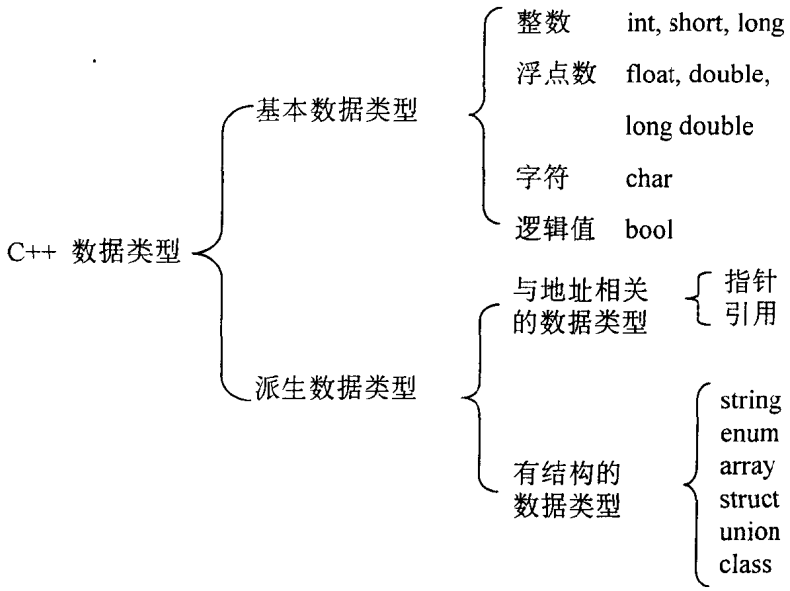
第3章

基本数据类型

在C++中，所有的变量和常量都属于某种特定的数据类型 (data type)。数据类型规范了所能占用的存储空间大小，以及对该数据所能进行的处理和操作方式。本章介绍整数、浮点数、字符和逻辑值等四种基本的数据类型。

前　言

计算机所能处理的数据种类包括文字、数字、声音、绘图、影像、动画等，种类繁多，但都可以视为是基于文字和数字两种数据类型的派生形式。C++ 有非常完整的数据类型，我们先将它们列表于下，以获得一个概括的认识，其细节将在本书中逐一说明：



3.1 整数和浮点数

整数

整数 (integer values) 是所有不具小数点的数值。例如：

```
6    -3728    0    +248571  -12     +0      005      4L
```

从上例可知整数的开头和结尾处可以有正负号和英文字母 l 及 L，但不能掺杂其它的符号。其中数值之后的 L 或 l 是表示长整数 (long integer)，具有较大的存储空间。下列为错误的整数写法：

```
6.  -3,728      0.0      248,571      -£12         $05
```

对于整数而言，预设的数据类型是 int，可以在数字之后加上 L 或 l 改设定成为 long int。当整数之前加上 0 (零)时，数值是以八进制的方式储存的，如果加上 0x，则是以十六进制的方式储存。整数后面加上 U 或 u，表示不加正负号的整数 (unsigned int)。例如:

```
48U          // unsigned int
75UL         // unsigned long
372L         // long
751          // long  (不建议使用，因为 1 和 l 不易区分)
012          // 12₈ = 10
0x12         // 12₁₆ = 18
```

我们可以使用下列程序 Int.cpp 实际操作，并使用输出数据流对象 cout 将执行结果显示出来，以检验上述语法。

```
    // Int.cpp
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  cout << "  48U   : " << 48U   << endl;
  cout << "  75UL  : " << 75UL  << endl;
  cout << "  372L  : " << 372L  << endl;
  cout << "   751  : " << 751   << endl;
  cout << "   012  : " << 012   << endl;
  cout << "  0x12  : " << 0x12  << endl;
}
```

当程序经由编译、执行后，可以从显示器上看到以下的结果:

```
 48U   :   48
 75UL  :   75
372L   :  372
 75l   :   75
 012   :   10
0x12   :   18
```

浮点数

浮点数 (floating point numbers)是带有小数点的数值，用以描述实数 (real numbers)。严格地说，由于计算机内部储存浮点数的位置是有限的，绝大部分情况对于实数的描述只是一个近似值，无法真正的描述诸如 π 或 $\sqrt{2}$ 这类无理数(irrational numbers)，或是 1/3 这类无穷位数的有理数 (rational numbers)。浮点数的例子如下：

```
36.0      -7.382   +3.92L   0.00   +4.0   3.7f   3.7F
```

浮点数必须有小数点，但是除了英文字母 f，F，l 或 L 外，不允许掺杂其它符号。数值之后的 f 或 F 代表 float (浮点数)，具有标准的存储空间；以 32 位为例，每个 float 数据占用 32 位大小。加上 l 或 L 则为 long double (长浮点数)。下列为错误的浮点数表示法：

```
6        3,728.0      $10.05        324P     365D
```

为了表示非常大或是非常小的数值，浮点数也有和“科学计数法”一样的指数表示法 (exponential notation)。如下表所示。注意，指数表示法的 e 为 exponent 的缩写，可以写为 E 或 e。

十进制表示法	科学表示法	指数表示法
2413.652	2.413652×10^3	2.413652E3 2.413652E+3 2.413652e3 2.413652e+3
-0.0000624	-6.24×10^{-5}	-6.24E-5 -6.24e-5

对于浮点数而言，预设的数据类型是 double，可以在数字之后加上 f 或 F 改设定成 float，加上 l 或 L 则为 long double。例如：

```
4.7              // double
48.0F            // float
48.0f            // float
3.75L            // long double
4.26e12          // double
4.26e+12L        // long double (+号可以省略)
4.26E-12         // double
```

我们可以使用下列程序 Float.cpp 实际操作，验证浮点数的相关语法，以及显示的形式。

```
// Float.cpp
#include <iostream>
using std::cout;
using std::endl;
// ----- 主程序 -------------------------
int main()
{
  cout << " 4.7      : " << 4.7       << endl;
  cout << " 48.0F    : " << 48.0F     << endl;
  cout << " 48.0f    : " << 48.0f     << endl;
  cout << " 3.75L    : " << 3.75L     << endl;
  cout << " 4.26e12  : " << 4.26e12   << endl;
  cout << " 4.26e+12L : " << 4.26e+12L << endl;
  cout << " 4.26E-12  : " << 4.26E-12  << endl;
}
```

程序执行结果

```
4.7       : 4.7
48.0F     : 48
48.0f     : 48
3.75L     : 3.75
4.26e12   : 4.26e+12
4.26e+12L : 4.26e+12
4.26E-12  : 4.26e-12
```

此为试读，需要完整PDF请访问：www.ertongbook.com

基本数值数据类型的实际取值范围

Borland C++ Compiler 所定义的基本数值数据类型的实际取值范围如表 3.1.1，此范围亦适用于其它 32 位的计算机系统：

表 3.1.1 基本数值数据类型的实际取值范围及有效位数

数据类型	数据长度	取值范围	小数点以下的有效位数
bool	8 bits	0 或 1	0
int	32 bits	从-2,147,483,648 至 2,147,483,647	0
short int short	16 bits	从-32,768 至 32,767	0
unsigned int unsigned	32 bits	从 0 至 4,294,967,295	0
long int long	32 bits	从-2,147,483,648 至 2,147,483,647	0
unsigned long	32 bits	从 0 至 4,294,967,295	0
float	32 bits	从 9.9×10^{-41} 至 3.402×10^{38}	7
		从 -3.402×10^{38} 至 -9.9×10^{-41}	
double	64 bits	从 9.9×10^{-320} 至 1.79×10^{308}	16
		从 -1.79×10^{308} 至 -9.9×10^{-320}	
long double	80 bits	(同 double)	16

在上表中，“取值范围”表示能够在计算机内部明显区分，并能正确表达的数值范围。以 float 数据类型为例，当数值 x 的范围为：

$$-9.9 \times 10^{-41} < x < 9.9 \times 10^{-41}$$

则以 0 储存及显示；当数值 x 的范围为：

$$x > 3.402 \times 10^{38}$$

则为 +INF，INF 为 infinity 的缩写，表示一个正无穷大的数字。如果 $x < -3.402 \times 10^{38}$ 则表示为 -INF，负无穷大。

实际上，我们很少有机会表达 $\pm 10^{30}$ 以外的数字；也不用区别 $\pm 10^{-30}$ 以内的数字。前者我们尽可以用 +INF 和 -INF 取代 (代表 $\pm \infty$)，表示数值已经发散或有错误，而后者则可以用 0 取代，表示数值已经收敛至零。以此观点，如果使用 double 而不用 float，通常是为了在计算的过程中有更多的有效位数，以获得更精确的计算结果，而不是为了表达更大范围的数值。此外，long double 使用了较大的数据长度，却没有获得更多的有效位数，因此目前没有实际的用途。而 unsigned long 和 unsigned 的效果完全一样，也很少使用。

3.2 变量和常量

在程序执行的时候，所有的数据都必须依序存放在主内存里。在这些数据里面，有些是允许变更其内容的，称之为变量 (variables)，不允许变更内容的则称为常量 (constants)。

C++是一种对数据的形态控制非常严谨的高级程序语言，所有的变量和常量都需要经由声明 (declaration) 才能取用。

■ 变量的声明和定义

声明一个变量意味着赋予它一个名称，并指定其所属的数据形态。其标准格式为数据类型之后加上变量名称，并以分号作为结尾。

例如

```
int     Age;
float   Height;
```

分别声明了一个叫做 Age 的整数和一个称为 Height 的浮点数。上面这两个语句同时也完成了 Age 和 Height 两个变量的定义 (definition)。

定义表示在内存内确实配置了足够容纳这个变量的存储空间。一般基本数据类型的变量之声明都同时完成了定义。唯一的例外是将在 6.4 节“变量的适用范围和生存期间”中介绍的“extern 全局变量”，不过这个用法即将淘汰。ANSI/ISO C++另外有了一个叫做“名称空间”(namespace) 的专用结构，用来规范文件间共享变量的问题 (我们将在第 16 章中详细介绍名称空间)。因此，对于变量和常量而言，声明通常也包含了定义。

除了关键词 int 用来指定整数的数据类型外，short 和 long 分别用来代表短的整数数据和长的整数数据，例如

```
long    int Ia;
long    Ib;
short   int Ic;
short   Id;
```

分别定义了长整数 Ia， Ib 和短整数 Ic，Id。我们也注意到，当 long 或 short 这两个修饰关键词使用时，int 可以忽略不写。

单精度的浮点数和双精确度的浮点数分别以 **float** 和 **double** 这两个关键词声明。例如

```
float Fa;              // Fa 为单精度的浮点数
double Db;             // Db 为双精确度的浮点数
```

分别声明了单精度浮点数 Fa 和双精确度的浮点数 Db。

程序 Average.cpp 是包括整数及浮点数的声明和运算的完整程序，并使用输出数据流对象 cout 将执行结果显示出来。

```
// Average.cpp
#include <iostream>
using std::cout;
using std::endl;

// -------主程序--------------------
int main()
{
    int Number = 3;
    float a, b;
    float c = 5.6;
    float Average;

    a = 7.8;    b=3.9;
    Average = (a + b + c) / Number;
```

```
        cout     << "a, b, c 的平均值是: "
<< Average << endl;
}
```

当程序经由编译、执行后，可以先从显示器上看到以下的结果:

```
a, b, c 的平均值是: 5.76667
```

讨 论

程序中的浮点数声明

```
float a, b;
```

称为多重声明 (multiple declarations)，相当于下列二式:

```
float a;
float b;
```

可同时完成多个相同数据类型的变量声明。

此外，声明语句

```
float c = 5.6;
```

则除了 float 类型的变量 c 的声明外，还将初始值 (initial value)5.6 给予变量 c。有些编译器会自动将变量的初始值设为 0，但并不是所有的编译器都会自动进行变量初始化的动作。我们必须确定所有的变量在运算前都能获得正确的初始值。

我们注意到，所有的变量，亦即 Number，a， b，c 和 Average，都必须先声明才能使用。变量的声明通常都集中放在用来标示段落范围的起始大括号 { 的后面，以统一管理。

变量声明以后，就会依所指定的数据类型分配内存，并将变量名称与该变量所在的内存的第一个字节的地址关联起来，如图 3.2.1 所示:

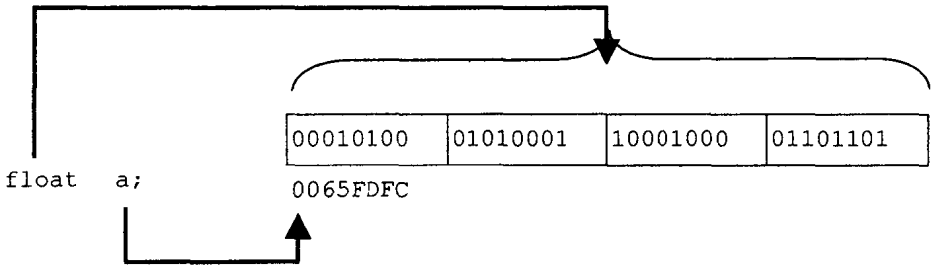


图 3.2.1 声明语句与内存空间的对应关系

在本例中，变量 a 的数据类型是 float，占用 4 个字节，而且 a 这个代号所关联的对象是内存中存放这些数据的第一个字节的地址。

常量的声明和定义

有些时候，某些特别的数值是不变的，但我们不想每一次都在程序中重复写出其数值，而只想使用代号。这时，就必须将该数值声明为常量 (constant)，并同时命名。使用常量代号有两个用途：

1. 避免重复直接写出数值，以减少输入的错误。
2. 可以只修改一处定义，就能同时改变程序中所有使用这个值的地方。

声明常量的关键词是 const (为 constant 的缩写)，如下列的例子：

```
const double TaxRate = 0.17;
const float  Inch2Meter = 0.024;
const int    Max = 1024;
```

注意，一旦某数值被声明为常量，则不能再是 lvalue (左值)，也就是说其内容已不能更改。譬如，如果我们在程序中接着上列声明写入下列的语句：

```
Max = 2048;    // 错误!
```

将造成错误，无法通过编译。

3.3　算术运算

整数和浮点数可以进行基本的加、减、乘、除四则运算。虽然不建议将这两种数据类型混合，当整数和浮点数写在同一个算式 (arithmetic expression) 里时，依然能够获得合理而且有规则可循的结果。

算术运算符

用于算术运算的运算符号，称为算术运算符 (arithmetic operators)，共有 7 个，如表 3.3.1 所列：

表 3.3.1　算术运算符

算术运算符	可进行的运算	算式范例	计算结果
+	加法	4 + 6	10
-	减法	7.83 - 6.2	1.63
*	乘法	2.5 * 6.7	16.75
/	除法	10.6 / 2.5	4.24
%	余数	18 % 4	2
++	递增	++a	a = a+1
--	递减	--a	a = a-1

表达式 (expression) 与 语句 (statement)

将常量、变量或函数以各种运算符联结起来的组合称为表达式 (expression)，例如下列各式：

```
x < 15.8
x = y + sin( z )
( x + y ) * 0.8 - z
```

所有的表达式都有一个相对应的值。以上面三个计算式而言，如果

```
x = 2.0;  y = 6.5; z = 2.6;
```

则上面三个表达式的计算结果分别为

```
1        7.0155        4.2
```

此为试读，需要完整PDF请访问：www.ertongbook.com

C++程序里面最小的可执行单元称为语句 (statement). 假如我们已经声明了

```
float a , b , c , x , y ;
a = 1.5 ;   b = 3.8 ;   c = 4.2 ;
```

则以下都是正确的语句：

```
x = a + b * sin( c ) ;
;
b++;
y = ( b > 10.0 ) ;  x = c + 5.0 ;
```

所有的语句都遵循以下三个规范：

1. 以分号当结尾.
   即使是单独一个分号也自成一个语句，表示不进行任何处理.
2. 可以任意放置，而且不管多长都可以任意断行. 甚至可以在同一行写上好几个语句. 例如：

   ```
   x = a + b * ( c + 6.5 ) +
       c * 105.8 - 4.0 ;
   ```

   与

   ```
   x = a + b * ( c + 6.5 ) + c * 105.8 - 4.0;
   ```

   是完全一样的. 为了提高程序的可读性，通常我们会将同一类型的语句上下之间靠左对齐.
3. 各个数值或变量名称与运算符 (operator) 之间的空格可有可无. 例如

   ```
   x=a+b;
   ```

   与

   ```
   x = a + b;
   ```

   是完全一样的. 不过，通常插入空格便于阅读.

   有等号的语句称为赋值语句 (assignment statement)，其标准格式为

   ```
   Value = expression;
   ```

   例如

   ```
   x = a + b;
   ```

   与一般数学上的等式不同，上式的处理分成两个步骤：

(1) 算出 expression 的值.

(2) 将此值放在 Value 里面，成为它的内容.

一元运算符和二元运算符

在 x * y 这个运算里，x 和 y 称为操作数 (operand)，“*”称为运算符 (operator). 作用于两个操作数之间的运算符称为二元运算符 (binary operator)，像 * / 和 % 这类的运算符都是. 如果只作用于一个操作数上，则称为一元运算符 (unary operator)，例如负号.

有些运算符号既是一元运算符也是二元运算符. 例如

```
-x
```

用来计算 x 的负值. 然而同样的符号在

```
x - y
```

中，却是用来计算 x 和 y 的差值. 我们将它视为两个不同的运算符.

算术运算的优先级和结合规则

C++所描述的算术运算与我们熟悉的“先乘除后加减”规则相吻合，也就是说

```
6 + 4 * 8 % 3 * 6 - 5
```

与

```
6 + ((4 * 8) % 3) * 6 - 5
```

的计算顺序是一样的. 严格的说，各运算符的优先级和结合关系由表 3.3.2 决定 (愈上面的运算符具有较高的优先级，会先被执行):

表 3.3.2　运算符的优先级和结合关系

算术运算符	结合规则	范例（下两行运算结果相同）	
- (负)	从右至左	- a * b	- (a * b)
* / %	从左至右	a * b / c	(a * b) / c
+ - (减法)	从左至右	a * b + c	(a * b) + c

在同一个语句内，当运算符的优先级相同时，则依照从左至右的顺序处理.

整数的除法

由于整数不包括小数点，因此其结果和一般的习惯又不尽相同。在C++中，整数相除时，其余数被直接舍弃。例如

```
1/2        得到 0
3/7        得到 0
15/2       得到 7
```

此外，C++提供了余数运算符(modulus operator)，%，可以用来计算余数。例如

```
1%2        得到 1
3%7        得到 3
15%2       得到 1
```

递增或递减的计算

在程序写作的时候，我们常常会进行数量的递增计算或递减计算．这类计数的处理通常写成

```
int Increment = 2;
int Num;
... 其它语句
Num = Num + Increment;  // 递增计算
```

当累进的数目为 1 时，C++提供了 ++ 和 -- 两个单元操作符；分别称为递增运算符(increment operator) 和递减运算符 (decrement operator)。

使用这两个运算符的语意如下：

表示式	递增或递减表示式
N = N + 1	N++ 或 ++N
N = N - 1	N-- 或 --N
M = M + 1	M++ 或 ++M
M = M - 1	M-- 或 --M

当++或--写在操作元 (operand，亦即上表的 M 和 N) 之前 (称为前缀，prefix) 或之后(称为后缀，postfix) 时，具有不同的意义。

例如

```
M = ++i;
```

相当于

```
i = i + 1;   M = i;
```

而

```
M = i++;
```

则相当于

```
M = i;        i = i + 1;
```

也就是说，运算符号写在操作元前面时，先进行递增的动作，再进行赋值 (assignment) 的处理，写在操作元后面时处理程序则倒过来．这个语法不仅较简洁，而且由于 CPU 内特殊的程序安排，可以用较快的速率计算．

因此，以下较复杂的语句

```
M = i++ * 5;
```

相当于

```
M = i * 5;  i = i + 1;
```

而

```
M = ++i * 5;
```

则相当于

```
i++;          M = i * 5;
```

我们也注意到，C++语言的名称“C++”相当于是“递增运算后的 C”，意味着“比 C 有更多内涵的程序语言”．

赋值运算符

除了加减法外，C++对于等号两边同时出现相同变量的算术运算语句，也提供了类似递增和递减运算符的简化语法：

```
+=   -=   *=   /=   %=
```

这些运算符统称为赋值运算符 (assignment operators)，它们不仅适用于整数，也适用于浮点数，而且同样可以提高计算效率．使用这些运算符的语义如下：

语　句	具有相同计算结果的语句(使用赋值操作符)
Num = Num + Inc;	Num += Inc;
x = x * TaxRage;	x *= TaxRate;
y = y / Ratio;	y /= Ratio;
M = M % N;	M %= N;

数据类型间的转换

在讨论前，我们先看一个简单的程序 Convert.cpp

```
// Convert.cpp
#include <iostream>
using std::cout;
using std::endl;

// -------主程序--------------------
int main()
{
  int a = 5;
  float x;
  x = a + 3.8;
  cout << "x 的值是: " << x << endl;
}
```

程序执行结果

```
x 的值是: 8.8
```

在这段程序里，牵涉到整数 a 和浮点数 3.8 之间的相加．C++为了避免损失计算精度，在计算 a + 3.8 时先将 a 的内容 5 晋升为 double 的数据类型，而得到这个语句的 rvalue．其后，由于 lvalue 的数据类型为 float，因此将 rvalue 从 double 转为 float，再存入 x 所代

表的变量中。当和 a 相加的数值是 float 时，则只将 a 的内容 5 晋升为 float．整理如下表：

表示式 (expression)	计算结果的数据类型
a + 3.8f	float
a + 3.8	double

上述两种数据类型间的转换是自动进行的，称为隐式数据类型转换 (implicit type conversions)。C++ 语句的计算依循以下的规则：

1. 如果混合不同的数据类型，则计算结果以最精确的数据类型为准。
2. 当所有的操作数都具有相同的数据类型，计算结果不作类型转换。
3. 赋值运算 (assignment operation) 依 lvalue 的数据类型储存。

一般而言，我们都尽量采取显式数据类型转换 (explicit type conversions)，以避免错误。C++提供两种显式数据类型转换的语法可供选择：

(数据类型) 变量名称;

数据类型 (变量名称);

以上述的程序为例，我们可以写成

```
x = float (a) + 3.8f ;
```

或是

```
x = (float) a + 3.8f ;
```

提示

rvalue 和 lvalue 分别为 right value (右值) 和 left value (左值) 的缩写，在程序语言的特性分析上占有很重要的地位。

变量的值或表达式的计算结果称为 rvalue ，因为它可以置于赋值操作符等号“=” 的右侧。rvalue 本身不可以被指定数值。

经由编译器规划过的记忆位置所储存的内容称为 lvalue ，因为它通常放在赋值操作符的左侧.lvalue 可以被指定存入某个数值.我们可以将赋值语句视为把 rvalue

此为试读,需要完整PDF请访问: www.ertongbook.com