1.1 C++程序结构 （Structure of a program）

下面我们从一个最简单的程序入手看一个 C++程序的组成结构。

```
// my first program in C++      Hello World!
#include <iostream.h>
using namespace std;

int main() {
       cout << “Hello
World!”;
       return 0;
}
```

上面左侧显示了我们的第一个程序的源代码，代码文件名称为 hellowworld.cpp。右边显示了程序被编译执行后的输出结果。编辑和编译一个程序的方法取决于你用的是什么编译器，根据它是否有图形化的界面及版本的不同，编译方法也有可能不同，具体请参照你所使用的编译器的使用说明。

以上程序是多数初学者学会写的第一个程序，它的运行结果是在屏幕上打出”Hello World!”这句话。 虽然它可能是 C++可写出的最简单的程序之一，但其中已经包含了每一个 C++程序的基本组成结构。 下面我们就逐个分析其组成结构的每一部分：

// my first program in C++

这是注释行。所有以两个斜线符号(//)开始的程序行都被认为是注释行，这些注释行是程序员写在程序源代码内，用来对程序作简单解释或描述的， 对程序本身的运行不会产生影响。在本例中， 这行注释对本程序是什么做了一个简要的描述。

include < iostream.h >

以#标志开始的句子是预处理器的指示语句。它们不是可执行代码，只是对编译器作出指示。在本例中这个句子# include < iostream.h > 告诉编译器的预处理器将输入输出流的标准头文件(iostream.h)包括在本程序中。这个头文件包括了 C++中定义的基本标准输入-输出程序库的声明。此处它被包括进来是因为在本程序的后面部分中将用到它的功能。

using namespace std;

C++标准函数库的所有元素都被声明在一个名空间中，这就是 std 名空间。因此为了能够访问它的功能，我们用这条语句来表达我们将使用标准名空间中定义的元素。这条语句在使用标准函数库的 C++程序中频繁出现，本教程中大部分代码例子中也将用到它。

int main()

这一行为主函数(main function)的起始声明。main function 是所有 C++程序的运行的起始点。 不管它是在代码的开头，结尾还是中间 - 此函数中的代码总是在程序开始运行时第一个被执行。并且，由于同样的原因，所有 C++程序都必须有一个 main function。
main 后面跟了一对圆括号（），表示它是一个函数。C++中所有函数都跟有一对圆括号)(，括号中可以有一些输入参数。如例题中显示，主函数(main function)的内容紧跟在它的声明之后，由花括号 （{}）括起来。

cout << "Hellow World!";

这个语句在本程序中最重要。 cout 是 C++中的标准输出流(通常为控制台，即屏幕)，这句话把一串字符串（本例中为"Hello World"）插入输出流（控制台输出）中。cout 在的声明在头文件 iostream.h 中，所以要想使用 cout 必须将该头文件包括在程序开始处。
注意这个句子以分号(;)结尾 。分号标示了一个语句的结束，C++的每一个语句都必须以分号结尾。(C++ 程序员最常犯的错误之一就是忘记在语句末尾写上分号) 。

return 0;

返回语句(return) 引起主函数 main()执行结束，并将该语句后面所跟代码(在本例中为 0) 返回。这是在程序执行没有出现任何错误的情况下最常见的程序结束方式。在后面的例子中你会看到所有 C++程序都以类似的语句结束。

你可能注意到并不是程序中的所有的行都会被执行。程序中可以有注释行（以//开头），有编译器预处理器的指示行（以#开头），然后有函数的声明（本例中 main 函数），最后是程序语句（例如调用 cout <<），最后这些语句行全部被括在主函数的花括号({})内。
本例中程序被写在不同的行中以方便阅读。其实这并不是必须的。例如，以下程序

```
int main ()
{
cout << " Hello World ";
return 0;
}
```

也可以被写成：

```
int main () { cout << " Hello World "; return 0; }
```

以上两段程序是完全相同的。

在 C++中，语句的分隔是以分号(;)为分隔符的。分行写代码只是为了更方便人阅读。
以下程序包含更多的语句：

```
// my second program in C++    Hello World! I'm a C++ program
#include <iostream.h>
```

```
int main ()
{
cout << "Hello World! ";
cout << "I'm a C++ program";

return 0;
}
```

在这个例子中，我们在两个不同的语句中调用了 cout << 函数两次。再一次说明分行写程序代码只是为了我们阅读方便，因为这个 main 函数也可以被写为以下形式而没有任何问题:

```
int main () { cout << " Hello World! "; cout << " I'm to C++ program ";
return 0; }
```

为方便起见，我们也可以把代码分为更多的行来写：

```
int main ()
{
cout <<
"Hello World!";
cout
<< "I'm a C++ program";
return 0;
}
```

它的运行结果将和上面的例子完全一样。

这个规则对预处理器指示行（以#号开始的行）并不适用，因为它们并不是真正的语句。它们由预处理器读取并忽略，并不会生成任何代码。因此他们每一个必须单独成行，末尾不需要分号(;)

注释 (Comments)

注释(comments)是源代码的一部分，但它们会被编译器忽略。它们不会生成任何执行代码。 使用注释的目的只是使程序员可以在源程序中插入一些说明解释性的内容。

C++ 支持两中插入注释的方法:

```
// line comment
/* block comment */
```

第一种方法为行注释，它告诉编译器忽略从//开始至本行结束的任何内容。第二种为块注释（段注释），告诉编译器忽略在/*符号和*/符号之间的所有内容，可能包含多行内容。

在以下我们的第二个程序中，我们插入了更多的注释。

```
/* my second program in C++
with more comments */

#include <iostream.h>

int main ()
{
cout << "Hello World! "; // says Hello World!

cout << "I'm a C++ program"; // says I'm a C++
program
return 0;
}
```

```
Hello World! I'm a C++
program
```

可能觉得这个“Hellow World”程序用处不大。我们写了好几行代码，编译，然后执行生成的程序只是为了在屏幕上看到一句话。的确，我们直接在屏幕上打出这句话会更快。但是编程并不仅限于在屏幕上打出文字这么简单的工作。为了能够进一步写出可以执行更有用的任务的程序，我们需要引入**变量（variable)**这个的概念。

让我们设想这样一个例子，我要求你在脑子里记住 5 这个数字，然后再记住 2 这个数字。你已经存储了两个数值在你的记忆里。现在我要求你在我说的第一个数值上加 1，你应该保留 6 （即 5+1）和 2 在你的记忆里。现在如果我们将两数相减可以得到结果 4。

所有这些你在脑子里做的事情与计算机用两个变量可以做的事情非常相似。同样的处理过程用 C++来表示可以写成下面一段代码：

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

很明显这是一个很简单的例子，因为我们只用了两个小的整数数值。但是想一想你的电脑可以同时存储成千上万这样的数值，并进行复杂的数学运算。

因此，我们可以将**变量（variable）**定义为内存的一部分，用以存储一个确定的值。

每一个变量 (variable)需要一个标识，以便将它与其他变量相区别，例如，在前面的代码中，变量标识是 a, b, 和 result。我们可以给变量起任何名字，只要它们是有效的标识符。

标识（Identifiers）

有效标识由字母(letter)，数字(digits)和下划线 （ _)组成。标识的长度没有限制，但是有些编译器只取前 32 个字符（剩下的字符会被忽略）。

空格(spaces)，标点(punctuation marks)和符号(symbols) 都不可以出现在标识中。 只有字母(letters)，数字(digits) 和下划线(_)是合法的。并且变量标识必须以字母开头。标识也可能以下划线（_）开头，但这种标识通常是保留给为外部连接用的。标识不可以以数字开头。

必须注意的另一条规则是当你给变量起名字时不可以和 C++语言的**关键字**或你所使用的编译器的**特殊关键字**同名，因为这样与这些关键字产生混淆。例如，以下列出标准保留关键字，他们不允许被用作变量标识名称：

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

另外，不要使用一些操作符的替代表示作为变量标识，因为在某些环境中它们可能被用作保留词：

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

你的编译器还可能包含一些特殊保留词，例如许多生成 16 位码的编译器（比如一些 DOS 编译器）把 far, huge 和 near 也作为关键字。

非常重要：C++语言是“ 大小写敏感”（“case sensitive”） 的，即同样的名字字母大小写不同代表不同的变量标识。因此，例如变量 RESULT，变量 result 和变量 Result 分别表示三个不同的变量标识.

基本数据类型（Fundamental Data types）

编程时我们将变量存储在计算机的内存中，但是计算机要知道我们要用这些变量存储什么样的值，因为一个简单的数值，一个字符，或一个巨大的数值在内存所占用的空间是不一样的。

计算机的内存是以字节(byte)为单位组织的。一个字节（byte）是我们在 C++中能够操作的最小的内存单位。一个字节(byte)可以存储相对较小数据：一个单个的字符或一个小整数（通常为一个 0 到 255 之间的整数）。但是计算机可以同时操作处理由多个字节组成复杂数据类型，比如长整数(long integers)和小数（decimals）。以下列表总结了现有的 C++基本数据类型，以及每一类型所能存储的数据范围：

数据类型（DATA TYPES）

名称	字节数	描述	范围
char	1	字符（character）或整数（integer），8 位（bits）长	有符号（signed）：-128 到 127 无符号（unsigned）：0 到 255
short int (short)	2	短整数(integer)16 位(bits)长	有符号（signed）：-32768 到 32767 无符号（unsigned）：0 到 65535
long int (long)	4	长整数(integer)32 位(bits)长	有符号(signed):-2147483648 到 2147483647 无符号（unsigned）：0 到 4294967295
int	4	整数（integer）	有符号(signed): -2147483648 到 2147483647 无符号（unsigned): 0 到 4294967295
float	4	浮点数（floating point number）	3.4e + / - 38 （7 个数字（7digits））
double	8	双精度浮点数（double precision floating point number）	1.7e + / - 308 (15 digits)
long double	8	长双精度浮点数（long double precision floating point number）	1.7e + / - 308 (15 digits)
bool	1	布尔 Boolean 值。它只能是真	true 或 false

		(true)或假(false)两值之一。	
wchar_t	2	宽字符(Wide character) 。这是为存储两字节(2 bytes) 长的国际字符而设计的类型。	一个宽字符（1 wide characters）

* 字节数一列和范围一列可能根据程序编译和运行的系统不同而有所不同。这里列出的数值是多数 32 位系统的常用数据。对于其他系统，通常的说法是整型(int)具有根据系统结构建议的自然长度（即一个字 one word 的长度），而 4 中整型数据 char, short, int, long 的长度必须是递增的，也就是说按顺序每一类型必须大于等于其前面一个类型的长度。同样的规则也适用于浮点数类型 float, double 和 long double，也是按递增顺序。

除以上列出的基本数据类型外，还有指针（pointer）和 void 参数表示类型，我们将在后面看到。

变量的声明（Declaration of variables）

在 C++中要使用一个变量必须先声明（declare）该变量的数据类型。声明一个新变量的语法是写出数据类型标识符（例如 int, short, float...）后面跟一个有效的变量标识名称。例如：

```
int a;
float mynumber;
```

以上两个均为有效的变量声明（variable declaration）。第一个声明一个标识为 a 的整型变量(int variable)，第二个声明一个标识为 mynumber 的浮点型变量(float variable)。声明之后，我们就可以在后面的程序中使用变量 a 和 mynumber 了。

如果你需要声明多个同一类型的变量，你可以将它们缩写在同一行声明中，在标识之间用逗号(comma) 分隔。例如：

```
int a, b, c;
```

以上语句同时定义了 a、b、c 3 个整型变量，它与下面的写法完全等同：

```
int a;
int b;
int c;
```

整型数据类型 (char, short, long 和 int) 可以是有符号的（signed）或无符号的（unsigned ），这取决于我们需要表示的数据范围。有符号类型(signed)可以表示正数和负数，而无符号类型(unsigned)只能表示正数和 0。在定义一个

整型数据变量时可以在数据类型前面加关键字 signed 或 unsigned 来声明数据的符号类型。例如：

```
unsigned short NumberOfSons;
signed int MyAccountBalance;
```

如果我们没有特别写出 signed 或 unsigned，变量默认为 signed，因此以上第二个声明我们也可以写成：

```
int MyAccountBalance;
```

因为以上两种表示方式意义完全一样，因此我们在源程序通常省略关键字 signed 。

唯一的例外是字符型（char）变量，这种变量独立存在，与 signed char 和 unsigned char 型均不相同。

short 和 long 可以被单独用来表示整型基本数据类型，short 相当于 short int， long 相当于 long int。也就是说 short year; 和 short int year; 两种声明是等价的。

最后，signed 和 unsigned 也可以被单独用来表示简单类型，意思分别同 signed int 和 unsigned int 相同，即以下两种声明互相等同：

```
unsigned MyBirthYear;
unsigned int MyBirthYear;
```

下面我们就用 C++代码来解决在这一节开头提到的记忆问题，来看一下变量定义是如何在程序中起作用的。

```
// operating with variables     4

#include <iostream.h>
using namespace std;

int main ()
{
       // declaring variables:

       int a, b;
       int result;

       // process:
       a = 5;
```

```
        b = 2;
        a = a + 1;
        result = a - b;

        // print out the result:

        cout << result;

        // terminate the
program:
        return 0;
}
```

如果以上程序中变量声明部分有你不熟悉的地方，不用担心，我们在后面的章节中很快会学到这些内容。

变量初始化（Initialization of variables）

当一个本地变量(local variable)被声明时，它的值默认为未定(undetermined)。但你可能希望在声明变量的同时赋给它一个具体的值。要想达到这个目的，需要对变量进行初始化。C++中有两种初始化方法：

第一种，又叫做类 C (c-like) 方法，是在声明变量的时候加上一个等于号，并在后面跟上想要的数值：

```
type identifier = initial_value ;
```

例如，如果我们想声明一个叫做 a 的 int 变量并同时赋予它 0 这个值，我们可以这样写：

```
int a = 0;
```

另外一种变量初始化的方法，又叫做构造函数(constructor)初始化， 是将初始值用小括号（parenthesis ()）括起来:

```
type identifier (initial_value) ;
```

例如：

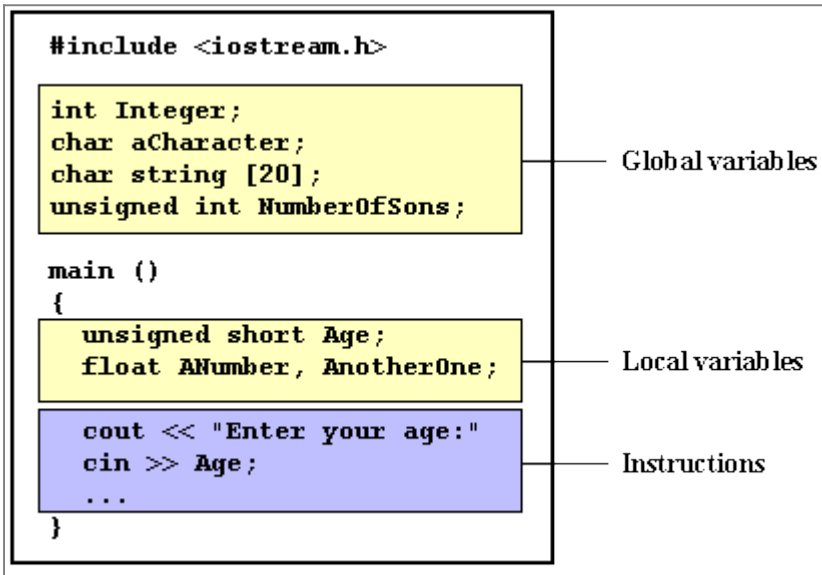
```
int a (0);
```

在 C++.中以上两种方法都正确并且两者等同 。

变量的范围（Scope of variables）

所有我们要使用的变量都必须事先声明过。C 和 C ++语言的一个重要区别是，在 C++语言中我们可以在源程序中任何地方声明变量，甚至可以在两个可执行(excutable)语句的中间声明变量，而不象在 C 语言中变量声明只能在程序的开头部分。

然而，我们还是建议在一定程度上遵循 C 语言的习惯来声明变量，因为将变量声明放在一处对 debug 程序有好处。因此，传统的 C 语言方式的变量声明就是把变量声明放在每一个函数(function)的开头（对本地变量 local variable）或直接放在程序开头所有函数(function)的外面（对全局变量 global variable）。



全局变量 Global variables 可以在程序中任何地方任何函数（function）中被引用，只要是在变量的声明之后。

本地变量 local variables 的作用范围被局限在声明它的程序范围内。如果它们是在一个函数的开头被声明的（例如 main 函数），它们的作用范围就是整个 main 函数。在左图的例子中，这就意味着如果在 main 函数外还另有一个函数，main 函数中声明的本地变量(Age, ANumber, AnotherOne) 不能够被另一个函数使用，反之亦然。

在 C++中 本地变量(local variable)的作用范围被定义在声明它的程序块内（一个程序块是被一对花括号（curly brackets{}）括起来的一组语句）。如果变量是在一个函数(function)中被声明的，那么它是一个函数范围内的变量，如果变量是在一个循环中(loop)中被声明的，那么它的作用范围只是在这个循环(loop)之中，以此类推。

除本地和全局范围外，还有一种外部范围，它使得一个变量不仅在同一源程序文件中可见，而且在其他所有将被链接在一起的源文件中均可见。

常量:字（Constants: Literals）

一个常量（constant）是一个有固定值的表达式。常量(constant)可以被分为整数(Integer Numbers)，浮点数(Floating-Point Numbers)，字符(Characters)和字符串(Strings)。

整数(Integer Numbers)

1776
707
-273

他们是整型常数，表示十进制整数值。注意表示整型常数时我们不需要些引号（quotes (")）或任何特殊字符。毫无疑问它是个常量：任何时候当我们在程序中写 1776，我们指的就是 1776 这个数值。

除十进制整数另外， C++还允许使用八进制（octal numbers）和十六进制（hexadecimal numbers）的字常量（literal constants）。如果我们想要表示一个八进制数，我们必须在它前面加上一个 0 字符(zero character)，而表示十六进制数我们需要在它前面加字符 0x (zero, x)。例如以下字常量（literal constants）互相等值：

```
75 // 十进制 decimal
0113 // 八进制 octal
0x4b // 十六进制 hexadecimal
```

所有这些都表示同一个整数： 75 (seventy five) ，分别以十进制数，八进制数和十六进制数表示。

[备注：你可以在文章 Numerical radixes 中看到更多关于十六进制和八进制表示方式的信息。]

浮点数(Floating Point Numbers)

浮点数以小数（decimals）和 / 或指数幂（ exponents）的形式表示。它们可以包括一个小数点，一个 e 字符（表示"by ten at the Xth height"，这里 X 是后面跟的整数值），或两者都包括。

```
3.14159 // 3.14159
6.02e23 // 6.02 x 1023
1.6e-19 // 1.6 x 10-19
3.0 // 3.0
```

以上是包含小数的以C++表示的4个有效数值。第一个是PI，第二个是Avogadro数之一，第三个是一个电子（electron）的电量（electric charge）（一个极小的数值） - 所有这些都是近似值。最后一个是浮点数字常量表示数3。

字符和字符串（Characters and strings）

此外还有非数字常量，例如：

```
'z'
'p'
"Hello world"
"How do you do?"
```

前两个表达式表示单独的字符（character），后面两个表示由若干字符组成的字符串(string) 。注意在表示单独字符的时候，我们用单引号（single quotes ('))，在表示字符串或多于一个字符的时候我们用双引号 (double quotes (")。

当以常量方式表示单个字符和字符串时，必须写上引号以便把他们和可能的变量标识或保留字区分开，注意以下例子：

```
x
'x'
```

x 指一个变量名称为 x，而'x'指字符常量'x'。

字符常量和字符串常量各有特点，例如 escape codes，这些是除此之外无法在源程序中表示的特殊的字符，例如换行符 newline (\n) 或跳跃符 tab (\t)。所有这些符号前面要加一个反斜杠 inverted slash (\)。这里列出了这些 escape codes：

\n	换行符 newline
\r	回车 carriage return
\t	跳跃符 tabulation
\v	垂直跳跃 vertical tabulation
\b	backspace
\f	page feed
\a	警告 alert (beep)
\'	单引号 single quotes (')
\"	双引号 double quotes (")

\?	问号 question (?)
\\	反斜杠 inverted slash (\)

例如：

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

另外你可以数字 ASCII 码表示一个字符，这种表示方式是在反斜杠(\)之后加以 8 进制数或十六进制数表示的 ASCII 码。在第一种（八进制 octal）表示中，数字必需紧跟反斜杠(例如\23 或\40)，第二种(十六进制 hexacedimal)，必须在数字之前写一个 x 字符(例如\x20 或\x4A)。

[关于此类 escape code 的更多信息，请参阅文件 ASCII Code]。

如果每一行代码以反斜杠 inverted slash (\)结束，字符串常量可以分多行代码表示：

```
"string expressed in \
two lines"
```

你还可以将多个被空格 blankspace、跳跃符 tabulator、换行符 newline 或其他有效空白符号分隔开的字符串常量连接在一起：

```
"we form" "a single" "string" "of characters"
```

定义常量 Defined constants (#define)

使用预处理器指令#define，你可以将那些你经常使用的常量定义为你自己取的名字而不需要借助于变量。它的格式是：

```
#define identifier value
```

例如：

```
#define PI 3.14159265
#define NEWLINE '\n'
#define WIDTH 100
```

以上定义了三个常量。一旦做了这些声明，你可以在后面的程序中使用这些常量，就像使用其它任何常量一样，例如：

```
circle = 2 * PI * r;
cout << NEWLINE;
```

实际上编译器在遇到#define 指令的时候做的只是把任何出现这些 常量名（在前面的例子中为 PI, NEWLINE 或 WIDTH）的地方替换成他们被定义为的代码（分别为 3.14159265, '\n'和 100）。因此，由#define 定义的常量被称为宏常量 macro constants。

#define 指令不是代码语句，它是预处理器指令，因此指令行末尾不需要加分号 semicolon (;) 。如果你在宏定义行末尾加了分号(;) ，当预处理器在程序中做常量替换的时候，分号也会被加到被替换的行中，这样可能导致错误。

声明常量 declared constants (const)

通过使用 const 前缀，你可以定义指定类型的常量，就像定义一个变量一样:

```
const int width = 100;
const char tab = '\t';
const zip = 12440;
```

如果没有指定类型(如上面最后例子中最后一行)，编译器会假设常量为整型 int。

1.3 操作符/运算符（Operators）

前面已经学习了变量和常量，我们可以开始操作他们。C++提供一系列的运算符，它们是一组关键字或非字母但是在所有键盘上都有的符号。运算符是 C++语言的基础，所以非常重要。

你不需要背下所有这一小节的内容，这些细节知识仅供你以后需要时参考 。

赋值 Assignation (=)

赋值运算符的功能是将一个值赋给一个变量。

```
a = 5;
```

将整数 5 赋给变量 a。= 运算符左边的部分叫做 lvalue (left value)，右边的部分叫做 rvalue (right value)。lvalue 必须是一个变量，而右边的部分可以是一个常量，一个变量，一个运算(operation)的结果或是前面几项的任意组合。

有必要强调赋值运算符永远是将右边的值赋给左边，永远不会反过来。

```
a = b;
```

将变量 b (rvalue)的值赋给变量 a (lvalue)，不论 a 当时存储的是什么值。同时考虑到我们只是将 b 的数值赋给 a，以后如果 b 的值改变了并不会影响到 a 的值.

例如：如果我们使用以下代码(变量值的变化显示在绿色注释部分):

```
int a, b; // a:? b:?
a = 10; // a:10 b:?
b = 4; // a:10 b:4
a = b; // a:4 b:4
b = 7; // a:4 b:7
```

以上代码结果是 a 的值为 4， b 的值为 7。最后一行中 b 的值被改变并不会影响到 a，虽然在此之前我们声明了 a = b; (从右到左规则 right-to-left rule)。

C++拥有而其他语言没有的一个特性是赋值符 (=) 可以被用作另一个赋值符的 rvalue (或 rvalue 的一部分) 。例如：

```
a = 2 + (b = 5);
```

等同于：

```
b = 5;
a = 2 + b;
```

它的意思是：先将 5 赋给变量 b，然后把前面对 b 的赋值运算的结果（即 5）加上 2 再赋给变量 a，这样最后 a 中的值为 7。因此，下面的表达式在 C++中也是正确的：

```
a = b = c = 5; //将 5 同时赋给 3 个变量 a, b 和 c。
```

数学运算符 Arithmetic operators (+, -, *, /, %)

C++语言支持的 5 种数学运算符为：

- + 加 addition
- - 减 subtraction
- * 乘 multiplication
- / 除 division
- % 取模 module

加减乘除运算想必大家都很了解，它们和一般的数学运算符没有区别。

唯一你可能不太熟悉的是用百分号(%)表示的取模运算（module）。取模运算是取两个整数相除的余数。例如，如果我们写 a = 11 % 3;，变量 a 的值将会为结果 2，因为 2 是 11 除以 3 的余数。

组合运算符 Compound assignation operators (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

C++以书写简练著称的一大特色就是这些组合运算符 compound assignation operators（+=, -=, *= 和 /= 及其他），这些运算符使得只用一个基本运算符就可改写变量的值：

value += increase; 等同于 value = value + increase;

a -= 5; 等同于 a = a - 5;

a /= b; 等同于 a = a / b;

price *= units + 1; 等同于 price = price * (units + 1);

其他运算符以此类推。

递增和递减 Increase and decrease

书写简练的另一个例子是递增（increase）运算符 (++)和递减（decrease） 运算符(--)。它们使得变量中存储的值加 1 或减 1。它们分别等同于+=1 和-=1。因此：

```
a++;
a+=1;
a=a+1;
```

在功能上全部等同，即全部使得变量 a 的值加 1。

它的存在是因为最早的 C 编译器将以上三种表达式的编译成不同的机器代码，不同的机器代码运行速度不一样。现在，编译器已经基本自动实行代码优化，所以以上三种不同的表达方式编译成的机器代码在实际运行上已基本相同。

这个运算符的一个特点是它既可以被用作 prefix 前缀，也可以被用作后缀 suffix，也就是说它既可以被写在变量标识的前面(++a)，也可以被写在后面(a++)。虽然在简单表达式如 a++或++a 中，这两种写法代表同样的意思，但当递增 increase 或递减 decrease 的运算结果被直接用在其它的运算式中时，它们就代表非常不同的意思了：当递增运算符被用作前缀 prefix（++a）时，变量 a 的

值线增加，然后再计算整个表达式的值，因此增加后的值被用在了表达式的计算中；当它被用作后缀 suffix (a++)时，变量 a 的值在表达式计算后才增加，因此 a 在增加前所存储的值被用在了表达式的计算中。注意以下两个例子的不同：

Example 1	Example 2
B=3; A=++B; // A is 4, B is 4	B=3; A=B++; // A is 3, B is 4

在第一个例子中，B 在它的值被赋给 A 之前增加 1。而在第二个例子中 B 原来的值 3 被赋给 A 然后 B 的值才加 1 变为 4。

关系运算符 Relational operators (==, !=, >, <, >=, <=)

我们用关系运算符来比较两个表达式。如 ANSI-C++ 标准中指出的，关系预算的结果是一个 bool 值，根据运算结果的不同，它的值只能是真 true 或 false。

例如我们想通过比较两个表达式来看它们是否相等或一个值是否比另一个的值大。以下为 C++的关系运算符：

==	相等 Equal
!=	不等 Different
>	大于 Greater than
<	小于 Less than
>=	大于等于 Greater or equal than
<=	小于等于 Less or equal than

下面你可以看到一些实际的例子：

(7 == 5)	将返回 false.
(5 > 4)	将返回 true.
(3 != 2)	将返回 true.
(6 >= 6)	将返回 true.
(5 < 5)	将返回 false.

当然，除了使用数字常量，我们也可以使用任何有效表达式，包括变量。假设有 a=2, b=3 和 c=6,

(a == 5)	将返回 false.
(a*b >= c)	将返回 true 因为它实际是(2*3 >= 6)
(b+4 > a*c)	将返回 false 因为它实际是(3+4 > 2*6)
((b=2) == a)	将返回 true.

注意:运算符= (单个等号)不同于运算符== (双等号)。第一个是赋值运算符(将等号右边的表达式值赋给左边的变量)；第二个(==)是一个判断等于的关系运算符，用来判断运算符两边的表达式是否相等。因此在上面例子中最后一个表达式((b=2) == a)，我们首先将数值 2 赋给变量 b，然后把它和变量 a 进行比较。因为变量 a 中存储的也是数值 2，所以整个运算的结果为 true。

在 ANSI-C++标准出现之前的许多编译器中，就像 C 语言中，关系运算并不返回值为真 true 或假 false 的 bool 值，而是返回一个整型数值最为结果，它的数值可以为 0，代表"false"或一个非 0 数值(通常为 1)来代表"true"。

逻辑运算符 Logic operators (!, &&, ||)

运算符 ！ 等同于 boolean 运算 NOT （取非），它只有一个操作数(operand)，写在它的右边。它做的唯一工作就是取该操作数的反面值，也就是说如果操作数值为真 true，那么运算后值变为假 false，如果操作数值为假 false，则运算结果为真 true。它就好像是说取与操作数相反的值。例如：

!(5 == 5)	返回 false，因为它右边的表达式(5 == 5)为真 true.
!(6 <= 4)	返回 true 因为(6 <= 4)为假 false.
!true	返回假 false.
!false	返回真 true.

逻辑运算符&&和||是用来计算两个表达式而获得一个结果值。它们分别对应逻辑运算中的与运算 AND 和或运算 OR。它们的运算结果取决于两个操作数(operand)的关系：

第一个操作数 a	第二个操作数 b	结果 a && b	结果 a || b

true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

例如 ：

((5 == 5) && (3 > 6)) 返回 false (true && false).
((5 == 5) || (3 > 6)) 返回 true (true || false).

条件运算符 Conditional operator (?)

条件运算符计算一个表达式的值并根据表达式的计算结果为真 true 或假 false 而返回不同值。它的格式是：

condition ? result1 : result2 （条件? 返回值 1：返回值 2）

如果条件 condition 为真 true，整个表达式将返回 esult1，否则将返回 result2。

7==5 ? 4 : 3	返回 3，因为 7 不等于 5.
7==5+2 ? 4 : 3	返回 4，因为 7 等于 5+2.
5>3 ? a : b	返回 a，因为 5 大于 3.
a>b ? a : b	返回较大值，a 或 b.

位运算符 Bitwise Operators (&, |, ^, ~, <<, >>)

位运算符以比特位改写变量存储的数值，也就是改写变量值的二进制表示：

op	asm	Description
&	AND	逻辑与 Logic AND
|	OR	逻辑或 Logic OR
^	XOR	逻辑异或 Logical exclusive OR
~	NOT	对 1 取补（位反转）Complement to one (bit inversion)
<<	SHL	左移 Shift Left
>>	SHR	右移 Shift Right

变量类型转换运算符 Explicit type casting operators

变量类型转换运算符可以将一种类型的数据转换为另一种类型的数据。在写 C++ 中有几种方法可以实现这种操作，最常用的一种，也是与 C 兼容的一种，是在原转换的表达式前面加用括号()括起的新数据类型：

```
int i;
float f = 3.14;
i = (int) f;
```

以上代码将浮点型数字 3.14 转换成一个整数值(3)。这里类型转换操作符为(int)。在 C++中实现这一操作的另一种方法是使用构造函数 constructor 的形式：在要转换的表达式前加变量类型并将表达式括在括号中：

```
i = int ( f );
```

以上两种类型转换的方法在 C++中都是合法的。另外 ANSI-C++针对面向对象编程（object oriented programming）增加了新的类型转换操作符（参考 Section 5.4, Advanced class type-casting).

sizeof()

这个运算符接受一个输入参数，该参数可以是一个变量类型或一个变量自己，返回该变量类型(variable type) 或对象(object)所占的字节数：

```
a = sizeof (char);
```

这将会返回 1 给 a，因为 char 是一个常为 1 个字节的变量类型。

sizeof 返回的值是一个常数，因此它总是在程序执行前就被固定了。

其它运算符

在本教程后面的章节里我们将看到更多的运算符，比如指向指针的运算或面向对象编程特有的运算，等等，我们会在它们各自的章节里进行详细讨论。

运算符的优先度 Priority of operators

当多个操作数组成复杂的表达式时，我们可能会疑惑哪个运算先被计算，哪个后被计算。例如以下表达式：

a = 5 + 7 % 2

我们可以怀疑它实际上表示：

a = 5 + (7 % 2) 结果为**6**，还是 a = (5 + 7) % 2 结果为**0**?

正确答案为第一个，结果为6。每一个运算符有一个固定的优先级，不仅对数学运算符（我们可能在学习数学的时候已经很了解它们的优先顺序了），所有在C++中出现的运算符都有优先级。从最从最高级到最低级，运算的优先级按下表排列：

优先级 Priority	运算符 Operator	描述 Description	结合方向 Associativity
1	::	范围 scope	Left
2	() [ ] -> . sizeof		Left
3	++ --	递增/递减 increment/decrement	Right
	~	求补 Complementto one (bitwise)	
	!	取非 unary NOT	
	& *	指针 Reference 和取地址 Dereference (pointers)	
	(type)	数据类型转换 Type casting	
	+ -	Unary less sign	
4	* / %	数学运算符 arithmetical operations	Left
5	+ -	数学运算符 arithmetical operations	Left
6	<< >>	位移 bit shifting (bitwise)	Left
7	< <= > >=	关系运算符 Relational operators	Left
8	== !=	关系运算符 Relational operators	Left
9	& ^ |	位操作 Bitwise operators	Left

10	&& ||	逻辑运算符 Logic operators	Left
11	?:	条件 Conditional	Right
12	= += -= *= /= %= >>= <<= &= ^= |=	赋值运算符 Assignation	Right
13	,	逗号 Comma, 分隔号 Separator	Left

结合方向 Associativity 定义了当有同优先级的多个运算符在一起时，哪一个必须被首先运算，最右边的还是最左边的。

所有这些运算符的优先级顺序可以通过使用括号 parenthesis signs (和)来控制，而且更易读懂，例如以下例子：

```
a = 5 + 7 % 2;
```

根据我们想要实现的计算的不同，可以写成：

```
a = 5 + (7 % 2); 或者
a = (5 + 7) % 2;
```

所以如果你想写一个复杂的表达式而不敢肯定各个运算的执行顺序，那么就加上括号。这样还可以使代码更易读懂。

1.4 控制台交互(Communication through console)

控制台(console)是电脑的最基本交互接口，通常包括键盘(keyboard)和屏幕(screen)。键盘通常为标准输入设备，而 屏幕为标准输出设备。

在 C++的 iostream 函数库中，一个程序的标准输入输出操作依靠两种数据流：cin 给输入使用和 cout 给输出使用。另外，cerr 和 clog 也已经被实现——它们是两种特殊设计的数据流专门用来显示出错信息。它们可以被重新定向到标准输出设备或到一个日志文件（log file）。

因此 cout (标准输出流)通常被定向到屏幕，而 cin (标准输入流)通常被定向到键盘。

通过控制这两种数据流，你可以在程序中与用户交互，因为你可以在屏幕上显示输出并从键盘接收用户的输入。

输出 Output (cout)

输出流 cout 与重载(overloaded)运算符<<一起使用：

```
cout << "Output sentence"; // 打印 Output sentence 到屏幕上
cout << 120; // 打印数字 120 到屏幕上
cout << x; // 打印变量 x 的值到屏幕上
```

运算符<<又叫插入运算符(insertion operator) 因为它将后面所跟的数据插入到它前面的数据流中。在以上的例子中，字符串常量 Output sentence，数字常量 120 和变量 x 先后被插入输出流 cout 中。注意第一句中字符串常量是被双引号引起来的。每当我们使用字符串常量的时候，必须用引号把字符串引起来，以便将它和变量名明显的区分开来。例如，下面两个语句是不同的：

```
cout << "Hello"; // 打印字符串 Hello 到屏幕上
cout << Hello; // 把变量 Hello 存储的内容打印到屏幕上
```

插入运算符 insertion operator (<<)可以在同一语句中被多次使用：

```
cout << "Hello, " << "I am " << "a C++ sentence";
```

上面这一行语句将会打印 **Hello, I am a C++ sentence** 到屏幕上。插入运算符(<<) 的重复使用在我们想要打印变量和内容的组合内容或多个变量时有所体现：

```
cout <<"Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

如果我们假设变量 age 的值为 24，变量 zipcode 的值为 90064，以上句子的输出将为： **Hello, I am 24 years old and my zipcode is 90064**

必须注意，除非我们明确指定，cout 并不会自动在其输出内容的末尾加换行符，因此下面的语句：

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

将会有如下内容输出到屏幕：

This is a sentence.This is another sentence.

虽然我们分别调用了两次 cout，两个句子还是被输出在同一行。所以，为了在输出中换行，我们必须插入一个换行符来明确表达这一要求。在 C++中换行符可以写作\n:

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

将会产生如下输出：

```
First sentence.
Second sentence.
Third sentence.
```

另外，你也可以用操作符 endl 来换行，例如：

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

将会输出：

```
First sentence.
Second sentence.
```

当操作符 endl 被用在 buffered streams 中时有一点特殊：它们被 flushed。不过 cout 默认为 unbuffered，所以不会被影响。 你可以暂时不管这一点。

你可以使用\n 或 endl 来指定 cout 输出换行，请注意前面所讲的两者的不同用法。

输入 Input (cin)

C++中的标准输入是通过在 cin 数据流上重载运算符 extraction (>>) 来实现的。它后面必须跟一个变量以便存储读入的数据。例如：

```
int age;
cin >> age;
```

声明一个整型变量 age 然后等待用户从键盘输入到 cin 并将输入值存储在这个变量中。

cin 只能在键盘输入回车键(RETURN)后才能处理前面输入的内容。因此即使你只要求输入一个单独的字符，在用户按下回车键(RETURN)之前 **cin** 将不会处理用户的输入的字符。

在使用 **cin** 输入的时候必须考虑后面的变量类型。如果你要求输入一个整数，extraction (>>) 后面必须跟一个整型变量，如果要求一个字符，后面必须跟一个字符型变量，如果要求一个字符串，后面必须跟一个字符串型变量。

```
// i/o example
#include <iostream.h>
int main ()
{
```

```
Please enter an integer
value: 702
The value you entered is
702 and its double is
```

```
int i;                                                    1404.
cout << "Please enter an integer value: ";
cin >> i;
cout << "The value you entered is " << i;
cout << " and its double is " << i*2 << ".\n";

return 0;
}
```

使用程序的用户可以使引起错误的原因之一，即使是在最简单的需要用 cin 做输入的程序中（就像我们上面看到的这个程序）。因为如果你要求输入一个整数数值，而用户输入了一个名字（一个字符串），其结果可能导致程序产生错误操作，因为它不是我们期望从用户处获得的数据。当你使用由 cin 输入的数据的时候，你不得不假设程序的用户将会完全合作而不会在程序要求输入整数的时候输入他的名字。后面当我们看到怎样使用字符串的时候，我们将会同时看到一些解决这一类出错问题的办法。

你也可以利用 cin 要求用户输入多个数据 ：

```
cin >> a >> b;
```

等同于：

```
cin >> a;
cin >> b;
```

在以上两种情况下用户都必须输入两个数据，一个给变量 a，一个给变量 b。输入时两个变量之间可以以任何有效的空白符号间隔，包括空格，跳跃符 tab 或换行。

2.1 控制结构（Control Structures）

一个程序的语句往往并不仅限于线性顺序结构。在程序的执行过程中它可能被分成两支执行，可能重复某些语句，也可能根据一些判断结果而执行不同的语句。因此 C++ 提供一些控制结构语句 (control structures) 来实现这些执行顺序。

为了介绍程序的执行顺序，我们需要先介绍一个新概念：语句块(block of instructions)。一个语句块(A block of instructions) 是一组互相之间由分号 semicolons (;) 分隔开但整体被花括号 curly bracket signs: { and }括起来的语句。

本节中我们将看到的大多数控制结构允许一个通用的 statement 做参数，这个 statement 根据需要可以是一条语句，也可以是一组语句组成的语句块。如果我

们只需要一条语句做 statement，它可以不被括在花括号（{}）内。但如果我们需要多条语句共同做 statement，则必须把它们括在花括号内（{}）以组成一个语句块。

条件结构 Conditional structure: if and else

条件结构用来实现仅在某种条件满足的情况下才执行一条语句或一个语句块。它的形式是：

```
if (condition) statement
```

这里 condition 是一个将被计算的表达式（expression）。如果表达式值为真，即条件(condition)为 true，statement 将被执行。否则，statement 将被忽略（不被执行），程序从整个条件结构之后的下一条语句继续执行。

例如，以下程序段实现只有当变量 x 存储的值确实为 100 的时候才输出"x is 100"：

```
if (x == 100)
cout << "x is 100";
```

如果我们需要在条件 condition 为真 true 的时候执行一条以上的语句，我们可以花括号{}将语句括起来组成一个语句块：

```
if (x == 100)
{
cout << "x is ";
cout << x;
}
```

我们可以用关键字 else 来指定当条件不能被满足时需要执行的语句，它需要和 if 一起使用，形式是：

```
if (condition) statement1 else statement2
```

例如：

```
if (x == 100)
cout << "x is 100";
else
cout << "x is not 100";
```

以上程序如果 x 的值为 100，则在屏幕上打出 x is 100，如果 x 不是 100，而且也只有在 x 不是 100 的时候，屏幕上将打出 x is not 100。

多个 if + else 的结构被连接起来使用来判断数值的范围。以下例子显示了如何用它来判断变量 x 中当前存储的数值是正值，负值还是既不正也不负，即等于 0 。

```
if (x > 0)
cout << "x is positive";
else if (x < 0)
cout << "x is negative";
else
cout << "x is 0";
```

记住当我们需要执行多条语句时，必须使用花括号{}将它们括起来以组成一个语句块 block of instructions。

重复结构 Repetitive structures 或循环 loops

循环 Loops 的目的是重复执行一组语句一定的次数或直到满足某种条件。

while 循环

格式是：

while（表达式 expression）语句 statement

它的功能是当 expression 的值为真 true 时重复执行 statement。

例如，下面我们将用 while 循环来写一个倒计数程序：

```
// custom countdown using while
#include <iostream.h>
int main ()
{
int n;
cout << "Enter the starting number > ";
cin >> n;
while (n>0) {
cout << n << ", ";
--n;
}
cout << "FIRE!";
return 0;
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

}

程序开始时提示用户输入一个倒计数的初始值。然后 while 循环开始，如果用户输入的数值满足条件 n>0（即 n 比 0 大），后面跟的语句块将会被执行一定的次数，直到条件（n>0）不再满足（变为 false）。

以上程序的所有处理过程可以用以下的描述来解释：

从 main 开始：

1. 用户输入一个数值赋给 n.
2. while 语句检查(n>0)是否成立，这时有两种可能：
   - true: 执行 statement (到第 3 步)
   - false: 跳过 statement. 程序直接执行第 5 步.
3. 执行 statement:
   cout << n << ", ";
   --n;

   (将 n 的值打印在屏幕上，然后将 n 的值减 1).

4. 语句块结束，自动返回第 2 步。
5. 继续执行语句块之后的程序：打印 FIRE! ，程序结束。

我们必须考虑到循环必须在某个点结束，因此在语句块之内（loop 的 statement 之内）我们必须提供一些方法使得条件 condition 可以在某个时刻变为假 false，否则循环将无限重复下去。在这个例子里，我们用语句--n;使得循环在重复一定的次数后变为 false ：当 n 变为 0， 倒计数结束。

do-while 循环

格式：

do 语句 statement while (条件 condition);

它的功能与 while 循环一抹一样，除了在 do-while 循环中是先执行 statement 然后才检查条件 condition ，而不想 while 循环中先检查条件然后才执行 statement。这样，即使条件 condition 从来没有被满足过，statement 仍至少被执行一次。例如，下面的程序重复输出（echoes）用户输入的任何数值，直到用户输入 0 为止。

```
// number echoer
#include <iostream.h>
```

```
Enter number (0 to end): 12345
You entered: 12345
```

```
int main ()
{
unsigned long n;
do {
cout << "Enter number (0 to end): ";
cin >> n;
cout << "You entered: " << n << "\n";
} while (n != 0);
return 0;
}
```

```
Enter number (0 to end):
160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

do-while 循环通常被用在判断循环结束的条件是在循环语句内部被决定的情况下，比如以上的例子，在循环的语句块内用户的输入决定了循环是否结束。如果用户永远不输入 0，则循环永远不会结束。

for 循环

格式是：

```
for (initialization; condition; increase) statement;
```

它的主要功能是当条件 condition 为真 true 时重复执行语句 statement ，类似 while 循环。但除此之外，for 还提供了写初始化语句 initialization 和增值语句 increase 的地方。因此这种循环结构是特别为执行由计数器控制的循环而设计的。

它按以下方式工作：

1. 执行初始化 initialization 。通常它是设置一个计数器变量（counter variable）的初始值，初始化仅被执行一次。
2. 检查条件 condition ，如果条件为真 true，继续循环，否则循环结束循环中语句 statement 被跳过。
3. 执行语句 statement 。像以前一样，它可以是一个单独的语句，也可以是一个由花括号{ }括起来的语句块。
4. 最后增值域（increase field）中的语句被执行，循环返回第 2 步。注意增值域中可能是任何语句，而不一定只是将计数器增加的语句。例如下面的例子中计数器实际为减 1，而不是加 1。

下面是用 for 循环实现的倒计数的例子：

```
// countdown using a for loop
#include <iostream.h>
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
FIRE!
```

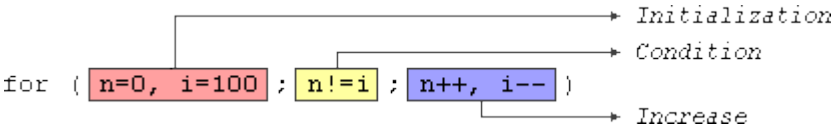
```
int main ()
{
for (int n=10; n>0; n--) {
cout << n << ", ";
}
cout << "FIRE!";
return 0;
}
```

初始化 initialization 和增值 increase 域是可选的（即可以为空）。但这些域为空的时候，它们和其他域之间间隔的分号不可以省略。例如我们可以写：for (;n<10;)来表示没有初始化和增值语句；或 for （;n<10;n++）来表示有增值语句但没有初始化语句。

另外我们也可以在 for 循环初始化或增值域中放一条以上的语句，中间用逗号 coma(,)隔开。例如假设我们想在循环中初始化一个以上的变量，可以用以下的程序来实现：

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
// whatever here...
}
```

这个循环将被执行 50 次，如果 n 和 i 在循还内部都不被改变的话：



n 初始值为 0，i 初始值为 100，条件是(n!=i)(即 n 不能等于 i)。因为每次循环 n 加 1，而且 i 减 1，循环的条件将会在第 50 次循环之后变为假 false(n 和 i 都等于 50）。

分支控制和跳转(Bifurcation of control and jumps)

break 语句

通过使用 break 语句，即使在结束条件没有满足的情况下，我们也可以跳出一个循环。它可以被用来结束一个无限循环（infinite loop），或强迫循环在其自然结束之前结束。例如，我们想要在倒计数自然结束之前强迫它停止（也许因为一个引擎故障）：

```
// break loop example                10, 9, 8, 7, 6, 5, 4, 3,
#include <iostream.h>                countdown aborted!
int main ()
{
int i;
for (n=10; n>0; n--) {
cout << n << ", ";
if (n==3)
{
cout << "countdown aborted!";
break;
}
return 0;
}
```

continue 语句

continue 语句使得程序跳过当前循环中剩下的部分而直接进入下一次循环，就好像循环中语句块的结尾已经到了使得循环进入下一次重复。例如，下面例子中倒计数时我们将跳过数字 5 的输出：

```
// continue loop example      10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
#include <iostream.h>
int main ()
{
for (int n=10; n>0; n--) {
if (n==5) continue;
cout << n << ", ";
}
cout << "FIRE!";
return 0;
}
```

goto 语句

通过使用 goto 语句可以使程序从一点跳转到另外一点。你必须谨慎只用这条语句，因为它的执行可以忽略任何嵌套限制。

跳转的目标点可以由一个标示符(label)来标明，该标示符作为 goto 语句的参数。一个标示符(label)由一个标识名称后面跟一个冒号 colon (:)组成。

通常除了底层程序爱好者使用这条语句，它在结构化或面向对象的编程中并不常用。下面的例子中我们用 goto 来实现倒计数循环：

```
// goto loop example        10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
#include <iostream.h>       FIRE!
int main ()
{
int n=10;
loop:
cout << n << ", ";
n--;
if (n>0) goto loop;
cout << "FIRE!";
return 0;
}
```

exit 函数

exit 是一个在 cstdlib (stdlib.h)库中定义的函数。

exit 的目的是一个特定的退出代码来结束程序的运行，它的原型(prototype)是：

```
void exit (int exit code);
```

exit code 是由操作系统使用或被调用程序使用。通常 exit code 为 0 表示程序正常结束，任何其他值表示程序执行过程中出现了错误。

选择结构 The selective Structure: switch

switch 语句的语法比较特殊。它的目标是对一个表达式检查多个可能常量值，有些像我们在本节开头学习的把几个 if 和 else if 语句连接起来的结构。它的形式是：

```
switch (expression) {
case constant1:
block of instructions 1
break;
case constant2:
block of instructions 2
break;
```

.
.
.
default:
default block of instructions
}

它按以下方式执行：

switch 计算表达式 expression 的值，并检查它是否与第一个常量 constant1 相等，如果相等，程序执行常量 1 后面的语句块 block of instructions 1 直到碰到关键字 break ，程序跳转到 switch 选择结构的结尾处。

如果 expression 不等于 constant1，程序检查表达式 expression 的值是否等于第二个常量 constant2， 如果相等，程序将执行常量 2 后面的语句块 block of instructions 2 直到碰到关键字 break。

依此类推，直到最后如果表达式 expression 的值不等于任何前面的常量（你可以用 case 语句指明任意数量的常量值来要求检查），程序将执行默认区 default: 后面的语句，如果它存在的话。default: 选项是可以省略的。

下面的两段代码段功能相同：

***switch* example**	***if-else* equivalent**

switch (x) {
case 1:
cout << "x is 1";
break;
case 2:
cout << "x is 2";
break;
default:
cout << "value of x unknown";
}
	if (x == 1) {
	cout << "x is 1";
	}
	else if (x == 2) {
	cout << "x is 2";
	}
	else {
	cout << "value of x unknown";
	}

前面已经提到 switch 的语法有点特殊。注意每个语句块结尾包含的 break 语句。这是必须的，因为如果不这样做，例如在语句块 block of instructions 1 的结尾没有 break，程序执行将不会跳转到 switch 选择的结尾处（}），而是继续执行下面的语句块，直到第一次遇到 break 语句或到 switch 选择结构的结尾。因此，不需要在每一个 case 域内加花括号{ } 。这个特点同时可以帮助实现对不同的可能值执行相同的语句块。例如：

switch (x) {

```
case 1:
case 2:
case 3:
cout << "x is 1, 2 or 3";
break;
default:
cout << "x is not 1, 2 nor 3";
}
```

注意 switch 只能被用来比较表达式和不同常量的值 constants。因此我们不能够把变量或范围放在 case 之后，例如 (case (n*2):) 或 (case (1..3):) 都不可以，因为它们不是有效的常量。 如果你需要检查范围或非常量数值，使用连续的 if 和 else if 语句。

2.2 函数 I（Functions I）

通过使用函数(functions)我们可以把我们的程序以更模块化的形式组织起来，从而利用 C++所能提供的所有结构化编程的潜力。

一个函数(function)是一个可以从程序其它地方调用执行的语句块。以下是它的格式：

```
type name ( argument1, argument2, ...) statement
```

这里：

- type 是函数返回的数据的类型
- name 是函数被调用时使用的名
- argument 是函数调用需要传入的参量(可以声明任意多个参量)。每个参量(argument)由一个数据类型后面跟一个标识名称组成，就像变量声明中一样(例如，int x)。参量仅在函数范围内有效，可以和函数中的其它变量一样使用， 它们使得函数在被调用时可以传入参数，不同的参数用逗号(comma)隔开.
- statement 是函数的内容。它可以是一句指令，也可以是一组指令组成的语句块。如果是一组指令，则语句块必须用花括号{}括起来，这也是我们最常见到情况。其实为了使程序的格式更加统一清晰，建议在仅有一条指令的时候也使用花括号，这是一个良好的编程习惯。

下面看一下第一个函数的例子：

```
// function example                    The result is 8
```

```
#include <iostream.h>
int addition (int a, int b)
{
       int r;
       r=a+b;
       return (r);
}

int main ()
{
       int z;
       z = addition (5,3);
       cout << "The result is " << z;

       return 0;
}
```

记得在我们教程开始时说过：一个 C++程序总是从 main 函数开始执行。 因此我们从那里开始。

我们可以看到 main 函数以定义一个整型变量 z 开始。紧跟着我们看到调用 addition 函数。我们可以看到函数调用的写法和上面函数定义本身十分相似：

```
int addition (int a, int b)
                 ↑     ↑
z = addition (   5   ,   3   );
```

参数有明显的对应关系。在 main 函数中我们调用 addition 函数，并传入两个数值： 5 和 3 ， 它们对应函数 addition 中定义的参数 int a 和 int b。

当函数在 main 中被调用时，程序执行的控制权从 main 转移到函数 addition。调用传递的两个参数的数值 (5 和 3) 被复制到函数的本地变量（local variables） int a 和 int b 中。

函数 addition 中定义了新的变量(int r;),通过表达式 r=a+b;, 它把 a 加 b 的结果赋给 r 。因为传过来的参数 a 和 b 的值分别为 5 和 3 ，所以结果是 8。

下面一行代码：

```
return (r);
```

结束函数 addition，并把控制权交还给调用它的函数(main) ，从调用 addition 的地方开始继续向下执行。另外，return 在调用的时候后面跟着变量 r (return (r);)，它当时的值为 8，这个值被称为函数的返回值。

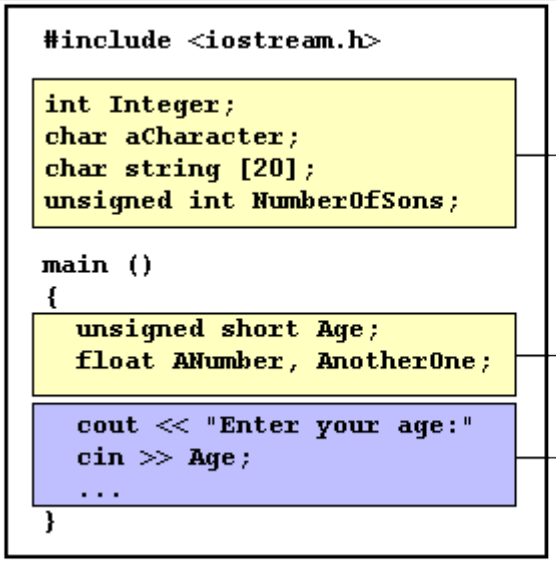
```
int addition (int a, int b)
  |8
  v
z = addition (  5  ,  3  );
```

函数返回的数值就是函数的计算结果，因此， z 将存储函数 addition (5, 3) 返回的数值，即 8。用另一种方式解释，你也可以想象成调用函数(addition (5,3)) 被替换成了它的返回值 (8)。

接下来 main 中的下一行代码是：

```
cout << "The result is " << z;
```

它把结果打印在屏幕上。



变量的范围(Scope of variables)

你必须考虑到变量的范围只是在定义该变量的函数或指令块内有效，而不能在它的函数或指令块之外使用。 例如，在上面的例子里就不可能在 main 中直接使用变量 a, b 或 r ，因为它们是函数 addition 的本地变量（local variable）。在函数 addition 中也不可能直接使用变量 z，因为它是 main 的本地变量。

因此，本地变量 (local variables)的范围是局限于声明它的嵌套范围之内的。尽管如此，你还可以定义全局变量(global variables)，它们可以在代码的任何位置被访问，不管在函数以内还是以外。要定义全局变量，你必须在所有函数或代码块之外定义它们，也就是说，直接在程序体中声明它们。

这里是另一个关于函数的例子：

```
// function example                          The first result is 5
#include <iostream.h>                        The second result is 5
int subtraction (int a, int b)               The third result is 2
{                                            The fourth result is 6
       int r;
       r=a-b;
       return (r);
}

int main ()
{
       int x=5, y=3, z;
       z = subtraction (7,2);
       cout << "The first result is " << z <<
'\n';
       cout << "The second result is " <<
subtraction (7,2) << '\n';
       cout << "The third result is " <<
subtraction (x,y) << '\n';
       z= 4 + subtraction (x,y);
       cout << "The fourth result is " << z <<
'\n';
       return 0;
}
```

在这个例子中，我们定义了函数 subtraction。这个函数的功能是计算传入的两个参数的差值并将结果返回。

在 main 函数中，函数 subtraction 被调用了多次。我们用了几种不同的调用方法，因此你可以看到在不同的情况下函数如何被调用。

为了更好的理解这些例子，你需要考虑到被调用的函数其实完全可以由它所返回的值来代替。例如在上面例子中第一种情况下 (这种调用你应该已经知道了，因为我们在前面的例子中已经用过这种形式的调用):

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

如果我们把函数调用用它的结果(也就是 5)替换，我们将得到：

```
z = 5;
cout << "The first result is " << z;
```

同样的

```
cout << "The second result is " << subtraction (7,2);
```

与前面的调用有同样的结果，但在这里我们把对函数 subtraction 的调用直接用作 cout 的参数。这可以简单想象成我们写的是：

```
cout << "The second result is " << 5;
```

因为 5 是 subtraction (7,2)的结果。

在

```
cout << "The third result is " << subtraction (x,y);
```

中，与前面的调用唯一的不同之处是这里调用 subtraction 时的参数使用的是变量而不是常量。这样用时毫无问题的。在这个例子里，传入函数 subtraction 的参数值是变量 x 和 y 中存储的数值，即分别为 5 和 3，结果为 2。

第四种调用也是一样的。只要知道除了

```
z = 4 + subtraction (x,y);
```

我们也可以写成：

```
z = subtraction (x,y) + 4;
```

它们的结果是完全一样的。注意在整个表达式的结尾写上分号 semicolon sign (;)。它并不需要总是跟在函数调用的后面，因为你可以有一次把它们想象成函数被它的结果所替代：

```
z = 4 + 2;
z = 2 + 4;
```

没有返回值类型的函数，使用 void.

如果你记得函数声明的格式：

```
type name ( argument1, argument2 ...) statement
```

就会知道函数声明必须以一个数据类型（type）开头，它是函数由 return 语句所返回数据类型。但是如果我们并不打算返回任何数据那该怎么办呢？

假设我们要写一个函数，它的功能是打印在屏幕上打印一些信息。我们不需要它返回任何值，而且我们也不需要它接受任何参数。C 语言为这些情况设计了 void 类型。让我们看一下下面的例子：

```
// void function example          I'm a function!
#include <iostream.h>
void dummyfunction (void)
{
       cout << "I'm a
function!";
}

int main ()
{
       dummyfunction ();
       return 0;
}
```

虽然在 C++ 中 void 可以被省略，我们还是建议写出 void，以便明确指出函数不需要参数。

你必须时刻知道的是调用一个函数时要写出它的名字并把参数写在后面的括号内。但如果函数不需要参数，后面的括号并不能省略。因此调用函数 dummyfunction 的格式是

```
dummyfunction ();
```

这就明确指出它是一个函数调用，而不是一个变量名称或其它什么。

2.3 函数 II（Functions II）

参数按数值传递和按地址传递(Arguments passed by value and by reference)

到目前为止，我们看到的所有函数中，传递到函数中的参数全部是按数值传递的(by value)。也就是说，当我们调用一个带有参数的函数时，我们传递到函数中的是变量的数值而不是变量本身。 例如，假设我们用下面的代码调用我们的第一个函数 addition ：

```
int x=5, y=3, z;
z = addition ( x , y );
```

在这个例子里我们调用函数 addition 同时将 **x** 和 **y** 的值传给它，即分别为 **5** 和 **3**，而不是两个变量：

```
int addition (int a, int b)
                 ↑5     ↑3
z = addition (  x  ,   y  );
```

这样，当函数 addition 被调用时，它的变量 a 和 b 的值分别变为 5 和 3，但在函数 addition 内对变量 a 或 b 所做的任何修改不会影响变量他外面的变量 x 和 y 的值，因为变量 x 和 y 并没有把它们自己传递给函数，而只是传递了他们的数值。

但在某些情况下你可能需要在一个函数内控制一个函数以外的变量。要实现这种操作，我们必须使用按地址传递的参数（arguments passed by reference），就象下面例子中的函数 duplicate：

```
// passing parameters by reference                x=2, y=6, z=14
#include <iostream.h>

void duplicate (int& a, int& b, int& c)
{
a*=2;
b*=2;
c*=2;
}

int main ()
{
int x=1, y=3, z=7;
duplicate (x, y, z);
cout << "x=" << x << ", y=" << y << ", z=" <<
z;
return 0;
}
```

第一个应该注意的事项是在函数 duplicate 的声明(declaration)中，每一个变量的类型后面跟了一个地址符 ampersand sign (&)，它的作用是指明变量是按地址传递的（by reference），而不是像通常一样按数值传递的（by value）。

当按地址传递（pass by reference）一个变量的时候，我们是在传递这个变量本身，我们在函数中对变量所做的任何修改将会影响到函数外面被传递的变量。

```
void duplicate (int& a,int& b,int& c)
                  ↕x    ↕y    ↕z
     duplicate (  x  ,  y  ,  z  );
```

用另一种方式来说，我们已经把变量 a，b，c 和调用函数时使用的参数(x，y 和 z)联系起来了，因此如果我们在函数内对 a 进行操作，函数外面的 x 值也会改变。同样，任何对 b 的改变也会影响 y，对 c 的改变也会影响 z>。

这就是为什么上面的程序中，主程序 main 中的三个变量 x，y 和 z 在调用函数 duplicate 后打印结果显示他们的值增加了一倍。

如果在声明下面的函数：

```
void duplicate (int& a, int& b, int& c)
```

时，我们是按这样声明的：

```
void duplicate (int a, int b, int c)
```

也就是不写地址符 ampersand (&)，我们也就没有将参数的地址传递给函数，而是传递了它们的值，因此，屏幕上显示的输出结果 x，y ，z 的值将不会改变，仍是 1，3，7。

这种用地址符 ampersand (&)来声明按地址"by reference"传递参数的方式只是在 C++中适用。在 C 语言中，我们必须用指针(pointers)来做相同的操作。

按地址传递(Passing by reference)是一个使函数返回多个值的有效方法。例如，下面是一个函数，它可以返回第一个输入参数的前一个和后一个数值。

```
// more than one returning value                Previous=99, Next=101
#include <iostream.h>
void prevnext (int x, int& prev, int& next)

{
prev = x-1;
next = x+1;
}

int main ()
{
int x=100, y, z;
prevnext (x, y, z);
cout << "Previous=" << y << ", Next=" << z;

return 0;
}
```

参数的默认值(Default values in arguments)

当声明一个函数的时候我们可以给每一个参数指定一个默认值。如果当函数被调用时没有给出该参数的值，那么这个默认值将被使用。指定参数默认值只需要在函数声明时把一个数值赋给参数。如果函数被调用时没有数值传递给该参数，那么默认值将被使用。但如果有指定的数值传递给参数，那么默认值将被指定的数值取代。例如：

```
// default values in functions   6
                                 5
#include <iostream.h>
int divide (int a, int b=2) {
int r;
r=a/b;
return (r);
}

int main () {
cout << divide (12);
cout << endl;
cout << divide (20,4);
return 0;
}
```

我们可以看到在程序中有两次调用函数 **divide**。第一次调用：

```
divide (12)
```

只有一个参数被指明，但函数 **divide** 允许有两个参数。因此函数 **divide** 假设第二个参数的值为 **2**，因为我们已经定义了它为该参数缺省的默认值(注意函数声明中的 int b=2)。因此这次函数调用的结果是 **6** (12/2)。

在第二次调用中：

```
divide (20,4)
```

这里有两个参数，所以默认值 (int b=2) 被传入的参数值 4 所取代，使得最后结果为 5 (20/4).

函数重载(Overloaded functions)

两个不同的函数可以用同样的名字，只要它们的参量(arguments)的原型(prototype)不同，也就是说你可以把同一个名字给多个函数，如果它们用不同数量的参数，或不同类型的参数。例如：

```
// overloaded function                2
#include <iostream.h>                 2.5

int divide (int a, int b) {
return (a/b);
}

float divide (float a, float b) {

return (a/b);
}

int main () {
int x=5,y=2;
float n=5.0,m=2.0;
cout << divide (x,y);
cout << "\n";
cout << divide (n,m);
cout << "\n";
return 0;
}
```

在这个例子里，我们用同一个名字定义了两个不同函数，当它们其中一个接受两个整型(int)参数，另一个则接受两个浮点型(float)参数。编译器 (compiler)通过检查传入的参数的类型来确定是哪一个函数被调用。如果调用传入的是两个整数参数，那么是原型定义中有两个整型(int)参量的函数被调用，如果传入的是两个浮点数，那么是原型定义中有两个浮点型(float)参量的函数被调用。

为了简单起见，这里我们用的两个函数的代码相同，但这并不是必须的。你可以让两个函数用同一个名字同时完成完全不同的操作。

Inline 函数（inline functions）

inline 指令可以被放在函数声明之前，要求该函数必须在被调用的地方以代码形式被编译。这相当于一个宏定义(macro)。它的好处只对短小的函数有效，这种情况下因为避免了调用函数的一些常规操作的时间(overhead)，如参数堆栈操作的时间，所以编译结果的运行代码会更快一些。

它的声明形式是：

```
inline type name ( arguments ... ) { instructions ... }
```

它的调用和其他的函数调用一样。调用函数的时候并不需要写关键字 inline ，只有在函数声明前需要写。

递归（Recursivity）

递归(recursivity)指函数将被自己调用的特点。它对排序(sorting)和阶乘(factorial)运算很有用。例如要获得一个数字 n 的阶乘，它的数学公式是：

n! = n * (n-1) * (n-2) * (n-3) ... * 1

更具体一些，5! (factorial of 5) 是：

5! = 5 * 4 * 3 * 2 * 1 = 120

而用一个递归函数来实现这个运算将如以下代码：

```
// factorial calculator
#include <iostream.h>

long factorial (long a){
if (a > 1) return (a * factorial (a-1));
else return (1);
}

int main () {
long l;
cout << "Type a number: ";
cin >> l;
cout << "!" << l << " = " << factorial (l);

return 0;
}
```

```
Type a number: 9
!9 = 362880
```

注意我们在函数 factorial 中是怎样调用它自己的，但只是在参数值大于 1 的时候才做调用，因为否则函数会进入死循环(an infinite recursive loop)，当参数到达 0 的时候，函数不继续用负数乘下去(最终可能导致运行时的堆栈溢出错误(stack overflow error)。

这个函数有一定的局限性，为简单起见，函数设计中使用的数据类型为长整型(long)。在实际的标准系统中，长整型 long 无法存储 12！以上的阶乘值。

函数原型（Prototyping functions）

到目前为止，我们定义的所有函数都是在它们第一次被调用（通常是在 main 中）之前，而把 main 函数放在最后。如果重复以上几个例子，但把 main 函数放在其它被它调用的函数之前，你就会遇到编译错误。原因是在调用一个函数之前，函数必须已经被定义了，就像我们前面例子中所做的。

但实际上还有一种方法来避免在 main 或其它函数之前写出所有被他们调用的函数的代码，那就是通过定义函数原型(prototyping functions)。定义函数原型就是对函数在的完整定义之前做一个短小重要的声明，以便让编译器了解函数的参数和返回值类型。

它的形式是：

```
type name ( argument_type1, argument_type2, ...);
```

它与一个函数的头定义（header definition）一样，除了：

- 它不包括函数的内容，也就是它不包括函数后面花括号{}内的所有语句。
- 它以一个分号 semicolon sign (;) 结束。
- 在参数列举中只需要写出各个参数的数据类型就够了，至于每个参数的名字可以写，也可以不写，但是我们建议写上。

例如：

```
// prototyping
#include <iostream.h>

void odd (int a);
void even (int a);

int main () {
int i;
do {
cout << "Type a number: (0 to exit)";
cin >> i;
odd (i);
} while (i!=0);
return 0;
}

void odd (int a) {
if ((a%2)!=0) cout << "Number is
```

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030

Number is even.
Type a number (0 to exit): 0
Number is even.
```

```
odd.\n";
else even (a);
}

void even (int a) {
if ((a%2)==0) cout << "Number is
even.\n";
else odd (a);
}
```

这个例子的确不是很有效率，我相信现在你已经可以只用一半行数的代码来完成同样的功能。但这个例子显示了函数原型（prototyping functions）是怎样工作的。并且在这个具体的例子中，两个函数中至少有一个是必须定义原型的。

这里我们首先看到的是函数 odd 和 even 的原型：

```
void odd (int a);
void even (int a);
```

这样使得这两个函数可以在它们被完整定义之前就被使用，例如在 main 中被调用，这样 main 就可以被放在逻辑上更合理的位置：即程序代码的开头部分。

尽管如此，这个程序需要至少一个函数原型定义的特殊原因是因为在 odd 函数里需要调用 even 函数，而在 even 函数里也同样需要调用 odd 函数。如果两个函数任何一个都没被提前定义原型的话，就会出现编译错误，因为或者 odd 在 even 函数中是不可见的(因为它还没有被定义)，或者 even 函数在 odd 函数中是不可见的。

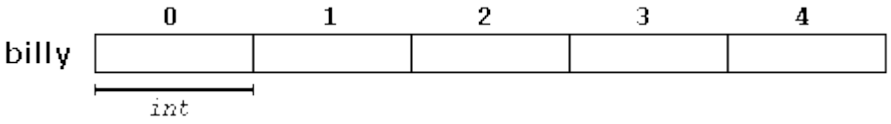
很多程序员建议给所有的函数定义原型。这也是我的建议，特别是在有很多函数或函数很长的情况下。把所有函数的原型定义放在一个地方，可以使我们在决定怎样调用这些函数的时候轻松一些，同时也有助于生成头文件。

3.1 数组 （Arrays）

数组(Arrays) 是在内存中连续存储的一组同种数据类型的元素（变量），每一数组有一个唯一名称，通过在名称后面加索引（index）的方式可以引用它的每一个元素。

也就是说，例如我们有 5 个整型数值需要存储，但我们不需要定义 5 个不同的变量名称，而是用一个数组(array)来存储这 5 个不同的数值。注意数组中的元素必须是同一数据类型的，在这个例子中为整型（int）。

例如一个存储 5 个整数叫做 billy 的数组可以用下图来表示：



这里每一个空白框代表数组的一个元素，在这个例子中为一个整数值。白框上面的数字 0 到 4 代表元素的索引(index)。注意无论数组的长度如何，它的第一个元素的索引总是从 0 开始的。

同其它的变量一样， 数组必须先被声明然后才能被使用。一种典型的数组声明显示如下：

```
type name [elements];
```

这里 type 是可以使任何一种有效的对象数据类型(object type)，如 int, float...等，name 是一个有效地变量标识(identifier)，而由中括号[]引起来的 elements 域指明数组的大小，即可以存储多少个元素。

因此，要定义上面图中显示的 billy 数组，用一下语句就可以了：

```
int billy [5];
```

备注：在定义一个数组的时候，中括号[]中的 elements 域必须是一个常量数值，因为数组是内存中一块有固定大小的静态空间，编译器必须在编译所有相关指令之前先能够确定要给该数组分配多少内存空间。

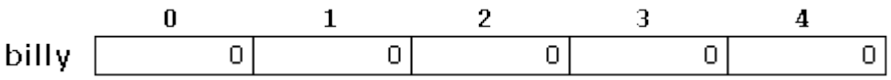
初始化数组(Initializing arrays)

当声明一个本地范围内（在一个函数内）的数组时，除非我们特别指定，否则数组将不会被初始化，因此它的内容在我们将数值存储进去之前是不定的。

如果我们声明一个全局数组（在所有函数之外），则它的内容将被初始化为所有元素均为 0。因此 ，如果全局范围内我们声明：

```
int billy [5];
```

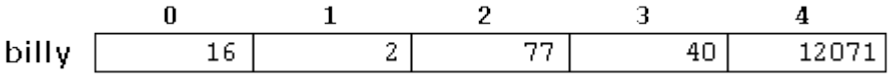
那么 billy 中的每一个元素将会被初始化为 0:



另外，我们还可以在声明一个变量的同时把初始值付给数组中的每一个元素，这个赋值用花括号{ }来完成。例如：

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

这个声明将生成如下数组：



花括号中我们要初始化的元素数值个数必须和数组声明时方括号[ ]中指定的数组长度相符。例如，在上面例子中数组 billy 声明中的长度为 5，因此在后面花括号中的初始值也有 5 个，每个元素一个数值。

因为这是一种信息的重复，因此 C++允许在这种情况下数组[ ]中为空白，而数组的长度将有后面花括号{}中数值的个数来决定，如下例所示。

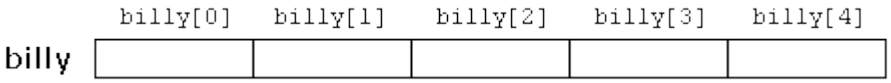
```
int billy [] = { 16, 2, 77, 40, 12071 };
```

存取数组中数值（Access to the values of an Array）

在程序中我们可以读取和修改数组任一元素的数值，就像操作其他普通变量一样。格式如下：

```
name[index]
```

继续上面的例子，数组 billy 有 5 个元素，其中每一元素都是整型 int，我们引用其中每一个元素的名字分别为如下所示：



例如，要把数值 75 存入数组 billy 中第 3 个元素的语句可以是：

```
billy[2] = 75;
```

又例如，要把数组 billy 中第 3 个元素的值赋给变量 a，我们可以这样写：

```
a = billy[2];
```

因此，在所有使用中，表达式 billy[2]就像任何其他整型变量一样。

注意数组 billy 的第 3 个元素为 billy[2]，因为索引(index)从 0 开始，第 1 个元素是 billy[0]，第 2 个元素是 billy[1]，因此第 3 个是 billy[2]。同样的原因，最后一个元素是 billy[4]。如果我们写 billy[5]，那么是在使用 billy 的第 6 个元素，因此会超出数组的长度。

在 C++ 中对数组使用超出范围的 index 是合法的，这就会产生问题，因为它不会产生编译错误而不易被察觉，但是在运行时会产生意想不到的结果，甚至导致严重运行错误。超出范围的 index 之所以合法的原因我们在后面学习指针(pointer)的时候会了解。

学到这里，我们必须能够清楚的了解方括号[ ]在对数组操作中的两种不同用法。它们完成两种任务：一种是在声明数组的时候定义数组的长度；另一种是在引用具体的数组元素的时候指明一个索引号(index)。我们要注意不要把这两种用法混淆。

```
int billy[5]; // 声明新数组(以数据类型名称开头)
billy[2] = 75; // 存储数组的一个元素
```

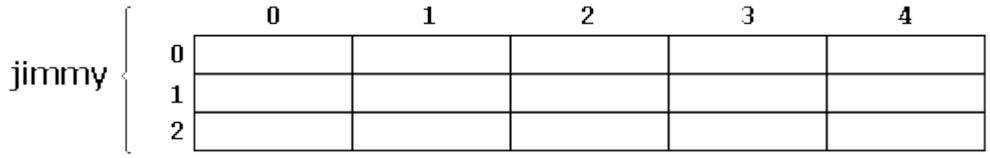
其它合法的数组操作：

```
billy[0] = a; // a 为一个整型变量
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// arrays example                                12206
#include <iostream.h>

int billy [ ] = {16, 2, 77, 40, 12071};

int n, result=0;

int main () {
for ( n=0 ; n<5 ; n++ ) {
result += billy[n];
}

cout << result;
return 0;
}
```

多维数组（Multidimensional Arrays）

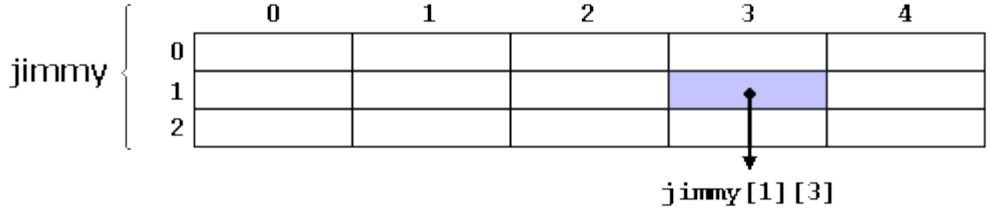
多维数组(Multidimensional Arrays)可以被描述为数组的数组。例如，一个 2 维数组(bidimensional array)可以被想象成一个有同一数据类型的 2 维表格。



jimmy 显示了一个整型(int)的 3x5 二维数组，声明这一数组的的方式是：

```
int jimmy [3][5];
```

而引用这一数组中第 2 列第 4 排元素的表达式为：**jimmy[1][3]**



(记住数组的索引总是从 0 开始)。

多维数组(Multidimensional arrays)并不局限于 2 维。如果需要，它可以有任意多维，虽然需要 3 维以上的时候并不多。但是考虑一下一个有很多维的数组所需要的内存空间，例如：

```
char century [100][365][24][60][60];
```

给一个世纪中的每一秒赋一个字符(char)，那么就是多于 30 亿的字符！如果我们定义这样一个数组，需要消耗 3000M 的内存。

多维数组只是一个抽象的概念，因为我们只需要把各个索引的乘积放入一个简单的数组中就可以获得同样的结果。例如：

```
int jimmy [3][5]; 效果上等价于
int jimmy [15]; (3 * 5 = 15)
```

唯一的区别是编译器帮我们记住每一个想象中的维度的深度。下面的例子中我们就可以看到，两段代码一个使用 2 维数组，另一个使用简单数组，都获得同样的结果，即都在内存中开辟了一块叫做 jimmy 的空间，这个空间有 15 个连续地址位置，程序结束后都在相同的位置上存储了相同的数值，如后面图中所示：

```
// multidimensional array    // pseudo-multidimensional array
```

```
#include <iostream.h>
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];

int n,m;

int main (){
for (n=0;n
for (m=0;m
jimmy[n][m]=(n+1)*(m+1);
}

return 0;
}
```

```
#include <iostream.h>
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main (){
for (n=0;n
for (m=0;m
jimmy[n * WIDTH + m]=(n+1)*(m+1);

}
return 0;
}
```

上面两段代码并不向屏幕输出，但都向内存中的叫做 jimmy 的内存块存入如下数值：

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

我们用了宏定义常量(#define)来简化未来可能出现的程序修改，例如，如果我们决定将数组的纵向由 3 扩大到 4，只需要将代码行：

```
#define HEIGHT 3
```

修改为 ：

```
#define HEIGHT 4
```

而不需要对程序的其他部分作任何修改。

数组参数（Arrays as parameters）

有时候我们需要将数组作为参数传给函数。在 C++ 中将一整块内存中的数值作为参数完整的传递给一个函数是不可能的，即使是一个规整的数组也不可能，但是允许传递它的地址。它们的实际作用是一样的，但传递地址更快速有效。

要定义数组为参数，我们只需要在声明函数的时候指明参数数组的基本数据类型，一个标识后面再跟一对空括号[]就可以了。例如以下的函数：

```
void procedure (int arg[])
```

接受一个叫做 arg 的整型数组为参数。为了给这个函数传递一个按如下定义的数组：

```
int myarray [40];
```

其调用方式可写为：

```
procedure (myarray);
```

下面我们来看一个完整的例子:

```
// arrays as parameters                    5 10 15
#include <iostream.h>                      2 4 6 8 10

void printarray (int arg[], int length) {

for (int n=0; n
cout << arg[n] << " ";

cout << "\n";
}

int main () {
int firstarray[ ] = {5, 10, 15};
int secondarray[ ] = {2, 4, 6, 8, 10};
printarray (firstarray,3);
printarray (secondarray,5);
return 0;
}
```

可以看到，函数的第一个参数(int arg[ ])接受任何整型数组为参数，不管其长度如何。因此，我们用了第 2 个参数来告知函数我们传给它的第一个参数数组的长度。这样函数中打印数组内容的 for 循环才能知道需要检查的数组范围。

在函数的声明中也包含多维数组参数。定义一个 3 维数组 (tridimensional array) 的形式是：

```
base_type[ ][depth][depth]
```

例如，一个函数包含多维数组参数的函数可以定义为：

```
void procedure (int myarray[ ][3][4])
```

注意第一对括号[ ]中为空，而后面两对不为空。这是必须的，因为编译器必须能够在函数中确定每一个增加的维度的深度。

数组作为函数的参数，不管是多维数组还是简单数组，都是初级程序员容易出错的地方。建议阅读章节 3.3, 指针(Pointers)，以便更好的理解数组(arrays)是如何操作的。

3.2 字符串 （Strings of Characters）

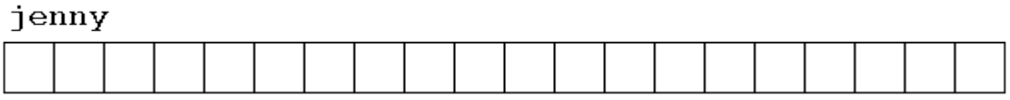
迄今为止我们看到的程序都是使用的数字型变量（numerical variables）。但除了数字变量，还有字符串变量，我们可以用来表示连续的字符，如词，句子，名称，文本等等。到目前为止我们只是使用常量表示它们，其实我们还没有考虑过用变量来存储他们。

在 C++ 中并没有一个单独的变量类型来专门存储字符串。为了完成这项功能，我们使用由连续字符(char)元素组成的字符数组。记住字符型数据(char) 是用来存储一个单个字符的，因此它的数组通常被用来存储字符串。

例如，如下数组(或字符串)：

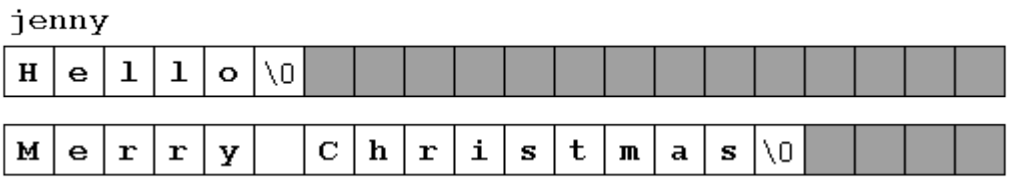
```
char jenny [20];
```

可以存储一个最多 20 个字符长的字符串。你可以把它想象成：



这 20 个字符并不一定总是满的。例如，jenny 在程序的某一点可以只存储字符串"Hello" 或者"Merry christmas"。因此，既然字符数组经常被用于存储短于其总长的字符串，就形成了一种习惯在字符串的有效内容的结尾处加一个空字符(null character)来表示字符结束，它的常量表示可写为 0 或'\0'。

我们可以用下图表示 jenny （一个长度为 20 的字符数组） 存储字符串"Hello" 和"Merry Christmas" ：



注意在有效内容结尾是如何用空字符 null character ('\0')来表示字符串结束的。 后面灰色的空格表示不确定数值。

字符串初始化(Initialization of strings)

因为字符串其实是普通数组，它与数组遵守同样的规则。例如，如果我们想将数组初始化为指定数值，我们可以像初始化其它数组一样用：

```
char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

在这里我们定义了一个有 6 个元素的字符数组,并将它初始化为字符串 Hello 加一个空字符（null character '\0'）。

除此之外，字符串还有另一个方法来进行初始化：用字符串常量。

在前几章的例子中，字符串常量已经出现过多次，它们是由双引号引起来的一组字符来表示的，例如：

```
"the result is: "
```

是一个字符串常量，我们在前面的例子中已经使用过。

与表示单个字符常量的单引号(')不同,双引号 (")是表示一串连续字符的常量。由双引号引起来的字符串末尾总是会被自动加上一个空字符 ('\0') 。

因此，我们可以用下面两种方法的任何一种来初始化字符串 mystring：

```
char mystring [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char mystring [ ] = "Hello";
```

在两种情况下字符串或数组 mystring 都被定义为 6 个字符长(元素类型为字符 char)：组成 Hello 的 5 个字符加上最后的空字符('\0')。在第二种用双引号的情况下，空字符('\0')是被自动加上的。

注意：同时给数组赋多个值只有在数组初始化时，也就是在声明数组时，才是合法的。象下面代码现实的表达式都是错误的：

```
mystring = "Hello";
mystring[ ] = "Hello";
mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

因此记住：我们只有在数组初始化时才能够同时赋多个值给它。其原因在学习了指针（pointer）之后会比较容易理解，因为那时你会看到一个数组其实只是一个指向被分配的内存块的常量指针（constant pointer），数组自己不能够被赋予任何数值，但我们可以给数组中的每一个元素赋值。

在数组初始化的时候是特殊情况，因为它不是一个赋值，虽然同样使用了等号(=) 。不管怎样，牢记前面标下画线的规则。

给字符串赋值(Assigning values to strings)

因为赋值运算的 lvalue 只能是数组的一个元素，而不能使整个数组，所以，用以下方式将一个字符串赋给一个字符数组是合法的：

```
mystring[0] = 'H';
mystring[1] = 'e';
mystring[2] = 'l';
mystring[3] = 'l';
mystring[4] = 'o';
mystring[5] = '\0';
```

但正如你可能想到的，这并不是一个实用的方法。通常给数组赋值，或更具体些，给字符串赋值的方法是使用一些函数，例如 strcpy。strcpy (string copy) 在函数库 cstring (string.h) 中被定义，可以用以下方式被调用：

```
strcpy (string1, string2);
```

这个函数将 string2 中的内容拷贝给 string1。string2 可以是一个数组，一个指针，或一个字符串常量 constant string。因此用下面的代码可以将字符串常量"Hello"赋给 mystring：

```
strcpy (mystring, "Hello");
```

例如：

```
// setting value to string       J. Soulie
#include <iostream.h>
#include <string.h>
```

```
int main () {
char szMyName [20];
strcpy (szMyName,"J. Soulie");

cout << szMyName;
return 0;
}
```

注意：我们需要包括头文件才能够使用函数 **strcpy**。

虽然我们通常可以写一个像下面 **setstring** 一样的简单程序来完成与 **cstring** 中 **strcpy** 同样的操作：

```
// setting value to string                          J. Soulie
#include <iostream.h>

void setstring (char szOut [ ], char szIn [ ])
{
int n=0;
do {
szOut[n] = szIn[n];
} while (szIn[n++] != '\0');
}

int main () {
char szMyName [20];
setstring (szMyName,"J. Soulie");
cout << szMyName;
return 0;
}
```

另一个给数组赋值的常用方法是直接使用输入流(cin)。在这种情况下，字符串的数值是在程序运行时由用户输入的。

当 cin 被用来输入字符串值时，它通常与函数 getline 一起使用，方法如下：

```
cin.getline ( char buffer[], int length, char delimiter = ' \n');
```

这里 buffer 是用来存储输入的地址（例如一个数组名），length 是一个缓存 buffer 的最大容量，而 delimiter 是用来判断用户输入结束的字符，它的默认值（如果我们不写这个参数时）是换行符 newline character ('\n')。

下面的例子重复输出用户在键盘上的任何输入。这个例子简单的显示了如何使用 cin.getline 来输入字符串：

```
// cin with strings
#include <iostream.h>

int main () {
char mybuffer [100];
cout << "What's your name? ";
cin.getline (mybuffer,100);
cout << "Hello " << mybuffer << ".\n";
cout << "Which is your favourite team? ";
cin.getline (mybuffer,100);
cout << "I like " << mybuffer << " too.\n";

return 0;
}
```

```
What's your name? Juan
Hello Juan.
Which is your favourite
team? Inter Milan
I like Inter Milan too.
```

注意上面例子中两次调用 cin.getline 时我们都使用了同一个字符串标识(mybuffer)。程序在第二次调用时将新输入的内容直接覆盖到第一次输入到 buffer 中的内容。

你可能还记得，在以前与控制台(console)交互的程序中，我们使用 extraction operator (>>) 来直接从标准输入设备接收数据。这个方法也同样可以被用来输入字符串，例如，在上面的例子中我们也可以用以下代码来读取用户输入：

```
cin >> mybuffer;
```

这种方法也可以工作，但它有以下局限性是 cin.getline 所没有的：

- 它只能接收单独的词(而不能是完整的句子)，因为这种方法以任何空白符为分隔符，包括空格 spaces，跳跃符 tabulators，换行符 newlines 和回车符 arriage returns。
- 它不能给 buffer 指定容量，这使得程序不稳定，如果用户输入超出数组长度，输入信息会被丢失。

因此，建议在需要用 cin 来输入字符串时，使用 cin.getline 来代替 cin >>。

字符串和其它数据类型的转换(Converting strings to other types)

鉴于字符串可能包含其他数据类型的内容，例如数字，将字符串内容转换成数字型变量的功能会有用处。例如一个字符串的内容可能是"1977"，但这一个 5 个字

符组成序列，并不容易转换为一个单独的整数。因此，函数库 cstdlib (stdlib.h) 提供了 3 个有用的函数：

- atoi: 将字符串 string 转换为整型 int
- atol: 将字符串 string 转换为长整型 long
- atof: 将字符串 string 转换为浮点型 float

所有这些函数接受一个参数，返回一个指定类型的数据(int, long 或 float)。这三个函数与 cin.getline 一起使用来获得用户输入的数值，比传统的 cin>> 方法更可靠：

```
// cin and ato* functions
#include <iostream.h>
#include <stdlib.h>

int main () {
char mybuffer [100];
float price;
int quantity;
cout << "Enter price: ";
cin.getline (mybuffer,100);
price = atof (mybuffer);
cout << "Enter quantity: ";
cin.getline (mybuffer,100);
quantity = atoi (mybuffer);
cout << "Total price: " << price*quantity;

return 0;
}
```

```
Enter price: 2.75
Enter quantity: 21
Total price: 57.75
```

字符串操作函数(Functions to manipulate strings)

函数库 cstring (string.h) 定义了许多可以像 C 语言类似的处理字符串的函数(如前面已经解释过的函数 strcpy)。这里再简单列举一些最常用的：

- strcat: char* strcat (char* dest, const char* src); //将字符串 src 附加到字符串 dest 的末尾，返回 dest。
- strcmp: int strcmp (const char* string1, const char* string2); //比较两个字符串 string1 和 string2。如果两个字符串相等，返回 0。
- strcpy: char* strcpy (char* dest, const char* src); //将字符串 src 的内容拷贝给 dest，返回 dest 。
- strlen: size_t strlen (const char* string); //返回字符串的长度。

注意：char* 与 char[] 相同。

关于 3.3 指针 （Pointers）

我们已经明白变量其实是可以由标识来存取的内存单元。但这些变量实际上是存储在内存中具体的位置上的。对我们的程序来说，计算机内存只是一串连续的单字节单元(1byte cell)，即最小数据单位，每一个单元有一个唯一地址。

计算机内存就好像城市中的街道。在一条街上，所有的房子被顺序编号，每所房子有唯一编号。因此如果我们说芝麻街 27 号，我们很容易找到它，因为只有一所房子会是这个编号，而且我们知道它会在 26 号和 28 号之间。

同房屋按街道地址编号一样，操作系统(operating system)也按照唯一顺序编号来组织内存。因此，当我们说内存中的位置 1776，我们知道内存中只有一个位置是这个地址，而且它在地址 1775 和 1777 之间。

地址操作符/去引操作符 Address/dereference operator (&)

当我们声明一个变量的同时，它必须被存储到内存中一个具体的单元中。通常我们并不会指定变量被存储到哪个具体的单元中—幸亏这通常是由编译器和操作系统自动完成的，但一旦操作系统指定了一个地址，有些时候我们可能会想知道变量被存储在哪里了。

这可以通过在变量标识前面加与符号 ampersand sign (&)来实现，它表示"...的地址" ("address of")，因此称为地址操作符(adress operator)，又称去引操作符(dereference operator)。例如：

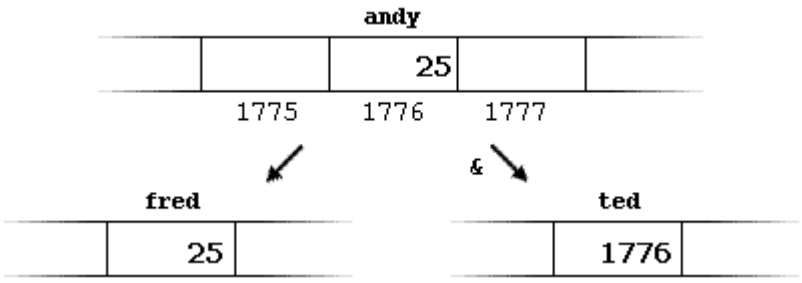
```
ted = &andy;
```

将变量 andy 的地址赋给变量 ted,因为当在变量名称 andy 前面加 ampersand (&)符号，我们指的将不再是该变量的内容，而是它在内存中的地址。

假设 andy 被放在了内存中地址 1776 的单元中，然后我们有下列代码：

```
andy = 25;
fred = andy;
ted = &andy;
```

其结果显示在下面的图片中：

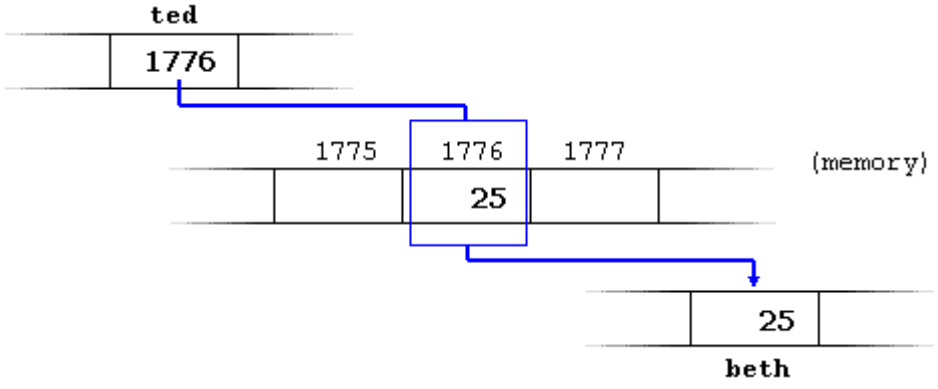


我们将变量 andy 的值赋给变量 fred，这与以前我们看到很多例子都相同，但对于 ted，我们把操作系统存储 andy 的内存地址赋给它，我们想像该地址为 1776 (它可以是任何地址，这里只是一个假设的地址)，原因是当给 ted 赋值的时候，我们在 andy 前面加了 ampersand (&) 符号。

存储其它变量地址的变量(如上面例子中的 ted)，我们称之为指针(pointer)。在 C++ 中，指针 pointers 有其特定的优点，因此经常被使用。在后面我们将会看到这种变量如何被声明。

引用操作符 Reference operator (*)

使用指针的时候，我们可以通过在指针标识的前面加星号 asterisk (*)来存储该指针指向的变量所存储的数值，它可以被翻译为“所指向的数值”（"value pointed by"）。因此，仍用前面例子中的数值，如果我们写： beth = *ted; (我们可以读作："beth 等与 ted 所指向的数值") beth 将会获得数值 25，因为 ted 是 1776，而 1776 所指向的数值为 25。



你必须清楚的区分 ted 存储的是 1776，但*ted (前面加 asterisk *) 指的是地址 1776 中存储的数值，即 25。注意加或不加星号*的不同(下面代码中注释显示了如何读这两个不同的表达式)：

```
beth = ted; // beth 等于 ted ( 1776 )
beth = *ted; // beth 等于 ted 所指向的数值 ( 25 )
```

地址或反引用操作符 Operator of address or dereference (&)

它被用作一个变量前缀，可以被翻译为“…的地址”("address of")，因此：&variable1 可以被读作 variable1 的地址("address of variable1")。

引用操作符 Operator of reference (*)

它表示要取的是表达式所表示的地址指向的内容。它可以被翻译为“…指向的数值” ("value pointed by")。

* mypointer 可以被读作 "mypointer 指向的数值"。

继续使用上面开始的例子，看下面的代码：

```
andy = 25;
ted = &andy;
```

现在你应该可以清楚的看到以下等式全部成立：

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

第一个表达式很容易理解，因为我们有赋值语句 andy=25;。第二个表达式使用了地址（或反引用）操作符(&) 来返回变量 andy 的地址，即 1776。第三个表达式很明显成立，因为第二个表达式为真，而我们给 ted 赋值的语句为 ted = &andy;。第四个表达式使用了引用操作符 (*)，相当于 ted 指向的地址中存储的数值，即 25。

由此你也可以推断出，只要 ted 所指向的地址中存储的数值不变，以下表达式也为真：

```
*ted == andy
```

声明指针型变量 Declaring variables of type pointer

由于指针可以直接引用它所指向的数值，因此有必要在声明指针的时候指明它所指向的数据类型。指向一个整型 int 或浮点型 float 数据的指针与指向一个字符型 char 数据的指针并不相同。

因此，声明指针的格式如下：

```
type * pointer_name;
```

这里，type 是指针所指向的数据的类型，而不是指针自己的类型。例如：

```
int * number;
char * character;
float * greatnumber;
```

它们是 3 个指针的声明，每一个指针指向一种不同数据类型。这三个指针本身其实在内存中占用同样大小的内存空间（指针的大小取决于不同的操作系统），但它们所指向的数据是不同的类型，并占用不同大小的内存空间，一个是整型 int，一个是字符型 char ，还有一个是浮点型 float。

需要强调的一点是，在指针声明时的星号 asterisk (*) 仅表示这里声明的是一个指针，不要把它和前面我们用过的引用操作符混淆，虽然那也是写成一个星号(*)。它们只是用同一符号表示的两个不同任务。

```
// my first pointer                                value1==10 / value2==20
#include <iostream.h>

int main ( ) {
int value1 = 5, value2 = 15;
int * mypointer;
mypointer = &value1;
*mypointer = 10;
mypointer = &value2;
*mypointer = 20;
cout << "value1==" << value1 << "/ value2==" <<
value2;
return 0;
}
```

注意变量 value1 和 value2 是怎样间接的被改变数值的。首先我们使用 ampersand sign (&) 将 value1 的地址赋给 mypointer 。然后我们将 10 赋给 mypointer 所指向的数值，它其实指向 value1 的地址，因此，我们间接的修改了 value1 的数值。

为了让你了解在同一个程序中一个指针可以被用作不同的数值，我们在这个程序中用 value2 和同一个指针重复了上面的过程。

下面是一个更复杂一些的例子：

```
// more pointers                                   value1==10 / value2==20
```

```
#include <iostream.h>

int main () {
int value1 = 5, value2 = 15;
int *p1, *p2;
p1 = &value1; // p1 = address of value1
p2 = &value2; // p2 = address of value2
*p1 = 10; // value pointed by p1 = 10
*p2 = *p1; // value pointed by p2 = value pointed
by p1
p1 = p2; // p1 = p2 (value of pointer copied)

*p1 = 20; // value pointed by p1 = 20
cout << "value1==" << value1 << "/ value2==" <<
value2;
return 0;
}
```

上面每一行都有注释说明代码的意思：ampersand (&) 为"address of"，asterisk (*) 为 "value pointed by"。注意有些包含 p1 和 p2 的表达式不带星号。加不加星号的含义十分不同：星号(*)后面跟指针名称表示指针所指向的地方，而指针名称不加星号(*)表示指针本身的数值，即它所指向的地方的地址。

另一个需要注意的地方是这一行：

```
int *p1, *p2;
```

声明了上例用到的两个指针，每个带一个星号(*)，因为是这一行定义的所有指针都是整型 int (而不是 int*)。原因是引用操作符(*) 的优先级顺序与类型声明的相同，因此，由于它们都是向右结合的操作，星号被优先计算。我们在 section 1.3: Operators 中已经讨论过这些。注意在声明每一个指针的时候前面加上星号 asterisk (*)。

指针和数组 Pointers and arrays

数组的概念与指针的概念联系非常解密。其实数组的标识相当于它的第一个元素的地址，就像一个指针相当于它所指向的第一个元素的地址，因此其实它们是同一个东西。例如，假设我们有以下声明：

```
int numbers [20];
int * p;
```

下面的赋值为合法的：

```
p = numbers;
```

这里指针 p 和 numbers 是等价的，它们有相同的属性，唯一的不同是我们可以给指针 p 赋其它的数值,而 numbers 总是指向被定义的 20 个整数组中的第一个。所以，p 只是一个普通的指针变量，而与之不同，numbers 是一个指针常量(constant pointer)，数组名的确是一个指针常量。因此虽然前面的赋值表达式是合法的，但下面的不是：

```
numbers = p;
```

因为 numbers 是一个数组(指针常量)，常量标识不可以被赋其它数值。

由于变量的特性，以下例子中所有包含指针的表达式都是合法的：

```
// more pointers                10, 20, 30, 40, 50,
#include <iostream.h>

int main () {
int numbers[5];
int * p;
p = numbers;
*p = 10;
p++;
*p = 20;
p = &numbers[2];
*p = 30;
p = numbers + 3;
*p = 40;
p = numbers;
*(p+4) = 50;
for (int n=0; n<5; n++)
cout << numbers[n] << ", ";

return 0;
}
```

在数组一章中我们使用了括号[]来指明我们要引用的数组元素的索引(index)。中括号[]也叫位移(offset)操作符，它相当于在指针中的地址上加上括号中的数字。例如，下面两个表达式互相等价：

```
a[5] = 0; // a [offset of 5] = 0
*(a+5) = 0; // pointed by (a+5) = 0
```

不管 a 是一个指针还是一个数组名， 这两个表达式都是合法的。

指针初始化 Pointer initialization

当声明一个指针的时候我们可能需要同时指定它们指向哪个变量，

```
int number;
int *tommy = &number;
```

这相当于：

```
int number;
int *tommy;
tommy = &number;
```

当给一个指针赋值的时候，我们总是赋给它一个地址值，而不是它所指向数据的值。你必须考虑到在声明一个指针的时候，星号 (*) 只是用来指明它是指针，而从不表示引用操作符 reference operator (*)。记住，它们是两种不同操作，虽然它们写成同样的符号。因此，我们要注意不要将以上的代码与下面的代码混淆：

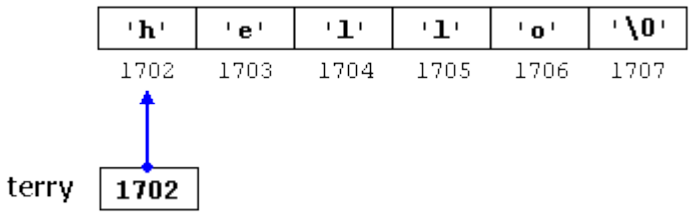
```
int number;
int *tommy;
*tommy = &number;
```

这些代码也没有什么实际意义。

在定义数组指针的时候，编译器允许我们在声明变量指针的同时对数组进行初始化，初始化的内容需要是常量，例如：

```
char * terry = "hello";
```

在这个例子中，内存中预留了存储"hello" 的空间，并且 terry 被赋予了向这个内存块的第一个字符（对应’ h’ ）的指针。假设"hello"存储在地址 1702，下图显示了上面的定义在内存中状态：

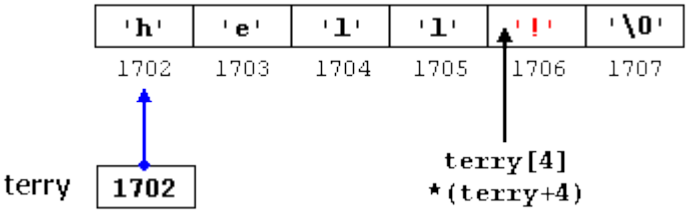


这里需要强调，terry 存储的是数值 1702 ，而不是'h' 或 "hello"，虽然 1702 指向这些字符。

指针 terry 指向一个字符串，可以被当作数组一样使用（数组只是一个常量指针）。例如，如果我们的心情变了，而想把 terry 指向的内容中的字符'o' 变为符号'!' ，我们可以用以下两种方式的任何一种来实现：

```
terry[4] = '!';
*(terry+4) = '!';
```

记住写 terry[4] 与*(terry+4)是一样的，虽然第一种表达方式更常用一些。以上两个表达式都会实现以下改变：



指针的数学运算 Arithmetic of pointers

对指针进行数学运算与其他整型数据类型进行数学运算稍有不同。首先，对指针只有加法和减法运算，其它运算在指针世界里没有意义。但是指针的加法和减法的具体运算根据它所指向的数据的类型的大小的不同而有所不同。

我们知道不同的数据类型在内存中占用的存储空间是不一样的。例如，对于整型数据，字符 char 占用 1 的字节(1 byte)，短整型 short 占用 2 个字节，长整型 long 占用 4 个字节。

假设我们有 3 个指针：

```
char *mychar;
short *myshort;
long *mylong;
```

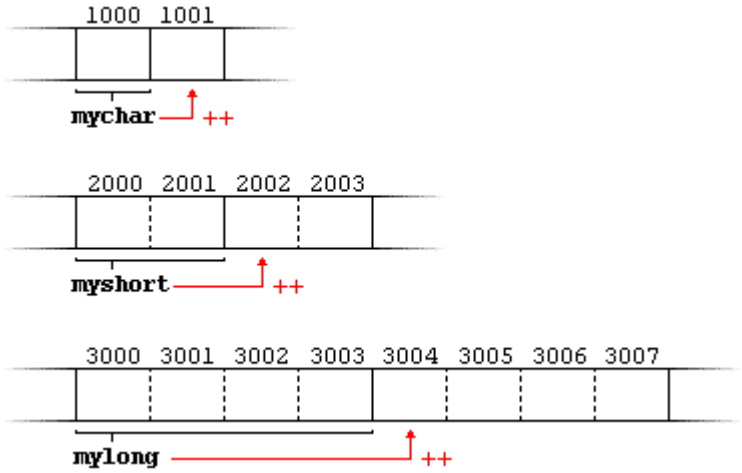
而且我们知道他们分别指向内存地址 1000, 2000 和 3000。

因此如果我们有以下代码：

```
mychar++;
myshort++;
mylong++;
```

就像你可能想到的，mychar 的值将会变为 1001。而 myshort 的值将会变为 2002，mylong 的值将会变为 3004。原因是当我们给指针加 1 时，我们实际是让该指针

指向下一个与它被定义的数据类型的相同的元素。因此，它所指向的数据类型的长度字节数将会被加到指针的数值上。以上过程可以由下图表示：



这一点对指针的加法和减法运算都适用。如果我们写以下代码，它们与上面例子的作用一抹一样： mychar = mychar + 1;

```
myshort = myshort + 1;
mylong = mylong + 1;
```

这里需要提醒你的是，递增 (++) 和递减 (--) 操作符比引用操作符 reference operator (*)有更高的优先级，因此，以下的表达式有可能引起歧义：

```
*p++;
*p++ = *q++;
```

第一个表达式等同于*(p++) ，它所作的是增加 p (它所指向的地址，而不是它存储的数值)。

在第二个表达式中，因为两个递增操作(++) 都是在整个表达式被计算之后进行而不是在之前，所以*q 的值首先被赋予*p ，然后 q 和 p 都增加 1。它相当于：

```
*p = *q;
p++;
q++;
```

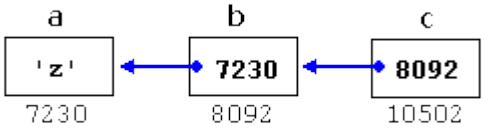
像通常一样，我们建议使用括号()以避免意想不到的结果。

指针的指针 Pointers to pointers

C++ 允许使用指向指针的指针。要做到这一点，我们只需要在每一层引用之前加星号(*)即可：

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

假设随机选择内存地址为 7230, 8092 和 10502，以上例子可以用下图表示：



(方框内为变量的内容；方框下面为内存地址)

这个例子中新的元素是变量 c，关于它我们可以从 3 个方面来讨论，每一个方面对应了不同的数值：

c 是一个(char **)类型的变量，它的值是 8092

c 是一个(char)类型的变量，它的值是 7230

**c 是一个(char)类型的变量，它的值是'z'

空指针 void pointers

指针 void 是一种特殊类型的指针。void 指针可以指向任意类型的数据，可以是整数，浮点数甚至字符串。唯一个限制是被指向的数值不可以被直接引用（不可以对它们使用引用星号*），因为它的长度是不定的，因此，必须使用类型转换操作或赋值操作来把 void 指针指向一个具体的数据类型。

它的应用之一是被用来给函数传递通用参数：

```
// integer increaser                                    6, 10, 13
#include <iostream.h>

void increase (void* data, int type) {
switch (type) {
```

```
case sizeof(char) : (*((char*)data))++; break;

case sizeof(short): (*((short*)data))++;
break;
case sizeof(long) : (*((long*)data))++; break;

}
}

int main () {
char a = 5;
short b = 9;
long c = 12;
increase (&a,sizeof(a));
increase (&b,sizeof(b));
increase (&c,sizeof(c));
cout << (int) a << ", " << b << ", " << c;
return 0;
}
```

sizeof 是 C++的一个操作符，用来返回其参数的长度字节数常量。例如，sizeof(char) 返回 1，因为 char 类型是 1 字节长数据类型。

函数指针 Pointers to functions

C++ 允许对指向函数的指针进行操作。它最大的作用是把一个函数作为参数传递给另外一个函数。声明一个函数指针像声明一个函数原型一样，除了函数的名字需要被括在括号内并在前面加星号 asterisk (*)。例如：

```
// pointer to functions                          8
#include <iostream.h>

int addition (int a, int b) {
       return (a+b);
}

int subtraction (int a, int b) {
       return (a-b);
}
```

```
int (*minus)(int,int) = subtraction;
int operation (int x, int y, int
(*functocall)(int,int)) {
        int g;
        g = (*functocall)(x,y);
        return (g);
}

int main () {
        int m,n;
        m = operation (7, 5, addition);
        n = operation (20, m, minus);
        cout <<n;
        return 0;
}
```

在这个例子里， minus 是一个全局指针，指向一个有两个整型参数的函数，它被赋值指向函数 subtraction，所有这些由一行代码实现：

```
int (* minus)(int,int) = subtraction;
```

这里似乎解释的不太清楚，有问题问为什么(int int)只有类型，没有参数，就再多说两句。

这里 int (*minus)(int int)实际是在定义一个指针变量，这个指针的名字叫做 minus，这个指针的类型是指向一个函数，函数的类型是有两个整型参数并返回一个整型值。

整句话“int (*minus)(int,int) = subtraction;”是定义了这样一个指针并把函数 subtraction 的值赋给它，也就是说有了这个定义后 minus 就代表了函数 subtraction。因此括号中的两个 int int 实际只是一种变量类型的声明，也就是说是一种形式参数而不是实际参数。

这个函数库的更多信息，参阅 C++ Reference 。

3.4 动态内存分配 （Dynamic memory）

到目前为止，我们的程序中我们只用了声明变量、数组和其他对象（objects）所必需的内存空间，这些内存空间的大小都在程序执行之前就已经确定了。但如果我们需要内存大小为一个变量，其数值只有在程序运行时 (runtime)才能确定，例如有些情况下我们需要根据用户输入来决定必需的内存空间，那么我们该怎么办呢？

答案是动态内存分配（dynamic memory），为此 C++ 集成了操作符 new 和 delete。

操作符 **new** 和 **delete** 是 C++执行指令。本节后面将会介绍这些操作符在 C 中的等价命令。

操作符 new 和 new[ ]

操作符 new 的存在是为了要求动态内存。new 后面跟一个数据类型，并跟一对可选的方括号[ ]里面为要求的元素数。它返回一个指向内存块开始位置的指针。其形式为：

```
pointer = new type
```

或者

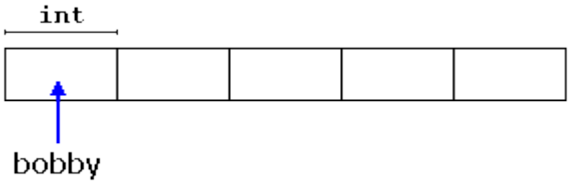
```
pointer = new type [elements]
```

第一个表达式用来给一个单元素的数据类型分配内存。第二个表达式用来给一个数组分配内存。

例如：

```
int * bobby;
bobby = new int [5];
```

在这个例子里，操作系统分配了可存储 5 个整型 int 元素的内存空间，返回指向这块空间开始位置的指针并将它赋给 bobby。因此，现在 bobby 指向一块可存储 5 个整型元素的合法的内存空间，如下图所示。



你可能会问我们刚才所作的给指针分配内存空间与定义一个普通的数组有什么不同。最重要的不同是，数组的长度必须是一个常量，这就将它的大小在程序执行之前的设计阶段就被决定了。而采用动态内存分配，数组的长度可以常量或变量，其值可以在程序执行过程中再确定。

动态内存分配通常由操作系统控制，在多任务的环境中，它可以被多个应用（applications）共享，因此内存有可能被用光。如果这种情况发生，操作系统将不能在遇到操作符 new 时分配所需的内存，一个无效指针(null pointer)将被返回。因此，我们建议在使用 new 之后总是检查返回的指针是否为空(null)，如下例所示：

```
int * bobby;
bobby = new int [5];
if (bobby == NULL) {
// error assigning memory. Take measures.
};
```

删除操作符 delete

既然动态分配的内存只是在程序运行的某一具体阶段才有用，那么一旦它不再被需要时就应该被释放，以便给后面的内存申请使用。操作符 delete 因此而产生，它的形式是：

```
delete pointer;
```

或

```
delete [ ] pointer;
```

第一种表达形式用来删除给单个元素分配的内存，第二种表达形式用来删除多元素（数组）的内存分配。在多数编译器中两种表达式等价，使用没有区别， 虽然它们实际上是两种不同的操作，需要考虑操作符重载 overloading （我们在后面的 section 4.2 节中将会看到）。

```
// rememb-o-matic
#include ‹iostream.h›
#include ‹stdlib.h›

int main ( ) {
char input [100];
int i,n;
long * l;
cout << "How many numbers do you want to
type in? ";
cin.getline (input,100); i=atoi
(input);
```

```
How many numbers do you want to
type in? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436,
1067, 8, 32,
```

```
l= new long[i];
if (l == NULL) exit (1);
for (n=0; n
cout << "Enter number: ";
cin.getline (input,100);
l[n]=atol (input);
}
cout << "You have entered: ";
for (n=0; n
cout << l[n] << ", ";
delete[] l;
return 0;
}
```

这个简单的例子可以记下用户想输入的任意多个数字，它的实现归功于我们动态地向系统申请用户要输入的数字所需的空间。

NULL 是 C++库中定义的一个常量，专门设计用来指代空指针的。如果这个常量没有被预先定义，你可以自己定以它为 0:

```
#define NULL 0
```

在检查指针的时候，0 和 NULL 并没有区别。但用 NULL 来表示空指针更为常用，并且更易懂。原因是指针很少被用来比较大小或被直接赋予一个除 0 以外的数字常量，使用 NULL，这一赋值行为就被符号化了。

ANSI-C 中的动态内存管理 Dynamic memory in ANSI-C

操作符 new 和 delete 仅在 C++中有效，而在 C 语言中没有。在 C 语言中，为了动态分配内存，我们必须求助于函数库 stdlib.h。因为该函数库在 C++中仍然有效，并且在一些现存的程序仍然使用，所以我们下面将学习一些关于这个函数库中的函数用法。

函数 malloc

这是给指针动态分配内存的通用函数。它的原型是：

```
void * malloc (size_t nbytes);
```

其中 nbytes 是我们想要给指针分配的内存字节数。这个函数返回一个 void*类型的指针，因此我们需要用类型转换(type cast)来把它转换成目标指针所需要的数据类型，例如：

```
char * ronny;
ronny = (char *) malloc (10);
```

这个例子将一个指向 10 个字节可用空间的指针赋给 ronny。当我们想给一组除 char 以外的类型（不是 1 字节长度的）的数值分配内存的时候，我们需要用元素数乘以每个元素的长度来确定所需内存的大小。幸运的是我们有操作符 sizeof，它可以返回一个具体数据类型的长度。

```
int * bobby;
bobby = (int *) malloc (5 * sizeof(int));
```

这一小段代码将一个指向可存储 5 个 int 型整数的内存块的指针赋给 bobby，它的实际长度可能是 2，4 或更多字节数，取决于程序是在什么操作系统下被编译的。

函数 calloc

calloc 与 malloc 在操作上非常相似，他们主要的区别是在原型上：

```
void * calloc (size_t nelements, size_t size);
```

因为它接收 2 个参数而不是 1 个。这两个参数相乘被用来计算所需内存块的总长度。通常第一个参数(nelements)是元素的个数，第二个参数 (size) 被用来表示每个元素的长度。例如，我们可以像下面这样用 calloc 定义 bobby：

```
int * bobby;
bobby = (int *) calloc (5, sizeof(int));
```

malloc 和 calloc 的另一点不同在于 calloc 会将所有的元素初始化为 0。

函数 realloc

它被用来改变已经被分配给一个指针的内存的长度。

```
void * realloc (void * pointer, size_t size);
```

参数 pointer 用来传递一个已经被分配内存的指针或一个空指针，而参数 size 用来指明新的内存长度。这个函数给指针分配 size 字节的内存。这个函数可能需要改变内存块的地址以便能够分配足够的内存来满足新的长度要求。在这种情况下，指针当前所指的内存中的数据内容将会被拷贝到新的地址中，以保证现存数据不会丢失。函数返回新的指针地址。如果新的内存尺寸不能够被满足，函数将会返回一个空指针，但原来参数中的指针 pointer 及其内容保持不变。

函数 free

这个函数用来释放被前面 malloc, calloc 或 realloc 所分配的内存块。

```
void free (void * pointer);
```

注意：这个函数只能被用来释放由函数 malloc, calloc 和 realloc 所分配的空间。

你可以参考 C++ reference for cstdlib 获得更多关于这些函数的信息。

3.5 数据结构 （Data Structures）

一个数据结构是组合到同一定义下的一组不同类型的数据，各个数据类型的长度可能不同。它的形式是：

```
struct model_name {
type1 element1;
type2 element2;
type3 element3;
.
.
} object_name;
```

这里 model_name 是一个这个结构类型的模块名称。object_name 为可选参数，是一个或多个具体结构对象的标识。在花括号{ }内是组成这一结构的各个元素的类型和子标识。

如果结构的定义包括参数 model_name (可选)，该参数即成为一个与该结构等价的有效的类型名称。例如：

```
struct products {
char name [30];
float price;
```

```
};
products apple;
products orange, melon;
```

我们首先定义了结构模块 products，它包含两个域：name 和 price，每一个域是不同的数据类型。然后我们用这个结构类型的名称 (products) 来声明了 3 个该类型的对象：apple, orange 和 melon。

一旦被定义，products 就成为一个新的有效数据类型名称，可以像其他基本数据类型，如 int, char 或 short 一样，被用来声明该数据类型的对象(object)变量。

在结构定义的结尾可以加可选项 object_name ，它的作用是直接声明该结构类型的对象。例如，我们也可以这样声明结构对象 apple, orange 和 melon：

```
struct products {
char name [30];
float price;
}apple, orange, melon;
```

并且，像上面的例子中如果我们在定义结构的同时声明结构的对象，参数 model_name (这个例子中的 products)将变为可选项。但是如果没有 model_name，我们将不能在后面的程序中用它来声明更多此类结构的对象。

清楚地区分结构模型 model 和它的对象的概念是很重要的。参考我们对变量所使用的术语，模型 model 是一个类型 type，而对象 object 是变量 variable。我们可以从同一个模型 model (type)实例化出很多对象 objects (variables)。

在我们声明了确定结构模型的 3 个对象(apple, orange 和 melon)之后，我们就可以对它们的各个域(field)进行操作，这通过在对象名和域名之间插入符号点(.)来实现。例如，我们可以像使用一般的标准变量一样对下面的元素进行操作：

```
apple.name
apple.price
orange.name
orange.price
melon.name
melon.price
```

它们每一个都有对应的数据类型：apple.name, orange.name 和 melon.name 是字符数组类型 char[30]，而 apple.price, orange.price 和 melon.price 是浮点型 float。

下面我们看另一个关于电影的例子：

```
// example about structures
#include ‹iostream.h›
#include ‹string.h›
#include ‹stdlib.h›

struct movies_t {
char title [50];
int year;
}mine, yours;

void printmovie (movies_t movie);

int main () {
char buffer [50];
strcpy (mine.title, "2001 A Space
Odyssey");
mine.year = 1968;
cout << "Enter title: ";
cin.getline (yours.title,50);
cout << "Enter year: ";
cin.getline (buffer,50);
yours.year = atoi (buffer);
cout << "My favourite movie is:\n ";
printmovie (mine);
cout << "And yours:\n";
printmovie (yours);
return 0;
}

void printmovie (movies_t movie) {
cout << movie.title;
cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Alien
Enter year: 1979
My favourite movie is:
2001 A Space Odyssey (1968)
And yours:
Alien (1979)
```

这个例子中我们可以看到如何像使用普通变量一样使用一个结构的元素及其本身。例如，yours.year 是一个整型数据 int，而 mine.title 是一个长度为 50 的字符数组。

注意这里 mine 和 yours 也是变量，他们是 movies_t 类型的变量，被传递给函数 printmovie()。因此，结构的重要优点之一就是我们既可以单独引用它的元素，也可以引用整个结构数据块。

结构经常被用来建立数据库，特别是当我们考虑结构数组的时候。

```
// array of structures
#include ‹iostream.h›
#include ‹stdlib.h›

#define N_MOVIES 5

struct movies_t {
char title [50];
int year;
} films [N_MOVIES];

void printmovie (movies_t movie);

int main () {
char buffer [50];
int n;
for (n=0; n
cout << "Enter title: ";
cin.getline (films[n].title,50);
cout << "Enter year: ";
cin.getline (buffer,50);
films[n].year = atoi (buffer);
}

cout << "\nYou have entered these
movies:\n";
for (n=0; n
printmovie (films[n]);
return 0;
}

void printmovie (movies_t movie) {
cout << movie.title;
cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Alien
Enter year: 1979
Enter title: Blade Runner
Enter year: 1982
Enter title: Matrix
Enter year: 1999
Enter title: Rear Window
Enter year: 1954
Enter title: Taxi Driver
Enter year: 1975

You have entered these movies:

Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)
```

结构指针(Pointers to structures)

就像其它数据类型一样，结构也可以有指针。其规则同其它基本数据类型一样：指针必须被声明为一个指向结构的指针：

```
struct movies_t {
char title [50];
int year;
};
movies_t amovie;
movies_t * pmovie;
```

这里 amovie 是一个结构类型 movies_t 的对象，而 pmovie 是一个指向结构类型 movies_t 的对象的指针。所以，同基本数据类型一样，以下表达式正确的：

```
pmovie = &amovie;
```

下面让我们看另一个例子，它将引入一种新的操作符：

```
// pointers to structures
#include ‹iostream.h›
#include ‹stdlib.h›

struct movies_t {
char title [50];
int year;
};

int main () {
char buffer[50];

movies_t amovie;
movies_t * pmovie;
pmovie = & amovie;

cout << "Enter title: ";
cin.getline (pmovie->title,50);
cout << "Enter year: ";

cin.getline (buffer,50);
pmovie->year = atoi (buffer);

cout << "\nYou have entered:\n";
cout << pmovie->title;
cout << " (" << pmovie->year << ")\n";


return 0;
}
```

```
Enter title: Matrix
Enter year: 1999

You have entered:
Matrix (1999)
```

上面的代码中引入了一个重要的操作符：->。这是一个引用操作符，常与结构或类的指针一起使用，以便引用其中的成员元素，这样就避免使用很多括号。例如，我们用：

```
pmovie->title
```

来代替：

```
(*pmovie).title
```

以上两种表达式 pmovie->title 和 (*pmovie).title 都是合法的，都表示取指针 pmovie 所指向的结构其元素 title 的值。我们要清楚将它和以下表达区分开：

```
*pmovie.title
```

它相当于

```
*(pmovie.title)
```

表示取结构 pmovie 的元素 title 作为指针所指向的值，这个表达式在本例中没有意义，因为 title 本身不是指针类型。

下表中总结了指针和结构组成的各种可能的组合：

表达式	描述	等价于
pmovie.title	结构 pmovie 的元素 title	
pmovie->title	指针 pmovie 所指向的结构其元素 title 的值	(*pmovie).title
*pmovie.title	结构 pmovie 的元素 title 作为指针所指向的值	*(pmovie.title)

结构嵌套(Nesting structures)

结构可以嵌套使用，即一个结构的元素本身又可以是另一个结构类型。例如：

```
struct movies_t {
char title [50];
int year;
}

struct friends_t {
char name [50];
char email [50];
```

```
movies_t favourite_movie;
} charlie, maria;

friends_t * pfriends = &charlie;
```

因此，在有以上声明之后，我们可以使用下面的表达式：

```
charlie.name
maria.favourite_movie.title
charlie.favourite_movie.year
pfriends->favourite_movie.year
```

(以上最后两个表达式等价)

本节中所讨论的结构的概念与 C 语言中结构概念是一样的。然而，在 C++中，结构的概念已经被扩展到与类(class)相同的程度，只是它所有的元素都是公开的(public)。在后面的章节 4.1“类”中，我们将进一步深入讨论这个问题。

3.6 自定义数据类型 （User defined data types）

前面我们已经看到过一种用户（程序员）定义的数据类型：结构。除此之外，还有一些其它类型的用户自定义数据类型：

定义自己的数据类型 (typedef)

C++ 允许我们在现有数据类型的基础上定义我们自己的数据类型。我们将用关键字 typedef 来实现这种定义，它的形式是：

```
typedef existing_type new_type_name;
```

这里 existing_type 是 C++ 基本数据类型或其它已经被定义了的数据类型，new_type_name 是我们将要定义的新数据类型的名称。例如：

```
typedef char C;
typedef unsigned int WORD;
typedef char * string_t;
typedef char field [50];
```

在上面的例子中，我们定义了四种新的数据类型： C, WORD, string_t 和 field ，它们分别代替 char, unsigned int, char* 和 char[50] 。这样，我们就可以安全的使用以下代码：

```
C achar, anotherchar, *ptchar1;
WORD myword;
string_t ptchar2;
field name;
```

如果在一个程序中我们反复使用一种数据类型，而在以后的版本中我们有可能改变该数据类型的情况下，typedef 就很有用了。或者如果一种数据类型的名称太长，你想用一个比较短的名字来代替，也可以是用 typedef。

联合(Union)

联合(Union) 使得同一段内存可以被按照不同的数据类型来访问，数据实际是存储在同一个位置的。它的声明和使用看起来与结构(structure)十分相似，但实际功能是完全不同的：

```
union model_name {
       type1 element1;
       type2 element2;
       type3 element3;
       .
       .
} object_name;
```

union 中的所有被声明的元素占据同一段内存空间，其大小取声明中最长的元素的大小。例如：

```
union mytypes_t {
       char c;
       int i;
       float f;
} mytypes;
```

定义了 3 个元素：

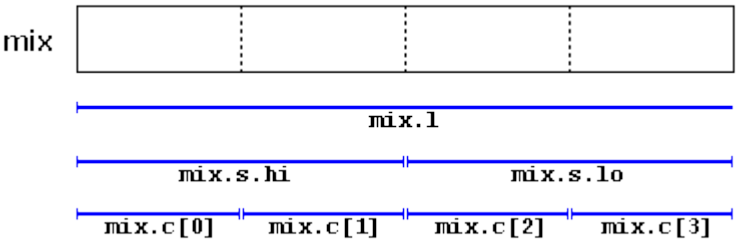
```
mytypes.c
mytypes.i
mytypes.f
```

每一个是一种不同的数据类型。既然它们都指向同一段内存空间，改变其中一个元素的值，将会影响所有其他元素的值。

union 的用途之一是将一种较长的基本类型与由其它比较小的数据类型组成的结构(structure)或数组(array)联合使用，例如：

```
union mix_t{
        long l;
        struct {
                short hi;
                short lo;
        } s;
        char c[4];
} mix;
```

以上例子中定义了 3 个名称：mix.l, mix.s 和 mix.c，我们可以通过这 3 个名字来访问同一段 4 bytes 长的内存空间。至于使用哪一个名字来访问，取决于我们想使用什么数据类型，是 long, short 还是 char 。下图显示了在这个联合(union)中各个元素在内存中的的可能结构，以及我们如何通过不同的数据类型进行访问：



匿名联合(Anonymous union)

在 C++ 我们可以选择使联合(union)匿名。如果我们将一个 union 包括在一个结构(structure)的定义中，并且不赋予它 object 名称 (就是跟在花括号{}后面的名字)，这个 union 就是匿名的。这种情况下我们可以直接使用 union 中元素的名字来访问该元素，而不需要再在前面加 union 对象的名称。在下面的例子中，我们可以看到这两种表达方式在使用上的区别：

union	anonymous union

struct {
  char title[50];
 	
  char author[50];
  union {
  float dollars;
	struct {
	  char title[50];
	
	  char author[50];
	  union {
	  float dollars;

int yens; } price; } book;	int yens; }; } book;

以上两种定义的唯一区别在于左边的定义中我们给了 union 一个名字 price，而在右边的定义中我们没给。在使用时的区别是当我们想访问一个对象(object)的元素 dollars 和 yens 时，在前一种定义的情况下，需要使用：

```
book.price.dollars
book.price.yens
```

而在后面一种定义下，我们直接使用：

```
book.dollars
book.yens
```

再一次提醒，因为这是一个联合(union)，域 dollars 和 yens 占据的是同一块内存空间，所以它们不能被用来存储两个不同的值。也就是你可以使用一个 dollars 或 yens 的价格，但不能同时使用两者。

枚举 Enumerations (enum)

枚举(Enumerations)可以用来生成一些任意类型的数据，不只限于数字类型或字符类型，甚至常量 true 和 false。它的定义形式如下：

```
enum model_name {
        value1,
        value2,
        value3,
        .
        .
} object_name;
```

例如，我们可以定义一种新的变量类型叫做 color_t 来存储不同的颜色：

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

注意在这个定义里我们没有使用任何基本数据类型。换句话说，我们创造了一种的新的数据类型，而它并没有基于任何已存在的数据类型：类型 color_t，花括

号{}中包括了它的所有的可能取值。例如，在定义了 colors_t 列举类型后，我们可以使用以下表达式 ：

```
colors_t mycolor;
mycolor = blue;
if (mycolor == green) mycolor = red;
```

实际上，我们的枚举数据类型在编译时是被编译为整型数值的，而它的数值列表可以是任何指定的整型常量 。如果没有指定常量，枚举中第一个列出的可能值为 0 ，后面的每一个值为前面一个值加 1。因此，在我们前面定义的数据类型 colors_t 中，black 相当于 0， blue 相当于 1， green 相当于 2 ，后面依此类推。

如果我们在定义枚举数据类型的时候明确指定某些可能值（例如第一个）的等价整数值，后面的数值将会在此基础上增加，例如：

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```

在这个例子中，枚举类型 months_t 的变量 y2k 可以是 12 种可能取值中的任何一个，从 january 到 december ，它们相当于数值 1 到 12，而不是 0 到 11 ，因为我们已经指定 january 等于 1。

4.1 类（Classes）

类(class)是一种将数据和函数组织在同一个结构里的逻辑方法。定义类的关键字为 class ，其功能与 C 语言中的 struct 类似，不同之处是 class 可以包含函数，而不像 struct 只能包含数据元素。

类定义的形式是：

```
    class class_name {
        permission_label_1:
            member1;
        permission_label_2:
            member2;
        ...
    } object_name;
```

其中 class_name 是类的名称（用户自定义的类型），而可选项 object_name 是一个或几个对象(object)标识。Class 的声明体中包含成员 members，成员可以是数据或函数定义，同时也可以包括允许范围标志 permission labels，范围

标志可以是以下三个关键字中任意一个：private:, public: 或 protected:。它们分别代表以下含义：

- private ： class 的 private 成员，只有同一个 class 的其他成员或该 class 的“friend” class 可以访问这些成员。
- protected ： class 的 protected 成员，只有同一个 class 的其他成员，或该 class 的“friend” class,或该 class 的子类(derived classes) 可以访问这些成员。
- public ： class 的 public 成员，任何可以看到这个 class 的地方都可以访问这些成员。

如果我们在定义一个 class 成员的时候没有声明其允许范围,这些成员将被默认为 private 范围。

例如：

```
class CRectangle {

        int x, y;
    public:
        void set_values (int,int);
        int area (void);
} rect;
```

上面例子定义了一个 class CRectangle 和该 class 类型的对象变量 rect 。这个 class 有 4 个成员：两个整型变量 (x 和 y) ，在 private 部分 (因为 private 是默认的允许范围)；以及两个函数， 在 public 部分：set_values() 和 area()，这里只包含了函数的原型(prototype)。

注意 class 名称与对象(object)名称的不同：在上面的例子中，CRectangle 是 class 名称 (即用户定义的类型名称)，而 rect 是一个 CRectangle 类型的对象名称。它们的区别就像下面例子中类型名 int 和 变量名 a 的区别一样：

```
int a;
```

int 是 class 名称 (类型名) ，而 a 是对象名 object name (变量)。

在程序中,我们可以通过使用对象名后面加一点再加成员名称 (同使用 C structs 一样) ，来引用对象 rect 的任何 public 成员，就像它们只是一般的函数或变量。例如：

```
rect.set_value (3,4);
myarea = rect.area();
```

但我们不能够引用 x 或 y ，因为它们是该 class 的 private 成员，它们只能够在该 class 的其它成员中被引用。晕了吗？下面是关于 class CRectangle 的一个复杂的例子：

```
    // classes example                    area: 12
    #include <iostream.h>
    class CRectangle {
            int x, y;
        public:
            void set_values (int,int);
            int area (void) {return
(x*y);}
    };

    void CRectangle::set_values (int a,
int b) {
        x = a;
        y = b;
    }

    int main () {
        CRectangle rect;
        rect.set_values (3,4);
        cout << "area: " << rect.area();
    }
```

上面代码中新的东西是在定义函数 set_values().使用的范围操作符(双冒号::)。它是用来在一个 class 之外定义该 class 的成员。注意，我们在 CRectangle class 内部已经定义了函数 area() 的具体操作，因为这个函数非常简单。而对函数 set_values() ，在 class 内部只是定义了它的原型 prototype，而其实现是在 class 之外定义的。这种在 class 之外定义其成员的情况必须使用范围操作符::。

范围操作符 (::) 声明了被定义的成员所属的 class 名称，并赋予被定义成员适当的范围属性，这些范围属性与在 class 内部定义成员的属性是一样的。例如，在上面的例子中，我们在函数 set_values() 中引用了 private 变量 x 和 y，这些变量只有在 class 内部和它的成员中才是可见的。

在 class 内部直接定义完整的函数，和只定义函数的原型而把具体实现放在 class 外部的唯一区别在于，在第一种情况中，编译器(compiler) 会自动将函数作为 inline 考虑，而在第二种情况下，函数只是一般的 class 成员函数。

我们把 x 和 y 定义为 private 成员（记住，如果没有特殊声明，所有 class 的成员均默认为 private ），原因是我们已经定义了一个设置这些变量值的函数(set_values()) ，这样一来，在程序的其它地方就没有办法直接访问它们。也许在一个这样简单的例子中，你无法看到这样保护两个变量有什么意义，但在比较复杂的程序中，这是非常重要的，因为它使得变量不会被意外修改 （这里意外指的是从 object 的角度来讲的意外）。

使用 class 的一个更大的好处是我们可以用它来定义多个不同对象(object)。例如，接着上面 class CRectangle 的例子，除了对象 rect 之外，我们还可以定义对象 rectb ：

```
    // class example                        rect area: 12
    #include <iostream.h>                   rectb area: 30

    class CRectangle {
            int x, y;
        public:
            void set_values (int,int);
            int area (void) {return
(x*y);}
    };

    void CRectangle::set_values (int a,
int b) {
        x = a;
        y = b;
    }

    int main () {
        CRectangle rect, rectb;
        rect.set_values (3,4);
        rectb.set_values (5,6);
        cout << "rect area: " <<
rect.area() << endl;
        cout << "rectb area: " <<
rectb.area() << endl;
    }
```

注意: 调用函数 rect.area() 与调用 rectb.area()所得到的结果是不一样的。这是因为每一个 class CRectangle 的对象都拥有它自己的变量 x 和 y，以及它自己的函数 set_value() 和 area()。

这是基于对象(object) 和 面向对象编程 (object-oriented programming)的概念的。这个概念中，数据和函数是对象(object)的属性(properties)，而不是像以前在结构化编程 (structured programming) 中所认为的对象(object)是函数参数。在本节及后面的小节中，我们将讨论面向对象编程的好处。

在这个具体的例子中，我们讨论的 class (object 的类型)是 CRectangle，有两个实例(instance)，或称对象(object)：rect 和 rectb，每一个有它自己的成员变量和成员函数。

构造函数和析构函数 (Constructors and destructors)

对象(object)在生成过程中通常需要初始化变量或分配动态内存，以便我们能够操作，或防止在执行过程中返回意外结果。例如，在前面的例子中，如果我们在调用函数 set_values() 之前就调用了函数 area()，将会产生什么样的结果呢？可能会是一个不确定的值，因为成员 x 和 y 还没有被赋于任何值。

为了避免这种情况发生，一个 class 可以包含一个特殊的函数：构造函数 constructor，它可以通过声明一个与 class 同名的函数来定义。当且仅当要生成一个 class 的新的实例 (instance)的时候，也就是当且仅当声明一个新的对象，或给该 class 的一个对象分配内存的时候，这个构造函数将自动被调用。下面，我们将实现包含一个构造函数的 CRectangle ：

```
// class example                    rect area: 12
#include <iostream.h>               rectb area: 30

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area (void) {return
(width*height);}
};

CRectangle::CRectangle (int a, int
b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
```

```
        cout << "rect area: " <<
rect.area() << endl;
        cout << "rectb area: " <<
rectb.area() << endl;
    }
```

正如你所看到的，这个例子的输出结果与前面一个没有区别。在这个例子中，我们只是把函数 set_values 换成了 class 的构造函数 constructor。注意这里参数是如何在 class 实例 (instance)生成的时候传递给构造函数的：

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

同时你可以看到，构造函数的原型和实现中都没有返回值(return value)，也没有 void 类型声明。构造函数必须这样写。一个构造函数永远没有返回值，也不用声明 void，就像我们在前面的例子中看到的。

析构函数 Destructor 完成相反的功能。它在 objects 被从内存中释放的时候被自动调用。释放可能是因为它存在的范围已经结束了（例如，如果 object 被定义为一个函数内的本地（local）对象变量，而该函数结束了）；或者是因为它是一个动态分配的对象，而被使用操作符 delete 释放了。

析构函数必须与 class 同名，加水波号 tilde (~) 前缀，必须无返回值。

析构函数特别适用于当一个对象被动态分别内存空间，而在对象被销毁的时我们希望释放它所占用的空间的时候。例如：

```
    // example on constructors and
destructors
    #include <iostream.h>

    class CRectangle {
        int *width, *height;
      public:
        CRectangle (int,int);
        ~CRectangle ();
        int area (void) {return (*width
* *height);}
    };

    CRectangle::CRectangle (int a, int
b) {
```

```
rect area: 12
rectb area: 30
```

```
        width = new int;
        height = new int;
        *width = a;
        *height = b;
    }

    CRectangle::~CRectangle () {
        delete width;
        delete height;
    }

    int main () {
        CRectangle rect (3,4), rectb
(5,6);
        cout << "rect area: " <<
rect.area() << endl;
        cout << "rectb area: " <<
rectb.area() << endl;
        return 0;
    }
```

构造函数重载(Overloading Constructors)

像其它函数一样，一个构造函数也可以被多次重载(overload)为同样名字的函数，但有不同的参数类型和个数。记住，编译器会调用与在调用时刻要求的参数类型和个数一样的那个函数(Section 2.3, Functions-II)。在这里则是调用与类对象被声明时一样的那个构造函数。

实际上，当我们定义一个 class 而没有明确定义构造函数的时候，编译器会自动假设两个重载的构造函数 (默认构造函数"default constructor" 和复制构造函数"copy constructor")。例如，对以下 class：

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
};
```

没有定义构造函数，编译器自动假设它有以下 constructor 成员函数：

- Empty constructor

它是一个没有任何参数的构造函数，被定义为 nop (没有语句)。它什么都不做。

```
CExample::CExample () { };
```

- Copy constructor

  它是一个只有一个参数的构造函数，该参数是这个 class 的一个对象，这个函数的功能是将被传入的对象（object）的所有非静态（non-static）成员变量的值都复制给自身这个 object。

```
CExample::CExample (const CExample& rv) {
    a=rv.a;  b=rv.b;  c=rv.c;
}
```

必须注意：这两个默认构造函数（empty construction 和 copy constructor ）只有在没有其它构造函数被明确定义的情况下才存在。如果任何其它有任意参数的构造函数被定义了，这两个构造函数就都不存在了。在这种情况下，如果你想要有 empty construction 和 copy constructor ，就必需要自己定义它们。

当然，如果你也可以重载 class 的构造函数，定义有不同的参数或完全没有参数的构造函数，见如下例子：

```
// overloading class constructors
#include <iostream.h>

Class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return
(width*height);}
};

CRectangle::CRectangle () {
    width = 5;
    height = 5;
}

CRectangle::CRectangle (int a, int
b) {
    width = a;
```

```
rect area: 12
rectb area: 25
```

```
        height = b;
    }

    int main () {
        CRectangle rect (3,4);
        CRectangle rectb;
        cout << "rect area: " <<
rect.area() << endl;
        cout << "rectb area: " <<
rectb.area() << endl;
    }
```

在上面的例子中，rectb 被声明的时候没有参数，所以它被使用没有参数的构造函数进行初始化，也就是 width 和 height 都被赋值为 5。

注意在我们声明一个新的 object 的时候，如果不想传入参数，则不需要写括号()：

```
CRectangle rectb; // right
CRectangle rectb(); // wrong!
```

类的指针(Pointers to classes)

类也是可以有指针的，要定义类的指针，我们只需要认识到，类一旦被定义就成为一种有效的数据类型，因此只需要用类的名字作为指针的名字就可以了。例如：

```
CRectangle * prect;
```

是一个指向 class CRectangle 类型的对象的指针。

就像数据机构中的情况一样，要想直接引用一个由指针指向的对象(object)中的成员，需要使用操作符 ->。这里是一个例子，显示了几种可能出现的情况：

```
    // pointer to classes example
    #include <iostream.h>

    class CRectangle {
        int width, height;
      public:
        void set_values (int, int);
        int area (void) {return (width
* height);}
```

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

```
    };

    void CRectangle::set_values (int a,
int b) {
        width = a;
        height = b;
    }

    int main () {
        CRectangle a, *b, *c;
        CRectangle * d = new
CRectangle[2];
        b= new CRectangle;
        c= &a;
        a.set_values (1,2);
        b->set_values (3,4);
        d->set_values (5,6);
        d[1].set_values (7,8);
        cout << "a area: " << a.area()
<< endl;
        cout <<"*b area: " <<b->area()
<< endl;
        cout <<"*c area: " <<c->area()
<< endl;
        cout << "d[0] area: " <<
d[0].area() << endl;
        cout << "d[1] area: " <<
d[1].area() << endl;
        return 0;
    }
```

以下是怎样读前面例子中出现的一些指针和类操作符 (*, &, ., ->, [ ]):

- *x 读作: pointed by x （由 x 指向的）
- &x 读作: address of x（x 的地址）
- x.y 读作: member y of object x （对象 x 的成员 y）
- (*x).y 读作: member y of object pointed by x（由 x 指向的对象的成员 y）
- x->y 读作: member y of object pointed by x (同上一个等价)
- x[0] 读作: first object pointed by x（由 x 指向的第一个对象）
- x[1] 读作: second object pointed by x（由 x 指向的第二个对象）
- x[n] 读作: (n+1)th object pointed by x（由 x 指向的第 n+1 个对象）

在继续向下阅读之前，一定要确定你明白所有这些的逻辑含义。如果你还有疑问，再读一遍这一笑节，或者同时参考 小节 "3.3, 指针(Pointers)" 和 "3.5, 数据结构(Structures)".

由关键字 struct 定义的类

C++ 语言以将C 语言中的关键字struct 扩展到与C++语言中的class 关键字具有基本相同的功能，除了它所有的成员都默认为 public 而不是 private。

不管怎样，因为 class 和 struct 在C++中几乎具有同样的功能，struct 通常被用在只有数据的结构中，而 class 则被用在有过程和成员函数的情况下。

4.2 操作符重载（Overloading operators）

C++ 实现了在类(class)之间使用语言标准操作符，而不只是在基本数据类型之间使用。例如：

```
int a, b, c;
a = b + c;
```

是有效操作，因为加号两边的变量都是基本数据类型。然而，我们是否可以进行下面的操作就不是那么显而易见了（它实际上是正确的）：

```
struct { char product [50]; float price; } a, b, c;
a = b + c;
```

将一个类class (或结构 struct)的对象赋给另一个同种类型的对象是允许的(通过使用默认的复制构造函数 copy constructor)。但相加操作就有可能产生错误，理论上讲它在非基本数据类型之间是无效的。

但归功于 C++ 的操作符重载（overload）能力，我们可以完成这个操作。像以上例子中这样的组合类型的对象在 C++中可以接受如果没有操作符重载则不能被接受的操作，我们甚至可以修改这些操作符的效果。以下是所有可以被重载的操作符的列表：

```
        +    -    *    /    =    <    >    +=   -=   *=   /=   <<
>>
        <<=  >>=  ==   !=   <=   >=   ++   --   %    &    ^    !    |
        ~    &=   ^=   |=   &&   ||   %=   []   ()   new  delete
```

要想重载一个操作符，我们只需要编写一个成员函数，名为 operator ，后面跟我们要重载的操作符，遵循以下原型定义：

```
type operator sign (parameters);
```

这里是一个操作符 +的例子。我们要计算二维向量(bidimensional vector) a(3,1) 与 b(1,2)的和。两个二维向量相加的操作很简单，就是将两个 x 轴的值相加获得结果的 x 轴值，将两个 y 轴值相加获得结果的 y 值。在这个例子里，结果是 (3+1,1+2) = (4,3)。

```
    // vectors: overloading operators    4,3
example
    #include <iostream.h>

    class CVector {
      public:
        int x,y;
        CVector () {};
        CVector (int,int);
        CVector operator + (CVector);
    };

    CVector::CVector (int a, int b) {
        x = a;
        y = b;
    }

    CVector CVector::operator+
(CVector param) {
        CVector temp;
        temp.x = x + param.x;
        temp.y = y + param.y;
        return (temp);
    }

    int main () {
        CVector a (3,1);
        CVector b (1,2);
        CVector c;
        c = a + b;
        cout << c.x << "," << c.y;
        return 0;
    }
```

如果你迷惑为什么看到这么多遍的 CVector，那是因为其中有些是指 class 名称 CVector ，而另一些是以它命名的函数名称，不要把它们搞混了：

```
   CVector (int, int);             // 函数名称 CVector (constructor)
   CVector operator+ (CVector);    // 函数 operator+ 返回 CVector 类型的值
```

Class CVector 的函数 operator+ 是对数学操作符+进行重载的函数。这个函数可以用以下两种方法进行调用：

```
c = a + b;
c = a.operator+ (b);
```

注意：我们在这个例子中包括了一个空构造函数 (无参数)，而且我们将它定义为无任何操作：

```
CVector ( ) { };
```

这是很必要的，因为例子中已经有另一个构造函数，

```
CVector (int, int);
```

因此，如果我们不像上面这样明确定义一个的话，CVector 的两个默认构造函数都不存在。

这样的话，main( )中包含的语句

```
CVector c;
```

将为不合法的。

尽管如此，我已经警告过一个空语句块 (no-op block)并不是一种值得推荐的构造函数的实现方式，因为它不能实现一个构造函数至少应该完成的基本功能，也就是初始化 class 中的所有变量。在我们的例子中，这个构造函数没有完成对变量 x 和 y 的定义。因此一个更值得推荐的构造函数定义应该像下面这样：

```
CVector ( ) { x=0; y=0; };
```

就像一个 class 默认包含一个空构造函数和一个复制构造函数一样，它同时包含一个对赋值操作符 assignation operator (=)的默认定义，该操作符用于两个同类对象之间。这个操作符将其参数对象(符号右边的对象) 的所有非静态

(non-static) 数据成员复制给其左边的对象。当然，你也可以将它重新定义为你想要的任何功能，例如，只拷贝某些特定 class 成员。

重载一个操作符并不要求保持其常规的数学含义，虽然这是推荐的。例如，虽然我们可以将操作符+定义为取两个对象的差值，或用==操作符将一个对象赋为 0，但这样做是没有什么逻辑意义的。

虽然函数 operator+ 的原型定义看起来很明显，因为它取操作符右边的对象为其左边对象的函数 operator+的参数，其它的操作符就不一定这么明显了。以下列表总结了不同的操作符函数是怎样定义声明的 (用操作符替换每个@):

Expression	Operator (@)	Function member	Global function
@a	+ - * & ! ~ ++ --	A::operator@( )	operator@(A)
a@	++ --	A::operator@(int)	operator@(A, int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A, B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= [ ]	A::operator@(B)	-
a(b, c...)	( )	A::operator()(B, C...)	-
a->b	->	A::operator->()	-

* 这里 a 是 class A 的一个对象，b 是 B 的一个对象，c 是 class C 的一个对象。

从上表可以看出有两种方法重载一些 class 操作符：作为成员函数（member function）或作为全域函数(global function)。它们的用法没有区别，但是我要提醒你，如果不是 class 的成员函数，则不能访问该 class 的 private 或 protected 成员，除非这个全域函数是该 class 的 friend (friend 的含义将在后面的章节解释)。

关键字 this

关键字 this 通常被用在一个 class 内部，指正在被执行的该 class 的对象(object)在内存中的地址。它是一个指针，其值永远是自身 object 的地址。

它可以被用来检查传入一个对象的成员函数的参数是否是该对象本身。例如：

```
    // this                                yes, &a is b
    #include <iostream.h>

    class CDummy {
      public:
        int isitme (CDummy& param);
    };

    int CDummy::isitme (CDummy& param)
{
        if (&param == this) return 1;
        else return 0;
    }

    int main () {
        CDummy a;
        CDummy* b = &a;
        if ( b->isitme(a) )
            cout << "yes, &a is b";
        return 0;
    }
```

它还经常被用在成员函数 operator= 中，用来返回对象的指针(避免使用临时对象)。以下用前面看到的向量(vector)的例子来看一下函数 operator= 是怎样实现的：

```
   CVector& CVector::operator= (const CVector& param) {
       x=param.x;
       y=param.y;
       return *this;
   }
```

实际上，如果我们没有定义成员函数 operator=，编译器自动为该 class 生成的默认代码有可能就是这个样子的。

静态成员(Static members)

一个 class 可以包含静态成员(static members)，可以是数据，也可以是函数。

一个 class 的静态数据成员也被称作类变量"class variables"，因为它们的内容不依赖于某个对象，对同一个 class 的所有 object 具有相同的值。

例如，它可以被用作计算一个 class 声明的 objects 的个数，见以下代码程序：

```
// static members in classes     7
#include <iostream.h>            6

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
    CDummy a;
    CDummy b[5];
    CDummy * c = new CDummy;
    cout << a.n << endl;
    delete c;
    cout << CDummy::n << endl;
    return 0;
}
```

实际上，静态成员与全域变量(global variable)具有相同的属性，但它享有类(class)的范围。因此，根据 ANSI-C++ 标准，为了避免它们被多次重复声明，在 class 的声明中只能够包括 static member 的原型(声明)，而不能够包括其定义(初始化操作)。为了初始化一个静态数据成员，我们必须在 class 之外(在全域范围内)，包括一个正式的定义，就像上面例子中做法一样。

因为它对同一个 class 的所有 object 是同一个值，所以它可以被作为该 class 的任何 object 的成员所引用，或者直接被作为 class 的成员引用(当然这只适用于 static 成员)：

```
cout << a.n;
cout << CDummy::n;
```

以上两个调用都指同一个变量：class CDummy 里的 static 变量 n 。

在提醒一次，它其实是一个全域变量。唯一的不同是它的名字跟在 class 的后面。

就像我们会在 class 中包含 static 数据一样，我们也可以使它包含 static 函数。它们表示相同的含义：static 函数是全域函数(global functions)，但是

像一个指定 class 的对象成员一样被调用。它们只能够引用 static 数据，永远不能引用 class 的非静态(nonstatic)成员。它们也不能够使用关键字 this，因为 this 实际引用了一个对象指针，但这些 static 函数却不是任何 object 的成员，而是 clas

4.3 类之间的关系(Relationships between classes)

Friend 函数 (friend 关键字)

在前面的章节中我们已经看到了对 class 的不同成员存在 3 个层次的内部保护：public， protected 和 private。在成员为 protected 和 private 的情况下，它们不能够被从所在的 class 以外的部分引用。然而，这个规则可以通过在一个 class 中使用关键字 friend 来绕过，这样我们可以允许一个外部函数获得访问 class 的 protected 和 private 成员的能力。

为了实现允许一个外部函数访问 class 的 private 和 protected 成员，我们必须在 class 内部用关键字 friend 来声明该外部函数的原型，以指定允许该函数共享 class 的成员。在下面的例子中我们声明了一个 friend 函数 duplicate：

```
    // friend functions                    24
    #include <iostream.h>

    class CRectangle {
        int width, height;
      public:
        void set_values (int, int);
        int area (void) {return (width
* height);}
        friend CRectangle duplicate
(CRectangle);
    };

    void CRectangle::set_values (int a,
int b) {
        width = a;
        height = b;
    }

    CRectangle duplicate (CRectangle
rectparam) {
        CRectangle rectres;
        rectres.width =
```

```
rectparam.width*2;
        rectres.height =
rectparam.height*2;
        return (rectres);
    }

    int main () {
        CRectangle rect, rectb;
        rect.set_values (2,3);
        rectb = duplicate (rect);
        cout << rectb.area();
    }
```

函数 duplicate 是 CRectangle 的 friend，因此在该函数之内，我们可以访问 CRectangle 类型的各个 object 的成员 width 和 height。注意，在 duplicate() 的声明中，及其在后面 main()里被调用的时候，我们并没有把 duplicate 当作 class CRectangle 的成员，它不是。

friend 函数可以被用来实现两个不同 class 之间的操作。广义来说，使用 friend 函数是面向对象编程之外的方法，因此，如果可能，应尽量使用 class 的成员函数来完成这些操作。比如在以上的例子中，将函数 duplicate() 集成在 class CRectangle 可以使程序更短。

Friend classes (friend)

就像我们可以定义一个 friend 函数，我们也可以定义一个 class 是另一个的 friend，以便允许第二个 class 访问第一个 class 的 protected 和 private 成员。

```
    // friend class                          16
    #include <iostream.h>

    class CSquare;

    class CRectangle {
        int width, height;
      public:
        int area (void) {return (width
* height);}
        void convert (CSquare a);
    };
```

```
Class CSquare {
  private:
    int side;
  public:
    void set_side (int a){side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare
a) {
    width = a.side;
    height = a.side;
}

int main () {
    CSquare sqr;
    CRectangle rect;
    sqr.set_side(4);
    rect.convert(sqr);
    cout << rect.area();
    return 0;
}
```

在这个例子中,我们声明了 CRectangle 是 CSquare 的 friend,因此 CRectangle 可以访问 CSquare 的 protected 和 private 成员，更具体地说，可以访问 CSquare::side，它定义了正方形的边长。

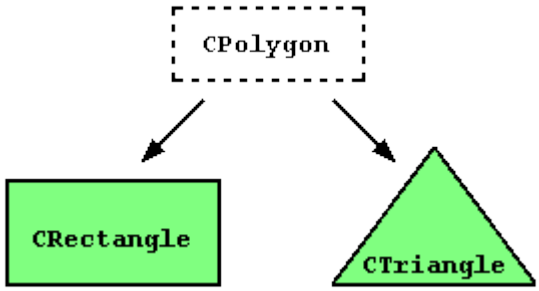
在上面程序的第一个语句里你可能也看到了一些新的东西，就是 class CSquare 空原型。这是必需的，因为在 CRectangle 的声明中我们引用了 CSquare (作为 convert()的参数)。CSquare 的定义在 CRectangle 的后面，因此如果我们没有在这个 class 之前包含一个 CSquare 的声明,它在 CRectangle 中就是不可见的。

这里要考虑到，如果没有特别指明，朋友关系（friendships）并不是相互的。在我们的CSquare 例子中,CRectangle 是一个 friend 类,但因为 CRectangle 并没有对 CSquare 作相应的声明，因此 CRectangle 可以访问 CSquare 的 protected 和 private 成员，但反过来并不行，除非我们将 CSquare 也定义为 CRectangle 的 friend。

类之间的继承(Inheritance between classes)

类的一个重要特征是继承，这使得我们可以基于一个类生成另一个类的对象，以便使后者拥有前者的某些成员，再加上它自己的一些成员。例如，假设我们要声明一系列类型的多边形，比如长方形 CRectangle 或三角形 CTriangle。它们有一些共同的特征，比如都可以只用两条边来描述：高（height）和底（base）。

这个特点可以用一个类 CPolygon 来表示，基于这个类我们可以引申出上面提到的两个类 CRectangle 和 CTriangle 。



类 CPolygon 包含所有多边形共有的成员。在我们的例子里就是： width 和 height。而 CRectangle 和 CTriangle 将为它的子类(derived classes)。

由其它类引申而来的子类继承基类的所有可视成员，意思是说，如果一个基类包含成员 A ，而我们将它引申为另一个包含成员 B 的类，则这个子类将同时包含 A 和 B。

要定义一个类的子类，我们必须在子类的声明中使用冒号(colon)操作符: ，如下所示：

```
class derived_class_name: public base_class_name;
```

这里 derived_class_name 为子类(derived class)名称，base_class_name 为基类(base class)名称。public 也可以根据需要换为 protected 或 private，描述了被继承的成员的访问权限，我们在以下例子后会很快看到：

```
    // derived classes                    20
    #include <iostream.h>                 10

    Class CPolygon {
      protected:
        int width, height;
      public:
        void set_values (int a, int b)
{ width=a; height=b;}
    };

    class CRectangle: public CPolygon {
```

```
  public:
    int area (void){ return (width
* height); }
};

class CTriangle: public CPolygon {
  public:
    int area (void){ return (width
* height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    rect.set_values (4,5);
    trgl.set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}
```

如上所示，类 CRectangle 和 CTriangle 的每一个对象都包含 CPolygon 的成员，即： width, height 和 set_values()。

标识符 protected 与 private 类似，它们的唯一区别在继承时才表现出来。当定义一个子类的时候，基类的 protected 成员可以被子类的其它成员所使用，然而 private 成员就不可以。因为我们希望 CPolygon 的成员 width 和 height 能够被子类 CRectangle 和 CTriangle 的成员所访问，而不只是被 CPolygon 自身的成员操作，我们使用了 protected 访问权限，而不是 private。

下表按照谁能访问总结了不同访问权限类型：

可以访问	public	protected	private
本 class 的成员	yes	yes	yes
子类的成员	yes	yes	no
非成员	yes	no	no

这里"非成员"指从class以外的任何地方引用,例如从main()中,从其它的class中或从全域(global)或本地(local)的任何函数中。

在我们的例子中，CRectangle 和 CTriangle 继承的成员与基类 CPolygon 拥有同样的访问限制：

```
CPolygon::width           // protected access
CRectangle::width         // protected access
CPolygon::set_values()    // public access
CRectangle::set_values()  // public access
```

这是因为我们在继承的时候使用的是 public，记得我们用的是：

```
class CRectangle: public CPolygon;
```

这里关键字 public 表示新的类(CRectangle)从基类(CPolygon)所继承的成员必须获得最低程度保护。这种被继承成员的访问限制的最低程度可以通过使用 protected 或 private 而不是 public 来改变。例如，daughter 是 mother 的一个子类，我们可以这样定义：

```
class daughter: protected mother;
```

这将使得 protected 成为 daughter 从 mother 处继承的成员的最低访问限制。也就是说，原来 mother 中的所有 public 成员到 daughter 中将会成为 protected 成员，这是它们能够被继承的最低访问限制。当然这并不是限制 daughter 不能有它自己的 public 成员。最低访问权限限制只是建立在从 mother 中 继承的成员上的。

最常用的继承限制除了 public 外就是 private ，它被用来将基类完全封装起来，因为在这种情况下，除了子类自身外，其它任何程序都不能访问那些从基类继承而来的成员。不过大多数情况下继承都是使用 public 的。

如果没有明确写出访问限制，所有由关键字 class 生成的类被默认为 private ，而所有由关键字 struct 生成的类被默认为 public。

什么是从基类中继承的? (What is inherited from the base class?)

理论上说，子类(drived class)继承了基类(base class)的所有成员，除了：

- 构造函数 Constructor 和析构函数 destructor
- operator=() 成员
- friends

虽然基类的构造函数和析构函数没有被继承，但是当一个子类的 object 被生成或销毁的时候，其基类的默认构造函数 (即，没有任何参数的构造函数)和析构函数总是被自动调用的。

如果基类没有默认构造函数，或你希望当子类生成新的 object 时，基类的某个重载的构造函数被调用，你需要在子类的每一个构造函数的定义中指定它：

```
derived_class_name (parameters) : base_class_name (parameters) {}
```

例如 （注意程序中黑体的部分）：

```
    // constructors and derivated
classes
    #include <iostream.h>

    class mother {
      public:
        mother ()
          { cout << "mother: no
parameters\n"; }
        mother (int a)
          { cout << "mother: int
parameter\n"; }
    };

    class daughter : public mother {
      public:
        daughter (int a)
          { cout << "daughter: int
parameter\n\n"; }
    };

    class son : public mother {
      public:
        son (int a) : mother (a)
          { cout << "son: int
parameter\n\n"; }
    };

    int main () {
        daughter cynthia (1);
        son daniel(1);
        return 0;
    }
```

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

观察当一个新的 daughter object 生成的时候 mother 的哪一个构造函数被调用了，而当新的 son object 生成的时候，又是哪一个被调用了。不同的构造函数被调用是因为 daughter 和 son 的构造函数的定义不同：

```
daughter (int a)          // 没有特别制定：调用默认 constructor
son (int a) : mother (a)  // 指定了 constructor: 调用被指定的构造函数
```

多重继承(Multiple inheritance)

在 C++ 中，一个 class 可以从多个 class 中继承属性或函数，只需要在子类的声明中用逗号将不同基类分开就可以了。例如，如果我们有一个特殊的 class COutput 可以实现向屏幕打印的功能，我们同时希望我们的类 CRectangle 和 CTriangle 在 CPolygon 之外还继承一些其它的成员，我们可以这样写：

```
class CRectangle: public CPolygon, public COutput {
class CTriangle: public CPolygon, public COutput {
```

以下是一个完整的例子：

```
// multiple inheritance                    20
#include <iostream.h>                      10

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class COutput {
  public:
    void output (int i);
};

void COutput::output (int i) {
    cout << i << endl;
}
```

```
    class CRectangle: public CPolygon,
public COutput {
      public:
        int area (void)
          { return (width * height); }
    };

    class CTriangle: public CPolygon,
public COutput {
      public:
        int area (void)
          { return (width * height /
2); }
    };

    int main () {
        CRectangle rect;
        CTriangle trgl;
        rect.set_values (4,5);
        trgl.set_values (4,5);
        rect.output (rect.area());
        trgl.output (trgl.area());
        return 0;
    }
```

s 的直接成员。

4.4 多态 (Polymorphism)

为了能更好的理解本节内容，你需要清楚的知道怎样使用指针 pointers 和类之间的继承 inheritance between classes。建议如果你觉得以下这些表达式比较生疏的的话， 请复习指定的章节：

```
int a::b(c) {};    // 类 Classes (Section 4.1)
a->b               // 指针和对象 pointers and
objects (Section 4.2)
class a: public b; // 类之间的关系 Relationships
between classes (Section 4.3)
```

基类的指针(Pointers to base class)

继承的好处之一是一个指向子类(derived class)的指针与一个指向基类(base class)的指针是 type-compatible 的。 本节就是重点介绍如何利用 C++的这一重要特性。例如，我们将结合 C++的这个功能，重写前面小节中关于长方形 rectangle 和三角形 triangle 的程序：

```
// pointers to base class                    20
#include <iostream.h>                        10
class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b) {
        width=a; height=b;
    }
};
class CRectangle: public CPolygon {
public:
    int area (void) {
        return (width * height);
    }
};
class CTriangle: public CPolygon {
public:
    int area (void) {
        return (width * height / 2);
    }
};
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}
```

在主函数 main 中定义了两个指向 class CPolygon 的对象的指针，即 *ppoly1 和 *ppoly2。 它们被赋值为 rect 和 trgl 的地址，因为 rect 和 trgl 是 CPolygon 的子类的对象，因此这种赋值是有效的。

使用*ppoly1 和 *ppoly2 取代 rect 和 trgl 的唯一限制是*ppoly1 和 *ppoly2 是 CPolygon* 类型的，因此我们只能够引用 CRectangle 和 CTriangle 从基类 CPolygon 中继承的成员。正是由于这个原因，我们不能够使用*ppoly1 和 *ppoly2 来调用成员函数 area()，而只能使用 rect 和 trgl 来调用这个函数。

要想使 CPolygon 的指针承认 area()为合法成员函数，必须在基类中声明它，而不能只在子类进行声明(见下一小节)。

虚拟成员(Virtual members)

如果想在基类中定义一个成员留待子类中进行细化，我们必须在它前面加关键字 virtual ，以便可以使用指针对指向相应的对象进行操作。

请看一下例子：

```
// virtual members                          20
#include <iostream.h>                       10
class CPolygon {                            0
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
{
            width=a;
            height=b;
        }
        virtual int area (void) { return
(0); }
};
class CRectangle: public CPolygon {
    public:
        int area (void) { return (width
* height); }
};
class CTriangle: public CPolygon {
    public:
        int area (void) {
```

```
            return (width * height / 2);
        }
};
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    CPolygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    cout << ppoly3->area() << endl;
    return 0;
}
```

现在这三个类(CPolygon, CRectangle 和 CTriangle) 都有同样的成员：width, height, set_values() 和 area()。

area() 被定义为 virtual 是因为它后来在子类中被细化了。你可以做一个试验，如果在代码种去掉这个关键字(virtual)，然后再执行这个程序，三个多边形的面积计算结果都将是 0 而不是 20,10,0。这是因为没有了关键字 virtual ，程序执行不再根据实际对象的不用而调用相应 area() 函数(即分别为 CRectangle::area(), CTriangle::area() 和 CPolygon::area())，取而代之，程序将全部调用 CPolygon::area()，因为这些调用是通过 CPolygon 类型的指针进行的。

因此，关键字 virtual 的作用就是在当使用基类的指针的时候，使子类中与基类同名的成员在适当的时候被调用，如前面例子中所示。

注意，虽然本身被定义为虚拟类型，我们还是可以声明一个 CPolygon 类型的对象并调用它的 area() 函数，它将返回 0 ，如前面例子结果所示。

抽象基类(Abstract base classes)

基本的抽象类与我们前面例子中的类 CPolygon 非常相似，唯一的区别是在我们前面的例子中，我们已经为类 CPolygon 的对象（例如对象 poly）定义了一个有

效地 area()函数，而在一个抽象类（abstract base class）中，我们可以对它不定义，而简单得在函数声明后面写 =0 (等于 0)。

类 CPolygon 可以写成这样：

```
// abstract class CPolygon
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a;
            height=b;
        }
        virtual int area (void) =0;
};
```

注意我们是如何在 virtual int area (void)加 =0 来代替函数的具体实现的。这种函数被称为纯虚拟函数（pure virtual function），而所有包含纯虚拟函数的类被称为抽象基类(abstract base classes)。

抽象基类的最大不同是它不能够有实例(对象)，但我们可以定义指向它的指针。因此，像这样的声明：

```
CPolygon poly;
```

对于前面定义的抽象基类是不合法的。

然而，指针：

```
CPolygon * ppoly1;
CPolygon * ppoly2
```

是完全合法的。这是因为该类包含的纯虚拟函数(pure virtual function) 是没有被实现的，而又不可能生成一个不包含它的所有成员定义的对象。然而，因为这个函数在其子类中被完整的定义了，所以生成一个指向其子类的对象的指针是完全合法的。

下面是完整的例子：

```
// virtual members                          20
#include <iostream.h>                       10
class CPolygon {
    protected:
```

```
        int width, height;
    public:
        void set_values (int a, int b)
{
            width=a;
            height=b;
        }
        virtual int area (void) =0;
};
class CRectangle: public CPolygon {
    public:
        int area (void) { return (width
* height); }
};
class CTriangle: public CPolygon {
    public:
        int area (void) {
            return (width * height / 2);
        }
};
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    return 0;
}
```

再看一遍这段程序，你会发现我们可以用同一种类型的指针(CPolygon*)指向不同类的对象，至一点非常有用。 想象一下，现在我们可以写一个 CPolygon 的成员函数，使得它可以将函数 area()的结果打印到屏幕上，而不必考虑具体是为哪一个子类。

```
// virtual members                      20
#include <iostream.h>                   10
class CPolygon {
    protected:
        int width, height;
```

```
    public:
        void set_values (int a, int b)
{
            width=a;
            height=b;
        }
        virtual int area (void) =0;
        void printarea (void) {
            cout << this->area() <<
endl;
        }
};
class CRectangle: public CPolygon {
    public:
        int area (void) { return (width
* height); }
};
class CTriangle: public CPolygon {
    public:
        int area (void) {
            return (width * height / 2);
        }
};
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    return 0;
}
```

记住，this 代表代码正在被执行的这一个对象的指针。

抽象类和虚拟成员赋予了 C++ 多态(polymorphic)的特征，使得面向对象的编程 object-oriented programming 成为一个有用的工具。这里只是展示了这些功能最简单的用途。想象一下如果在对象数组或动态分配的对象上使用这些功能，将会节省多少麻烦。

5.1 模板(Templates)

模板(Templates)是 ANSI-C++ 标准中新引入的概念。如果你使用的 C++ 编译器不符合这个标准，则你很可能不能使用模板。

函数模板(Function templates)

模板(Templates)使得我们可以生成通用的函数，这些函数能够接受任意数据类型的参数,可返回任意类型的值,而不需要对所有可能的数据类型进行函数重载。这在一定程度上实现了宏（macro）的作用。它们的原型定义可以是下面两种中的任何一个：

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

上面两种原型定义的不同之处在关键字 class 或 typename 的使用。它们实际是完全等价的，因为两种表达的意思和执行都一模一样。

例如，要生成一个模板，返回两个对象中较大的一个，我们可以这样写：

```
template <class GenericType>

GenericType GetMax (GenericType a, GenericType b) { return (a>b?a:b); }
```

在第一行声明中,我们已经生成了一个通用数据类型的模板,叫做 GenericType。因此在其后面的函数中,GenericType 成为一个有效的数据类型，它被用来定义了两个参数 a 和 b ，并被用作了函数 GetMax 的返回值类型。

GenericType 仍没有代表任何具体的数据类型；当函数 GetMax 被调用的时候,我们可以使用任何有效的数据类型来调用它。这个数据类型将被作为 pattern 来代替函数中 GenericType 出现的地方。用一个类型 pattern 来调用一个模板的方法如下：

```
function <type> (parameters);
```

例如，要调用 GetMax 来比较两个 int 类型的整数可以这样写：

```
int x,y;
GetMax <int> (x,y);
```

因此,GetMax 的调用就好像所有的 GenericType 出现的地方都用 int 来代替一样。

这里是一个例子：

```
// function template                    6
                                        10
#include <iostream.h>

template <class T> T GetMax (T a, T b)
{

        T result;

        result = (a>b)? a : b;

        return (result);

}

int main () {

        int i=5, j=6, k;

        long l=10, m=5, n;

        k=GetMax(i,j);

        n=GetMax(l,m);

        cout << k << endl;

        cout << n << endl;

        return 0;
}
```

(在这个例子中，我们将通用数据类型命名为 T 而不是 GenericType ，因为 T 短一些，并且它是模板更为通用的标示之一，虽然使用任何有效的标示符都是可以的。)

在上面的例子中，我们对同样的函数 GetMax()使用了两种参数类型：int 和 long，而只写了一种函数的实现，也就是说我们写了一个函数的模板，用了两种不同的 pattern 来调用它。

如你所见，在我们的模板函数 GetMax() 里，类型 T 可以被用来声明新的对象

```
T result;
```

result 是一个 T 类型的对象， 就像 a 和 b 一样，也就是说，它们都是同一类型的，这种类型就是当我们调用模板函数时写在尖括号<> 中的类型。

在这个具体的例子中，通用类型 T 被用作函数 GetMax 的参数，不需要说明<int>或 <long>，编译器也可以自动检测到传入的数据类型，因此，我们也可以这样写这个例子：

```
int i,j;
GetMax (i,j);
```

因为 i 和 j 都是 int 类型,编译器会自动假设我们想要函数按照 int 进行调用。这种暗示的方法更为有用，并产生同样的结果：

```
// function template II                    6
                                           10
#include <iostream.h>

template <class T> T GetMax (T a, T b)
{

        return (a>b?a:b);

}

int main () {

        int i=5, j=6, k;

        long l=10, m=5, n;

        k=GetMax(i,j);

        n=GetMax(l,m);

        cout << k << endl;

        cout << n << endl;

        return 0;

}
```

注意在这个例子的 main() 中我们如何调用模板函数 GetMax() 而没有在括号<> 中指明具体数据类型的。编译器自动决定每一个调用需要什么数据类型。

因为我们的模板函数只包括一种数据类型 (class T)， 而且它的两个参数都是同一种类型，我们不能够用两个不同类型的参数来调用它：

```
int i;
long l;
k = GetMax (i,l);
```

上面的调用就是不对的，因为我们的函数等待的是两个同种类型的参数。

我们也可以使得模板函数接受两种或两种以上类型的数据，例如：

```
template <class T>

T GetMin (T a, U b) { return (a<b?a:b); }
```

在这个例子中，我们的模板函数 GetMin() 接受两个不同类型的参数，并返回一个与第一个参数同类型的对象。在这种定义下，我们可以这样调用该函数：

```
int i,j;
long l;
i = GetMin <int, long> (j,l);
```

或者，简单的用

```
i = GetMin (j,l);
```

虽然 j 和 l 是不同的类型。

类模板(Class templates)

我们也可以定义类模板（class templates），使得一个类可以有基于通用类型的成员，而不需要在类生成的时候定义具体的数据类型，例如：

```
template <class T>

class pair {

        T values [2];

public:

        pair (T first, T second) {
```

```
                values[0]=first;

                values[1]=second;

        }

};
```

上面我们定义的类可以用来存储两个任意类型的元素。例如，如果我们想要定义该类的一个对象，用来存储两个整型数据 115 和 36 ，我们可以这样写：

```
pair<int> myobject (115, 36);
```

我们同时可以用这个类来生成另一个对象用来存储任何其他类型数据，例如：

```
pair<float> myfloats (3.0, 2.18);
```

在上面的例子中，类的唯一一个成员函数已经被 inline 定义。如果我们要在类之外定义它的一个成员函数，我们必须在每一函数前面加 template <... >。

```
// class templates                          100

#include <iostream.h>

template <class T> class pair {
         T value1, value2;
public:
        pair (T first, T second) {

                value1=first;

                value2=second;

        }

        T getmax ();

};

template <class T>

T pair::getmax (){

        T retval;
```

```
        retval = value1>value2? value1 :
value2;

        return retval;

}

int main () {

        pair myobject (100, 75);

        cout << myobject.getmax();

        return 0;

}
```

注意成员函数 getmax 是怎样开始定义的：

```
template <class T>
T pair::getmax ()
```

所有写 T 的地方都是必需的，每次你定义模板类的成员函数的时候都需要遵循类似的格式（这里第二个 T 表示函数返回值的类型，这个根据需要可能会有变化）。

模板特殊化(Template specialization)

模板的特殊化是当模板中的 pattern 有确定的类型时，模板有一个具体的实现。例如假设我们的类模板 pair 包含一个取模计算（module operation）的函数，而我们希望这个函数只有当对象中存储的数据为整型(int)的时候才能工作，其他时候，我们需要这个函数总是返回 0。这可以通过下面的代码来实现：

```
// Template specialization                25
                                          0
#include <iostream.h>

template <class T> class pair {

        T value1, value2;
```

```
public:
        pair (T first, T second){
                value1=first;
                value2=second;
        }
        T module () {return 0;}
};
template <>
class pair <int> {
        int value1, value2;
public:
        pair (int first, int second){
                value1=first;
                value2=second;
        }
        int module ();
};
template <>
int pair<int>::module() {
        return value1%value2;
}
int main () {
```

```
        pair <int> myints (100,75);

        pair <float> myfloats
(100.0,75.0);

        cout << myints.module() << '\n';

        cout << myfloats.module() <<
'\n';

        return 0;

}
```

由上面的代码可以看到，模板特殊化由以下格式定义：

```
template <> class class_name <type>
```

这个特殊化本身也是模板定义的一部分，因此，我们必须在该定义开头写 template <>。而且因为它确实为一个具体类型的特殊定义，通用数据类型在这里不能够使用，所以第一对尖括号<> 内必须为空。在类名称后面，我们必须将这个特殊化中使用的具体数据类型写在尖括号<>中。

当我们特殊化模板的一个数据类型的时候，同时还必须重新定义类的所有成员的特殊化实现（如果你仔细看上面的例子，会发现我们不得不在特殊化的定义中包含它自己的构造函数 constructor，虽然它与通用模板中的构造函数是一样的）。这样做的原因就是特殊化不会继承通用模板的任何一个成员。

模板的参数值(Parameter values for templates)

除了模板参数前面跟关键字 class 或 typename 表示一个通用类型外，函数模板和类模板还可以包含其它不是代表一个类型的参数，例如代表一个常数，这些通常是基本数据类型的。例如，下面的例子定义了一个用来存储数组的类模板：

```
// array template                          100
                                           3.1416
#include <iostream.h>

template <class T, int N>
```

```
class array {
        T memblock [N];
public:
        void setmember (int x, T value);
        T getmember (int x);
};
template <class T, int N>
void array<T,N>::setmember (int x, T
value) {
        memblock[x]=value;
}
template <class T, int N>
T array<T,N>::getmember (int x) {
        return memblock[x];
}
int main () {
        array <int,5> myints;
        array <float,5> myfloats;
        myints.setmember (0,100);
        myfloats.setmember (3,3.1416);
        cout << myints.getmember(0) <<
'\n';
        cout << myfloats.getmember(3) <<
'\n';
```

```
        return 0;
}
```

我们也可以为模板参数设置默认值，就像为函数参数设置默认值一样。

下面是一些模板定义的例子：

```
template <class T> // 最常用的：一个 class 参数。
template <class T, class U> // 两个 class 参数。
template <class T, int N> // 一个 class 和一个整数。
template <class T = char> // 有一个默认值。
template <int Tfunc (int)> // 参数为一个函数。
```

模板与多文件工程 (Templates and multiple-file projects)

从编译器的角度来看，模板不同于一般的函数或类。它们在需要时才被编译(compiled on demand)，也就是说一个模板的代码直到需要生成一个对象的时候(instantiation)才被编译。当需要 instantiation 的时候，编译器根据模板为特定的调用数据类型生成一个特殊的函数。

当工程变得越来越大的时候，程序代码通常会被分割为多个源程序文件。在这种情况下，通常接口(interface)和实现(implementation)是分开的。用一个函数库做例子，接口通常包括所有能被调用的函数的原型定义。它们通常被定义在以.h 为扩展名的头文件 (header file) 中；而实现 (函数的定义) 则在独立的C++代码文件中。

模板这种类似宏(macro-like) 的功能，对多文件工程有一定的限制：函数或类模板的实现 (定义) 必须与原型声明在同一个文件中。也就是说我们不能再 将接口(interface)存储在单独的头文件中，而必须将接口和实现放在使用模板的同一个文件中。

回到函数库的例子，如果我们想要建立一个函数模板的库，我们不能再使用头文件(.h) ，取而代之，我们应该生成一个模板文件(template file)，将函数模板的接口和实现 都放在这个文件中 (这种文件没有惯用扩展名，除了不要使用.h 扩展名或不要不加任何扩展名)。在一个工程中多次包含同时具有声明和实现的模板文件并不会产生链接错误 (linkage errors)，因为它们只有在需要时才被编译，而兼容模板的编译器应该已经考虑到这种情况，不会生成重复的代码。

5.2 名空间 (Namespaces)

通过使用名空间(Namespaces)我们可以将一组全局范围有效的类、对象或函数组织到一个名字下面。换种说法，就是它将全局范围分割成许多子域范围，每个子域范围叫做一个名空间(namespaces).

使用名空间的格式是：

```
namespace identifier
{
    namespace-body
}
```

这里 identifier 是一个有效的标示符，namespace-body 是该名空间包含的一组类、对象和函数。例如：

```
namespace general
{
    int a, b;
}
```

在这个例子中，a 和 b 是名空间 general 中的整型变量。要想在这个名空间外面访问这两个变量，我们必须使用范围操作符::。例如，要想访问前面的两个变量，我们需要这样写：

```
general::a
general::b
```

名空间(namespaces)的作用在于全局对象或函数很有可能重名而造成重复定义的错误，名空间的使用可以避免这些错误的发生。例如：

```
// namespaces                    5
#include <iostream.h>            3.1416

namespace first {
    int var = 5;
}

namespace second {
    double var = 3.1416;
}
```

```
int main () {
    cout << first::var << endl;
    cout << second::var << endl;
    return 0;
}
```

在这个例子中，两个都叫做 var 全局变量同时存在，一个在名空间 first 下面定义，另一个在 second 下面定义，由于我们使用了名空间，这里不会产生重复定义的错误。

名空间的使用 (using namespace)

使用 using 指令后面跟 namespace 可以将当前的嵌套层与一个指定的名空间连在一起，以便使该名空间下定义的对象和函数可以被访问，就好像它们是在全局范围内被定义的一样。它的使用遵循以下原型定义：

```
using namespace identifier;
```

例如：

```
// using namespace example      3.1416
#include <iostream.h>           6.2832

namespace first {
    int var = 5;
}

namespace second {
    double var = 3.1416;
}

int main () {
    using namespace second;
    cout << var << endl;
    cout << (var*2) << endl;
    return 0;
}
```

在这个例子中的 main 函数中可以看到，我们能够直接使用变量 var 而不用在前面加任何范围操作符。

这里要注意，语句 using namespace 只在其被声明的语句块内有效（一个语句块指在一对花括号{}内的一组指令），如果 using namespace 是在全局范围内被声明的，则在所有代码中都有效。例如，如果我们想在一段程序中使用一个名空间，而在另一段程序中使用另一个名空间，则可以像以下代码中那样做：

```
// using namespace example          5
#include <iostream.h>               3.1416

namespace first {
    int var = 5;
}

namespace second {
    double var = 3.1416;
}

int main () {
    {
        using namespace first;
        cout << var << endl;
    }
    {
        using namespace second;
        cout << var << endl;
    }
    return 0;
}
```

别名定义(alias definition)

我们以可以为已经存在的名空间定义别名，格式为：

```
namespace new_name = current_name ;
```

标准名空间(Namespace std)

我们能够找到的关于名空间的最好的例子就是标准 C++ 函数库本身。如 ANSI C++ 标准定义，标准 C++库中的所有类、对象和函数都是定义在名空间 std 下面的。

你可能已经注意到，我们在这个教程中全部忽略了这一点。作者决定这么做是因为这条规则几乎和 ANSI 标准本身一样年轻 (1997)，许多老一点的编译器并不兼容这条规则。

几乎所有的编译器，即使是那些与 ANSI 标准兼容的编译器，都允许使用传统的头文件 (如 iostream.h, stdlib.h, 等等)，就像我们在这个教程中所使用的一样。然而，ANSI 标准完全重新设计了这些函数库，利用了模板功能，而且遵循了这条规则将所有的函数和变量定义在了名空间 std 下。

该标准为这些头文件定义了新的名字，对针对 C++的文件基本上是使用同样的名字，但没有.h 的扩展名，例如, iostream.h 变成了 iostream。

如果我们使用 ANSI-C++ 兼容的包含文件，我们必须记住所有的函数、类和对象是定义在名空间 std 下面的，例如：

```
    // ANSI-C++ compliant hello world   Hello world in ANSI-C++
    #include <iostream>

    int main () {
        std::cout << "Hello world in
ANSI-C++\n";
        return 0;
    }
```

更常用的方法是使用 using namespace ，这样我们就不必在所有标准空间中定义的函数或对象前面总是使用范围操作符::了 ：

```
    // ANSI-C++ compliant hello world   Hello world in ANSI-C++
(II)
    #include <iostream>
    using namespace std;

    int main () {
        cout << "Hello world in
ANSI-C++\n";
        return 0;
    }
```

对于 STL 用户，强烈建议使用 ANSI-compliant 方式来包含标准函数库。

5.3 出错处理 (Exception handling)

本节介绍的出错处理是 ANSI-C++ 标准引入的新功能。如果你使用的 C++ 编译器不兼容这个标准，则你可能无法使用这些功能。

在编程过程中，很多时候我们是无法确定一段代码是否总是能够正常工作的，或者因为程序访问了并不存在的资源，或者由于一些变量超出了预期的范围，等等。

这些情况我们统称为出错（例外），C++ 新近引入的三种操作符能够帮助我们处理这些出错情况： try, throw 和 catch 。

它们的一般用法是：

```
try {
    // code to be tried
    throw exception;
}
catch (type  exception)
{
    // code to be executed in case of exception
}
```

它们所进行的操作是：

- try 语句块中的代码被正常执行。如果有例外发生，代码必须使用关键字 throw 和一个参数来扔出一个例外。这个参数可以是任何有效的数据类型，它的类型反映了例外的特征。
- 如果有例外发生，也就是说在 try 语句块中有一个 throw 指令被执行了，则 catch 语句块会被执行，用来接收 throw 传来的例外参数。

例如：

```
// exceptions                          Exception: Out of range
#include <iostream.h>

int main () {
    char myarray[10];
    try {
        for (int n=0; n<=10; n++) {
            if (n>9) throw "Out of
range";
            myarray[n]='z';
        }
```

```
        } catch (char * str) {
            cout << "Exception: " << str
<< endl;
        }
        return 0;
    }
```

在这个例子中，如果在 n 循环中，n 变的大于 9 了，则仍出一个例外，因为数组 myarray[n] 在这种情况下会指向一个无效的内存地址。当 throw 被执行的时候，try 语句块立即被停止执行，在 try 语句块中生成的所有对象会被销毁。此后，控制权被传递给相应的 catch 语句块（上面的例子中即执行仅有的一个 catch）。最后程序紧跟着 catch 语句块继续向下执行，在上面的例子中就是执行 return 0;.

throw 语法与 return 相似，只是参数不需要扩在括号。

catch 语句块必须紧跟着 try 语句块后面，中间不能有其它的代码。catch 捕获的参数可以是任何有效的数据类型。catch 甚至可以被重载以便能够接受不同类型的参数。在这种情况下被执行 catch 语句块是相应的符合被 throw 扔出的参数类型的那一个：

```
    // exceptions: multiple catch
blocks
    #include <iostream.h>

    int main () {
        try {
            char * mystring;
            mystring = new char [10];
            if (mystring == NULL) throw
"Allocation failure";
            for (int n=0; n<=100; n++)
{
                if (n>9) throw n;
                mystring[n]='z';
            }
        } catch (int i) {
            cout << "Exception: ";
            cout << "index " << i << "
is out of range" << endl;
        } catch (char * str) {
            cout << "Exception: " << str
```

```
Exception: index 10 is out of
range
```

```
<< endl;
        }
        return 0;
    }
```

在上面的例子中，有两种不同的例外可能发生：

1. 如果要求的 10 个字符空间不能够被赋值（这种情况很少，但是有可能发生）：这种情况下扔出的例外会被 catch (char * str)捕获。
2. n 超过了 mystring 的最大索引值(index)：这种情况下扔出的例外将被 catch (int i)捕获，因为它的参数是一个整型值。

我们还可以定义一个 catch 语句块来捕获所有的例外，不论扔出的是什么类型的参数。这种情况我们需要在 catch 或面的括号中写三个点来代替原来的参数类型和名称，如：

```
try {
    // code here
}
catch (...) {
    cout << "Exception occurred";
}
```

try-catch 也是可以被嵌套使用的。在这种情况下，我们可以使用表达式 throw;(不带参数)将里面的 catch 语句块捕获的例外传递到外面一层，例如：

```
try {
    try {
        // code here
    }
    catch (int n) {
        throw;
    }
}
catch (...) {
    cout << "Exception occurred";
}
```

没有捕获的例外 (Exceptions not caught)

如果由于没有对应的类型，一个例外没有被任何 catch 语句捕获，特殊函数 terminate 将被调用。

这个函数通常已被定义了，以便立即结束当前的进程(process)，并显示一个“非正常结束”("Abnormal termination")的出错信息。它的格式是：

```
void terminate();
```

标准例外 (Standard exceptions)

一些 C++ 标准语言库中的函数也会扔出一些列外，我们可以用 try 语句来捕获它们。这些例外扔出的参数都是 std::exception 引申出的子类类型的。这个类(std::exception) 被定义在 C++ 标准头文件 中，用来作为 exceptions 标准结构的模型：

```
exception
  bad_alloc          (thrown by new)
  bad_cast           (thrown by dynamic_cast when fails with a referenced type)
  bad_exception      (thrown when an exception doesn't match any catch)
  bad_typeid         (thrown by typeid)
  logic_error
    domain_error
    invalid_argument
    length_error
    out_of_range
  runtime_error
    overflow_error
    range_error
    underflow_error
  ios_base::failure  (thrown by ios::clear)
```

因为这是一个类结构，如果你包括了一个 catch 语句块使用地址(reference)来捕获这个结构中的任意一种列外 (也就是说在类型后面加地址符 &)，你同时可以捕获所有引申类的例外 (C++的继承原则)。

下面的例子中，一个类型为 bad_typeid 的例外(exception 的引申类)，在要求类型信息的对象为一个空指针的时候被捕获：

```
// standard exceptions
#include <iostream.h>
#include <exception>
#include <typeinfo>

class A {virtual f() {}; };

int main () {
    try {
        A * a = NULL;
        typeid (*a);
    } catch (std::exception& e) {
```

```
Exception: Attempted typeid of
NULL pointer
```

```
            cout << "Exception: " <<
e.what();
        }
        return 0;
    }
```

你可以用这个标准的例外层次结构来定义的你的例外或从它们引申出新的例外类型。

5.4 类型转换高级 (Advacned Class Type-casting)

目前为止，我们一直使用传统的类型转换符来进行简单对象的类型转换。例如，要把一个 double 类型的浮点型数字转换为 int 的整型数字，我们是这样做的：

```
int i;
double d;
i = (int) d;
```

或者

```
i = int (d);
```

这样做对基本数据类型时没问题的，因为基本数据类型的转换已经有标准的定义。同样的操作也可以被在类或类的指针上，因此以下例子中的写法也是没有问题的：

```
    // class type-casting
    #include <iostream.h>

    class CDummy {
        int i;
    };

    class CAddition {
        int x,y;
      public:
        CAddition (int a, int b) { x=a;
y=b; }
        int result() { return x+y;}
    };
```

```
int main () {
    CDummy d;
    CAddition * padd;
    padd = (CAddition*) &d;
    cout << padd->result();
    return 0;
}
```

虽然以上程序在 C++中是没有语法错误的(多数编译器甚至不会产生警告信息)，但这段程序没有什么实际的逻辑意义。我们使用了 CAddition 的成员函数 result 而没有定义一个相应的该类的对象：padd 并不是一个对象，它只是一个指针，被我们赋值指向一个毫无关系的对象的地址。当在程序运行到访问它的 result 成员函数时，将会有一个运行错误(run-time error)产生，或生成一个意外的结果。

为了控制这种类之间的转换，ANSI-C++ 标准定义了 4 种新的类型转换操作符：reinterpret_cast, static_cast, dynamic_cast 和 const_cast。所有这些操作符都是同样的使用格式：

```
reinterpret_cast <new_type> (expression)
dynamic_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

这里 new_type 是要转换成的目标类型，expression 是要被转换的内容。为了便于理解，模仿传统转换操作符，它们的含义是这样的：

```
(new_type) expression
new_type (expression)
```

reinterpret_cast

reinterpret_cast 可以将一个指针转换为任意其它类型的指针。它也可以用来将一个指针转换为一个整型，或反之亦然。

这个操作符可以在互不相关的类之间进行指针转换，操作的结果是简单的将一个指针的二进制数据(binary copy)复制到另一个指针。对指针指向的内容不做任何检查或转换。

如果这种复制发生在一个指针到一个整数之间，则对其内容的解释取决于不同的系统，因此任何实现都是不可移植(non portable)的。一个指针如果被转换为一个能够完全存储它的足够大的整数中，则是可以再被转换回来成为指针的。

例如：

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

reinterpret_cast 对所有指针的处理与传统的类型转换符所作的一模一样。

static_cast

static_cast 可以执行所有能够隐含执行的类型转换，以及它们的反向操作（即使这种方向操作是不允许隐含执行的）。

用于类的指针，也就是说，它允许将一个引申类的指针转换为其基类类型（这是可以被隐含执行的有效转换），同时也允许进行相反的转换：将一个基类转换为一个引申类类型。

在后面一种情况中，不会检查被转换的基类是否真正完全是目标类型的。例如下面的代码是合法的：

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast(a);
```

static_cast 除了能够对类指针进行操作，还可以被用来进行类中明确定义的转换，以及对基本类型的标准转换：

```
double d=3.14159265;
int i = static_cast<int>(d);
```

译者注：如果你对这部分看不太懂，请结合下面的 dynamic_cast 一起看，也许会帮助理解。

dynamic_cast

dynamic_cast 完全被用来进行指针的操作。它可以用来进行任何可以隐含进行的转换操作以及它们被用于多态类情况下的方向操作。然而与 static_cast 不同的是， dynamic_cast 会检查后一种情况的操作是否合法，也就是说它会检查类型转换操作是否会返回一个被要求类型的有效的完整的对象。

这种检查是在程序运行过程中进行的。如果被转换的指针所指向的对象不是一个被要求类型的有效完整的对象，返回值将会是一个空指针 NULL 。

```
class Base { virtual dummy(){}; };
class Derived : public Base { };


Base* b1 = new Derived;
Base* b2 = new Base;
Derived* d1 = dynamic_cast(b1);   // succeeds
Derived* d2 = dynamic_cast(b2);   // fails: returns NULL
```

如果类型转换被用在引用(reference)类型上，而这个转换不可能进行的话，一个 bad_cast 类型的例外(exception)将会被抛出：

```
class Base { virtual dummy(){}; };
class Derived : public Base { };

Base* b1 = new Derived;
Base* b2 = new Base;
Derived d1 = dynamic_cast(b1);   // succeeds
Derived d2 = dynamic_cast(b2);   // fails: exception thrown
```

const_cast

这种类型转换对常量 const 进行设置或取消操作：

```
class C {};
const C * a = new C;
C * b = const_cast<C*> (a);
```

其他 3 种 cast 操作符都不可以修改一个对象的常量属性(constness)。

typeid

ANSI-C++ 还定义了一个新的操作符叫做 typeid ，它检查一个表达式的类型：

```
typeid (expression)
```

这个操作符返回一个类型为 type_info 的常量对象指针，这种类型定义在标准头函数中。这种返回值可以用操作符 == 和 != 来互相进行比较，也可以用来通过 name()函数获得一个描述数据类型或类名称的字符串，例如：

```
    // typeid, typeinfo                    a and b are of different types:
```

```
    #include <iostream.h>
    #include <typeinfo>
    class CDummy { };

    int main () {
        CDummy* a,b;
        if (typeid(a) != typeid(b)) {
            cout << "a and b are of
different types:\n";
            cout << "a is: " <<
typeid(a).name() << '\n';
            cout << "b is: " <<
typeid(b).name() << '\n';
        }
        return 0;
    }
```

```
a is: class CDummy *
b is: class CDummy
```

5.5 预处理指令 (Preprocessor Directives)

预处理指令是我们写在程序代码中的给预处理器(preprocessor)的 命令，而不是程序本身的语句。预处理器在我们编译一个 C++程序时由编译器自动执行，它负责控制对程序代码的第一次验证和消化。

所有这些指令必须写在单独的一行中，它们不需要加结尾的分号;。

#define

在这个教程的开头我们已经提到了一种预处理指令： #define ，可以被用来生成宏定义常量(defined constantants 或 macros)，它的形式是：

```
#define name value
```

它的作用是定义一个叫做 name 的宏定义，然后每当在程序中遇到这个名字的时候，它就会被 value 代替，例如：

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
char str2[MAX_WIDTH];
```

它定义了两个最多可以存储 100 个字符的字符串。

#define 也可以被用来定义宏函数：

```
#define getmax(a,b) a>b?a:b
int x=5, y;
y = getmax(x,2);
```

这段代码执行后 y 的值为 5 。

#undef

#undef 完成与 #define 相反的工作，它取消对传入的参数的宏定义：

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
#undef MAX_WIDTH
#define MAX_WIDTH 200
char str2[MAX_WIDTH];
```

#ifdef, #ifndef, #if, #endif, #else and #elif

这些指令可以使程序的一部分在某种条件下被忽略。

#ifdef 可以使一段程序只有在某个指定常量已经被定义了的情况下才被编译，无论被定义的值是什么。它的操作是：

```
#ifdef name
// code here
#endif
```

例如：

```
#ifdef MAX_WIDTH
char str[MAX_WIDTH];
#endif
```

在这个例子中，语句 char str[MAX_WIDTH]; 只有在宏定义常量 MAX_WIDTH 已经被定义的情况下才被编译器考虑，不管它的值是什么。如果它还没有被定义，这一行代码则不会被包括在程序中。

#ifndef 起相反的作用：在指令#ifndef 和 #endif 之间的代码只有在某个常量没有被定义的情况下才被编译，例如：

```
#ifndef MAX_WIDTH
#define MAX_WIDTH 100
```

```
#endif
char str[MAX_WIDTH];
```

这个例子中，如果当处理到这段代码的时候 MAX_WIDTH 还没有被定义，则它会被定义为值 100。而如果它已经被定义了，那么它会保持原值（因为#define 语句这一行不会被执行）。

指令#if, #else 和 #elif (elif = else if) 用来使得其后面所跟的程序部分只有在特定条件下才被编译。这些条件只能够是常量表达式，例如：

```
#if MAX_WIDTH>200
#undef MAX_WIDTH
#define MAX_WIDTH 200

#elsif MAX_WIDTH<50
#undef MAX_WIDTH
#define MAX_WIDTH 50

#else
#undef MAX_WIDTH
#define MAX_WIDTH 100
#endif

char str[MAX_WIDTH];
```

注意看这一连串的指令 #if, #elsif 和 #else 是怎样以 #endif 结尾的。

#line

当我们编译一段程序的时候，如果有错误发生，编译器会在错误前面显示出错文件的名称以及文件中的第几行发生的错误。

指令#line 可以使我们对这两点进行控制，也就是说当出错时显示文件中的行数以及我们希望显示的文件名。它的格式是：

```
#line number "filename"
```

这里 number 是将会赋给下一行的新行数。它后面的行数从这一点逐个递增。

filename 是一个可选参数，用来替换自此行以后出错时显示的文件名，直到有另外一个#line 指令替换它或直到文件的末尾。例如：

```
#line 1 "assigning variable"
int a?;
```

这段代码将会产生一个错误，显示为在文件"assigning variable", line 1 。

#error

这个指令将中断编译过程并返回一个参数中定义的出错信息，例如：

```
#ifndef __cplusplus
#error A C++ compiler is required
#endif
```

这个例子中如果 __cplusplus 没有被定义就会中断编译过程。

#include

这个指令我们已经见到很多次。当预处理器找到一个#include 指令时，它用指定文件的全部内容替换这条语句。声明包含一个文件有两种方式：

```
#include "file"
#include <file>
```

两种表达的唯一区别是编译器应该在什么路经下寻找指定的文件。第一种情况下，文件名被写在双引号中，编译器首先在包含这条指令的文件所在的目录下进行寻找，如果找不到指定文件，编译器再到被配置的默认路径下（也就是标准头文件路径下）进行寻找。

如果文件名是在尖括号 <> 中，编译器会直接到默认标准头文件路径下寻找。

#pragma

这个指令是用来对编译器进行配置的，针对你所使用的平台和编译器而有所不同。要了解更多信息，请参考你的编译器手册。

第六章 C++ 标准函数库

C++ Standard Library

文件的输入输出 (Input/Output with files)

C++ 通过以下几个类支持文件的输入输出：

- ofstream: 写操作（输出）的文件类（由 ostream 引申而来）
- ifstream: 读操作（输入）的文件类(由 istream 引申而来)
- fstream: 可同时读写操作的文件类（由 iostream 引申而来）

打开文件(Open a file)

对这些类的一个对象所做的第一个操作通常就是将它和一个真正的文件联系起来，也就是说打开一个文件。被打开的文件在程序中由一个流对象(stream object)来表示（这些类的一个实例），而对这个流对象所做的任何输入输出操作实际就是对该文件所做的操作。

要通过一个流对象打开一个文件，我们使用它的成员函数 open()：

```
void open (const char * filename, openmode mode);
```

这里 filename 是一个字符串，代表要打开的文件名，mode 是以下标志符的一个组合：

ios::in	为输入(读)而打开文件
ios::out	为输出(写)而打开文件
ios::ate	初始位置：文件尾
ios::app	所有输出附加在文件末尾
ios::trunc	如果文件已存在则先删除该文件
ios::binary	二进制方式

这些标识符可以被组合使用，中间以”或”操作符(|)间隔。例如，如果我们想要以二进制方式打开文件"example.bin" 来写入一些数据，我们可以通过以下方式调用成员函数 open（）来实现：

```
ofstream file;
file.open ("example.bin", ios::out | ios::app | ios::binary);
```

ofstream, ifstream 和 fstream 所有这些类的成员函数 open 都包含了一个默认打开文件的方式，这三个类的默认方式各不相同：

类	参数的默认方式
ofstream	ios::out | ios::trunc
ifstream	ios::in
fstream	ios::in | ios::out

只有当函数被调用时没有声明方式参数的情况下，默认值才会被采用。如果函数被调用时声明了任何参数，默认值将被完全改写，而不会与调用参数组合。

由于对类 ofstream, ifstream 和 fstream 的对象所进行的第一个操作通常都是打开文件，这些类都有一个构造函数可以直接调用 open 函数，并拥有同样的参数。这样，我们就可以通过以下方式进行与上面同样的定义对象和打开文件的操作：

```
ofstream file ("example.bin", ios::out | ios::app | ios::binary);
```

两种打开文件的方式都是正确的。

你可以通过调用成员函数 is_open()来检查一个文件是否已经被顺利的打开了：

```
bool is_open();
```

它返回一个布尔(bool)值，为真（true）代表文件已经被顺利打开，假（false）则相反。

关闭文件(Closing a file)

当文件读写操作完成之后，我们必须将文件关闭以使文件重新变为可访问的。关闭文件需要调用成员函数 close()，它负责将缓存中的数据排放出来并关闭文件。它的格式很简单：

```
void close ();
```

这个函数一旦被调用，原先的流对象(stream object)就可以被用来打开其它的文件了，这个文件也就可以重新被其它的进程(process)所有访问了。

为防止流对象被销毁时还联系着打开的文件，析构函数(destructor)将会自动调用关闭函数 close。

文本文件(Text mode files)

类ofstream, ifstream 和fstream 是分别从ostream, istream 和iostream 中引申而来的。这就是为什么 fstream 的对象可以使用其父类的成员来访问数据。

一般来说，我们将使用这些类与同控制台(console)交互同样的成员函数(cin 和 cout)来进行输入输出。如下面的例题所示，我们使用重载的插入操作符<<：

```
// writing on a text file
#include <fiostream.h>

int main () {
    ofstream examplefile
("example.txt");
    if (examplefile.is_open()) {
        examplefile << "This is a
line.\n";
        examplefile << "This is
another line.\n";
        examplefile.close();
    }
    return 0;
}
```

```
file example.txt
This is a line.
This is another line.
```

从文件中读入数据也可以用与 cin 的使用同样的方法：

```
// reading a text file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main () {
    char buffer[256];
    ifstream examplefile
("example.txt");
    if (! examplefile.is_open())
    { cout << "Error opening file";
exit (1); }
    while (! examplefile.eof() ) {
        examplefile.getline
(buffer,100);
        cout << buffer << endl;
    }
    return 0;
}
```

```
This is a line.
This is another line.
```

上面的例子读入一个文本文件的内容，然后将它打印到屏幕上。注意我们使用了一个新的成员函数叫做 eof ，它是 ifstream 从类 ios 中继承过来的，当到达文件末尾时返回 true 。

状态标志符的验证(Verification of state flags)

除了 eof()以外，还有一些验证流的状态的成员函数（所有都返回 bool 型返回值）：

- bad()

  如果在读写过程中出错，返回 true 。例如：当我们要对一个不是打开为写状态的文件进行写入时，或者我们要写入的设备没有剩余空间的时候。

- fail()

  除了与 bad() 同样的情况下会返回 true 以外，加上格式错误时也返回 true ，例如当想要读入一个整数，而获得了一个字母的时候。

- eof()

  如果读文件到达文件末尾，返回 true。

- good()

  这是最通用的：如果调用以上任何一个函数返回 true 的话，此函数返回 false 。

要想重置以上成员函数所检查的状态标志，你可以使用成员函数 clear()，没有参数。

获得和设置流指针(get and put stream pointers)

所有输入/输出流对象(i/o streams objects)都有至少一个流指针：

- ifstream， 类似 istream，有一个被称为 get pointer 的指针，指向下一个将被读取的元素。
- ofstream，类似 ostream，有一个指针 put pointer ，指向写入下一个元素的位置。
- fstream，类似 iostream，同时继承了 get 和 put

我们可以通过使用以下成员函数来读出或配置这些指向流中读写位置的流指针：

- tellg() 和 tellp()

  这两个成员函数不用传入参数，返回 pos_type 类型的值(根据 ANSI-C++ 标准），就是一个整数，代表当前 get 流指针的位置 (用 tellg) 或 put 流指针的位置(用 tellp).

- seekg() 和 seekp()

  这对函数分别用来改变流指针 get 和 put 的位置。两个函数都被重载为两种不同的原型：

  ```
  seekg ( pos_type position );
  seekp ( pos_type position );
  ```

  使用这个原型，流指针被改变为指向从文件开始计算的一个绝对位置。要求传入的参数类型与函数 tellg 和 tellp 的返回值类型相同。

  ```
  seekg ( off_type offset, seekdir direction );
  seekp ( off_type offset, seekdir direction );
  ```

  使用这个原型可以指定由参数 direction 决定的一个具体的指针开始计算的一个位移(offset)。它可以是：

ios::beg	从流开始位置计算的位移
ios::cur	从流指针当前位置开始计算的位移
ios::end	从流末尾处开始计算的位移

流指针 get 和 put 的值对文本文件(text file)和二进制文件(binary file)的计算方法都是不同的，因为文本模式的文件中某些特殊字符可能被修改。由于这个原因，建议对以文本文件模式打开的文件总是使用 seekg 和 seekp 的第一种原型，而且不要对 tellg 或 tellp 的返回值进行修改。对二进制文件，你可以任意使用这些函数，应该不会有任何意外的行为产生。

以下例子使用这些函数来获得一个二进制文件的大小：

```
// obtaining file size
#include <iostream.h>
#include <fstream.h>

const char * filename =
"example.txt";

int main () {
```

```
size of example.txt is 40
bytes.
```

```
        long l,m;
        ifstream file (filename,
ios::in|ios::binary);
        l = file.tellg();
        file.seekg (0, ios::end);
        m = file.tellg();
        file.close();
        cout << "size of " << filename;
        cout << " is " << (m-l) << "
bytes.\n";
        return 0;
    }
```

二进制文件(Binary files)

在二进制文件中，使用<< 和>>，以及函数（如 getline）来操作符输入和输出数据，没有什么实际意义，虽然它们是符合语法的。

文件流包括两个为顺序读写数据特殊设计的成员函数：write 和 read。第一个函数 (write) 是ostream 的一个成员函数,都是被ofstream所继承。而read 是 istream 的一个成员函数，被 ifstream 所继承。类 fstream 的对象同时拥有这两个函数。它们的原型是：

```
write ( char * buffer, streamsize size );
read ( char * buffer, streamsize size );
```

这里 buffer 是一块内存的地址，用来存储或读出数据。参数 size 是一个整数值，表示要从缓存（buffer）中读出或写入的字符数。

```
    // reading binary file
    #include <iostream>
    #include <fstream.h>

    const char * filename =
"example.txt";

    int main () {
        char * buffer;
        long size;
        ifstream file (filename,
ios::in|ios::binary|ios::ate);
```

```
The complete file is in a
buffer
```

```
        size = file.tellg();
        file.seekg (0, ios::beg);
        buffer = new char [size];
        file.read (buffer, size);
        file.close();

        cout << "the complete file is in
a buffer";

        delete[] buffer;
        return 0;
    }
```

缓存和同步(Buffers and Synchronization)

当我们对文件流进行操作的时候，它们与一个 streambuf 类型的缓存(buffer)联系在一起。这个缓存（buffer）实际是一块内存空间，作为流(stream)和物理文件的媒介。例如，对于一个输出流，每次成员函数 put （写一个单个字符)被调用，这个字符不是直接被写入该输出流所对应的物理文件中的，而是首先被插入到该流的缓存（buffer）中。

当缓存被排放出来(flush)时，它里面的所有数据或者被写入物理媒质中（如果是一个输出流的话），或者简单的被抹掉(如果是一个输入流的话)。这个过程称为同步(synchronization)，它会在以下任一情况下发生：

- **当文件被关闭时**：在文件被关闭之前，所有还没有被完全写出或读取的缓存都将被同步。
- **当缓存 buffer 满时**:缓存 Buffers 有一定的空间限制。当缓存满时，它会被自动同步。
- **控制符明确指明**:当遇到流中某些特定的控制符时，同步会发生。这些控制符包括：flush 和 endl。
- **明确调用函数 sync()**：调用成员函数 sync() （无参数)可以引发立即同步。这个函数返回一个 int 值，等于-1 表示流没有联系的缓存或操作失败。