第 21 章　杂项内容

首先祝贺你！你就要完成全面介绍 C++的 3 周课程了。现在，你对 C++已有深入了解，但现代编程中，总是有更多的知识需要学习。本章补充一些遗漏的细节，然后为继续学习提供一些建议。

你在源代码文件中编写的大部分代码都是 C++。它由编译器编译并生成可执行程序。然而，在编译器运行之前将运行预处理器，这提供了条件编译的机会。

本章介绍以下内容：

- 什么是条件编译？如何管理条件编译？
- 如何使用编译指令编写宏？
- 如何在查找错误时使用预处理器？
- 如何操纵位以及将它们用作标记？
- 学习高效使用 C++的后续步骤。

21.1　预处理器和编译器

每次运行编译器时，预处理器都将首先运行。预处理器查找预处理器指令，每条预处理器指令都以#打头。这些指令的作用是修改源代码的文本。结果为一个新的源代码文件：一个通常看不到的临时文件，但你可以命令编译器保存它，以使在需要时查看。

编译器不是读取原始的源代码文件，而是读取预处理器的输出并对其进行编译。读者已经通过使用编译指令#include 看到了这种效果。#include 命令预处理器查找其名称位于#include 后面的文件，并将其写入到中间文件这个位置。这就像你将整个文件的内容输入到了源代码中一样，编译器看到源代码时，包含的文件内容已经在源代码中了。

提示：几乎每个编译器都有一个可在集成开发环境（IDE）或命令行中设置的开关，用于指示编译器保存中间文件。如果要查看中间文件，可参阅编译器手册来了解需要为编译器设置哪个开关。

21.2　预处理器指令#define

可使用命令#define 来定义字符串替换，下面的代码指示预编译器将所有的“BIG”替换为 512：

```
#define BIG 512
```

这不是 C++意义上的字符串。源代码中所有的“BIG”都将被替换为“512”。对于如下源代码：

```
#define BIG 512
int myArray[BIG];
```

经预编译器处理后将变成：

```
int myArray[512];
```

#define 语句不见了。在中间文件中，预编译器语句都将被删除，它们根本不会出现在最终的源代码中。

21.2.1 使用#define 来定义常量

#define 用途之一是定义常量。然而，由于#define 只是进行字符串替换而不做类型检查，因此这几乎不是什么好主意。正如在介绍常量时指出的，相对于使用#define，使用关键字 const 有很多优点。

21.2.2 将#define 用于检测

#define 的另一种用途是，指出某个字符串定义过。可以编写这样的代码：

```
#define DEBUG
```

然后在程序中检测 DEBUG 是否被定义，并采取相应的措施。要检查字符串是否被定义过，可使用预处理器命令#if 和命令 defined：

```
#if defined DEBUG
cout << "Debug defined";
#endif
```

如果检测的字符串（这里为 DEBUG）已定义过，则 defined 表达式的结果为 true。别忘了，这发生在预处理器中，而不是编译器中或执行程序时。

预处理器遇到#if defined 时，将检查一个已创建的表，看#if defined 后面的值是否定义过。如果定义了，则 defined 表达式的结果为 true，#if defined DEBUG 和#endif 之间的所有内容都将被写入中间文件进行编译；如果为 false，#if definde DEBUG 和#endif 之间的所有内容都不会被写入中间文件，就好像它们本来就不在源代码中一样。

还有一个简化的编译指令可用于检测某个值是否被定义，这就是#ifdef：

```
#ifdef DEBUG
cout << "Debug defined";
#endif
```

还可以检测某个值是否未被定义，为此可将运算符!用于编译指令 defined：

```
#if !defined DEBUG
cout << "Debug is not defined";
#endif
```

也可使用其简化版本#ifndef：

```
#ifndef DEBUG
cout << "Debug is not defined.";
#endif
```

注意，#ifndef 是#ifdef 的逻辑逆，如果在此之前没有定义被检测的字符串，则#ifndef 的结果为 true。读者应该注意到了，这些检测都需要使用#enddif 来指定受检测影响的代码到什么地方结束。

21.2.3 预编译器命令#else

读者可能想到了，可在#ifdef（或#ifndef）和#endif 之间使用编译指令#else，程序清单 21.1 演示了如何使用这些编译指令。

程序清单 21.1 使用#define

```
0:  #define DemoVersion
1:  #define SW_VERSION 5
```

```
 2: #include <iostream>
 3:
 4: using std::endl;
 5: using std::cout;
 6:
 7: int main()
 8: {
 9:    cout << "Checking on the definitions of DemoVersion,";
10:    cout << "SW_VERSION, and WINDOWS_VERSION..." << endl;
11:
12:    #ifdef DemoVersion
13:       cout << "DemoVersion defined." << endl;
14:    #else
15:       cout << "DemoVersion not defined." << endl;
16:    #endif
17:
18:    #ifndef SW_VERSION
19:       cout << "SW_VERSION not defined!" << endl;
20:    #else
21:       cout << "SW_VERSION defined as: "
22:            << SW_VERSION << endl;
23:    #endif
24:
25:    #ifdef WINDOWS_VERSION
26:       cout << "WINDOWS_VERSION defined!" << endl;
27:    #else
28:       cout << "WINDOWS_VERSION was not defined." << endl;
29:    #endif
30:
31:    cout << "Done."  << endl;
32:    return 0;
33: }
```

输出：

```
Checking on the definitions of DemoVersion, NT VERSION, and
WINDOWS_VERSION...
DemoVersion defined.
NT_VERSION defined as: 5
WINDOWS_VERSION was not defined.
Done.
```

分析：

第 0 行和第 1 行定义了 DemoVersion 和 SW_VERSION，其中后者被定义为 5。第 12 行检测 DemoVersion 是否已定义。由于 DemoVersion 已定义（虽然没有值），因此检测结果为 true，执行第 13 行的代码。

第 18 行检测 SW_VERSION 是否没有定义。由于 SW_VERSION 已定义，因此检测结果为 false，跳到第 21 行执行。在这里，SW_VERSION 已被替换为 5，编译器看到的代码如下：

```
cout << "SW_VERSION defined as: " << 5 << endl;
```

注意，第一个 SW_VERSION 没有被替换，因为它位于用引号括起的字符串中；然而，第二个 SW_VERSION 被替换，因此编译器看到的是 5，就像在这里输入了 5 一样。

最后在 25 行检测 WINDOWS_VERSION 是否已定义。由于没有定义 WINDOWS_VERSION，因此检测

结果为 false，执行第 28 行打印一条消息。

21.3 包含和多重包含防范

创建项目时将使用很多不同的文件。可能需要组织目录，使每个类都有包含类声明的头文件（如.hpp）和包含类方法源代码的实现文件（如.cpp）。

main( )函数保存在一个独立的.cpp 文件中，所有.cpp 文件都被编译为.obj 文件，然后由链接器将它们链接成一个程序。

由于程序使用了很多类的方法，因此很多头文件都将被包含到每个文件中。另外，头文件经常需要包含其他头文件。例如，派生类的头文件必须包含基类的头文件。

假设 Animal 类是在文件 ANIMAL.hpp 中声明的。在从 Animal 类派生而来的 Dog 类的头文件 DOG.hpp 中，必须包含头文件 ANIMAL.hpp，否则将不能从 Animal 派生出 Dog。鉴于同样的原因，Cat 类的头文件也必须包含 ANIMAL.hpp。

如果创建使用了 Cat 和 Dog 的程序时，可能将 ANIMAL.hpp 包含两次。这将导致编译错误，因为不能将同一个类（Animal）声明两次，虽然这两次声明是相同的。

为解决这种问题，可使用多重包含防范（inclusion guard）。在 ANIMAL 的头文件开头，添加如下代码行：

```
#ifndef ANIMAL_HPP
#define ANIMAL_HPP
...                    // the whole file goes here
#endif
```

上述代码的含义是，如果没有定义 ANIMAL_HPP，则现在定义它。在#define 语句和#endif 语句之间是整个文件的内容。

程序首次包含该文件时，它读取第 1 行，检测结果为 true：还没有定义 ANIMAL_HPP，因此定义它并包含整个文件。

程序第二次包含文件 ANIMAL.HPP 时，它读取第 1 行，检测结果为 false，因为已经包含了 ANIMAL.HPP。因此预处理器不处理下一个#else（在这个例子中没有）或#endif（在文件末尾）之前的所有代码。这样就跳过了整个文件的内容，类不会被声明两次。

符号的实际名称（ANIMAL_HPP）并不重要，虽然习惯上使用全部大写的文件名，并将句点（.）改为下划线。然而，这纯粹是一种约定，由于两个文件不能同名，因此这种约定是可行的。

注意：使用多重包含防范不会有任何害处。使用它常常可以节省大量的调试时间。

21.4 宏

编译指令#define 也可以用于创建宏。宏是使用#define 创建的符号，它像函数那样能够接受参数。预处理器将用指定的参数值替换宏中的替换字符串。例如，如果将宏 TWICE 定义为：

```
#define TWICE(x) ( (x) * 2)
```

然后程序中包含如下代码：

```
TWICE(4)
```

则预处理器将把 TWICE(4)替换为((4)*2)，结果为 8。

宏可以接受多个参数，每个参数都可在替换文本中多次出现。两个常见的宏是 MAX 和 MIN：

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
```

```
#define MIN(x,y) ( (x) < (y) ? (x) : (y) )
```

注意，在宏的定义中，参数列表的左括号必须紧跟在宏名后面，中间不能有空格。预处理器不像编译器那样允许使用空格。如果有空格，将像本章前面介绍的那样使用标准替换。

例如，如果编写了如下宏：

```
#define MAX (x,y) ( (x) > (y) ? (x) : (y) )
```

然后像下面这样使用 MAX：

```
int x = 5, y = 7, z;
z = MAX(x,y);
```

中间代码将为：

```
int x = 5, y = 7, z;
z = (x,y) ( (x) > (y) ? (x) : (y) )(x,y)
```

执行了简单的文本替换，而不是调用宏函数。因此，符号 MAX 被替换为(x，y)((x) > (y)?(x):(y))，然后是 MAX 后面的(x，y)。

如果将 MAX 和(x，y)之间的空格删除，中间代码将为：

```
int x = 5, y = 7, z;
z = ( (5) > (7) ? (5) : (7) );
```

为什么要使用括号

你可能会问，为什么前面介绍的很多宏中包含那么多括号。预处理器并不要求在替换字符串中用括号将参数括起，但将复杂的值传递给宏时，括号可帮助避免副作用。例如，如果将 MAX 定义为：

```
#define MAX(x,y) x > y ? x : y
```

然后调用该宏并将 5 和 7 传递给它，它将按期望的方式运行。然而，如果传递的是更复杂的表达式，将不能得到预期的结果，如程序清单 21.2 所示。

程序清单 21.2　在宏中使用括号

```
 0: // Listing 21.2 Macro Expansion
 1: #include <iostream>
 2: using namespace std;
 3:
 4: #define CUBE(a) ( (a) * (a) * (a) )
 5: #define THREE(a) a * a * a
 6:
 7: int main()
 8: {
 9:    long x = 5;
10:    long y = CUBE(x);
11:    long z = THREE(x);
12:
13:    cout << "y: " << y << endl;
14:    cout << "z: " << z << endl;
15:
16:    long a = 5, b = 7;
17:    y = CUBE(a+b);
18:    z = THREE(a+b);
19:
20:    cout << "y: " << y << endl;
```

```
21:     cout << "z: " << z << endl;
22:     return 0;
23: }
```

输出：

```
y: 125
z: 125
y: 1728
z: 82
```

分析：

第 4 行定义了宏 CUBE，其中在每次使用参数 a 时都将其用括号括起。第 5 行定义了宏 THREE，但没有括号。

在第 10 和 11 行首次调用这两个宏时，传递的参数值为 5，这两个宏都运行正常。CUBE(5) 展开为((5)*(5)*(5))，结果为 125；THREE(5) 展开为 5*5*5，结果也是 125。

在第 16～18 行第二次调用它们时，参数为 5+7。在这种情况下，CUBE(5+7) 展开为((5+7) * (5+7) * (5+7))，这相当于((12) * (12) * (12))，结果为 1728。而 THREE(5+7) 展开为 5 + 7 * 5 + 7 * 5 + 7。由于乘法的优先级高于加法，因此这相当于 5 + (7 * 5) + (7 * 5) + 7，即 5 + (35) + (35) + 7，结果为 82。

21.5 字符串操纵

预处理器提供了两个特殊的运算符，用于在宏中操纵字符串。字符串化运算符（#）将其后面的内容转换为用引号括起的字符串；拼接运算符将两个字符串合并成一个。

21.5.1 字符串化

字符串化运算符将其后面到下一个空格为止的所有字符用引号括起。如果编写了如下宏：

```
#define WRITESTRING(x) cout << #x
```

然后这样调用它：

```
WRITESTRING(This is a string);
```

预编译器将把它转换为：

```
cout << "This is a string";
```

注意，按 cout 的要求将字符串 This is a string 用引号括起。

21.5.2 拼接

拼接运算符让你能够将多个单词合并成一个新词。该新词实际上是一个符号，可用作类名、变量名、数组下标或出现任何可使用字符串的地方。

假设有 5 个函数，分别名为 fOnePrint、fTwoPrint、fThreePrint、fFourPrint 和 fFivePrint。可以这样声明一个宏：

```
#define fPRINT(x) f ## x ## Print
```

然后，使用 fPRINT(Two)来生成 fTwoPrint，使用 fPRINT(Three)来生成 fThreePrint。

在第二周间顾中，开发了一个 PartsList 类。这个链表只能处理 Part 对象。假设该链表能正常工作，而你希望能够创建动物链表、汽车链表、计算机链表等。

一种方法是创建 AnimalList、CarList、ComputerList 等类，复制并粘贴 PartsList 类的代码。这将变成一场恶梦，因为对一个链表做任何修改，都必须在其他链表中做相应的修改。

另一种方法是使用宏和拼接运算符。例如，可以这样定义：

```
#define Listof(Type)  class Type##List \
{ \
public: \
Type##List(){} \
private:        \
int itsLength; \
};
```

这个例子极其简单，但其思想是添加所有必要的方法和数据。如果要声明 AnimalList，可以编写下述代码：

```
Listof(Animal)
```

这将变成 AnimalList 类的声明。这种解决方法也存在一些问题，第 19 章详细讨论过这些问题。

21.6　预定义的宏

很多编译器预定义了大量有用的宏，其中包括__DATE__、__TIME__、__LINE__和__FILE__。每个宏名都以两个下划线字符开头和结尾，以降低这些宏名与程序员在程序中使用的名称发生冲突的可能性。

预编译器看到这些宏时，将执行合适的替换。对于__DATE__，替换为当前日期；对于__TIME__，替换为当前时间；对于__LINE__和__FILE__，分别替换为源代码行数和文件名。需要注意的是，这些替换是在预编译源代码时进行的，而不是在程序运行时进行的。如果要求程序打印__DATE__，打印的将不是当前日期，而是程序被编译时的日期。这些预定义的宏在调试中是非常有用的。

21.7　assert( )宏

很多编译器都提供了一个 assert( )宏。参数的值为 true 时，assert( )返回 true；否则执行某种操作。很多编译器在 assert( ) 的参数值为 false 时中止程序，其他编译器则引发异常（见第 20 章）。

assert( )宏用于在发布程序前对其进行调试。事实上，如果 DEBUG 没有定义，预处理器将简化 assert( )，使得其中的任何代码都不会出现在为编译器生成的源代码中。这在开发过程中很有帮助，发布最终产品时，assert( )不会影响性能，也不增加程序可执行版本的大小。

也可以使用编译器提供的 assert( )，而编写自己的 assert( )宏。程序清单 21.3 提供了一个简单的自定义 assert( )宏并演示了其用法。

程序清单 21.3　一个简单的 assert( )宏

```
0:  // Listing 21.3 ASSERTS
1:  #define DEBUG
2:  #include <iostream>
3:  using namespace std;
4:
5:  #ifndef DEBUG
6:     #define ASSERT(x)
7:  #else
8:     #define ASSERT(x) \
9:           if (! (x)) \
10:          { \
```

```
11:            cout << "ERROR!! Assert " << #x << " failed << endl; \
12:            cout << " on line " << __LINE__ << endl; \
13:            cout << " in file " << __FILE__ << endl;  \
14:         }
15: #endif
16:
17: int main()
18: {
19:    int x = 5;
20:    cout << "First assert: " << endl;
21:    ASSERT(x==5);
22:    cout << "\nSecond assert: " << endl;
23:    ASSERT(x != 5);
24:    cout << "\nDone. << endl";
25:    return 0;
26: }
```

输出：

```
First assert:

Second assert:
ERROR!! Assert x !=5 failed
  on line 24
  in file List2104.cpp

Done.
```

分析：

第 1 行定义了 DEBUG。通常，这是编译阶段通过命令行（或 IDE）完成的，这样可以根据需要打开或关闭它。第 8～14 行定义了 ASSERT( )宏。通常，这是在头文件中完成的，在所有实现文件中都应包含该头文件（ASSERT.hpp）。

第 5 行检测 DEBUG 是否已定义。如果 DEBUG 没有定义，ASSERT( )被定义为不创建任何代码。如果定义了 DEBUG，则应用第 8～14 行定义的功能。

对预编译器来说，ASSERT( )本身是一条很长的语句，它跨越 7 行。第 9 行检测通过参数传入的值，如果为 false，则执行第 11～13 行的语句：打印一条错误消息；如果传入的值为 true，则不执行任何操作。

21.7.1 使用 assert( )进行调试

编写程序时，通常知道某些事情是真的：函数有特定的值、指针是有效的等。Bug 的本质是，在某些情况下本为真的事实确是假的。例如，你知道某个指针是有效的，但程序却崩溃了。assert( )可帮助你发现这类 bug，但仅当你养成在代码中大量使用 assert( )的习惯时才能如此。每当你给指针赋值、将其作为函数参数或返回值时，务必确定该指针是有效的。每当代码依赖于某个变量包含的特定值时，都应使用 assert( )来确认这一点。

频繁使用 assert( )没有坏处。解除对 DEBUG 的定义后，assert( )将从代码中删除。它还提供了优秀的内部文档，提醒阅读者在程序执行的某个时刻，你认为是真的事实。

21.7.2 assert( )与异常之比较

前一章介绍了如何使用异常处理错误状态。需要指出的是，assert( )并非为处理运行阶段的错误状态（如输入无效、内存不足、不能打开文件等）而设计的，而是为捕获编程错误而设计的。也就是说，如果 assert( )“开火了”，说明代码中有 bug。

这非常重要，因为将代码交付给用户时，assert( )实例将被删除。不能依靠 assert( )来处理运行阶段的问题，因为此时 assert( )已不存在了。

一种常见的错误是，使用 assert( )来测试内存分配的返回值：

```
Animal *pCat = new Cat;
Assert(pCat); // bad use of assert
pCat->SomeFunction();
```

这是一种经典的编程错误：每当程序员运行程序时，有足够的内存可用，assert( )从不“开火”。毕竟，为提高编译器、调试器等的速度，程序员在安装了大量 RAM 的机器上运行程序。程序员然后发布可执行文件，而可怜的用户没有那么多内存，当执行到这部分时，调用 new 失败并返回 NULL。然而，代码中不再有 assert( )，没有任何东西指出指针为 NULL。执行到语句 pCat->SomeFounction( )后，程序将崩溃。

虽然内存分配返回 NULL 是一种异常情形，但并不是编程错误。程序只需引发异常就能够恢复。请切记：DEBUG 未被定义时，assert( )语句将消失。异常在第 20 章详细介绍过。

21.7.3　副作用

仅当 assert( )实例被删除后才出现的 bug 很常见。这几乎总是由于程序无意中依赖了 assert( )和其他调试代码的副作用引起的。例如，如果本想检测 x 是否等于 5 时编写了如下代码：

```
ASSERT (x=5)
```

这将引入一个非常难以发现的 bug。

假设在这个 assert( )前，你调用了将 x 设置为 0 的函数。你以为该 assert( )检测 x 是否等于 5，而实际上却将 x 设置为 5。测试结果为 true，因为 x=5 不仅将 x 设置为 5，还返回 5，由于 5 不等于零，因此为 true。

执行完该 assert( )语句后，x 确实等于 5（刚设置的）。程序运行得很好，你打算交付，于是关闭调试。现在 assert( )消失了，不再将 x 设置为 5。由于在此之前 x 刚刚被设置为 0，因此它仍然为 0，程序中断。

倍感挫折后，你又打开调试，但就像变魔术一样，bug 不见了。这看上去很有趣，但还是经不住考验，因此要当心调试代码的副作用。如果发现仅在调试关闭后才出现的 bug，应查看调试代码，注意讨厌的副作用。

21.7.4　类的不变量

大多数类都有一些条件，每当执行完成员函数后，这些条件都将满足。这些类不变量是类的要素。例如，Circle 对象的半径不能为 0，Animal 的年龄总是在 0 到 100 之间。

声明一个这样的 Invariants( )方法可能会很有帮助：它仅在所有这些条件都为 true 时才返回 true。这样，可以在调用每个类方法之前和之后使用 ASSERT(Invariants( ))。例外情况是，在构造函数执行前和析构函数结束后，Invariants( )将不会返回 true。程序清单 21.4 演示了如何在一个小型类中使用 Invariants( )。

程序清单 21.4　使用 Invariants( )

```
 0:  #define DEBUG
 1:  #define SHOW_INVARIANTS
 2:  #include <iostream>
 3:  #include <string.h>
 4:  using namespace std;
 5:
 6:  #ifndef DEBUG
 7:     #define ASSERT(x)
 8:  #else
 9:     #define ASSERT(x) \
10:           if (! (x)) \
11:           { \
```

```
12:           cout << "ERROR!! Assert " << #x << " failed" << endl; \
13:           cout << " on line " << __LINE__ << endl; \
14:           cout << " in file " << __FILE__ << endl;  \
15:         }
16: #endif
17:
18:
19: const int FALSE = 0;
20: const int TRUE = 1;
21: typedef int BOOL;
22:
23:
24: class String
25: {
26:   public:
27:     // constructors
28:     String();
29:     String(const char *const);
30:     String(const String &);
31:     ~String();
32:
33:     char & operator[](int offset);
34:     char operator[](int offset) const;
35:
36:     String & operator= (const String &);
37:     int GetLen()const { return itsLen; }
38:     const char * GetString() const { return itsString; }
39:     BOOL Invariants() const;
40:
41:   private:
42:     String (int);        // private constructor
43:     char * itsString;
44:     // unsigned short itsLen;
45:     int itsLen;
46: };
47:
48: // default constructor creates string of 0 bytes
49: String::String()
50: {
51:   itsString = new char[1];
52:   itsString[0] = '\0';
53:   itsLen=0;
54:   ASSERT(Invariants());
55: }
56:
57: // private (helper) constructor, used only by
58: // class methods for creating a new string of
59: // required size.  Null filled.
60: String::String(int len)
61: {
62:   itsString = new char[len+1];
63:   for (int i = 0; i <= len; i++)
```

```
64:       itsString[i] = '\0';
65:    itsLen=len;
66:    ASSERT(Invariants());
67: }
68:
69: // Converts a character array to a String
70: String::String(const char * const cString)
71: {
72:    itsLen = strlen(cString);
73:    itsString = new char[itsLen+1];
74:    for (int i = 0; i < itsLen; i++)
75:       itsString[i] = cString[i];
76:       itsString[itsLen] = '\0';
77:    ASSERT(Invariants());
78: }
79:
80: // copy constructor
81: String::String (const String & rhs)
82: {
83:    itsLen=rhs.GetLen();
84:    itsString = new char[itsLen+1];
85:    for (int i = 0; i < itsLen;i++)
86:       itsString[i] = rhs[i];
87:    itsString[itsLen] = '\0';
88:    ASSERT(Invariants());
89: }
90:
91: // destructor, frees allocated memory
92: String::~String ()
93: {
94:    ASSERT(Invariants());
95:    delete [] itsString;
96:    itsLen = 0;
97: }
98:
99: // operator equals, frees existing memory
100: // then copies string and size
101: String& String::operator=(const String & rhs)
102: {
103:    ASSERT(Invariants());
104:    if (this == &rhs)
105:       return *this;
106:    delete [] itsString;
107:    itsLen=rhs.GetLen();
108:    itsString = new char[itsLen+1];
109:    for (int i = 0; i < itsLen;i++)
110:       itsString[i] = rhs[i];
111:    itsString[itsLen] = '\0';
112:    ASSERT(Invariants());
113:    return *this;
114: }
115:
```

```
116: //non constant offset operator
117: char & String::operator[](int offset)
118: {
119:   ASSERT(Invariants());
120:   if (offset > itsLen)
121:   {
122:     ASSERT(Invariants());
123:     return itsString[itsLen-1];
124:   }
125:   else
126:   {
127:     ASSERT(Invariants());
128:     return itsString[offset];
129:   }
130: }
131:
132: // constant offset operator
133: char String::operator[](int offset) const
134: {
135:   ASSERT(Invariants());
136:   char retVal;
137:   if (offset > itsLen)
138:     retVal = itsString[itsLen-1];
139:   else
140:     retVal = itsString[offset];
141:   ASSERT(Invariants());
142:   return retVal;
143: }
144:
145: BOOL String::Invariants() const
146: {
147:   #ifdef SHOW_INVARIANTS
148:     cout << "String Tested OK ";
149:   #endif
150:   return ( (itsLen && itsString) || (!itsLen && !itsString) );
151: }
152:
153: class Animal
154: {
155:  public:
156:    Animal():itsAge(1),itsName("John Q. Anima_")
157:      {ASSERT(Invariants());}
158:    Animal(int, const String&);
159:    ~Animal(){}
160:    int GetAge() { ASSERT(Invariants()); return itsAge;}
161:    void SetAge(int Age)
162:    {
163:      ASSERT(Invariants());
164:      itsAge = Age;
165:      ASSERT(Invariants());
166:    }
167:    String& GetName()
```

```
168:     {
169:       ASSERT(Invariants());
170:       return itsName;
171:     }
172:     void SetName(const String& name)
173:     {
174:       ASSERT(Invariants());
175:       itsName = name;
176:       ASSERT(Invariants());
177:     }
178:     BOOL Invariants();
179:   private:
180:     int itsAge;
181:     String itsName;
182: };
183:
184: Animal::Animal(int age, const String& name):
185:    itsAge(age),
186:    itsName(name)
187: {
188:    ASSERT(Invariants());
189: }
190:
191: BOOL Animal::Invariants()
192: {
193:    #ifdef SHOW_INVARIANTS
194:       cout << "Animal Tested OK";
195:    #endif
196:       return (itsAge > 0 && itsName.GetLen());
197: }
198:
199: int main()
200: {
201:    Animal sparky(5,"Sparky");
202:    cout << endl << sparky.GetName().GetString() << " is ";
203:    cout << sparky.GetAge() << " years old.";
204:    sparky.SetAge(8);
205:    cout << endl << sparky.GetName().GetString() << " is ";
206:    cout << sparky.GetAge() << " years old.";
207:    return 0;
208: }
```

输出:

```
String Tested OK String Tested OK String Tested OK String Tested OK
String Tested OK String Tested OK String Tested OK String Tested OK
String Tested OK StringTested OK String Tested OK String Tested OK
String Tested OK String Tested OK Animal Tested OK String Tested OK
Animal Tested OK
Sparky is Animal Tested OK 5 years old.Animal Tested OK Animal Tested OK
Animal
Tested OK
Sparky is Animal Tested OK 8 years old.String Tested OK
```

分析：

在第 9～15 行定义了 ASSERT( )宏。如果定义了 DEBUG，则将 ASSERT( )定义为在参数为 false 时打印一条错误消息。

第 39 行声明了 String 类成员函数 Invariants( )，其实现位于第 143～150 行定义。第 49～55 行声明了构造函数，在对象构造好后，第 54 行调用 Invariants( )来确认是否被正确构造。

对于其他构造函数重复该模式；析构函数在它要开始毁坏对象前调用 Invariants( )；其他类函数在执行操作前调用 Invariants( )，并在返回前再次调用 Invariants( )。这确认和证实了 C++的一个基本原则：除构造函数和析构函数外，所有成员函数都应作用于有效的对象，并保持对象的有效状态。

第 176 行声明了 Animal 类的 Invariants( )方法，该方法是在第 189～195 行实现的。注意，在第 155、158、161 和 163 行，内联函数可以调用 Invariants( )方法。

21.7.5 打印中间值

除使用 ASSERT( )宏确认事实为真外，你还可能想打印指针、变量和字符串的当前值。这对于检验有关程序进程的假定以及确定 bug 在循环中的位置都很有帮助。程序清单 21.5 演示了这种思想。

程序清单 21.5 在调试模式下打印值

```
0:  // Listing 21.5 - Printing values in DEBUG mode
1:  #include <iostream>
2:  using namespace std;
3:  #define DEBUG
4:
5:  #ifndef DEBUG
6:     #define PRINT(x)
7:  #else
8:     #define PRINT(x) \
9:          cout << #x << ":\t" << x << endl;
10: #endif
11:
12: enum BOOL { FALSE, TRUE } ;
13:
14: int main()
15: {
16:    int x = 5;
17:    long y = 738981;
18:    PRINT(x);
19:    for (int i = 0; i < x; i++)
20:    {
21:       PRINT(i);
22:    }
23:
24:    PRINT (y);
25:    PRINT("Hi.");
26:    int *px = &x;
27:    PRINT(px);
28:    PRINT (*px);
29:    return 0;
30: }
```

输出：

```
x:      5
```

```
i:      0
i:      1
i:      2
i:      3
i:      4
y:      73898
"Hi.":  Hi.
px:       0012FEDC
*px:    5
```

分析：

第 6～9 行的 PRINT( )宏打印参数的当前值。注意，在第 9 行，传递给 cout 的第一个值是参数的字符串化版本，也就是说，如果传递 x，cout 将收到“x”。

接下来 cout 收到用引号括起的字符串“:\t”，这将打印一个冒号和一个制表符。cout 收到的第三个值是参数（x）的值，最后收到的是 endl，它换行并刷新缓冲区。

注意，读者运行该程序时，显示的地址可能不是 0012FEDC。

21.7.6　宏与函数及模板之比较

在 C++中，宏存在 4 个问题。首先，如果宏很大将令人迷惑，因为所有宏都必须在一行中定义。可以使用反斜杠（\）来扩展到下一行，但大型宏将很快变得难以管理。

其次，宏在每次被使用时都按内联方式展开。这意味着，如果一个宏被使用了 12 次，程序中将包含 12 次替换结果，而不像函数调用那样只出现一次。另一方面，它们的速度通常比函数调用快，因为没有函数调用的开销。

宏按内联方式扩展导致了第三个问题，宏不能出现在编译器使用的中间源代码中，因此在大多数调试器中无法看到。这就使得宏的调试非常棘手。

最后一个也是最大的问题是，宏不是类型安全的。虽然使用宏时几乎可以提供任何类型的参数，这很方便，但这完全破坏了 C++的强类型功能，因此被 C++程序员视为瘟疫。当然，解决这种问题的正确方法是使用模板，这在第 19 章介绍过。

21.8　内联函数

通常可以不使用宏，而声明一个内联函数。例如，程序清单 21.6 创建了内联函数 Cube( )，该函数的功能与程序清单 21.2 中的 CUBE 宏相同，但以类型安全的方式完成其工作。

程序清单 21.6　使用内联函数而不是宏

```
0:  #include <iostream>
1:  using namespace std;
2:
3:  inline unsigned long Square(unsigned long a) { return a * a; }
4:  inline unsigned long Cube(unsigned long a)
5:     { return a * a * a; }
6:  int main()
7:  {
8:     unsigned long x=1 ;
9:     for (;;)
10:    {
11:       cout << "Enter a number (0 to quit): ";
```

```
12:       cin >> x;
13:       if (x == 0)
14:         break;
15:       cout << "You entered: " << x;
16:       cout << ". Square(" << x << "): ";
17:       cout  << Square(x);
18:       cout<< ". Cube(" << x << "): ";
19:       cout << Cube(x) << "." << endl;
20:     }
21:     return 0;
22: }
```

输出：

```
Enter a number (0 to quit): 1
You entered: 1. Square(1): 1. Cube(1): 1.
Enter a number (0 to quit): 2
You entered: 2. Square(2): 4. Cube(2): 8.
Enter a number (0 to quit): 3
You entered: 3. Square(3): 9. Cube(3): 27.
Enter a number (0 to quit): 4
You entered: 4. Square(4): 16. Cube(4): 64.
Enter a number (0 to quit): 5
You entered: 5. Square(5): 25. Cube(5): 125.
Enter a number (0 to quit): 6
You entered: 6. Square(6): 36. Cube(6): 216.
```

分析：

第 3 和第 4 行定义了两个内联函数：Square( )和 Cube( )。它们都被声明为内联的，因此像宏一样，每次调用时都就地展开，没有函数调用的开销。

注意，展开为内联函数意味着在调用函数的地方（如第 17 行）放置函数的代码。由于没有进行函数调用，因此没有将返回地址和参数压入堆栈的开销。

第 17 行调用函数 Square( )，第 19 行调用函数 Cube( )。由于它们是内联函数，因此就像第 16～19 行被编写成下面这样：

```
16:       cout << ". Square(" << x << "): "  ;
17:       cout << x * x ;
18:       cout << ". Cube(" << x << "): " ;
19:       cout << x * x * x << "." << endl;
```

应该：

宏名应大写。这是一种通用约定，如果不这样做，其他程序员将感到迷惑。

应用括号将宏中每个参数括起。

不应该：

不要让宏产生副作用。不要在宏中递增变量或给变量赋值。

在使用 const 变量可行的情况下，不要使用#define 来定义常量。

21.9 位 运 算

经常需要在对象中设置标记以跟踪对象的状态：是否处于警告状态（AlarmState）？有没有初始化？要来

还是要走？

为此，可以使用用户定义的布尔变量，但有些应用程序（尤其是低级驱动程序）和硬件设备要求你能够将变量的位用作标记。

每个字节包含 8 位，因此 4 字节的 long 变量能够存储 32 个标记。位的值为 1 时称为被设置，为 0 时称为被清除。设置位时使其值为 1；清除位时使其值为 0。可以通过修改 long 变量的值来设置或清除其位，但这样做很烦琐且容易令人迷惑。

注意：附录 A 提供了有关操纵二进制和十六进制数的宝贵信息。

C++提供的位运算符可以对变量的各个位进行操作。这些运算符看起来很像逻辑运算符，但实际上并不同，因此，很多编程新手常常弄混它们。表 21.1 列出了位运算符。

表 21.1　　位运算符符号

符　号	运 算 符
&	与
|	或
^	异或
~	求补

21.9.1　“与”运算符

“与”运算符为单个&，而逻辑“与”为两个&。对两个位进行“与”运算时，如果它们都是 1，则结果为 1；如果至少有一个为 0，则结果为 0。可以这样看待“与”运算：如果两个为都被设置，则结果为 1。

21.9.2　“或”运算符

第二个位运算符是“或”(|)。它用单个竖杠表示，而逻辑“或”用两个竖杠表示。对两个位进行“或”运算时，如果其中至少有一个被设置，则结果为 1。

21.9.3　“异或”运算符

第三个位运算符是“异或”(^)。对两个位进行“异或”运算时，如果两个位的值相同（都为 0 或都为 1），则结果为 0。

21.9.4　“求补”运算符

“求补”运算符（~）清除被设置的位并设置被清除的位。如果当前值为 1010 0011，则“求补”结果为 0101 1100。

21.9.5　设置位

要设置或清除某个位，可使用掩码运算。如果有一个 4 字节的标记，要将第 8 位设置为 true，可对该标记和 128 执行“或”运算。

为什么呢？128 的二进制表示为 1000 0000。因此，第 8 位的位置值为 128。无论该位的当前值是什么（设置或清除），如果将其与值 128 执行“或”运算，都将设置该位而不改变其他位。假定变量的当前值为 1010 0110 0010 0110，将其与 128 执行“或”运算时结果如下：

```
          9 8765 4321
  1010 0110 0010 0110   // bit 8 is clear
| 0000 0000 1000 0000   // 128
  ---------------
  1010 0110 1010 0110   // bit 8 is set
```

读者应该发现了其他几点。首先，位通常从右向左编号；其次在 128 的二进制表示中，除第 8 位（要设置的位）外，其他位皆为 0；第二，与 128 进行“或”运算后，原始值 1010 0110 0010 0110 除第 8 位被设置外，其他位不变。如果第 8 位已经被设置，它将保持被设置状态，这正是你希望的。

21.9.6 清除位

要清除第 8 位，可以将其同 128 的补码执行“与”运算。要得到 128 的补码，只需将 128 的位模式（1000 0000）中被设置的位清除，将被清除的位设置即可，结果为 0111 1111。将 128 的补码同变量执行“与”运算时，除第 8 位变为 0 外，变量的其他位不变。

```
  1010 0110 1010 0110  // bit 8 is set
& 1111 1111 0111 1111  // ~128

  ---------------
  1010 0110 0010 0110  // bit 8 cleared
```

要理解这种解决方案，应自己执行运算。两个位都为 1 时，在结果中书写 1；至少有一位为 0 时，在结果中书写 0。将结果与原来值进行比较，将发现除第 8 位被清除外，其他位保持不变。

21.9.7 反转位

最后，要反转第 8 位（不管其当前状态如何），将变量同 128 执行“异或”运算。如果执行这种运算两次，结果将与原来的值相同。

```
  1010 0110 1010 0110  // number
^ 0000 0000 1000 0000  // 128

  ---------------
  1010 0110 0010 0110  // bit flipped
^ 0000 0000 1000 0000  // 128

  ---------------
  1010 0110 1010 0110  // flipped back
```

应该：

应使用掩码和“或”运算符来设置位。

应使用掩码和“与”运算符来清除位。

应使用掩码和“异或”运算符来反转位。

不应该：

不要混淆各种位运算。

反转位时，别忘了考虑它左边的位。一个字节包含 8 位，需要知道使用的变量有多少个字节。

21.9.8 位字段

在有些情况下，每个字节的空间都很宝贵，在类中节省 6 个和 8 个字节有天壤之别。如果类或结构中有一系列布尔变量或只有少数几个可能取值的变量，可以使用位字段来节省内存空间。

在类中可以使用的最小的标准 C++数据类型是 char，它可能只占用 1 个字节。通常你会使用 int 类型，在使用 32 位处理器的计算机上，它占用 4 个字节。通过使用位字段，可以在 char 变量中存储 8 个二进制值，在 4 字节的 int 变量中存储 32 个二进制值。

位字段的命名和访问方式与其他类成员相同。它们的类型总是 unsigned int，并在位字段名后加上冒号和数字。

这个数字告诉编译器，为该变量分配多少位内存。如果为 1，位字段将能够表示 0 或 1。如果为 2，位字段将能够表示 0、1、2 或 3，总共 4 个值。长 3 位的字段可表示 8 个值，依此类推。附录 A 简要地介绍了二进制数。程序清单 21.7 演示了位字段的用法。

程序清单21.7 使用位字段

```
#include <iostream>
using namespace std;
#include <string.h>

enum STATUS { FullTime, PartTime } ;
enum GRADLEVEL { UnderGrad, Grad } ;
enum HOUSING { Dorm, OffCampus };
enum FOODPLAN { OneMeal, AllMeals, WeekEnds, NoMeals };

class student
{
  public:
    student():
       myStatus(FullTime),
       myGradLevel(UnderGrad),
       myHousing(Dorm),
       myFoodPlan(NoMeals)
    {}
    ~student(){}
    STATUS GetStatus();
    void SetStatus(STATUS);
    unsigned GetPlan() { return myFoodPlan; }

  private:
    unsigned myStatus : 1;
    unsigned myGradLevel: 1;
    unsigned myHousing : 1;
    unsigned myFoodPlan : 2;
};

STATUS student::GetStatus()
{
   if (myStatus)
      return FullTime;
   else
      return PartTime;
}

void student::SetStatus(STATUS theStatus)
{
   myStatus = theStatus;
}

int main()
{
   student Jim;

   if (Jim.GetStatus()== PartTime)
      cout << "Jim is part-time" << endl;
   else
      cout << "Jim is full-time" << endl;
```

```
51:
52:    Jim.SetStatus(PartTime);
53:
54:    if (Jim.GetStatus())
55:       cout << "Jim is part-time" << endl;
56:    else
57:       cout << "Jim is full-time" << endl;
58:
59:    cout << "Jim is on the " ;
60:
61:    char Plan[80];
62:    switch (Jim.GetPlan())
63:    {
64:      case OneMeal:  strcpy(Plan,"One meal"); break;
65:      case AllMeals: strcpy(Plan,"All meals"); break;
66:      case WeekEnds: strcpy(Plan,"Weekend meals"); break;
67:      case NoMeals:  strcpy(Plan,"No Meals");break;
68:      default :    cout << "Something bad went wrong! "<< endl;
69:                   break;
70:    }
71:    cout << Plan << " food plan." << endl;
72:    return 0;
73: }
```

输出：

```
Jim is part-time
Jim is full-time
Jim is on the No Meals food plan.
```

分析：

第 4～7 行定义了几种枚举类型，它们用于定义 student 类中位字段的可能取值。

第 9～28 行声明了 student 类。虽然这是一个很小的类，但很有趣，因为其所有数据都存储在 5 个位字段中（第 24～27 行）。第 1 个位字段（第 24 行）表示学生的类别：全职或业余；第二个位字段（第 25 行）表示学生是否为本科生；第三个位字段（第 26 行）表示学生是否住校；最后一个两位的位字段表示 4 种就餐方式。

类方法的编写方式与其他类相同，不受数据以位字段而不是整型或枚举型的影响。

第 30～36 行的成员函数 GetStatus( )读取位字段的值，并返回一个枚举类型，但并非必须这样做，而可以直接返回位字段的值，编译器将进行转换。

为证明这一点，可以用下面的代码代替 GetStatus( )的实现：

```
STATUS student::GetStatus()
{
    return myStatus;
}
```

程序的功能不会有任何变化。这只是一个代码可读性问题，不影响编译器。

注意，第 47 行的代码必须检查类别，然后打印有意义的消息。如果写成下面这样：

```
cout << "Jim is " << Jim.GetStatus() << endl;
```

将打印：

```
Jim is 0
```

编译器不能将枚举常量 PartTime 转换为有意义的文本。

第 62 行使用 switch 语句判断就餐方式，并据此将一条相应的消息存储到字符数组 plan 中；然后在第 71 行打印该字符数组。也可以将 switch 语句写成这样：

```
case  0: strcpy(Plan,"One meal"); break;
case  1: strcpy(Plan,"All meals"); break;
case  2: strcpy(Plan,"Weekend meals"); break;
case  3: strcpy(Plan,"No Meals");break;
```

使用位字段时最重要的一点是，类的客户无需关心数据存储实现。由于位字段被声明为私有的，因此以后可以修改它们，而无需修改接口。

21.10　编程风格

正如在本书其他地方指出的，采用一致的编码风格很重要，虽然从很多方面说，采用哪种编码风格无关紧要。一致的编程风格让人更容易猜测某部分代码的意图，还可避免查看上次调用函数时，首字母是否大写。

下面是一些指导原则，它们基于已完成的项目中使用的指导原则，事实证明它们很管用。你也可以制定自己的指导原则，但这些可供你开始时参考。

Emerson 说，“愚蠢的一致是一些零碎想法的混杂”，但在代码中保持一致是好事。制定自己的指导原则，就像它是由编程高手制定的一样对待它。

21.10.1　缩进

如果使用制表符，它应为 3 个空格。确保编辑器将制表符转换为 3 个空格。

21.10.2　大括号

如何对齐大括号是 C++程序员之间争论最为激烈的话题。下面推荐了一些做法：

- 匹配的大括号应水平对齐。
- 定义和声明中最外面的大括号应在最左边；内部的语句应该缩进；其他所有大括号应与相关联的命令左对齐。
- 大括号应单独占一行。例如：

```
if (condition==true)
{
      j = k;
      SomeFunction();
}
i++;
```

注意：正如前面指出的，程序员们对大括号的对齐方式存在争议。很多 C++程序员认为，应将左大括号与相关联的命令放在同一行，并将右大括号与相关联的命令左对齐，如下所示：

```
if (condition==true) {
      j = k;
      SomeFunction();
}
```

这种格式被认为不好阅读，因为大括号没有对齐。

21.10.3　长代码行和函数长度

确保不用水平滚动就能看到整行代码。超出右边界的代码易被忽略，且水平滚动很烦人。

将一行代码分成多行时，对后续行进行缩进。尽量在合理的地方分行，将插入运算符放在前一行的末尾

（而不是下一行的开头），这样可清楚地知道，该行并不是完整的，后面还有其他代码。

在 C++中，函数通常比在 C 语言中短得多，但以前的合理建议仍然适用。尽量使函数足够短，以便可以在一页中打印函数的全部代码。

21.10.4 格式化 switch 语句

像下面这样缩进各个分支：

```
switch(variable)
{
      case ValueOne:
            ActionOne();
            break;
      case ValueTwo:
            ActionTwo();
            break;
      default:
            assert("bad Action");
            break;
}
```

正如读者看到的，稍微向右缩进了 case 语句。另外，对齐了每个分支中的语句。采用这种布局时，容易找到 case 语句及其后面的代码。

21.10.5 程序文本

可使用多种技巧来使代码易于阅读。易读的代码通常更容易维护。

- 使用空格来提高可读性。
- 不要在对象、数组名和运算符（.、->、[]）之间使用空格。
- 单目运算符与其操作数相关联，因此不要在它们之间添加空格，但在操作数的另一边添加空格。单目运算符包括!、~、++、--、-、*（用于指针）、&（取地址）和 sizeof。
- 双目运算符两边都应有空格，双目运算符包括+、=、*、/、%、>>、<<、<、>、==、!=、&、|、&&、||、?:、=、+=等。
- 不要通过省略空格来指示优先级，如(4+3*2)。
- 在逗号和分号后面加上空格，但在它们前面不加。
- 圆括号两边不应有空格。
- 用空格将关键字（如 if）分开，如 if (a == b)。
- 使用空格将单行注释的内容同//分开。
- 将指针或引用指示符紧靠类型名，而不是变量名，如下所示：

```
char* foo;
int& theInt;
```

- 不在一行中声明多个变量。

21.10.6 标识符命名

下面是一些给标识符命名的指导原则：

- 标识符名称应足够长以便具有描述性。
- 避免意义不明确的缩写。
- 花时间和精力将含义拼写出来。
- 不使用匈牙利表示法。C++是强类型的，没有必要在变量名中包含类型名。对于用户定义的类型（类），匈牙利表示根本不管用。例外的情况是，在指针名中使用前缀 p、在引用名中使用前缀 r 以及在类成员变量

名中使用前缀 its。

- 仅当简短性可提高代码的可读性或用途十分明显不需要描述性名称时，才使用短名称（i、p、x 等）。然而，通常应避免使用这样的名称；另外，应避免在变量名中使用字母 i、l 和 o，因为它们容易与数字混淆。
- 变量名的长度应与其作用域相称。
- 确保标识符看上去和听上去都相互不同，以尽可能减少混淆。
- 函数（或方法）名通常为动词或动词-名词短语，如 Search( )、Reset( )、FindParagraph( )、ShowCursor( )。变量名通常为抽象名词，可以带一个附加名词，如 count、state、windSpeed、windowHeight。布尔变量应相应地命名，如 windowIconized、fileIsOpen。

21.10.7　名称的拼写和大写

制定自己的编程风格时，不要忽略拼写和大写。下面是这方面的一些技巧：

- 对于使用 #define 定义的常量，采用全部大写并用下划线将其中的单词分开，如 SOURCE_FILE_TEMPLATE。然而，在 C++中很少采用这种方式来定义常量，大多数情况下可考虑使用 const 和模板。
- 其他标识符应采用大小写混合，没有下划线。函数、方法、类、typedef 和结构的名称应采用首字母大写。诸如数据成员和局部变量等不应采用首字母大写。
- 枚举常量应以表示枚举类型缩写的小写字母开头，如：

```
enum TextStyle
{
        tsPlain,
        tsBold,
        tsItalic,
        tsUnderscore,
};
```

21.10.8　注释

注释可使程序更容易理解。有时候，编写程序期间可能暂停几天甚至几个月。在此期间，你可能忘记某些代码的功能或不知道为什么使用这些代码。别人阅读你的代码时，也可能无法理解。使用一致、深思熟虑的注释风格是值得的。请记住下面几条关于注释的技巧：

- 尽可能使用 C++单行注释（//）而不是多行注释（/* */）。将多行注释用于将可能包含单行注释的代码块注释掉。
- 高级注释比处理细节更重要。要增加价值而不要复述代码，下面的注释根本不值得花时间去输入：n++; // n is incremented by one。将重点放在函数和代码块的语义上：指出函数的功能、副作用、参数类型和返回值；描述做出（或没有做出）的所有假设，如“假设 n 非负”或“如果 x 非法将返回-1”。在复杂的逻辑中，使用注释来指出当前的状态。
- 使用采用合适标点符号和大小写的完整句子。这种额外的输入是值得的。注释不要过于晦涩，也不要使用缩写。编写代码时显而易见的事情几个月后将变得极其难懂。
- 使用空行来帮助读者理解所发生的事情：将语句分成逻辑组。

21.10.9　设置访问权限

访问程序各部分的方式应保持一致。下面是一些设置访问权限的技巧：

- 总是使用 public:、private:和 protected:，不要依赖于默认访问设置。
- 先列出公有成员，其次是保护成员，然后是私有成员。在方法后面集中列出数据成员。
- 在各个部分首先列出构造函数，然后是析构函数。集中列出同名的重载方法。尽可能集中列出存取器函数。

- 考虑按字母顺序排列每组中的方法和成员变量。包含文件时，按字母顺序排列它们。
- 虽然覆盖函数时关键字 virtual 是可选的，但应尽可能使用它。它有助于提醒函数是虚函数以及保持声明的一致性。

21.10.10 类定义

尽可能确保方法实现的排列顺序与声明顺序一致，这可使方法更容易找到。

定义函数时，将返回值类型和其他所有限定符都放在前一行，让类名和函数名位于行首，这样更容易查找函数。

21.10.11 包含文件

尽可能少用#include，最大限度地减少在头文件中包含的文件。理想情况是，只包含基类的头文件，其他必须包含的文件是声明类成员对象的头文件。

不要因为相应的.cpp 文件需要包含某个文件，而在头文件中也包含该文件。不要因为被包含的文件需要包含某个文件而再包含该文件。

提示：所有头文件都应使用多重包含防范。

21.10.12 使用 assert( )

本章前面介绍了 assert( )。尽可能使用 assert( )，它有助于发现错误，还可帮助了解程序编写者所做的假设以及他认为什么是合法的、什么是非法的。

21.10.13 使用 const

在合适的地方使用 const：参数、变量和方法。通常，方法需要有 const 版本和非 const 版本，不要以此为借口遗漏其中之一。显式地在 const 和非 const 之间进行转换时（有时，这是解决某些问题的惟一方式）务必小心，要确保这种转换有意义并进行注释。

21.11 C++开发工作的下一步

经过 3 周漫长而艰辛的 C++学习，你具备了成为合格的 C++程序员所需的基本知识，但学习并没有结束。要从 C++新手成为 C++专家，还有很多的东西需要学习，需要从其他的地方获得有价值的信息。

接下来的几小节推荐了很多信息源，这些推荐只反映笔者个人的经验和观点。有关这些主题的书籍和文章很多，购买之前一定要听听其他的观点。

21.11.1 从何处获得帮助和建议

作为一名 C++程序员，要做的第一件事就是加入 Internet 上的一个或多个 C++社区。这些小组让你能够及时地与数百甚至数千名 C++程序员取得联系，他们可以回答你的问题、提供建议以及发表看法的途径。

Internet C++新闻组（comp.lang.c++和 comp.lang.c++.moderated）是作者推荐的极好的信息和支持源；还有诸如 http://www.CodeGuru.com 和 http://www.CodeProject.com 等网站，每个月都有数以 10 万计的 C++程序员光顾这些网站，它们提供诸如有关 C++的文章、教程、新闻和讨论等。当然，这样的社区数不胜数。

此外，你可能想寻找当地的用户小组。很多城市都有 C++兴趣小组，在那里可以遇到其他程序员，同它们交流看法。

最后，诸如 Borland 和 Microsoft 等编译器厂商也有新闻组，这些新闻组提供了有关开发环境和 C++语言的宝贵信息。

21.11.2　相关的 C++主题：受控 C++、C#和 Microsoft 的.NET

Microsoft 新的.NET 平台极大地改变了 Internet 开发方法，.NET 的一个重要组成部分是新语言 C#和大量被称为受控扩展（Managed Extension）的 C++扩展。

C#是 C++的扩展，为 C++编程人员转向.NET 平台提供了桥梁。有关 C#的优秀图书很多，其中包括 *Programming C#*（O'Reilly 出版社）和组织结构与本书类似的 *Sams Teach Yourself the C# Language in 21 Days*。

作为一种编程语言，C#有一些不同于 C++的地方。例如，C#不支持多继承，但使用接口提供了类似的功能。另外，C#不使用指针，这消除了悬浮指针等问题，但代价是降低了该语言的低级、实时编程功能。有关 C#需要指出的一点是，它使用了一个运行阶段库和无用单元收集程序（GC）。GC 负责在必要时释放资源，避免了程序员这样做。

受控 C++也来自 Microsoft，它是.NET 的组成部分。简单地说，这是一个 C++扩展，让 C++能够使用包括无用单元收集程序在内的所有.NET 特性。

21.11.3　保持联系

如果读者对本书或其他图书有什么建议或想法，作者很乐意听到。请通过网站 www.libertyassociates.com 同作者联系，期待收到你的来信。

应该：

应阅读其他书籍。要学的知识很多，一本书不可能教给你需要的所有知识。

应加入一个优秀的 C++用户组。

不应该：

不要只阅读代码！学习 C++的最佳方法是编写 C++程序。

21.12　小　　结

本章介绍了有关使用预处理器的更多细节。每次运行编译器时，预处理器都将首先运行，对诸如#define 和#ifdef 等预处理指令进行转换。

预处理器进行文本替换，虽然使用宏时，这种转换有点复杂。通过使用#ifdef、#else 和#ifndef，可以实现条件编译：在一组条件下编译一些语句，在另一组条件下编译另一些语句。这有助于编写用于多个平台的程序，常常用来有条件地包含调试信息。

宏函数根据编译时传递给宏的参数，提供复杂的文本替换。应用括号将宏的每个参数括起以确保进行正确的替换，这很重要。

相当于 C 语言，宏和预处理指令在 C++中不那么重要。C++提供了诸如 const 变量和模板等语言特性，它们可替代预处理指令且更高级。

本章还介绍了如何设置和测试位以及如何给类成员分配有限数目的位。

最后，讨论了 C++编程风格，提供了进一步学习的资源。

21.13　问　与　答

问：既然 C++提供了比预处理器更好的替代方案，为什么仍保留这种选项呢？

答：首先 C++向后与 C 语言兼容，C++必须支持 C 语言中所有的重要部分；其次，预处理器的某些用法在 C++中仍被频繁使用，如多重包含防范。

问：为什么在可以使用常规函数时仍使用宏函数？

答：宏函数以内联方式被展开，用于避免重复输入变化很小的命令。然而，模板提供一个更好的解决方案。

问：什么时候使用宏？什么时候使用内联函数？

答：尽可能使用内联函数。虽然宏提供字符替换、字符串化和字符串拼接功能，但它们不是类型安全的，可能导致代码更难以维护。

问：可使用什么来代替编译指令在调试期间打印中间值？

答：最好的替代方案是在调试器中使用 watch 语句。有关 watch 语句的信息，请参阅编译器或调试器文档。

问：何时该使用 assert( )？何时该引发异常？

答：如果测试的条件在没有编程错误的情况下可以为真，则使用异常。如果仅当程序有错误时该条件才为真，则使用 assert( )。

问：什么时候应使用位字段而不是 int 变量？

答：在对象的大小很重要时。如果可用的内存有限或编写的是通信软件，将发现使用位字段来节省空间对产品的成功至关重要。

问：可以将指针赋给位字段吗？

答：不可以。内存地址通常指向字节的开头，而位字段可能位于字节中间。

问：为什么有关风格的争论如此激烈？

答：程序员的习惯是根深蒂固的。如果你习惯了下面的缩进方式：

```
if (SomeCondition){
     // statements
}   // closing brace
```

改变起来将非常困难，总是认为新的编程风格不对，容易带来混乱。可以试着登录到某个流行的在线服务，问一下哪种缩进风格最好，哪一种编辑器最适合 C++，哪种产品是最好的字处理器，将看到上万条回信，所有回信都相互矛盾。

问：C++内容就这么多吗？

答：是的！你已学会了 C++，但总是有更多的知识需要学习。10 年前，学习所有关于某种计算机编程语言机的知识是可能的，至少感到非常接近这种程度；而今天这是不可能的。即使尽最大努力，也不可能跟上潮流，行业在不断变化。务必不断阅读，经常查阅资源（杂志和在线服务），这样才能了解最新的变化。

21.14 作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

21.14.1 测验

1. 什么是多重包含防范？
2. 如何命令编译器打印出中间文件的内容，以了解预处理器的效果？
3. #define debug 0 和#undef debug 有什么区别？
4. 请看下面的宏：

```
#define HALVE(x) x / 2
```

如果使用 4 作为参数来调用它，结果是什么？

5. 如果使用 10+10 作为参数来调用问题 4 中的 HALVE 宏，结果是多少？

6. 如果修改 HALVE 宏以避免错误结果？

7. 在两字节的变量中可以存储多少位值？

8. 5 位可以存储多少个可能的值？

9. 0011 1100 | 1111 1111 的结果是什么？

10. 0011 1100 & 1111 1111 结果是什么？

21.14.2　练习

1. 编写头文件 STRING.H 的多重包含防范语句。

2. 编写一个 assert( )宏。如果调试级别是 2，它打印一条错误消息、文件名和行数；如果调试级别为 1，只打印一条消息（没有文件名和行数）；如果调试级别为 0，则什么都不打印。

3. 编写一个 DPrint 宏，它检测 DEBUG 是否被定义，如果被定义，则打印作为参数传入的值。

4. 编写一个类声明，这个类使用位字段来存储月份、年份和天。