一、C++ 基础

1. 开始

学习一门新的程序设计语言的 最好方法就是练习编写程序。本章编写一个程序来解决简单的书店问题。

熟点保存所有销售记录的档案，每条记录保存了某本书的一次销售信息（一册或多册）。每条记录包含三个数据项：

0-201-70353-X　4　24.99

三项分别是：书的 ISBN 号（国际标准书号，一本书的唯一标识）、售出的册数、书的单价。书店老板需要查询此档案，计算每本书的销量、销售额及平均售价。

编写这个程序，需要使用若干 C++ 的基本特性。而且，需要了解如何编译及运行程序。

虽然还没有编写这个程序，但显然它必须

- 定义变量
- 进行输入和输出
- 使用数据结构保存数据
- 检测两条记录是否有相同的 ISBN
- 包含一个循环来处理销售档案中的每条记录

首先介绍如何用 C++ 来解决这些子问题，然后编写书店程序。

1.1 编写一个简单的 C++ 程序

编写好程序后，我们就需要编译它。如何编译程序依赖于你使用的操作系统和编译器。

集成开发环境（Integrated Developed Environment，IDE），将编译器与其他程序创建和分析工具包装在一起。需要学习如何高效地使用它们。

程序源文件命名

后缀告诉系统这个文件是一个C++程序。不同编译器使用不同的后缀命名约定，常见的包括.cc、.cxx、.cpp、.cp 及 .c

从命令行运行编译器

假定 main 程序保存在文件 prog1.cc 中，用如下命令编译它

```
$ CC prog1.cc
```

CC 是编译器程序的名字，$ 是系统提示符。编译器生成一个可执行文件，Windows 系统将这个可执行文件命名为 prog1.exe。UNIX 系统的编译器通常将可执行文件命名为 a.out。

Windows 系统运行一个可执行文件，需要提供可执行文件的文件名，可以忽略扩展名.exe：

```
$ prog1
```

一些系统中，即使文件就在当前目录或文件夹中，也必须显式指出文件的位置。

```
$ .\prog1
```

. 后跟一个反斜杠指出该文件在当前目录中

UNIX 运行一个可执行文件，需要使用全文件名，包括扩展名：

```
$ a.out

//指定文件位置，用一个 . 后跟一个斜杠指出可执行文件位于当前目录
$ ./a.out
```

访问 main 的返回值的方法依赖于系统。在 UNIX 和Windows 中，执行完一个程序后，可以通过 echo 命令获得其返回值。

```
// UNIX，如下命令获得状态
$ echo $?

// Windows
$ echo %ERRORLEVEL%
```

> 常用的编译器是 GNU 编译器和 Visual Studio 编译器。默认情况下，运行 GNU 编译器的命令是 g++:
>
> ```
> $ g++ -o prog1 prog1.cc
> ```

> -o prog1 是编译器参数，指定了可执行文件的文件名。

1.2 初始输入输出

C++ 没有定义任何输入输出（IO）语句，而是包含了一个全面的标准库（standard library）来提供 IO 机制（以及很多其他设施）。

iostream 库包含两个基础类型 istream 和 ostream，分别表示输入流和输出流。一个流就是一个字符序列，是从 IO 设备读出或写入 IO 设备的。“流” （stream）表达的是，随着时间推移，字符是顺序生成或消耗的。

标准输入输出对象

标准库定义了 4个IO对象。处理输入，使用一个名为 cin 的 istream 类型的对象。这个对象也被称为标准输入（standard input）。输出，使用一个名为 cout 的 ostream 类型的对象。此对象也被称为标准输出。标准库还定义了其他两个 ostream 对象，cerr 用来输出警告和错误消息，称为标准错误。clog 用来输出程序运行时的一般性信息。

系统通常将程序运行的窗口和这些对象关联起来。当我们读取 cin，数据将从程序正在运行的窗口读入，向 cout、cerr 和 clog 写入数据时，将会写到同一个窗口。

endl 是一个被称为操纵符（manipulator）的特殊值。写入 endl 的效果是结束当前行，并将与设备关联的缓冲区（buffer）中的内容刷到设备中。缓冲刷新操作可以保证到目前为止程序所产生的所有输出都真正写入输出流中，而不是仅停留在内存中等待写入流。

使用标准库中的名字

前缀 std:: 指出名字 cout 和 endl 是定义在名为 std 的命名空间（namespace）中的。命名空间可以帮助我们避免不经意的名字定义冲突，以及使用库中相同名字导致的冲突。标准库定义的所有名字都在命名空间 std 中。

从流读取数据

输入运算符（>>）与输出运算符类似，接受一个 istream 作为其左侧运算对象，接受一个对象作为其右侧运算对象。它从给定的 istream 读入数据，并存入给定对象中。

1.3 注释

1.4 控制流

1.5 类

为了使用标准库，我们必须包含相关的头文件。类似，我们也需要使用头文件来访问自己定义的类。习惯上，头文件根据定义的类的名字来命名。通常使用 .h 作为头文件的后缀，但也有一些程序员习惯 .H、.hpp 或 .hxx。标准库头文件通常不带后缀。编译器一般不关心头文件名的形式，但有的 IDE 对此有特定要求。

1.5.1 Sales_item 类

Sales_item 类的作用是表示一本书的总销售额、售出册数和平均售价。我们现在不关心这些数据如何存储、如何计算。为了使用一个类，我们不必关心它是如何实现的，只需要知道类对象可以执行什么操作。

每个类实际上都定义了一个新的类型，其类型名就是类名。Sales_item 类定义了一个名为 Sales_item 的类型，与内置类型一样，可以定义类类型的变量：

```
1 Sales_item item;
```

表达 item 是一个 Sales_item 对象

除了可以定义 Sales_item 类型的变量之外，我们还可以：

- 调用一个名为 isbn 的函数从一个 Sales_item 对象中提取 ISBN 书号
- 用输入运算符（>>）和输出运算符（<<）读、写 Sales_item 类型的对象
- 用赋值运算符（=）将一个 Sales_item 对象的值赋予另一个 Sales_item 对象
- 用加法运算符（+）将两个 Sales_item 对象相加。两个对象必须表示同一本书（相同的 ISBN）。加法结果是一个新的 Sales_item 对象，其 ISBN 与两个运算对象相同，而其总销售额和售出册数则是两个运算对象的对应值之和。
- 使用复合赋值运算符（+=）将一个 Sales_item 对象加到另一个对象上

> 关键概念：类定义了行为
>
> 当你读这些程序时，一件要牢记的事情是：类 Sales_item 的作者定义了类对象可以执行的所有动作。即 Sales_item 类定义了创建一个 Sales_item 对象时会发生什么事情，以及对 Sales_item 对象进行赋值、加法或输入输出运算时会发生什么事情。
>
> 一般而言，类的作者决定了类类型对象上可以使用的所有操作。

读写 Sales_item

知道了可以对 Sales_item 对象执行哪些操作后，就可以编写使用类的程序了。例如，下面程序从标准输入读入数据，存入一个 Sales_item 对象中，然后将 Sales_item 的内容写回标准输出：

```
1 #include <iostream>
```

```
#include "Sales_item.h"

int main (){
    Sales_item book;
    // 读入 ISBN号、售出的册数以及销售价格
    std::cin >> book;
    // 写入 ISBN、售出的册数、总销售额和平均价格
    std::cout << book << std::endl;
    return 0;
}
```

包含来自标准库的头文件时，应该用尖括号（<>）包围头文件。对于不属于标准库的头文件，则用双引号（""）包围。

Sales_item 对象的加法

> 使用文件重定向
>
> 测试程序时，反复从键盘敲入这些销售记录作为程序输入，是非常乏味的。大多数操作系统支持文件重定向，这种机制允许我们将标准输入和标准输出与命名文件关联起来：
>
> ```
> $ addItems <infile >outfile
> ```
>
> $ 是操作系统提示符，我们的加法程序已经边翼卫名为 addItems.exe 的可执行文件（UNIX 中是 addItems），上述命令会从一个名为 infile 的文件的读取销售记录，并将输出结果写入到一个名为 outfile 的文件中，两个文件都位于当前目录。

1.5.2 成员函数

1.6 书店程序

```
#include<iostream>
#include "Sales_item.h"

int main () {
    Salse_item total;    //保存下一条交易记录的变量
    //读入第一条交易记录，确保有数据可以处理
    if (std::cin >> total) {
        Sales_item trans;    //保存和的变量
        while (std::cin >> trans) {
```

```
            //如果我们仍在处理相同的书
            if (total.isbn() == trans.isbn())
                total += trans;    // 更新总销售额
            else {
                // 打印前一本书的结果
                std::cout << total << std::endl;
                total = trans;    // total现在表示下一本书的销售额
            }
        }
    } else {
        //没有输入，error
        std::cerr << "No data?!" << std::endl;
        return -1;    //表示失败
    }
    return 0;
}
```

小结

介绍了 C++ 知识，能够编译、运行简单的 C++ 程序。如何定义一个 main 函数，他是操作系统执行你的程序的调用入口。还看到如何定义变量，如何进行输入输出，以及如何编写 if、for 和 while 语句。最后介绍了 C++ 最基本的特性——类。对于其他人定义的一个类，我们应该如何创建、使用其对象。我们如何定义自己的类。

第 I 部分 C++ 基础

任何常用的编程语言都具备一组公共的语法特征，不同语言仅在特征的细节上有所区别。想要学习并掌握一种编程语言，理解其语法特征的实现细节是第一步。最基本的特征包括：

- 整形、字符型等内置类型
- 变量，用来为对象命名
- 表达式和语句，用于操纵上述数据类型的具体值
- if 或 while 等控制结构，允许我们有选择地执行一些语句或者重复地执行一些语句
- 函数，用于定义可供随时调用的计算单元

多数编程语言通过两种方式进一步补充其基本特征：一是赋予程序员自定义数据类型的权利，从而实现对语言的扩展；二是将一些有用的功能封装成库函数提供给程序员。

与大多数编程语言一样，C++ 的对象类型决定了能对该对象进行的操作，一条表达式是否合法依赖于其中参与运算的对象的类型。如 Smalltalk 和 Python 等，在程序运行时检查数据类型；与之相反，

C++ 是一种静态数据类型语言，它的类型检查发生在编译时。因此，编译器必须知道程序中每一个变量对应的数据类型。

C++ 提供了一组内置数据类型、相应的运算符以及几种程序流控制语句，这些元素共同构成了 C++ 语句的基本形态。

C++ 最重要的语法特征就是类，通过它，程序员可以定义自己的数据类型。为了与C++的内置类型区别开来，它们通常被称为“类类型（class type）”。C++ 主要的一个设计目标就是让程序员自定义的数据类型像内置类型一样好用。基于此，标准 C++ 库实现了丰富的类和函数。

第 I 部分的主题是学习 C++ 的基础知识，这是掌握 C++ 的第一步。第 2 章详述内置类型，初步介绍了自定义数据类型的方法。第 3 章介绍两种最基本的数据类型：字符串和向量。C++ 和许多编程语言所共有的一种底层数据结构——数组也在本章提及。第4~6章依次介绍了表达式、语句和函数。第7章描述了如何构建我们自己的类。

2. 变量和基础类型

数据类型是程序的基础：它告诉我们数据的意义以及我们能在数据上执行的操作。

C++ 支持广泛的数据类型。它定义了几种基本内置的类型（如字符、整型、浮点数等），同时提供了自定义数据类型的机制。基于此，C++标准库定义了一些更加复杂的数据类型，比如可变长的字符串和向量等。本章主要讲述内置类型，并初步了解C++是如何支持更复杂数据类型的。

2.1 基本内置类型

C++ 定义了一套包括算数类型（arithmetic type）和空类型（void）在内的基本数据类型。其中算数类型包括了字符、整型数、布尔值和浮点数。空类型不对应具体的值，仅用于一些特殊场合。

2.1.1 算数类型

算数类型分成两类：整形（intergral type，包括字符和布尔类型）和浮点型

算数类型的大小（该类型数据所占的比特数）在不同机器有所差别。下表列出了 C++ 标准规定的尺寸的最小值，允许编译器赋予这些类型更大的尺寸。某一类型所占的比特数不同，所能表示的数据范围也不一样。

类型	含义	最小尺寸
bool	布尔类型	未定义
char	字符	8 位

wchar_t	宽字符	16 位
char16_t	Unicode 字符	16 位
char32_t	Unicode 字符	32 位
short	短整型	16 位
int	整型	16 位
long	长整型	32 位
long long	长整型	64 位
float	单精度浮点数	6 位有效数字
double	双精度浮点数	10 位有效数字
long double	扩展精度浮点数	10 位有效数字

一个 char 的大小和一个机器字节一样。

Unicode 是用于表示所有自然语言中字符的标准。

> **内置的机器实现**
>
> 计算机以 bit 序列存储数据，每个 bit 非 0 即 1，例如：
>
> 000110110111000101100100001110 11...
>
> 大多数计算机以 2 的整数次幂个 bit 作为块来处理内存，可寻址的最小内存块成为 字节（byte），存储的基本单元称为 字（word），它通常由几个字节组成。C++ 中，一个字节要至少能容纳机器基本字符集中的字符。大多数机器的字节由 8 bit 构成，字则由 32 或 64 bit构成，也就是 4 或 8 字节。
>
> 大多数计算机将内存中的每个字节与一个数字（地址 address）关联起来，在一个字节为 8 bit、字为 32 bit 的机器上，可能看到一个字的内存区域如下：

736424	00111011
736425	00011011
736426	01110001
736427	01100100

左侧是字节的地址，右侧是字节中 8 bit的具体内容。

我们能够使用某个地址表示从这个地址开始的大小不同的bit串，例如，我们可能会说地址 736424 的那个字或者地址 736427 的那个字节。为了赋予内存中某个地址明确的含义，必须首先知道存储在该地址的数据的类型。类型决定了数据所占的 bit 数以及该如何解释这些bit 的内容。

若 736424 处的对象类型是 float，并且该机器中 float 以 32bit存储，那么就能知道这个对象的内容占满了整个字。

带符号类型和无符号类型

除去布尔型和扩展的字符型之外，其他整型可以划分为 带符号的（signed）和 无符号的（unsigned）两种。带符号类型可以表示正数、负数或 0，无符号类型仅能表示大于等于 0 的值。

int、short、long 和 long long 都是带符号的。

与其他整型不同，字符型被分为了三种：char、signed char 和 unsigned char。注意：char 和 signed char 不一样。类型 char 实际上会表现为带符号和无符号形式中的一种，具体由编译器决定。

无符号类型中所有bit用来存储值。8 bit 的 unsigned char 可以表示 0 到 255 区间内的值。

signed char 一般表示定为 -128 到 127.

如何选择类型

- 明确知道数值不可能为负时，选用无符号类型
- 使用 int 整数运算。short 太小而long一般和int尺寸一样。如果数值超过了 int 的表示范围，用 long long
- 算术表达式不要使用 char 或 bool，只有存放字符或布尔值时才用它们。char 不同机器上可能是有符号或者无符号的，运算容易出问题。如果需要一个不大的整数，明确指定它的类型是 signed char 或者 unsigned char

> - 浮点数运算使用 double，因为 float 通常精度不够而且双精度浮点数和单精度浮点数的计算代价相差无几。long double 提供的精度一般是没有必要的，它带来的运行时消耗不容忽视

2.1.2 类型转换

对象的类型定义了对象能包含的数据和能参与的运算，其中一种运算被大多数类型支持，就是将对象从一种给定的类型转换（convert）为另一种相关类型。

当在程序的某处我们使用了一种类型而其实对象应该取另一种类型时，程序会自动进行类型转换。

当给某种类型的对象强行赋了另一种类型的值时（下面把一种算数类型的值赋给另外一种类型时）：

```
bool b = 42;    // b 为true
int i = b;    // i 的值为 3
double pi = i;    // pi 的值为 3.0
unsigned char c = -1;    // 假设 char 占 8 bit，c 的值为 255
signed char c2 = 256;    // 假设 char 占 8 bit，c2 的值是未定义的
```

类型所能表示的值的范围决定了转换的过程

- 一个整数值赋给浮点类型时，小数部分为 0。如果该整数所占的空间超过浮点类型的容量，精度可能有损失
- 赋给无符号类型一个超出它表示范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数。例如，8 bit 大小的 unsigned char 可以表示 0 至 255 区间内的值，如果赋了一个区间以外的值，则实际的结果是该值对 256 取模后所得的余数。-1 赋给 8bit 大小的 unsigned char 所得结果是 255
- 赋给带符号类型一个超出它表示范围的值时，结果是未定义的（undefined）。此时，程序可能继续工作、可能崩溃，也可能生成垃圾数据。

> **避免无法预知和依赖于实现环境的行为**
>
> 无法预知的行为源于编译器无须（不能）检测的错误。即使代码编译通过了，如果程序执行了一条未定义的表达式，仍有可能产生错误。
>
> 在某些情况和某些编译器下，含有无法预知行为的程序也能正确执行。但是无法保证同样一个程序在别的编译器下能正常工作，甚至已经编译通过的代码再次执行也可能会出错。

含有无符号类型的表达式

> 切勿混用带符号类型和无符号类型

2.1.3 字面值常量

一个形如 42 的值被称作字面值常量（literal）。每个字面值常量都对一种数据类型，字面值常量的形式和值决定了它的数据类型。

整型和浮点型字面值

字符和字符串字面值

转义序列

有两类字符不能直接使用：一是不可打印的字符，如退格或其他控制字符，因为它们没有可视的图符；二是C++中有特殊含义的字符（单引号、双引号、问号、反斜杠）。这时需要用到转义序列，以反斜线开始。

指定字面值的类型

布尔字面值和指针字面值

nullptr 是指针字面值

2.2 变量

变量提供一个具名的、可供程序操作的存储空间。C++中的每个变量都有其数据类型，数据类型决定着变量所占内存空间的大小和布局方式、该空间能存储的值的范围，以及变量能参与的运算。变量和对象一般可以互换使用。

2.2.1 变量定义

> 对象是指一块能存储数据并具有某种类型的内存空间。

初始值

列表初始化

默认初始化

> 未初始化变量引发运行时故障

2.2.2 变量声明和定义的关系

为了允许把程序拆分成多个逻辑部分来编写，C++支持分离式编译（separate compilation）机制，该机制允许将程序分割为若干个文件，每个文件可被独立编译。

如果将程序分为多个文件，则需要有在文件间共享代码的方法。例如，一个文件的代码可能需要使用另一个文件中定义的变量。例子是 std::cout和std::cin，它们定义于标准库，却能被我们的程序使用。

为了支持分离式编译，C++ 将声明和定义区分开来。声明（declaration）使得名字为程序所知，一个文件如果想使用别处定义的名字则必须包含对那个名字的声明。而定义（definiition）负责创建与名字关联的实体。

变量声明规定了变量的类型和名字，这一点上定义与之相同。此外，定义还申请存储空间，也可能会为变量赋一个初始值。

声明一个变量而非定义它，就在变量名前添加关键字 extern，不要显示初始化变量：

```
1 extern int i;    // 声明 i 而非定义 i
2 int j;    // 声明并定义 j
```

任何包含了显示初始化的声明即成为定义。

> 变量能且只能被定义一次，但是可以被多次声明。

如果要在多个文件中使用同一个变量，就必须将声明和定义分离。变量的定义必须出现且只能出现在一个文件中，其他用到该变量的文件必须对其进行声明，不能重复定义。

2.2.3 标识符

2.2.4 名字的作用域

> 当你第一次使用变量时再定义它

2.3 复合类型

2.3.1 引用

> C++11 新增 “右值引用”，主要用于内置类。通常 “引用”，指的是 “左值引用”

引用（reference）为对象起了另外一个名字，引用类型引用（refers to）另外一种类型。

```
int ival = 1024;
int &refVal = ival;    // refVal 指向 ival
int &refVal2;    // 报错：引用必须被初始化
```

初始化变量时，初始值会被拷贝到新建的对象中。然而定义引用时，程序把引用和它的初始值绑定（bind）在一起，而不是将初始值拷贝给引用。一旦初始化完成，引用将和它的初始值对象一直绑定在一起，因为无法令引用重新绑定到另一个对象，因此引用必须初始化。

引用即别名

> 引用并非对象，相反，它只是为一个车已经存在的对象所起的另外一个名字。

定义了一个引用后，对其进行的所有操作都是在与之绑定的对象上进行的。

引用的定义

引用只能绑定到对象上。

2.3.2 指针

指针（pointer）是指向（point to）另外一种类型的复合类型。与引用类似，指针也实现了对其他对象的间接访问。然而指针与引用相比又有很多不同点。

- 指针本身就是一个对象。
- 指针无须在定义时赋初值。和其他内置类型一样，在块作用域内定义的指针如果没有被初始化，也将拥有一个不确定的值。

获取对象的地址

指针存放某个对象的地址，想要获取该地址，需要使用取地址符（操作符&）：

```
int ival = 42;
int *p = &ival;    // p 存放变量ival的地址，或者说p是指向变量ival的指针
```

引用不是对象，没有实际地址，所以不能定义指向引用的指针。

除了后面介绍的两种例外情况，其他所有指针的类型都要和它所指向的对象严格匹配。

指针值

指针的值（即地址）应属下列4种状态之一：

1. 指向一个对象
2. 指向紧邻对象所占空间的下一个位置
3. 空指针，没有指向任何对象
4. 无效指针，上述情况之外的其他值

试图拷贝或以其他方式访问无效指针的值都将引发错误。编译器并不负责检查此类错误，这点和试图使用未经初始化的变量是一样的。

2、3种形式的指针是有效的，但其使用同样受到限制。

利用指针访问对象

指针指向了一个对象，则允许使用解引用符（操作符*）来访问该对象：

```
int ival = 42;
int *p = &ival;    // p存放变量ival的地址，或者说p是指向变量ival的指针
cout << *p;    // 由符号*得到指针p所指的对象，输出 42
```

对指针解引用会得出所指的对象，因此如果给解引用的结果赋值，实际上就是给指针所指的对象赋值。

> 解引用操作仅适用于那些指向了某个对象的有效指针

> **某些符号有多重含义**
>
> & 和 * 这样的符号，既能用作表达式里的运算符，也能作为声明的一部分出现，符号的上下文决定了符号的意义：
>
> ```
> int i = 42;
> int &r = i;    // &随类型名出现，因此是声明的一部分，r 是一个引用
> int *p;    // *随类型名出现，因此是声明的一部分，p 是一个指针
> p = &i;    // &出现在表达式中，是一个取地址符
> *p = i;    // *出现在表达式中，是一个解引用符
> int &r2 = *p;    // &是声明的一部分，*是一个解引用符
> ```

空指针

空指针（null pointer）不指向任何对象，试图使用一个指针之前代码可以先检查它是否为空。

```
// 生成空指针的方法:
int *p1 = nullptr;    //建议使用
int *p2 = 0;
//需要首先 #include cstdlib
int *p3 = NULL;
```

字面值 nullptr，C++11 引入。

NULL，预处理变量，在头文件 cstdlib 中定义，它的值就是0。

预处理器是运行于编译过程之前的一段程序。预处理变量不属于命名空间 std，它由预处理器负责管理。用到一个预处理变量时，预处理器会自动地将它替换为实际值。

不能把 int 变量直接赋给指针。

> **初始化所有指针**
>
> 使用未经初始化的指针是引发运行时错误的一大原因。
>
> 和变量一样，访问未经初始化的指针所引发的后果也是无法预计的。通常会造成程序崩溃，一旦崩溃，定位出错位置很棘手。
>
> 建议初始化所有指针，尽量等定义了对象之后再定义指向它的指针。不清楚指针应该指向何处，就把它初始化为nullptr。

赋值和指针

和其他任何变量（只要不是引用）一样，给指针赋值就是令它存放一个新的地址，从而指向一个新的对象。

其他指针操作

void* 指针

void* 是一种特殊的指针类型，可用于存放任意对象的地址。

2.3.3 理解复合类型的声明

```
// i 是一个int型的数，p 是一个int型指针，r是一个int型引用
int i = 1024, *p = &i, &r = i;
```

定义多个变量

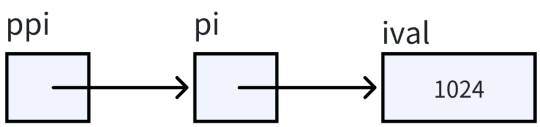
```
int* p1, p2;    // p1是指向int的指针，p2是int
```

指向指针的指针

声明符中修饰符的个数没有限制。当有多个修饰符连写在一起时，按照其逻辑关系详加解释即可。以指针为例，指针是内存中的对象，像其他对象一样也有自己的地址，因此允许把指针的地址再存放到另一个指针当中。

** 表示指向指针的指针，***表示指向指针的指针的指针：

```
int ival = 1024;
int *pi = &ival;    // pi指向一个int型的数
int **ppi = &pi;    // ppi指向一个int型的指针
```



解引用 int 型指针会得到一个 int型的数，解引用指向指针的指针会得到一个指针。此时为了得到最原始的那个对象，需要对指针的指针做两次解引用。

指向指针的引用

指针是对象，存在对指针的引用：

```
int i = 42;
int *p;         // p是一个int型指针
int *&r = p;    // r是一个对指针p的引用

r = &i;         // r引用了一个指针，因此给r赋值&i就是令p指向i
*r = 0;         // 解引用r得到i，也就是p指向的对象，将i值改为0
```

> 一条复杂的指针或引用的声明语句，从右向左阅读有助于弄清楚它的真实含义

2.4 const 限定符

用一个变量来表示缓冲区的大小。

默认状态下，const 对象仅在文件内有效

有着一种 const 变量，它的初始值不是一个常量表达式，但又确实有必要在文件间共享。这种情况下，不希望编译器为每个文件分别生成独立的变量。相反，想让这类 const 对象想其他（非常量）对象一样工作。只在一个文件中定义 const，而在其他多个文件中声明并使用它。

解决方法：对于 const 变量不管声明还是定义都添加 extern 关键字，这样只需定义一次就可以了：

```
// file_1.cc 定义并初始化了一个常量，该常量能被其他文件访问
extern const int bufSize = fcn();
// file_1.h 头文件
extern const int bufSize;    // 与 file_1.cc 中定义的 bufSize 是同一个
```

file_1.cc 定义并初始化了 bufSize，它是一个常量，必须用 extern 加以限定使其被其他文件使用。

file_1.h 头文件中的声明也由 extern 做了限定，作用是指明 bufSize 并非本文件所独有，它的定义将在别处出现。

2.4.1 const 的引用

可以把引用绑定到 const 对象上，就像绑定到其他对象上一样，称之为对常量的引用（reference to const）。与普通引用不同，对常量的引用不能被用作修改它所绑定的对象：

```
const int ci = 1024;
const int &r1 = ci;    // ture，引用及其对应的对象都是常量
r1 = 42;               // flase，r1是对常量的引用
int &r2 = ci;          // false，试图让一个非常量引用指向一个常量对象
```

> C++ 不允许随意改变引用所绑定的对象，所以从这层意义上理解所有的引用都算是常量。引用的对象是常量还是非常量可以决定其所能参与的操作，却不会影响引用和对象的绑定关系本身。

初始化和对 const 的引用

引用的类型必须与其所引用对象的类型一致，但有两个例外：1. 初始化常量引用时允许用任意表达式作为初始值，只要该表达式的结果能转换成引用的类型即可；2. 允许为一个常量引用绑定非常量的对象、字面值，甚至是一般表达式：

```
int i = 42;
const int &r1 = i;        // 允许将cosnt int& 绑定到一个普通int对象上
```

```
3 const int &r2 = 42;        // true, r2 是一个常量引用
4 const int &r3 = r1 * 2;    // true, r3 是一个常量引用
5 int &r4 = r1 * 2;          // false, r4 是一个普通的非常量引用
```

要理解例外情况的原因，简单的办法是弄清楚当一个常量引用被绑定到另外一种类型上时到底发生什么：

```
1 double dval = 3.14
2 const int &ri = dval;
```

为了确保让 ri 绑定一个整数，编译器把上述代码变成如下形式

```
1 const int  temp = dval;    // 由双精度浮点数生成一个临时的整型常量
2 const int &ri = temp;      // 让 ri 绑定这个临时量
```

这种情况下，ri 绑定了一个临时量（temporary）对象。临时量对象就是当编译器需要一个空间来暂存表达式的求值结果时临时创建的一个未命名对象。

ri 不是常量时。对 ri 赋值，就会改变 ri 所引用对象的值。此时绑定对象是一个临时量而非 dval。程序员既然让 ri 引用 dval，就是想通过 ri 改变 dval 的值。因此，基本上不会把引用绑定到临时量上。C++ 把这种行为归为非法。

对 const 的引用可能引用一个并非 const 的对象

常量引用仅对引用可参与的操作作出了限定，对于引用的对象本身是不是一个常量未做限定。因此对象可能是个非常量，允许通过其他途径改变它的值。

2.4.2 指针和 const

与引用一样，也可以令指针指向常量或非常量。类似常量引用，指向常量的指针（pointer to const）不能用于改变其所指对象的值。要存放常量对象的地址，只能使用指向常量的指针：

```
1 const double pi = 3.14;    //
2 double *ptr = &pi;         // flase, ptr是一个普通指针
3 const double *cptr = &pi;  // true, cptr 可以指向一个双精度常量
4 *cptr = 42;                // flase, 不能给 *cptr 赋值
```

和常量引用一样，指向常量的指针也没有规定其所指的对象必须是一个常量。所谓指向常量的指针仅仅要求不能通过该指针改变对象的值，而没有规定那个对象的值不能通过其他途径改变。

const 指针

指针是对象而引用不是，因此就像其他对象类型一样，允许把指针本身定位常量。常量指针（const pointer）必须初始化，而且一旦初始化完成，则它的值（存放在指针中的地址）就不能再改变了。把 * 放在 const 关键字之前用以说明指针是一个常量，还意味着，不变的是指针本身的值，而非指向的那个值：

```
int errNumb = 0;
int *const curErr = &errNumb;    // curErr 将一直指向 errNumb
const double pi = 3.14159;
const double *const pip = &pi;   // pip 是一个指向常量对象的常量指针
```

弄清楚这些声明的含义最有效的办法是从右向左阅读。此例中，离 curErr 最近的符号是 const，意味着 curErr 本身是一个常量对象，对象的类型由声明符的其余部分确定。声明符下一个符号是 *，意思是 curErr 是一个常量指针。最后，声明语句的基本数据类型部分确定了常量指针指向的是一个int对象。同理，可推断出 pip 是一个常量指针，它指向的对象是一个双精度浮点型常量。

指针本身是一个常量并不意味着不能通过指针修改其所指对象的值，能否这样做完全依赖于所指对象的类型。

2.4.3 顶层 const

指针本身是一个对象，它又可以指向另一个对象。因此，指针本身是不是常量以及指针所指的是不是一个常量就是两个相互独立的问题。

顶层 const（top-level const）表示指针本身是个常量，而底层 const（low-level const）表示指针所指的对象是一个常量。

2.4.4 constexpr 和 常量表达式

C++11，允许将变量声明为 constexpr 类型以便由编译器来验证变量的值是否是一个常量表达式。声明为 constexpr 的变量一定是一个常量，而且必须用常量表达式初始化：

```
constexpr int sz = size();    //只有当size是一个constexpr函数时才是正确的声明语句
```

新标准允许定义一种特殊的 constexpr 函数。这种函数足够简单使得编译时就可以计算其结果，这样就可以用 constexpr 函数去初始化 constexpr 变量了。

> 一般如果你认定变量是一个常量表达式，那就把它声明成 constexpr 类型。

字面值类型

常量表达式的值需要在编译时就得到计算，因此声明 constexpr 时用到的类型必须有所限制。这些类型成为 “字面值类型” （literal type）。

算数类型、引用和指针都属于字面值类型。自定义类 Sales_item、IO 库、string 类型则不属于字面值类型，也就不能被定义成 constexpr。

尽管指针和引用都能定义成 constexpr，但它们的初始值却受到严格限制。一个 constexpr 指针的初始值必须是 nullptr 或者 0，或者是存储于某个固定地址中的对象。

6.1.1 节提到，函数体内定义的变量一般来说并非存放在固定地址中，因此 constexpr 指针不能指向这样的变量。相反，定义于所有函数体之外的对象其地址固定不变，能用来初始化 constexpr 指针。允许函数定义一类有效范围超出函数本身的变量，这类变量和定义在函数体之外的变量一样也有固定地址。因此，constexpr 引用能绑定到这样的变量上，constexpr 指针也能指向这样的变量。

指针和 constexpr

在 constexpr 声明中如果定义了一个指针，限定符 constexpr 仅对指针有效，与指针指向的对象无关：

```
1 const int *p = nullptr;        // p是一个指向整型常量的指针
2 constexpr int *q = nullptr;    // q是一个指向整数的常量指针
```

p 和 q 的类型相差甚远，p 是一个指向常量的指针，而 q 是一个常量指针，其中的关键在于 constexpr 把它定义的对象置为了 顶层 const。

2.5 处理类型

2.5.1 类型别名

有两种方法可用于定义类型别名。传统方法是使用关键字 typedef：

```
1 typedef double wages;
2 typedef wages base, *p;
```

新标准规定了一种新的方法，使用别名声明（alias declaration）来定义类型的别名：

```
using SI = Sales_item;
```

指针、常量和类型别名

某个类型别名指代的是复合类型或常量，把它用到声明语句里就会产生意想不到的效果。下面的声明语句用到了类型 pstring，它实际上是类型 char* 的别名：

```
typedef char *pstring;
const pstring cstr = 0;     // cstr是指向char的常量指针
const pstring *ps;          // ps是一个指针，它的对象是指向char的常量指针
```

```
const char *cstr = 0;     // 声明了一个指向const char 的指针
```

2.5.2 auto类型说明符

编程时需要把表达式的值赋给变量，这就要求在声明变量时清楚知道表达式的类型。有时根本做不到。C++11 引入了auto类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应一种特定类型的说明符不同，auto让编译器通过初始值来推算变量的类型。

```
auto item = val1 + val2;
```

复合类型、常量和 auto

编译器推断出来的 auto 类型有时候和初始值的类型不完全一样，编译器会适当改变结果类型使其更符合初始化规则。

2.5.3 decltype类型指示符

有时希望从表达式的类型推断出要定义的变量的类型，但是不想用该表达式的值初始化变量。C++11 引入了第二种类型说明符 decltype，它的作用是选择并返回操作数的数据类型。此过程中，编译器分

析表达式并得到它的类型，却不实际计算表达式的值：

```
decltype (f()) sum = x;     // sum的类型就是函数f的返回类型
```

Delctype 和引用

2.6 自定义数据结构

数据结构就是把一组相关的数据元素组织起来然后使用它们的策略和方法。

2.6.1 定义 Sales_data 类型

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

class body 右侧花括号后必须写一个分号，因为类体后面可以紧跟变量名以示对该类型对象的定义，所以分号不可少；

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;
// 与上一条语句等价，但可能更好一些
struct Sales_data { /* ... */ };
Sales_data accum, trans, *salesptr;
```

2.6.2 使用 Sales_data 类

添加两个 Sales_data 对象

```
#include <iostream>
#include <string>
#include "Sales_data.h"
```

```
int main()
{
    Sales_data data1, data2;
    // 读入data1 和data2 的代码
    // 检查data1 和data2 的ISBN是否相同
    // 如果相同，求 data1和data2 的总和
}
```

Sales_data 对象读入数据

```
    double price = 0;
    // 读入第一笔交易
    std::cin >> data1.bookNo >> data1.units_sold >> price;
    // 计算收入
    data1.revenue = data1.units_sold * price;

    // 读入第二笔交易，同上
```

输出两个 Sales_data 对象的和

```
    if (data1.bookNo == data2.bookNo) {
        ...
    } else {

    }
```

2.6.3 编写自己的头文件

为了确保各个文件中类的定义一致，类通常被定义在头文件中，而且类所在头文件的名字应与类的名字一样。

头文件通常包含那些只能被定义一次的实体，如类、const 和 constexpr 变量等。头文件也经常用到其他头文件的功能。例如，Sales_data 类包含一个 string 成员，所以 Sales_data.h 必须包含 string.h 头文件。同时，使用 Sales_data 类的程序为了能操作 bookNo 成员需要再一次包含 string.h 头文件。

Sales_data 类的程序就先后两次包含了 string.h 头文件。有必要再书写头文件时做适当处理，使其遇到多次包含的情况也能安全和正常地工作。

> 头文件一旦改变，相关的源文件必须重新编译以获取更新过的声明。

预处理器概述

确保头文件多次包含仍能安全工作的技术是预处理器（preprocessor），它由C++从C继承而来。预处理器是在编译之前执行的一段程序，可以部分地改变我们所写的程序。之前用到了一项预处理功能#include，当预处理器看到#include标记时就会用制定的头文件的内容代替#include

C++还会用到的一项预处理功能是头文件保护符（header guard），头文件保护符依赖于预处理变量。预处理变量有两种状态：已定义和未定义。#define 指令把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量是否已经定义：#ifdef 当且仅当变量已定义时 为真，#ifndef 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直到遇到 #endif 指令为止。

使用这些功能有效防止重复包含的发生：

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

第一次包含 Sales_data.h 时，#ifndef 的检查结果为真，预处理器将顺序执行后面的操作直到遇到 #endif 为止。此时，预处理变量 SALES_DATA_H 的值将变为已定义，而且 Sales_data.h 也会被拷贝到我们的程序中来。后面如果再一次包含 Sales_data.h，则#ifndef的检查结果将为假，编译器将忽略#ifndef到#endif之间的部分。

> 预处理变量无视C++中关于作用域的规则。

整个程序中的预处理变量包括头文件保护符必须唯一，通常做法是基于头文件中类的名字来构建保护符的名字，以确保其唯一性。为了避免与程序中的其他实体发生名字冲突，一般把预处理变量的名字全部大写。

> 头文件即使没有被包含在任何其他头文件中，也应该设置保护符。

3. 字符串、向量和数组

3.1 命名空间的 using 声明

每个名字都需要独立的 using 声明

头文件不应包含 using 声明

3.2 标准库类型 string

3.2.1 定义和初始化 string对象

3.2.2 string对象上的操作

读写 string 对象

读取未知数量的 string 对象

使用 getline 读取一整行

string 的 empty 和 size 操作

string::size_type 类型

比较 string 对象

为 string 对象赋值

两个 string对象相加

字面值和 string对象相加

3.2.3 处理string对象中的字符

> 建议使用C++版本的C标准库头文件
>
> C 的头文件形如 name.h，C++ 则将这些文件命名为 cname。
>
> 在名为 cname 的头文件中定义的名字从属于命名空间 std，而定义在 .h 的头文件中的则不然。

处理每个字符？使用基于范围的for语句

使用范围 for 语句改变字符串中的字符

想要改变 string 对象中字符的值，必须把循环变量定义成引用类型。引用值给定对象的一个别名，当使用引用作为循环控制变量时，这个变量实际上被依次绑定到了序列的每个元素上。

只处理一部分字符？

下标运算符、使用下标执行迭代

3.3 标准库类型 vector

vector 表示对象的集合，其中所有对象的类型都相同。集合中的每个对象都有一个与之对应的索引，索引用于访问对象。vector 容纳其他对象，也称作容器（container）。

C++既有类模板（class template），也有函数模板，vector 是一个类模板。只有深入理解C++才能写出模板。

模板本身不是类或函数，相反可以将模板看作编译器生成类或函数编写的一份说明。编译器根据模板创建类或函数的过程成为实例化（instantiation），当使用模板时，需要指出编译器应把类或函数实例化成何种类型。

3.3.1 定义和初始化 vector对象

和任何一种类类型一样，vector 模板控制着定义和初始化向量的方法。

列表初始化 vector 对象

创建指定数量的元素

值初始化

列表初始值还是元素数量?

3.3.2 向 vector对象中添加元素

对 vector 对象来说，直接初始化的方式适用于三种情况：初始值已知且数量较少、初始值是另一个vector对象的副本、所有元素的初始值一样。然而常见情况是：创建一个vector对象时不清楚实际所需的元素个数，元素的值也经常无法确定。还有时候即使元素的初值已知，但这些值总量较大而各不相同，那么在创建vector对象的时候执行初始化操作也会过于繁琐。

运行时利用 vector 的成员函数push_back向其中添加元素。

> vector 对象能高效增长

向 vector 对象添加元素蕴含的编程假定

隐含要求：如果循环体内部包含向vector对象添加元素的语句，则不能使用范围for循环，具体原因在5.4.3节。

> 范围for语句体内不应改变其所遍历序列的大小。

3.3.3 其他 vector操作

vector 提供了几种操作，多数和 string 的相关操作类似

表3.5：vector 支持的操作	
v.empty	
v.size()	
v.push_back(t)	向 v 的尾端添加一个值为t的元素
v[n]	返回v中第n个位置上元素的引用
v1 = v2	
v1 = {a,b,c...}	
v1 == v2	
<, <=, >, >=	以字典顺序进行比较

计算 vector 内对象的索引

不能用下标形式添加元素

> vector 对象（以及string对象）的下标运算符可用于访问已存在的元素，而不能用于添加元素。
>
> 只能对确知已存在的元素执行下标操作
>
> 确保下标合法的一种有效手段就是尽可能使用范围for语句

3.4 迭代器介绍

可以使用下标运算符来访问 string对象的字符或 vector对象的元素，还有另一种更通用的机制也可以实现同样的目的，就是迭代器（iterator）。除了vector外，标准库还定义了其他几种容器。所有标准库容器都可以使用迭代器，但是其中只有少数几种才同时支持下标运算符。严格来说，string对象不属于容器类型，但是string支持很多与容器类型类似的操作。vector支持下标运算符，这点和string一样；string支持迭代器，和vector一样。

类似于指针类型，迭代器也提供了对对象的间接访问，就迭代器而言，其对象是容器中的元素或者string对象中的字符。使用迭代器可以访问某个元素，迭代器也能从一个元素移动到另外一个元素。迭代器有有效和无效之分，这一点和指针差不多。有效的迭代器或者指向某个元素，或者指向容器中尾元素的下一位置；其他所有情况都属于无效。

3.4.1 使用迭代器

和指针不一样，获取迭代器不是使用取地址符，有迭代器的类型同时拥有返回迭代器的成员。这些类型都拥有名为 **begin** 和 **end** 的成员，begin 成员负责返回指向第一个元素（或第一个字符）的迭代器。

```
// 由编译器决定 b 和 e的类型
// b表示v的第一个元素，e表示v尾元素的下一位置
auto b = v.begin(), e = v.end();     //b和e的类型相同
```

end 成员负责返回指向容器（或 string对象） “尾元素的下一位置（one past the end）” 的迭代器，其指示的是容器的一个不存在的 “尾后（off the end）” 元素。end成员返回的迭代器常被称作尾后迭代器（off-the-end iterator）或者简称为尾迭代器（end iterator）。如果容器为空，begin 和 end返回的是同一个迭代器，都是尾后迭代器。

表3.6：标准容器迭代器的运算符

*iter	返回迭代器 iter 所指元素的引用
iter->mem	解引用iter并获取该元素的名为mem的成员，等价于(*iter).mem
++iter	令iter指示容器中的下一个元素
--iter	令iter指示容器中的上一个元素
iter1 == iter2	判断两个迭代器是否相等，如果两个迭代器指示的是同一个元素或者它们是同一个容器的尾后迭代器，则相等；反之不相等
iter1 != iter2	

将迭代器从一个元素移动到另外一个元素

迭代器使用递增（++）运算符来从一个元素移动到下一个元素。逻辑上，迭代器的递增和整数的递增类似。

> end返回的迭代器并不实际指示某个元素，所以不能对其进行递增或解引用的操作。

把stirng对象中第一个单词改写为大写形式，现在利用迭代器及其递增运算符可以实现相同的功能：

```
// 依次处理 s 的字符串直至我们处理完全部字符或者遇到空白
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it);    // 将当前字符改写成大写形式
```

迭代器类型

就像不知道 string 和 vector 的 size_type 成员到底是什么类型一样，一般我们也不知道迭代器的精确类型。实际上，拥有迭代器的标准库类型使用 iterator 和 const_iterator 来表示迭代器的类型：

```
vector<int>::iterator it;    // it能读写vector<int>的元素
string::iterator it2;        // it2能读写string对象中的字符

vector<int>::const_iterator it3;    // it3只能读元素，不能写元素
string::const_iterator it4;         // it4只能读字符，不能写字符
```

const_iterator 和常量指针差不多，能读取但不能修改它所指的元素值。相反，iterator的对象可读可写。如果vector对象或string 对象是一个常量，只能使用 const_iterator；如果vector对象或string对象不是常量，那么既能使用 iterator 和 const_iterator。

begin 和 end 运算符

begin 和 end 返回的具体类型由对象是否是常量决定，如果对象是常量，begin 和 end 返回 const_iterator；不是常量，返回 iterator。

如果对象只需读操作而无需写操作最好使用常量类型（const_iterator）。为了便于专门得到 const_iterator 类型的返回值，C++11 引入了两个新函数，cbegin、cend，不论 vector 对象（或 string对象）本身是否是常量，返回值都是 const_iterator。

结合解引用和成员访问操作

解引用迭代器可获得迭代器所指的对象，如果该对象的类型恰好是类，可能希望进一步访问它的成员。对于一个由字符串组成的 vector 对象来说，想检查其元素是否为空，令 it 是该 vector 对象的迭代器，检查 it 所指字符串是否为空即可。

```
(*it).empty()
```

如果不加圆括号，点运算符将由 it 来执行，而非 it 解引用的结果。

为简化上述表达式，C++ 定义了箭头运算符（->）。箭头运算符将解引用和成员访问两个操作结合在一起。

`it->mem` 和 `(*it).mem` 意思相同。

某些对 vector 对象的操作会使迭代器失效

vector 对象可以动态增长，但是也会有副作用。已知的一个限制是不能在范围 for 循环中向 vector 对象添加元素。另一个限制是任何一种可能改变 vector 对象容量的操作，比如 push_back，都会使该 vector 对象的迭代器失效。9.3.6节将详细解释迭代器是如何失效的。

> 凡是使用了迭代器的循环体，都不要向迭代器所属的容器添加元素。

3.4.2 迭代器运算

string 和 vector 的迭代器提供了更多额外的运算符，一方面可使得迭代器的每次移动跨过多个元素，另外也支持迭代器进行关系运算。所有这些运算被称作迭代器运算（iterator arithmetic）

迭代器的算术运算

使用迭代器运算

使用迭代器运算的一个经典算法是二分搜索。二分搜索从有序序列中寻找某个给定的值。二分搜索从序列中间的位置开始搜索，如果中间位置的元素正好就是要找的元素，搜索完成；如果不是，假如该元素小于要找的元素，则在序列的后半部分继续搜索；假如该元素大于要找的元素，则在序列的前半部分继续搜索。在缩小的范围中计算一个新的中间元素并重复之前的过程，直至最终找到目标或者没有元素可供继续搜索。

```
// 使用迭代器完成二分搜索
// beg 和 end 表示搜索的范围
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2;     //初始状态下的中间点
// 当还有元素尚未检查并且还没有找到 sought 时执行循环
while (mid != end && *mid != sought) {
    if (sought < *mid)     // 要找的元素在前半部分吗?
        end = mid;         // 若是，调整搜索范围使得忽略掉后半部分
    else                   // 要找的元素在后半部分
        beg = mid + 1;     // 在mid之后寻找
    mid = beg + (end - beg)/2     // 新的中间点
}
```

3.5 数组

数组是一种类似于标准库类型 vector 的数据结构，但是在性能和灵活性的权衡上又与 vector 有所不同。与 vector 相似的地方是，数组也是存放类型相同的对象的容器，这些对象本身没有名字，需要通过其所在位置访问。数组大小确定不变，不能随意向数组中增加元素。因为数组的大小固定，因此对某些特殊的应用来说程序的运行时性能较好，但是也损失了一些灵活性。

> 如果不清楚元素的确切个数，请使用 vector。

3.5.1 定义和初始化内置数组

默认情况下，数组的元素被默认初始化。

定义数组的时候必须指定数组的类型，不允许用 auto 关键字由初始值的列表推断类型。另外和 vector 一样，数组的元素应为对象，因此不存在引用的数组。

显示初始化数组元素

字符数组的特殊性

字符数组有一种额外的初始化方式，可以用字符串字面值对此类数组初始化。使用这种方式时，要注意字符串字面值的结尾处还有一个空字符，这个空字符也会像字符串的其他字符一样被拷贝到字符数组中去。

```
1 char a3[] = "C++";            // 自动添加表示字符串结束的空字符
2 const char a4[6] = "Daniel";  // 错误，没有空间存放空字符
```

不允许拷贝和赋值

理解复杂的数组声明

> 理解数组声明的含义，最好的办法是从数组的名字开始按照有内向外的顺序阅读

3.5.2 访问数组元素

使用数组下标时，通常将其定义为 size_t 类型。

与 vector 和 string 一样，遍历数组所有元素时，最好使用范围 for 语句。

检查下标的值

与 vector 和 string 一样，数组的下标是否在合理范围之内由程序员负责检查，合理就是说下标应该大于等于 0 而且小于数组的大小。要防止数组下标越界，除了小心谨慎注意细节以及对代码进行彻底的测试之外，没有其他好办法。即使顺利通过编译并执行，也不能肯定它不包含此类致命的错误。

> 多数常见的安全问题都源于缓冲区溢出错误。当数组或其他类似数据结构的下标越界并试图访问非法内存区域时，就会产生此类错误。

3.5.3 指针和数组

使用数组的时候编译器一般会把它转换成指针。

> 大多数表达式中，使用数组类型的对象其实是使用一个指向该数组首元素的指针。

指针也是迭代器

标准库函数 begin 和 end

> 尾后指针不能执行解引用和递增操作。

指针运算

解引用和指针运算的交互

下标和指针

```
1 内置的下标运算符所用的索引值不是无符号类型，这点与 vector 和 string 不一样
```

3.5.4 C 风格字符串

C 标准库 String 函数

比较字符串

目标字符串的大小由调用者指定

3.5.5 与旧代码的接口

混用 string 对象与 C 风格字符串

使用数组初始化 vector 对象

> **建议：尽量使用标准库类型而非数组**
>
> 使用指针和数组容易出错。一部分原因是概念上的问题：指针常用于底层操作，容易引发一些与细节有关的错误。其他问题则源于语法错误，特别是声明指针时的语法错误。
>
> 应当尽量使用 vector 和迭代器，避免使用内置数组和指针；应该尽量使用 string，避免使用 C 风格的基于数组的字符串

3.6 多维数组

严格来说，C++ 没有多维数组，通常说的多维数组其实是数组的数组。这点对今后理解和使用多维数组大有益处。

多维数组的初始化

多维数组的下标引用

使用范围for语句处理多维数组

> 要是用范围 for 语句处理多维数组，除了最内层的循环外，其他所有循环的控制变量都应该是引用类型。

指针和多维数组

使用多维数组的名字时，也会自动将其转换成指向数组首元素的指针（指向第一个内层数组的指针）。

> 定义指向多维数组的指针时，别忘了这个多维数组实际上是数组的数组

类型别名简化多维数组的指针

小结

string 和 vector 是两种最重要的标准库类型。string 对象是一个可变长的字符序列，vector 对象是一组同类型对象的容器。

迭代器允许对容器中的对象进行间接访问，对于 string 对象和 vector 对象来说，可以通过迭代器访问元素或者在元素间移动。

数组和指向数组元素的指针在一个较低的层次上实现了与标准库类型 stirng 和 vector 类似的功能。一般来说，应该优先选用标准库提供的类型，之后再考虑C++内置的低层替代品数组和指针。

4. 表达式

4.1 基础

4.1.1 基本概念

C++ 定义了一元运算符（unary operator）和二元运算符（binary operator）。作用于一个运算对象的e运算符是一元运算符，如取地址符（&）和解引用符（*）；二元运算符，如相等运算符（==）和乘法运算符（*）。还有作用于三个运算对象的三元运算符。函数调用也是一种特殊的运算符，它对运算对象的数量没有限制。

一些符号既能作为一元运算符也能作为二元运算符。如 *，对于这类符号来说，它的两种用法互不相干，完全可以当成两个不同的符号。

组合运算符和运算对象

对于含有多个运算符的复杂表达式来说，要理解它的含义首先要理解运算符的优先级（precedence）、结合律（associativity）、以及运算对象的求值顺序（order of evaluation）。

运算对象转换

类型转换的规则有点复杂。指针不能转换成浮点数。小整数类型（如bool、char、short）通常会被提升（promoted）较大的整数类型，主要是 int。

重载运算符

左值和右值

C 语言：左值可以位于赋值语句的左侧，右值则不能

C++中，一个左值表达式的求值结果是一个对象或者一个函数，然而以常量对象为代表的某些实际上不能作为赋值语句的左侧运算对象。此外，某些表达式的求值结果是对象，但它们是右值而非左值。可以做一个简单的归纳：当一个对象被用作右值的时候，用的是对象的值（内容）；当对象被用作左值的时候，用的是对象的身份（在内存中的位置）

不同运算符对运算对象的要求各不相同，有的需要左值运算对象、有的需要右值运算对象；返回值也有差异、有的得到左值结果、有的得到右值结果。一个重要原则（13.6节介绍一种例外情况）是在需要右值的地方可以用左值来代替，但是不能把右值当成左值（也就是位置）使用。当一个左值被当成右值使用时，实际使用的是它的内容（值）。

目前为止，有几种熟悉的运算符是要用到左值的：

- 赋值运算符需要一个（非常量）左值作为其左侧运算对象，得到的结果也仍然是一个左值。
- 取地址符作用于一个左值运算对象，返回一个指向该运算对象的指针，该指针是一个右值
- 内置解引用运算符、下标运算符、迭代器解引用运算符、string 和 vector 的下标运算符 的求值结果都是左值
- 内置类型和迭代器的递增递减运算符作用于左值运算对象，其前置版本所得的结果也是左值

使用关键字 decltype 的时候，左值和右值也有所不同。如果表达式的求值结果是左值，decltype 作用于该表达式（不是变量）得到一个引用类型。

4.1.2 优先级与结合律

括号无视优先级和结合律

优先级与结合律有何影响

4.1.3 求值顺序

优先级规定了运算对象的组合方式，但是没有说明运算对象按照什么顺序求值。大多数情况下，不会明确指定求值的顺序。如下表达式

```
1 int i = f1() * f2();
```

我们知道 f1 和 f2 一定会在执行乘法之前被调用，但是无法知道 f1 在 f2 之前调用，还是 f2 在 f1 之前调用。

对于没有指定执行顺序的运算符来说，如果表达式指向并修改了同一个对象，将会引发错误并产生未定义的行为。举例，<< 运算符没有明确规定何时以及如何对运算对象求值，因此下面的输出表达式是未定义的：

```
int i = 0;
cout << i << " " << ++i << endl;    // undefined
```

程序是未定义的，所以无法推断它的行为。编译器可能先求 ++i 的值再求 i 的值，也可能先求 i 的值再求 ++i 的值；甚至编译器还可能做完全不同的操作。因为此表达式的行为不可预知，因此不论编译器生成什么样的代码程序都是错误的。

有4种运算符明确规定了运算对象的求值顺序。一是逻辑与（&&）运算符，它规定先求左侧运算对象的值，只有当左侧运算对象的值为真时才继续求右侧运算对象的值。另外三种分别是逻辑或（||）运算符、条件（?:）运算符和逗号（,）运算符。

求值顺序、优先级、结合律

运算对象的求值顺序与优先级和结合律无关，在一条形如 f() + g() * h() + j() 的表达式中：

- 优先级规定，g() 的返回值和 h() 的返回值相乘
- 结合律规定，f() 的返回值先与 g() 和 h() 的乘积相加，所得结果再与 j() 的返回值相加
- 对于这些函数的调用顺序没有明确规定

如果 f、g、h和j 是无关函数，它们既不会改变同一对象的状态也不执行IO任务，那么函数的调用顺序不受限制。反之，如果其中某几个函数影响同一对象，则它是一条错误的表达式，将产生 undefined 行为。

```
建立：处理复合表达式
1. 拿不准的时候最好用括号来强制让表达式的组合关系符合程序逻辑的要求
2. 如果改变了某个运算对象的值，在表达式的其他地方不要再使用这个运算对象。
规则 2 有一个例外，当改变运算对象的子表达式本身就是另一个子表达式的运算对象时改规则无效。
例如，在表达式 *++iter 中，递增运算符改变 iter 的值，iter（已经改变的值）又是解引用运算符的运算对象。
此时（或类似情况下），求值的顺序才不会成为问题，因为递增运算（即改变运算对象的子表达式）必须先求值，
然后才轮到解引用运算。这是一种常见用法，没有问题。
```

4.2 算术运算符

算术运算符都能作用于任意算数类型以及任意能转换为算数类型的类型。算术运算符的运算对象和求值结果都是右值。在表达式求值之前，小整数类型的运算对象被提升成较大的整数类型，所有运算对象最终会转换成同一类型。

一元正号运算符（+）、加减法运算符都能作用于指针。当一个一元正号运算符作用于一个指针或者算术值时，返回运算对象值的一个（提升后）副本。

一元负号运算符对运算对象值取负后，返回其（提升后的）副本：

```
bool b = true;
bool b2 = -b;    //b2是true
```

布尔值不应该参与运算。

> tips：移除和其他算术运算异常
>
> 算术表达式有可能产生未定义的结果。一部分原因是数学性质本身：例如除数是0的情况；另一部分则源于计算机的特点：例如溢出，当计算的结果超出该类型所能表示的范围时就会产生溢出。
>
> ```
> // 某个机器 short 类型占16位，最大的 short 数值是 32767
> short short_value = 32767;    // short 占16位，能表示的最大值是 32767
> short_value += 1;             // 该计算导致溢出
> ```
>
> 很多系统在编译和运行时都不报出溢出错误，像其他undefined行为一样，溢出的结果是不可预知的。在这个系统中，程序输出是
>
> ```
> short_value: -32768
> ```
>
> 该值发生了 “环绕（wrapped around）”，符号位本来是0，溢出改成了1，结果变成了负值。别的系统可能有其他结果，程序的行为可能不同甚至直接崩溃。

4.3 逻辑和关系运算符

关系运算符作用于算术类型或指针类型，逻辑运算符作用于任意能转换成布尔值的类型。逻辑运算符和关系运算符的返回值都是布尔类型。值为0的运算对象（算术类型或指针类型）表示 false，否则表示 true。对这两类运算符来说，运算对象和求值结果都是右值。

逻辑与、逻辑或运算符

对于逻辑与运算符（&&）、逻辑或运算符（||）都是先求左侧运算对象的值再求右侧运算对象的值，当且仅当左侧运算对象无法确定表达式结果时才会计算右侧运算对象的值。这种策略称为短路求值（short-circuit evaluation）

下面的逻辑与运算符，它们的左侧运算对象是为了确保右侧对象求值过程的正确性和安全性。

```

```

```
index != s.size() && !isspace(s[index])
```

首先检查 index 是否到达 string 对象的末尾，以此确保只有当 index 在合理范围之内时才会计算右侧运算对象的值。

举例，有一个存储若干 string 对象的 vector 对象，要求输出 stirng 对象的内容并且在遇到空字符串或者以句号结束的字符串时进行换行。使用基于范围的 for 循环处理 string 对象中的每个元素：

```
// s是对常量的引用；元素既没有被拷贝也不会被改变
for (const auto &s : text) {
    cout << s;
    // 遇到空字符串或者以句号结束的字符串进行换行
    if (s.empty() || s[s.size() -1] == '.')
        cout << endl;
    else
        cout << " ";
}
```

利用逻辑或运算符的短路求值策略确保只有当 s 非空时才会用下标运算符去访问它。

注意，s 被声明成了对常量的引用。因为 text 的元素是 string 对象，可能非常大，所以将 s 声明成引用类型可以避免对元素的拷贝；因为不需要对 string 对象做写操作，所以 s 被声明成对常量的引用。

逻辑非运算符

关系运算符

关系运算符比较运算对象的大小关系并返回布尔值。

相等性测试与布尔字面值

> 进行比较运算时除非比较的对象是布尔类型，否则不要使用布尔字面值 true 和 false 作为运算对象。

4.4 赋值运算符

赋值运算符的左侧运算对象必须是一个可修改的左值。

赋值运算满足右结合律

赋值运算优先级较低

> 赋值运算符的优先级低于关系运算符的优先级，所以在条件语句中，赋值部分通常应该加上括号

切勿混淆相等运算符和赋值运算符

复合赋值运算符

```
1 +=    -=    *=    /=    %=    // 算术运算符
2 <<=   >>=   &=    ^=    |=    // 位运算符
```

任何一种复合运算符都完全等价于

`a = a op b;`

唯一的区别是左侧运算对象的求值次数：使用复合运算符只求值一次，使用普通运算符则求值两次。两次包括：一次是作为右边子表达式的一部分求值，另一次是作为赋值运算的左侧运算对象求值。这种区别除了对程序性能有些许影响外几乎可以忽略不计。

4.5 递增和递减运算符

前置后置。目前为止，本书使用的都是前置版本。前置形式的运算符首先将运算对象加1（或减1），然后将改变后的对象作为求值结果。后置版本也会将运算对象加1，但是求值结果是运算对象改变之前的那个值的副本。

```
1 int i = 0, j;
2 j = ++i;    // j = 1,  i = 1: 前置版本得到递增之后的值
3 j = i++;    // j = 1,  i = 2: 后置版本得到递增之前的值
```

两种运算符必须作用于左值运算对象。前置版本将对象本身作为左值返回，后置版本则将对象原始值的副本作为右值返回。

> 建议：尽量不用递增递减运算符的后置版本
>
> 前置版本的递增运算符避免了不必要的工作，它把值加 1 后直接返回改变了的运算对象。后置版本需要将原始值存储下来以便返回这个未修改的内容。如果不需要修改前的值，那么后置版本的操作就是一种浪费。
>
> 对于整数和指针类型来说，编译器可能对这种额外的工作进行了一定优化；但是对于相对复杂的迭代器类型，这种额外的工作消耗巨大。建议使用前置版本，不仅不需要担心性能的问题，而且写出的代

> 码更符合编程的初衷。

在一条语句中混用解引用和递增运算符

想在一条复合表达式中既将变量加 1 或减 1又能使用它原来的值，这时可以使用递增和递减运算符的后置版本。

举例，可以使用后置的递增运算符来控制循环输出一个 vector 对象内容直至遇到（但不包括）第一个负值为止：

```
auto pbeg = v.begin();
// 输出元素直至遇到第一个负值为止
while (pbeg != v.end && *beg >= 0)
    cout << *pbeg++ << endl;    // 输出当前值并将pbeg向前移动一个元素
```

`*pbeg++` 不太容易理解，这种写法非常普遍，一定要理解其含义。

后置递增运算符的优先级高于解引用运算符，因此 `*pbeg++` 等价于 `*(pbeg++)`。`pbeg++` 把 pebg 的值加1，然后返回 pbeg 的初始值的副本作为其求值结果，此时解引用运算符的运算对象是 pbeg 未增加之前的值。最终，这条语句输出 pbeg 开始时指向的那个元素，并将指针向前移动一个位置。

这种用法基于一个事实，既后置递增运算符返回初始的未加1的值。如果返回的是加1后的值，解引用该值将产生错误的结果。不但无法输出第一个元素，而且如果序列中没有负值，程序将可能试图解引用一个根本不存在的元素。

> 建议：简洁
>
> 形如 *pbeg++ 的表达式一开始可能不太容易理解，但这是一种被广泛使用的、有效的写法。
>
> `cout << *iter++ << endl;`
>
> 要比下面的等价语句更简洁，更少出错
>
> `cout << *iter << endl;`
>
> `++iter;`

运算对象可按任意顺序求值

多数运算符都没有规定运算对象的求值顺序，一般情况不会有什么影响。但是，如果一条子表达式改变了某个运算对象的值，另一条子表达式又要使用该值的话，运算对象的求值顺序就很关键了。因为递增递减运算符会改变运算对象的值，所以要提防在符合表达式中错用这两个运算符。

为说明这一问题，程序使用 for 循环将输入的第一个单词改写成大写

```
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it);    //将当前字符改写成大写
```

上面程序，把解引用 it 和递增 it 分开来完成。如果用一个看似等价的 while 循环代替

```
// 该循环的行为是未定义的!
while (beg != s.end() && !isspace(*beg))
    *beg = toupper(*beg++);    // 错误：该赋值语句未定义
```

产生undefined行为。赋值运算符左右两端的运算对象都用到了 beg，并且右侧运算符对象还改变了 beg 的值，所以该赋值语句是未定义的。编译器可能按照下面的任意一种思路处理该表达式：

```
*beg = toupper(*beg);         //如果先求左侧的值
*(beg + 1) = toupper(*beg);   //如果先求右侧的值
```

也可能使用别的什么方式处理它

4.6 成员访问运算符

点运算符和箭头运算符都可用于访问成员。点运算符获取类对象的一个成员；箭头运算符与点运算符有关，表达式 `ptr->mem` 等价于 `(*ptr).mem`

箭头运算符作用于一个指针类型的运算对象，结果是一个左值。点运算符分成两种情况：如果成员所属的对象是左值，那么结果是左值；反之，如果成员所属的对象是右值，那么结果是右值。

4.7 条件运算符

条件运算符（?:）允许把简单的 if-else 逻辑嵌入到单个表达式当中

```
cond ? expr1 : expr2;
```

cond 是判断条件的表达式，而 expr1 和 expr2 是两个类型相同或可能转换为某个公共类型的表达式。条件运算符的执行过程是，先求 cond 的值，if true 对 expr1 求值并返回值，否则对 expr2 求值并返回该值。

当条件运算符的两个表达式都是左值或者能转换成同一种左值类型时，运算结果是左值，否则运算结果是右值。

嵌套条件运算符

条件运算的嵌套最好别超过两到三层。

在输出表达式中使用条件运算符

4.8 位运算符

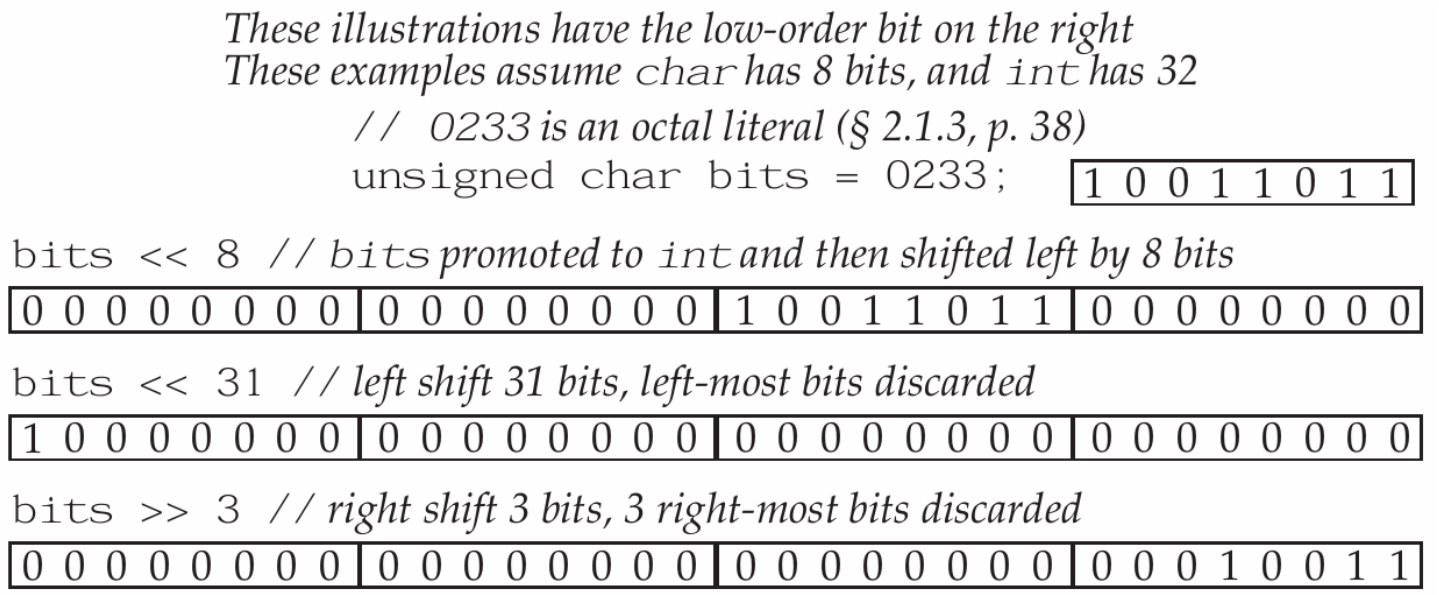
位运算符作用于整数类型的运算对象，并把运算对象看成是二进制位的集合。位运算符提供检查和设置二进制位的功能。bitset 的标准库类型也可以表示任意大小的二进制位集合，所以位运算符同样能作用于 bitset 类。

一般运算对象是小整型，它的值会被自动提升成较大的整数类型

> 关于符号位如何处理没有明确的规定，建议仅将位运算符用于处理无符号类型。

位移运算符

左侧运算对象的内容按照右侧运算对象的要求移动指定位数，然后将经过移动的（可能还进行了提升）左侧运算对象的拷贝作为求值结果。其中，右侧的运算对象一定不能为负，而且值必须严格小于结果的位数，否则就会产生未定义的行为。二进制位左移、右移，移出边界之外的位被舍弃掉。



位求反运算符

位求反运算符（~）将运算对象逐位求反后生成一个新值，将1置位0、将0置为1。

char类型的运算符首先提升成 int 类型，提升时运算对象原来的位保持不变。

位与、位或、位异或运算符

异或运算符（~），如果两个运算对象的对应位置有且只有一个为1则运算结果中该位为1，否则为0.

使用位运算符

使用位运算符的例子：假设班级有30个学生，老师每周都会对学生进行一次测验，测验的结果只有通过和不通过两种。为了更好地追踪测验的结果，用一个二进制位代表某个学生再一次测验中是否通过，显然全班的测验结果可以用一个无符号整数来表示

```
1 unsigned long quiz1 = 0;     // 把这个值当成是位的集合来使用
```

unsigned long 在任何机器上都至少拥有 32位；给 quiz1 赋一个明确的初始值，使得它的每一位在开始时都有统一且固定的值。

教师必须有权设置并检查每一个二进制位。例如，我们需要对序号为27的学生对应的位进行设置，以表示他通过了测验。为了达到这一目的，首先创建一个值，该值只有第 27位是1其他位都是0，然后将这个值与quiz1进行位或运算，这样就能强行将guiz1 的第27位设置为1，其他位都保持不变。

为了实现本例的目的，我们将guiz1的低阶位赋值为0、下一位赋值为1，以此类推，最后统计guiz1各个位的情况。

使用左移运算符和一个unsigned long类型的整数字面值1就能得到一个表示学生27通过了测验的数值：

```
1 1UL << 27     //生成一个值，该值只有第27位为1
```

接下来将所得的值与 quiz1 进行位或运算。为了同时更新 quiz 1 的值，使用一条复合赋值语句

```
1 quiz1 |= 1UL << 27;     // 表示学生 27 通过了测验
```

重新核对测验结果时发现学生 27 实际没有通过测验，必须把第27位的值置为0.

```
1 quiz1 &= ~(1UL << 27);     // 学生27没有通过测验
```

检查学生 27 测验的情况到底怎样

```
1 bool status = quiz1 & (1UL << 27);     // 27是否通过了测验?
```

位移运算符（IO运算符）满足左结合律

```
cout << "hi" << "there" << endl;
// 等同于
((cout << "hi") << "there") << endl;
```

移位运算符的优先级不高不低，介于中间：比算术运算符的优先级低，但比关系运算符、赋值运算符和条件运算符的优先级高。

4.9 sizeof运算符

sizeof 运算符返回一条表达式或一个类型名字所占的字节数。sizeof 运算符满足左结合律，所得的值是一个 size_t 类型的常量表达式。

sizeof运算符的结果部分地依赖于其作用的类型：

- 对char或者类型为char的表达式执行sizeof运算，结果得1。
- 对引用类型执行sizeof运算得到被引用对象所占空间的大小。
- 对指针执行sizeof运算得到指针本身所占空间的大小。
- 对解引用指针执行sizeof运算得到指针指向的对象所占空间的大小，指针不需有效。
- 对数组执行sizeof运算得到整个数组所占空间的大小，等价于对数组中所有的元素各执行一次sizeof运算并将所得结果求和。注意，sizeof运算不会把数组转换成指针来处理。
- 对string对象或vector对象执行sizeof运算只返回该类型固定部分的大小，不会计算对象中的元素占用了多少空间。

4.10 逗号运算符

4.11 类型转换

何时发生隐式类型转换

4.11.1 算数转换

整型提升

整形提升（integral promotion）负责把小整数类型转换成较大的整数类型。对于 bool、char、signed char、unsigned char、short 和 unsigned short 等类型来说，只要它们所有可能得值都能存在 int 里，它们就会提升成 int 类型；否则，提升成 unsigned int 类型。像 布尔值 false 提升成 0、true 提升成 1。

较大的 char 类型（wchar_t、char16_t、char32_t）提升成int、unsigned int、long、unsigned long、long long 和 unsigned long long 中最小的一种类型，前提是转换后的类型要能容纳原类型所有可能的值。

无符号类型的运算对象

理解算术转换

4.11.2 其他隐式类型转换

数组转换成指针

指针的转换

转换成常量

类类型定义的转换

4.11.3 显式转换

命名的强制类型转换

> reinterpret_cast 本质依赖于机器。想安全使用 reinterpret_cast 必须对涉及的类型和编译器实现转换的过程都非常了解。

> 避免强制类型转换

4.12 运算符优先级表

小结

术语

5. 语句

C++ 提供了一组控制流（flow-of-control）语句以支持更复杂的执行路径。

5.1 简单语句

空语句

语法上需要一条语句但是逻辑上不需要，此时应该使用空语句。

```
;    //空语句
```

复合语句（块）

复合语句（compund statement）是指用花括号括起来的语句和声明的序列，复合语句也称作块（block）。一个块就是一个作用域。

语法上需要一条语句，但是逻辑上需要多条语句，则应该使用复合语句。例如，while 或 for 的循环体必须是一条语句，但是常常需要在循环体内做很多事情，此时需要将多条语句用花括号括起来，把语句序列转变成块。

5.2 语句作用域

5.3 条件语句

5.4 迭代语句

范围 for 语句

范围 for 语句可以遍历容器或其他序列的所有元素。范围 for 语句（range for statement）

```
for (declaration : expression)
    statement
```

expression 表示的是一个序列，比如初始值列表、数组、vector 或 string 等类型的对象，它们的共同特点是拥有能返回迭代器的 begin 和 end 成员。

```
vector<int> v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
// 范围变量必须是引用类型，这样才能对元素执行写操作
for (auto &r : v)    // 对于 v 中的每一个元素
    r *= 2;    // 将 v 中每个元素的值翻倍
```

5.5 跳转语句

break

continue

goto

return

5.6 try 语句块和异常处理

异常是指存在于运行时的反常行为，这些行为超出了函数正常功能的范围。典型异常包括失去数据库连接以及遇到意外输入等。处理反常行为可能是涉及所有系统最难的一部分。

程序的某部分检测到无法处理的问题时，需要用到异常处理。此时检测出问题的地方应该发出某种信号表明程序遇到了故障，无法继续，而且信号的发出方无需知道故障将在何处解决。一旦发出异常信号，检测出问题的部分就完成了任务。

如果程序中含有可能引发异常的代码，那么通常会有专门的代码处理问题。例如，如果程序的问题是输入无效，则异常处理部分可能会要求用户重新输入正确的数据；如果丢失了数据库连接，会发出报警信息。

异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持。C++ 中异常处理包括：

- throw 表达式（throw expression），异常检测部分使用 throw 表达式表示它遇到了无法处理的问题。throw 引发（raise）了异常
- try 语句块（try block），异常处理部分使用 try 语句出差异常。try 语句块以关键字 try 开始，并以一个或多个 catch 子句（catch clause）结束。try 语句块中代码抛出的异常通常会被某个 catch 子句处理。catch 子句被称作异常处理代码（exception handler）。
- 一套异常类（exception class），用于在 throw 表达式和相关的 catch 子句之间传递异常的具体信息。

5.6.1 throw 表达式

5.6.2 try 语句块

编写处理代码

函数在寻找处理代码的过程中退出

> tips：编写异常安全的代码非常困难
>
> 异常中断了程序的正常流程。异常发生时，调用者请求的一部分计算可能已经完成了，另一部分尚未完成。通常情况下，略过部分程序意味着某些对象处理到一半就戛然而止了，从而导致对象处于无效或未完成的状态，或者资源没有正常释放，等等。那些在异常发生期间正确执行了 “清理” 工作的程序被称作异常安全（exception safe）的代码。经验表明，编写异常安全的代码非常困难，这部分只是也超出了本书的范围。
>
> 对于一些程序来说，当程序发生时只是简单地终止程序。此时，不怎么需要担心异常安全的问题。
>
> 但是对于那些确实要处理异常并继续执行的程序，就要加倍注意。必须时刻清楚异常何时发生，异常发生后程序应如何确保对象有效、资源无泄漏、程序处于合理状态，等等。
>
> 本书介绍一些比较常规的提升异常安全性的技术。但是，如果程序要求非常鲁棒的异常处理，这些技术还不够。

5.6.3 标准异常

C++标准库定义了一组类，用于报告标准库函数遇到的问题。这些异常类也可以在用户编写的程序中使用，它们分别定义在4个头文件中：

- exception 头文件定义了最通用的异常类 exception。它只报告异常的发生，不提供任何额外信息。
- stdexcept 头文件定义了一种常用的异常类
- new 头文件定义了 bad_alloc 异常类型
- type_info 头文件定义了 bad_cast 异常类型

表5.1：<stdexcept> 定义的异常类

exception	最常见的问题
runtime_error	只有在运行时才能检测出问题
range_error	运行时错误：生成的结果超出了有意义的值域范围

overflow_error	运行时错误：计算上溢
underflow_error	运行时错误：计算下溢
logic_error	程序逻辑错误
domain_error	逻辑错误：参数对应的结果值不存在
invalid_argument	逻辑错误：无效参数
length_error	逻辑错误：创建一个超出该类型最大长度的对象
out_of_range	逻辑错误：使用一个超出有效范围的值

小结

术语

6. 函数

6.1 函数基础

编写函数

调用函数

函数调用完成两项工作：一是用实参初始化函数对应的形参，二是将控制权转移给被调用函数。此时，主调函数（calling function）的执行被暂时中断，被调函数（called function）开始执行。

形参和实参

函数的形参列表

函数返回类型

函数的返回类型不能是数组类型或函数类型，但可以是指向数组或函数的指针。

6.1.1 局部对象

C++中，名字有作用域，对象有生命周期（lifetime）。理解这两个概念非常重要。

- 名字的作用域是程序文本的一部分，名字在其中可见
- 对象的生命周期是程序执行过程中该对象存在的一段时间

函数体是一个语句块。块构成一个新的作用域，可以在其中定义变量。形参和函数体内部定义的变量统称为局部变量（local variable）。它们对函数而言是局部的，仅在函数的作用域内可见，同时局部变量还会隐藏（hide）在外层作用域中同名的其他所有声明中。

在所有函数体之外定义的对象存在于程序的整个执行过程中。此类对象在程序启动时被创建，直到程序结束时才会销毁。局部变量的生命周期依赖于定义的方式。

自动对象

对于普通局部变量对应的对象来说，当函数的控制路径经过变量定义语句时创建该对象，当到达定义所在的块末尾时销毁它。把只存在于块执行期间的对象称为自动对象（automatic object）。当块的执行结束后，块中创建的自动对象的值就变成未定义的了。

形参是一种自动对象。函数开始时为形参申请存储空间，因为形参定义在函数体作用域之内，所以一旦函数终止，形参也就被销毁。

用传递给函数的实参初始化形参对应的自动对象。对于局部变量对应的自动对象来说，分为两种情况：如果变量定义本身含有初始值，就用这个初始值进行初始化；否则，如果变量定义本身不含初始值，执行默认初始化。意味着内置类型的未初始化局部变量将产生undefined的值。

局部静态对象

某些时候，有必要令局部变量的生命周期贯穿函数调用及之后的时间。可以将局部变量定义成 static 类型从而获得这样的对象。局部静态对象（local static object）在程序的执行路径第一次经过对象定义语句时初始化，并且直到程序终止才被销毁，在此期间即使对象所在的函数结束执行也不会对它有影响。

```
1  // 函数统计自己被调用了多少次
2  size_t count_calls() {
3      static size_t ctr = 0;    // 调用结束后，该值仍有效
4      return ++ctr;
5  }
6
7  int main() {
8      for (size_t i = 0; i != 10; ++i)
9          cout << count_calls() << endl;
10     return 0;
11 }
```

6.1.2 函数声明

函数的声明和函数的定义非常类似，唯一的区别是函数声明无须函数体，用一个分号替代即可。

函数的三要素（返回类型、函数名、形参类型）描述了函数的接口，说明了调用该函数所需的全部信息。函数声明也称作函数原型（function prototype）。

在头文件中进行函数声明

> 含有函数声明的头文件应该被包含到定义函数的源文件中

6.1.3 分离式编译

C++ 支持分离式编译（separate compilation）。分离式编译允许我们把程序分割到几个文件中去，每个文件独立编译。

编译和链接多个源文件

举例，fact 函数的定义位于一个名为 fact.cc 的文件中，它的声明位于名为 Chapter6.h 的头文件中。另外，在名为 factMain.cc 的文件中创建 main 函数，main 函数将调用 fact 函数，要生成可执行文件（executable file），必须告诉编译器我们用到的代码在哪里。对于上述文件，编译的过程如下：

```
$ CC factMain.cc fact.cc    # generates factMain.exe or a.out
$ CC factMain.cc fact.cc -o main # generates main or main.exe
```

CC 是编译器的名字，$ 是系统提示符，# 后面是命令行下的注释语句。

如果修改了其中一个源文件，只需重新编译那个改动了的文件。大多数编译器提供了分离式编译每个文件的机制，这一过程通常会产生一个后缀名是 .obj（Windows）或 .o（UNIX）的文件，后缀名的含义是该文件包含对象代码（object code）。

接下来编译器负责把对象文件链接在一起形成可执行文件。在我们系统中，编译的过程如下：

```
$ CC -c factMain.cc    # generates factMain.o
$ CC -c fact.cc    # generates fact.o
$ CC factMain.o fact.o    # generates factMain.exe or a.out
$ CC factMain.o fact.o -o main    # generates main or main.exe
```

可以阅读编译器用户手册，弄清楚有多个文件组成的程序是如何编译并执行的。

6.2 参数传递

每次调用函数 时都会重新创建它的形参，并用传入的实参对形参进行初始化。

> 形参初始化的机理与变量初始化一样。

和其他变量一样，形参的类型决定了形参和实参交互的方式。如果形参是引用类型，它将绑定到对应的实参上；否则，将实参的值拷贝后赋给形参。

形参是引用类型时，我们说它对应的实参被引用传递（passed by reference）或者函数被传引用调用（called by reference）。和其他引用一样，引用形参也是它绑定的对象的别名（即引用形参是它对应实参的别名）。

当实参的值被拷贝给形参时，形参和实参是两个相互独立的对象。我们说这样的实参被值传递（passed by value）或者函数被传值调用（called by value）。

6.2.1 传值参数

指针形参

> 熟悉 C 的程序员常常使用指针类型的形参访问函数外部的对象。C++ 中，建议使用引用类型的形参替代指针。

6.2.2 传引用参数

使用引用避免拷贝

拷贝大的类类型对象或者容器对象比较低效，甚至有的类类型（包括 IO 类型在内）根本就不支持拷贝操作。当某种类型不支持拷贝操作时，函数只能通过引用形参访问该类型的对象。

编写一个函数比较两个 string 对象的长度。因为 string 对象可能会非常长，所以应该尽量避免直接拷贝它们，这时使用引用形参是比较明智的选择。又因为比较长度无须改变 string 对象的内容，所以把形参定义成对常量的引用。

```
1 // 比较两个string 对象的长度
2 bool isShorter(const string &s1, const string &s2) {
3     return s1.size() < s2.size()
4 }
```

当函数无须修改引用形参的值时最好使用常量引用。

使用引用形参返回额外信息

一个函数只能返回一个值，然而有时函数需要同时返回多个值，引用形参为我们一次返回多个结果提供了有效的途径。

举例，定义一个名为 find_char 的函数，它返回在 string 对象中某个指定字符第一次出现的位置。同时，希望函数能返回该字符出现的总次数。

如何定义函数使得它既能返回位置也返回出现次数呢？一种方法是定义一个新的数据类型，让它包含位置和数量两个成员。还有更简单的方法，可以给函数传入一个额外的引用实参，令其保存字符出现的次数。

6.2.3 const 形参和实参

顶层 const 作用于对象本身

```
const int ci = 42;     // 不能改变ci
int i = ci;            // 正确，拷贝 ci，忽略顶层const
int * const p = &i;    // const是顶层的，不能给p赋值
*p = 0;                // 正确：通过 p 改变对象的内容是可以的
```

```
void fcn(const int i) { /* fcn能够读取 i，不能向 i 写值*/ }
void fcn(int i) { /* ... */ }    //错误，重复定义了 fcn(int)
```

指针或引用形参与 const

尽量使用常量引用

6.2.4 数组形参

数组的两个特殊性质对我们定义和使用作用在数组上的函数有影响。两个性质分别是：不允许拷贝数组以及使用数组时（通常）会将其转换成指针。不能拷贝数组，所以无法以值传递的方式使用数组参数。因为数组会被转换成指针，所以为函数传递一个数组时，实际传递的是指向数组首元素的指针。

可以把形参写成类似数组的形式：

```
// 尽管形式不同，但这三个 print 函数是等价的
void print(const int*);
```

```
3 void print(const int[]);     // 可以看出函数的意图是作用于一个数组
4 void print(const int[10]);   // 维度表示期望数组含有多少元素，实际不一定
```

> 和其他使用数组的代码一样，以数组作为形参的函数也必须确保使用数组时不会越界

数组是以指针的形式传递给参数的，所以一开始函数并不知道数组的确切尺寸，调用者应该为此提供一些额外的信息。管理指针形参有三种常用技术

使用标记指定数组长度

管理数组实参的方法一是要求数组本身含有一个结束标记。典型示例是 C 风格字符串。C 字符串存储在字符数组中，并且在最后一个字符后面跟着一个空字符。函数在处理 C 字符串时遇到空字符停止。

该方法适用于那些有明显结束标记且该标记不会与普通数据混淆的情况，但是对于像 int

这样所有取值都是合法值的数据不太有效。

使用标准库规范

管理数组实参的方法二是传递指向数组首元素和尾后元素的指针，这种方法收到了标准库技术的启发。

显示传递一个表示数组大小的形参

方法三

数组形参和 const

数组引用形参

```
1 f(int &arr[10])      //错误：将 arr 声明成了引用的数组
2 f(int (&arr)[10])    //正确，arr是整型数组的引用
```

传递多维数组

6.2.5 main：处理命令行选项

6.2.6 含有可变形参的函数

有时无法预知应该向函数传递几个实参。为了编写能处理不同数量的实参的函数，C++11 提供了两种主要方法：如果所有的实参类型相同，可以传递一个名为 initializer_list 的标准库类型；如果实参类型不同，可以编写一种特殊的函数，就是所谓的可变参数模版。

C++还有一种特殊的形参类型（省略符），可以用它传递可变数量的实参。一般只用于与C函数交互的接口程序。

initializer_list 形参

省略符形参

6.3 返回类型和 return语句

6.3.1 无返回值函数

6.3.2 有返回值函数

> 在含有retrun语句的循环后面应该也有一条 return 语句，没有的话该程序就是错误的。很多编译器无法发现此类错误。

值是如何被返回的

返回一个值的方式和初始化一个变量或形参的方式完全一样：返回的值用于初始化调用点的一个临时量，该临时量就是函数调用的结果。

必须注意当函数返回局部变量时的初始化规则。

```
1 // 如果ctr的值大于1，返回word的复数形式
2 string make_plural(size_t ctr, const string &word,
3                                const string &ending)
4 {
5     return (ctr > 1) ? word + ending : word;
6 }
7
```

该函数的返回类型是 string，意味着返回值被拷贝到调用点。该函数将返回 word 的副本或者一个未命名的临时 string 对象，该对象内容是 word 和 ending 的和。

同其他引用类型一样，如果函数返回引用，该引用只是它引用对象的一个别名。

```
1 // 挑出两个 string 对象中较短的那个，返回其引用
```

```
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size ? s1 : s2;
}
```

其中形参和返回类型都是 const string 的引用，不管是调用函数还是返回结果都不会真正拷贝 string 对象。

不要返回局部对象的引用或指针

函数完成后，它所引用的存储空间随之被释放掉。函数终止意味着局部变量点的引用将指向不再有效的内存区域：

```
// 错误：这个函数试图返回局部对象的引用
const string &manip() {
    string ret;
    //以某种方式改变一下ret
    if (!ret.enpty())
        return ret;     // 错误：返回局部对象的引用
    else
        return "Empty";     // 错误：“Empty”是一个局部对象
}
```

两条 return 语句都将返回未定义的值。第一条 return，它返回的是局部对象的引用。第二条 return，字符串字面值转换成一个局部临时 string 对象。

> 要确保返回值安全，问下自己：引用所引的实在函数之前已经存在的哪个对象？

返回局部对象的引用是错误的；同样，返回局部对象的指针也是错误的。一旦函数完成，局部对象被释放，指针指向一个不存在的对象。

返回类类型的函数和调用运算符

如果函数返回指针、引用或类的对象，就能使用函数调用的结果访问结果对象的成员。

引用返回左值

函数的返回类型决定函数调用是否是左值。

列表初始化返回值

主函数 main 的返回值

main 函数的返回值可以看做是状态指示器。返回 0 表示执行成功，其他表示执行失败。

递归

函数调用了它自身，不管调用是直接的还是间接的，都称该函数为递归函数（recursive function ）

6.3.3 返回数组指针

数组不能被拷贝，所以函数不能返回数组。不过函数可以返回数组的指针或引用。语法上，想要定义一个返回数组的指针或引用的函数比较繁琐。有方法可以简化这一任务，其中最直接的方法是使用类型别名：

```
typedef int arrT[10];     // arrT是一个类型别名，它表示的类型是含有10个整数的数组
using arrT = int[10];     // arrT的等价声明
arrT* func(int i);        // func返回一个指向含有10个整型数组的指针
```

声明一个返回数组指针的函数

想要在声明 func 时不使用类型别名，必须牢记被定义的名字后面数组的维度：

```
int arr[10];         // arr
int *p1[10];         // p1是一个含有10个指针的数组
int (*p2)[10] = &arr;     // p2是一个指针，它指向含有10个整型的数组
```

和声明一样，想定义一个返回数组指针的函数，则数组的维度必须跟在函数名字之后。然而，函数的形参列表也跟在函数名字后面且形参列表应该先与数组的维度。因此，返回数组指针的函数形式如下：

```
Type (*function(parameter_list) [dimension]
```

类似于其他数组的声明，Type 表示元素的类型，dimension 表示数组的大小。
(*function(parameter_list)) 两端的括号必须存在，就像定义 p2 时两端必须有括号一样。没有这对括号，函数的返回类型将是指针的数组。

```
// 举例，func函数的声明没有使用类型别名
int (*func(int i))[10];
```

```
3
4 // 可以按照以下的顺序来逐层理解该声明的含义：
5 // func(int i) 表示调用 func 函数时需要一个 int 类型的实参
6 // (*func(int i)) 意味着我们可以对函数调用的结果执行解引用操作
7 // (*func(int i))[10] 表示解引用 func 的调用将得到一个大小是 10 的数组
8 // int (*func(int i))[10] 表示数组中的元素是 int 类型
```

使用尾置返回类型

C++11还有一种方法简化上述 func 声明，就是使用尾置返回类型（trailing return type）。任何函数的定义都能使用尾置返回，但是这种形式对于返回类型比较复杂的函数最有效，比如返回类型是数组的指针或者数组的引用。尾置返回类型跟在形参列表后面并以一个 -> 符号开头。为了表示函数真正的返回类型跟在形参列表之后，我们在本应该出现返回类型的地方放置一个 auto:

```
1 // func 接受一个 int 类型的实参，返回一个指针，该指针指向含有10个整数的数组
2 auto func(int i) -> int(*)[10];
```

使用 decltype

还有种情况，如果知道函数返回的指针将指向哪个数组，就可以使用 decltype 关键字声明返回类型。例如，下面的函数返回一个车支付，该指针根据参数 i 的不同指向两个已知数组中的某一个：

```
1 int odd[] = {1,3,5,7,9};
2 int even[] = {0,2,4,6,8};
3 // 返回一个指针，该指针指向含有5个整数的数组
4 decltype(odd) *arrPtr(int i)
5 {
6     return (i % 2) ? &odd : &even;     // 返回一个指向数组的指针
7 }
```

arrPtr 使用关键字 decltype 表示它的返回类型是个指针，并且该指针所指的对象与 odd 类型一致。

6.4 函数重载

同一作用域内的几个函数名字相同但形参列表不同，称之为重载（overloaded）函数。编译器会根据传递的实参类型推断想要的是哪个函数。

定义重载函数

对于重载的函数来说，它们应该在形参数量或形参类型上有所不同。

不允许两个函数除了返回类型外其他所有的要素都相同。两个函数，它们的形参列表一样但是返回类型不同，则第二个函数的声明是错误的：

```
Record lookup(const Account&);
bool lookup(const Account&);    //错误：与上一个函数相比只有返回类型不同
```

判断两个形参的类型是否相异

重载和 const 形参

顶层 const 不影响传入函数的对象。一个拥有顶层 const 的形参无法和另一个没有顶层 const 的形参区分开来：

```
Record lookup(Phone);
Record lookup(const Phone);//重复声明了Record lookup(Phone)

Record lookup(Phone*);
Record lookup(Phone* const);//重复声明了Record lookup(Phone*)
```

另一方面，如果形参是某种类型的指针或引用，则通过区分其指向的是常量对象还是非常量对象可以实现函数重载，此时的 const 是底层的：

```
// 对于接受引用或指针的函数来说，对象是常量还是非常量对应的形参不同
// 定义了 4 个独立的重载函数
Record lookup(Account&);         // 函数作用于 Account的引用
Record lookup(const Account&);   // 新函数，作用于常量引用

Record lookup(Account*);         // 新函数，作用于指向Account的指针
Record lookup(const Account*);   // 新函数，作用于指向常量的指针

```

因为非常量可以转换成 const，所以上面四个函数都能作用于非常量对象或者指向非常量对象的指针。当我们传递一个非常量对象或者指向非常量对象的指针时，编译器会优先选用非常量版本的函数。

> 何时不应该重载函数？

> 函数重载可以在一定程度上减轻我们为函数起名字、记名字的负担，但是最好只重载那些确实非常相似的操作。

const_cast 和重载

const_cast 在重载函数的情景中最有用。6.3.2节的 shorterString 函数

```
// 挑出两个 string 对象中较短的那个，返回其引用
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size ? s1 : s2;
}
```

这个函数的参数和返回类型都是 const string 的引用。可以对两个非常量的string 实参调用这个函数，但返回的结果仍然是 const string 的引用。因此需要一种新的 shorterString 函数，当它的实参不是常量时，得到的结果是一个普通的引用，使用 const_cast 可以做到这一点：

```
string &shorterString(string &s1, string &s2)
{
    auto &r = shorterString(const_cast<const string&>(s1),
                            const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
```

这个函数中，首先将它的实参强制转换成对 const 的引用，然后调用了 shorterString 函数的 const 版本。const 版本返回对 const string 的引用，这个引用事实上绑定在了某个初始的非常量实参上。因此，可以将其转换回一个普通的 string&，这显然是安全的。

调用重载的函数

函数匹配（function matching）是指把函数调用与一组重载函数中的某一个关联起来的过程，也称重载确定（overload resolution）。

有些情况下选择函数比较困难，比如当两个重载函数参数数量相同而且参数类型可以相互转换时，6.6节介绍当函数调用存在类型转换时编译器处理的方法。

现在需要掌握，当调用重载函数时有三种可能得结果：

- 编译器找到一个与实参最佳匹配的函数，并生成调用该函数的代码
- 找不到任何一个函数与调用的实参匹配，此时编译器发出无匹配的错误信息

- 有多于一个函数可以匹配，但是每一个都不是明显的最佳选择。此时也将发生错误，称为二义性调用（ambiguous call）

6.4.1 重载与作用域

> 将函数声明置于局部作用域内不是一个明智的选择。

如果在内层作用域中声明名字，它将隐藏外层作用域中声明的同名实体。在不同的作用域中无法重载函数名：

```
string read();
void print(const string&);
void print(double);     // 重载 print 函数
void fooBar(int ival)
{
    bool read = false;  // 新作用域：隐藏了外层的read
    string s = read();  // 错误：read是一个布尔值，而非函数
    //不好的习惯：在局部作用域中声明函数不是一个好选择
    void print(int);    // 新作用域：隐藏了之前的print
    print("Value: ")    // 错误：print(const string &)被隐藏掉了
    print(ival);    //正确：当前 print(int) 可见
    print(3.14);    //正确：调用print(int);print(double)被隐藏了
}
```

假设把 print(int) 和其他 print 函数声明放在同一个作用域中，则它将称为另一种重载形式。

```
void print(const string &);
void print(double);
void print(int);
void fooBar2(int ival)
{
    print("Value: ");    // 调用print(const string&)
    print(ival);         // 调用print(int)
    print(3.14);         // 调用print(double)
}
```

6.5 特殊用途语言特性

三种函数相关的语言特性：默认实参、内联函数和 constexpr 函数，以及在程序调试过程中常用的一些功能。

6.5.1 默认实参

一旦某个形参被赋予了默认值，他后面的所有形参都必须有默认值。

使用默认实参调用函数

函数调用时实参按其位置解析，默认实参负责填补函数调用缺少的尾部实参（靠右侧位置）

当设计含有默认实参的函数时，其中一项任务是合理设置形参，尽量让不怎么使用默认值的形参出现在前面，而让那些经常使用默认值的形参出现在后面。

默认实参声明

默认实参初始值

6.5.2 内联函数和constexpr函数

6.3.2节编写了一个小函数，它的功能是比较两个string形参的长度并返回长度较小的string的引用。把这种规模较小的操作定义成函数有很多好处，主要包括：

- 阅读和理解 shorterString 函数的调用要比读懂等价的条件表达式容易得多
- 使用函数可以确保行为的统一，每次相关操作都能保证按照同样的方式进行
- 如果需要修改计算过程，显然修改函数要比先找到等价表达式所有出现的地方再逐一修改更容易
- 函数可以被其他应用重复利用，省去重新编写的代价

然而，使用 shorterString 函数也存在一个潜在的缺点：调用函数一般比求等价表达式的值要慢一些。大多数机器上，一次函数调用其实包含着一系列工作：调用前要先保存寄存器，并在返回时恢复；可能需要拷贝实参；程序转向一个新的位置继续执行。

内联函数可避免函数调用的开销

将函数指定为内联函数（inline），通常就是将它在每个调用点上 “内联地” 展开。假设把 shorteString 函数定义成内联函数，则如下调用

```
1  cout << shorterString(s1, s2) << endl;
```

将在编译过程中展开成类似于下面的形式

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

在 shorterString 函数的返回类型前面加上关键字 inline，这样就可以将它声明成内联函数：

```
// 内联版本：寻找两个 string 对象中较短的那个
inline const string &
shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

> 内联说明只是向编译器发出的一个请求，编译器可以选择忽略该请求。

一般来说，内联机制用于优化规模较小、流程直接、频繁调用的函数。很多编译器不支持内联递归函数，而且一个75行的函数也不大可能在调用点内联展开。

constexpr 函数

constexpr 函数（constexpr function）是指能用于常量表达式的函数。定义 constexpr 函数：函数返回类型及所有形参的类型都得是字面值类型，而且函数体中必须有且只有一条return语句：

把内敛函数和 constexpr 函数放在头文件内

内联函数和 constexpr 函数通常定义在头文件内。

6.5.3 调试帮助

类似头文件保护的技术，以便有选择地执行调试代码。基本思想是，程序可以包含一些用于调试的代码，但是这些代码只在开发程序时使用。当应用编写完成准备发布时，要先屏蔽掉调试代码。这种方法用到两种预处理功能：assert 和 NDEBUG。

assert 预处理宏

预处理宏是一个预处理变量，它的行为类似于内联函数。assert 宏使用一个表达式作为它的条件：

```
assert(expr);
```

对 expr 求值，如果表达式为false（即0），assert 输出信息并终止程序的执行。如果表达式为真（即非0），assert 什么也不做。

assert 宏定义在 cassert 头文件中。预处理器名字由预处理器而非编译器管理，因此可以直接使用预处理名字而无须提供using声明。

NDEBUG 预处理变量

assert 的行为依赖于一个名为 NDEBUG 的预处理变量的状态。如果定义了 NDEBUG，则 assert 什么也不做。默认状态下没有定义 NDEBUG，此时 assert 将执行运行时检查。

可以使用一个 #define 语句定义 NDEBUG，从而关闭调试状态。同时，很多编译器提供了一个命令行选项可以定义预处理变量：

```
$ CC -D NDEBUG main.c
```

这条命令等价于在 main.c 文件的一开始写 #define NDEBUG

定义 NDEBUG 能避免检查各种条件所需的运行时开销，当然此时根本不会执行运行时检查。因此，assert 应该仅用于验证那些确实不可能发生的事情。可以把 assert 当成调试程序的一种辅助手段，但是不能用它替代真正的运行时逻辑检查，也不能替代程序本身应该包含的错误检查。

除了使用 assert 外，还可以使用 NDEBUG 编写自己的条件调试代码。如果 NDEBUG 未定义，将执行 #ifndef 和 #endif 之间的代码：如果定义了 NDEBUG，这些代码将被忽略掉。

```
void print(const int ia[], size_t size)
{
#ifndef NDEBUG
    // _ _func_ _ 是编译器定义的一个局部静态变量，用于存放函数的名字
    cerr << _ _ func _ _ << ": array size is " << size << endl;
#endif
// ...
```

除了C++编译器定义的 _ _func_ _ 之外，预处理器还定义了另外 4 个对于程序调试很有用的名字：

_ _FILE_ _ 存放文件名的字符串字面值

_ _LINE_ _ 存放当前行号的整型字面值

_ _TIME_ _ 存放文件编译时间的字符串字面值

_ _DATE_ _ 存放文件编译日期的字符串字面值

6.6 函数匹配

确定候选函数和可行函数

寻找最佳匹配

含有多个形参的函数匹配

6.6.1 实参类型转换

需要类型提升和算术类型转换的匹配

函数匹配和 const 实参

6.7 函数指针

使用函数指针

重载函数的指针

函数指针形参

返回指向函数的指针

将 auto 和 decltype 用于函数指针类型

小结

术语表

7. 类

C++中，使用类定义自己的数据类型。通过定义新的类型来反映待解决问题中的各种概念，可以使我们更容易编写、调试和修改程序。

本章主要关注数据抽象的重要性。数据抽象能帮助我们将对象的具体实现与对象所能执行的操作分离开来。13章讨论如何控制对象拷贝、移动和销毁等行为，14章将学习如何自定义运算符。

类的基本思想是数据抽象（data abstraction）和封装（encapsulation）。数据抽象是一种依赖于接口（interface）和实现（implementation）分离的编程（以及设计）技术。类的接口包括用户所能执行的操作；类的实现则包括类的数据成员、负责接口实现的函数体以及定义类所需的各种私有函数。

封装实现了类的接口和实现的分离。封装后的类隐藏了实现细节，类的用户只能使用接口而无法访问实现部分。

类想要实现数据抽象和封装，需要先定义一个抽象数据类型（abstract data type）。在抽象数据类型中，由类的设计者负责考虑类的实现过程；使用该类的程序员只需要抽象思考类型做了什么，而无须了解类型的工作细节。

7.1 定义抽象数据类型

7.1.1 设计 Sales_data 类

7.1.2 定义改进的 Sales_data 类

成员函数必须声明在类的内部，它的定义既可以在类的内部也可以在类的外部。作为接口组成部分的非成员函数，例如 add、read 和 print 等，它们的定义和声明都在类的外部。

```
struct Sales_data {
      // newmembers: operations on Sales_data objects
      std::string isbn() const { return bookNo; }
      Sales_data& combine(const Sales_data&);
      double avg_price() const;
     // data members are unchanged from § 2.6.1
      std::string bookNo;
      unsigned units_sold = 0;
      double revenue = 0.0;
};
// nonmemberSales_datainterface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
```

> 定义在类内部的函数是隐式的 inline 函数。

定义成员函数

引入 this

this 总是指向这个对象，this 是一个常量指针 。

引入 const 成员函数

this 的类型是 Sales_data *const。C++ 允许把 cosnt 关键字放在成员函数的参数列表之后。此时，紧跟在参数列表后面的 const 表示 this 是一个指向常量的指针。像这样使用 const 的成员函数被称作常量成员函数（const member function）。

isbn 的函数体等同下面的形式：

```
1 // 伪代码，说明隐式的this指针是如何使用的
2 // 此处的this 是一个指向常量的指针，因为isbn是一个常量成员
3 std::string Sales_data::isbn(const Sales_data *const this)
4 { return this->isbn; }
```

this 是指向常量的指针，所以常量成员函数不能改变调用它的对象的内容。

> 常量对象，以及常量对象的引用或指针都只能调用常量成员函数

梳理c++ const 修饰函数

类作用域和成员函数

编译器分两步处理类：首先编译成员的声明然后才轮到成员函数体。因此，成员函数体可以随意使用类中的其他成员而无需在意这些成员出现的次序。

在类的外部定义成员函数

定义一个返回 this 对象的函数

7.1.3 定义类相关的非成员函数

> 如果非成员函数是类接口的组成部分，则这些函数的声明应该与类在同一个头文件内。

定义 read 和 print 函数

定义 add 函数

7.1.4 构造函数

合成的默认构造函数

某些类不能依赖于合成的默认构造函数

> 只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函数

定义 Sales_data 的构造函数

= default 的含义

构造函数初始值列表

在类的外部定义构造函数

7.1.5 拷贝、赋值和析构

某些类不能依赖于合成的版本

7.2 访问控制与封装

使用 class 或 struct 关键字 定义类唯一的区别就是默认的访问权限。

出于统一风格的考虑，定义的类的所有成员是 public 的时，使用 struct；反之，希望成员是 private 的，使用 class。

7.2.1 友元

类可以允许其他类或者函数访问它的非公有成员，方法是令其它类或者函数成为它的友元（friend）。

7.3 类的其它特性

7.3.1 类成员再探

定义一个类型成员

Screen 表示显示器中的一个窗口。每个 Screen 包含一个用于保存 Screen 内容的 string 成员和三个 string::size_type 类型的成员，它们分别表示光标的位置以及屏幕的高和宽。

```
1 class Screen {
```

```
public:
    typedef std::string::size_type pos;
    // 等同于
    // using pos = std::string::size_type
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

用于定义类型的成员必须先定义后使用。

Screen 类的成员函数

添加一个构造函数令用户可以定义屏幕的尺寸和内容，以及两个其他成员，分别负责移动光标和读取给定位置的字符：

```
class Screen {
public:
    typedef std::stirng::size_type pos;
    Screen() = default;    //已经有构造函数，需要默认函数要显示声明
    Screen(pos ht, pos wd, char c): height(ht), width(wd),
    contents(ht * wd, c) {}
    char get() const       //读取光标处字符
        { return contents[cursor]; }    //隐式内联
    inline char get(pos ht, pos wd) const;    //显式内联
    Screen &move(pos r, pos c);    //能在之后被设为内联
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

令成员作为内联函数

定义在类内部的成员函数是自动 inline 的。

可以在类的内部把inline作为声明的一部分显式地声明成员函数，也能在类的外部用 inline 修饰函数的定义。

重载成员函数

可变数据成员

有时希望修改类的某个数据成员，即使在一个 const 成员函数内。变量声明加入 mutable 关键字。

一个可变数据成员（mutable data member）永远不会是 const。

类数据成员的初始值

窗口管理类 Window_mgr 类总是拥有一个默认初始化的 Screen。C++11 就是把这个默认值声明成一个类内初始值

```
class Window_mgr {
private:
    std::vector<Screen> screens{Screen(24, 80, ' '}
}
```

> 提供一个类内初始值时，必须以符号 = 或者花括号表示

7.3.2 返回 *this 的成员函数

继续添加一些函数，它们负责设置光标所在位置的字符或者其他任一给定位置的字符：

```
class Screen {
public:
    Screen &set(char);
    Screen &set(pos, pos, char);
};
inline Screen &Screen::set(char c)
{
    contents[cursor] = c;     // 设置当前光标所在位置的新值
    return *this;             // 将 this 对象作为左值返回
}
```

和 move 一样，set 成员返回值是调用 set 的对象的引用。返回引用的函数是左值的，意味着函数返回的是对象本身而非对象的副本。

```
// 把光标移动到一个指定的位置，然后设置该位置的字符值
myScreen.move(4,0).set('#');
```

如果定义的返回类型不是引用，则 move 的返回值将是 *this 的副本，调用set只能改变临时副本，不能改变 myScreen 的值。

从 const 成员函数返回 *this

> 一个 const 成员函数如果以引用的形式返回 *this，那么它的返回类型将是常量引用

基于 const 的重载

> 建议：对于公共代码使用私有功能函数

7.3.3 类类型

类的声明

对于一个类，创建它的对象之前该类必须被定义过，而不能仅仅被声明。否则，编译器无法了解这样的对象需要多少存储空间。类似，类必须被定义过，然后才能用引用或者指针访问其成员。

一个类的名字出现后，它就被认为是声明过了（但尚未定义），因此类允许包含指向它自身类型的引用或指针。

```
class Link_screen {
    Screen window;
    Link_screen *next;
    Link_screen *prev;
};
```

7.3.4 友元再探

友元函数能定义在类的内部，这样的函数是隐式内联的。

类之间的友元关系

Window_mgr 类的某些成员可能需要访问它管理的 Screen 类的内部数据。Screen 需要把 Window_mgr 指定成它的友元：

```
class Screen {
    // Window_mgr 的成员可以访问 Screen 类的私有部分
    friend class Window_mgr;
```

```
};
```

友元关系不存在传递性。每个类负责控制自己的友元类或友元函数。

令成员函数作为友元

Screen 只为 Window_mgr::clear 提供访问权限

```
class Screen {
    friend void Window_mgr::clear(ScreenIndex);
};
```

函数重载和友元

友元声明和作用域

7.4 类的作用域

作用域和定义在类外部的成员

在类的外部，成员的名字被隐藏起来了。

一旦遇到了类名，定义的剩余部分（包括参数列表和函数体）就在类的作用于之内了，我们可以直接使用类的其他成员而无须再授权了。

```
Window_mgr::ScreenIndex
Window_mgr::addScreen(cosnt Screen &s)
{
    screens.push_back(s);
    return screens.size()-1;
}
```

返回类型出现在类名之前，它是位于 Window_mgr 类的作用域之外的。想要使用 ScreenIndex 需要明确指定哪个类定义了它。

7.4.1 名字查找与类的作用域

> 编译器处理完类中的全部声明后才会处理成员函数的定义

用于类成员声明的名字查找

类型名要特殊处理

> 类型名的定义通常出现在类的开始处，这样就能确保所有使用该类型的成员都出现在类名的定义之后。

成员定义中的普通块作用域的名字查找

类作用域之后，在外围的作用域中查找

在文件中名字的出现处对其进行解析

7.5 构造函数再探

7.5.1 构造函数初始值列表

构造函数的初始值有时必不可少

有时可以忽略数据成员初始化和赋值之间的差异。但是成员是 const 或者引用的话，必须将其初始化。

```
class  ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

和其他常量对象或者引用一样，成员 ci 和 ri 都必须被初始化。

随着构造函数体一开始执行，初始化就完成了。初始化 const 会饿引用类型的数据成员的唯一机会就是通过构造函数初始值。

```
// 显式初始化引用和 const 成员
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) {}
```

> 如果成员是 const、引用，或者属于某种未提供默认构造函数的类类型，我们必须通过构造函数初始值列表为这些成员提供初值。

> **建议：使用构造函数初始值**
>
> 很多类中，初始化和赋值区别事关底层效率问题：前者直接初始化数据成员，后者则先初始化再赋值。
>
> 除了效率问题外更重要的是，一些数据成员必须被初始化。建议使用构造函数初始值，能避免某些意想不到的编译错误，特别是遇到有的类含有需要构造函数初始值的成员时。

成员初始化的顺序

构造函数初始值列表只说明用于初始化成员的值，而不限定初始化的具体执行顺序。

成员初始化的顺序与它们在类定义中的出现顺序一致：第一个成员先被初始化，然后第二个，以此类推。构造函数初始值列表中初始值的前后位置关系不会影响实际的初始化顺序。

> 最好令构造函数初始值的顺序与成员声明的顺序保持一致。尽量避免使用某些成员初始化其他成员。

默认实参和构造函数

> 如果一个构造函数为所有参数都提供了默认实参，则它实际上也定义了默认构造函数。

7.5.2 委托构造函数

委托构造函数（delegating constructor）使用它所属类的其他构造函数执行它自己的初始化过程，或者说它把它自己的一些职责委托给了其他构造函数。

7.5.3 默认构造函数的作用

如果定义了其他构造函数，最好也提供一个默认构造函数

使用默认构造函数

7.5.4 隐式的类类型转换

只允许一步类类型转换

类类型转换不是总有效

抑制构造函数定义的隐式转换

explicit 构造函数只能用于直接初始化

为转换显式地使用构造函数

7.5.5 聚合类

7.5.6 字面值常量类

constexpr 构造函数

7.6 类的静态成员

有时候类需要它的一些成员与类本身直接相关，而不是与类的各个对象保持关联。

声明静态成员

类静态成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据。类似，静态成员函数也不与任何对象绑定在一起，它们不包含this 指针。静态成员函数不能声明成 const 的，而且也不能在 static 函数体内使用 this 指针。

使用类的静态成员

定义静态成员

和其他成员函数一样，既可以在类的内部也可以在类的外部定义静态成员函数。当在类的外部定义静态成员时，不能重复 static 关键字，该关键字只出现在类内部的声明语句。

> 和类的所有成员一样，当我们指向类外部的静态成员时，必须指明成员所属的类名。static 关键字则只出现在类内部的声明语句中。

静态成员的类内初始化

> 即使一个常量静态数据成员在类内部被初始化了，通常也应该在类的外部定义一下该成员。

静态成员能用于某些场景，而普通成员不能

静态成员独立于任何对象。静态数据成员可以是不完全类型，它的类型可以就是它所属的类类型。非静态数据成员只能声明它所属类的指针或引用。

静态成员和普通成员的另一个区别是我们可以使用静态成员作为默认实参。

小结

术语