。

首先，我们将尝试学习如何操作变量，随后再研究如何以及为什么要进行条件判断。接下来，请带着这些目标来学习 C++的算术运算符及赋值运算符[1]。

2.2.1 C++算术运算符与赋值运算符

为了操作变量，C++提供了大量的**算术运算符**(arithmetic operator)及相应的**赋值运算符**(assignment operator)。大多数运算符的用法一目了然，其余运算符解释起来也不难。下面请先研究表 2.2 中的算术运算符，它们与随后表 2.3 中的赋值运算符是本书中最常用的运算符。

表 2.2 算术运算符

算术运算符	名称	功能
+	加法运算符	把两个变量的值或两个值加起来
-	减法运算符	从一个变量的值或一个值中减去另一个变量的值或另一个值
*	乘法运算符	把变量的值或数值相乘
/	除法运算符	变量的值或数值相除
%	**取余**(modulo)运算符	在用一个数值或一个变量的值除以另一个值或变量之后，取此运算的余数

表 2.3 是赋值运算符表。

表 2.3 赋值运算符

赋值运算符	解释
=	我们曾介绍过此运算符；它是赋值运算符，用于设置变量的值或初始化变量
+=	把右侧的值加到左侧变量之上
-=	把右侧的值从左侧变量内减去
*=	把右侧的值乘到左侧变量之上
/=	用等号右侧的值除以左侧变量，让左侧变量取其商
++	**自增**(increment)运算符让一个变量自加 1
--	**自减**(decrement)运算符让一个变量减去 1
<=>	宇宙飞船运算符是 C++20 标准[2]向 C++引入的一种新型运算符，用于进行三路比较操作，我们会在下一个项目中使用

从技术角度来说，表 2.3 中除=、--、++之外的 5 种运算符又称**复合赋值运算符**(compound assignment operator)，因为它们由多种运算符组合而成。

1 运算符(operator)，简称“算符”或“符”(例如，有时会简称为“赋值符”等)，是C++中一个很有趣但也很重要的概念。运算符需要操作数(operand)，可根据所要求的数量分为一元运算符、二元运算符和三元运算符等。运算符的本质为一个函数，操作数则相当于函数的参数。此外，第3章中有一条提示与运算符有关，可以参考学习。

2 C++11 是在 2011 年发布的，C++20 是 2020 年所发布的C++语言新标准。

至此，我们已学习了大量的算术运算符和赋值运算符，接下来便能学习如何通过组合运算符、变量与值来构成**表达式**并操作变量了。

2.2.2　使用表达式完成工作

表达式(expression)是变量、运算符与值的组合，这与英语中的表达式是单词与标点符号的组合是类似的。使用表达式能够得到一个结果。在条件判断中，同样可以使用表达式，而这些判断将决定接下来所运行的代码。

1. 赋值

首先来看能够在我们的游戏代码中见到的一些简单表达式：

```
// 玩家取得新的高分纪录
hiScore = score;
```

或者：

```
// 把 score 设定为 100
score = 100;
```

在第一行代码中，我们把 score 变量所保存的数值赋给 hiScore 变量，按照前面的描述，此后 hiScore 变量将持有 score 的值，无论这个值具体是多少。这行代码可能出现在游戏结束后，玩家的得分比某些乃至全部高分纪录更高的时候。具体而言，在执行语句 hiScore = score; 后，score 的值可能需要重置为 0，以记录下一局的分数，而 hiScore 则仍旧持有 score 变量原先所存储的分数。当然，如果每局游戏都要执行此操作，可能会记录下错误的最高分数，这要求我们进行条件判断以及数值比较。这里先继续介绍赋值的知识，随后再介绍进行判断与比较的方法。

下面看一看与赋值运算符连用的加法运算符：

```
// 击中外星人会得分
score = aliensShot + wavesCleared;
```

或者(注意，变量完全可以同时出现在运算符两侧)：

```
// 直接加上 100 分
score = score + 100;
```

前面第一行代码把 aliensShot 与 wavesCleared 变量的值之和赋予 score 变量，第二行代码则把 score 当前的值加上 100，然后再将结果赋回给它。此例还有个更实用的变体：

```
score = score + pointsPerAlien;
```

此时，pointsPerAlien 的值将加到 score 的既有值上，即用变量代替具体的数值。在运算符两侧使用同一变量的操作其实非常常见，请再次研究这些代码，务必理解其功能。

下面，让我们看一看与赋值运算符连用的减法运算符。下面的代码用减法运算符左边的值减去它右边的值，而减法运算符经常与赋值运算符连用，例如，

```
// 哦好吧，丢了一条命
lives = lives - 1;
```

或者：

```
// 游戏结束时还剩下多少外星人?
aliensRemaining = aliensTotal - aliensDestroyed;
```

以下是除法运算符的一种用法，其中用除法运算符左边的值除以右边的值，并再次与赋值运算符连用：

```
// 用 swordLevel 压低剩余生命值
hitPoints = hitPoints / swordLevel;
```

还有：

```
// 回收 block 时需要付出一定的代价
recycledValueOfBlock = originalValue / 1.1f;
```

这里 recycledValueOfBlock 变量需要为 float 类型，才能保存这种运算的结果，好在理解其语法很简单。如果读到这里，仿佛有种正在教授学龄儿童算术运算的感觉，那你很可能已经掌握了其中的精髓。在进入下一话题前，关于赋值运算符还有最后一个例子需要演示，我们可以这样使用乘法运算符：

```
// answer 显然是 100
answer = 10 * 10;
```

或者：

```
// biggerAnswer 显然是 1000
biggerAnswer = 10 * 10 * 10;
```

这些代码基本上不言自明：我们进行了两个数的乘法，然后是 3 个 10 相乘，并分别把乘积赋给 answer 和 biggerAnswer 两个变量。

2. 自增与自减

本小节将介绍自增运算符的用法，此运算符能够更清晰地让游戏所用变量的值增加 1。此外，还会引用一则与此有关的趣闻轶事。

我们曾读过以下这行代码，这里不再解释：

```
// 为 myVariable 加 1
myVariable = myVariable + 1;
```

有时可以不在运算符两侧写出相同的变量，而避免重复也是一种让代码变得更清晰的方法，还能省去一些敲键盘的时间。以下代码的效果与前面代码完全相同：

```
// 明显更清晰漂亮，敲得也更快
myVariable ++;
```

这里的自增运算符与 C++中的“++”是相同的。

有个有趣的问题：你有没有好奇过C++得名的原因？实际上，C++可以称得上是C语言的一种扩展，其发明者即 **Bjarne Stroupstrup**，最初称其为“带有类的C(语言)”，但显然这个名字后来有所演进。若有兴趣，可以访问 http://www.cplusplus.com/info/history/以了解C++的发展史。

你可能已经猜到，自减运算符“--”是让某变量减1的捷径。相比于

```
playerHealth = playerHealth -1;
```

下面的代码显然更快、更清晰，而完成的功能也完全相同：

```
playerHealth --;
```

以下是运算符的更多实践，请你在阅读时试着理解每行代码的运行效果。之后，我们将继续构建Timber!!!游戏。

```
int someVariable = 10;

// 让变量的值乘以10，并将乘积放回此变量中保存
someVariable *= 10;
// someVariable现在等于100

// 让变量的值除以5，并把商放回其中
someVariable /= 5;
// someVariable现在等于20

//让变量的值加上3，并将和放回其中
someVariable += 3;
// someVariable现在等于23

// 让变量的值减去25，并将差放回其中
someVariable -= 25;
// someVariable现在等于-2
```

这段代码通过合并使用加减法运算以及赋值运算而综合利用了这些运算符，其中我们不再仅仅加减1。在使用*=、/=、+=、-=等运算符时，可以乘、除、加、减对应运算符之后变量所保存的任意数值(当然不能除以 0)。例如，在乘法代码中，`someVariable` 持有值 10，而 `someVariable *= 10` 的代码将把此值乘以 10，并把乘积赋回 `someVariable`，这种语法显然更短、更快、更清晰。

你如果已经完全理解了这些示例代码，便可以利用所学的知识来强化游戏，并让图像动起来。也就是说，是时候为游戏添加更多精灵了。

2.3　添加云朵、蜜蜂和树

首先，我们会添加一棵树，这个操作相对简单，毕竟树不必移动。我们将重用上一章绘制背景的流程。本节将准备好静态树画面，以及蜜蜂和云朵这两个可移动的精灵，随后再考虑让后面两者动起来的方法，这会用到更多C++知识。

2.3.1 准备树

请添加以下高亮显示的代码，注意，未高亮显示的代码是我们之前已经编写的代码。这种形式有助于定位新代码的添加位置，而这些新代码需要放在设定背景位置之后，并在开启游戏主循环之前。我们随后会介绍新代码的具体功能效果。

```
int main()
{
  // 创建 VideoMode 对象
  VideoMode vm(1920, 1080);

  // 为游戏创建窗口
  RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

  // 创建 Texture 对象，以便在 GPU 中保存图片
  Texture textureBackground;

  // 向此对象中加载图片
  textureBackground.loadFromFile("graphics/background.png");

  // 创建 Sprite 对象
  Sprite spriteBackground;

  // 让 Sprite 对象与 Texture 对象建立关联
  spriteBackground.setTexture(textureBackground);

  // 让 spriteBackground 占据整个屏幕
  spriteBackground.setPosition(0,0);

  // 构建树精灵
  Texture textureTree;
  textureTree.loadFromFile("graphics/tree.png");
  Sprite spriteTree;
  spriteTree.setTexture(textureTree);
  spriteTree.setPosition(810, 0);

  while (window.isOpen())
  {
```

这5行新添加的代码(不算注释)依次完成了以下工作：

- 创建 Texture 类型的对象 textureTree。
- 将图像文件 tree.png 加载到该对象中。
- 声明 Sprite 类型的对象 spriteTree。
- 让 textureTree 与 spriteTree 关联起来，从而在绘制后者时显示纹理 textureTree，即一棵树。
- 设定树的位置，其 x 坐标是810，而 y 坐标是0。

之所以设定为(810, 0)，是因为实践证明，此坐标在所选用的分辨率上的显示效果最好。这里直接为代码指定了具体的数值，但真正编程时，更应该使用变量而非具体的值，因为变量能让代码的意义更明显一些。如果这些数值保持不变，则应该使用前面讨论过的常量。为此，可以在游戏循环之外这样声明一些常量：

```
const float TREE_HORIZONTAL_POSITION = 810;
const float TREE_VERTICAL_POSITION = 0;
```

以便在绘制树精灵时这样编程：

```
spriteTree.setPosition(TREE_HORIZONTAL_POSITION,
    TREE_VERTICAL_POSITION);
```

这里，常量的声明位置与其使用位置(即 setPosition 函数)相邻，我认为 setPosition 已经足够清楚地表明了这些值的含义。如果读者认为引入两个常量更有助于厘清代码的意义，并且愿意修改代码，我将其留给读者作为练习。

另外，当我们的代码中直接包含这种数值时，它们往往由于意义不明而被批评为**魔幻数字**(magic number)。相比之下，变量名的意义一般非常明确，其意义远比纯粹的数值清晰。代码规模越大、越复杂，越不应该使用魔幻数字，在与他人协同工作或雇主要求严谨时尤其如此。诚然，本书偶尔会使用魔幻数字，但仅是为简便起见，而且一般也不会引入歧义。

接下来，让我们转而设计蜜蜂，它更有趣。

2.3.2 准备蜜蜂

下面的代码与前面的代码之间的差异不大，却很重要。既然蜜蜂需要移动，那么我们还需要声明另外两个关于蜜蜂的变量。请添加以下高亮显示的代码，同时试着猜测 beeActive 与 beeSpeed 这两个变量的作用。

```
// 构建树精灵
Texture textureTree;
textureTree.loadFromFile("graphics/tree.png");
Sprite spriteTree;
spriteTree.setTexture(textureTree);
spriteTree.setPosition(810, 0);

// 构建蜜蜂精灵
Texture textureBee;
textureBee.loadFromFile("graphics/bee.png");
Sprite spriteBee;
spriteBee.setTexture(textureBee);
spriteBee.setPosition(0, 800);

// 蜜蜂当前能否移动?
bool beeActive = false;

// 蜜蜂的飞行速度
float beeSpeed = 0.0f;

while (window.isOpen())
{
```

在这些新插入的代码中，我们用与背景图和树相同的方法创建了一只蜜蜂，即分别创建 Texture 与 Sprite 对象，并将二者关联起来。

还需注意的是，前面蜜蜂的代码中还有一些在项目中不曾出现过的新代码(虽然在本章前面讨论变量时曾大致提过)，其中那个 bool 型变量用于标识蜜蜂是否处于活动状态(别忘了，布尔型

变量只能取值为 true 或 false)，目前此 beeActive 变量被初始化为 false。在其下面，我们声明了一个新的 float 变量 beeSpeed，它将保存蜜蜂在屏幕中移动的速度，单位为像素/秒。

很快，我们便能看到如何使用这两个新变量来移动蜜蜂，但在这之前，让我们先用近乎完全相同的方式来设定云朵。

2.3.3 准备云朵

请添加并研究以下高亮显示的代码，尝试分析其功能，我们随后将进行解释。

```
// 构建蜜蜂精灵
Texture textureBee;
textureBee.loadFromFile("graphics/bee.png");
Sprite spriteBee;
spriteBee.setTexture(textureBee);
spriteBee.setPosition(0, 800);

// 蜜蜂当前能否移动?
bool beeActive = false;

// 蜜蜂的飞行速度
float beeSpeed = 0.0f;

// 通过一个纹理对象构建三个云朵精灵
Texture textureCloud;
// 加载新纹理
textureCloud.loadFromFile("graphics/cloud.png");

// 三个纹理相同的新精灵
Sprite spriteCloud1;
Sprite spriteCloud2;
Sprite spriteCloud3;
spriteCloud1.setTexture(textureCloud);
spriteCloud2.setTexture(textureCloud);
spriteCloud3.setTexture(textureCloud);

// 让三个云朵精灵位于屏幕左端不同高度上
spriteCloud1.setPosition(0, 0);
spriteCloud2.setPosition(0, 250);
spriteCloud3.setPosition(0, 500);

// 云朵精灵是否位于屏幕内?
bool cloud1Active = false;
bool cloud2Active = false;
bool cloud3Active = false;

// 云朵精灵的移动速度
```

```
float cloud1Speed = 0.0f;
float cloud2Speed = 0.0f;
float cloud3Speed = 0.0f;

while (window.isOpen())
{
```

在刚刚添加的代码中，唯一可能让人困惑的一点在于其中仅使用了一个 Texture 对象，但事实上，让多个 Sprite 对象共享同一个 Texture 对象是很常见的事情。将 Texture 对象加载到 GPU 内存后，其与 Sprite 对象建立关联的操作是非常快的，最初利用 loadFromFile 函数加载图片文件的操作反而相对缓慢。当然，如果希望使用三种形状的云朵，则还是需要使用三个 Texture 对象。

除了这种略显意外的纹理共享操作，这些新代码与前面蜜蜂段的代码基本类似，仅仅是数量有所差异：三个云朵精灵需要使用三个 bool 变量，分别标识每个云朵精灵是否处在活动状态，还需要三个 float 变量，用于处理每个云朵精灵的运动速度。

2.3.4　绘制树、蜜蜂和云朵

最后，我们可以通过在绘制阶段添加以下高亮显示的代码，把这些精灵全部绘制在屏幕上：

```
/*
****************************************
绘制场景
****************************************
*/
// 清空上一帧内的所有内容
window.clear();

// 在这里绘制我们的游戏场景
window.draw(spriteBackground);

// 绘制云朵
window.draw(spriteCloud1);
window.draw(spriteCloud2);
window.draw(spriteCloud3);

// 绘制树木
window.draw(spriteTree);

// 绘制那只昆虫
window.draw(spriteBee);

// 展示所绘制的全部内容
window.display();
```

绘制三朵云、蜜蜂与树的方法与绘制背景相同，但需要注意的是绘制顺序。由于后绘制的精

灵会遮挡先绘制的精灵，因此必须先绘制背景，否则它便会遮挡其余各项，而且需要在树之前绘制云朵，否则云朵将在树前飘荡，这不符合当前场景。此外，无论在树之前还是之后绘制蜜蜂，效果都还不错，但我推荐在树之前绘制蜜蜂，这种蜜蜂可能让我们在砍树时分心，而现实中的蜜蜂便具有这种能力。

运行 Timber!!!游戏，很快便能发现树木、蜜蜂和三朵云根本就一动不动！它们看起来就像是停在起跑线上准备开赛的选手，而在这场比赛中，蜜蜂又需要向后飞。

图 2.1　绘制树、蜜蜂与云朵

基于目前我们对 C++运算符的了解，我们仍旧可以尝试让新加入的图片元素动起来，但这里面仍有一些问题。首先，真实的云朵和蜜蜂都是无规则运动的，它们不可能长期维持同一速度或同一位置，即便这些位置可能由风以及蜜蜂忙碌程度等因素决定。对于不经意的观察者而言，云朵和蜜蜂所选的路径及其速度看起来是随机的。下面我们深入探究随机性。

2.4　随机数

随机数(random number)在游戏中用处多多，例如，决定玩家会拿到哪张纸牌，或在某范围内确定实际给敌方造成的伤害等。这里我们则用随机数来确定蜜蜂和云朵的起始位置及其运动速度。

在 C++中生成随机数

为了生成随机数，需要使用更多 C++函数。本小节只是通过一些示例代码来介绍相关的语法和步骤，请不要将这些代码添加到游戏中。

计算机自身无法创建真正的随机数，而仅能使用一些**算法**(algorithm)来挑出一些数字，使其看起来随机，这一般称为伪随机数。同样，这类算法虽然自身不会一直返回相同的值，但我们仍然需要为**随机数生成器**(random number generator)**提供种子**(seed)，而且虽然任何整数均可用作种子，但为保持结果的随机性，每次所提供的种子绝对不能重复。以下代码为随机数生成器提供了种子：

```
// 用当前时刻作为随机数生成器的种子
srand((int)time(0));
```