第2章　设计专业的C++程序

在编写应用程序之前，先不要具体写任何代码，而应当首先设计程序。你要使用什么数据结构？要编写什么类？如果你们是一个团队，要共同开发程序，先做出这样一个计划就显得极其重要。如果你对你的同事一无所知，不知道还有哪些人在一同开发同一个程序，或者不清楚谁打算与你共同开发一个程序，就此坐下来直接上手编写程序的话，其后果可想而知！在本章中，我们将教你如何使用专业的C++方法来完成C++设计。

尽管设计很重要，但这也是软件工程中最容易遭到误解和误用的一个方面。许多程序员都可能直接陷入到应用程序中，而在此之前并没有一个清晰的计划，他们是在编写代码的过程中进行设计的，这种情况屡见不鲜。如果采用这种方法，无疑会招致混乱不堪的设计，而且这样的设计往往也极为复杂。基于这种设计，开发、调试和维护工作都会更加困难。尽管不那么直截了当，但是磨刀不误砍柴工，在项目之初多花一些时间来适当地做出设计，将会大大节省整个项目生命期的时间。

第1章已经就C++的语法和特性集上了一堂复习课。第7章还会回过头来介绍C++语法的具体细节，不过，第1部分的余下各章不打算再讨论C++语法，而主要强调编程设计。

读完本章后，你会了解：

- 编程设计的定义
- 编程设计的重要性
- C++所特有的一些设计问题
- 实现有效C++设计的两种基本原则：抽象和重用
- C++程序设计中的具体组件

2.1　什么是编程设计

程序设计（program design）或软件设计（software design）就是程序体系结构的规范，要实现这个规范来满足程序的功能和性能需求。所谓设计，就是你打算如何编写程序。一般需要以一份设计文档的形式完成设计。尽管每家公司或每个项目都有自己特定的设计文档格式，但大多数设计文档的一般布局都是一样的，其中包括两个主要部分：

1. 将程序按部分划分为多个子系统，包括子系统间的接口和依赖关系、子系统间的数据流、在各子系统之间来回的输入和输出，以及总的线程模型。

2. 各个子系统的具体细节，包括进一步细分的类、类层次体系、数据结构、算法、特定的线程模型和错误处理细节。

设计文档中通常包含有一些图表，用以显示子系统交互和类层次体系。设计文档的具体格式并不太重要，重要的是设计的考虑过程。

> 设计的关键在于，要在写程序之前考虑程序。

一般应当在开始着手编写代码之前就完成设计。设计应当提供程序的一个“路线图”，这样一般的程序员都能按照这个路线图来实现应用。当然，一旦开始编写代码，设计总免不了需要修改，而且总会遇到一些原先未曾想到的问题。通过软件工程过程，应该能够提供足够的灵活性来应对这样一些改变。第6章将更为详细地介绍几种不同的软件工程过程模型。

2.2　编程设计的重要性

为了能尽早地开始编写程序，人们往往想要跳过设计这一步，或者只是草率地匆匆完成设计。好像只有编译代码和运行代码才能有成就感，觉得自己有所长进。如果你已经或多或少地知道了想要怎样建立程序的结构，在这种情况下还来完成正式的设计似乎只是浪费时间。另外，与编写代码相比，撰写设计文档可没有那么有意思。如果只想整天写文档，那你肯定做不了真正的计算机程序员！作为程序员，我们自己也很清楚这种直接上手编写代码的诱惑有多大，不仅如此，我们偶尔也确实会陷入其中。不过，除了最简单的项目外，这样仓促上阵无疑会带来问题。

为了帮助理解编程设计的重要性，我们可以做一个实际的对比。假设你有一块土地，想在上面盖一座房子。建筑工人开始动工的时候，你告诉他想看看设计蓝图，“什么蓝图?”他这样回答，“我当然知道自己在做什么。我可不想提前为每个小细节都做出规划。你想盖两层的房子？没问题，几个月前我刚盖过一个一层的房子，我可以按着那个模型动手。”

尽管对他不是很相信，但你还是克制住自己，容他继续开工。几个月之后，你注意到在房子外面出现了一些管线，而正常情况下，应该是在墙内走线才对。对于这种不正常的情况，你会质询建筑工人，他可能会这样回答，“哦，我忘了在墙上为管线留出空间了。刚想出这种全新的净墙（drywall）技术时，真是让我兴奋不已。不过，即使管线在外面也无所谓，它还是能正常工作的，而且功能才是第一位的。”你开始对他的方法存在置疑了，不过，尽管你的判断正确，但是没有采取任何行动，而仍然允许他继续工作。

房子完工后，第一次参观房子时，你注意到厨房居然没有水池。建筑工人可能解释说“我们完成了厨房2/3的工作时才发现没有为水池留出空间。我们不想重新返工，可以在隔壁增设一个单独的水池间，这样也不错，对不对?”

> 编写程序而没有一个设计，就像是盖房子没有蓝图一样。

如果把这种情况搬到软件领域中来，对于建筑工人的诸项解释，听上去是不是很熟悉呢？你自己是不是也有这样的经历，就像把管线置于房子之外一样，你是不是也做过此类“丑陋”的解决方案呢？举例来说，对于由多个线程共享的队列数据结构，你可能忘记了要在其中设置加锁机制。等到意识到存在这个问题时，好像更容易的做法是要求所有线程都记住自行完成加锁。你可能会说，这么做确实不算“漂亮”，但起码能工作。不过，以后有新的成员加入到项目中来，他可能认为数据结构内置有加锁机制，但由于事实并非如此，最后将无法保证对共享数据的互斥访问，这就会导致一个竞争条件bug，可能需要花费3周的时间才能找出问题所在。直到此时，你才会发现原先的看法是不妥当的。

在编写代码之前建立一个正式的设计，这有助于确定如何将所有东西加以结合。就像房屋蓝图可以显示出房间之间的相互关系，以及各个房间如何共同满足房屋的需求一样，程序的设计也可以显示出程序中子系统间的相互关系，以及各个子系统如何协作来满足软件需求。如果没有设计计划，很可能会遗漏掉子系统间的连接，原本可以重用或共享的信息未能真正重用或共享，另外可能会错过一些完成任务的捷径。如果没有获得设计所给予的“整体图”，就会过分沉溺于单个的实现细节中，而忘记了支持架构和所要实现的目标才是重点。不仅如此，设计还能向项目的所有成员提供成文的文档，以供参考。

如果上述对照分析还不足以说服你先设计再编写代码，下面再提供一个例子，由此可以看出，如果

直接陷入代码编写，就无法得到一个最优的设计。假设你希望编写一个象棋程序，但在开始编程之前并没有对整个程序进行设计，而是决定直接完成最简单的部分，然后再慢慢地实现更困难的部分。根据第 1 章介绍的面向对象观点（在第 3 章还将更详细地介绍），你决定用类来建立棋子的模型。卒是最简单的棋子，你准备由此开始。考虑了卒的属性和行为之后，你写出了一个类，其属性和行为如表 2-1 所示。

表　2-1

类	属　性	行　为
卒	在棋盘上的位置 颜色（黑或白） 是否被吃掉	移动 检查移动是否合法 绘制 提升（走到棋盘对岸时）

当然，你并没有写这样一个表，而是直接开始实现的。你很高兴地转而开始实现下一个最简单的棋子：相。考虑了它的属性和功能之后，你又写了一个类，其属性和行为如表 2-2 所示。

表　2-2

类	属　性	行　为
相	在棋盘上的位置 颜色（黑或白） 是否被吃掉	移动 检查移动是否合法 绘制

同样地，因为你直接进入了编写代码阶段，也没有生成这样一个表。不过，此时你开始有些怀疑了，是不是哪些地方出了问题。相和卒看上去很像。实际上，它们的属性完全相同，许多行为也是一样的。尽管卒和相在移动这个行为上的具体实现有所不同，但它们都需要能够移动，这一点是不争的事实。如果在投入编写代码之前先对程序做了设计，可能早就能意识到不同的棋子实际上是很相似的，而且会得出某种方法，使这些公共功能只编写一次。第 3 章将介绍有关面向对象设计的技术。

另外，棋子的不同方面还取决于程序的其他子系统。例如，如果不知道怎么对棋盘建模，就无法具体地表示棋子在棋盘上的位置。另一方面，也许你想这样设计程序：让棋盘以一种特定的方式管理棋子，使得棋子无需知道自己的位置。无论哪一种情况，设计棋盘之前先在棋子类中加入位置这个属性都会带来问题。再举一个例子，如果还没有确定程序的用户界面，怎么编写一个棋子的绘制方法呢？这个绘制是基于图形的还是基于文本的？棋盘是什么样子的？这里的问题在于，程序的子系统并不是独立存在的，它们要与其他子系统相互关联。设计所做的大部分工作就是要确定和设计这些关系。

2.3　C++ 设计有什么不同之处

与其他语言的设计相比，C++ 语言的许多方面都使得 C++ 设计有所不同，而且更为复杂。

- 首先，C++ 有一个丰富的特性集。这几乎就是 C 语言的一个完备超集，另外还要加上类和对象、操作符重载、异常、模板和其他一些特性。由于这个语言的规模如此之大，就使得设计成为一项很棘手的任务。
- 其次，C++ 是一种面向对象语言。这说明，设计应当包含类层次体系、类接口和对象交互。这种设计与 C 或其他过程性语言的“传统”设计大相径庭。第 3 章将重点介绍 C++ 中的面向对象设计。
- C++ 还有一个独特的方面，这就是它提供了大量工具来设计通用的和可重用的代码。除了基本的类和继承，还可以使用其他一些语言工具来完成有效的设计，如模板和操作符重载。第 5 章将介

绍有关可重用代码的设计技术。

- 另外，C++提供了一个有用的标准库，其中包括一个字符串类、一些I/O工具，以及许多常用的数据结构和算法。而且，许多设计模式或解决问题的常用方法同样适用于C++。第4章将涉及如何使用标准库进行设计，还将介绍设计模式。

正是由于上述问题的存在，完成一个C++程序的设计可能相当困难。本书作者之一就曾经花了数天之久在纸上先勾画出设计思想，再划掉，又加上一些思想，再把它们划掉，如此往复地完成这个过程。有时这个过程是大有裨益的，数天（或者数周）之后，你会得到一个简洁、高效的设计。但有时这个过程可能很令人厌烦，而且不会提供任何帮助。你要一直都很清楚自己是否真的有进步，这一点很重要。如果发现自己陷入了困境，可以采取以下某个行动：

- 寻求帮助。咨询同事、导师，或者参考相关的书、新闻组或上网查看网页。
- 与别人合作一段时间。以后再回过头来做出设计选择。
- 做出决定，并继续。尽管某个决定不能算理想方案，但先做这个选择，尝试看能否成行。如果是一个不正确的选择，将很快暴露出来。不过，你也可能发现这是一个可以接受的方法。对于想要完成的任务，除了这种设计之外，可能并没有其他的捷径。有时即使是一种“不漂亮”的解决方案，你也必须接受，因为这是惟一能够满足需求的实际策略。

> 要记住，得到好的设计会很难，要想达到目的需要大量实践。不要期望一夜之间就能变成专家，你可能会发现掌握C++设计比掌握C++编码更为困难，对此请不要感到奇怪。

2.4 C++设计的两个原则

C++中有两个基本设计原则：抽象（abstraction）和重用（reuse）。这些原则如此重要，甚至可以把它们作为这本书的主旨。全书中将反复出现这两个原则，另外在各个领域的有效C++程序设计中也时常会出现他们的身影。

2.4.1 抽象

要理解抽象（abstraction）原则，最简单的方法是通过一个实际对照来说明。电视作为一个简单的技术，在大多数家庭中都已普及。你对电视的功能可能已经很熟悉了：可以打开和关掉电视，可以转换频道，可以调节音量，还可以增加一些外部部件，如喇叭、VCR和DVD机等。不过，你能解释在这个黑盒子里到底是怎么工作的吗？也就是说，你知道它是怎么通过大气或者通过线缆接收信号的，又是如何完成信号转换，并在屏幕上显示出来的？我们当然无法将电视如何工作解释清楚，不过对于怎么使用它却可能很精通。这是因为电视很清楚地将其内部实现与外部接口相分离。我们只是通过其接口与电视交互：包括电源开关、频道转换按钮和音量控制按钮等。我们不知道电视的工作原理，对此也不关心，我们不想知道它是怎样使用阴极管或另外某种技术在屏幕上生成图像的。这些并不重要，因为这一切不会对接口产生影响。

抽象带来的好处

在软件领域中，抽象原则也是类似的。可以使用代码，而无需了解其底层实现。先举一个简单的例子，你的程序可能调用了一个sqrt()函数（此函数在头文件<cmath>中声明），而无需知道这个函数具体使用了什么算法来计算平方根。实际上，对于不同版本的库，平方根计算的底层实现可以有所不同，只要接口保持不变，函数调用就仍能正常工作。抽象的原则还可以延伸至类。在第1章中曾介绍过，可以使用类ostream的cout对象将数据作为流传送至标准输出，如下所示：

```
cout << "This call will display this line of text\n";
```

在以上代码行中，使用了 cout 插入（insertion）操作符（带一个字符数组）的公开接口。不过，cout 是如何管理从而在用户界面上显示这行文本的呢？对此无需了解。只要知道公共接口就可以了。cout 的底层实现完全可以修改，只要保证所提供的行为和接口保持不变即可。第 14 章将更详细地讨论 I/O 流。

在设计中纳入抽象

应当适当地设计函数和类，从而使你和其他程序员可以直接使用这些函数和类，而无需知道（或依赖于）函数和类的底层实现。作为一个将实现暴露在外的设计，与将实现隐藏在接口内的设计相比，要了解它们之间有什么区别，还是来考虑前面的象棋程序。你可能希望用一个二维的 ChessPiece 对象指针数组来实现棋盘。可以如下声明和使用这个棋盘：

```
ChessPiece* chessBoard[10][10];
...
ChessBoard[0][0] = new Rook();
```

不过，这种方法没有用到抽象的概念。每个程序员要想使用这个棋盘，都会知道它实现为一个二维数组。要想对这个实现做某种修改，例如修改为一个向量数组，将会很困难，这是因为需要把整个程序中用到棋盘的地方都加以修改。在此接口与实现就未做分离。

还有一个更好的办法，就是将棋盘建模为一个类。这样就可以提供一个接口，而该接口隐藏了底层的实现细节。以下是一个 ChessBoard 类的例子：

```
Class ChessBoard {
public:
    // This example omits constructors, destructors, and the assignment operator.
    void setPieceAt(ChessPiece* piece, int x, int y);
    ChessPiece& getPieceAt(int x, int y);
    bool isEmpty(int x, int y);
protected:
    // This example omits data members.
};
```

注意，这个接口没有明确指定任何底层实现。ChessBoard 可以是一个二维数组，不过接口对此并未做严格要求。修改实现时无需修改接口。另外，实现还可以提供额外的功能，如越界检查，对于这一点前面的第一种方法是无能为力的。

通过这个例子，希望你能了解到抽象是 C++编程中的一项重要技术。第 3 章和第 5 章将更为详细地介绍抽象和面向对象设计，另外第 8 章和第 9 章将完备地提供有关的详细内容，介绍如何编写自己的类。

2.4.2　重用

C++的第二个基本设计原则是重用（reuse）。同样地，通过做一个实际的对照分析将有助于理解这个概念。假设你放弃编程，成为了一个面包师。在第一天的工作中，大师傅让你烤些饼干。为了达到他的要求，你在烹饪书里找到了巧克力饼干的食谱，接下来把配料混合在一起，在饼干模子上做出饼干，然后把饼干模子放到烤箱里烘烤。看到烤出来的饼干，大师傅可能会很满意。

下面我们要指出一些显而易见的情况，专门把它们指出来似乎有些小题大做，因为在我们看来，这实在再显然不过了。在此你没有自己造炉子来烤饼干，也没有自己搅奶油、自己磨面、自己做巧克力片。你肯定会说，“那是当然的了”。如果你是一个真正的厨师，这一点无庸置疑，但是如果你是一个程序员，要编写一个烤面包模拟游戏，又会怎样呢？倘若如此，你可能只好考虑自行编写程序的每个组件，从巧

克力片到炉子都要亲力而为。不过，你可以看看周围有没有可以重用的代码来节省时间。也许你的某个同事写过一个烹饪模拟游戏，其中有一些很不错的炉子代码。尽管这些代码不完全满足要求，没有完成你需要的每一件工作，不过可以对它进行修改，并增加必要的功能。

还有一样东西是直接拿来用的，你在这里参考了一个食谱，而不是自己研究出来一个食谱。同样的，这当然也无需多说。不过，在C++编程中可并不像这么显而易见。尽管C++中有许多标准方法可以用来解决一些反复出现的问题，但是许多程序员还是很固执，每次设计时都会把这些策略重新再设计一次。

重用代码

使用既有代码，这个思想对你来说可能并不陌生。从用cout进行打印的那一天起你就在重用代码。你并没有编写具体的代码在屏幕上打印数据，而是使用了既有的ostream实现来完成这个工作。

遗憾的是，程序员通常并没有充分利用所有可用的代码。你的设计应当考虑到既有的代码，并在适当的时候加以重用。

例如，假设想编写一个操作系统调度器。这个调度器是操作系统的一个组件，负责确定哪个进程运行，以及要运行多久。因为你想实现一种基于优先级的调度，所以可能意识到需要一个优先队列，并在这个队列中存储等待运行的进程。对于这种设计，一种直接的方法就是编写自己的优先队列。不过，你应该知道，C++标准模板库（standard template library，STL）已经提供了一个priority_queue容器，可以使用这个容器存储任何类型的对象。因此，应当在调度器的设计中加入STL中的这个priority_queue，而不是重新编写自己的优先队列。第4章将更为详细地说明代码重用，还将介绍标准模板库。

编写可重用代码

重用设计原则不仅适用于你编写的代码，还适用于你使用的代码。应当适当地编写程序，从而可以重用类、算法和数据结构。你和你的同事不仅能够在当前项目中利用这些组件，还能在将来的项目中使用这些可重用的代码。总的说来，应当避免设计过于特定的代码（仅适用于当前情况）。

C++中提供了一个编写通用代码的技术，这就是模板。以下的例子显示了一个模板化的数据结构。如果你以前从未见过这个语法，也不必担心！第11章将深入地介绍这个语法。

这里没有编写一个保存ChessPiece的特定ChessBoard类（如前所示），而是考虑编写了一个通用的GameBoard模板，这个模板可以用于任何类型的二维棋类游戏，如象棋或跳棋。只需修改类声明，使之将所要存储的棋子作为一个模板参数，而不是将棋子硬编码（直接编写）到接口中。此模板如下所示：

```
template <typename PieceType>
Class GameBoard {
public:
    // This example omits constructors, destructors, and the assignment operator.
    void setPieceAt(PieceType* piece, int x, int y);
    PieceType& getPieceAt(int x, int y);
    bool isEmpty(int x, int y);
protected:
    // This example omits data members.
};
```

通过对接口做这样一个简单的修改，就有了一个通用的游戏棋盘类，它可以用于任何二维棋类游戏。尽管这里的代码修改很简单，但是在设计阶段做出这样一些决策是非常重要的，这样一来，就能有效而且高效率地实现代码了。

重用思想

如前面的烤饼干例子所示，如果每做一道菜（每烤一种饼干）都自造一个食谱，这是很可笑的。不过，程序员通常会在设计中犯这种错误。他们可能没有利用既有的“食谱”（或模式）来设计程序，而是每设计一个程序都重新实现这些技术。在各种不同的 C++ 应用中已经出现了许多设计模式（design pattern）。作为一个 C++ 程序员，你应当熟悉这样一些模式，并在程序设计中有效地结合这些模式。

例如，你可能希望如下设计象棋程序，使之有惟一的一个 ErrorLogger 对象，由它将不同组件的所有错误串行化记录到一个日志文件中。在尝试设计 ErrorLogger 类时，你意识到，在一个程序中从 ErrorLogger 类实例化多个对象可能是灾难性的。你可能还希望能够从程序的任何位置访问这个 ErrorLogger 对象。在 C++ 程序中，经常会出现这种需求，即要求有惟一的、可全局访问的类实例，而且为实现这样的实例已经有了一种标准策略，这称为单例模式（singleton）。因此，此时如果打算使用单例模式，这就是一个好的设计。第 5 章、第 25 章和第 26 章将更详细地介绍设计模式和技术。

2.5　设计一个象棋程序

这一节将介绍如何采用一种系统的方法来设计一个 C++ 程序，在此以一个简单的象棋游戏应用为背景。为了提供一个完备的例子，这里的某些步骤会涉及后面章节中才会谈到的概念。你可以现在看这个例子，对设计过程有一个整体的认识，不过等看完第 1 部分之后，可能还希望再把这个例子回顾一遍。

2.5.1　需求

着手设计之前，有一点很重要，这就是要准确地把握程序的功能需求和性能需求。理想情况下，这些需求应当以需求规范的形式明确地给出。棋盘需求包含以下各类规范，不过实际的需求可能会更多，更为详细：

- 程序要支持象棋的标准规则（下法）。
- 程序要支持两个玩家（真正的人）。这个程序不支持人工智能机器玩家。
- 程序要支持一个基于文本的界面：
  - 程序要以 ASCII 文本方式显示棋盘和棋子。
  - 玩家要输入数字（表示棋盘上的位置）来表示如何走棋。

基于上述需求，可以确保你对程序做了正确的设计，从而得到用户所期望的表现。如果用户只需要一个基于文本的界面，你可能不会希望花时间为这个象棋游戏设计和编写一个图形用户界面。反过来，如果用户更喜欢图形用户界面，你也必须有所了解，这一点很重要，这样就不至于将程序设计为仅有基于文本的界面，而把基于图形的可能性完全排除在外。

2.5.2　设计步骤

应当采用一种系统的方法来设计程序，即从一般到特殊。以下步骤并不适用于所有程序，不过，这些步骤确实提供了一个一般原则。设计应当适当地包括一些图表。这个例子就包括有一些示例图表。可以遵循这里所用的格式，也可以建立自己的格式。

在画软件设计图时，没有对或错之分，只要设计图清晰，你和你的同事能够了解其含义即可。

将程序划分为子系统

第一步是把程序划分为通用功能子系统，并明确子系统间的接口和交互。此时，不必操心数据结构和算法（甚至类）的特定细节，只需对程序的各个部分及其交互有一个总的认识就可以了。可以把子系统列在一个表中，在这个表中表示出子系统的高层行为或功能、子系统向其他子系统提供的接口，以及

此子系统使用了其他子系统的哪些接口。象棋游戏子系统的表可能如表 2-3 所示。

表　2-3

子系统名	个　数	功　能	所提供的接口	所使用的接口
GamePlay	1	开始游戏 控制游戏流 控制绘制 宣布赢家 结束游戏	Game Over（游戏结束）	Take Turn (Player) Draw (Chess Board)
Chess Board	1	保存棋子 检查是否平局和是否有输赢 自行绘制	Get Piece At（获得棋子） Set Piece At（设置棋子） Draw（绘制）	Game Over (Game-Play) Draw (Chess Piece)
Chess Piece	32	自行绘制 自行移动 检查移动是否合法	Draw（绘制） Move（移动） Check Move（检查移动）	Get Piece At (Game Board) Set Piece At (Game Board)
Player	2	与用户交互：提示用户走棋、得到用户走棋信息 移动棋子	Take Turn（轮流下棋）	Get Piece At (Game Board) Move (Chess Piece) Check Move (Chess Piece)
ErrorLogger	1	将错误消息写至日志文件	Log Error（记录错误）	无

如表 2-3 所示，这个象棋游戏的功能子系统包括 1 个 GamePlay 子系统、1 个 ChessBoard、32 个 ChessPiece、2 个 Player，以及 1 个 ErrorLogger。不过，这并不是实现象棋游戏惟一可行的方法。在软件设计中，与编程一样，为达到同一个目标通常都有多种不同的方法。并非所有方法都一样，有些方法肯定要优于另外一些方法。不过，一般都会有多种同样有效的方法。

通过适当地划分子系统，可以将程序分解为基本功能部件。例如，在象棋游戏中，Player 就是一个区别于 Class Board、Chess Piece 或 GamePlay 的子系统。把玩家规到 GamePlay 对象中是没有意义的，因为它们在逻辑上是完全分离的子系统。其他的选择则没有这么明显。增加一个单独的用户界面子系统是否可行？如果你想提供不同类型的用户界面，或者希望以后能轻松地修改用户界面，可能会把这方面的内容分离到一个单独的子系统中。要做出此类选择，不仅需要考虑程序当前要达到的目标，还要考虑将来可能会有哪些目标。

因为从表中通常很难看出子系统间的关系，用图的形式显示程序中的子系统往往很有帮助，如图 2-1 所示。在这个图中，箭头表示从一个子系统到另一个子系统的调用（为简单起见，在此省略了 Error Logger 子系统）。

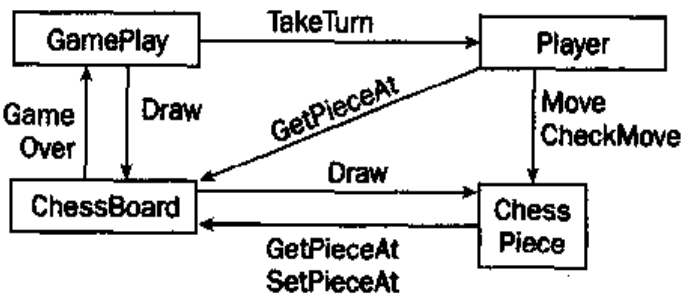

图　2-1

选择线程模型

在这一步中，要选择程序中使用多少个线程，并明确线程的交互。还要为共享数据指定加锁机制。如果你对多线程程序不熟悉，或者你的平台不支持多线程，就应该建立单线程程序。不过，如果程序有多项不同的任务，每个任务都应当并行进行，那么多线程则是一个不错的选择。例如，图形用户界面应用通常就有一个线程专门完成主要应用工作，而另一个线程用于等待用户按键或选择菜单项。

由于线程与具体平台有关，这本书不打算讨论多线程编程。第 18 章将讨论使用 C++ 时有关平台的一些考虑。

这个象棋程序只需要一个线程来控制游戏流。

为每个子系统指定类层次体系

在这一步中，要确定程序中要编写怎样的类层次体系。象棋程序只需要一个类层次体系，用以表示各种棋子。这个层次体系可能如图 2-2 所示。

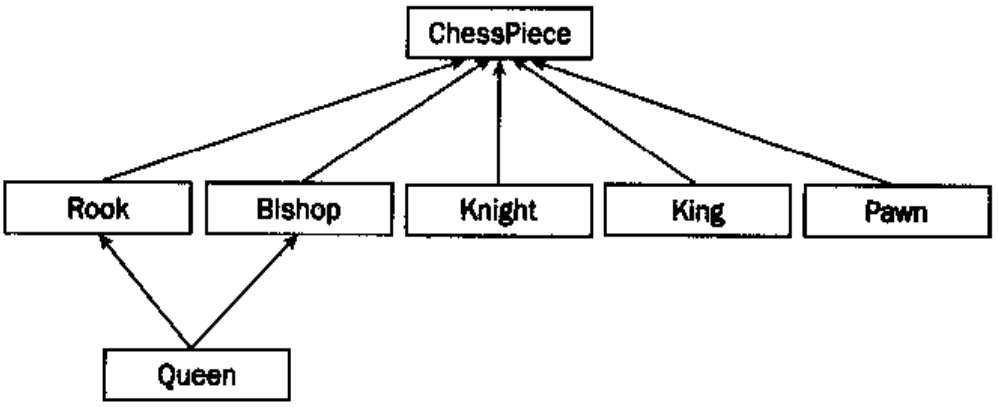

图　2-2

在这个层次体系中，有一个通用的 ChessPiece 类，这个类将用作为超类。以上层次体系使用了多重继承来显示王后实际上是车和相的结合。

第 3 章将解释设计类和类层次体系的详细内容。

为每个子系统指定类、数据结构、算法和模式

这一步要考虑更深层次的细节，并指定各个子系统的一些特定的内容，其中包括为每个子系统编写的特定类。最后很可能发现，你会为每个子系统本身建立一个类。有关信息还是用表 2-4 来总结。

表　2-4

子系统名	类	数据结构	算　法	模　式
GamePlay	GamePlay 类	GamePlay 对象包括一个 ChessBoard 和两个 Player 对象	简单循环，让每个玩家轮流走棋	无
Chess Board	ChessBoard 类	ChessBoard 对象是一个二维的 ChessPiece 数组，ChessBoard 存储有 32 个 ChessPiece	每走一步，检查是否有输赢（或平局）	无
Chess Piece	ChessPiece 抽象超类，Rook、Bishop、Knight、King、Pawn 和 Queen 类	每个棋子会存储它在棋盘上的位置	棋子向棋盘询问其他位置上的棋子，检查走得是否合法	无

（续）

子系统名	类	数据结构	算法	模式
Player	一个 Player 类	两个玩家对象（黑和白）	轮流算法：循环提示用户走棋，检查走得是否合法，并移动棋子	无
ErrorLogger	一个 ErrorLogger 类	要记录的消息队列	将消息放入缓冲区，并周期性地写入一个日志文件	单例模式，用以确保只有一个 ErrorLogger 对象

正式设计文档的这一部分通常还会表示每个类的具体接口，不过，这个例子没有做到那么细。

设计类、选择数据结构、算法和模式，这些工作可能很复杂。要时刻谨记本章前面讨论的抽象和重用这两个原则。对于抽象，关键是要分别考虑接口和实现。首先，要从用户角度指定接口，确定希望组件做些什么。再通过选择数据结构和算法确定组件如何达到目的。对于重用，要先熟悉标准的数据结构、算法和模式。另外，还要确保自己了解C++中的标准库代码，以及可供使用的任何专用代码。

第3章、第4章和第5章将更为详细地讨论这些问题。

为每个子系统指定错误处理

在这个设计步骤中，要明确每个子系统中的错误处理。错误处理应当包括系统错误（如内存分配失败）和用户错误（如非法输入）的处理。要指定各个子系统是否使用异常。下面还是通过表2-5来总结有关的信息。

表 2-5

子系统名	处理系统错误	处理用户错误
GamePlay	如果无法为 ChessBoard 或 Player 分配内存，用 ErrorLogger 记录一个错误，并终止程序	无（没有直接的用户界面）
Chess Board	如果无法为其自身或 ChessPiece 分配内存，用 ErrorLogger 记录一个错误，并抛出一个异常	无（没有直接的用户界面）
Chess Piece	如果无法分配内存，则用 ErrorLogger 记录一个错误，并抛出一个异常	无（没有直接的用户界面）
Player	如果无法分配内存，则用 ErrorLogger 记录一个错误，并抛出一个异常	检查用户输入的走法是否正常，确保不至于走到棋盘外面去；倘若真的会走到棋盘外面去，提示用户重新输入 在真正移动棋子之前先检查各步是否合法；如果不合法，提示用户再考虑其他走法
ErrorLogger	尝试记录一个错误，如果无法分配内存，则终止程序	无（没有直接的用户界面）

错误控制的一般原则是一切都要处理。要仔细周全地考虑到所有可能的错误条件。如果漏掉了一种可能性，最后它就会在程序中作为一个 bug 出现在你面前！不要让问题变成“未预料到的”错误才暴露出来。要考虑到所有可能性：内存分配失败，非法的用户输入，磁盘故障，网络故障等等。不过，从针对象棋游戏的这个表可以看到，对用户错误的处理应当与内部错误的处理有所区别。例如，用户输入了一个非法的走法时，不应导致象棋程序终止。

第15章将更深入地讨论错误处理。

2.6　小结

本章介绍了专业 C++设计方法。我们希望，你能由此了解到软件设计的的确确是所有编程项目中首要的一步。在此，你还了解了 C++的哪些方面导致了设计困难，这包括 C++的面向对象性，丰富的特性集和标准库，以及编写通用代码的诸多工具。有了以上基础，你应该有更充分的准备，可以着手进行 C++设计了。

本章介绍了两个设计原则：抽象和重用。所谓抽象，就是将接口与实现相分离，这个概念贯穿本书始终，应当作为所有设计工作的指导原则。重用的思想（无论是代码重用还是思想重用）在实际的项目经常会出现，在本书中也会屡屡看到它的身影。应当重用既有的代码和思想，并尽可能地将代码编写为可以重用。

既然已经了解了设计的重要性，也知道了基本的设计原则，下面可以开始学习第 1 部分的余下内容了。第 3 章将介绍在设计中利用 C++的面向对象方面时有哪些策略。第 4 章和第 5 章对重用既有代码和思想以及为编写可重用的代码提供了相应的原则。第 1 部分的最后是第 6 章，其中讨论了软件工程模型和过程。