## 第一章 快速入门

### 1.2 初窥输入/输出

#include <iostream>

int main()

{

std::cout << "Enter two numbers:" << std::endl;

int v1, v2;

std::cin >> v1 >> v2;

std::cout << "The sum of " << v1 << " and " << v2

<< " is " << v1 + v2 << std::endl;

return 0;

}

细心的读者会注意到这个程序中使用的是 std::cout 和 std::endl，而不是 cout 和 endl。前缀 std:: 表明 cout 和 endl 是定义在[命名空间](http://manual.51yip.com/c++/0201721481/ch01lev1sec8.html" \l "gloss01_31) [std](http://manual.51yip.com/c++/0201721481/ch01lev1sec8.html" \l "gloss01_42) 中的。使用命名空间程序员可以避免与库中定义的名字相同而引起无意冲突。因为标准库定义的名字是定义在命名空间中，所以我们可以按自己的意图使用相同的名字。

标准库使用命名空间的副作用是，当我们使用标准库中的名字时，必须显式地表达出使用的是命名空间 std 下的名字。std::cout 的写法使用了作用域操作符（[scope operator](http://manual.51yip.com/c++/0201721481/ch01lev1sec8.html" \l "gloss01_51)，:: 操作符），表示使用的是定义在命名空间 std 中的 cout。我们将在[第 3.1 节](http://manual.51yip.com/c++/0201721481/ch03lev1sec1.html" \l "ch03lev1sec1)学习到程序中经常使用的避免这种冗长句法的方法。

endl 是一个特殊值，称为[操纵符](http://manual.51yip.com/c++/0201721481/ch01lev1sec8.html" \l "gloss01_28)，将它写入输出流时，具有输出换行的效果，并刷新与设备相关联的*[缓冲区](http://manual.51yip.com/c++/0201721481/ch01lev1sec8.html" \l "gloss01_03)*。通过刷新缓冲区，用户可立即看到写入到流中的输出。

### 1.4. 控制结构

**声明错误。C++ 程序中使用的每个名字必须在使用之前声明。没有声明名字通常会导致错误信息。最常见的两种声明错误，是从标准库中访问名字时忘记使用“**std::**”，以及由于疏忽而拼错标识符名:**

#include <iostream>

int main()

{

int v1, v2;

std::cin >> v >> v2; // *error: uses "* v *"not"* v1*"*

// cout *not defined, should be* std::cout

cout << v1 + v2 << std::endl;

return 0;

}

* **1.4.4. 读入未知数目的输入**

对[第 1.4.1 节](http://manual.51yip.com/c++/0201721481/ch01lev1sec4.html" \l "ch01lev2sec5)的求和程序稍作改变，还可以允许用户指定一组数求和。这种情况下，我们不知道要对多少个数求和，而是要一直读数直到程序输入结束。输入结束时，程序将总和写到标准输出：

#include <iostream>

int main()

{

int sum = 0, value;

// *read till end-of-file, calculating a running total of all values read*

while (std::cin >> value)

sum += value; // *equivalent to* *sum = sum + value*

std::cout << "Sum is: " << sum << std::endl;

return 0;

}

当我们使用 istream 对象作为条件，结果是测试流的状态。如果流是有效的（也就是说，如果读入下一个输入是可能的）那么测试成功。遇到*[文件结束符](http://manual.51yip.com/c++/0201721481/ch01lev1sec8.html" \l "gloss01_16)*或遇到无效输入时，如读取了一个不是整数的值，则 istream 对象是无效的。处于无效状态的 istream 对象将导致条件失败。

**操作系统使用不同的值作为文件结束符。Windows 系统下我们通过键入 control—z——同时键入“ctrl”键和“z”键，来输入文件结束符。Unix 系统中，包括 Mac OS—X 机器，通常用 control—d。**

* **第一部分 基本语言**

和大多数程序设计语言一样，C++ 中对象的类型决定了该对象可以执行的操作。语句正确与否取决于该语句中对象的类型。一些程序设计语言，特别是 Smalltalk 和 Python，在[运行时](http://manual.51yip.com/c++/0201721481/ch02lev1sec11.html" \l "gloss02_41)才检查语句中对象的类型。相反，C++ 是静态类型（statically typed）语言，在编译时执行类型检查。结果是程序中使用某个名字之前，必须先告知编译器该名字的类型。

## 第二章 变量和基本类型

### 2.1. 基本内置类型

#### 2.1.1. 整型

字符类型有两种：char 和 wchar\_t。char 类型保证了有足够的空间，能够存储机器基本字符集中任何字符相应的数值，因此，char 类型通常是单个机器字节（byte）。wchar\_t 类型用于扩展字符集，比如汉字和日语，这些字符集中的一些字符不能用单个 char 表示。

整型 int、short 和 long 都默认为带符号型。要获得无符号型则必须指定该类型为 unsigned，比如 unsigned long。unsigned int 类型可以简写为 unsigned，也就是说，unsigned 后不加其他类型说明符意味着是 unsigned int 。

对于 unsigned 类型来说，编译器必须调整越界值使其满足要求。编译器会将该值对 unsigned 类型的可能取值数目求模，然后取所得值。比如 8 位的 unsigned char，其取值范围从 0 到 255（包括 255）。如果赋给超出这个范围的值，那么编译器将会取该值对 256 求模后的值。例如，如果试图将 336 存储到 8 位的 unsigned char 中，则实际赋值为 80，因为 80 是 336 对 256 求模后的值。

C++ 中，把负值赋给 unsigned 对象是完全合法的，其结果是该负数对该类型的取值个数求模后的值。所以，如果把 -1 赋给8位的 unsigned char，那么结果是 255，因为 255 是 -1 对 256 求模后的值。（计算机中负数是以补码形式存储的，-1的补码11111111，转换成无符号数即是255的二进制编码。）

**2.2. 字面值常量**

为了兼容 C 语言，C++ 中所有的字符串字面值都由编译器自动在末尾添加一个空字符。字符字面值

'A' 表示单个字符 A，然而

"A" 表示包含字母 A 和空字符两个字符的字符串。

### 2.3. 变量

**2.3.3. 定义对象**

C++ 支持两种[初始化变量](http://manual.51yip.com/c++/0201721481/ch02lev1sec11.html" \l "gloss02_60)的形式：**[复制初始化](http://manual.51yip.com/c++/0201721481/ch02lev1sec11.html" \l "gloss02_12)**和**[直接初始化](http://manual.51yip.com/c++/0201721481/ch02lev1sec11.html" \l "gloss02_17)**。复制初始化语法用等号（=），直接初始化则是把初始化式放在括号中：

int ival(1024); // *direct-initialization*

int ival = 1024; // *copy-initialization*

**2.3.4. 变量初始化规则**

如果定义某个类的变量时没有提供初始化式，这个类也可以定义初始化时的操作。它是通过定义一个特殊的构造函数即**[默认构造函数](http://manual.51yip.com/c++/0201721481/ch02lev1sec11.html" \l "gloss02_15)**来实现的。这个构造函数之所以被称作默认构造函数，是因为它是“默认”运行的。如果没有提供初始化式，那么就会使用默认构造函数。不管变量在哪里定义，默认构造函数都会被使用。

大多数类都提供了默认构造函数。如果类具有默认构造函数，那么就可以在定义该类的变量时不用显式地初始化变量。例如，string 类定义了默认构造函数来初始化 string 变量为空字符串，即没有字符的字符串：

std::string empty; // *empty* *is the empty string;* *empty =""*

有些类类型没有默认构造函数。对于这些类型来说，每个定义都必须提供显式的初始化式。没有初始值是根本不可能定义这种类型的变量的。

**2.3.5. 声明和定义**

extern 声明不是定义，也*不*分配存储空间。事实上，它只是说明变量定义在程序的其他地方。程序中变量可以声明多次，但只能定义一次。

只有当声明也是定义时，声明才可以有初始化式，因为只有定义才分配存储空间。初始化式必须要有存储空间来进行初始化。如果声明有初始化式，那么它可被当作是定义，即使声明标记为 extern：

extern double pi = 3.1416; // *definition*

虽然使用了 extern ，但是这条语句还是定义了 pi，分配并初始化了存储空间。只有当 extern 声明位于函数外部时，才可以含有初始化式。

**2.4. const 限定符**

定义 bufSize 为常量并初始化为 512。变量 bufSize 仍然是一个左值（[第 2.3.1 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec3.html" \l "ch02lev2sec10)），但是现在这个左值是不可修改的。任何修改 bufSize 的尝试都会导致编译错误：

bufSize = 0; // *error: attempt to write to const object*

因为常量在定义后就不能被修改，所以定义时必须初始化：

#### const 对象默认为文件的局部变量

非 const 变量默认为 extern。要使 const 变量能够在其他的文件中访问，必须地指定它为 extern。

与其他变量不同，除非特别说明，在全局作用域声明的 const 变量是定义该对象的文件的局部变量。此变量只存在于那个文件中，不能被其他文件访问。

通过指定 const 变更为 extern，就可以在整个程序中访问 const 对象：

// *file\_1.cc*

// *defines and initializes a* *const* *that is accessible to other files*

extern const int bufSize = fcn();

// *file\_2.cc*

extern const int bufSize; // *uses* *bufSize* *from* *file\_1*

// *uses* *bufSize* *defined in* *file\_1*

for (int index = 0; index != bufSize; ++index)

// ...

**2.5. 引用**

当引用初始化后，只要该引用存在，它就保持绑定到初始化时指向的对象。不可能将引用绑定到另一个对象。

#### const 引用

const int ival = 1024;

const int &refVal = ival; // *ok: both reference and object are* *const*

int &ref2 = ival; // *error: non* *const* *reference to a* *const* *object*

可以读取但不能修改 refVal ，因此，任何对 refVal 的赋值都是不合法的。这个限制有其意义：不能直接对 ival 赋值，因此不能通过使用 refVal 来修改 ival。

同理，用 ival 初始化 ref2 也是不合法的：ref2 是普通的**[非 const 引用](http://manual.51yip.com/c++/0201721481/ch02lev1sec11.html" \l "gloss02_34)**，因此可以用来修改 ref2 指向的对象的值。通过 ref2 对 ival 赋值会导致修改 const 对象的值。为阻止这样的修改，需要规定将普通的引用绑定到 const 对象是不合法的。

const 引用可以初始化为不同类型的对象或者初始化为右值（[第 2.3.1 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec3.html" \l "ch02lev2sec10)），如字面值常量：

int i = 42;

// *legal for* *const* *references only*

const int &r = 42;

const int &r2 = r + i;

同样的初始化对于非 const 引用却是不合法的，而且会导致编译时错误。其原因非常微妙，值得解释一下。

int &refVal2; // *error: a reference must be initialized*

int &refVal3 = 10; // *error: initializer must be an object*

观察将引用绑定到不同的类型时所发生的事情，最容易理解上述行为。假如我们编写

double dval = 3.14;

const int &ri = dval;

编译器会把这些代码转换成如以下形式的编码：

int temp = dval; // *create temporary* *int* *from the* *double*

const int &ri = temp; // *bind* *ri* *to that temporary*

如果 ri 不是 const，那么可以给 ri 赋一新值。这样做不会修改 dval，而是修改了 temp。期望对 ri 的赋值会修改 dval 的程序员会发现 dval 并没有被修改。仅允许 const 引用绑定到需要临时使用的值完全避免了这个问题，因为 const 引用是只读的。

非 const 引用只能绑定到与该引用同类型的对象。

const 引用则可以绑定到不同但相关的类型的对象或绑定到右值。（const引用类型可以和它绑定的const对象/值不同）

### 2.7. 枚举

#### 定义和初始化枚举

枚举的定义包括关键字 enum，其后是一个可选的枚举类型名，和一个用花括号括起来、用逗号分开的**[枚举成员](http://manual.51yip.com/c++/0201721481/ch02lev1sec11.html" \l "gloss02_19)**列表。

enum Points { point2d, point2w, point3d, point3w };

默认地，第一个枚举成员赋值为 0，后面的每个枚举成员赋的值比前面的大 1。

枚举类型的对象的初始化或赋值，只能通过其枚举成员或同一枚举类型的其他对象来进行：

Points pt3d = point3d; // *ok:* *point3d* *is a* *Points* *enumerator*

Points pt2w = 3; // *error:* *pt2w* *initialized with* *int*

pt2w = polygon; // *error:* *polygon* *is not a* *Points* *enumerator*

pt2w = pt3d; // *ok: both are objects of* *Points enum* *type*

**2.8. 类类型**

**类的数据成员**

定义变量和定义数据成员存在非常重要的区别：一般不能把类成员的初始化作为其定义的一部分。当定义数据成员时，只能指定该数据成员的名字和类型。类不是在类定义里定义数据成员时初始化数据成员，而是通过称为构造函数（[第 2.3.3 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec3.html" \l "ch02lev2sec12)）的特殊成员函数控制初始化。我们将在[第 7.7.3 节](http://manual.51yip.com/c++/0201721481/ch07lev1sec7.html" \l "ch07lev2sec24)定义 Sales\_item 的构造函数。

**使用 struct 关键字**

如果使用 class 关键字来定义类，那么定义在第一个访问标号前的任何成员都隐式指定为 private；如果使用 struct 关键字，那么这些成员都是 public。使用 class 还是 struct 关键字来定义类，仅仅影响默认的初始访问级别。

**2.9. 编写自己的头文件**

#### 2.9.1. 设计自己的头文件

**一些 const 对象定义在头文件中**

回想一下，const 变量（[第 2.4 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec4.html" \l "ch02lev1sec4)）默认时是定义该变量的文件的局部变量。正如我们现在所看到的，这样设置默认情况的原因在于允许 const 变量定义在头文件中。

在 C++ 中，有些地方需要放置常量表达式（[第 2.7 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec7.html" \l "ch02lev1sec7)）。

一般来说，常量表达式是编译器在编译时就能够计算出结果的表达式。当 const 整型变量通过常量表达式自我初始化时，这个 const 整型变量就可能是常量表达式。而 const 变量要成为常量表达式，初始化式必须为编译器可见。为了能够让多个文件使用相同的常量值，const 变量和它的初始化式必须是每个文件都可见的。而要使初始化式可见，一般都把这样的 const 变量定义在头文件中。那样的话，无论该 const 变量何时使用，编译器都能够看见其初始化式。

但是，C++ 中的任何变量都只能定义一次（[第 2.3.5 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec3.html" \l "ch02lev2sec14)）。定义会分配存储空间，而所有对该变量的使用都关联到同一存储空间。因为 const 对象默认为定义它的文件的局部变量，所以把它们的定义放在头文件中是合法的。

这种行为有一个很重要的含义：当我们在头文件中定义了 const 变量后，每个包含该头文件的源文件都有了自己的 const 变量，其名称和值都一样。

当该 const 变量是用常量表达式初始化时，可以保证所有的变量都有相同的值。但是在实践中，大部分的编译器在编译时都会用相应的常量表达式替换这些 const 变量的任何使用。所以，在实践中不会有任何存储空间用于存储用常量表达式初始化的 const 变量。

如果 const 变量不是用常量表达式初始化，那么它就不应该在头文件中定义。相反，和其他的变量一样，该 const 变量应该在一个源文件中定义并初始化。应在头文件中为它添加 extern 声明，以使其能被多个文件共享。

#### 2.9.2. 预处理器的简单介绍

##### 避免多重包含

在编写头文件之前，我们需要引入一些额外的预处理器设施。预处理器允许我们自定义变量。

为了避免名字冲突，预处理器变量经常用全大写字母表示。

预处理器变量有两种状态：已定义或未定义。定义预处理器变量和检测其状态所用的预处理器指示不同。#define 指示接受一个名字并定义该名字为预处理器变量。#ifndef 指示检测指定的预处理器变量是否未定义。如果预处理器变量未定义，那么跟在其后的所有指示都被处理，直到出现 #endif。

可以使用这些设施来预防多次包含同一头文件：

#ifndef SALESITEM\_H

#define SALESITEM\_H

// *Definition of* Sales\_itemclass *and related functions goes here*

#endif

## 第三章 标准库类型

**3.1. 命名空间的 using 声明**

一个 using 声明一次只能作用于一个命名空间成员。using 声明可用来明确指定在程序中用到的命名空间中的名字，如果希望使用 std（或其他的命名空间）中的几个名字，则必须为要用到的每个名字都提供一个 using 声明。例如，利用 using 声明可以这样重新编写[第 1.2.2 节](http://manual.51yip.com/c++/0201721481/ch01lev1sec2.html)中的加法程序：

#include <iostream>

// *using declarations for names from the standard library*

using std::cin;

using std::cout;

using std::endl;

int main()

{

cout << "Enter two numbers:" << endl;

int v1, v2;

cin >> v1 >> v2;

cout << "The sum of " << v1

<< " and " << v2

<< " is " << v1 + v2 << endl;

return 0;

}

**使用标准库类型的类定义(?)**

有一种情况下，必须*总是*使用完全限定的标准库名字：在头文件中。理由是头文件的内容会被预处理器复制到程序中。用 #include 包含文件时，相当于头文件中的文本将成为我们编写的文件的一部分。如果在头文件中放置 using 声明，就相当于在包含该头文件 using 的每个程序中都放置了同一 *using*，不论该程序是否需要 using 声明。

通常，头文件中应该只定义确实必要的东西。请养成这个好习惯。

**3.2. 标准库**

**3.2.2. string 对象的读写**

cin >> s; // *read whitespace-separated string into* *s*

从标准输入读取 string 并将读入的串存储在 s 中。string 类型的输入操作符：

* 读取并忽略开头所有的空白字符（如空格，换行符，制表符）。
* 读取字符直至再次遇到空白字符，读取终止。

#### 3.2.3. string 对象的操作

s.size() 返回 s 中字符的个数 string 类型

任何存储 string 的 size 操作结果的变量必须为 string::size\_type 类型。特别重要的是，还要把 size 的返回值赋给一个 int 变量。

**和字符串字面值的连接**

当进行 string 对象和字符串字面值混合连接操作时，+ 操作符的左右操作数必须至少有一个是 string 类型的：

string s3 = s1 + ", "; // *ok: adding a* *string* *and a literal*

string s4 = "hello" + ", "; // *error: no* *string* *operand*

s3 和 s4 的初始化只用了一个单独的操作。在这些例子中，很容易判断 s3 的初始化是合法的：把一个 string 对象和一个字符串字面值连接起来。而 s4 的初始化试图将两个字符串字面值相加，因此是非法的。

**从 string 对象获取字符**

string 类型通过下标操作符（[ ]）来访问 string 对象中的单个字符。下标操作符需要取一个 size\_type 类型的值，来标明要访问字符的位置。这个下标中的值通常被称为“下标”或“索引”**（****[index](http://manual.51yip.com/c++/0201721481/ch03lev1sec7.html" \l "gloss03_10)）**

**建议：采用 C 标准库头文件的 C++ 版本**

**C++ 标准库除了定义了一些选定于 C++ 的设施外，还包括 C 标准库。C++ 中的头文件**cctype**其实就是利用了 C 标准库函数，这些库函数就定义在 C 标准库的**ctype.h**头文件中。**

C 标准库头文件命名形式为 ***name*** 而 C++ 版本则命名为 c*name* ，少了后缀，.h**而在头文件名前加了 c 表示这个头文件源自**C**标准库。因此，**cctype**与**ctype.h**文件的内容是一样的，只是采用了更适合**C++**程序的形式。特别地，c*name* 头文件中定义的名字都定义在命名空间**std**内，而**.h**版本中的名字却不是这样。**

**通常，C++ 程序中应采用**c***name* 这种头文件的版本，而不采用**name.h**版本，这样，标准库中的名字在命名空间**std**中保持一致。使用**.h**版本会给程序员带来负担，因为他们必须记得哪些标准库名字是从**C**继承来的，而哪些是**C++**所特有的。**

**3.3. 标准库 vector 类型**

**vector 的下标操作**

和 string 类型的下标操作符一样，vector 下标操作的结果为左值，因此可以像循环体中所做的那样实现写入。另外，和 string 对象的下标操作类似，这里用 size\_type 类型作为 vector 下标的类型。

**下标操作不添加元素**

初学 C++ 的程序员可能会认为 vector 的下标操作可以添加元素，其实不然：

vector<int> ivec; // *empty* *vector*

for (vector<int>::size\_type ix = 0; ix != 10; ++ix)

ivec[ix] = ix; // *disaster:* *ivec* *has no elements*

上述程序试图在 ivec 中插入 10 个新元素，元素值依次为 0 到 9 的整数。但是，这里 ivec 是空的 vector 对象，而且下标只能用于获取已存在的元素。

这个循环的正确写法应该是：

for (vector<int>::size\_type ix = 0; ix != 10; ++ix)

ivec.push\_back(ix); // *ok: adds new element with value* *ix*

**3.4. 迭代器简介**

**begin 和 end 操作**

由 end 操作返回的迭代器指向 vector 的“末端元素的下一个”。**[“超出末端迭代器”（off-the-end iterator）](http://manual.51yip.com/c++/0201721481/ch03lev1sec7.html" \l "gloss03_14)**。表明它指向了一个不存在的元素。如果 vector 为空，begin 返回的迭代器与 end 返回的迭代器相同。

由 end 操作返回的迭代器并不指向 vector 中任何实际的元素，相反，它只是起一个**[哨兵（sentinel）](http://manual.51yip.com/c++/0201721481/ch03lev1sec7.html" \l "gloss03_16)**的作用，表示我们已处理完 vector 中所有元素。

由于 end 操作返回的迭代器不指向任何元素，因此不能对它进行解引用或自增操作。

**3.5. 标准库 bitset**

**用 string 对象初始化 bitset 对象**

当用 string 对象初始化 bitset 对象时，string 对象直接表示为位模式。从 string 对象读入位集的顺序是*从右向左（from right to left）*：

string strval("1100");

bitset<32> bitvec4(strval);

bitvec4 的位模式中第 2 和 3 的位置为 1，其余位置都为 0。如果 string 对象的字符个数小于 bitset 类型的长度，则高阶位置为 0。

string 对象和 bitsets 对象之间是反向转化的：string 对象的最右边字符（即下标最大的那个字符）用来初始化 bitset 对象的低阶位（即下标为 0 的位）。当用 string 对象初始化 bitset 对象时，记住这一差别很重要。

**第四章 数组和指针**

现代 C++ 程序应尽量使用 vector 和迭代器类型，而避免使用低级的数组和指针。设计良好的程序只有在强调速度时才在类实现的内部使用数组和指针。

### 4.1. 数组

#### 4.1.1. 数组的定义和初始化

数组的维数必须用值大于等于1的常量表达式定义（[第 2.7 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec7.html" \l "ch02lev1sec7)）。此常量表达式*只能*包含整型字面值常量、枚举常量（[第 2.7 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec7.html" \l "ch02lev1sec7)）或者用常量表达式初始化的整型 const 对象。非 const 变量以及要到运行阶段才知道其值的 const 变量都不能用于定义数组的维数。

const unsigned buf\_size = 512, max\_files = 20;

int staff\_size = 27; // *nonconst*

const unsigned sz = get\_size(); // *const* *value not known until run time*

char input\_buffer[buf\_size]; // *ok:* *const* *variable*

string fileTable[max\_files + 1]; // *ok: constant expression*

double salaries[staff\_size]; // *error: non* *const* *variable*

int test\_scores[get\_size()]; // *error: non* *const* *expression*

int vals[sz]; // *error:* *size* *not known until run time*

虽然 staff\_size 是用字面值常量进行初始化，但 staff\_size 本身是一个非 const 对象，只有在运行时才能获得它的值，因此，使用该变量来定义数组维数是非法的。而对于 sz，尽管它是一个 const 对象，但它的值要到运行时调用 get\_size 函数后才知道，因此，它也不能用于定义数组维数。

##### 特殊的字符数组

字符数组既可以用一组由花括号括起来、逗号隔开的字符字面值进行初始化，也可以用一个字符串字面值进行初始化。然而，要注意这两种初始化形式并不完全相同，字符串字面值（[第 2.2 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec2.html" \l "ch02lev1sec2)）包含一个额外的空字符（null）用于结束字符串。当使用字符串字面值来初始化创建的新数组时，将在新数组中加入空字符：

char ca1[] = {'C', '+', '+'}; // *no null*

char ca2[] = {'C', '+', '+', '\0'}; // *explicit null*

char ca3[] = "C++"; // *null terminator added automatically*

ca1 的维数是 3，而 ca2 和 ca3 的维数则是 4。使用一组字符字面值初始化字符数组时，一定要记得添加结束字符串的空字符。

##### 不允许数组直接复制和赋值

与vector不同，一个数组不能用另外一个数组初始化，也不能将一个数组赋值给另一个数组，这些操作都是非法的：

int ia[] = {0, 1, 2}; // *ok: array of ints*

int ia2[](ia); // *error: cannot initialize one array with another*

一些编译器允许将数组赋值作为**[编译器扩展](http://manual.51yip.com/c++/0201721481/ch04lev1sec6.html" \l "gloss04_02)**。但是如果希望编写的程序能在不同的编译器上运行，则应该避免使用像数组赋值这类依赖于编译器的非标准功能。

##### 避免使用未初始化的指针

如果必须分开定义指针和其所指向的对象，则将指针初始化为 0。因为编译器可检测出 0 值的指针，程序可判断该指针并未指向一个对象。

##### 指针初始化和赋值操作的约束

除了使用数值0或在编译时值为 0 的 const 量外，还可以使用 C++ 语言从 C 语言中继承下来的预处理器变量 NULL（[第 2.9.2 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec9.html" \l "ch02lev2sec31)），该变量在 cstdlib 头文件中定义，其值为 0。如果在代码中使用了这个预处理器变量，则编译时会自动被数值 0 替换。因此，把指针初始化为 NULL 等效于初始化为 0 值：

// *cstdlib #defines NULL* *to 0*

int \*pi = NULL; // *ok: equivalent to* *int \*pi = 0;*

##### 指针和引用的比较

虽然使用引用（reference）和指针都可间接访问另一个值，但它们之间有两个重要区别。第一个区别在于引用总是指向某个对象：定义引用时没有初始化是错误的。第二个重要区别则是赋值行为的差异：给引用赋值修改的是该引用所关联的对象的值，而并不是使引用与另一个对象关联。引用一经初始化，就*始终*指向同一个特定对象（这就是为什么引用必须在定义时初始化的原因）。

考虑以下两个程序段。第一个程序段将一个指针赋给另一指针：

int ival = 1024, ival2 = 2048;

int \*pi = &ival, \*pi2 = &ival2;

pi = pi2; // *pi* *now points to* *ival2*

赋值结束后，pi 所指向的 ival 对象值保持不变，赋值操作修改了 pi 指针的值，使其指向另一个不同的对象。现在考虑另一段相似的程序，使用两个引用赋值：

int &ri = ival, &ri2 = ival2;

ri = ri2; // *assigns* *ival2* *to* *ival*

这个赋值操作修改了 ri 引用的值 ival 对象，而并非引用本身。赋值后，这两个引用还是分别指向原来关联的对象，此时这两个对象的值相等。

### 4.2. 指针的引入

#### 4.2.5. 指针和 const 限定符

##### 指向 const 对象的指针（常量指针：指向‘常量’的指针，即\*p=?错误，但p=?正确）

到目前为止，我们使用指针来修改其所指对象的值。但是如果指针指向 const 对象，则不允许用指针来改变其所指的 const 值。为了保证这个特性，C++ 语言强制要求指向 const 对象的指针也必须具有 const 特性：

const double \*cptr; // *cptr* *may point to a* *double* *that is* *const*

这里的 cptr 是一个指向 double 类型 const 对象的指针，const 限定了 cptr 指针所指向的对象类型，而并非 cptr 本身。也就是说，cptr 本身并不是 const。在定义时不需要对它进行初始化，如果需要的话，允许给 cptr 重新赋值，使其指向另一个 const 对象。但不能通过 cptr 修改其所指对象的值：

\*cptr = 42; // *error:* *\*cptr* *might be* *const*

允许把非 const 对象的地址赋给指向 const 对象的指针，例如：

double dval = 3.14; // *dval* *is a* *double;* *its value can be changed*

cptr = &dval; // *ok: but can't change* *dval* *through* *cptr*

尽管 dval 不是 const 对象，但任何企图通过指针 cptr 修改其值的行为都会导致编译时的错误。cptr 一经定义，就不允许修改其所指对象的值。如果该指针恰好指向非 const 对象时，同样必须遵循这个规则。

不能使用指向 const 对象的指针修改基础对象，然而如果该指针指向的是一个非 const 对象，可用其他方法修改其所指的对象。

dval = 3.14159; // *dval* *is not* *const*

\*cptr = 3.14159; // *error:* *cptr* *is a pointer to* *const*

double \*ptr = &dval; // *ok:* *ptr* *points at non-const double*

\*ptr = 2.72; // *ok:* *ptr* *is plain pointer*

cout << \*cptr; // *ok: prints 2.72*

如果把指向 const 的指针理解为“自以为指向 const 的指针”，这可能会对理解有所帮助。

在实际的程序中，指向 const 的指针常用作函数的形参。将形参定义为指向 const 的指针，以此确保传递给函数的实际对象在函数中不因为形参而被修改。

##### const 指针（指针常量：指针指向一个常量地址，即\*p=?正确，但p=?错误）

除指向 const 对象的指针外，C++ 语言还提供了 const 指针——本身的值不能修改：

int errNumb = 0;

int \*const curErr = &errNumb; // *curErr* *is a constant pointer*

我们可以从右向左把上述定义语句读作“curErr 是指向 int 型对象的 const 指针”。

Const修饰的是它之后的一个类型！！！

### 4.3. C 风格字符串

**4.3.1. 创建动态数组**

##### 初始化动态分配的数组

动态分配数组时，如果数组元素具有类类型，将使用该类的默认构造函数（[第 2.3.4 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec3.html" \l "ch02lev2sec13)）实现初始化；如果数组元素是内置类型，则无初始化：

string \*psa = new string[10]; // *array of 10 empty* *strings*

int \*pia = new int[10]; // *array of 10 uninitialized* *ints*

也可使用跟在数组长度后面的一对空圆括号，对数组元素做值初始化（[第 3.3.1 节](http://manual.51yip.com/c++/0201721481/ch03lev1sec3.html" \l "ch03lev2sec6)）：

int \*pia2 = new int[10] (); // *array of 10 uninitialized* *ints*

圆括号要求编译器对数组做值初始化，在本例中即把数组元素都设置为0。

对于动态分配的数组，其元素只能初始化为元素类型的默认值，而不能像数组变量一样，用初始化列表为数组元素提供各不相同的初值。

##### 允许动态分配空数组

C++ 虽然不允许定义长度为 0 的数组变量，但明确指出，调用 new 动态创建长度为 0 的数组是合法的：

char arr[0]; // *error: cannot define zero-length array*

char \*cp = new char[0]; // *ok: but* *cp* *can't be dereferenced*

## 第五章 表达式

**5.2. 关系操作符和逻辑操作符**

##### 不应该串接使用关系操作符

关系操作符（<、<=、>、<=）具有左结合特性。事实上，由于关系操作符返回bool类型的结果，因此很少使用其左结合特性。如果把多个关系操作符串接起来使用，结果往往出乎预料：

if (i < j < k) { /\* ... \*/ }

这种写法只要 k 大于 1，上述表达式的值就为 true。这是因为第二个小于操作符的左操作数是第一个小于操作符的结果：true 或 false。也就是，该条件将 k 与整数 0 或 1 做比较。

**5.3. 位操作符**

对于位操作符，由于系统不能确保如何处理其操作数的符号位，所以强烈建议使用unsigned整型操作数。

**5.6. 箭头操作符**

(\*p).foo; // *dereference* *p* *to get an object and fetch its member named* *foo*

p->foo; // *equivalent way to fetch the* *foo* *from the object to which* *p* *points*

**5.11. new 和 delete 表达式**

int i(1024); // *value of* *i* *is 1024*

int \*pi = new int(1024); // *object to which* *pi* *points is 1024*

string s(10, '9'); // *value of* *s* *is "9999999999"*

string \*ps = new string(10, '9'); // *\*ps* *is "9999999999"*

##### 撤销动态创建的对象

如果指针指向不是用 new 分配的内存地址，则在该指针上使用 delete 是不合法的。

int i;

int \*pi = &i;

string str = "dwarves";

double \*pd = new double(33);

delete str; // *error:* *str* *is not a dynamic object*

delete pi; // *error:* *pi* *refers to a local*

delete pd; // *ok*

值得注意的是：编译器可能会拒绝编译 str 的 delete 语句。编译器知道 str 并不是一个指针，因此会在编译时就能检查出这个错误。第二个错误则比较隐蔽：通常来说，编译器不能断定一个指针指向什么类型的对象，因此尽管这个语句是错误的，但在大部分编译器上仍能通过。

执行语句

delete p;

后，p 变成没有定义。在很多机器上，尽管 p 没有定义，但仍然存放了它之前所指向对象的地址，然而 p 所指向的内存已经被释放，因此 p 不再有效。

删除指针后，该指针变成**[悬垂指针](http://manual.51yip.com/c++/0201721481/ch05lev1sec14.html" \l "gloss05_08)**。悬垂指针指向曾经存放对象的内存，但该对象已经不再存在了。悬垂指针往往导致程序错误，而且很难检测出来。

一旦删除了指针所指向的对象，立即将指针置为 0，这样就非常清楚地表明指针不再指向任何对象。

## 第七章 函数

**7.2. 参数传递**

#### 7.2.1. 非引用形参

##### 指针形参

如果保护指针指向的值，则形参需定义为指向 const 对象的指针：

void use\_ptr(const int \*p)

{

// *use\_ptr* *may read but not write to* *\*p*

}

指针形参是指向 const 类型还是非 const 类型，将影响函数调用所使用的实参。我们既可以用 int\* 也可以用 const int\* 类型的实参调用 use\_ptr 函数；但仅能将 int\* 类型的实参传递给 reset 函数。这个差别来源于指针的初始化规则（[第 4.2.5 节](http://manual.51yip.com/c++/0201721481/ch04lev1sec2.html" \l "ch04lev2sec7)）。可以将指向 const 对象的指针初始化为指向非 const对象，但不可以让指向非 const 对象的指针向 const 对象。

void reset(int \*ip)

{

\*ip = 0; // *changes the value of the object to which* *ip* *points*

ip = 0; // *changes only the local value of* *ip; the argument is unchanged*

}

##### const 形参

如果将形参定义为非引用的 const 类型：

void fcn(const int i) { /\* *fcn* *can read but not write to* *i* \*/ }

则在函数中，不可以改变实参的局部副本。由于实参仍然是以副本的形式传递，因此传递给 fcn 的既可以是 const 对象也可以是非 const 对象。

令人吃惊的是，尽管函数的形参是 const，但是编译器却将 fcn 的定义视为其形码被声明为普通的 int 型：

void fcn(const int i) { /\* *fcn* *can read but not write to* *i* \*/ }

void fcn(int i) { /\* ... \*/ } // *error: redefines* *fcn(int)*

这种用法是为了支持对 C 语言的兼容，因为在 C 语言中，具有 const 形参或非 const 形参的函数并无区别。

<https://www.cnblogs.com/langren1992/p/5225652.html>

普通形参加不加const限定符对实参没有影响，引用形参和指针形参前面没有const限定符时，实参必须是非const的，而前面有const限定符时对实参也没有什么影响。

##### 复制实参的局限性

复制实参并不是在所有的情况下都适合，不适宜复制实参的情况包括：

* 当需要在函数中修改实参的值时。
* 当需要以大型对象作为实参传递时。对实际的应用而言，复制对象所付出的时间和存储空间代价往往过在。
* 当没有办法实现对象的复制时。

对于上述几种情况，有效的解决办法是将形参定义为引用或指针类型。

##### 利用 const 引用避免复制

编写一个比较两个 string 对象长度的函数作为例子。这个函数需要访问每个 string 对象的 size，但不必修改这些对象。由于 string 对象可能相当长，所以我们希望避免复制操作。使用 const 引用就可避免复制：

// *compare the length of two strings*

bool isShorter(const string &s1, const string &s2)

{

return s1.size() < s2.size();

}

如果使用引用形参的唯一目的是避免复制实参，则应将形参定义为 const 引用。

##### 更灵活的指向 const 的引用

如果函数具有普通的非 const 引用形参，则显然不能通过 const 对象进行调用。毕竟，此时函数可以修改传递进来的对象，这样就违背了实参的 const 特性。

但比较容易忽略的是，调用这样的函数时，传递一个右值（[第 2.3.1 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec3.html" \l "ch02lev2sec10)）或具有需要转换的类型的对象同样是不允许的：

// *function takes a non-const reference parameter*

int incr(int &val)

{

return ++val;

}

int main()

{

short v1 = 0;

const int v2 = 42;

int v3 = incr(v1); // *error:* *v1* *is not an* *int*

v3 = incr(v2); // *error:* *v2* *is* *const*

v3 = incr(0); // *error: literals are not lvalues*

v3 = incr(v1 + v2); // *error: addition doesn't yield an lvalue*

int v4 = incr(v3); // *ok:* *v3* *is a non* *const* *object type* *int*

}

问题的关键是非 const 引用形参（[第 2.5 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec5.html" \l "ch02lev1sec5)）只能与完全同类型的非 const 对象关联。

应该将不修改相应实参的形参定义为 const 引用。如果将这样的形参定义为非 const 引用，则毫无必要地限制了该函数的使用。例如，可编写下面的程序在一个 string 对象中查找一个指定的字符：

// *returns index of first occurrence of* *c* *in* *s* *or* *s.size()* *if* *c* *isn't in* *s*

// *Note:* *s* *doesn't change, so it should be a reference to* *const*

string::size\_type find\_char(string &s, char c)

{

string::size\_type i = 0;

while (i != s.size() && s[i] != c)

++i; // *not found, look at next character*

return i;

}

这个函数将其 string 类型的实参当作普通（非 const）的引用，尽管函数并没有修改这个形参的值。这样的定义带来的问题是不能通过字符串字面值来调用这个函数：

if (find\_char("Hello World", 'o')) // ...

虽然字符串字面值可以转换为 string 对象，但上述调用仍然会导致编译失败。

应该将不需要修改的引用形参定义为 const 引用。普通的非 const 引用形参在使用时不太灵活。这样的形参既不能用 const 对象初始化，也不能用字面值或产生右值的表达式实参初始化。

##### 传递指向指针的引用

形参

int \*&v1

的定义应从右至左理解：v1 是一个引用，与指向 int 型对象的指针相关联。也就是说，v1 只是传递进 ptrswap 函数的任意指针的别名。

#### 7.2.4. 数组形参

##### 形参的长度会引起误解

编译器忽略为任何数组形参指定的长度。根据数组长度（权且这样说），可将函数 printValues 编写为：

// *parameter treated as* *const int\*, size of array is ignored*

void printValues(const int ia[10])

{

// *this code assumes array has 10 elements;*

// *disaster if argument has fewer than 10 elements!*

for (size\_t i = 0; i != 10; ++i)

{

cout << ia[i] << endl;

}

}

尽管上述代码*假定*所传递的数组至少含有 10 个元素，但 C++ 语言没有任何机制强制实现这个假设。下面的调用都是合法的：

int main()

{

int i = 0, j[2] = {0, 1};

printValues(&i); // *ok:* *&i* *is* *int\*; probable run-time error*

printValues(j); // *ok:* *j* *is converted to pointer to 0th*

// *element; argument has type* *int\*;*

// *probable run-time error*

return 0;

}

虽然编译没有问题，但是这两个调用都是错误的，可能导致运行失败。在这两个调用中，由于函数 printValues 假设传递进来的数组至少含有 10 个元素，因此造成数组内在的越界访问。程序的执行可能产生错误的输出，也可能崩溃，这取决于越界访问的内存中恰好存储的数值是什么。

#### 7.2.5. 传递给函数的数组的处理

有三种常见的编程技巧确保函数的操作不超出数组实参的边界。

第一种方法是在数组本身放置一个标记来检测数组的结束。C 风格字符串就是采用这种方法的一个例子，它是一种字符数组，并且以空字符 null 作为结束的标记。处理 C 风格字符串的程序就是使用这个标记停止数组元素的处理。

第二种方法是传递指向数组第一个和最后一个元素的下一个位置的指针。这种编程风格由标准库所使用的技术启发而得，在[第二部分](http://manual.51yip.com/c++/0201721481/part02.html" \l "part02)将会进一步介绍这种编程风格。

使用这种方法重写函数 printValues 并调用该函数，如下所示：

void printValues(const int \*beg, const int \*end)

{

while (beg != end) {

cout << \*beg++ << endl;

}

}

int main()

{

int j[2] = {0, 1};

// *ok:* *j* *is converted to pointer to 0th element in* *j*

// *j + 2* *refers one past the end of* *j*

printValues(j, j + 2);

return 0;

}

第三种方法是将第二个形参定义为表示数组的大小，这种用法在 C 程序和标准化之前的 C++ 程序中十分普遍。

用这种方法再次重写函数 printValues，新版本及其调用如下所示：

// *const int ia[]* *is equivalent to* *const int\* ia*

// *size* *is passed explicitly and used to control access to elements of* *ia*

void printValues(const int ia[], size\_t size)

{

for (size\_t i = 0; i != size; ++i) {

cout << ia[i] << endl;

}

}

int main()

{

int j[] = { 0, 1 }; // *int* *array of size 2*

printValues(j, sizeof(j)/sizeof(\*j));

return 0;

}

#### 7.2.6. main: 处理命令行选项

传统上，主函数的实参是可选的，用来确定程序要执行的操作。比如，假设我们的主函数 main 位于名为 prog 的可执行文件中，可如下将实参选项传递给程序：

prog -d -o ofile data0

这种用法的处理方法实际上是在主函数 main 中定义了两个形参：

int main(int argc, char \*argv[]) { ... }

第二个形参 argv 是一个 C 风格字符串数组。第一个形参 argc 则用于传递该数组中字符串的个数。由于第二个参数是一个数组，主函数 main 也可以这样定义：

int main(int argc, char \*\*argv) { ... }

表示 argv 是指向 char\* 的指针。

当将实参传递给主函数 main 时，argv 中的第一个字符串（如果有的话）通常是程序的名字。接下来的元素将额外的可选字符串传递给主函数 main。以前面的命令行为例，argc 应设为 5，argv 会保存下面几个 C 风格字符串：

argv[0] = "prog";

argv[1] = "-d";

argv[2] = "-o";

argv[3] = "ofile";

argv[4] = "data0";

### 7.3. return 语句

#### 7.3.1. 没有返回值的函数

一般情况下，返回类型是 void 的函数使用 return 语句是为了引起函数的强制结束，这种 return 的用法类似于循环结构中的 break 语句（[第 6.10 节](http://manual.51yip.com/c++/0201721481/ch06lev1sec10.html" \l "ch06lev1sec10)）的作用。

例如，可如下重写 swap 程序，使之在输入的两个数值相同时不执行任何工作：

// *ok:* *swap* *acts on references to its arguments*

void swap(int &v1, int &v2)

{

// *if values already the same, no need to swap, just return*

if (v1 == v2)

return;

// *ok, have work to do*

int tmp = v2;

v2 = v1;

v1 = tmp;

// *no explicit return necessary*

}

#### 7.3.2. 具有返回值的函数

##### 千万不要返回局部对象的引用

当函数执行完毕时，将释放分配给局部对象的存储空间。此时，对局部对象的引用就会指向不确定的内存。考虑下面的程序：

// *Disaster: Function returns a reference to a local object*

const string &manip(const string& s)

{

string ret = s;

// *transform ret in some way*

return ret; // *Wrong: Returning reference to a local object!*

}

这个函数会在运行时出错，因为它返回了局部对象的引用。当函数执行完毕，字符串 ret 占用的储存空间被释放，函数返回值指向了对于这个程序来说不再有效的内存空间。

##### 引用返回左值

返回引用的函数返回一个左值。因此，这样的函数可用于任何要求使用左值的地方：

char &get\_val(string &str, string::size\_type ix)

{

return str[ix];

}

int main()

{

string s("a value");

cout << s << endl; // *prints* *a value*

get\_val(s, 0) = 'A'; // *changes* *s[0]* *to* *A*

cout << s << endl; // *prints* *A value*

return 0;

}

给函数返回值赋值可能让人惊讶，由于函数返回的是一个引用，因此这是正确的，该引用是被返回元素的同义词。

const char &get\_val(...)

### 7.4. 函数声明

#### 7.4.1. 默认实参

默认实参是通过给形参表中的形参提供明确的初始值来指定的。程序员可为一个或多个形参定义默认值。但是，如果有一个形参具有默认实参，那么，它后面所有的形参都必须有默认实参。

例如，下面的函数创建并初始化了一个 string 对象，用于模拟窗口屏幕。此函数为窗口屏幕的高、宽和背景字符提供了默认实参：

string screenInit(string::size\_type height = 24,

string::size\_type width = 80,

char background = ' ' );

函数调用的实参按位置解析，默认实参只能用来替换函数调用缺少的*尾部*实参。例如，如果要给 background 提供实参，那么也必须给 height 和 width 提供实参：

screen = screenInit(, , '?'); // *error, can omit only trailing arguments*

screen = screenInit( '?'); // *calls* *screenInit('?',80,' ')*

### 7.6. 内联函数

#### inline 函数避免函数调用的开销

// *inline version: find longer of two* *strings*

inline const string &

shorterString(const string &s1, const string &s2)

{

return s1.size() < s2.size() ? s1 : s2;

}

将函数指定为 inline 函数，（通常）就是将它在程序中每个调用点上“内联地”展开。假设我们将 shorterString 定义为内联函数，则调用：

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

在编译时将展开为：

cout << (s1.size() < s2.size() ? s1 : s2)

<< endl;

从而消除了把 shorterString 写成函数的额外执行开销。

一般来说，内联机制适用于优化小的、只有几行的而且经常被调用的函数。大多数的编译器都不支持递归函数的内联。一个 1200 行的函数也不太可能在调用点内联展开。内联函数应该在头文件中定义，这一点不同于其他函数。

### 7.7. 类的成员函数

class Sales\_item {

public:

// *operations on* *Sales\_item* *objects*

double avg\_price() const;

bool same\_isbn(const Sales\_item &rhs) const

{ return isbn == rhs.isbn; }

// *private members as before*

private:

std::string isbn;

unsigned units\_sold;

double revenue;

};

#### 7.7.1. 定义成员函数的函数体

##### this 指针的引入

每个成员函数（除了在[第 12.6 节](http://manual.51yip.com/c++/0201721481/ch12lev1sec6.html" \l "ch12lev1sec6)介绍的 static 成员函数外）都有一个额外的、隐含的形参 **this**。在调用成员函数时，形参 this 初始化为调用函数的对象的地址。为了理解成员函数的调用，可考虑下面的语句：

total.same\_isbn(trans);

就如编译器这样重写这个函数调用：

// *pseudo-code illustration of how a call to a member function is translated*

Sales\_item::same\_isbn(&total, trans);

##### const 成员函数的引入

现在，可以理解跟在 Sales\_item 成员函数声明的形参表后面的 const 所起的作用了：const 改变了隐含的 this 形参的类型。在调用 total.same\_isbn(trans) 时，隐含的 this 形参将是一个指向 total 对象的 const Sales\_Item\* 类型的指针。就像如下编写 same\_isbn 的函数体一样：

// *pseudo-code illustration of how the implicit* *this* *pointer is used*

// *This code is illegal: We may not explicitly define the* *this* *pointer ourselves*

// *Note that* *this* *is a pointer to* *const* *because* *same\_isbn* *is a* *const* *member*

bool Sales\_item::same\_isbn(const Sales\_item\* this,

const Sales\_item &rhs) const

{ return (this->isbn == rhs.isbn); }

用这种方式使用 const 的函数称为**[常量成员函数](http://manual.51yip.com/c++/0201721481/ch07lev1sec11.html" \l "gloss07_07)**。由于 this 是指向 const 对象的指针，const 成员函数不能修改调用该函数的对象。因此，函数 avg\_price 和函数 same\_isbn 只能读取而不能修改调用它们的对象的数据成员。

const 对象、指向 const 对象的指针或引用只能用于调用其 const 成员函数，如果尝试用它们来调用非 const成员函数，则是错误的。

#### 7.7.3. 编写 Sales\_item 类的构造函数

##### 构造函数是特殊的成员函数

**[构造函数](http://manual.51yip.com/c++/0201721481/ch07lev1sec11.html" \l "gloss07_08)**是特殊的成员函数，与其他成员函数不同，构造函数和类同名，而且没有返回类型。而与其他成员函数相同的是，构造函数也有形参表（可能为空）和函数体。一个类可以有多个构造函数，每个构造函数必须有与其他构造函数不同数目或类型的形参。

##### 构造函数的定义

class Sales\_item {

public:

// *operations on* *Sales\_item* *objects*

double avg\_price() const;

bool same\_isbn(const Sales\_item &rhs) const

{ return isbn == rhs.isbn; }

// *default constructor needed to initialize members of built-in type*

Sales\_item(): units\_sold(0), revenue(0.0) { }

// *private members as before*

private:

std::string isbn;

unsigned units\_sold;

double revenue;

};

构造函数是放在类的 public 部分的。通常构造函数会作为类的接口的一部分，这个例子也是这样。毕竟，我们希望使用类 Sales\_item 的代码可以定义和初始化类 Sales\_item 的对象。如果将构造函数定义为 private 的，则不能定义类 Sales\_item 的对象，这样的话，这个类就没有什么用了。

### 7.8. 重载函数

#### 函数重载和重复声明的区别

如果两个函数声明的返回类型和形参表完全匹配，则将第二个函数声明视为第一个的重复声明。如果两个函数的形参表完全相同，但返回类型不同，则第二个声明是错误的：

Record lookup(const Account&);

bool lookup(const Account&); // *error: only return type is different*

函数不能仅仅基于不同的返回类型而实现重载。

typedef Phone Telno;

Record lookup(const Phone&, const Name& = "");

// *const* *is irrelevent for nonreference parameters*

Record lookup(Phone);

Record lookup(const Phone); // *redeclaration*

区别仅在于是否将形参定义为 const。这种差异并不影响传递至函数的对象；第二个函数声明被视为第一个的重复声明。其原因在于实参传递的方式。复制形参时并不考虑形参是否为 const——函数操纵的只是副本。函数的无法修改实参。结果，既可将 const 对象传递给 const 形参，也可传递给非 const 形参，这两种形参并无本质区别。

值得注意的是，形参与 const 形参的等价性仅适用于非引用形参。有 const 引用形参的函数与有非 const 引用形参的函数是不同的。类似地，如果函数带有指向 const 类型的指针形参，则与带有指向相同类型的非 const 对象的指针形参的函数不相同。

#### 7.8.3. 重载确定的三个步骤

考虑下面的这组函数和函数调用：

void f();

void f(int);

void f(int, int);

void f(double, double = 3.14);

f(5.6); // *calls* *void f(double, double)*

##### （1）候选函数

##### （2）选择可行函数

* f(int) 是一个可行函数，因为通过隐式转换可将函数调用中的 double 型实参转换为该函数唯一的 int 型形参。
* f(double, double) 也是一个可行函数，因为该函数为其第二个形参提供了默认实参，而且第一个形参是 double 类型，与实参类型精确匹配。

##### （3）寻找最佳匹配（如果有的话）

函数重载确定的第三步是确定与函数调用中使用的实际参数匹配最佳的可行函数。这个过程考虑函数调用中的每一个实参，选择对应形参与之最匹配的一个或多个可行函数。这里所谓“最佳”的细节将在下一节中解释，其原则是实参类型与形参类型越接近则匹配越佳。因此，实参类型与形参类型之间的精确类型匹配比需要转换的匹配好。

在上述例子中，只需考虑一个 double 类型的显式实参。如果调用 f(int)，实参需从 double 型转换为 int 型。而另一个可行函数 f(double, double) 则与该实参精确匹配。由于精确匹配优于需要类型转换的匹配，因此编译器将会把函数调用 f(5.6) 解释为对带有两个 double 形参的 f 函数的调用。

##### 含有多个形参的重载确定

f(42, 2.56);

可行函数将以同样的方式选出。编译器将选出形参个数和类型都与实参匹配的函数。在本例中，可行函数是 f(int, int) 和 f(double, double)。接下来，编译器通过依次检查每一个实参来决定哪个或哪些函数匹配最佳。如果有且仅有一个函数满足下列条件，则匹配成功：

1. 其每个实参的匹配都不劣于其他可行函数需要的匹配。
2. 至少有一个实参的匹配优于其他可行函数提供的匹配。

如果在检查了所有实参后，仍找不到唯一最佳匹配函数，则该调用错误。编译器将提示该调用具有二义性。

在本例子的调用中，首先分析第一个实参，发现函数 f(int, int) 匹配精确。如果使之与第二个函数匹配，就必须将 int 型实参 42 转换为 double 型的值。通过内置转换的匹配“劣于”精确匹配。所以，如果只考虑这个形参，带有两个 int 型形参的函数比带有两个 double 型形参的函数匹配更佳。

但是，当分析第二个实参时，有两个 double 型形参的函数为实参 2.56 提供了精确匹配。而调用两个 int 型形参的 f 函数版本则需要把 2.56 从 double 型转换为 int 型。所以只考虑第二个形参的话，函数 f(double, double) 匹配更佳。

因此，这个调用有二义性：每个可行函数都对函数调用的一个实参实现更好的匹配。编译器将产生错误。解决这样的二义性，可通过显式的强制类型转换强制函数匹配：

#### 7.8.4. 实参类型转换

##### 重载和 const 形参

仅当形参是引用或指针时，形参是否为 const 才有影响。

可基于函数的引用形参是指向 const 对象还是指向非 const 对象，实现函数重载。将引用形参定义为 const 来重载函数是合法的，因为编译器可以根据实参是否为 const 确定调用哪一个函数：

Record lookup(Account&);

Record lookup(const Account&); // *new function*

const Account a(0);

Account b;

lookup(a); // *calls* *lookup(const Account&)*

lookup(b); // *calls* *lookup(Account&)*

如果形参是普通的引用，则不能将 const 对象传递给这个形参。如果传递了 const 对象，则只有带 const 引用形参的版本才是该调用的可行函数。

如果传递的是非 const 对象，则上述任意一种函数皆可行。非 const 对象既可用于初始化 const 引用，也可用于初始化非 const 引用。但是，将 const 引用初始化为非 const 对象，需通过转换来实现，而非 const 形参的初始化则是精确匹配。

对指针形参的相关处理如出一辙。可将 const 对象的地址值只传递给带有指向 const 对象的指针形参的函数。也可将指向非 const 对象的指针传递给函数的 const 或非 const 类型的指针形参。如果两个函数仅在指针形参时是否指向 const 对象上不同，则指向非 const 对象的指针形参对于指向非 const 对象的指针（实参）来说是更佳的匹配。

注意不能基于指针本身是否为 const 来实现函数的重载：

f(int \*);

f(int \*const); // *redeclaration*

此时，const 用于修改指针本身，而不是修饰指针所指向的类型。在上述两种情况中，都复制了指针，指针本身是否为 const 并没有带来区别。正如前面[第 7.8 节](http://manual.51yip.com/c++/0201721481/ch07lev1sec8.html" \l "page_267)所提到的，当形参以副本传递时，不能基于形参是否为 const 来实现重载。(可见7.8节-函数重载和声明的区别)

### 7.9. 指向函数的指针（函数指针）

函数指针是指指向函数而非指向对象的指针。像其他指针一样，函数指针也指向某个特定的类型。函数类型由其返回类型以及形参表确定，而与函数名无关：

// *pf* *points to function returning* *bool* *that takes two* *const string* *references*

bool (\*pf)(const string &, const string &);

这个语句将 pf 声明为指向函数的指针，它所指向的函数带有两个 const string& 类型的形参和 bool 类型的返回值。

\*pf 两侧的圆括号是必需的：

// *declares a function named* *pf* *that returns a* *bool\**

bool \*pf(const string &, const string &);

#### 用 typedef 简化函数指针的定义

函数指针类型相当地冗长。使用 typedef 为指针类型定义同义词，可将函数指针的使用大大简化：（[第 2.6 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec6.html" \l "ch02lev1sec6)）：

typedef bool (\*cmpFcn)(const string &, const string &);

该定义表示 cmpFcn 是一种指向函数的指针类型的名字。该指针类型为“指向返回 bool 类型并带有两个 const string 引用形参的函数的指针”。在要使用这种函数指针类型时，只需直接使用 cmpFcn 即可，不必每次都把整个类型声明全部写出来。

#### 指向函数的指针的初始化和赋值

可使用函数名对函数指针做初始化或赋值：

cmpFcn pf1 = 0; // *ok: unbound pointer to function*

cmpFcn pf2 = lengthCompare; // *ok: pointer type matches function's type*

pf1 = lengthCompare; // *ok: pointer type matches function's type*

pf2 = pf1; // *ok: pointer types match*

此时，直接引用函数名等效于在函数名上应用取地址操作符：

cmpFcn pf1 = lengthCompare;

cmpFcn pf2 = &lengthCompare;

指向函数的指针可用于调用它所指向的函数。可以不需要使用解引用操作符，直接通过指针调用函数：

cmpFcn pf = lengthCompare;

lengthCompare("hi", "bye"); // *direct call*

pf("hi", "bye"); // *equivalent call:* *pf1* *implicitly dereferenced 隐式引用*

(\*pf)("hi", "bye"); // *equivalent call:* *pf1* *explicitly dereferenced 显示引用*

函数指针只能通过同类型的函数或函数指针或 0 值常量表达式进行初始化或赋值。将函数指针初始化为 0，表示该指针不指向任何函数。

#### 函数指针形参

函数的形参可以是指向函数的指针。这种形参可以用以下两种形式编写：

/\* *useBigger* *function's third parameter is a pointer to function*

\* *that function returns a* *bool* *and takes two* *const string* *references*

\* *two ways to specify that parameter*:

\*/

// *third parameter is a function type and is automatically treated as a pointer to function*

void useBigger(const string &, const string &,

bool(const string &, const string &));

// *equivalent declaration: explicitly define the parameter as a pointer to function*

void useBigger(const string &, const string &,

bool (\*)(const string &, const string &));

#### 返回指向函数的指针

函数可以返回指向函数的指针，但是，正确写出这种返回类型相当不容易：

// *ff* *is a function taking an* *int* *and returning a function pointer*

// *the function pointed to returns an* *int* *and takes an* *int\** *and an* *int*

int (\*ff(int))(int\*, int);

要理解该声明的含义，首先观察：

ff(int)

将 ff 声明为一个函数，它带有一个 int 型的形参。该函数返回

int (\*)(int\*, int);

它是一个指向函数的指针，所指向的函数返回 int 型并带有两个分别是 int\* 型和 int 型的形参。

使用 typedef 可使该定义更简明易懂：

// *PF* *is a pointer to a function returning an* *int, taking an* *int\** *and an* *int*

typedef int (\*PF)(int\*, int);

PF ff(int); // *ff* *returns a pointer to function*

允许将形参定义为函数类型，但函数的返回类型则必须是指向函数的指针，而不能是函数。

// *func* *is a function type, not a pointer to function!*

typedef int func(int\*, int);

void f1(func); // *ok:* *f1* *has a parameter of function type*

func f2(int); // *error:* *f2* *has a return type of function type*

func \*f3(int); // *ok:* *f3* *returns a pointer to function type*

## 第八章 标准 IO 库

# 第三部分：类和数据抽象

## 第十二章 类

### 12.1. 类的定义和声明

#### 12.1.2. 数据抽象和封装

##### 访问标号实施抽象和封装

可以在任意的访问标号出现之前定义类成员。在类的左花括号之后、第一个访问标号之前定义成员的访问级别，其值依赖于类是如何定义的。如果类是用 struct 关键字定义的，则在第一个访问标号之前的成员是公有的；如果类是用 class 关键字是定义的，则这些成员是私有的。

#### 12.1.3. 关于类定义的更多内容

##### 使用类型别名来简化类

除了定义数据和函数成员之外，类还可以定义自己的局部类型名字。如果为 std::string::size\_type 提供一个类型别名，那么 Screen 类将是一个更好的抽象：

class Screen {

public:

// *interface member functions*

typedef std::string::size\_type index;

private:

std::string contents;

index cursor;

index height, width;

};

##### 显式指定 inline 成员函数

在类内部定义的成员函数，例如不接受实参的 get 成员，将自动作为 inline 处理。也就是说，当它们被调用时，编译器将试图在同一行内扩展该函数（[第 7.6 节](http://manual.51yip.com/c++/0201721481/ch07lev1sec6.html" \l "ch07lev1sec6)）。也可以显式地将成员函数声明为 inline：

class Screen {

public:

typedef std::string::size\_type index;

// *implicitly* *inline* *when defined inside the class declaration*

char get() const { return contents[cursor]; }

// *explicitly declared as* *inline*; *will be defined outside the class declaration*

inline char get(index ht, index wd) const;

// *inline* *not specified in class declaration, but can be defined* *inline* *later*

index get\_cursor() const;

// ...

};

// *inline* *declared in the class declaration; no need to repeat on the definition*

char Screen::get(index r, index c) const

{

index row = r \* width; // *compute the row location*

return contents[row + c]; // *offset by* *c* *to fetch specified character*

}

// *not declared as* *inline* *in the class declaration, but ok to make* *inline* *in definition*

inline Screen::index Screen::get\_cursor() const

{

return cursor;

}

可以在类定义体内部指定一个成员为inline，作为其声明的一部分。或者，也可以在类定义外部的函数定义上指定 inline。在声明和定义处指定 inline 都是合法的。在类的外部定义 inline 的一个好处是可以使得类比较容易阅读。

像其他 inline 一样，inline 成员函数的定义必须在调用该函数的每个源文件中是可见的。不在类定义体内定义的 inline 成员函数，其定义通常应放在有类定义的同一头文件中。

#### 12.1.4. 类声明与类定义

##### 前向声明（声明一个类而不定义）

可以声明一个类而不定义它：

class Screen; // *declaration of the Screen class*

这个声明，有时称为**[前向声明（forward declaraton）](http://manual.51yip.com/c++/0201721481/ch12lev1sec8.html" \l "gloss12_15)**，在程序中引入了类类型的 Screen。在声明之后、定义之前，类 Screen 是一个**不完全类型（incompete type）**，即已知 Screen 是一个类型，但不知道包含哪些成员。

[不完全类型（incomplete type）](http://manual.51yip.com/c++/0201721481/ch12lev1sec8.html" \l "gloss12_17)只能以有限方式使用。不能定义该类型的对象。不完全类型只能用于定义指向该类型的指针及引用，或者用于声明（而不是定义）使用该类型作为形参类型或返回类型的函数。

##### 为类的成员使用类声明（前身声明）

只有当类定义已经在前面出现过，数据成员才能被指定为该类类型。因为只有当类定义体完成后才能定义类，因此类不能具有自身类型的数据成员。然而，只要类名一出现就可以认为该类已声明。因此，类的数据成员可以是指向自身类型的指针或引用：

class LinkScreen {

Screen window;

LinkScreen \*next;

LinkScreen \*prev;

};

类的前身声明一般用来编写相互依赖的类。在第 13.4 节中，我们将看到用法的一个例子。

##### 定义类类型的对象

定义了一个类类型之后，可以按以下两种方式使用。

* 将类的名字直接用作类型名。
* 指定关键字 class 或 struct，后面跟着类的名字：

Sales\_item item1; // *default initialized object of type* *Sales\_item*

class Sales\_item item1; // *equivalent definition of* *item1*

两种引用类类型方法是等价的。第二种方法是从 C 继承而来的，在 C++ 中仍然有效。第一种更为简练，由 C++ 语言引入，使得类类型更容易使用。

### 12.2. 隐含的 this 指针

#### 何时使用 this 指针

尽管在成员函数内部显式引用 this 通常是不必要的，但有一种情况下必须这样做：当我们需要将一个对象作为整体引用而不是引用对象的一个成员时。最常见的情况是在这样的函数中使用 this：该函数返回对调用该函数的对象的引用。

某种类可能具有某些操作，这些操作应该返回引用，Screen 类就是这样的一个类。迄今为止，我们的类只有一对 get 操作。逻辑上，我们可以添加下面的操作。

* 一对 set 操作，将特定字符或光标指向的字符设置为给定值。
* 一个 move 操作，给定两个 index 值，将光标移至新位置。

理想情况下，希望用户能够将这些操作的序列连接成一个单独的表达式：

// *move cursor to given position, and set that character*

myScreen.move(4,0).set('#');

这个语句等价于：

myScreen.move(4,0);

myScreen.set('#');

#### 返回 \*this

下面是对两个新成员的实现：

Screen& Screen::set(char c)

{

contents[cursor] = c;

return \*this;

}

Screen& Screen::move(index r, index c)

{

index row = r \* width; // *row location*

cursor = row + c;

return \*this;

}

函数中唯一需要关注的部分是 return 语句。在这两个操作中，每个函数都返回 \*this。在这些函数中，this 是一个指向非常量 Screen 的指针。如同任意的指针一样，可以通过对 this 指针解引用来访问 this 指向的对象。

#### 从 const 成员函数返回 \*this

在普通的非 const 成员函数中，this 的类型是一个指向类类型的 const 指针（[第 4.2.5 节](http://manual.51yip.com/c++/0201721481/ch04lev1sec2.html" \l "ch04lev2sec7)）。可以改变 this 所指向的值，但不能改变 this 所保存的地址。在 const 成员函数中，this 的类型是一个指向 const 类类型对象的 const 指针。既不能改变 this 所指向的对象，也不能改变 this 所保存的地址。

不能从 const 成员函数返回指向类对象的普通引用。const 成员函数只能返回 \*this 作为一个 const 引用。

例如，我们可以给 Screen 类增加一个 display 操作。这个函数应该在给定的 ostream 上打印 contents。逻辑上，这个操作应该是一个 const 成员。打印 contents 不会改变对象。如果将 display 作为 Screen 的 const 成员，则 display 内部的 this 指针将是一个 const Screen\* 型的 const。

然而，与 move 和 set 操作一样，我们希望能够在一个操作序列中使用 display：

// *move cursor to given position, set that character and display the screen*

myScreen.move(4,0).set('#').display(cout);

这个用法暗示了 display 应该返回一个 Screen 引用，并接受一个 ostream 引用。如果 display 是一个 const 成员，则它的返回类型必须是 const Screen&。

不幸的是，这个设计存在一个问题。如果将 display 定义为 const 成员，就可以在非 const 对象上调用 display，但不能将对 display 的调用嵌入到一个长表达式中。下面的代码将是非法的：

Screen myScreen;

// *this code fails if* *display* *is a* *const* *member function*

// *display* *return a* *const* *reference; we cannot call* *set* *on a* *const*

myScreen.display().set('\*');

问题在于这个表达式是在由 display 返回的对象上运行 set。该对象是 const，因为 display 将其对象作为 const 返回。我们不能在 const 对象上调用 set。

#### 基于 const 的重载

为了解决这个问题，我们必须定义两个 display 操作：一个是 const，另一个不是 const。基于成员函数是否为 const，可以重载一个成员函数；同样地，基于一个指针形参是否指向 const（[第 7.8.4 节](http://manual.51yip.com/c++/0201721481/ch07lev1sec8.html" \l "ch07lev2sec30)），可以重载一个函数。const 对象只能使用 const 成员。非 const 对象可以使用任一成员，但非 const 版本是一个更好的匹配。

在此，我们将定义一个名为 do\_display 的 private 成员来打印 Screen。每个 display 操作都将调用此函数，然后返回调用自己的那个对象：

class Screen {

public:

// *interface member functions*

// *display overloaded on whether the object is const or not*

Screen& display(std::ostream &os)

{ do\_display(os); return \*this; }

const Screen& display(std::ostream &os) const

{ do\_display(os); return \*this; }

private:

// *single function to do the work of displaying a Screen,*

// *will be called by the display operations*

void do\_display(std::ostream &os) const

{ os << contents; }

// *as before*

};

现在，当我们将 display 嵌入到一个长表达式中时，将调用非 const 版本。当我们 display 一个 const 对象时，就调用 const 版本：

Screen myScreen(5,3);

const Screen blank(5, 3);

myScreen.set('#').display(cout); // *calls nonconst* *version*

blank.display(cout); // *calls* *const* *version*

#### 可变数据成员

有时（但不是很经常），我们希望类的数据成员（甚至在 const 成员函数内）可以修改。这可以通过将它们声明为 mutable 来实现。

**[可变数据成员（mutable data member）](http://manual.51yip.com/c++/0201721481/ch12lev1sec8.html" \l "gloss12_19)**永远都不能为 const，甚至当它是 const 对象的成员时也如此。因此，const 成员函数可以改变 mutable 成员。要将数据成员声明为可变的，必须将关键字 mutable 放在成员声明之前：

class Screen {

public:

// *interface member functions*

private:

mutable size\_t access\_ctr; // *may change in a* *const* *members*

// *other data members as before*

};

我们给 Screen 添加了一个新的可变数据成员 access\_ctr。使用 access\_ctr 来跟踪调用 Screen 成员函数的频繁程度：

void Screen::do\_display(std::ostream& os) const

{

++access\_ctr; // *keep count of calls to any member function*

os << contents;

}

尽管 do\_display 是 const，它也可以增加 access\_ctr。该成员是可变成员，所以，任意成员函数，包括 const 函数，都可以改变 access\_ctr 的值。

### 12.3. 类作用域

#### 使用类的成员

在类作用域之外，成员只能通过对象或指针分别使用成员访问操作符 . 或 -> 来访问。这些操作符左边的操作数分别是一个类对象或指向类对象的指针。

Class obj; // *Class* *is some class type*

Class \*ptr = &obj;

// *member* *is a data member of that class*

ptr->member; // *fetches* *member* *from the object to which* *ptr* *points*

obj.member; // *fetches* *member* *from the object named* *obj*

#### 形参表和函数体处于类作用域中

#### 函数返回类型不一定在类作用域中

与形参类型相比，返回类型出现在成员名字前面。如果函数在类定义体之外定义，则用于返回类型的名字在类作用域之外。如果返回类型使用由类定义的类型，则必须使用完全限定名。例如，考虑 get\_cursor 函数：

class Screen {

public:

typedef std::string::size\_type index;

index get\_cursor() const;

};

inline Screen::index Screen::get\_cursor() const

{

return cursor;

}

### 12.4. 构造函数

#### 用于 const 对象的构造函数

构造函数不能声明为 const [第 7.7.1 节](http://manual.51yip.com/c++/0201721481/ch07lev1sec7.html" \l "ch07lev2sec22)：

class Sales\_item {

public:

Sales\_item() const; // *error*

};

const 构造函数是不必要的。创建类类型的 const 对象时，运行一个普通构造函数来初始化该 const 对象。构造函数的工作是初始化对象。不管对象是否为 const，都用一个构造函数来初始化化该对象。

#### 12.4.1. 构造函数初始化式

##### 有时需要构造函数初始化列表

省略初始化列表在构造函数的函数体内对数据成员*赋值*是合法的。例如，可以将接受一个 string 的 Sales\_item 构造函数编写为：

// *legal but sloppier way to write the constructor:*

// *no constructor initializer*

Sales\_item::Sales\_item(const string &book)

{

isbn = book;

units\_sold = 0;

revenue = 0.0;

}

如果没有为类成员提供初始化式，则编译器会隐式地使用成员类型的默认构造函数。如果那个类没有默认构造函数，则编译器尝试使用默认构造函数将会失败。在这种情况下，为了初始化数据成员，必须提供初始化式。

有些成员*必须*在构造函数初始化列表中进行初始化。对于这样的成员，在构造函数函数体中对它们赋值不起作用。没有默认构造函数的类类型的成员，以及 const 或引用类型的成员，不管是哪种类型，都必须在构造函数初始化列表中进行初始化。

因为内置类型的成员不进行隐式初始化，所以对这些成员是进行初始化还是赋值似乎都无关紧要。除了两个例外，对非类类型的数据成员进行赋值或使用初始化式在结果和性能上都是等价的。

例如，下面的构造函数是错误的：

class ConstRef {

public:

ConstRef(int ii);

private:

int i;

const int ci;

int &ri;

};

// *no explicit constructor initializer: error* *ri* *is uninitialized*

ConstRef::ConstRef(int ii)

{ // *assignments:*

i = ii; // *ok*

ci = ii; // *error: cannot assign to a* *const*

ri = i; // *assigns to* *ri* *which was not bound to an object*

}

记住，可以初始化 const 对象或引用类型的对象，但不能对它们赋值。在开始执行构造函数的函数体之前，要完成初始化。初始化 const 或引用类型数据成员的唯一机会是构造函数初始化列表中。编写该构造函数的正确方式为

// *ok: explicitly initialize reference and* *const* *members*

ConstRef::ConstRef(int ii): i(ii), ci(i), ri(ii) { }

##### 成员初始化的次序

构造函数初始化列表仅指定用于初始化成员的值，并不指定这些初始化执行的次序。成员被初始化的次序就是定义成员的次序。第一个成员首先被初始化，然后是第二个，依次类推。

初始化的次序常常无关紧要。然而，如果一个成员是根据其他成员而初始化，则成员初始化的次序是至关重要的。

按照与成员声明一致的次序编写构造函数初始化列表是个好主意。此外，尽可能避免使用成员来初始化其他成员。

#### 12.4.3. 默认构造函数(没有形参的构造函数称为[默认构造函数](http://manual.51yip.com/c++/0201721481/ch07lev1sec11.html" \l "gloss07_10))

##### 使用默认构造函数

初级 C++ 程序员常犯的一个错误是，采用以下方式声明一个用默认构造函数初始化的对象：

// *oops! declares a function, not an object*

Sales\_item myobj();

编译 myobj 的声明没有问题。然而，当我们试图使用 myobj 时

Sales\_item myobj(); // *ok: but defines a function, not an object*

if (myobj.same\_isbn(Primer\_3rd\_ed)) // *error:* *myobj* *is a function*

编译器会指出不能将成员访问符号用于一个函数！问题在于 myobj 的定义被编译器解释为一个函数的声明，该函数不接受参数并返回一个 Sales\_item 类型的对象——与我们的意图大相径庭！使用默认构造函数定义一个对象的正确方式是去掉最后的空括号：

// *ok: defines a class object ...*

Sales\_item myobj;

另一方面，下面这段代码也是正确的：

// *ok: create an unnamed, empty* *Sales\_itemand* *use to initialize* *myobj*

Sales\_item myobj = Sales\_item();

在这里，我们创建并初始化一个 Sales\_item 对象，然后用它来按值初始化 myobj。编译器通过运行 Sales\_item 的默认构造函数来按值初始化一个 Sales\_item。

#### 12.4.4. 隐式类类型转换

可以用单个实参来调用的构造函数定义了从形参类型到该类类型的一个隐式转换。

string null\_book = "9-999-99999-9";

// *ok: builds a* *Sales\_itemwith* *0* *units\_soldand revenue* *from*

// *and* *isbn* *equal to* *null\_book*

item.same\_isbn(null\_book);

这段程序使用一个 string 类型对象作为实参传给 Sales\_item 的 same\_isbn 函数。该函数期待一个 Sales\_item 对象作为实参。编译器使用接受一个 string 的 Sales\_item 构造函数从 null\_book 生成一个新的 Sales\_item 对象。新生成的（临时的）Sales\_item 被传递给 same\_isbn。

##### 抑制由构造函数定义的隐式转换

可以通过将构造函数声明为 explicit，来防止在需要隐式转换的上下文中使用构造函数：

class Sales\_item {

public:

// *default argument for book is the empty string*

explicit Sales\_item(const std::string &book = ""):

isbn(book), units\_sold(0), revenue(0.0) { }

explicit Sales\_item(std::istream &is);

// *as before*

};

explicit 关键字只能用于类内部的构造函数声明上。在类的定义体外部所做的定义上不再重复它：

// *error:* *explicit* *allowed only on constructor declaration in class header*

explicit Sales\_item::Sales\_item(istream& is)

{

is >> \*this; // *uses* *Sales\_iteminput* *operator to read the members*

}

现在，两个构造函数都不能用于隐式地创建对象。前两个使用都不能编译：

item.same\_isbn(null\_book); // *error:* *string* *constructor is* *explicit*

item.same\_isbn(cin); // *error:* *istream* *constructor is* *explicit*

当构造函数被声明 explicit 时，编译器将*不*使用它作为转换操作符。

##### 为转换而显式地使用构造函数

只要显式地按下面这样做，就可以用[显式的构造函数](http://manual.51yip.com/c++/0201721481/ch12lev1sec8.html" \l "gloss12_14)来生成转换：

string null\_book = "9-999-99999-9";

// *ok: builds a* *Sales\_itemwith* *0* *units\_soldand revenue* *from*

// *and* *isbn* *equal to* *null\_book*

item.same\_isbn(Sales\_item(null\_book));

在这段代码中，从 null\_book 创建一个 Sales\_item。尽管构造函数为显式的，但这个用法是允许的。显式使用构造函数只是中止了隐式地使用构造函数。任何构造函数都可以用来显式地创建临时对象。

通常，除非有明显的理由想要定义隐式转换，否则，单形参构造函数应该为 explicit。将构造函数设置为 explicit 可以避免错误，并且当转换有用时，用户可以显式地构造对象。

通常，除非有明显的理由想要定义隐式转换，否则，单形参构造函数应该为 explicit。将构造函数设置为 explicit 可以避免错误，并且当转换有用时，用户可以显式地构造对象

### 12.5. 友元

**[友元](http://manual.51yip.com/c++/0201721481/ch12lev1sec8.html" \l "gloss12_16)**机制允许一个类将对其非公有成员的访问权授予指定的函数或类。友元的声明以关键字 friend 开始。它只能出现在类定义的内部。友元声明可以出现在类中的任何地方：友元不是授予友元关系的那个类的成员，所以它们不受声明出现部分的访问控制影响。

通常，将友元声明成组地放在类定义的开始或结尾是个好主意。

#### 友元关系：一个例子

想像一下，除了 Screen 类之外，还有一个窗口管理器，管理给定显示器上的一组 Screen。窗口管理类在逻辑上可能需要访问由其管理的 Screen 对象的内部数据。假定 Window\_Mgr 是该窗口管理类的名字，Screen 应该允许 Window\_Mgr 像下面这样访问其成员：

class Screen {

// *Window\_Mgr members can access private parts of class Screen*

friend class Window\_Mgr;

// *...restofthe* *Screen* *class*

};

友元可以是普通的非成员函数，或前面定义的其他类的成员函数，或整个类。将一个类设为友元，友元类的所有成员函数都可以访问授予友元关系的那个类的非公有成员。

#### 使其他类的成员函数成为友元

如果不是将整个 Window\_Mgr 类设为友元，Screen 就可以指定只允许 relocate 成员访问：

class Screen {

// *Window\_Mgrmust* *be defined before class* *Screen*

friend Window\_Mgr&

Window\_Mgr::relocate(Window\_Mgr::index,

Window\_Mgr::index,

Screen&);

// *...restofthe* *Screen* *class*

};

当我们将成员函数声明为友元时，函数名必须用该函数所属的类名字加以限定。

### 12.6. static 类成员

#### 定义 static 成员

在成员声明前加上关键字 static 将成员设为 static。static 成员遵循正常的公有／私有访问规则。

例如，考虑一个简单的表示银行账户的类。每个账户具有余额和拥有者，并且按月获得利息，但应用于每个账户的利率总是相同的。可以按下面的这样编写这个类

class Account {

public:

// *interface functions here*

void applyint() { amount += amount \* interestRate; }

static double rate() { return interestRate; }

static void rate(double); // *sets a new rate*

private:

std::string owner;

double amount;

static double interestRate;

static double initRate();

};

这个类的每个对象具有两个数据成员：owner 和 amount。对象没有与 static 数据成员对应的数据成员，但是，存在一个单独的 interestRate 对象，由 Account 类型的全体对象共享。

#### 12.6.1. static 成员函数

Account 类有两个名为 rate 的 static 成员函数，其中一个定义在类的内部。当我们在类的外部定义 static 成员时，无须重复指定 static 保留字，该保留字只出现在类定义体内部的声明处：

void Account::rate(double newRate)

{

interestRate = newRate;

}

##### static 函数没有 this 指针

static 成员是类的组成部分但不是任何对象的组成部分，因此，static 成员函数没有 this 指针。通过使用非 static 成员显式或隐式地引用 this 是一个编译时错误。

因为 static 成员不是任何对象的组成部分，所以 static 成员函数不能被声明为 const。毕竟，将成员函数声明为 const 就是承诺不会修改该函数所属的对象。最后，static 成员函数也不能被声明为虚函数。我们将在[第 15.2.4 节](http://manual.51yip.com/c++/0201721481/ch15lev1sec2.html" \l "ch15lev2sec6)学习虚函数。

#### 12.6.2. static 数据成员

static 数据成员必须在类定义体的外部定义（正好一次）。不像普通数据成员，static 成员不是通过类构造函数进行初始化，而是应该在定义时进行初始化。

可以定义如下 interestRate：

// *define and initialize static class member*

double Account::interestRate = initRate();

像使用任意的类成员一样，在类定义体外部引用类的 static 成员时，必须指定成员是在哪个类中定义的。然而，static 关键字只能用于类定义体内部的声明中，定义不能标示为 static。

##### 特殊的整型 const static 成员

一般而言，类的 static 成员，像普通数据成员一样，不能在类的定义体中初始化。相反，static 数据成员通常在定义时才初始化。

这个规则的一个例外是，只要初始化式是一个常量表达式，整型 const static 数据成员就可以在类的定义体中进行初始化：

class Account {

public:

static double rate() { return interestRate; }

static void rate(double); // *sets a new rate*

private:

static const int period = 30; // *interest posted every 30 days*

double daily\_tbl[period]; // *ok:* *period* *is constant expression*

};

const static 数据成员在类的定义体中初始化时，该数据成员仍必须在类的定义体之外进行定义。

在类内部提供初始化式时，成员的定义不必再指定初始值：

// *definition of static member with no initializer;*

// *the initial value is specified inside the class definition*

const int Account::period;

##### static 成员不是类对象的组成部分

普通成员都是给定类的每个对象的组成部分。static 成员独立于任何对象而存在，不是类类型对象的组成部分。因为 static 数据成员不是任何对象的组成部分，所以它们的使用方式对于非 static 数据成员而言是不合法的。

例如，static 数据成员的类型可以是该成员所属的类类型。非 static 成员被限定声明为其自身类对象的指针或引用：

class Bar {

public:

// ...

private:

static Bar mem1; // *ok*

Bar \*mem2; // *ok*

Bar mem3; // *error*

};

类似地，static 数据成员可用作默认实参：

class Screen {

public:

// *bkground refers to the static member*

// *declared later in the class definition*

Screen& clear(char = bkground);

private:

static const char bkground = '#';

};

非 static 数据成员不能用作默认实参，因为它的值不能独立于所属的对象而使用。使用非 static 数据成员作默认实参，将无法提供对象以获取该成员的值，因而是错误的。

## 第十三章 复制控制

### 13.1. 复制构造函数

只有单个形参，而且该形参是对本类类型对象的引用（常用 const 修饰），这样的构造函数称为复制构造函数。

#### 对象的定义形式

当用于类类型对象时，初始化的复制形式和直接形式有所不同：直接初始化直接调用与实参匹配的构造函数，复制初始化总是调用复制构造函数。(复制初始化使用 = 符号，而直接初始化将初始化式放在圆括号中。)

string null\_book = "9-999-99999-9"; // *copy-initialization*

string dots(10, '.'); // *direct-initialization*

string empty\_copy = string(); // *copy-initialization*

string empty\_direct; // *direct-initialization*

通常直接初始化和复制初始化仅在低级别上存在差异。然而，对于不支持复制的类型，或者使用非[explicit 构造函数](#_抑制由构造函数定义的隐式转换)（[第 12.4.4 节](http://manual.51yip.com/c++/0201721481/ch12lev1sec4.html" \l "ch12lev2sec23)）的进修，它们有本质区别：

ifstream file1("filename"); // *ok: direct initialization*

ifstream file2 = "filename"; // *error: copy constructor is* *private*

// *This initialization is okay only if*

// *the* *Sales\_item(const string&)* *constructor is not* *explicit*

Sales\_item item = string("9-999-99999-9");

file1 的初始化是正确的。ifstream 类定义了一个可用 C 风格字符串调用的构造函数，使用该构造函数初始化 file1。

看上去等效的 file2 初始化使用复制初始化，但该定义不正确。由于不能复制 IO 类型的对象（[第 8.1 节](http://manual.51yip.com/c++/0201721481/ch08lev1sec1.html" \l "ch08lev1sec1)），所以不能对那些类型的对象使用复制初始化。

item 的初始化是否正确，取决于正在使用哪个版本的 [Sales\_item](#_抑制由构造函数定义的隐式转换) 类。某些版本将参数为一个 string 的构造函数定义为 explicit。如果构造函数是显式的，则初始化失败；如果构造函数不是显式的，则初始化成功。

### 13.3. 析构函数

构造函数的一个用途是自动获取资源。例如，构造函数可以分配一个缓冲区或打开一个文件，在构造函数中分配了资源之后，需要一个对应操作自动回收或释放资源。析构函数就是这样的一个特殊函数，它可以完成所需的资源回收，作为类构造函数的补充。

#### 何时编写显式析构函数

许多类不需要显式析构函数，尤其是具有构造函数的类不一定需要定义自己的析构函数。仅在有些工作需要析构函数完成时，才需要析构函数。析构函数通常用于释放在构造函数或在对象生命期内获取的资源。

如果类需要析构函数，则它也需要赋值操作符和复制构造函数，这是一个有用的经验法则。这个规则常称为**[三法则](http://manual.51yip.com/c++/0201721481/ch13lev1sec7.html" \l "gloss13_09)**，指的是如果需要析构函数，则需要所有这三个复制控制成员。

析构函数并不仅限于用来释放资源。一般而言，析构函数可以执行任意操作，该操作是类设计者希望在该类对象的使用完毕之后执行的。

#### 如何编写析构函数

Sales\_item 类是类没有分配资源因此不需要自己的析构函数的一个例子。分配了资源的类一般需要定义析构函数以释放那些资源。析构函数是个成员函数，它的名字是在类名字之前加上一个代字号（~），它没有返回值，没有形参。因为不能指定任何形参，所以不能重载析构函数。虽然可以为一个类定义多个构造函数，但只能提供一个析构函数，应用于类的所有对象。

**第十四章 重载操作符与转换**

# 第四部分：面向对象编程与泛型编程

## 第十五章 面向对象编程(OOP)

面向对象编程(Object-oriented programming)基于三个基本概念：数据抽象、继承和动态绑定。在 C++ 中，用类进行数据抽象，用类派生从一个类继承另一个：派生类继承基类的成员。动态绑定使编译器能够在运行时决定是使用基类中定义的函数还是派生类中定义的函数。

### 15.1. 面向对象编程：概述

#### 继承

我们经常称因继承而相关联的类为构成了一个**[继承层次](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_04)**。其中有一个类称为根，所以其他类直接或间接继承根类。

在 C++ 中，基类必须指出希望派生类重写哪些函数，定义为 **[virtual](http://manual.51yip.com/c++/0201721481/ch15lev2sec11.html" \l "gloss15_22)** 的函数是基类期待派生类重新定义的，基类希望派生类继承的函数不能定义为虚函数。

#### 动态绑定

在 C++ 中，通过基类的引用（或指针）调用虚函数时，发生动态绑定。引用（或指针）既可以指向基类对象也可以指向派生类对象，这一事实是动态绑定的关键。用引用（或指针）调用的虚函数在运行时确定，被调用的函数是引用（或指针）所指对象的实际类型所定义的。

### 15.2. 定义基类和派生类

#### 15.2.1. 定义基类

像任意其他类一样，基类也有定义其接口和实现的数据和函数成员。在（非常简化的）书店定价应用程序的例子中，Item\_base 类定义了 book 和 net\_price 函数并且需要存储每本书的 ISBN 和标准价格：

// *Item sold at an undiscounted price*

// *derived classes will define various discount strategies*

class Item\_base {

public:

Item\_base(const std::string &book = "",

double sales\_price = 0.0):

isbn(book), price(sales\_price) { }

std::string book() const { return isbn; }

// *returns total sales price for a specified number of items*

// *derived classes will override and apply different discount algorithms*

virtual double net\_price(std::size\_t n) const

{ return n \* price; }

virtual ~Item\_base() { }

private:

std::string isbn; // *identifier for the item*

protected:

double price; // *normal, undiscounted price*

};

##### 基类成员函数

除了构造函数之外，任意非 static 成员函数都可以是虚函数。保留字只在类内部的成员函数声明中出现，不能用在类定义体外部出现的函数定义上。

基类通常应将派生类需要重定义的任意函数定义为虚函数。

##### 访问控制和继承

在基类中，public 和 private 标号具有普通含义：用户代码可以访问类的 public 成员而不能访问 private 成员，private 成员只能由基类的成员和友元访问。派生类对基类的 public 和 private 成员的访问权限与程序中任意其他部分一样：它可以访问 public 成员而不能访问 private 成员。

有时作为基类的类具有一些成员，它希望允许派生类访问但仍禁止其他用户访问这些成员。对于这样的成员应使用**[受保护的访问标号](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_15)**。protected 成员可以被派生类对象访问但不能被该类型的普通用户访问。

#### 15.2.2. protected 成员

派生类只能通过派生类对象访问其基类的 protected 成员，派生类对其基类类型对象的 protected 成员没有特殊访问权限。

例如，假定 Bulk\_item 定义了一个成员函数，接受一个 Bulk\_item 对象的引用和一个 Item\_base 对象的引用，该函数可以访问自己对象的 protected 成员以及 Bulk\_item 形参的 protected 成员，但是，它不能访问 Item\_base 形参的 protected 成员。

void Bulk\_item::memfcn(const Bulk\_item &d, const Item\_base &b)

{

// *attempt to use* *protected* *member*

double ret = price; // *ok: uses* *this->price*

ret = d.price; // *ok: uses* *price* *from a* *Bulk\_item* *object*

ret = b.price; // *error: no access to* *price* *from an* *Item\_base*

}

d.price 的使用正确，因为是通过 Bulk\_item 类型对象引用 price；b.price 的使用非法，因为对 Item\_item 类型的对象没有特殊访问访问权限。

#### 15.2.3. 派生类

为了定义派生类，使用**[类派生列表](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_03)**指定基类。类派生列表指定了一个或多个基类，具有如下形式：

class *classname: access-label base-class*

这里 *access-label* 是 public、protected 或 private，*base-class* 是已定义的类的名字。类派生列表可以指定多个基类。

##### 定义派生类

在书店应用程序中，将从 Item\_base 类派生 Bulk\_item 类，因此 Bulk\_item 类将继承 book、isbn 和 price 成员。Bulk\_item 类必须重定义 net\_price 函数定义该操作所需要的数据成员：

// *discount kicks in when a specified number of copies of same book are sold*

// *the discount is expressed as a fraction used to reduce the normal price*

class Bulk\_item : public Item\_base {

public:

// *redefines base version so as to implement bulk purchase discount policy*

double net\_price(std::size\_t) const;

private:

std::size\_t min\_qty; // *minimum purchase for discount to apply*

double discount; // *fractional discount to apply*

};

##### 派生类和虚函数

尽管不是必须这样做，派生类一般会重定义所继承的虚函数。派生类没有重定义某个虚函数，则使用基类中定义的版本。

派生类中虚函数的声明（[第 7.4 节](http://manual.51yip.com/c++/0201721481/ch07lev1sec4.html" \l "ch07lev1sec4)）必须与基类中的定义方式完全匹配，但有一个例外：返回对基类型的引用（或指针）的虚函数。派生类中的虚函数可以返回基类函数所返回类型的派生类的引用（或指针）。

例如，Item\_base 类可以定义返回 Item\_base\* 的虚函数，如果这样，Bulk\_item 类中定义的实例可以定义为返回 Item\_base\* 或 Bulk\_item\*。[第 15.9 节](http://manual.51yip.com/c++/0201721481/ch15lev1sec9.html" \l "ch15lev1sec9)将介绍这种虚函数的一个例子。

一旦函数在基类中声明为虚函数，它就一直为虚函数，派生类无法改变该函数为虚函数这一事实。派生类重定义虚函数时，可以使用 virtual 保留字，但不是必须这样做。

##### 用作基类的类必须是已定义的

已定义的类才可以用作基类。如果已经声明了 Item\_base 类，但没有定义它，则不能用 Item\_base 作基类：

class Item\_base; // *declared but not defined*

// *error: Item\_base must be defined*

class Bulk\_item : public Item\_base { ... };

##### 派生类的声明

如果需要声明（但并不实现）一个派生类，则声明包含类名但不包含派生列表。例如，下面的前向声明会导致编译时错误：

// *error: a forward declaration must not include the derivation list*

class Bulk\_item : public Item\_base;

正确的[前向声明](#_前向声明（声明一个类而不定义）)为：

// *forward declarations of both derived and nonderived class*

class Bulk\_item;

class Item\_base;

#### 15.2.4. virtual 与其他成员函数

C++ 中的函数调用默认不使用动态绑定。要触发动态绑定，满足两个条件：第一，只有指定为虚函数的成员函数才能进行动态绑定，成员函数默认为非虚函数，非虚函数不进行动态绑定；第二，必须通过基类类型的引用或指针进行函数调用。

##### 从派生类型到基类的转换

因为每个派生类对象都包含基类部分，所以可将基类类型的引用绑定到派生类对象的基类部分，也可以用指向基类的指针指向派生类对象：

// *function with an* *Item\_base* *reference parameter*

double print\_total(const Item\_base&, size\_t);

Item\_base item; // *object of base type*

// *ok: use pointer or reference to* *Item\_base* *to refer to an* *Item\_base* *object*

print\_total(item, 10); // *passes reference to an* *Item\_base* *object*

Item\_base \*p = &item; // *p* *points to an* *Item\_base* *object*

Bulk\_item bulk; // *object of derived type*

// *ok: can bind a pointer or reference to* *Item\_base* *to a* *Bulk\_item* *object*

print\_total(bulk, 10); // *passes reference to the* *Item\_base* *part of* *bulk*

p = &bulk; // *p* *points to the* *Item\_base* *part of* *bulk*

基类类型引用和指针的关键点在于**[静态类型](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_21)**（在编译时可知的引用类型或指针类型）和**[动态类型](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_07)**（指针或引用所绑定的对象的类型这是仅在运行时可知的）可能不同。

##### 可以在运行时确定 virtual 函数的调用

我们再来看 print\_total 函数，因为 item 形参是一个引用且 net\_price 是虚函数，item.net\_price(n) 所调用的 net\_price 版本取决于在运行时绑定到 item 形参的实参类型：

Item\_base base;

Bulk\_item derived;

// *print\_total* *makes a virtual call to* *net\_price*

print\_total(cout, base, 10); // *calls* *Item\_base::net\_price*

print\_total(cout, derived, 10); // *calls* *Bulk\_item::net\_price*

在第一个调用中，item 形参在运行时绑定到 Item\_base 类型的对象，因此，print\_total 内部调用 Item\_base 中定义的 net\_price 版本。在第二个调用中，item 形参绑定到 Bulk\_item 类型的对象，从 print\_total 调用的是 Bulk\_item 类定义的 net\_price 版本。

##### 覆盖虚函数机制

在某些情况下，希望覆盖虚函数机制并强制函数调用使用虚函数的特定版本，这里可以使用作用域操作符：

Item\_base \*baseP = &derived;

// *calls version from the base class regardless of the dynamic type of* *baseP*

double d = baseP->Item\_base::net\_price(42);

只有成员函数中的代码才应该使用作用域操作符覆盖虚函数机制。

为什么会希望覆盖虚函数机制？最常见的理由是为了派生类虚函数调用基类中的版本。在这种情况下，基类版本可以完成继承层次中所有类型的公共任务，而每个派生类型只添加自己的特殊工作。

例如，可以定义一个具有虚操作的 Camera 类层次。Camera 类中的 display 函数可以显示所有的公共信息，派生类（如 PerspectiveCamera）可能既需要显示公共信息又需要显示自己的独特信息。可以显式调用 Camera 版本以显示公共信息，而不是在 PerspectiveCamera 的 display 实现中复制 Camera 的操作。

派生类虚函数调用基类版本时，*必须*显式使用作用域操作符。如果派生类函数忽略了这样做，则函数调用会在运行时确定并且将是一个自身调用，从而导致无穷递归。

#### 15.2.5. 公用、私有和受保护的继承

基类本身指定对自身成员的最小访问控制。如果成员在基类中为 private，则只有基类和基类的友元可以访问该成员。派生类不能访问基类的 private 成员，也不能使自己的用户能够访问那些成员。如果基类成员为 public 或 protected，则派生列表中使用的访问标号决定该成员在派生类中的访问级别：

* 如果是**[公用继承](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_17)**，基类成员保持自己的访问级别：基类的 public 成员为派生类的 public 成员，基类的 protected 成员为派生类的 protected 成员。
* 如果是**[受保护继承](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_16)**，基类的 public 和 protected 成员在派生类中为 protected 成员。
* 如果是**[私有继承](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_14)**，基类的的所有成员在派生类中为 private 成员。

##### 接口继承与实现继承

public 派生类继承基类的接口，它具有与基类相同的接口，被称为接口继承。设计良好的类层次中，public 派生类的对象可以用在任何需要基类对象的地方。

使用 private 或 protected 派生的类不继承基类的接口，相反，这些派生通常被称为实现继承。派生类在实现中使用被继承但继承基类的部分并未成为其接口的一部分。

##### 去除个别成员

如果进行 private 或 protected 继承，则基类成员的访问级别在派生类中比在基类中更受限：

class Base {

public:

std::size\_t size() const { return n; }

protected:

std::size\_t n;

};

class Derived : private Base { . . . };

在这一继承层次中，size 在 Base 中为 public，但在 Derived 中为 private。为了使 size 在 Derived 中成为 public，可以在 Derived 的 public 部分增加一个 using 声明。如下这样改变 Derived 的定义，可以使 size 成员能够被用户访问，并使 n 能够被从 Derived 派生的类访问：

class Derived : private Base {

public:

// *maintain access levels for members related to the size of the object*

using Base::size;

protected:

using Base::n;

// ...

};

##### 默认继承保护级别

在[第 2.8 节](http://manual.51yip.com/c++/0201721481/ch02lev1sec8.html" \l "ch02lev1sec8)介绍过用 struct 和 class 保留字定义的类具有不同的默认访问级别，同样，默认继承访问级别根据使用哪个保留字定义派生类也不相同。使用 class 保留字定义的派生默认具有 private 继承，而用 struct 保留字定义的类默认具有 public 继承：

有一种常见的误解认为用 struct 保留字定义的类与用 class 定义的类有更大的区别。唯一的不同只是默认的成员保护级别和默认的派生保护级别，没有其他区别：

#### 15.2.6. 友元关系与继承

友元关系不能继承。基类的友元对派生类的成员没有特殊访问权限。如果基类被授予友元关系，则只有基类具有特殊访问权限，该基类的派生类不能访问授予友元关系的类。

每个类控制对自己的成员的友元关系。

如果派生类想要将自己成员的访问权授予其基类的友元，派生类必须显式地这样做：基类的友元对从该基类派生的类型没有特殊访问权限。同样，如果基类和派生类都需要访问另一个类，那个类必须特地将访问权限授予基类的和每一个派生类。

#### 15.2.7. 继承与静态成员

如果基类定义 static 成员（[第 12.6 节](http://manual.51yip.com/c++/0201721481/ch12lev1sec6.html" \l "ch12lev1sec6)），则整个继承层次中只有一个这样的成员。无论从基类派生出多少个派生类，每个 static 成员只有一个实例。static 成员遵循常规访问控制：如果成员在基类中为 private，则派生类不能访问它。假定可以访问成员，则既可以通过基类访问 static 成员，也可以通过派生类访问 static 成员。一般而言，既可以使用作用域操作符也可以使用点或箭头成员访问操作符。

struct Base {

static void statmem(); // *public* *by default*

};

struct Derived : Base {

void f(const Derived&);

};

void Derived::f(const Derived &derived\_obj)

{

Base::statmem(); // *ok:* *Base* *defines* *statmem*

Derived::statmem(); // *ok:* *Derived* *in herits* *statmem*

// *ok: derived objects can be used to access static from base*

derived\_obj.statmem(); // *accessed through* *Derived* *object*

statmem(); // *accessed through this class*

### 15.3. 转换与继承(…)

每个派生类对象包含一个基类部分，这意味着可以像使用基类对象一样在派生类对象上执行操作。因为派生类对象也是基类对象，所以存在从派生类型引用到基类类型引用的自动转换，即，可以将派生类对象的引用转换为基类子对象的引用，对指针也类似。(派生类是由基类继承而来，含有基类部分)

基类类型对象既可以作为独立对象存在，也可以作为派生类对象的一部分而存在，因此，一个基类对象可能是也可能不是一个派生类对象的部分，结果，没有从基类引用（或基类指针）到派生类引用（或派生类指针）的（自动）转换。（基类没有派生类新增加的部分）

### 15.6. 纯虚函数

函数形参表后面写上 = 0 以指定纯虚函数：

class Disc\_item : public Item\_base {

public:

double net\_price(std::size\_t) const = 0;

};

将函数定义为纯虚能够说明，该函数为后代类型提供了可以覆盖的接口，但是这个类中的版本决不会调用。重要的是，用户将不能创建 Disc\_item 类型的对象。

含有（或继承）一个或多个纯虚函数的类是**[抽象基类](http://manual.51yip.com/c++/0201721481/ch15lev1sec11.html" \l "gloss15_01)**。除了作为抽象基类的派生类的对象的组成部分，不能创建抽象类型的对象。

# 第五部分　高级主题

## 第十七章　用于大型程序的工具

### 17.2. 命名空间

在使用来自多个供应商的库编写应用程序的时候，这些名字中有一些几乎不可避免地会发生冲突，这种名字冲突问题称为**[命名空间污染](http://manual.51yip.com/c++/0201721481/ch17lev1sec5.html" \l "gloss17_19)**问题。

传统上，程序员通过将全局实体的名字设得很长来避免命名空间污染，经常用特定字符序列作为程序中名字的前缀：

class cplusplus\_primer\_Query { ... };

ifstream&

cplusplus\_primer\_open\_file(ifstream&, const string&);

**[命名空间](http://manual.51yip.com/c++/0201721481/ch17lev1sec5.html" \l "gloss17_17)**为防止名字冲突提供了更加可控的机制，命名空间能够划分全局命名空间，这样使用独立开发的库就更加容易了。一个命名空间是一个作用域，通过在命名空间内部定义库中的名字，库的作者（以及用户）可以避免全局名字固有的限制。