

第一篇

C++ 概述

我们编写的程序由两个主要方面组成：

1. 算法的集合。（就是将指令组织成程序来解决某个特定的问题。）
2. 数据的集合。（算法在这些数据上操作，以提供问题的解决方案。）

纵观短暂的计算机发展史，这两个主要方面（算法和数据）一直保持不变。发展演化的是它们之间的关系，就是所谓的**程序设计方法**（programming paradigm）。

在过程化程序设计方法（procedural programming）中，一个问题可直接由一组算法来建立模型。例如，公共图书馆的资料借阅/登记（check out/check in）系统是由一系列过程表现出来的。其中两个主要的过程是资料的借阅和登记。这些数据被独立存储起来，我们既可以在某个全局位置上访问这些数据，或者把数据传递给过程以便它能够访问这些数据。Fortran、C 和 Pascal 是三种著名的过程语言。C++也支持过程化程序设计。单独的过程，如 check_in()、check_out()、over_due()、fine()等等，都被称为函数。第二篇将集中讨论 C++对过程化程序设计方法的支持，尤其将重点讨论函数、函数模板和通用算法。

在 20 世纪 70 年代，程序设计的焦点从过程化程序设计方法转移到了**抽象数据类型**（abstract data type，简称为 ADT）的程序设计上，现在通常称之为**基于对象**（object based）的程序设计。在基于对象的程序设计方法中，我们通过一组数据抽象来建立问题的模型，在 C++中我们把这些抽象称为**类**（class）。例如，在这种方法下，图书馆资料借阅/登记系统就由类的对象实例（比如书、借阅者、还书时间、罚款等）之间的相互作用表现出来，以此表示出图书馆的抽象概念。与每个类相关的算法被称为该类的**公有接口**（public interface）。数据以私有形式被存储在每个对象中，对数据的访问应与一般的程序代码隔离开来。CLU、Ada 和 Modula-2 是三种支持抽象数据类型的程序设计语言。第四篇将说明和讨论 C++对抽象数据类型程序设计方法的支持。

面向对象的程序设计方法通过**继承**（inheritance）机制和**动态绑定**（dynamic binding）机制扩展了抽象数据类型；继承机制是对现有实现代码的重用，动态绑定是指对现有的公有接口的重用。以前独立的类型现在有了类型/子类型的特定关系。一本书、一盒录像带、一段录音，甚至孩子的宠物，尽管它们有各自的借阅/登记方式，但都可以成为图书馆的收藏资料。共享的公有接口和私有的数据都放在一个抽象类（图书馆资料，LibraryMaterial）中。每个特殊的图书馆资料类都从 LibraryMaterial 抽象类继承共享的行为，它们只需要提供与自身行为

相关的算法和数据。Simula、Smalltalk 和 Java 是三种支持面向对象程序设计方法的著名语言。第五篇将集中讨论 C++对面向对象程序设计方法的支持。

C++是一种支持多种程序设计方法的语言。虽然我们主要把它当作面向对象的语言，但实际上它也提供对过程化的和基于对象的程序设计方法的支持。这样做的好处是对每个问题都能够提供最合适的解决方案。事实上，没有一种程序设计方法能够对所有的问题都提供最好的解决方案。这样做带来的缺点是使得语言过于庞大、复杂。

第一篇将对整个 C++进行快速浏览。这样做的一个原因是，它可以提供对语言特性的介绍，以便我们在完全面对这些特性之前，可以自由地引用语言的各个部分。例如，直到第 13 章我们才会详细介绍类，但如果到那时候才提起类，那么在此之前我们将不得不使用很多非典型的、不恰当的程序例子。

提供快速浏览的第二个原因是从美学的角度出发。除非首先让你领略到贝多芬交响曲的美丽与博大，否则，无关联的升半音、降半音、八度音符、和弦……等一定会让你厌烦。但是，只有掌握了这些细节，才有可能创作音乐。程序设计也一样，精通 C++程序设计的基础是首先要穿过操作符优先级或标准算术转换规则的迷宫，这样做既是必要的，也是非常枯燥的。

第 1 章将首先介绍 C++语言的基本元素，包括内置数据类型、变量、表达式、语句、函数。它将通过一个最小的，并且是合法的 C++程序，来讨论程序的编译过程、预处理、以及 C++对输入/输出的支持。这一章将给出多个简单但完整的 C++程序，鼓励读者亲自编译并执行这些程序。

第 2 章我们将浏览一个过程化程序、一个基于对象的程序和一个面向对象的程序，它们都实现了一个数组（一个由相同类型的元素组成的有限元素的集合）。然后，我们将这些程序中的数组抽象与 C++标准库中的向量（vector）类进行比较，同时也将首次介绍标准库中的通用算法。沿着这条路线，我们还将介绍 C++对异常处理、模板、名字空间的支持。实际上，这一章对整个 C++语言作了大致的介绍，细节部分将在以后各章节中详细介绍。

部分读者可能会感觉第 2 章很难理解，给出的许多资料没有初学者所期望的完整说明（这些细节在以后的章节中讨论）。如果你对某些细节部分感到吃力或失去耐心，建议略读或跳过它们，等到熟悉这些资料以后，再回头重读这些内容。第 3 章将以传统的叙述方式进行，建议对第 2 章不够适应的读者，从第 3 章开始。

开 始

本章介绍 C++ 语言的基本元素，包括内置数据类型、对象的定义、表达式、语句、函数的定义和使用。本章将给出一个最小的合法 C++ 程序，主要用它来讨论程序的编译过程、预处理，并将首次介绍 C++ 对输入/输出的支持。我们还将给出一些简单但完整的 C++ 程序。

1.1 问题的解决

程序常常是针对某些要解决的问题和任务而编写的。我们来看一个例子，某个书店将每本售出图书的书名和出版社，输入到一个文件中，这些信息以书售出的时间顺序输入，每两周店主将手工计算每本书的销售量，以及每个出版社的销售量。报表以出版社名称的字母顺序排列，以便下订单。现在，我们希望写一个程序来完成这项工作。

解决大问题的一种方法，是把它分解成许多小问题。理想情况下，这些小问题可以很容易地被解决。然后，再把它们合在一起，就可以解决大问题了。如果新分割的小问题解决起来还是太大，就把它分割得再小一些，重复整个过程，直到能够解决每个小问题。这个策略就是分而治之（divide and conquer）和逐步求精（stepwise refinement）。书店问题可以分解成四个子问题（或任务）：

1. 读销售文件；
2. 根据书名和出版社计算销售量；
3. 以出版社名称对书名进行排序；
4. 输出结果。

我们知道怎样解决第 1、2 和 4 个子问题，因此它们不需要进一步分解。但是，第 3 个子问题解决起来还是有些大，所以对这个问题重复我们的做法，继续分解：

- 3a. 按出版社排序；
- 3b. 对每个出版社的书，按书名排序；
- 3c. 在每个出版社的组中，比较相邻的书名，如果两者匹配，增加第一个的数量，删除第二个。

3a、3b 和 3c 所代表的问题，现在都已经能够解决了。由于我们能够解决这些子问题，

因此也就能够有效地解决原始的大问题了。而且，我们也知道任务的原始顺序是不正确的，正确的动作序列应该是：

1. 读销售文件；
2. 对文件排序——先按出版社，然后在出版社内部按书名排序；
3. 压缩重复的书名；
4. 将结果写入文件。

这个动作序列就是**算法**（algorithm）。下一步我们把算法转换成一种特定的程序设计语言——在这里是 C++ 语言。

1.2 C++ 程序

在 C++ 中，动作被称为**表达式**（expression）。以分号结尾的表达式被称作**语句**（statement）。C++ 中最小的程序单元是语句，在自然语言中，与此类似的结构就是句子。例如，下面是一组 C++ 语句：

```
int book_count = 0;
book_count = books_on_shelf + books_on_order;
cout << "the value of book_count: " << book_count;
```

第一条语句是一个**声明**（declaration）语句，book_count 被称为**标识符**（identifier）或**符号变量**（symbolic variable，简称变量），或者**对象**（object），它定义了计算机内存的一块区域，并且与名字 book_count 相关联，被用来存储整数。0 是一个**文字常量**（literal constant），book_count 被初始化为 0。

第二条语句是一个**赋值**（assignment）语句，它把 books_on_shelf 和 books_on_order 的值相加，并把结果放入与 book_count 相关联的计算机内存区域中。这里假定 books_on_shelf 和 books_on_order 已经在前面的代码中被声明为整型，并赋了初值。

第三条是**输出**（output）语句，cout 是与用户终端相关联的输出目标。<< 是输出操作符，该语句向 cout（即用户终端）先输出用引号括起来的字符串文字，然后输出存储在与名字 book_count 相关联的内存区域中的值。假设此时 book_count 中的值为 11273，那么输出结果为：

```
the value of book_count: 11273
```

把语句按逻辑语义分组，就形成了一些有名字的单元，这些单元被称为**函数**（function）。例如，把所有需要读取销售文件的语句组织到一个被称为 readIn() 的函数中。类似地，我们可以构成 sort()、compact() 和 print() 函数。

在 C++ 中，每个程序必须包含一个被称作 main() 的函数，它是由程序员提供的，并且只有这样的程序才能运行。下面是按前述算法定义的一种 main() 函数：

```
int main()
{
    readIn();
    sort();
    compact();
    print();
}
```

```
    return 0;
}
```

C++程序从 `main()` 函数的第一条语句开始执行，在本例中，程序从函数 `readIn()` 开始。并且，程序按顺序执行 `main()` 函数中的语句。在执行完 `main()` 函数的最后一条语句之后，程序正常结束。

函数由四部分组成：返回类型、函数名、参数表，以及函数体。前三部分合起来称为**函数原型**（function prototype）。

参数表由小括号括起来，包含一个或多个由逗号分开的参数。函数体由一对花括号括起来，由程序语句序列构成。

在本例中，`main()` 函数的函数体调用（invoke）函数 `readIn()`、`sort()`、`compact()` 和 `print()`。当这些函数调用都完成时，下面的语句：

```
    return 0;
```

将被执行。`return` 是 C++ 预定义的语句，它提供了终止函数执行的一种方法。当 `return` 语句提供了一个值时，例如 `0`，这个值就成为函数的**返回值**（return value）。本例中，返回值为 `0` 表示 `main()` 函数成功执行完毕。（标准 C++ 中，如果 `main()` 函数没有显式地提供返回语句，则它缺省返回 `0`。）

现在我们来了解一下，如果想让我们的程序能够执行起来，我们还需要做哪些准备工作。首先，必须提供函数 `readIn()`、`sort()`、`compact()` 以及 `print()` 的定义。下面的哑函数实例已经足够满足这个要求了。

```
void readIn() { cout << "readIn()\n"; }
void sort()   { cout << "sort()\n";   }
void compact() { cout << "compact()\n"; }
void print()  { cout << "print()\n";  }
```

`void` 用来指定一个没有返回值的函数。正如上面的定义所示，每个函数在被 `main()` 函数调用时只会简单地在用户终端上显示它的存在，以后，我们会用真正的实现函数来代替这些哑函数。

这种渐进式生成程序的方法，为控制程序设计中不可避免的错误，提供了一种有效的控制手段。试图一下子就能写出一个完全成功的程序，几乎是不可能的。

程序源文件的名字，一般包括两部分：文件名（例如 `bookstore`）以及文件后缀。文件后缀一般用来标识文件的内容。比如文件：

```
bookstore.h
```

在 C 或 C++ 中习惯上被称为**头**（header）文件。（标准 C++ 头文件没有后缀，这是个例外。）而以下文件：

```
bookstore.c
```

习惯上被当作 C 程序文本文件。但在 UNIX 操作系统中，以下文件：

```
bookstore.C
```

习惯上被当作 C++ 程序的文本文件。C++ 程序文件的后缀在不同的 C++ 实现产品中是不

同的，特别在 DOS 中大写的字母 C 与小写的字母 c 是不能区分的。其他常用来识别 C++ 程序文本文件的后缀还包括：

```
bookstore.cxx
bookstore.cpp
```

类似地，头文件的后缀在 C++ 的不同实现中也不相同（这也是标准 C++ 没有指定头文件后缀的一个原因）。请查阅你的编译器的用户指南，以确定在当前平台上使用什么后缀。

使用某个文本编辑器，将下面这段完整的程序输入到一个 C++ 文本文件中。

```
#include <iostream>
using namespace std;

void read()    { cout << "read()\n";    }
void sort()    { cout << "sort()\n";    }
void compact() { cout << "compact()\n"; }
void write()   { cout << "write()\n";   }

int main() {
    read();
    sort();
    compact();
    write();

    return 0;
}
```

`iostream` 是输入/输出流库标准文件（注意它没有后缀），它包含 `cout` 的信息，这对我们的程序是必需的。`#include` 是预处理器指示符（preprocessor directive），它 `iostream` 的内容读入我们的文本文件中（1.3 节将讨论预处理器指示符）。

在 C++ 标准库中定义的名字，如 `cout`，不能在程序中直接使用，除非在预处理器指示符：

```
#include <iostream>
```

后面加上语句：

```
using namespace std;
```

这条语句被称作 `using` 指示符（`using directive`）。C++ 标准库中的名字都是在一个称作 `std` 的名字空间中声明的，这些名字在我们的程序文本文件中是不可见的，除非我们显式地使它们可见。`using` 指示符告诉编译器要使用在名字空间 `std` 中声明的名字。（在 2.7 和 2.8 节中将更进一步讨论名字空间与 `using` 指示符。）^①

程序已经被输入到文件（如 `prog1.c`）中之后，接下来就应将其编译。在 Unix 系统中，按以下步骤进行（\$ 表示系统提示符）。

```
$ CC prog1.C
```

用来调用 C++ 编译器的命令的名字，在不同的实现中也不相同（在 Windows 中，通常通

^① 在本书写作时（大约指 1997 年前后——译注），并不是所有的 C++ 实现都支持名字空间。如果你使用的 C++ 实现不支持名字空间，那么 `using` 指示符必须要忽略掉。因为本书的许多例子都是用不支持名字空间的 C++ 实现来编译的，所以绝大多数的代码例子都省略了 `using` 指示符。

过选择菜单项来调用命令)。CC 是 Unix 工作站上使用 C++编译器的命令名。可以通过参考手册或系统管理员获得系统的 C++命令名。

编译器的一部分工作是分析程序代码的正确性。编译器不能判断程序的语义是否正确，但能够判断出程序形式(form)上的错误，下面是两个常见的程序形式错误。

1. 语法错误。程序员犯了 C++语言的语法错误。例如：

```
int main ( {    // 错误：缺少 ')'
    readIn();  // 错误：非法字符 ':'
    sort();
    compact();
    print();

    return 0    // 错误：缺少 ';'
}
```

2. 类型错误。在 C++中，每个数据项都有一个相对应的数据类型，例如，10 是一个整型数值。由双引号括起来的词“hello”是一个字符串。如果为一个需要整型参数的函数提供了一个字符串，编译器就会报告类型错误。

错误消息包含一个行号以及编译器对错误的简要描述。按报告的顺序逐一修正错误，是个好的习惯。一个简单的错误常常有很多关联影响，会使编译器报告的错误比实际要多得多，因此，一旦错误被改正后，应当马上重新编译。这个循环过程通常被称为编辑—编译—调试(edit-compile-debug)。

编译器的第二部分工作是转换正确的程序代码。这种转换被称为代码生成(code generation)。典型情况下，它生成目标代码或汇编指令代码。这些代码能够被运行该程序的计算机所理解。

成功编译的结果是一个可执行文件。前面的程序执行时，其输出结果如下：

```
readIn()
sort()
compact()
print()
```

C++定义了一组内置的基本数据类型：整数类型(int)、浮点数类型(float)、字符类型(char)、以及只有 false 和 true 两个值的布尔类型(boolean)。每种类型都与 C++语言中某一个关键字(keyword)相关联，程序中的每个对象都与一个特定的类型相关联。例如，下面的代码：

```
Int age = 10;
double price = 19.99;
Char delimiter = ' ';
Bool found = false;
```

定义了四个对象：age、price、delimiter 和 found，分别是整数类型、双精度浮点数类型、字符类型和布尔类型。每个类型都赋予了一个文字常量初始值：整数 10、浮点数 19.99、空格字符、布尔值 false。

在内置类型之间经常发生隐式的类型转换(conversion)。例如，将一个 double 双精度型的常量赋给一个 int 型的 age：

```
age = 33.333;
```

实际上, 赋给 `age` 的是被截断后的整数值 33。[标准转换 (standard conversion) 以及一般类型的转换将在 4.14 节中详细讨论。]

C++ 标准库还提供了一组扩展的基本数据类型。其中包括字符串 (string)、复数 (complex number)、向量 (vector) 和列表 (list)。例如:

```
// 为了使用string对象, 下面的头文件是必需的
#include <string>
string current_chapter = "Getting Started";

// 为了使用vector对象, 下面的头文件是必需的
#include <vector>
vector<string> chapter_titles( 20 );
```

`current_chapter` 是一个字符串对象, 被初始化为字符串文字“Getting Started”。`chapter_title` 是一个向量, 包含有 20 个字符串类型的元素。以下这种特殊语法:

```
vector<string>
```

指示编译器创建一个能够存放字符串元素的向量类型。要定义一个能够存放 20 个整数的向量对象, 我们可以这样写:

```
vector<int> ivec( 20 );
```

(本书对向量还将进行更多的描述。)

无论是一种语言还是它的标准库, 都不可能提供实际程序设计环境要求的所有数据类型, 因此, 现代语言都提供了类型定义工具设施, 使我们能够在语言中引入新的类型, 这些类型的用法与内置类型的用法一样方便。在 C++ 中, 这种设施就是类机制。在 C++ 中, 像 `string`、`complex`、`vector`、`list` 这样的标准库类型都被设计成类。实际上, 输入/输出库也是这样的。

类设施可能是 C++ 中最重要的组成部分, 第 2 章对整个类机制作了基本的概述性介绍。

1.2.1 程序流程控制

缺省情况下, 语句将按顺序执行。例如, 在前面的程序中 (重新列在下面), `read()` 总是先被执行, 然后是 `sort()`、`compact()`、`write()`。

```
int main()
{
    read();
    sort();
    compact();
    write();

    return 0;
}
```

然而, 如果销售进展得很慢, 例如, 只有 0 或 1 项, 那么就没有必要排序和压缩了, 但是我们仍然需要输出这一项销售记录, 或者指出没有销售记录产生。通过条件语句 `if` 我们可以完成这项工作。(假设已经重写了 `readIn()` 函数, 使其能够返回读入的项数。)

```
// read() 返回读入的项数
```



```
// 返回值的类型为int
int read() { ... }

// ...

int main()
{
    int count = read();

    // 如果读入的项数大于1
    // 就调用sort()和compact()

    if ( count > 1 ) {
        sort();
        compact();
    }

    if ( count == 0 )
        cout << "no sales for this month\n";
    else write();

    return 0;
}
```

第一个 if 语句给出了在括号中的条件表达式为真的情况下应该执行的动作。在这个被修改过的程序中，只有在 count 大于 1 的时候 sort()、compact() 函数才会被调用。在第二个 if 语句中，有两个执行分支。如果条件为真（在这里，即如果 count 等于 0）则简单地输出没有销售产生，否则，只要 count 不等于 0，就调用 write()。（我们将在 5.3 节中详细讨论 if 语句。）

第二种非顺序执行的语句是**迭代**（iterate）或称**循环**（loop）语句。当条件保持为真的时候，循环重复执行一条或多条语句。例如：

```
int main()
{
    int iterations = 0;
    bool continue_loop = true;
    while ( continue_loop != false )
    {
        iterations++;

        cout << "the while loop has executed "
              << iterations << " times\n";

        if ( iterations == 5 )
            continue_loop = false;
    }

    return 0;
}
```

在这个看似人为构造的例子中，while 循环执行 5 次，直到 iterations 等于 5 并且 continue_loop 被赋值为 false。如下语句：

```
iterations++;
```

将使 iterations 加 1。在 1.5 节中将有更实际的 while 循环的例子。第 15 章将详细讲解循环语句。

1.3 预处理器指示符

头文件通过 include 预处理器指示符（preprocessor include directive）而成为我们程序的一部分。预处理器指示符用“#”号标识，这个符号将放在程序中该行的最起始一列上。处理这些指示符的程序被称做预处理器（preprocessor）（通常捆绑在编译器中）。

#include 指示符读入指定文件的内容，它有两种格式：

```
#include <some_file.h>
#include "my_file.h"
```

如果文件名用尖括号“<”和“>”括起来，表明这个文件是一个工程或标准头文件，查找过程会检查预定义的目录。我们可以通过设置搜索路径环境变量或命令行选项来修改这些目录。（在不同的平台上这些方法大不相同，建议你请教同事或查阅编译器手册以获得更进一步的信息。）如果文件名用一对引号括起来，则表明该文件是用户提供的头文件，查找该文件时将从当前文件目录开始。

被包含的文件还可以含有#include 指示符。由于嵌套包含文件的原因，一个头文件可能会被多次包含在一个源文件中。条件指示符可防止这种头文件的重复处理。例如：

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H
/* Bookstore.h 的内容 */
#endif
```

条件指示符#ifndef 检查 BOOKSTORE_H 在前面是否已经被定义。这里，BOOKSTORE_H 是一个预编译器常量。（习惯上预编译器常量往往被写成大写字母。）如果 BOOKSTORE_H 在前面没有被定义，则条件指示符的值为真，于是从#ifndef 到#endif 之间的所有语句都被包含进来进行处理。相反，如果#ifndef 指示符的值为假，则它与#endif 指示符之间的行将被忽略。

为了保证头文件只被处理一次，把如下#define 指示符：

```
#define BOOKSTORE_H
```

放在#ifndef 后面，这样在头文件的内容第一次被处理时，BOOKSTORE_H 将被定义，从而防止了在程序文本文件中以后#ifndef 指示符的值为真。

只要不存在“两个必须包含的头文件要检查一个同名的预处理器常量”这样的情形，这个策略就能够很好地运作。

#ifdef 指示符常被用来判断一个预处理器常量是否已被定义，以便有条件地包含程序代码。例如：

```
int main()
{
#ifdef DEBUG
    cout << "Beginning execution of main()\n";
#endif
```

```
    string word;
    vector< string > text;

    while ( cin >> word )
    {
#ifdef DEBUG
        cout << "word read: " << word << "\n";
#endif
        text.push_back( word );
    }

    // ...
}
```

本例中，如果没有定义 `DEBUG`，实际被编译的程序代码如下：

```
int main()
{
    string word;
    vector< string > text;

    while ( cin >> word )
    {
        text.push_back( word );
    }

    // ...
}
```

反之，如果定义了 `DEBUG`，则传给编译器的程序代码是：

```
int main()
{
    cout << "Beginning execution of main()\n";

    string word;
    vector< string > text;

    while ( cin >> word )
    {
        cout << "word read: " << word << "\n";
        text.push_back( word );
    }

    // ...
}
```

我们在编译程序时可以使用 `-D` 选项，并且在后面写上预处理器常量的名字，这样就能在命令行中定义预处理器常量：^②

```
$ CC -DDEBUG main.C
```

也可以在程序中用 `#define` 指示符定义预处理器常量。

② 对于 UNIX 系统，确实是这样的。Windows 程序员应该检查一下编译器的用户指南。

编译 C++ 程序时, 编译器自动定义了一个预处理器名字 `__cplusplus` (注意前面有两个下划线)。因此, 我们可以根据它来判断该程序是否是 C++ 程序, 以便有条件地包含一些代码。例如:

```
#ifndef __cplusplus
// 不错, 我们要编译C++
// extern "C"到第7章再解释!
extern "C"
#endif
int min( int, int );
```

在编译标准 C 时, 编译器将自动定义名字 `__STDC__`。当然, `__cplusplus` 与 `__STDC__` 不会同时被定义。

另外两个比较有用的预定义名字是: `__LINE__` 和 `__FILE__`。`__LINE__` 记录文件已经被编译的行数, `__FILE__` 包含正在被编译的文件的名字。可以这样使用它们:

```
if ( element_count == 0 )
    cerr << "Error: " << __FILE__
        << " : line " << __LINE__
        << "element_count must be non-zero.\n";
```

另外两个预定义名字分别包含当前被编译文件的编译时间 (`__TIME__`) 和日期 (`__DATE__`)。时间格式为 hh:mm:ss, 因此如果在上午 8 点 17 分编译一个文件, 则时间表示为 08:17:05。如果这一天是 1996 年 10 月 31 日, 星期四, 则日期表示为:

```
Oct 31 1996
```

若当前处理的行或文件发生变化, 则 `__LINE__` 和 `__FILE__` 的值将分别被改变, 其他四个预定义名字在编译期间保持不变。它们的值也不能被修改。

`assert()` 是 C 语言标准库中提供的一个通用预处理器宏。在代码中常利用 `assert()` 来判断一个必需的前提条件, 以便程序能够正确执行。例如, 假定我们要读入一个文本文件, 并对其中的词进行排序, 必需的前提条件是文件名已经提供给我们了, 这样我们才能打开这个文件。为了使用 `assert()`, 必须包含与之相关联的头文件:

```
#include <assert.h>
```

下面是一个简单的使用示例:

```
assert( filename != 0 );
```

`assert()` 将测试 `filename` 不等于 0 的条件是否满足。这表示, 为了后面的程序能够正确执行, 我们必须断言一个必需的前提条件。如果这个条件为假 (即: `filename` 等于 0), 断言失败, 则程序将输出诊断消息, 然后终止。

`assert.h` 是 C 库头文件的 C 名字, C++ 程序可以通过 C 库的 C 名字或 C++ 名字来使用它。这个头文件的 C++ 名字是 `cassert`。C 库头文件的 C++ 名字总是以字母 C 开头, 后面是去掉后缀 .h 的 C 名字。(正如前面所解释的, 由于在各种 C++ 实现中, 头文件的后缀各不相同, 因此标准 C++ 头文件没有指定后缀。)

使用头文件的 C 名字, 或者 C++ 名字, 两种情况下头文件的 `#include` 预处理器指示符的效果也会不同。下面的 `#include` 指示符:

```
#include <cassert>
```

将 `cassert` 的内容被读入到我们的文本文件中。但是由于所有的 C++ 库名字是在名字空间 `std` 中被定义的，因而在我们的程序文本文件中，它们是不可见的，除非用下面的 `using` 指示符显式地使其可见：

```
using namespace std;
```

使用 C 头文件的 `#include` 指示符：

```
#include <assert.h>
```

就可以直接在程序文本文件中使用名字 `assert()`，而无需使用 `using` 指示符。^③[库文件厂商用名字空间来控制全局名字空间污染（即名字冲突）问题，以避免它们的库“污染”了用户程序的名字空间。8.5 节将讨论这些细节。]

1.4 注释

注释是用来帮助程序员读程序的语言结构，它是一种程序礼仪，可以用来概括程序的算法、标识变量的意义、或者阐明一段比较难懂的程序代码。注释不会增加程序的可执行代码的长度。在代码生成以前，编译器会将注释从程序中剔除掉。

C++ 中有两种注释符号，一种是注释对 `(/*, */)`，与 C 语言中的一样。注释的开始用 `/*` 标记，编译器会把 `/*` 与 `*/` 之间的代码当作注释。注释可以放在程序的任意位置，可以含有制表符（`tab`）、空格或换行，还可以跨越多行程序。例如：

```
/*
 * 这是我们第一次看到C++的类定义
 * 类可用于基于对象和
 * 面向对象编程中，Screen类的
 * 实现代码在第13章中
 */

class Screen {
    /* 此部分称为类体*/
public:
    void home();      /* 将光标移到0,0 */
    void refresh();  /* 重绘Screen */
private:
    /* 类支持“信息隐藏” */
    /* 信息隐藏限制了程序 */
    /* 对类的内部表示（其数据）的 */
    /* 访问，这是用“private” */
    /* 来表示的 */
    int height, width;
};
```

在代码中混杂过多的注释会使程序更难于理解。例如，注释几乎淹没了 `width` 和 `height`

^③ 在本书写作时，并不是所有的 C++ 实现都支持 C 库头文件的 C++ 名字。因为本书的许多例子是在不支持 C++ 头文件名的实现中编译的，所以有时候例子代码会用 C 名字引用 C 库头文件，而有时候用 C++ 名字引用 C 库头文件。

的声明。通常，把注释放在要描述的文本之上比较合适。与其他软件文档一样，考虑到有效性问题，注释必须随着软件的发展而升级。但是，注释与所描述的代码随时间推移而渐行渐远的情况却是经常发生的。

注释对不能嵌套，即一个注释对不能出现在另外一个注释对之中。请尝试在系统中编译下面的程序，它会使大多数编译器无法正常处理：

```
#include <iostream>

/*
 * 注释对 /* */ 不能嵌套
 * 这里“不能嵌套”几个字将被认为是代码
 * 还包括接下来的这几行
 */

int main() {
    cout << "hello, world\n";
}
```

解决这种嵌套注释的一个办法是在星号和斜线之间加一个空格。

```
/* * /
```

对于星号和斜线序列，只有当这两个字符之间没有被空格分割时，它们才被看作是注释符。

第二种注释符是双斜线（//），它可用来注释一个单行，程序行中注释符右边的内容都将被当作注释而被编译器忽略。例如，下面的 **Screen** 类使用了两种注释：

```
/*
 * 这是我们第一次看到C++的类定义
 * 类可用于基于对象和
 * 面向对象编程中，Screen类的
 * 实现代码在第13章中
 */

class Screen {
    // 此部分称为类体
public:
    void home(); // 将光标移到 0,0
    void refresh(); // 重绘Screen
private:
    /* 类支持“信息隐藏” */
    /* 信息隐藏限制了程序 */
    /* 对类的内部表示（其数据）的 */
    /* 访问，这是用“private” */
    /* 来表示的 */

    // private数据省略...
};
```

大多数程序往往包含两种格式的注释。多行的说明通常被放在注释对中，半行或单行的注释则由双斜线指出。

1.5 输入/输出初步

C++的输入/输出功能由输入/输出流（`iostream`）库提供。输入/输出流库是 C++ 中一个面向对象的类层次结构，也是标准库的一部分。

终端输入，也被称为**标准输入**（`standard input`），与预定义的 `iostream` 对象 `cin`（发音为 *see-in*）绑定在一起。直接向终端输出，也被称为**标准输出**（`standard output`），与预定义的 `iostream` 对象 `cout`（发音为 *see-out*）绑定在一起。第三个预定义 `iostream` 对象 `cerr`（发音为 *see-err*）称为**标准错误**（`standard error`），也与终端绑定。`Cerr` 通常用来产生给程序用户的警告或错误信息。

任何要想使用 `iostream` 库的程序必须包含相关的系统头文件：

```
#include <iostream>
```

输出操作符 `<<` 用来将一个值导向到标准输出（`cout`）或标准错误（`cerr`）上。例如：

```
int v1, v2;
// ...
cout << "The sum of v1 + v2 = ";
cout << v1 + v2;
cout << '\n';
```

双字符序列 “`\n`” 表示换行符（`newline`）。输出换行符时，它结束当前的行，并将随后的输出导向到下一行。除了显式地使用换行符外，我们还可以使用预定义的 `iostream` 操纵符（`manipulator`）`endl`。

操纵符在 `iostream` 上执行的是一个操作，而不只是简单地提供数据。例如，`endl` 在输出流中插入一个换行符，然后刷新输出缓冲区。我们一般不写：

```
cout << '\n';
```

而是写成：

```
cout << endl;
```

（预定义 `iostream` 操纵符将在第 20 章中讨论。）

连续出现的输出操作符可以连接在一起，例如：

```
cout << "The sum of v1 + v2 = " << v1 + v2 << endl;
```

连续的输出操作符按顺序应用在 `cout` 上。为了便于阅读，连接在一起的输出操作符，可以分写在几行。下面的三行组成一条输出语句：

```
cout << "The sum of "
     << v1 << " + "
     << v2 << " = " << v1 + v2 << endl;
```

类似地，输入操作符（`>>`）用来从标准输入读入一个值。例如：

```
string file_name;
// ...
cout << "Please enter the file to be opened: ";
cin >> file_name;
```

连续出现的输入操作符，也可以连接起来。例如：

```
string ifile, ofile;
// ...
cout << "Please enter input and output file names: ";
cin >> ifile >> ofile;
```

怎样读入未知个数的输入值呢？在 1.2 节结束的时候，我们已经做过。请看下面的代码序列：

```
string word;
while ( cin >> word )
    // ...
```

在 `while` 循环中，每次迭代都从标准输入读入一个字符串，直到所有的串都读进来。当到达文件结束处（`end-of-file`）时，条件：

```
( cin >> word )
```

为假（第 20 章将解释这是如何发生的）。下面是使用这段代码序列的一个例子：

```
#include <iostream>
#include <string>

int main()
{
    string word;

    while ( cin >> word )
        cout << "word read is: " << word << '\n';

    cout << "ok: no more words to read: bye!\n";
    return 0;
}
```

下面是 James Joyce 的小说《Finnegans Wake》的前五个词：

```
riverrun, past Eve and Adam's
```

从键盘上输入这些词，程序的输出是：

```
word read is: riverrun,
word read is: past
word read is: Eve
word read is: and
word read is: Adam's
word read is: ok: no more words to read: bye!
```

（在第 6 章，我们将会看到怎样从各种输入字符串中删除标点符号。）

1.5.1 文件输入和输出

`iostream` 库也支持文件的输入和输出。所有能应用在标准输入和输出上的操作符，也都可以应用到已经被打开的输入或输出（或两者兼有）文件上。为了打开一个文件供输入或输出，除了 `iostream` 头文件外，还必须包含头文件：

```
#include <fstream>
```


为了打开一个输出文件，我们必须声明一个 `ofstream` 类型的对象：

```
ofstream outfile( "name-of-file" );
```

为了测试是否已经成功地打开了一个文件，我们可以写出这样的代码：

```
// 如文件不能打开值为false
if ( ! outfile )
    cerr << "Sorry! We were unable to open the file!\n";
```

类似地，为了打开一个文件供输入，我们必须声明一个 `ifstream` 类型的对象：

```
ifstream infile( "name of file" );
if ( ! infile )
    cerr << "Sorry! We were unable to open the file!\n";
```

下面是一个简单的程序。它从一个名为 `in_file` 的文本文件中读取单词，然后把每个词写到一个名为 `out_file` 的输出文件中，并且每个词之间用空格分开。

```
#include <iostream>
#include <fstream>
#include <string>
int main()
{
    ofstream outfile( "out_file" );
    ifstream infile( "in_file" );

    if ( ! infile ) {
        cerr << "error: unable to open input file!\n";
        return -1;
    }

    if ( ! outfile ) {
        cerr << "error: unable to open output file!\n";
        return -2;
    }

    string word;
    while ( infile >> word )
        outfile << word << ' ';

    return 0;
}
```

第 20 章将对 `iostream` 库进行全面的讨论，包括文件输入和输出。现在，我们对 C++ 提供的内容有了大致的了解，下一步，我们将通过使用类和模板设施把新的类型引入到语言中。

C++ 浏览

本章将首先讲述 C++ 对数组类型的支持。数组是相同类型元素的集合，例如整型数组，可能代表考试的分数；或者字符串数组，可能代表在文本文件中包含的单词。然后，我们会看一看内置数组类型的缺点，以及怎样通过提供一个基于对象的 `Array` 类型的类来改善这些缺点。在这之后再将其扩展成一个含有特化的 `Array` 子类型的面向对象层次结构。最后，我们还要比较一下 `Array` 类型与 C++ 标准库的 `vector` 类，并第一次了解泛型算法。沿着这条路，我们将进一步了解 C++ 对异常处理、模板和名字空间的支持。

2.1 内置数组数据类型

正如第 1 章所介绍的那样，C++ 为基本算术数据类型（如整数类型）提供了内置的支持。如：

```
// 声明一个整型对象，ival
// 初始化为初始值1024
int ival = 1024;
```

同时，它也支持双精度和单精度浮点数据类型：

```
// 声明一个双精度浮点对象，dval
// 初始化为初始值3.14159
double dval = 3.14159;
```

```
// 声明一个单精度浮点对象，fval
// 初始化为初始值3.14159
float fval = 3.14159;
```

此外，C++ 还支持布尔类型以及用来存放字符集中单个元素的字符类型。

C++ 为算术数据类型提供了赋值、一般算术运算以及关系运算的内置支持。算术运算如加、减、乘、除，关系运算如等于、不等于、小于和大于。例如：

```
int ival2 = ival + 4096;    // addition(加)
int ival3 = ival2 - ival;   // subtraction(减)
```

```
dval = fval * ival; // multiplication(乘)
ival = ival3 / 2;   // division(除)

bool result = ival2 == ival3;    // equality(等于)
result = ival2 + ival != ival3;  // inequality(不等于)
result = fval + ival2 < dval;    // less-than(小于)
result = ival > ival2;           // greater-than(大于)
```

另外，标准库还支持基本类抽象的组合，例如字符串、复数（在 2.7 节之前我们暂时不考虑标准库提供的 `vector` 类）。

在内置数据类型与标准库类的类型之间是复合类型（compound type），特别是指针和数组类型（我们将在 2.2 节中介绍指针类型）。

数组（array）是一种顺序容器，它包含单一类型的元素。例如，序列：

```
0 1 1 2 3 5 8 13 21
```

代表菲波那契数列的前 9 个数。（只要给出最前面两个元素，后面的元素依次可以由前面两个元素相加得出。）

为了定义和初始化一个数组以便存放这些数，我们可以这样写：

```
int fibon[ 9 ] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

数组对象的名字是 `fibon`，这是一个包含 9 个元素的整型一维（dimension）数组。第一个元素为 0，最后一个为 21，通过数组下标（subscript）操作符，我们可以以索引方式访问数组的元素。例如，为了读取数组的第一个元素，我们可能会这样写：

```
int first_elem = fibon[ 1 ]; // 不正确
```

不幸的是，这是不正确的，虽然它本身并没有语言错误。

在 C++ 中，数组下标从 0 开始，而不是 1。在位置 1 上的元素实际上是数组的第二个元素。类似地，位置 0 上的元素才是第一个元素，为了访问数组的最后一个元素，我们总是要索引到数组长度-1 的位置处的元素。

```
fibon[ 0 ]; // 第一个元素
fibon[ 1 ]; // 第二个元素
...
fibon[ 8 ]; // 最后一个元素
fibon[ 9 ]; // 喔! ...
```

用 9 索引的元素不是数组的元素，`fibon` 的 9 个元素由下标 0~8 索引。初学者常见的错误就是用位置 1~9 来索引。事实上，这非常普遍，以至于这个错误有了自己专用的名字：一位偏移（off-by-one）错误。

通常，我们用循环来遍历数组中的元素。例如，下面的程序初始化了一个包含 10 个元素的数组，其值分别从 0 到 9。然后再在标准输出上以降序输出：

```
int main()
{
    int ia[ 10 ];
    int index;

    for ( index = 0; index < 10; ++index )
        // ia[0] = 0, ia[1] = 1, 等等
```

```
        ia[ index ] = index;

    for ( index = 9; index >= 0; --index )
        cout << ia[ index ] << " ";

    cout << endl;
}
```

两个循环各迭代 10 次，关键字 `for` 后面的三条语句控制循环，第一条语句向 `index` 赋值 0：

```
index = 0;
```

它只在循环开始真正工作之前被执行一次。第二条语句：

```
index < 10;
```

表示循环的结束条件（stopping condition），它开始真正的循环序列。如果它的值为真，则与 `for` 循环相关联的语句（或一组语句）将执行。如果它的值为假，则循环终止。在本例中，每次当 `index` 值小于 10 时，会执行如下语句：

```
ia[ index ] = index;
```

第三条语句：

```
++index
```

是对一个算术对象加一的简短写法，它等价于：

```
index = index + 1;
```

它在与 `for` 循环相关联的语句执行之后才执行，这里与 `for` 循环相关联的语句是“把 `index` 的值赋给以 `index` 为下标的元素”。这第三条语句的执行完成了 `for` 循环的一次迭代。序列的每次重复都重新测试条件，条件为假时，循环终止。（第 5 章将详细介绍 `for` 循环。）第二个循环则以相反的顺序输出这些值。

虽然 C++ 对数组类型提供了内置支持，但是这种支持仅限于“用来读写单个元素”的机制。C++ 不支持数组的抽象（abstraction），也不支持对整个数组的操作，我们有时会希望对整个数组进行操作，例如，把一个数组赋值给另外一个数组、对两个数组进行相等比较，或者想知道数组的大小（size）。例如，给出两个数组，我们不能用赋值操作符把一个数组拷贝到另一个中去：

```
int array0[ 10 ], array1[ 10 ];
...
// 错误：不能直接把一个数组赋值给另一个数组
array0 = array1;
```

如果我们希望把一个数组赋值给另外一个，则必须自己写程序，按顺序拷贝每个元素：

```
for ( int index = 0; index < 10; ++index )
    array0[ index ] = array1[ index ];
```

而且，数组类型本身没有自我意识，它不知道自己的长度，我们必须另外记录数组本身的这些信息。当我们希望把数组作为一个参数传递给一个函数的时候，问题就出现了。在 C++ 中，数组不同于整数类型和浮点数类型，它不是 C++ 语言的一等（first-class）公民。数组是

从 C 语言中继承来的，它反映了数据与对其进行操作的算法的分离，而这正是过程化程序设计的特征。在本章后面部分，我们将会了解一些不同的策略，通过这些策略可以使数组具有一些额外的公民特权。

练习 2.1

为什么内置数组类型不支持数组之间的赋值，支持这种操作需要什么信息？

练习 2.2

你认为作为一等公民的数组应该支持什么操作？

2.2 动态内存分配和指针

在开始基于对象的设计之前，我们需要暂时偏离一下主题，先来介绍一下 C++ 程序内存分配的问题。原因是，我们必须首先介绍在程序执行期间怎样申请和访问内存，否则，没有办法真正实现我们的设计（并且展示理想的 C++ 代码），这是本小节的目的。

在 C++ 中，对象可以静态分配——即编译器在处理程序源代码时分配，也可以动态分配——即程序执行时调用运行时刻库函数来分配。这两种内存分配方法的主要区别是效率与灵活性之间的平衡准则不同。由于静态内存分配是在程序执行之前进行的，因而效率比较高。但是，它缺少灵活性，它要求在程序执行之前就知道所需内存的类型和数量。例如，利用静态分配的字符串数组，我们无法很容易地处理和存储任意的文本文件。一般来说，存储未知数目的元素需要动态内存分配的灵活性。

到目前为止，所有的内存分配都是静态的。例如，以下定义：

```
int ival = 1024;
```

指示编译器分配足够的存储区以存放一个整型值，该存储区与名字 `ival` 相关联。然后，用数值 1024 初始化该存储区。这些工作都在程序执行之前完成。

有两个值与对象 `ival` 相关联：一个是它包含的值——本例中为 1024，另一个是存放这个值的存储区的地址。在 C++ 中，这两个值都可以被访问。当我们写出下面的代码时：

```
int ival2 = ival + 1;
```

我们访问 `ival` 所包含的值，并把它加 1，然后再用该新值初始化 `ival2`。在本例中，`ival2` 初始值为 1025。怎样访问和存储内存地址呢？

C++ 支持用指针类型来存放对象的内存地址值。例如，为了声明一个能存放 `ival` 内存地址的指针类型，我们可以这样写：

```
// 一个指向int类型的指针
int *pint;
```

C++ 预定义了一个专门的取地址（address-of）操作符（`&`）。当我们把它应用在一个对象上时，返回的是对象的地址值。因此，为了将 `ival` 内存地址值赋给 `pint`，我们可以这样写：

```
int *pint;
pint = &ival; // 把ival的地址赋给pint
```

为了访问 `pint` 所指向的实际对象，我们必须先用解引用（dereference）操作符（*）来解除 `pint` 的引用（dereference `pint`）。例如，下面我们通过 `pint` 间接地给 `ival` 加 1：

```
// 通过pint间接地给ival加1
*pint = *pint + 1;
```

它等价于下面直接对 `ival` 操作的语句：

```
// 直接给ival加1
ival = ival + 1;
```

在本例中，使用指针间接地操作 `ival` 没有什么实际的好处，这样做比直接操作 `ival` 的效率要低，而且又容易出错。我们只是用它来简单地介绍一下指针。在 C++ 中，指针的主要用处是管理和操纵动态分配的内存。

静态与动态内存分配的两个主要区别是：

1. 静态对象是有名字的变量，我们直接对其进行操作。而动态对象是没有名字的变量，我们通过指针间接地对它进行操作。稍后我们会看到一个例子。

2. 静态对象的分配与释放由编译器自动处理。程序员需要理解这一点，但不需要做任何事情。相反，动态对象的分配与释放，必须由程序员显式地管理，相对来说比较容易出错，它通过 `new` 和 `delete` 两个表达式来完成。

对象的动态分配可通过 `new` 表达式的两个版本之一来完成。第一个版本用于分配特定类型的单个对象。例如：

```
int *pint = new int( 1024 );
```

分配了一个没有名字的 `int` 类型的对象，对象初始值为 1024。然后，表达式返回对象在内存中的地址。接着，这个地址被用来初始化指针对象 `pint`。对于动态分配的内存，惟一的访问方式是通过指针间接地访问。

`new` 表达式的第二个版本，用于分配特定类型和维数的数组。例如：

```
int *pia = new int[ 4 ];
```

分配了一个含有四个整数元素的数组。不幸的是，我们没有办法给动态分配的数组的每个元素显式地指定一个初始值。

分配动态数组时一个常令人迷惑的问题是，返回值只是一个指针，与分配单一动态对象的返回类型相同。例如，`pint` 与 `pia` 的不同之处在于，`pia` 拥有四元素数组的第一个元素的地址，而 `pint` 只是简单地包含单一对象的地址。当用完了动态分配的对象或对象的数组时，我们必须显式地释放这些内存。我们可以通过使用 `delete` 表达式的两个版本之一来完成这件事情，而释放之后的内存则可以被程序重新使用。单一对象的 `delete` 表达式形式如下：

```
// 删除单个对象
delete pint;
```

数组形式的 `delete` 表达式如下：

```
// 删除一个对象数组
delete [] pia;
```

如果忘了删除动态分配的内存，又会怎么样呢？如果真的如此，程序就会在结束时出现内存泄漏（memory leak）的问题。内存泄漏是指一块动态分配的内存，我们不再拥有指向这

块内存的指针，因此我们没有办法将它返还给程序供以后重新使用。（现在大多数系统提供识别内存泄漏的工具，可以向系统管理员咨询。）

对指针类型和动态内存分配讲得这么快，可能会留下很多应该回答的问题。但是，动态内存分配和指针操作是 C++ 实际编程基础的一个方面，我们不想推迟到后面再介绍它。在本章接下去的介绍中，我们将在基于对象的与面向对象的 Array 类的实现中，看到它的用法。

8.4 节将详细介绍动态内存分配以及 new 与 delete 表达式的用法。

练习 2.3

说出下面定义四个对象之间的区别：

```
(a) int ival = 1024;      (c) int *pi2 = new int( 1024 );
(b) int *pi = &ival;      (d) int *pi3 = new int[ 1024 ];
```

练习 2.4

下面的代码段是做什么的，有什么严重错误？（注意，指针 pia 的下标操作符的用法是正确的，在 3.9.2 节中我们会解释其理由。）

```
int *pi = new int( 10 );
int *pia = new int[ 10 ];

while ( *pi < 10 ) {
    pia[ *pi ] = *pi;
    *pi = *pi + 1;
}

delete pi;
delete [] pia;
```

2.3 基于对象的设计

在本节中，我们将使用 C++ 的类机制来设计和实现一个数组抽象。我们最初的实现只支持一个整型数组。以后，我们将用模板机制对这种抽象进行扩展，使其能够支持无限数目的数据类型。

第一步，我们需要决定数组应该提供哪些操作。尽管我们希望能提供所有的操作，但是我们却不能一次提供所有的功能。下面是第一步所支持操作的集合：

1. 数组类的实现中有内置的自我意识。首先，它知道自己大小。
2. 数组类支持数组之间的赋值、以及两个数组之间的相等和不相等的比较操作。
3. 数组类应该支持对其所含的值进行下列查询操作：数组中最小值是什么？最大值是什么？某个特殊的值是否在数组中？如果存在，它的第一个位置的索引是什么？
4. 数组类支持自排序。为了便于讨论，假定存在一群用户，他们认为数组支持排序的功能很重要，而另外一些人对此却不以为然。

除了支持数组操作，还必须支持数组本身的机制，包括：

5. 能够指定长度，以此来创建数组（这个值无需在编译时刻知道）。

6. 能够用一组值初始化数组。

7. 能够通过一个索引来访问数组中的单个元素。为便于讨论, 假设用户强烈要求用数组下标操作符来实现这项功能。

8. 能够截获并指出错误的索引值。假设我们认为这很有必要, 所以没有询问用户的想法。我们认为这是一个设计良好的数组所必须实现的。

我们与潜在用户的讨论已经引起了极大的热情, 现在我们要真正实现它。但是怎样把这个设计转换成 C++ 代码呢? 支持基于对象设计的类的一般形式如下:

```
class classname {  
public:  
    // 公有操作集合  
private:  
    // 私有实现代码  
};
```

这里, `class`、`public` 和 `private` 是 C++ 语言的关键字, `classname` 是用户定义的标识符, 它用来命名这个类, 以便在后面引用该类。我们将前面设计的类命名为 `IntArray`, 等到我们使它支持的数据类型更广泛时, 再将它改名为 `Array`。

类名代表的是一个新的数据类型, 我们可以用它来定义这种类型的对象, 就像用内置的类型定义对象一样。例如:

```
// 单个 IntArray 类对象  
IntArray myArray;  
  
// 指向 IntArray 类对象的指针  
IntArray *pArray = new IntArray;
```

类定义包括两个部分: 类头 (class head), 由关键字 `class` 与相关联的类名构成。类体 (class body), 由花括号括起来, 以分号结束。类头本身也用作类的声明。例如:

```
// 在程序中声明 IntArray 类, 但是不提供定义  
class IntArray;
```

类体包含成员定义, 以及访问标签, 如 `public` 和 `private`。类的成员包括“该类能执行的操作”和“代表类抽象所必需的数据”。这些操作称为成员函数 (member function) 或方法 (method)。对于 `IntArray` 类来说, 它由以下的内容构成:

```
class IntArray {  
public:  
    // 相等与不相等操作: #2b  
    bool operator==( const IntArray& ) const;  
    bool operator!=( const IntArray& ) const;  
  
    // 赋值操作符: #2a  
    IntArray& operator=( const IntArray& );  
  
    int size() const;    // #1  
    void sort();        // #4  
  
    int min() const;    // #3a  
    int max() const;    // #3b
```



```
// 如值在数组中找到
// 返回第一次出现的索引
// 否则返回-1
int find( int value ) const; // #3c

private:
    // 私有实现代码
};
```

成员函数右边的数字对应着我们前面定义的规范表中的条目，我们现在不打算解释参数表中的 `const` 修饰符或参数表后面的 `const` 修饰符，现在还没必要详细解释。但是在实际的程序中，它们还是必需的。

通过使用两个成员访问操作符（member access operator）中的一个，我们可以调用一个有名字的成员函数，如 `min()`。这两个操作符为：用于类对象的点操作符（`.`），以及用于类对象指针的箭头操作符（`->`）。例如，为了在数组 `myArray` 类对象中找到最小值，我们可以这样写：

```
// 用myArray数组中的最小元素来初始化min_val
int min_val = myArray.min();
```

为了在动态分配的 `IntArray` 对象中查找最大值，我们可以这样写：

```
int max_val = pArray->max();
```

[是的，我们还没介绍怎样用一个长度与一组值来初始化 `IntArray` 类对象。有个特殊的称为构造函数（constructor）的成员函数可以完成这些工作，我们将会简要地介绍它。]

把操作符应用在这些类对象上的方式与应用在内置数据类型上的对象一样。下面，给出两个 `IntArray` 对象：

```
IntArray myArray0, myArray1;
```

赋值操作符可以这样应用：

```
// 调用拷贝赋值成员函数：
// myArray0.operator=( myArray1 )
myArray0 = myArray1;
```

等于操作符的调用如下所示：

```
// 调用等于成员函数：
// myArray0.operator==( myArray1 )
if ( myArray0 == myArray1 )
    cout << "!!our assignment operator works!\n";
```

关键字 `private` 和 `public` 控制对类成员的访问。出现在类体中有公（`public`）部分的成员，在一般程序的任何地方都可以访问它们。出现在私有（`private`）部分的成员只能在该类的成员函数或友元（`friend`）中被访问。（我们要到 15.2 节才会解释友元。）

一般来说，公有成员提供了该类的公有接口（`public interface`）——即实现了这个类的行为的操作集合。它包括该类的所有成员函数，或者只包括其中一个子集。私有成员提供私有实现代码（`private implementation`）——即存储信息的数据。

这种“类的公共接口与私有实现代码的分离”，被称为信息隐藏（`information hiding`）。信息隐藏是软件工程中一个非常重要的概念，在后面的章节中将有详细的介绍。简要说来，

它为程序提供了两个主要好处：

1. 如果类的私有实现代码需要修改或扩展，那么，只有相对很小一部分“要求访问这些实现代码的成员函数”需要修改。而许多使用该类的用户程序无需修改，但是要求重新编译。（6.18 节将演示这个过程。）

2. 如果类的私有实现代码有错误，那么通常需要检查的代码数量只局限在相对较少的需要访问这些实现代码的成员函数上，而无需检查整个程序。

哪些数据成员是 `IntArray` 必需的呢？当声明一个 `IntArray` 对象时，用户会指定数组大小，我们需要存储它。因此，我们将定义一个数据成员来做到这一点。另外，我们需要实际分配并存储底层的数组，这将通过 `new` 表达式来实现，我们将定义一个指针数据成员来存储 `new` 表达式返回的地址值：

```
class IntArray {
public:
    // ...
    int size() const { return _size; }
private:
    // 内部数据
    int _size;
    int *ia;
};
```

由于我们把 `_size` 放在类的私有区内，因此我们有必要声明一个公有成员函数，以便允许用户访问它的值。由于 C++ 不允许成员函数与数据成员共享同一个名字，所以在这样的情况下，一般的习惯是在数据成员名字前面加一个下划线（`_`）。因此，我们有了公有访问函数 `size()` 和私有数据成员 `_size`。在本书以前的版本中，我们在访问函数前加上 `get` 或 `set`。实践证明这样做有些累赘。

尽管这种公有访问函数的用法允许用户读取相应的值，但是这种实现似乎还有些根本的错误，至少第一眼看上去是这样的。你看出来了吗？考虑下面的语句：

```
IntArray array;
int array_size = array.size();
```

还有：

```
// 假设_size是public的
int array_size = array._size;
```

将 `array_size` 初始化为数组的维数。但是很显然，第一个例子需要一个函数调用，而第二个只需直接访问内存就行了。一般来说，函数调用比直接访问内存的开销要大得多。因而信息隐藏是否给程序的执行效率增加了严重的额外负担，或许是阻碍性的负担呢？幸运的是，在一般情况下，回答是：不。

C++ 提供的解决方案是内联函数（`inline function`）机制。内联函数在它的调用点上被展开。一般来说，内联函数不会引入任何函数调用。^①例如，在 `for` 循环的条件子句中的 `size()` 调用：

^① 然而，实际并不总是这样的。对于编译器来说，内联函数是一种请求，而不是一种保证。参见 7.6 节的讨论。

```
for ( int index = 0; index < array.size(); ++index )  
    // ...
```

并没有真的被调用 `_size` 次，而是在编译期间被内联扩展为下面的一般形式：

```
// array.size() 的内联扩展  
for ( int index = 0; index < array._size; ++index )  
    // ...
```

在类定义中被定义的成员函数 [如 `size()`] 会被自动当作是内联函数。此外，我们也可以使用 `inline` 关键字显式地要求一个函数被视为内联函数。（7.6 节中有更多关于内联函数的说明。）

到目前为止，我们已经提供了 `IntArray` 类所要求的操作（前面的 1~4 项），但是还没有提供初始化机制和数组中单个元素的访问方式（5~8 项）。

程序设计中的一个常见错误是使用事先并没有被正确初始化的对象。实际上，这是一个极为常见的错误，所以 C++ 为用户定义的类提供了一种自动初始化机制：类构造函数（class constructor）。

构造函数是一种特殊的类成员函数，专门用于初始化对象。如果构造函数被定义了，那么在类的每个对象第一次被使用之前，该构造函数就被自动应用在对象上。构造函数由谁来定义呢？类的提供者——也就是我们来定义构造函数。为一个类确定必要的构造函数是程序设计不可缺少的一部分。

为了定义一个构造函数，我们只要给它与类相同的名字即可。另外，我们不能给构造函数指定返回值。但是可以给类定义多个构造函数，尽管它们都具有相同的名字，但只要编译器能够根据参数表区分它们就行。

更一般地，C++ 支持被称为函数重载（function overloading）的机制。函数重载允许两个或更多个函数使用同一个名字，限制条件是它们的参数表必须不同：参数类型不同，或参数的数目不同。根据不同的参数表，编译器就能够判断出，对某个特定的调用应该选择哪一个版本的重载函数。下面是一组合法的 `min()` 重载函数（这些函数也可以是类成员函数）：

```
// 一组 min() 重载函数  
// 每个函数都有一个特有的参数表  
#include <string>;  
  
int    min( const int *pia, int size );  
int    min( int, int );  
int    min( const char *str );  
char   min( string );  
string min( string, string );
```

重载函数在运行时刻的行为与非重载函数完全一样，主要的负担是在编译时刻用来决定应该调用哪个实例所需要的时间。如果 C++ 不提供函数重载支持，那么我们就必须为程序中每个函数都要提供一个独一无二的名字。（第 9 章将详细讨论函数重载的内容。）

我们为 `IntArray` 类指定了下面三个构造函数：

```
class IntArray {  
public:  
    explicit IntArray( int sz = DefaultArraySize );  
    IntArray( int *array, int array_size );
```

```

    IntArray( const IntArray &rhs );
    // ...

private:
    static const int DefaultArraySize = 12;
    // ...
};

```

构造函数:

```

    IntArray( int sz = DefaultArraySize );

```

被称为缺省构造函数 (default constructor)，因为它不需要用户提供任何参数。(我们现在不打算解释这个缺省构造函数声明中出现的关键字 `explicit`，之所以显示它仅仅是为了完整性。) 如果程序员提供参数，则该值将被传递给构造函数。例如:

```

    IntArray array1( 1024 );

```

将参数 1024 传递给构造函数。另一方面，如果用户不指定长度，那么构造函数将使用 `DefaultArraySize` 的值。例如:

```

    IntArray array2;

```

将导致用 `DefaultArraySize` 的值来调用构造函数。(被声明为 `static` 的数据成员是一类特殊的共享数据成员，无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份。这是在类的所有对象之间共享数据的一种方式。13.5 节有全面的讨论。)

下面是缺省构造函数的一个简化实现版本，简化到没有考虑出错的可能性。(可能出现什么错误呢？本例中有两个：首先，提供给程序的动态内存不是无限的，因此，`new` 表达式有可能失败，2.6 节中将介绍如何处理这种情况。第二，传递给参数 `sz` 的值可能是无效的，例如，负数或 0，或者一个很大的值，以至于无法存储在 `int` 类型的变量中。)

```

IntArray::
IntArray( int sz )
{
    // 设置数据成员
    _size = sz;
    ia = new int[ _size ];

    // 初始化内存
    for ( int ix=0; ix < _size; ++ix )
        ia[ ix ] = 0;
}

```

这是我们第一次在类体的外面定义类的成员函数。惟一的语法区别是要指出成员函数属于哪个类，这可以通过类域操作符 (class scope operator) 来实现:

```

IntArray::

```

双冒号 (::) 操作符被称为域操作符 (scope operator)。当与一个类名相连的时候，像上面例子中那样，它就成为一个类域操作符。我们可以非正式地把域看作是一个可视窗口。全局域的对象在它被定义的整个文件里 (一直到文件末尾) 都是可见的。在一个函数内被定义的对象是局域的 (local scope)，它只在定义其的函数体内可见。每个类维持一个域，在这个域之外，它的成员是不可见的。类域操作符告诉编译器，后面的标识符可在该类的范围内

被找到。本例中：

```
IntArray::
IntArray( int sz )
```

告诉编译器 `IntArray()` 函数被定义为 `IntArray` 类的成员。（尽管程序域不是我们现在应该关注的事情，但是最终我们还是理解域的概念。在第 8 章中我们将详细讲解程序域，而类域将在 13.9 节中特别讨论。）

`IntArray` 类的第二个构造函数用内置的整数数组初始化一个新的 `IntArray` 类对象。它需要两个参数：一个是实际的数组，另一个参数指明数组的大小。例如：

```
int ia[10] = {0,1,2,3,4,5,6,7,8,9};
IntArray ia3(ia,10);
```

这个构造函数的实现几乎与第一个构造函数相同。（我们又一次没有保护自己的代码会出错。）

```
IntArray::
IntArray( int *array, int sz )
{
    // 设置数据成员
    _size = sz;
    ia = new int[ _size ];

    // 拷贝数据成员
    for ( int ix=0; ix < _size; ++ix )
        ia[ ix ] = array[ ix ];
}
```

最后一个 `IntArray` 构造函数用另外一个 `IntArray` 对象来初始化当前的 `IntArray` 对象，对于下面两种形式，无论是哪一种，它都将被自动调用：

```
IntArray array;

// 等价的初始化方式
IntArray ia1 = array;
IntArray ia2( array );
```

这种构造函数被称为类的**拷贝构造函数**（`copy constructor`）。在后面的章节中我们将会看到更多的示例。下面的实现再次忽略了可能出现的运行时刻程序异常：

```
IntArray::
IntArray( const IntArray &rhs )
{ // 拷贝构造函数
    _size = rhs._size;
    ia = new int[ _size ];

    for ( int ix=0; ix < _size; ++ix )
        ia[ ix ] = rhs.ia[ ix ];
}
```

本例引入了一种新的复合类型：引用（`reference`），即 `IntArray &rhs`。引用是一种没有指针语法的指针。（因此，我们写成 `rhs._size`，而不是 `rhs->_size`。）与指针一样，引用提供对对象的间接访问。（3.6 节中我们将对指针和引用作更多的介绍。）

注意，这三个构造函数都是以相似的方式来实现的。一般来说，当两个或多个函数重复相同的代码时，就会将这部分代码抽取出来，形成独立的函数，以便共享。以后，如果需要改变这些实现，则只需改变一次。而且，这种实现的共享本质更容易为大家所理解。

怎么样把构造函数中的代码抽取出来形成独立的共享函数呢？下面是一种可能的实现：

```
class IntArray {
public:
    // ...
private:
    void init( int sz, int *array );
    // ...
};

void
IntArray::
init( int sz, int *array )
{
    _size = sz;
    ia = new int[ _size ];

    for ( int ix=0; ix < _size; ++ix )
        if ( ! array )
            ia[ ix ] = 0;
        else ia[ ix ] = array[ ix ];
}
```

三个构造函数可重写为：

```
IntArray::IntArray( int sz ){ init( sz, 0 ); }
IntArray::IntArray( int *array, int sz )
    { init( sz, array ); }

IntArray::IntArray( const IntArray &rhs )
    { init( rhs._size, rhs.ia ); }
```

类机制还支持特殊的析构成员函数（**destructor member function**）。每个类对象在被程序最后一次使用之后，它的析构函数就会被自动调用。我们通过在类的名字前面加一个波浪线（~）来标识析构函数。一般地，析构函数会释放在类对象使用和构造过程中所获得的资源。例如，在 IntArray 的析构函数中，它会删除构造时分配的内存。（我们将在第 14 章详细讨论构造函数和析构函数。）下面是我们的实现：

```
class IntArray {
public:
    // 构造函数
    explicit IntArray( int size = DefaultArraySize );
    IntArray( int *array, int array_size );
    IntArray( const IntArray &rhs );

    // 析构函数
    ~IntArray() { delete [] ia; }

    // ...

private:
```

```
// ...
};
```

除非用户能够很容易地通过索引访问单个元素，否则数组类就没有更实际的用处。例如，我们的类需要支持下面的一般用法：

```
IntArray array;
int last_pos = array.size()-1;

int temp = array[ 0 ];
array[ 0 ] = array[ last_pos ];
array[ last_pos ] = temp;
```

我们通过提供“专用于一个类的下标操作符实例”，来支持索引 IntArray 类对象。下面是支持这种用法的一个实现：

```
#include <cassert>
int&
IntArray::
operator[]( int index )
{
    assert( index >= 0 && index < size );
    return ia[ index ];
}
```

一般而言，C++语言支持操作符重载（operator overloading），这样就可以为特定的类（类型）定义新的操作符实例。典型地，类提供一个或多个赋值操作符、等于操作符，可能还有一个或多个关系操作符，以及 iostream 输入和输出操作符。（3.15 节有关于操作符重载的进一步的说明。在第 15 章我们将详细讨论操作符重载。）

类定义以及相关的常数值或 typedef 名通常都存储在头文件中，并且头文件以类名来命名。因此，假如我们创建一对头文件：IntArray.h 和 Matrix.h，则所有打算使用 IntArray 类或 Matrix 类的程序就都必须包含相关的头文件。

类似地，不在类定义内部定义类成员函数都存储在与类名同名的程序文本文件中。例如，我们将创建一对程序文本文件：IntArray.C 和 Matrix.C，用来存储相关类的成员函数。（记住，程序文本文件的后缀因编译系统而不同，你应该检查自己的系统所使用的命名习惯。）这些函数不用随每个使用相关类的程序而重新编译，这些成员函数经过预编译之后被保存在类库中，iostream 库就是这样一个例子。

练习 2.5

C++类的关键特征是接口与实现的分离。接口是一些“用户可以应用到类对象上的操作”的集合。它由三部分构成：这些操作的名字、它们的返回值，以及它们的参数表。一般地，这些就是该类用户所需要知道的全部内容。私有实现包括为支持公有接口所必需的算法和数据。理想情况下，即使类的接口增长了，它也不用变得与以前的版本不相兼容。另一方面，在类的生命周期内其实现可以自由演化。从下面选择一个抽象（指类），并为该类编写一个公共接口。

- | | | |
|-------------|------------|-------------|
| (a) Matrix | (c) Person | (e) Pointer |
| (b) Boolean | (d) Date | (f) Point |

练习 2.6

构造函数和析构函数是程序员提供的函数，它们既不构造也不销毁类的对象（编译器自动把它们作用到这些对象上）。因此构造函数（constructor）和析构函数（destructor）这两个词多少有些误导，当我们写：

```
int main() {
    IntArray myArray( 1024 );
    // ...
    return 0;
}
```

在构造函数被应用之前，用于维护 myArray 中数据成员的内存已经被分配了。实际上，编译器在内部把程序转换成如下的代码（注意这不是合法的 C++代码^②）：

```
int main() {
    IntArray myArray;

    // 伪C++代码——应用构造函数
    myArray.IntArray::IntArray( 1024 );
    // ...
    // 伪C++代码——应用析构函数
    myArray.IntArray::~~IntArray();

    return 0;
}
```

类的构造函数主要用来初始化类对象的数据成员。析构函数主要负责释放类对象在生命期内申请到的所有资源。请定义在练习 2.5 中选择的类所需要的构造函数集，你的类需要析构函数吗？

练习 2.7

在练习 2.5 和练习 2.6 中，你差不多已经定义了使用该类的完整公有接口。（我们还需要定义一个拷贝赋值操作符，但是现在我们忽略这个事实——C++为“从一个类对象向另一个类对象赋值”提供了缺省支持。问题在于，缺省的行为常常是不够的。这将在 14.6 节中讨论。）写一个程序来实践在前面两个练习中定义的公有接口。用起来容易还是麻烦？你希望重写这些定义吗？你能在重写的同时保持兼容性吗？

2.4 面向对象的设计

max()与 min()函数的实现没有对数组元素的存储做特殊的假设，因此，我们需要检查数组的每个元素。如果我们要求所有的元素已经排序，则这两个操作就变得非常简单，只要索引第一个元素和最后一个元素即可。而且，如果已知元素已经排序，那么查找一个元素的存在就会更加高效。但是，对数组进行排序增加了 Array 类实现的复杂性。我们的设计出错了么？

^② 对于感兴趣的读者，可以参见本书的姐妹篇《Inside the C++ Object Model》，该书讨论了这方面的内容。

实际上，我们现在是否犯了错误与我们做出的选择息息相关。排序的数组是一种特殊的实现：需要时，它完全必要；否则，支持排序数组的额外开销就是一项负担。我们的实现是比较通用的，在大多数情况下是足够的。它支持更广阔范围内的用户。不幸的是，如果用户绝对需要排序数组的行为，那么我们的实现就不能提供支持。对用户来说，他没有办法对 `min()`、`max()` 以及 `find()` 这些函数比较通用的实现进行改写。实际上，我们选择的通用实现，并不适合特殊的环境。

另一方面，我们的实现对另外一类用户而言又针对性太强了：对索引的范围检查为每次访问元素增加了额外的负担。在设计中，我们没有考虑这样的开销（2.3 节中第 8 条），而是假设：如果结果不正确，那么速度再快也没有价值。但是，这种设计至少对于某一类主要用户：实时虚拟和虚拟现实提供商，就不成立。在这种情况下，数组代表复杂 3D 几何图形的顶点。场景飞快地变化，以至于一些偶然的错误不会被看到。但如果访问速度太慢了，那么实时效果就会被打破。我们实现的数组类虽然比没有范围检查的数组类会更安全，但是在这样的实时应用领域却不够实际。

我们怎样才能支持这三种用户的需要呢？解决的方案已经多多少少体现在代码中了。例如，范围检查局限在下标操作符中。去掉 `check_range()` 的调用，重新命名该数组。现在我们就有了两种实现：一个有范围检查，一个没有范围检查。进一步拷贝一份代码，并把它修改成针对已排序的数组。现在我们就有了对已排序数组的支持：

```
// 未排序，也没有边界检查
class IntArray{ ... };

// 未排序，但支持边界检查
class IntArrayRC{ ... };

// 已排序，但没有边界检查
class IntSortedArray{ ... };
```

这种方案的缺点是什么呢？

1. 我们必须维护三个包含大量重复代码的数组实现。我们更希望把这些公共代码只保留一份，然后由“这三个数组类（以及其他一些我们以后会选择支持的数组类）”共享。（比如，可能会是一个带有边界检查的排序数组。）

2. 由于三个数组实现是完全独立的类型，所以我们必须编写独立的函数来操作它们，尽管函数内的一般性操作都是相同的。例如：

```
void process_array( IntArray& );
void process_array( IntArrayRC& );
void process_array( IntSortedArray& );
```

我们希望只编写一个函数，它不但能接受现有的数组类，而且，还能够接受任意将来的数组类，只要同样的操作集合也能够应用到这些类上。

面向对象的程序设计方法正是为我们提供了这样一种能力。上面第 1 项可由继承（inheritance）机制提供。当一个 `IntArrayRC` 类（也就是一个带有范围检查的 `IntArray` 类）继承了 `IntArray` 类时，它就可以访问 `IntArray` 的数据成员和成员函数，而不要求我们维护两份代码拷贝。新的类只需提供实现其额外语义所必需的数据成员和成员函数。

在 C++ 中，被继承的类，如本例中的 `IntArray`，被称作基类（base class）。新类被称作

从基类派生 (derived) 而来, 我们把它叫做基类的派生类 (derived class) 或子类型 (subtype)。我们说 `IntArrayRC` 是一种有特殊行为的 `IntArray`, 它支持对索引值的范围检查。子类型与基类共享公共的接口 (common interface) —— 公有操作的公共集。由于共享公共接口, 允许了子类和基类在程序内部可互换使用, 而无需考虑对象的实际类型。从某种意义上来说, 公共接口封装了单个子类型中与类型相关的细节。类之间的类型/子类型关系形成了继承或派生层次关系 (inheritance or derivation hierarchy)。例如, 下面的非成员函数 `swap()` 把指向基类 `IntArray` 对象的引用作为第一个参数, 该函数交换索引 `i` 和 `j` 处的元素:

```
#include <IntArray.h>

void swap( IntArray &ia, int i, int j )
{
    int tmp = ia[ i ];
    ia[ i ] = ia[ j ];
    ia[ j ] = tmp;
}
```

下面是 `swap()` 函数的三个合法调用:

```
IntArray      ia;
IntArrayRC    iarc;
IntSortedArray ias;
// ok: ia是一个IntArray
swap( ia, 0, 10 );

// ok: iarc是IntArray的子类型
swap( iarc, 0, 10 );

// ok: ias也是IntArray的子类型
swap( ias, 0, 10 );

// error: string不是IntArray的子类型
string str( "not an IntArray!" );
swap( str, 0, 10 );
```

三个数组类都提供了自己的下标操作符实现。当然, 我们的要求是, 当调用如下函数时:

```
swap( iarc, 0, 10 );
```

`IntArrayRC` 的下标操作符被调用。当调用如下函数时:

```
swap( ias, 0, 10 );
```

`IntSortedArray` 下标操作符被调用等等。`swap()` 调用的下标操作符必须潜在地随着每次调用而改变, 它必须由被交换元素的数组的实际类型来决定。在 C++ 中, 这可以由虚拟函数 (virtual function) 机制来自动完成。

为使 `IntArray` 类能够被继承, 我们需要在语法上做一点小小的改变: 必须 (可选择的) 减少封装的层次, 以便允许派生类访问非公有的实现, 而且我们也必须显式地指明哪些函数应该是虚拟的。最重要的变化在于我们如何把一个类设计成为基类。

在基于对象的程序设计中, 通常类的提供者只有一个, 但是类的用户有许多个。提供者设计并且通常也会实现类。用户使用提供者提供的公有接口, 行为的分离可通过将类分成公

有与私有访问级别而反映出来。

在继承机制下有多个类的提供者：一个提供基类实现（可能还有一些派生类），另外一个或多个提供者在继承层次的生命周期内提供派生类，这种行为也是一种实现行为。子类的提供者经常（但并不总是）需要访问基类的实现。为了提供这种能力，同时还要防止对基类实现的一般性访问，C++提供了另外一个访问级别：**保护**（**protected**）级别。在类的保护区域内的数据成员和成员函数，不提供给一般的程序，只提供给派生类。（放在基类的私有区域内的成员只能供该类自己使用，派生类不能使用。）下面是修改过的 `IntArray` 类：

```
class IntArray {
public:
    // 构造函数
    explicit IntArray( int size = DefaultArraySize );
    IntArray( int *array, int array_size );
    IntArray( const IntArray &rhs );

    // 虚拟析构函数!
    virtual ~IntArray() { delete [] ia; }

    // 等于和不等子操作:
    bool operator==( const IntArray& ) const;
    bool operator!=( const IntArray& ) const;

    IntArray& operator=( const IntArray& );
    int size() const { return _size; }

    // 去掉了索引检查功能 ...
    virtual int& operator[](int index) { return ia[index]; }
    virtual void sort();

    virtual int min() const;
    virtual int max() const;
    virtual int find( int value ) const;

protected:
    // 参见13.5节的说明
    static const int DefaultArraySize = 12;
    void init( int sz, int *array );

    int _size;
    int *ia;
};
```

在面向对象与基于对象的设计中，指明一个类的成员是 **public** 的准则没有变化。重新设计的 `IntArray` 类将被用作基类，它仍然把构造函数、析构函数、下标操作符、`min()`和 `max()`等等声明为公有成员。这些成员继续提供公有接口，但现在接口不只为 `IntArray` 类服务，同时也为从它派生的整个继承层次服务。

非公有的成员到底该声明为 **protected** 还是 **private** 类成员是新的设计准则。如果希望防止派生类直接访问某个成员，我们就把该成员声明为基类的 **private** 成员。如果确信某个成员提供了派生类需要直接访问的操作或数据存储，而且通过这个成员，派生类的实现会更有效，则我们把该成员声明为 **protected**。对于 `IntArray` 类，我们已经将全部数据成员设置成 **protected**，

也就是实际上允许后续派生的类访问 `IntArray` 的实现细节。

为了把一个类设计成基类，要做的第二个设计考虑是找出类型相关的成员函数，并把这些成员函数标记为 `virtual`（虚拟的）。

对于类型相关的成员函数，它的算法由特定的基类或派生类的行为或实现来决定。例如，对每种数组类型，下标操作符的实现是不同的，所以，我们将它声明为 `virtual`。

等于、不等于操作符和 `size()` 成员函数的实现对于其应用的数组类型来说是独立的，因此，不把它声明成 `virtual`。

对于一个非虚拟函数的调用，编译器在编译时刻选择被调用的函数。而虚拟函数调用的决定则要等到运行时刻。在执行程序内部的每个调用点上，系统根据被调用对象的实际基类或派生类的类型来决定选择哪一个虚拟函数实例。例如，考虑下面的代码：

```
void init( IntArray &ia )
{
    for ( int ix = 0; ix < ia.size(); ++ix )
        ia[ ix ] = ix;
}
```

形式参数 `ia` 可以引用 `IntSortedArray`、`IntArrayRC` 或 `IntArray` 类的对象（我们将简要介绍这里的派生类）。函数 `size()` 作为非虚拟函数，由编译器处理并内联展开。但是，下标操作符要直到执行循环的每次迭代时才能被处理，因为在编译期间编译器不知道数组 `ia` 指向的实际类型。

（第 17 章将详细讨论虚拟函数，包括虚拟析构函数的主题，以及使用虚拟函数设计带来的效率问题。[LIPPMAN96a] 对虚拟函数的实现与效率有更深入的讨论。）

一旦我们定好了设计方案，C++ 的实现就很容易了。例如，下面这个完整的 `IntArrayRC` 派生类定义，被放在一个独立的头文件 `IntArrayRC.h` 中，该文件包含头文件 `IntArray.h`，而 `IntArray.h` 包含有 `IntArray` 类的定义：

```
#ifndef IntArrayRC_H
#define IntArrayRC_H

#include "IntArray.h"

class IntArrayRC : public IntArray {
public:
    IntArrayRC( int sz = DefaultArraySize );
    IntArrayRC( int *array, int array_size );
    IntArrayRC( const IntArrayRC &rhs );

    virtual int& operator[]( int );

private:
    void check_range( int );
};

#endif
```

`IntArrayRC` 只需定义不同于 `IntArray` 实现的那些方面，或者加上对 `IntArray` 扩展的实现。

1. 它必须提供自己的下标操作符实例，以支持范围检查。

2. 它必须提供一个操作来做实际的检查工作（由于它不是公有接口的一部分，所以我们把它声明为 `private`）。

3. 它必须提供一组自动初始化函数，即自己的构造函数集。

`IntArray` 的成员函数与数据成员对于 `IntArrayRC` 来说都是可用的，就如同 `IntArrayRC` 已经显式地定义了它们一样。这正是下面这句话的含义：

```
class IntArrayRC : public IntArray
```

冒号定义了 `IntArrayRC` 是从 `IntArray` 派生而来的。[关键字 `public` 表明派生类共享基类的公有接口。`IntArrayRC` 类型的对象可以用在任何“可以使用基类类型对象”的位置上，比如在 `swap()` 例子中。第 18 章会详细解释这一点。] `IntArrayRC` 可以看作是 `IntArray` 的扩展，它增加了下标范围检查的额外特性。下面是下标操作符的一个实现：

```
inline int&
IntArrayRC::operator[]( int index )
{
    check_range( index );
    return ia[ index ];
}
```

这里，`check_range()` 被实现为一个内联成员函数。它调用 `assert()` 宏 [关于 `assert()` 宏的讨论见 1.3 节。]：

```
#include <cassert>

inline void
IntArrayRC::check_range( int index )
{
    assert( index >= 0 && index < size );
}
```

[我们把 `check_range()` 函数作为一个独立的函数，以便说明私有成员函数并且将范围检查的处理封装起来，方便我们以后改变边界错误的处理方式，或是用异常处理代替 `assert()`。]

派生类对象实际上由几部分构成：每个基类是一个类的子对象（`subobject`），它在新定义的派生类中有独立的一部分。派生类对象的初始化过程是这样的，首先自动调用每个基类的构造函数来初始化相关的基类子对象，然后再执行派生类的构造函数。从设计的角度来看，派生类的构造函数应该只初始化那些在派生类中被定义的数据成员，而不是基类中的数据成员。

虽然我们引入了与类相关的下标操作符版本，以及一个私有的 `check_range()` 辅助函数，但是我们并没有引入需要初始化的额外数据成员。因此，我们可以合理地假设，继承基类的构造函数已经足够了，我们不需要再提供 `IntArrayRC` 的构造函数——因为不需要它们做任何事情。

但是，实际上，我们还是需要提供 `IntArrayRC` 的构造函数，因为基类的构造函数并没有被派生类继承（析构函数和拷贝赋值操作符同样也没有），还因为我们需要某个接口，以便通过这个接口把必要的参数传递给基类 `IntArray` 的构造函数。

例如，假设我们定义了一个 `IntArrayRC` 对象：

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13 };
```

```
IntArrayRC iar( ia, 8 );
```

怎样才能把 `ia` 和 `8` 传递给基类的构造函数呢？（不可否认，如果 `IntArray` 构造函数被继承了，那么就没有这个问题。实际上，那样的话我们会有其他更严重的问题，但现在没有足够的篇幅向你证明这一点。）无论如何，派生类构造函数的语法提供了向基类构造函数传递参数的接口。例如，下面是两个必需的 `IntArrayRC` 构造函数。（第 14 章与第 17 章将对构造函数作更多的讲解，其中包括关于“为什么我们不需要提供 `IntArrayRC` 拷贝构造函数”的解释）：

```
inline IntArrayRC::IntArrayRC( int sz )
    : IntArray( sz ) {}

inline IntArrayRC::IntArrayRC( const int *iar, int sz )
    : IntArray( iar, sz ) {}
```

由冒号分割出来的部分称作成员初始化列表（member initialization list），它提供了一种机制，通过这种机制，我们可以向 `IntArray` 的构造函数传递参数。两个 `IntArrayRC` 构造函数的函数体都是空的，因为它们的工作就是把参数传递给相关的 `IntArray` 构造函数。我们无需提供显式的 `IntArrayRC` 析构函数，因为派生类没有引入任何需要析构的数据成员。继承过来的、需要析构的 `IntArray` 成员都由 `IntArray` 的析构函数来处理。

头文件 `IntArrayRC.h` 中含有 `IntArrayRC` 的定义，以及在类定义之外定义的全部内联成员函数的定义。如果我们定义了非内联函数，则把它们放在 `IntArrayRC.C` ——这个相关联的程序文本文件中。

下面这个小程序实现了 `IntArray` 与 `IntArrayRC` 两个类的层次结构：

```
#include <iostream>
#include <IntArray.h>
#include <IntArrayRC.h>

extern void swap(IntArray&,int,int);

int main()
{
    int array[ 4 ] = { 0, 1, 2, 3 };
    IntArray ia1( array, 4 );
    IntArrayRC ia2( array, 4 );

    // 错误：一位偏移(off-by-one)：应该是size-1
    // IntArray 对象捕捉不到这个错误
    cout << "swap() with IntArray ia1\n";
    swap( ia1, 1, ia1.size() );

    // ok: IntArrayRC对象可以捕捉到这样的错误
    cout << "swap() with IntArrayRC ia2\n";
    swap( ia2, 1, ia2.size() );

    return 0;
}
```

编译并执行这个程序，产生如下结果：

```
swap() with IntArray ia1
swap() with IntArrayRC ia2
```

```
Assertion failed: index >= 0 && index < size
```

C++支持另外两种形式的继承：**多继承**（multiple inheritance，也译多重继承），也就是一个类可以从两个或多个基类派生而来。以及**虚拟继承**（virtual inheritance），在这种继承方式下基类的单个实例在多个派生类之间共享。第18章将讨论这些内容。面向对象程序设计的另一个较为深入的方面是，在程序执行过程中任意一个点上，我们都能够查询基类的引用或指针所指向的实际类型。这是由RTTI（运行时刻类型识别）设施提供的，我们将在19.1节中讨论它。

练习 2.8

一般来说，类型/子类型继承关系反映了一种“is-a（是一种）”的关系：具有范围检查功能的ArrayRC是一种Array。一本书(Book)是一种图书外借资源(LibraryRentalMaterial)。有声书(AudioBook)是一种书(Book)等等。下面哪些反映出这种“is-a”关系？

- (a) 成员函数是一种(isA_kindOf)函数
- (b) 成员函数是一种类
- (c) 构造函数是一种成员函数
- (d) 飞机是一种交通工具
- (e) 摩托车是一种卡车
- (f) 圆形是一种几何图形
- (g) 正方形是一种矩形
- (h) 汽车是一种飞机
- (i) 借阅者是一种图书馆

练习 2.9

判断以下操作哪些可能是类型相关的，因此可把它们定义为虚拟函数？哪些可以在所有类之间共享？对单个基类或派生类来说哪些是惟一的？

- | | |
|----------------|---------------------|
| (a) rotate(); | (b) print(); |
| (c) size(); | (d) dateBorrowed(); |
| (e) rewind(); | (f) borrower(); |
| (g) is_late(); | (h) is_on_loan(); |

练习 2.10

对于保护(protected)访问级别的使用已经有了一些争论。有人认为，使用保护访问级别允许派生类直接访问基类的成员，这破坏了封装的概念，因此，所有的基类实现细节都应该是私有的(private)。另外一些人认为，如果派生类不能直接访问基类的成员，那么派生类的实现将无法有足够的效率供用户使用；如果没有关键字protected，类的设计者将被迫把基类成员设置为public。你怎样认为？

练习 2.11

第二个争论是关于将成员函数显式地声明为virtual的必要性。一些人认为，这意味着如

果类的设计者没有意识到一个函数需要被声明为 `virtual`，则派生类的设计者就没有办法改写这个关键函数。因此，他们建议把所有成员函数都设置为 `virtual` 的。另一方面，虚拟函数比非虚拟函数的效率要低一些。^③因为它们不能被内联（内联发生在编译时刻，而虚拟函数是在运行时刻被处理的），所以它们可能是运行时刻效率低下的原因之一，尤其是小巧而又被频繁调用的、与类型无关的函数，比如 `Array`（数组）的 `size` 函数。你又怎样认为呢？

练习 2.12

下面的每个抽象类型都隐式地包含一族抽象子类型。例如，图书馆藏资料（`LibraryRentalMaterial`）抽象隐式地包含书（`Book`）、音像（`Puppet`）、视盘（`Video`）等。选择其中一个，找出该抽象的子类型层次，并为这个层次指定一个小的公有接口，且其中包括构造函数。如果存在的话，指出哪些函数是虚拟的，并且写一小段伪代码程序来练习使用这个公有接口。

- | | |
|-------------------------------|-----------------------------------|
| (a) <code>Points</code> | (b) <code>Employees</code> |
| (c) <code>Shapes</code> | (d) <code>TelephoneNumbers</code> |
| (e) <code>BankAccounts</code> | (f) <code>CourseOfferings</code> |

2.5 泛型设计

`IntArray` 类为预定义的整型数组类型提供了一个有用的替代类型。如果用户希望使用一个 `double` 或 `string` 类型的数组，那该怎么办呢？实现一个 `double` 类型的数组与 `IntArray` 类的区别只是在其所包含的元素的类型不同，而代码本身无需改变。

C++ 的模板设施提供了一种机制，它能够将类或函数定义内部的类型和值参数化（parameterizing）（我们要到 10.1 节才会讨论值参数）。这些参数在其他方面不变的代码中用作占位符，以后，这些参数会被绑定到实际类型上，可能是内置的类型，也可能是用户定义的类型。例如，在 `Array` 类模板中，我们把数组所包含的元素的类型参数化。以后，当我们实例化（instantiate）一个特定类型的实例时，如 `int`、`double` 或 `string` 类型的 `Array`（数组），就可以在程序中直接使用这三个实例，就好像我们已经显式地为它们编写过代码一样。现在来看一下，怎样把 `IntArray` 类转换成 `Array` 类模板。下面是定义：

```
template < class elemType >
class Array {
public:
    // 把元素类型参数化
    explicit Array( int size = DefaultArraySize );
    Array( elemType *array, int array_size );
    Array( const Array &rhs );

    virtual ~Array() { delete [] ia; }

    bool operator==( const Array& ) const;
    bool operator!=( const Array& ) const;
```

^③ 参见[LIPPMAN96a]，其中讨论了虚拟函数性能的问题。


```

    Array& operator=( const Array& );
    int size() const { return _size; }

    virtual elemType& operator[](int index){ return ia[index]; }
    virtual void sort();

    virtual elemType min() const;
    virtual elemType max() const;
    virtual int      find( const elemType &value ) const;

protected:
    static const int DefaultArraySize = 12;

    int _size;
    elemType *ia;
};

```

关键字 `template` 引入模板，参数由一对尖括号 (`<, >`) 括起来——本例中，有一个参数 `elemType`。关键字 `class` 表明这个参数代表一个类型。标识符 `elemType` 代表实际的参数名，它在 `Array` 类定义中出现了七次，都是作为实际类型的占位符。

在 `Array` 类的每次实例化中，不论是实例化为 `int`、`double` 或 `string` 等等，实例化的实际类型都将代替 `elemType` 参数。下面的例子演示了怎样使用 `Array` 类模板：

```

#include <iostream>
#include "Array.h"

int main()
{
    const int array_size = 4;

    // elemType变成了int
    Array<int> ia(array_size);

    // elemType变成了double
    Array<double> da(array_size);

    // elemType变成了char
    Array<char> ca(array_size);

    int ix;
    for ( ix = 0; ix < array_size; ++ix ) {
        ia[ix] = ix;
        da[ix] = ix * 1.75;
        ca[ix] = ix + 'a';
    }

    for ( ix = 0; ix < array_size; ++ix )
        cout << "[ " << ix << " ] ia: " << ia[ix]
              << "\tca: " << ca[ix]
              << "\tda: " << da[ix] << endl;

    return 0;
}

```

本例中，我们定义了三个独立的 `Array` 类模板的实例：

```
Array<int>    ia(array_size);
Array<double> da(array_size);
Array<char>   ca(array_size);
```

这些实例声明就是在类模板名的后面加上一对尖括号,然后在里面写上数组的实际类型。当我们定义类模板对象,如 ia、da 或 ca 时,会发生什么事情呢?编译器必须为相关的对象分配内存。为了做到这一点,形式模板参数被绑定到指定的实际参数类型上。对 ia 来说,Array 类模板通过将 elemType 绑定到类型 int 上,产生如下的类数据成员:

```
// Array<int> ia(array_size);
int _size;
int *ia;
```

结果是一个类,它与我们前面手工编码实现的 IntArray 类等价。对 da 来说,通过将 elemType 绑定到类型 double 上,成员变为:

```
// Array<double> da(array_size);
int _size;
double *ia;
```

类似地,对 ca 来说,通过将 elemType 绑定到类型 char 上,成员变为:

```
// Array<char> ca(array_size);
int _size;
char *ia;
```

类模板的成员函数会怎么样呢?不是所有的成员函数都能自动地随类模板的实例化而被实例化,只有真正被程序使用到的成员函数才会被实例化,这一般发生在程序生成过程中的一个独立阶段。(16.8 节将详细讨论这个过程。)

编译并运行程序,会产生如下结果:

```
[ 0 ]  ia: 0    ca: a    da: 0
[ 1 ]  ia: 1    ca: b    da: 1.75
[ 2 ]  ia: 2    ca: c    da: 3.5
[ 3 ]  ia: 3    ca: d    da: 5.25
```

模板机制也支持面向对象的程序设计,类模板可以作为基类或派生类。下面是一个带有范围检查的 Array 类模板的定义:

```
#include <cassert>
#include "Array.h"

template <class elemType>
class ArrayRC : public Array<elemType> {
public:
    ArrayRC( int sz = Array<elemType>::DefaultArraySize )
        : Array< elemType >( sz ){};

    ArrayRC( elemType *ia, int sz )
        : Array< elemType >( ia, sz ) {}

    ArrayRC( const ArrayRC &rhs )
        : Array< elemType >( rhs ) {}

    virtual elemType&
```

```

operator[]( int index )
{
    assert( index >= 0 && index < Array<elemType>::size() );
    return ia[ index ];
}

private:
    // ...
};

```

每个 `ArrayRC` 类的实例化过程都会实例化相应的 `Array` 类模板的实例。例如，下面的定义：

```
ArrayRC<int> ia_rc( 10 );
```

引起 `Array` 类和 `ArrayRC` 类的一个 `int` 实例被实例化。`ia_rc` 同上一节的非模板实例相同。为了说明这一点，我们重写前面的程序来练习 `Array` 和 `ArrayRC` 类模板类型。首先，为了支持语句：

```
// 现在swap()必须也是一个模板
swap( ial, 1, ial.size() );
```

我们必须将 `swap()` 定义成一个函数模板。例如：

```

#include "Array.h"
template <class elemType>
void swap( Array<elemType> &array, int i, int j )
{
    elemType tmp = array[ i ];
    array[ i ] = array[ j ];
    array[ j ] = tmp;
}

```

每个 `swap()` 调用都会根据数组的类型产生适当的实例。下面是重新改写之后的 `main()` 函数，它使用了 `Array` 和 `ArrayRC` 类模板：

```

#include <iostream>

#include "Array.h"
#include "ArrayRC.h"

template <class elemType>
inline void
swap( Array<elemType> &array, int i, int j )
{
    elemType tmp = array[ i ];
    array[ i ] = array[ j ];
    array[ j ] = tmp;
}

int main()
{
    Array<int>    ial;
    ArrayRC<int> ia2;

    cout << "swap() with Array<int> ial\n";
    int size = ial.size();

```

```

        swap( ia1, 1, size );

        cout << "swap() with ArrayRC<int> ia2\n";
        size = ia2.size();
        swap( ia2, 1, size );

        return 0;
    }

```

程序的输出结果与非模板的 `IntArray` 类实现相同。

练习 2.13

给出下列类型声明：

```

template <class elemType> class Array;
enum Status { ... };
typedef string *Pstring;

```

如果存在的话，下面哪些对象的定义是错误的？

- (a) `Array< int*& > pri(1024);`
- (b) `Array< Array<int> > aai(1024);`
- (c) `Array< complex< double > > acd(1024);`
- (d) `Array< Status > as(1024);`
- (e) `Array< Pstring > aps(1024);`

练习 2.14

重写下面的类定义，使它成为一个类模板：

```

class example1 {
public:
    example1( double min, double max );
    example1( const double *array, int size );

    double& operator[]( int index );
    bool operator==( const example1& ) const;

    bool insert( const double*, int );
    bool insert( double );

    double min() const { return _min; };
    double max() const { return _max; };

    void min( double );
    void max( double );

    int count( double value ) const;

private:
    int size;
    double *parray;
    double _min;
    double _max;
};

```

练习 2.15

给出如下的类模板：

```
template <class elemType>
class Example2 {
public:
    explicit Example2( elemType val = 0 )
        : _val( val ){}

    bool min( elemType value )      { return _val < value; }
    void value( elemType new_val ) { _val = new_val;      }
    void print( ostream &os )      { os << _val;          }

private:
    elemType _val;
};

template<class elemType>
ostream& operator<<( ostream &os, const Example2<elemType> &ex )
    { ex.print( os ); return os; }
```

如下这样写会发生什么事情？

```
(a) Example2< Array<int>* > ex1;
(b) ex1.min( &ex1 );
(c) Example2< int > sa( 1024 ), sb;
(d) sa = sb;
(e) Example2< string > exs( "Walden" );
(f) cout << "exs: " << exs << endl;
```

练习 2.16

在 Example2 的定义中，我们写：

```
explicit Example2( elemType val = 0 )
    : _val( val ){}

```

其意图是指定一个缺省值，以便用户可以写：

```
Example2< Type > ex1( value );
Example2< Type > ex2;
```

但是，我们的实现把 Type 限制在一个“不能用 0 进行初始化的类型”的子集中（例如，用 0 初始化一个 string 类型，就是一个错误）。^④类似的情况是，如果 Type 不支持输出操作符，那么 print()调用就会失败（因此，Example2 的输出操作符也会失败）。如果 Type 不支持小于操作符，那么 min()调用就会失败。

C++语言本身并没有提供可以指示在实例化模板时 Type 有哪些隐含限制的方法。在最坏的情况下，当程序编译失败时程序员才发现这些限制。你认为 C++语言应该支持限制 Type 的语法吗？如果你认为应该的话，请说明语法，并用它重写 Example2 的定义。如果认为不需要，请说明理由。

④ 通常解决这个问题的做法是：Example2(elemType nval = elemType()) : _val(nval) {}。

练习 2.17

在上一个练习中，我们说如果 `Type` 不支持输出操作符和小于操作符，那么对 `print()` 和 `min()` 的调用就会出错。在标准 C++ 中，错误的产生不是发生在类模板被创建的时候，而是在 `print()` 与 `min()` 被调用的时候。你认为这样的语义正确吗？是否应该在模板定义中标记这个错误？为什么？

2.6 基于异常的设计

异常 (exception) 是指在运行时刻程序出现的反情形，例如数组下标越界、打开文件失败以及可用动态内存耗尽等等。程序员一般有自己的处理异常的风格，这导致了不同的编码习惯，因而很难整合到一个单一的应用程序中。

异常处理 (exception handling) 为“响应运行时刻的程序异常”提供了一个标准的语言级的设施，它支持统一的语法和风格，也允许每个程序员进行微调。异常处理使得我们不需要在程序中处处显式地测试异常状态，从而可以将测试异常状态的代码抽取出来，放入指定的、显式标记的代码块中，因此异常处理设施大大地减少了程序代码的长度和复杂度。

异常处理机制的主要构成如下：

1. 程序中异常出现的点。一旦识别出程序异常，就会导致抛出 (raise 或 throw) 异常。当异常被抛出时，正常的程序就被挂起，直到异常被处理完毕。在 C++ 中，异常的抛出由 `throw` 表达式来执行。例如，在下面的程序段中，一个 `string` 类型的异常被抛出来以便响应打开文件失败异常。

```
if ( ! infile ) {
    string errMsg( "unable to open file: " );
    errMsg += fileName;
    throw errMsg;
}
```

2. 程序中异常被处理的点。典型地，程序异常的抛出与处理位于独立的函数或成员函数调用中。找到处理代码通常要涉及到展开程序调用栈 (program call stack)。一旦异常被处理完毕，就恢复正常的程序执行。但不是在发生异常的地方恢复执行过程，而是在处理异常的地方恢复执行过程。C++ 中，异常的处理由 `catch` 子句来执行。例如，下面的 `catch` 子句处理在第 1 项中被抛出的异常：

```
catch( string exceptionMsg ) {
    log_message( exceptionMsg );
    return false;
}
```

`catch` 子句与 `try` 块相关联。一个 `try` 块用一个或多个 `catch` 子句将一条或多条语句组织起来。例如，下面是函数 `stats()`：

```
int*
stats( const int *ia, int size )
{
    int *pstats = new int[ 4 ];
```

```
try {
    pstats[ 0 ] = sum_it( ia, size );
    pstats[ 1 ] = min_val( ia, size );
    pstats[ 2 ] = max_val( ia, size );
}
catch( string exceptionMsg )
    { /* 处理异常的代码 */ }

catch( const statsException &statsExcp )
    { /* 处理异常的代码 */ }

pstats[ 3 ] = pstats[ 0 ]/size;
do_something( pstats );

return pstats;
}
```

在 stats() 内部有 4 条语句在 try 块之外，在下面两条语句完成之前，可能会有异常被抛出：

```
(1) int *pstats = new int[ 4 ];
(2) do_something( pstats );
```

在语句 (1) 中，new 表达式可能会失败。如果发生了这样的情况，标准库将产生 bad_alloc 标准异常。由于 bad_alloc 是在 try 块之外被抛出的，所以在 stats() 中并没有试图要处理它，于是函数将终止：pstats 没有被初始化，stats() 中后面的语句也不会被执行。异常机制承接了控制权并一直保持直到异常处理完毕。

在语句 (2) 中，在 do_something() 中的语句，以及在 do_something() 中被调用的语句，或 do_something() 函数中被调用的函数所调用的语句等等，都可能会抛出异常。在从 do_something() 调用开始的函数调用链返回之前，这个异常可能（也可能不）会被捕捉到。如果异常被处理了，那么 stats() 继续执行，就像什么也没有发生过一样。如果在 do_something() 结束之前，异常没有被处理，那么 stats() 也会被终止，因为异常发生在 try 块之外。

（注意，如果 size 等于 0，那么：

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

将导致一个除以 0 的除法。尽管这将导致向 pstats[3] 赋一个未定义的数据值，但是对于除以 0 并没有标准异常被抛出。）

try 块内的三条语句会怎么样呢？不同的行为区别如下：如果在 stats() 里面 sum_it()、min_val() 及 max_val() 终止之后，被抛出的异常是活动的（有效的），那么系统不是简单地终止 stats()，而是顺序地检查与 try 块相关联的 catch 子句，试图处理被抛出来的异常。假设 sum_it() 抛出如下异常：

```
throw string( "internal error: adump27832" );
```

则 pstats[0] 不会被初始化，在 try 块中接下来的两条语句也不会被执行，异常机制意识到 sum_it() 是在 try 块中被调用的，因而它将检查相关的两条 catch() 子句。

系统根据被抛出来的异常与 catch 子句中异常类型的匹配情况来选择 catch 子句。在本例中，异常是 string 类型，与下面的 catch 子句相匹配：

```
catch( string exceptionMsg )
{ /* 处理异常的代码 */ }
```

系统把控制传递给被选中的 `catch` 子句体，其中的语句将顺序执行。完成之后，除非在处理该异常的子句中又抛出异常，否则控制将被传回到程序的当前点上。例如，如果我们已经这样写：

```
catch( string exceptionMsg )
{
    cerr << "stats(): exception occurred: "
          << exceptionMsg << endl;
    pstats[0] = pstats[1] = pstats[2] = 0;
}
```

那么，在 `catch` 子句完成时，控制将被传递给 `catch` 子句集后面的可执行语句。本例中，语句：

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

被执行，然后是 `do_something()` 调用，以及返回 `pstats`。而调用 `stats()` 的函数根本不知道曾经有异常被抛出。

一段更为合理的异常处理代码可能如下所示：

```
catch( string exceptionMsg )
{
    cerr << "stats(): exception occurred: "
          << exceptionMsg
          << " unable to stat array "
          << endl;
    delete [] pstats;
    return 0;
}
```

在上面的代码中，`catch` 子句直接把控制返回给外面的调用函数。我们希望外面的函数在把返回值用作索引数组之前，先测试它是否为 0。

如果 `try` 块内抛出的异常不能被相关联的 `catch` 子句处理，那么函数将被终止。然后，异常机制再在调用 `stats()` 的函数中查找处理代码。

如果异常机制按照函数被调用的顺序回查每个函数直到 `main()` 函数，仍然没有找到处理代码，那么它将调用标准库函数 `terminate()`。缺省情况下，`terminate()` 函数结束程序。

一种特殊的、能够处理全部异常的 `catch` 子句如下：

```
catch( ... )
{
    // 处理所有异常，虽然它无法
    // 访问异常对象
}
```

我们可以把它看作是一种捕捉所有异常（`catch-all`）的 `catch` 子句。

异常处理机制为统一地处理程序异常提供了语言一级的设施。第 11 章与 19 章将进一步详细讨论。另一本配套的书《Inside the C++ Object Model》([LIPPMAN96a])中讨论了实现与性能的话题。Josee Lajoie 在 [LIPPMAN96b] 中的文章 “Exception Handling: Behind the Scenes” 对此也有讨论。[LIPPMAN96b] 中 Tom Cargill 的文章 “Exception Handling: A False Sense of

Security” 则对使用异常处理过程中易犯的错误做了很好的讨论。

练习 2.18

下面的函数对可能的非法数据以及可能的操作失败完全没有提供检查。找出程序中所有可能出错的地方。（本练习中，我们不关心可能会抛出的异常。）

```
int *alloc_and_init( string file_name )
{
    ifstream infile( file_name );
    int elem_cnt;
    infile >> elem_cnt;
    int *pi = allocate_array( elem_cnt );

    int elem;
    int index = 0;
    while ( cin >> elem )
        pi[ index++ ] = elem;

    sort_array( pi, elem_cnt );
    register_data( pi );

    return pi;
}
```

练习 2.19

alloc_and_init()函数会调用到下面的函数，如果这些函数调用失败了，它们将抛出相应类型的异常：

```
allocate_array() noMem
sort_array()    int
register_data()  string
```

请在合适的地方插入一个或多个 try 块以及相应的 catch 子句来处理这些异常。在 catch 子句中只需简单地输出错误的出现情况。

练习 2.20

检查在练习 2.18 中的函数 alloc_and_init()中所有可能出现的错误，指出哪些错误会抛出异常。修改该函数（或用练习 2.18 的版本，或用练习 2.19 的版本）来抛出可被识别的异常（抛出字符串就可以了）。

2.7 用其他名字来命名数组

把代码分发给其他部门的诸多困难中，有一个是我们不知道全局名字会有什么样的影响。例如，在 Intel 公司，有人写了：

```
class Array { ... };
```

那么他就不能在相同的程序中既使用上面的 Array 类，又使用我们实现的那个 Array 类。

名字的可视性使这两份实现代码相互排斥。

在 C++ 标准化之前, 解决这个问题的传统做法是在全局可见的名字前加上一个惟一的字符串前缀。例如, 我们可以这样发行数组 Array 类:

```
class Cplusplus_Primer_Third_Edition_Array { ... };
```

虽然这个名字可能是惟一的(我们不能保证这一点), 但是写起来并不方便。标准 C++ 的名字空间机制是 C++ 语言针对这个问题提供的语言一级的解决方案。

名字空间机制允许我们封装名字, 否则这些名字就有可能会污染(影响)全局名字空间(pollute the global namespace)。一般来说, 只有当我们希望自己的代码被外部软件开发部门使用时, 才使用名字空间。例如, 我们可以这样封装 Array 类:

```
namespace Cplusplus_Primer_3E {
    template <class elemType>
        class Array { ... };
    // ...
}
```

关键字 namespace 后面的名字标识了一个名字空间, 它独立于全局名字空间, 我们可以在里面放一些希望声明在函数或类之外的实体。名字空间并不改变其中的声明的意义, 只是改变了它们的可视性。在继续讨论之前, 先扩展我们的可用名字空间集:

```
namespace IBM_Canada_Laboratory {
    template <class elemType>
        class Array { ... };
        class Matrix { ... };
    // ...
}

namespace Disney_Feature_Animation {
    class Point { ... };
    template <class elemType, int size>
        class Array { ... };
    // ...
}
```

如果名字空间内的声明对程序而言不是立即可见的, 那么我们怎样访问它们呢? 我们可以使用限定修饰名字符(qualified name notation), 格式如下:

```
namespace_identifier::entity_name;
```

如在:

```
Cplusplus_Primer_3E::Array<string> text;
IBM_Canada_Laboratory::Matrix mat;
Disney_Feature_Animation::Point origin( 5000, 5000 );
```

虽然 Disney_Feature_Animation、IBM_Canada_Laboratory 以及 Cplusplus_Primer_3E 都能够惟一地标识相应的名字空间, 但是, 如果在程序中经常这样使用, 则多少会有些麻烦。使用名字空间标识符如 P3E、DFA 或 IBM_CL 会更方便一些, 但是它们表达的信息相对比较少, 同时也增加了名字冲突的可能性。为了提供有意义的名字空间标识符, 同时程序员又能很方

便地访问在名字空间内定义的实体，C++提供了别名设施。

名字空间别名 (`namespace alias`) 允许用一个可替代的、短的或更一般的名字与一个现有的名字空间关联起来。例如：

```
// 提供一个更一般化的别名
namespace LIB = IBM_Canada_Laboratory;

// 提供一个更短的别名
namespace DFA = Disney_Feature_Animation;
```

然后这个别名就可以用作原始名字空间的同义词。例如：

```
#include "IBM_Canada.h"
namespace LIB = IBM_Canada_Laboratory;

int main()
{
    LIB::Array<int> ia(1024);
    // ...
}
```

别名也可以用来封装正在使用的实际名字空间。例如，在此情形下，我们可以通过改变分配给别名的名字空间，来改变所使用的声明集，而无需改变“通过别名访问这些声明”的实际代码。例如：

```
namespace LIB = Cplusplus_Primer_3E;

int main()
{
    // 在这种情况下，下面的声明无须改变
    LIB::Array<int> ia(1024);
    // ...
}
```

但是，如果要让这项技术在实际工作中发挥作用，那么两个名字空间中的声明必须提供同样的接口。例如，下面的代码就不能工作，因为 `Disney` 的 `Array` 类需要一个类型参数和一个数组长度参数：

```
namespace LIB = Disney_Feature_Animation;

int main()
{
    // 不再是一个有效的声明
    LIB::Array<int> ia(1024);
    // ...
}
```

程序员常常希望在访问名字空间内声明的名字时不加限定修饰符。即使我们已经为名字空间标识符提供了较短的别名，在每次访问该名字空间内声明的名字时也都要进行限定，还是太麻烦！**using 指示符** (`using directive`) 使名字空间内的所有声明都可见，这样这些声明能够不加限定地使用。例如：

```
#include "IBM_Canada_Laboratory.h"
```

```
// 使所有的名字都可见
using namespace IBM_Canada_Laboratory;

int main()
{
    // ok: IBM_Canada_Laboratory::Matrix
    Matrix mat( 4,4 );

    // ok: IBM_Canada_Laboratory::Array
    Array<int> ia( 1024 );
    // ...
}
```

`using` 与 `namespace` 都是关键字。被引用的名字空间必须已经被声明了，否则会引起编译错误。

`using 声明` (`using declaration`) 提供了选择更为精细的名字可视性机制。它允许使名字空间中的单个声明可见。例如：

```
#include "IBM_Canada_Laboratory.h"

// 只让Matrix可见
using IBM_Canada_Laboratory::Matrix;

int main()
{
    // ok: IBM_Canada_Laboratory::Matrix
    Matrix mat(4,4);

    // 错误: IBM_Canada_Laboratory::Array不可见
    Array<int> ia( 1024 );
    // ...
}
```

为了防止标准 C++ 库的组件污染用户程序的全局名字空间，所有标准 C++ 库的组件都声明在一个被称为 `std` 的名字空间内。正如在第 1 章中提到的，即使我们在程序文本文件中包含了 C++ 库头文件，头文件中声明的组件在我们的文本文件中也不是自动可见的。例如，在标准 C++ 中，下面的代码实例就不能正常编译：

```
#include <string>
// 错误: string不是可见的
string current_chapter = "A Tour of C++";
```

在 `<string>` 头文件中的所有声明都包含在名字空间 `std` 中。正如第 1 章所提到的，我们可以用“在 `#include` 预处理器指示符后面加上 `using` 指示符”的办法，使 C++ 头文件 `<string>` 中的、在名字空间 `std` 中声明的组件对于我们的程序都是可见的：

```
#include <string>
using namespace std;

// ok: string是可见的
string current_chapter = "A Tour of C++";
```

为了使在名字空间 `std` 中声明的名字在我们的程序中可见，指示符 `using` 通常被看作是一种比较差的选择方案。在上面的例子中，指示符 `using` 使头文件 `<string>` 中声明的、并且位于

名字空间 `std` 中的所有组件在程序文本文件中都是可见的，这又将全局名字空间污染问题带回来了。而这个问题正是 `std` 名字空间首先要努力避免的，它增加了“C++标准库组件的名字”与“我们程序中声明的全局名字”冲突的机会。

现在，我们对名字空间机制已经有了一些了解，知道有另外两种机制可代替指示符 `using`，使我们能够引用到隐藏在名字空间 `std` 中的名字 `string`。我们可以使用限定的名字，例如：

```
#include <string>
// ok: 使用限定的名字
std::string current_chapter = "A Tour of C++";
```

或如下使用 `using` 声明：

```
#include <string>
using std::string;

// ok: 下面的using声明使string可见
string current_chapter = "A Tour of C++";
```

为了使用名字空间中声明的名字，建议使用带有精细选择功能的 `using` 声明代替 `using` 指示符。这也正是本书的代码示例中没有出现 `using` 指示符的另一个原因。理想情况下，每一个代码示例对它所用到的每个库组件都应该有一个 `using` 声明。为了限制例子代码的长度，也因为本书的许多例子是在不支持名字空间的情况下被编译的，所以 `using` 声明就没有显示出来。8.6 节将进一步讨论怎样对标准 C++ 库的组件使用 `using` 声明。

在接下来的四章中，我们将讲述另外四个类的设计。第 3 章讲的是 `String`（字符串）类的设计与实现，第 4 章介绍整数 `Stack`（栈）类的设计，第 5 章是 `List`（列表）类，第 6 章对第 4 章定义的 `Stack`（栈）类进行重新设计。名字空间机制允许我们把每个类放在单独的头文件中，但是仍然能把它们的名字封装到单个 `Cplusplus_Primer_3E` 名字空间中。在第 8 章中，我们将讨论这项技术，并对名字空间作更多的介绍。

练习 2.21

给出如下名字空间定义：

```
namespace Exercise {
    template <class elemType>
        class Array { ... };

    template <class Etype>
        void print( Array< Etype > );

    class String { ... };
    template <class listType>
        class List { ... };
}
```

以及下面的程序：

```
int main() {
    const int size = 1024;
    Array< String > as( size );
```

```

List< int > il( size );

// ...

Array< String > *pas = new Array<String>(as);
List <int> *pil = new List<int>(il);

print( *pas );
}

```

因为类型名被封装在名字空间中，所以当前程序编译失败。把程序修改为：

1. 用限定名字修饰符来访问名字空间 `Exercise` 中的类型定义。
2. 使用 `using` 声明来访问类型定义。
3. 用名字空间别名机制。
4. 用 `using` 指示符。

2.8 标准数组——向量

正如我们已经看到的，尽管 C++ 内置的数组支持容器的机制，但是它不支持容器抽象的语义。为了在这样的层次上编写程序，在标准 C++ 之前，我们要么从某个途径获得这样的类，要么自己实现这样的类。在标准 C++ 中，数组类是 C++ 标准库的一部分，现在它不叫数组，而叫向量（`vector`）了。

当然，向量是一个类模板，所以我们这样写：

```

vector<int>      ivec( 10 );
vector<string>   svec( 10 );

```

上面的代码分别定义了一个包含 10 个整型对象的向量和一个包含 10 个字符串对象的向量。

在我们实现的 `Array` 类模板与 `vector` 类模板的实现之间有两个主要区别。第一个区别是 `vector` 类模板支持“向现有的数组元素赋值”的概念以及“插入附加元素”的概念——即 `vector` 数组可以在运行时刻动态增长（如果程序员希望使用这个特性的话）。第二个区别是更加广泛，代表了设计方法的重要转变。`vector` 类不是提供一个巨大的“可以适用于向量”的成员操作集，如 `sort()`、`min()`、`max()` 及 `find()` 等等，而是只提供了一个最小集：如等于、小于操作符、`size()`、`empty()` 等操作。而一些通用的操作如 `sort()`、`min()`、`max()` 和 `find()` 等等，则是作为独立的泛型算法（`generic algorithm`）被提供的。

要定义一个向量，我们必须包含相关的头文件：

```
#include < vector >
```

下面都是 `vector` 对象的合法定义：

```

#include < vector >

// 创建vector对象的各种方法
vector<int> vec0;      // 空的vector

```

```

const int size = 8;
const int value = 1024;

// size为8的vector
// 每个元素都被初始化为0
vector<int> vec1( size );

// size为8的vector
// 每个元素都被初始化为1024
vector<int> vec2( size, value );

// vec3的size为4
// 被初始化为ia的4个值
int ia[4] = { 0, 1, 1, 2 };
vector<int> vec3( ia, ia+4 );

// vec4是vec2的拷贝
vector<int> vec4( vec2 );

```

既然定义了向量，我们就需要遍历里边的元素。与 Array 类模板一样，标准 vector 类模板也支持使用下标操作符，例如：

```

#include <vector>
extern int getSize();

void mumble()
{
    int size = getSize();
    vector< int > vec( size );

    for ( int ix = 0; ix < size; ++ix )
        vec[ ix ] = ix;

    // ...
}

```

另外一种遍历方法是使用**迭代器对**（iterator pair）来标记向量的起始处和结束处。迭代器是一个支持指针类型抽象的类对象。vector 类模板提供了一对操作 begin()和 end()，它们分别返回指向“向量开始处”和“结束处后 1 个”的迭代器。这一对迭代器合起来可以标记出待遍历元素的范围。例如，下面的代码是前面代码段的一个等价实现：

```

#include < vector >
extern int getSize();

void mumble()
{
    int size = getSize();
    vector< int > vec( size );

    vector< int >::iterator iter = vec.begin();

    for (int ix = 0; iter != vec.end(); ++iter, ++ix )
        *iter = ix;
}

```

```
// ...
}
```

iter 的定义:

```
vector< int >::iterator iter = vec.begin();
```

将其初始值指向 vec 的第一个元素。iterator 是 vector 类模板中用 typedef 定义的类型，而这里的 vector 类实例包含 int 类型的元素。下面的代码使迭代器指向 vector 的下一个元素:

```
++iter
```

下面的代码解除迭代器的引用，以便访问实际的元素:

```
*iter
```

能够应用到向量上的操作惊人地多，但是它们并不是作为 vector 类模板的成员函数提供的。它们是以一个独立的泛型算法集的形式，由标准库提供。下面是一组可供使用的泛型算法的示例:

- **搜索 (search) 算法:** find()、find_if()、search()、binary_search()、count()和 count_if()。
- **分类排序 (sorting) 与通用排序 (ordering) 算法:** sort()、partial_sort()、merge()、partition()、rotate()、reverse()和 random_shuffle()。
- **删除 (deletion) 算法:** unique()和 remove()。
- **算术 (numeric) 算法:** accumulate()、partial_sum()、inner_product()和 adjacent_difference()。
- **生成 (generation) 和变异 (mutation) 算法:** generate()、fill()、transformation()、copy()和 for_each()。
- **关系 (Relational) 算法:** equal()、min()和 max()。

泛型算法接受一对迭代器，它们标记了要遍历元素的范围。例如，ivec 是一个包含某种类型元素的、某个长度的向量，要用 sort()对它的全部元素进行排序，我们只需简单地这样写:

```
sort( ivec.begin(), ivec.end() );
```

只想对 ivec 向量的前面一半进行排序，可以这样写:

```
sort( ivec.begin(), ivec.begin()+ivec.size()/2 );
```

泛型算法还能接受指向内置数组的指针对，例如，已知数组:

```
int ia[7] = { 10, 7, 9, 5, 3, 7, 1 };
```

我们可以如下对整个数组排序:

```
sort( ia, ia+7 );
```

我们还可以只对前四个元素排序:

```
sort( ia, ia+4 );
```

要使用这些算法，我们必须包含与它们相关的头文件:

```
#include <algorithm>
```


下面的代码显示了怎样把各种各样的泛型算法应用到 `vector` 类对象上:

```
class object:
#include <vector>
#include <algorithm>
#include <iostream>

int ia[ 10 ] = {
    51, 23, 7, 88, 41, 98, 12, 103, 37, 6 };

int main()
{
    vector< int > vec( ia, ia+10 );

    // 排序数组
    sort( vec.begin(), vec.end() );

    // 获取值
    int search_value;
    cin >> search_value;

    // 搜索元素
    vector<int>::iterator found;
    found = find( vec.begin(), vec.end(), search_value );
    if ( found != vec.end() )
        cout << "search_value found!\n";
    else cout << "search_value not found!\n";

    // 反转数组
    reverse( vec.begin(), vec.end() );

    // ...
}
```

标准库还提供了对 `map` 关联数组的支持, 即数组元素可以被整数之外的其他东西索引。例如, 我们可以这样来支持一个电话目录: 这个电话目录是电话号码的数组, 但是它的元素可以由该号码所属人的姓名来索引:

```
#include <map>
#include <string>
#include "TelephoneNumber.h"

map< string, telephoneNum > telephone_directory;
```

在第 6 章我们将看到 `vector`、`map` 以及标准 C++ 库支持的其他容器类型, 本书将通过一个文本查询系统的实现来说明这些类型的用法。而第 12 章会讲解泛型算法。附录按字母顺序提供了每个算法的解释及其用法。

本章大致地讲述了 C++ 为数据抽象 (基于对象的程序设计)、面向对象的程序设计、泛型程序设计 (模板、容器类型以及泛型算法)、大型程序设计 (异常处理与名字空间) 提供的支持。而本书余下的部分将更详细地介绍这些内容, 逐步讲解 C++ 中基本、但又非常先进的特性。

练习 2.22

解释每个 vector 定义的结果:

```
string pals[] = {
    "pooh", "tigger", "piglet", "eeyore", "kanga" };

(a) vector<string> svec1( pals, pals+5 );
(b) vector<int>   ivec1( 10 );
(c) vector<int>   ivec2( 10, 10 );
(d) vector<string> svec2( svec1 );
(e) vector<double> dvec;
```

练习 2.23

已知下列函数声明, 请实现 min() 的函数体, 它查找并返回 vec 的最小元素。要求首先使用“索引 vec 中元素的 for 循环”来实现 min(), 然后, 再使用“通过迭代器遍历 vec 的 for 循环”来实现 min():

```
template <class elemType>
elemType
min( const vector<elemType> &vec );
```