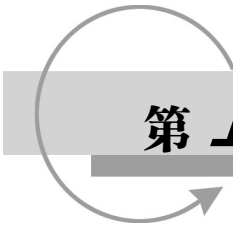
第1部分　实 用 教 程



第 1 章　基本 C++语言

要学习和应用 Visual C++，C++语言是基础。本章先来介绍基本 C++语言，第 2 章介绍 C++面向对象程序设计。第 3 章开始介绍 Visual C++程序设计，其中 MFC 是其核心。

1.1　C++程序结构

C++是在 20 世纪 80 年代初期由贝尔实验室设计的一种在 C 语言的基础上增加了支持面向对象程序设计的语言，它是目前应用最为广泛的编程语言。

1.1.1　C++概述

C++是在 C 语言基础上研制出来的一种通用的程序设计语言，它是在 1980 年由贝尔实验室的 Bjarne Stroustrup 创建的。研制 C++的一个重要目标是使 C++首先是一个更好的 C，所以 C++根除了 C 中存在的问题（如对数据类型检查机制比较弱，缺少支持代码重用的结构，是一种面向过程的编程语言等）。C++的另一个重要目标就是面向对象的程序设计，因此在 C++中引入了类的机制。最初的 C++被称为“带类的 C”，1983 年正式命名为 C++（C Plus Plus，C++）。以后经过不断完善，形成了目前的 C++。

为了使 C++具有良好的可移植性，1990 年，美国国家标准局（ANSI）设立了 ANSI X3J16 委员会，专门负责制定 C++标准。很快，国际标准化组织（ISO）也成立了自己的委员会（ISO-WG-21）。同年，ANSI 与 ISO 将两个委员会合并，统称为 ANSI/ISO，共同合作进行标准化工作。经过长达 9 年的努力，C++的国际标准（ISO/IEC）在 1998 年获得了 ISO、IEC（国际电工技术委员会）和 ANSI 的批准，这是第一个 C++的国际标准 ISO/IEC 14882:1998，常称为 C++ 98、标准 C++或 ANSI/ISO C++。2003 年，发布了 C++标准第二版（ISO/IEC 14882:2003）。本书以 ANSI/ISO C++内容为基础。

1.1.2　C++程序创建

使用 C++等高级语言编写的程序称为**源程序**。由于计算机只能识别和执行的是由 0 和 1 组成的二进制指令，称为**机器代码**，因而 C++源程序是不能被计算机直接执行的，必须转换成机器代码才能被计算机执行。这个转换过程就是编译器对源代码进行**编译**和**连接**的过程。如图 1.1 所示。

事实上，对于 C++程序的源代码编辑、编译和连接的步骤，许多 C++编程工具软件商都提供了各自的 C++集成开发环境（Integrated Development Environment，IDE）用于程序的一体化操作，常见的

有 Microsoft Visual C++、各种版本的 Borland C++（如 Turbo C++、C++ Builder 等）、IBM Visual Age C++和 Bloodshed 免费的 Dev-C++等。但 Visual C++在项目文件管理、调试以及操作的亲和力等方面都略胜一筹，从而成为目前使用极为广泛的基于 Windows 平台的可视化编程环境。如图 1.2 所示。

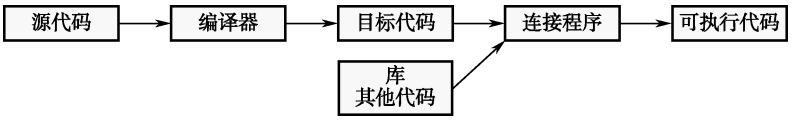


图 1.1　C++程序创建过程

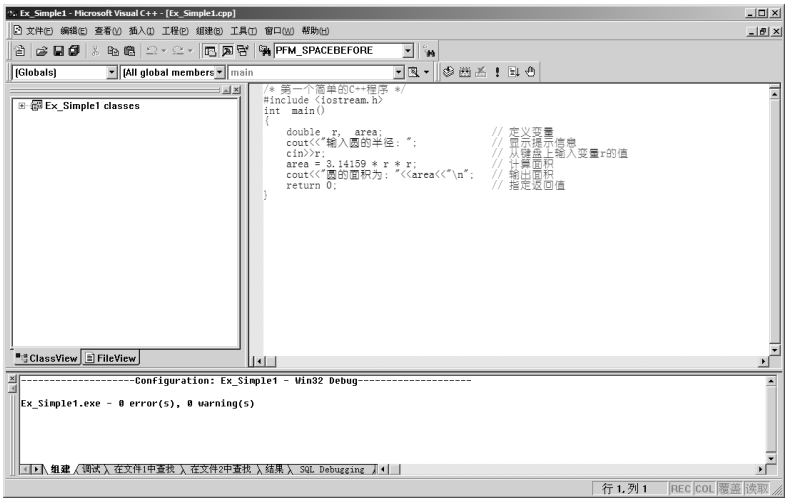


图 1.2　Visual C++6.0 SP6 中文版开发环境

Visual C++ 6.0 分为标准版、专业版和企业版三种，但其基本功能是相同的。Visual C++ 6.0 SP6 中文版是在 Visual C++ 6.0 SP6 基础上进行汉化的一个版本，本书以此版本作为编程环境。为统一起见，仍称之为 **Visual C++ 6.0***。

需要说明的是，由于 Visual C++对应用程序是采用文件夹的方式来管理的，即一个程序项目的所有源代码、编译的中间代码、连接的可执行文件等内容均放置在与程序项目名同名的文件夹中及其 debug（调试）或 release（发行）子文件夹中（以后还会讨论）。因此，在用 Visual C++进行应用程序开发时，一般先要创建一个工作文件夹，以便于集中管理和查找。下面以一个简单的 C++程序为例来说明在 Visual C++中创建和运行的一般过程。

1. 创建工作文件夹

创建 Visual C++ 6.0 的工作文件夹“D:\Visual C++程序”，以后所有创建的 C++程序都在此文件夹下。在文件夹“D:\ Visual C++程序”下再创建一个子文件夹“第 1 章”用于存放第 1 章中的 C++程序；对于第 2 章程序就存放在子文件夹“第 2 章”中，以此类推。

2. 启动 Visual C++ 6.0

选择“开始”→“程序”→“Microsoft Visual Studio 6.0” →“Microsoft Visual C++ 6.0”命令，运行 Visual C++ 6.0。第一次运行时，将显示如图 1.3 所示的“每日提示”对话框。单击 下一条(N) 按

* 对于使用 Win7 及之后版本的操作系统用户，最好用虚拟机（如 VMware）安装一份 Windows XP，在该环境中运行 Visual C++ 6.0。

钮，可看到有关各种操作的提示。如果取消对“启动时显示提示”复选框的选中，那么下一次运行 Visual C++ 6.0 时将不再出现此对话框。单击 关闭(C) 按钮关闭此对话框，进入 Visual C++ 6.0 开发环境。

图 1.3　“每日提示”对话框

3. 添加 C++程序

（1）单击标准工具栏 上的“新建”按钮，打开一个新的文档窗口，在这个窗口中输入下列 C++代码。

【例 Ex_Simple1】 一个简单的 C++程序

```
/* 第一个简单的 C++程序 */
#include <iostream.h>
int   main()
{
      double   r, area;                          // 定义变量 r,area 双精度整数类型
      cout<<"输入圆的半径：";                    // 显示提示信息
      cin>>r;                                    // 从键盘上输入的值存放到 r 中
      area = 3.14159 * r * r;                    // 计算圆面积，结果存放到 area 中
      cout<<"圆的面积为："<<area<<"\n";          // 输出结果
      return 0;                                  // 指定返回值
}
```

本书约定：书中凡是需要用户添加或修改的代码均用填充底纹来标明。

（2）选择“文件”→“保存”菜单或按快捷键【Ctrl+S】或单击标准工具栏中的按钮，弹出“保存为”文件对话框。将文件定位到“D:\Visual C++程序\第 1 章”文件夹中，文件名指定为“Ex_Simple1.cpp”（注意扩展名.cpp 不能省略）。

此时在文档窗口中所有代码的颜色都发生改变，这是 Visual C++ 6.0 的文本编辑器所具有的语法颜色功能，绿色表示注释（如//…），蓝色表示关键词（如 double）等。

4. 编译和运行

（1）单击编译工具条 上的“生成”按钮或直接按快捷键【F7】，系统弹出一个对话框，询问是否为该程序创建默认的活动工作区间文件夹，单击 是(Y) 按钮，系统开始对 Ex_Simple1 进行编译、连接，同时在输出窗口中显示编连信息，当出现“Ex_Simple1.exe - 0 error(s), 0 warning(s)”时表示 Ex_Simple1.exe 可执行文件已经正确无误地生成了。

（2）单击编译工具条 上的“运行”按钮或直接按快捷键【Ctrl+F5】，就可以运行刚刚生成的 Ex_Simple1.exe 了，结果弹出下面的窗口（其属性已被修改过，具体修改方法见实验 1），它称为控制台窗口，是一种为兼容传统 DOS 程序而设定的屏幕窗口：

此时等待用户输入一个数。当输入 10 并按【Enter】键后，控制台窗口显示为

其中，“Press any key to continue”是 Visual C++自动加上去的，表示 Ex_Simple1 运行后，按一个任意键将返回到 Visual C++开发环境，这就是 C++程序的创建、编连和运行过程。

本书约定：在以后的 C++程序运行结果中，本书不再完整显示其控制台窗口，也不再显示“Press any key to continue”，仅将控制台窗口中的运行结果列出。

1.1.3 C++代码结构

从上面的程序可以看出，一个 C++程序由编译预处理指令、数据或数据结构定义和若干个函数组成。在 C++中，一个程序可以存放在一个或多个文件中，这样的文件称为**源程序文件**。为了与其他文件相区别，每一个 C++源程序文件通常以.cpp 为扩展名。这里再以【例 Ex_Simple1】的程序代码来分析 C++程序的组成和结构。

1. main 函数

代码中，main 表示**主函数**，由于每一个程序执行时都必须从 main 开始，而不管该函数在整个程序中的具体位置，因此每一个 C++程序或由多个源文件组成的 C++项目都必须包含一个且只有一个 main 函数。

在 main 函数代码中，“int main()”称为 main 函数的**函数头**，函数头下面是用一对花括号“{”和“}”括起来的部分，称为 main 函数的**函数体**，函数体中包括若干条语句（按书写次序依次顺序执行），每一条语句都由分号“;”结束。由于 main 函数名的前面有一个 int，它表示 main 函数的类型是整型，须在函数体中使用关键字 return，用来将其后面的值作为函数的返回值。

2. 输入输出

main 函数体内的第 1 条语句是用来定义两个双精度实型（double）变量 r 和 area，第 2 条语句是一条输出语句，它将双引号中的内容（即字符串）输出到屏幕上，cout 表示标准输出流对象（屏幕），“<<”是插入符，它将后面的内容插入到 cout 中，即输出到屏幕上；第 3 条语句是一条输入语句，cin 表示标准输入流对象（键盘），“>>”是提取符，用来将用户键入的内容保存到后面的变量 r 中；“return 0;”之前的最后一条语句是采用多个“<<”将字符串和变量 area 的内容输出到屏幕中，后面的“\n”是换行符，即在内容输出后回车换行。

3. 预处理指令

#include <iostream.h>称为预处理指令（即编译之前进行的指令，以后还会讨论）。iostream.h 是 C++编译器自带的文件，称为 **C++库文件**，它定义了标准输入/输出流的相关数据及其操作。由于程序用到了输入/输出流对象 cin 和 cout，因而需要用#include 将其合并到程序中。又由于它们总是被放置在源程序文件的起始处，所以这些文件被称为**头文件**（Header File）。C++编译器自带了许多这样的头文件，每个头文件都支持一组特定的“功能”，用于实现基本输入输出、数值计算、字符串处理等方面的操作。

4. 注释

在 C++中，“/*...*/”之间的内容称为**块注释**，它可以出现在程序中的任何位置，包括在语句或表达式之间。而“//”只能实现单行的注释，它是将“//”开始一直到行尾的内容作为注释，称为**行注释**。注释的目的只是为了提高程序的可读性，对编译和运行并不起作用。正是因为这一点，所注释的内容既可以用汉字来表示（如 Ex_Simple1 中的注释），也可以用英文来说明，只要便于理解就行。

一般来说，注释应在编程的过程中同时进行，不要指望程序编制完成后再补写注释。那样只会多花好几倍的时间，更为严重的是，时间长了以后甚至会读不懂自己写的程序。通常，在源文件头部进行源程序总体注释，如版权说明、版本号、生成日期、作者、内容、功能、与其他文件的关系、修改日志等；若是头文件，注释中还应有函数功能简要说明。对于函数来说，其注释往往包括函数的目的/功能、输入参数、输出参数、返回值、调用关系（函数、表）等。当然，像全局变量的功能、取值范围等也属于注释内容，但千万不要陈述那些一目了然的内容，否则会使注释的效果适得其反。

5. 缩进

缩进是指程序在书写时不要将程序的每一行都由第一列开始，而是在适当的地方加进一些空格，和注释一样，也是为了提高程序的可读性。通常，在书写代码时，每个“}”花括号占一行，并与使用花括号的语句对齐。花括号内的语句采用缩进书写格式，缩进量为 4 个字符（一个默认的制表符）。

事实上，为了提高程序的可读性，还应注意对齐和分段（块）。所谓对齐，即同一层次的语句需从同一列开始，同一层次的左右花括号分开时应在同一列上。而分段（块）是指将程序代码根据其作用、功能和属性分成几个段落或几个块，段落或块之间添加一个或多个空行。

1.2　数据类型和基本输入/输出

程序的数据必须依附其内存空间方可操作，每个数据在内存中存储的格式以及所占的内存空间的大小取决于它的数据类型。在 C++中，数据可分为变量或常量两种，是贯穿整个程序的一种流，依托 C++流的独特机制可对流进行输入/输出操作。

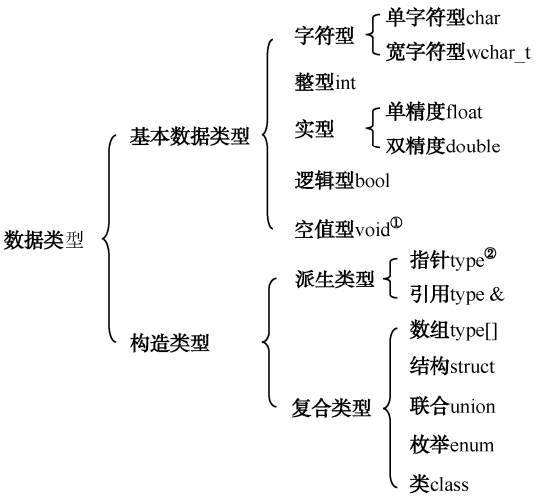
1.2.1　基本数据类型

为了能精确表征数据在计算机内存中的存储（格式及大小）和操作，C++将数据类型分为**基本数据类型**、派生类型和复合类型三类，后两种类型又可统称为**构造类型**，如图 1.4 所示。

基本数据类型是 C++系统内部预定义的数据类型；派生类型是将已有的数据类型定义成指针和引用；而复合类型是根据基本类型和派生类型定义的复杂数据类型，如数组、类、结构和联合等。这里先来介绍基本数据类型。

C++基本数据类型有 int（整型）、float（单精度实型）、double（双精度实型）、char（字符型）和 bool（布尔型，值为 false 或 true，而 false 用 0 表示，true 用 1 表示）等。

对于上述基本数据类型，还可以使用 short（短型）、long（长型）、signed（有符号）和 unsigned（无符号）来区分，以便更准确地适应各种情况的需要。表 1.1 列出了 C++各种基本数据的类型、字宽（以字节数为单位）和范围，它们是根据 ANSI 标准而定的。



注：①void是空值型，用于描述没有返回值的函数以及通用指针类型。
②图中的type是指任意一个C++合法的数据类型。

图 1.4　C++的数据类型

表 1.1　C++的基本数据类型

类 型 名	类 型 描 述	字　宽	范　围
bool	布尔型	1	false(0)或 true(1)
char	单字符型	1	−128～127
unsigned char	无符号字符型	1	0～255(0xff)
signed char	有符号字符型	1	−128～127
wchar_t	宽字符型	2	视系统而定
short [int]	短整型	2	−32768～32767
unsigned short [int]	无符号短整型	2	0～65535(0xffff)
signed short [int]	有符号短整型（与 short int 相同）	2	−32768～32767
Int	整型	4	−2147483648～2147483647
unsigned [int]	无符号整型	4	0～4294967295(0xffffffff)
signed [int]	有符号整型（与 int 相同）	4	−2147483648～2147483647
long [int]	长整型	4	−2147483648～2147483647
unsigned long [int]	无符号长整型	4	0～4294967295(0xffffffff)
signed long [int]	有符号长整型（与 long int 相同）	4	−2147483648～2147483647
float	单精度实型	4	7 位有效位
double	双精度实型	8	15 位有效位
long double	长双精度实型	10	19 位有效位，视系统而定

注：① 表中的字宽和范围是 32 位系统的结果，若在 16 位系统中，则 int、signed int、unsigned int 的字宽为 2 个字节，其余相同。

② 表中的[int]表示可以省略，即在 int 之前有 signed、unsigned、short、long 时，可以省略 int 关键字。

short 只能修饰 int，写成 short int，也可省略为 short。大多数计算机上，short int 表示 2 个字节长。

long 只能修饰 int 和 double。当为 long int 时，可略写成 long，一般表示 4 个字节。而 long double 一般表示 10 个字节。

signed（有符号）和 unsigned（无符号）只能修饰 char 和 int。实型 float 和 double 一般总是有符号的，因此不能用 unsigned 来修饰。char、short、int 和 long 可统称为整型，当没有任何修饰时，它们本身是有符号（signed）的。

1.2.2　字面常量

根据 C++程序中数据的可变性，可将数据分为常量和变量两大类。

在 C++程序运行过程中，其值始终保持不变的数据称为常量。常量可分字面常量和标识符常量两类。所谓字面常量，是指能直接从其字面形式即可判别其类型的常量，又称为直接量。如 1、20、0、−6 为整数，1.2、−3.5 为实数，‘a’、‘b’为字符，“C++语言”为字符串，等等。而标识符常量是用标识符来说明的常量，如 const 修饰的只读变量、#define 定义的常量及 enum 类型的枚举常量等。下面先来介绍常用类型的字面常量的表示方法。

1. 整数常量

C++中的**整数**可用十进制、八进制和十六进制来表示。其中，八进制整数是以数字 0 开头且由 0～7 的数字组成的数。如 045，即$(45)_8$，表示八进制数 45，等于十进制数 37；−023 表示八进制数−23，等于十进制数−19。而十六进制整数是以 0x 或 0X 开头且由 0～9、A～F 或 a～f 组成的数。如 0x7B，即$(7B)_{16}$，等于十进制的 123，−0X1a 等于十进制的−26。

需要说明的是，整数后面还可以有后缀 l、L、u、U 等。若以 L 或其小写字母 l 作为结尾的整数表示长整型（long）整数，如 78L、496l、0X23L、023l 等；以 U 或 u 作为结尾的整数表示无符号（unsigned）整数，如 2100U、6u、0X91U、023u 等；以 U（u）和 L（小写字母 l）的组合作为结尾的整数表示无符号长整型（unsigned long）整数，如 23UL、23ul、23LU、23lu、23Ul、23uL 等。若一个整数没有后缀，则可能是 int 或 long 类型，这取决于该整数的大小。

2. 实数常量

实数即**浮点数**，它有十进制数和指数两种表示形式。

十进制数形式是由整数部分和小数部分组成的（注意必须有小数点）。例如 0.12、.12、1.2、12.0、12.、0.0 都是合法的十进制数形式实数。

指数形式采用科学表示法，它能表示出很大或很小的实数。例如 1.2e9 或 1.2E9 都表示 1.2×10^9，注意字母 E（或 e）前必须有数字，且 E（或 e）后面的指数必须是整数。

需要说明的是，若实数是以 F（或 f）结尾的，如 1.2f，则表示单精度浮点数（float），以 L（或小写字母 l）结尾的，如 1.2L，表示长双精度浮点数（long double）。若一个实数没有任何后缀，则表示双精度浮点数（double）。

3. 字符常量

在 C++中，用单引号将其括起来的字符称为**字符常量**。如‘B’、‘b’、‘%’、‘␣’等都是合法的字符，但若只有一对单引号‘’则是不合法的，因为 C++不支持空字符常量。注意‘B’和‘b’是两个不同的字符。

本书约定：由于阅读时，书中的空格难以看出，故用符号␣表示一个空格。

除了上述形式的字符常量外，C++还可以用“\”开头的字符序列来表示特殊形式的字符。例如在以前程序中的‘\n’，它代表回车换行，即相当于按【Enter】键，而不是表示字母 n。这种将反斜杠（\）

后面的字符转换成另外意义的方法称为转义序列表示法。‘\n’称为**转义字符**，“\”称为转义字符引导符，单独使用没有任何意义，因此若要表示反斜杠字符，则应为‘\\’。表 1.2 列出了常见的转义序列符。

表 1.2　C++中常见转义序列符

字 符 形 式	含　义	ASCII 码值
\a	响铃（BEL）	07H
\b	退格（相当于按 Backspace 键）（BS）	08H
\f	进纸（仅对打印机有效）（FF）	0CH
\n	换行（相当于按 Enter 键）（CR、LF）	0DH、0AH
\r	回车（CR）	0DH
\t	水平制表（相当于按 Tab 键）（HT）	09H
\v	垂直制表（仅对打印机有效）（VT）	0BH
\'	单引号	27H
\"	双引号	22H
\\	反斜杠	5CH
\?	问号	3FH
\ooo	用 1 位、2 位或 3 位八制数表示的字符	$(ooo)_8$
\x*hh*	用 1 位或多位十六制数表示的字符	*hh*H

需要说明的是，当转义字符引导符后接数字时，用来指定字符的 ASCII 码值。默认时，数字为八进制，此时数字可以是 1 位、2 位或 3 位。若采用十六进制，则需在数字前面加上 **X** 或 **x**，此时数字可以是 1 位或多位。例如，‘\101’和‘\x41’都是表示字符‘A’。若为‘\0’，则表示 ASCII 码值为 0 的字符。

注意：

ANSI/ISO C++中由于允许出现多字节编码的字符，因此对于“\x”或“\X”后接的 16 进制的数字位数已不再限制。

不是每个以转义序列表示的字符都是有效的转义字符，当 C++无法识别时，就会将该转义字符解释为原来的字符。例如，‘\A’和‘\N’等虽都是合法的转义字符，但却都不能被 C++识别，此时‘\A’当作‘A’，‘\N’当作‘N’。

注意 0、‘0’和‘\0’的区别：0 表示整数，‘0’表示数字 0 字符。‘\0’表示 ASCII 码值为 0 的字符。

4. 字符串常量*

C++语言除了允许使用字符常量外，还允许使用字符串常量。字符串常量是由一对双引号括起来的字符序列，简称为字符串。字符串常量中除一般字符外，还可以包含空格、转义序列符或其他字符（如汉字）等。例如：

```
"Hello, World!\n"
```

*书中表示字符串的一对双引号“”是汉字字符，在程序代码中是没有的，它们均用"来表示；类似的，一对单引号‘’在程序代码中均用'表示。注意不要在程序代码中误用这些汉字字符。

```
"C++语言"
```

都是合法的字符串常量。字符串常量的字符个数称为字符串长度。若只有一对双引号“”，则这样的字符串常量的长度为 0，称为空字符串。

由于双引号是字符串的分界符，因此如果需要在字符串中出现双引号则必须用“\””表示。例如：

```
"Please press \"F1\" to help!"
```

这个字符串被解释为：

```
Please press "F1" to help!
```

字符串常量应尽量在同一行书写，若一行写不下，可用“\”来连接，例如：

```
"ABCD \
EFGHIJK…"
```

注意不要将字符常量和字符串常量混淆不清，它们主要的区别如下：

（1）字符常量是用单引号括起来的，仅占 1 个字节；而字符串常量是用双引号括起来的，至少需要 2 字节，但空字符串除外，它只需 1 个字节。例如，字符串“a”的字符个数为 1，即长度为 1，但它所需要的字节大小不是 1 而是 2，因为除了字符 a 需要 1 个字节外，字符串结束符‘\0’还需 1 个字节。如图 1.5 所示。

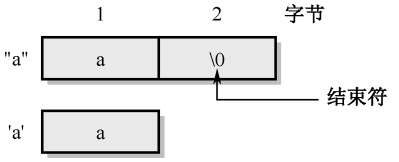


图 1.5　"a"和'a'的区别

（2）内存中，字符是以 ASCII 码值来存储的，因此可将字符看作整型常量的特殊形式，它可以参与常用的算术运算，而字符串常量则不能。例如：

```
int   b = 'a' + 3;                    // 结果 b 为 100，这是因为让'a'的 ASCII 码值 97 参与了运算
```

1.2.3　变量及其命名规则

变量是指在程序执行中其值可以改变的量。变量的作用是存取程序中需要处理的数据，它“对应”于某个内存空间。变量有 3 个基本要素：C++合法的变量名、变量的数据类型和变量的数值。

1. 变量名命名

变量名需用标识符来标识。所谓**标识符**，是用来标识变量名、函数名、数组名、类名、对象名等的有效字符序列。标识符命名的好坏直接影响程序的可读性，下面几个原则是命名时所必须注意的。

（1）合法性。C++规定标识符由大小写字母、数字字符（0～9）和下划线组成，且第一个字符必须为字母或下划线。任何标识符中都不能有空格、标点符号及其他字符，例如下面的标识符是**不合法**的：

93Salary，　Peter.Ding，　$178，　#5f68，　r<d

而且，用户定义的标识符不能和系统关键字同名。以下是 63 个 ANSI/ISO C++标准关键字：

asm	auto	bool	break	case	catch	char
class	const	const_cast	continue	default	delete	do
double	dynamic_cast	else	enum	explicit	export	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

需要说明的是，程序中定义的标识符除了不能与关键字同名外，还不能与系统库文件中预定义的标识符同名，如以前遇到的 cin、cout 等。最好也不要与这些关键字相似或只是大小写区别，如 Int、Long 等，虽然在 ANSI/ISO C++中是合法的，但在有的 C++系统中常将 Int 和 Long 看作 int 和 long 的别名，因此它们都是不好的标识符。再如，include、define 虽是合法的标识符，但它和预处理指令#include、#define 相似，也是不好的。

另外，C++中标识符的大小写是有区别的。例如，data、Data、DaTa、DATA 等都是不同的标识符。尽管如此，也不要将两个标识符定义成字母相同、大小写不同的标识符。

（2）有效性。因为有的编译器只能识别前 32 个字符，也就是说前 32 个字符相同的两个不同标识符被有的系统认为是同一个标识符。因此，虽然标识符的长度（组成标识符的字符个数）是任意的，但最好不能超过 32 个。

（3）易读性。在定义标识符时，若能做到“见名知意”就可以达到易读性的目的。实际上，许多程序员还采用“匈牙利标记法”来定义标识符（见附录 D），这种方法是在每个变量名前面加上表示数据类型的小写字符，变量名中每个单词的首字母均大写。例如，用 nWidth 或 iWidth（宽度）表示整型（int）变量。

2. 变量定义

C++中，定义变量的最简单的方法是先写数据类型，然后是变量名，数据类型和变量名之间必须用 1 个或多个空格来分隔，最后以分号来结尾，即如下列格式的语句：

```
<数据类型>  <变量名 1>[，<变量名 2>，…];
```

本书约定：凡格式中出现的尖括号“<>”，表示括号中的内容是必须指定的，若为方括号“[ ]”，则括号中的内容是可选的。

数据类型是告诉编译器要为由变量名指定的变量分配多少字节的内存空间，以及变量中要存取的是什么类型的数据。例如：

```
double    x;                    // 双精度实型变量
```

这样，x 占用了 8 个字节连续的内存空间，存取的数据类型是 double 型，称为**双精度实型变量**。再如：

```
float     y;                    // 单精度实型变量
```

则 y 占用了 4 个字节连续的内存空间，存取的数据类型是 float 型，称为**单精度实型变量**。此后，变量 x、y 就分别对应于各自的内存空间，换句话说，开辟的那块 8 字节的内存空间就叫 x，那块 4 字节的内存空间就叫 y。又如：

```
int   nNum1;                    // 整型变量
int   nNum2;                    // 整型变量
int   nNum3;                    // 整型变量
```

则 nNum1、nNum2、nNum3 分别占用 4 个字节的内存空间，其存取的数据类型是 int 型，称为**整型变量**。由于它们都是同一类型的变量，因此为了使代码简洁，可将同类型的变量定义在一行语句中，不过同类型的变量名要用逗号（,）分隔（逗号前后可以有 0 个或多个空格）。例如上述三个整型变量可这样定义（注意，只有最后一个变量 nNum3 的后面才有分号）：

```
int   nNum1,  nNum2,  nNum3;
```

需要说明的是，除了上述整型变量、实型变量外，还可有**字符型变量**，即用 char 定义的变量。这些都是最基本的数据类型变量。实际上，只要是合法的 C++数据类型，均可以用来定义变量。例如：

```
unsigned short    x,  y,  z;                    // 无符号短整型变量
long double       pi;                           // 长双精度实型变量
```

在 C++中没有基本数据类型的字符串变量。**字符串变量**是用字符类型的数组、指针或 string 类来定义的（以后会讨论）。

在同一个**作用域**（以后会讨论）中，不能对同一个变量重新定义。或者说，在同一个作用域中，不能有两个或两个以上的变量名相同。例如：

```
float       x,  y,  z;                          // 单精度实型变量
int         x;                                  // 错误，变量 x 重复定义
float       y;                                  // 错误，变量 y 重复定义
```

C++变量满足即用即定义的编程习惯，也就是说，变量定义的位置可以不固定，比较自由，但一定要遵循先定义后使用的原则。例如：

```
int         x;                                  // 即用即定义
x = 8;
int         y;
cout<<z<<endl;                                  // 错误，z 还没有定义
```

3. 变量赋值和初始化

变量一旦定义后，就可以通过变量名来引用变量进行赋值等操作。所谓**引用变量**，就是使用变量名对其内存空间进行操作。例如：

```
int   x,  y;
x = 8;                                          // 给 x 赋值
y = x;                                          // 将 x 的值赋给 y
```

“x = 8;”和“y = x;”都是变量的赋值操作，“=”是赋值运算符（以后还会讨论）。由于变量名 x 和 y 标识它们的内存空间，因此，“x = 8;”是将“=”右边的数据 8 存储到左边变量 x 的内存空间中。而“y = x;”这个操作则包括两个过程：先获取 x 的内存空间中存储的值（此时为 8），然后将该值存储到 y 的内存空间中。

当首次引用一个变量时，变量必须有一个确定的值，这个值就是变量的**初值**。在 C++中，可用下列形式或方法给变量赋初值。

（1）在变量定义后，使用赋值语句来赋初值。如前面的“x = 8;”和“y = x;”，使 x 和 y 的初值都设为 8。

（2）在变量定义的同时赋给变量初值，这个过程称为**变量初始化**。例如：

```
int         nNum1 = 3;                          // 指定 nNum1 为整型变量，初值为 3
double      x = 1.28;                           // 指定 x 为双精度实变量，初值为 1.28
char        c = 'G';                            // 指定 c 为字符变量，初值为'G'
```

（3）也可以在多个变量的定义语句中单独对某个变量进行初始化，如：

```
int   nNum1,  nNum2 = 3,  nNum3;
```

表示 nNum1、nNum2、nNum3 为整型变量，但只有 nNum2 的初值为 3。

（4）在 C++中，变量的初始化还有另外一种形式，例如：

```
int   nX(1),  nY(3),  nZ;
```

表示 nX、nY 和 nZ 都是整型变量，其中紧随 nX 和 nY 后面的括号中的数值 1 和 3 分别为 nX 和 nY 的初值。

注意：

一个没有初值的变量并不表示它所在的内存空间没有数值，而是取决于编译为其开辟内存空间时的处理方式，它可能是系统默认值或者该内存空间以前操作后留下来的无效值。

1.2.4 标识符常量和枚举

标识符常量又称为**符号常量**，它是用一个标识符来代替一个数值。在程序中使用标识符常量不仅可以提高程序的可读性，且修改方便，并能预防程序出错。例如，整个程序的许多地方都要用一个常数 π，每次输入时都得写上 3.14159，如果在某些地方写错这个值，则会导致计算结果的错误。如果给这个 π 取一个名字 PI（标识符），每处需要的地方都用 PI 来代替，不仅输入方便，而且即便写错了，编译器一般还能检查出来。如果 π 值不需要太精确的话，那么只要修改 PI 的值（如 3.1416）即可。事实上，布尔常量中的 false 和 true 就是系统内部预定义的两个标识符常量，即用 false 代替 0，用 true 代替 1。

与变量相似，标识符常量在使用前同样需要先作**声明***。在 C++中，标识符常量可以有 const 修饰的只读变量、enum 类型的枚举常量及#define 定义的常量 3 种形式。这里先介绍前面两种。

1. const 只读变量

在变量定义时，可以使用关键字 const 来修饰，这样的变量是只读的，即在程序中对其只能读取不能修改。由于不可修改，因而它是一个标识符常量，且在定义时必须初始化。需要说明的是，通常将标识符常量中的标识符写成大写字母以与其他标识符相区别。例如：

```
const       float   PI = 3.14159265f;              // 指定后缀 f 使其类型相同，否则会有警告错误
```

因 π 字符不能作为 C++的标识符，因此这里用 PI 来表示。PI 被定义成一个 float 类型的只读变量，由于 float 变量只能存储 7 位有效位精度的实数，因此 PI 的实际值为 **3.141592**。若将 PI 定义成 double，则全部接受上述数字。事实上，const 还可放在类型名之后（它们的区别以后会讨论），如下列语句：

```
double      const   PI = 3.14159265;
```

这样，就可在程序中使用 PI 这个标识符常量来代替 3.14159265 了。

【例 Ex_PI】 用 const 定义标识符常量

```
#include <iostream.h>
const   double   PI = 3.14159265;                   // PI 是一个只读变量
int   main()
{
      double r = 100.0, area;
      area = PI * r * r;                            // 引用 PI
      cout<<"圆的面积是："<<area<< "\n";
      return 0;                                     // 指定返回值
}
```

程序运行结果如下：

```
圆的面积是：31415.9
```

需要说明的是，由于 const 标识符常量的值不能修改，因此下列语句是错误的：

```
const   float   PI;                                 // 此时 PI 的值无法确定
PI = 3.14159265;                                    // 错误：只读变量不能放在赋值运算符的左边
```

*所谓**声明**，就是告诉编译器，有某类型的变量会被使用，但编译器往往并不会为它分配任何内存空间。而**定义**不同，它是要分配内存空间的。

2. 枚举常量

枚举常量是在由关键字 enum 指定的枚举类型中定义的。枚举类型属于构造类型，它是一系列有标识符的整型常量的集合，因此每一个枚举常量实质上就是一个整型标识符常量。

定义时，先写关键字 enum，然后是要定义的枚举类型名、一对花括号（{}），最后以分号结尾。enum 和类型名之间至少要有一个空格，花括号里面是指定的各个枚举常量名，各枚举常量名之间要用逗号分隔。即如下列格式：

```
enum  <枚举类型名> {<枚举常量 1, 枚举常量 2, …>};
```

例如：

```
enum  COLORS { Black,  Red,  Green,  Blue,  White };
```

其中 COLORS 是要定义的枚举类型名，通常将枚举类型名写成大写字母以与其他标识符相区别。它有 5 个枚举常量（又称为**枚举值**、**枚举元素**），系统默认为每一个枚举常量对应一个整数，并从 0 开始，逐个增 1，也就是说枚举常量 Black 等于 0，Red 等于 1，Green 等于 2，以此类推。

当然，这些枚举常量默认的值可单独重新指定，也可部分指定，例如：

```
enum  COLORS { Black = 5,  Red,  Green = 3,  Blue,  White = 7 };
```

由于 Red 没有赋值，则其值自动为前一个枚举常量值增 1，即为 6。同样，Blue 为 4，这样各枚举常量的值依次为 5，6，3，4，7。以后就可直接使用这些枚举常量了，例如：

```
int n = Red;                          // n 的初值为 6
cout << Blue + White <<endl;          // 输出 11
```

事实上，在枚举定义时可不指定枚举类型名。例如：

```
enum { Black = 5,  Red,  Green = 3,  Blue,  White = 7 };
```

显然，用 enum 一次可以定义多个标识符常量，不像 const 和#define 每次只能定义一个。又如，若在程序中使用 TRUE 表示 true，FALSE 表示 false，则可定义为：

```
enum  { FALSE,  TRUE };               // 或 enum  { TRUE = true,  FALSE = false };
```

1.2.5　基本输入/输出

通过前面程序的学习，已经对 C++的标准输入流 cin 和标准输出流 cout 有所了解。所谓“流”是从数据的传输（流动）抽象而来的，可以把它理解成“特殊的文件”；从操作系统的角度来说，每一个与主机相联的输入输出设备都可以看作一个文件。例如，终端键盘是输入文件（输入流），显示器和打印机是输出文件（输出流）。

cin 和 cout 是 C++预定义的流对象，分别代表标准输入设备（键盘）和标准输出设备（显示器）。这里将进一步介绍用 cin 和 cout 进行输入/输出的方法。

1. 输入流（cin）

cin 可以获得多个键盘的输入值，它具有下列格式：

```
cin>> <变量 1 > [>> <变量 2>…];
```

其中，提取运算符“>>”可以连续写多个，每个提取运算符后面跟一个获得输入值的变量。例如：

```
int  nNum1, nNum2, nNum3;
cin>>nNum1>>nNum2>>nNum3;
```

要求从键盘上输入三个整数。输入时，必须在 3 个数值之间加上一些空格来分隔，空格个数不限，最后用回车键结束输入；或者在每个数值之后按回车键。例如，上述输入语句执行时，可以输入：

```
12␣9␣20↵
```

或

```
12↵
```

```
9↵
20↵
```

本书约定：书中出现的“↵” 表示输入一个回车键。

此后变量 nNum1、nNum2 和 nNum3 的值分别为 12、9 和 20。需要说明的是，提取运算符“>>”能自动将 cin 输入值转换成相应变量的数据类型，但从键盘输入数据的个数、数据类型及顺序，必须与 cin 中列举的变量一一匹配。如：

```
char        c;
int         i;
float       f;
long        l;
cin>> c >> i >> f >> l;
```

上述语句运行后，若输入：

```
1␣2␣9␣20.↵
```

则变量 c 等于字符‘1’，i 等于 2，f 等于 9.0f，l 等于 20L。要注意输入字符时，不能像字符常量那样输入‘1’，而是直接输入字符，否则不会有正确的结果。例如，当输入：

```
'1'␣2␣9␣20.↵
```

由于 c 是字符型变量，占一个字节，故无论输入的字符后面是否有空格，c 总是等于输入的第 1 个字符，即为一个单引号。此后，i 就等于“1'”，由于 i 需要输入的是一个整数，而此时的输入值有一个单引号，因而产生错误，但单引号前面有一个“1”，于是就将 1 提取给 i，故 i 的值为 1。一旦产生错误，输入语句运行中断，后面的输入就变为无效，因此 f 和 l 都不会有正确的值（对于流的错误处理以后会讨论）。

2. 输出流(cout)

与 cin 相对应，通过 cout 可以输出一个整数、实数、字符及字符串等，如下列格式：

```
cout<< <对象 1> [<< <对象 2> ...];
```

cout 中的插入运算符“<<”可以连续写多个，每个后面可以跟一个要输出的常量、变量、转义序列符及表达式等，例如：

【例 Ex_Cout】 cout 的输出及 endl 算子

```
#include <iostream.h>
int   main()
{
      cout<<"ABCD\t"<<1234<<"\t"<<endl;
      return 0;                                          // 指定返回值
}
```

执行该程序，结果如下：

```
ABCD      1234
```

程序中，转义字符‘\t’是制表符，endl 是 C++中控制输出流的一个操作算子（预定义对象），它的作用和‘\n’等价，都是结束当前行并另起一行。

3. 使用格式算子 oct、dec 和 hex

格式算子 oct、dec 和 hex 能分别将输入或输出的整数转换成八进制、十进制及十六进制。

【例 Ex_ODH】 格式算子的使用

```
#include <iostream.h>
int   main()
{
      int    nNum;
```

```
    cout<<"Please input a Hex integer:";
    cin>>hex>>nNum;
    cout<<"Oct\t"<<oct<<nNum<<endl;
    cout<<"Dec\t"<<dec<<nNum<<endl;
    cout<<"Hex\t"<<hex<<nNum<<endl;
    return 0;
}
```

程序运行后，其结果如下：

```
Please input a Hex integer:7b↵
Oct     173
Dec     123
Hex     7b
```

综上所述，在外观上，提取运算符“>>”和插入运算符“<<”好比是一个箭头，它表示流的方向。显然，将数据从 cin 流入到一个变量时，则流的方向一定指向变量，即如“cin…>>a”格式；而将数据流入到 cout 时，则流的方向一定指向 cout，即如“cout<<…”格式。

1.3　运算符和表达式

和其他程序设计语言一样，C++记述运算的符号称为“运算符”，运算符的运算对象称为“操作数”。对一个操作数运算的运算符称为“单目运算符”或称为“一元运算符”，如-a、--i；对两个操作数运算的运算符称为“双目（二元）运算符”，如 3+5；对三个操作数运算的运算符称为“三目（三元）运算符”，如 x?a:b。而表达式是由变量、常量等操作数通过一个或多个运算符组合而成的，一个合法的 C++表达式经过运算应有一个确定的值和类型。本节重点介绍 C++的常用运算符及其表达式。

1.3.1　算术运算符

数学中，算术运算包括加、减、乘、除、乘方及开方等。在 C++中，算术运算符可以实现这些数学运算。但乘方和开方没有专门的运算符，它们一般通过 pow（幂）、sqrt（平方根）等库函数来实现，这些库函数是在头文件 math.h 中定义的（见附录 C）。

由操作数和算术运算符构成的**算术表达式**常用于数值运算，与数学中的代数表达式相对应。C++算术运算符有双目的加减乘除四则运算符、求余运算符及单目的正负运算符，如下所示：

```
+       正号运算符，如+4，+1.22 等
-       负号运算符，如-4，-1.22 等
*       乘法运算符，如 6*8，1.4*3.56 等
/       除法运算符，如 6/8，1.4/3.56 等
%       模运算符或求余运算符，如 40%11 等
+       加法运算符，如 6+8，1.4+3.56 等
-       减法运算符，如 6-8，1.4-3.56 等
```

在算术表达式中，C++算术运算符和数学运算的概念及运算方法是一致的，但要注意以下几点。

（1）除法运算。两个整数相除，结果为整数，如 7/5 的结果为 1，它是将小数部分去掉，而不是四舍五入。若除数和被除数中有一个是实数，则进行实数除法，结果是实型。如 7/5.0、7.0/5、7.0/5.0 的结果都是 1.4。

（2）求余运算。求余运算要求参与运算的两个操作数都是整型，其结果是两个数相除的余数。例如 40%5 的结果是 0，40%11 的结果是 7。要理解负值的求余运算，例如 40%-11 的结果是 7，-40%11 的结果是-7，-40%-11 的结果也是-7。

（3）优先级和结合性。在算术表达式中，先乘、除后加、减的运算规则是由运算符的优先级来保

证的。其中，单目的正负运算符的优先级最高，其次是乘、除和求余，最后是加、减。优先级相同的运算符，则按它们的结合性进行处理。所谓运算符的结合性是指运算符和操作数的结合方式，它有“从左至右”和“从右至左”两种。“从左至右”的结合是指运算符左边的操作数先与运算符相结合，再与运算符右边的操作数进行运算；而“自右至左”的结合次序刚好相反，它是将运算符右边的操作数先与运算符相结合。在算术运算符中，除单目运算符外，其余运算符的结合性都是从左至右。要注意，只有当两个同级运算符共用一个操作数时，结合性才会起作用。例如 2*3 + 4*5，则 2*3 和 4*5 不会按其结合性来运算，究竟是先计算 2*3 还是 4*5 由编译器来决定。若有 2*3*4，则因为两个“*”运算符共用一个操作数 3，因此按其结合性来运算，即先计算 2*3，然后再与 4 进行“*”运算。

（4）关于书写格式。在使用运算符进行数值运算时，往往要在双目运算符的两边加上一些空格，否则编译器会做出与自己理解完全不同的结果。例如：

```
-5*-6--7
```

和

```
-5␣*␣-6␣-␣-7                                    // 注意空格
```

结果是不一样的。前者发生编译错误，而后果的结果是 37。但对于**单目运算符**来说，虽然也可以与操作数之间存在空格，但最好与操作数写在一起。事实上，在书写 C++表达式时，应尽可能地有意识地加上一些圆括号。这不仅能增强程序的可读性，而且尤其当对优先关系犹豫时，加上括号是保证正确结果的最好方法，因为括号运算符“( )”的优先级几乎是最高的。

1.3.2 赋值运算符

前面已经多次遇到过赋值操作的示例。在 C++中，赋值运算是使用赋值符“=”来操作的，它是一个使用最多的双目运算符。这里就左值和右值、数值溢出、复合赋值、多重赋值以及赋值过程中的运算次序等几个方面的内容来讨论。

1. 左值和右值

赋值运算符“=”的结合性从右到左，其作用是将赋值符右边操作数的值存储到左边的操作数所在的内存空间中，显然，左边的操作数应是一个**左值**。

所谓左值（L-Value, L 是 Left 的首字母），即出现在赋值运算符左边（Left）的操作数，但它还必须满足两个条件：一是必须对应于一块内存空间，二是所对应的内存空间中的内容必须可以改变，也就是说左值的值必须可以改变。

这就是说，用 const 定义的只读变量，虽有一个确定的内存空间，但它的值不能被改变，因此 const 变量不能作为左值。同样，字面常量等由于不能被修改，因此也不能作为左值。大多数表达式如“a+b”，由于不能明确对应于一块内存空间，因而也不能成为左值。

与左值相对的是**右值**，即出现在赋值运算符右边的操作数，它可以是函数、常量、变量以及表达式等，但右边的操作数必须有具体的值且可以进行取值操作。

2. 数值截取和数值溢出

每一个合法的表达式在求值后都有一个确定的值和类型。对于赋值表达式来说，其值和类型就是左值的值和类型。例如：

```
float   fTemp;
fTemp = 18;                                      // fTemp 是左值，整数 18 是右值
```

对实型变量 fTemp 的赋值表达式“fTemp = 18”完成后，该赋值表达式的类型是左值 fTemp 的类型 float，表达式的值经类型自动转换后变成 18.0f，即左值 fTemp 的值和类型。

显然，在赋值表达式中，当右值的数据类型低于左值的数据类型时，C++会自动进行数据类型的转换。但若右值的数据类型高于左值的数据类型，且不超过左值的范围时，则 C++会自动进行**数值截取**。例如，若有“fTemp = 18.0”，因为常量 18.0 默认时是 double 型，高于 fTemp 指定的 float 型，但 18.0 没有超出 float 型数值范围。因而此时编译后，会出现警告，但不会影响 fTemp 结果的正确性。

但如果一个数值超出一个数据类型所表示的数据范围时，则会出现**数值溢出**。数值溢出的一个典型的特例是当某数除以 0，这种严重情况编译器将报告错误并终止程序运行。而超出一个数据类型所表示的数据范围的溢出在编译时往往不会显示错误信息，也不会引起程序终止，因此在编程时需要特别小心。

【例 Ex_OverFlow】　一个整数溢出的例子

```
#include <iostream.h>
int   main()
{
      short   nTotal, nNum1, nNum2;
      nNum1 = nNum2 = 1000;
      nTotal = nNum1*nNum2;
      cout<<nTotal<<"\n";
      return 0;
}
```

程序运行后，显示的结果是 16960。　　这个结果与想象中的 1000000 相差太远。这是因为，任何变量的值在计算机内部都是以二进制存储的，nNum1*nNum2 的 1000000 结果很显然超过了短整型数的最大值 32767，将 1000000 放入 nTotal 中，就必然产生高位溢出，也就是说，1000000 的二进制数 $(11110100001001000000)_2$ 中只有后面 16 位的 $(0100001001000000)_2$ 有效，结果是十进制的 16960。这个问题可以通过改变变量的类型来解决，例如将 nTotal 类型定义成整型（int）或长整型（long）。

3. 复合赋值

在 C++中，规定了下列 10 种复合赋值运算符：

+=	加赋值	&=	位与赋值
-=	减赋值	|=	位或赋值
*=	乘赋值	^=	位异或赋值
/=	除赋值	<<=	左移位赋值
%=	求余赋值	>>=	右移位赋值

它们都是在赋值符“=”之前加上其他运算符而构成的，其中算术复合赋值运算符的含义如表 1.3 所示，其他复合赋值运算符的含义均与其相似。

表 1.3　复合赋值运算符

运　算　符	含　　义	例　　子	等 效 表 示
+=	加赋值	a += b	a = a + b
-=	减赋值	a -= b	a = a - b
*=	乘赋值	a *= b	a = a * b
/=	除赋值	a /= b	a = a / b
%=	求余赋值	nNum %= 8	nNum = nNum % 8

尽管复合赋值运算符看起来有些古怪，但它却能简化代码，使程序精练，更主要的是在编译时能产生高效的执行代码。需要说明的是，在复合赋值运算符之间不能有空格，例如 += 不能写成 +␣= ，

否则编译时将提示出错信息。复合赋值运算符的优先级和赋值符“=”的优先级一样，在 C++的所有运算符中只高于逗号运算符，而且复合赋值运算符的结合性也和赋值符“=”一样，也是从右至左。因此，在组成复杂的表达式时要特别小心。例如：

```
a *= b - 4/c + d
```

等效于

```
a = a * ( b - 4/c + d)
```

而不等效于

```
a = a * b - 4/c + d
```

4. 多重赋值

所谓**多重赋值**是指在一个赋值表达式中出现两个或更多的赋值符“=”，例如：

```
nNum1 = nNum2 = nNum3 = 100                    // 若结尾有分号“；”，则表示是一条语句
```

由于赋值符的结合性是从右至左的，因此上述的赋值是这样的过程：首先对赋值表达式 nNum3 = 100 求值，即将 100 赋值给 nNum3，同时该赋值表达式的结果是其左值 nNum3，值为 100。然后将 nNum3 的值赋给 nNum2，这是第二个赋值表达式，该赋值表达式的结果是其左值 nNum2，值也为 100。最后将 nNum2 的值赋给 nNum1，整个表达式的结果是左值 nNum1。

由于赋值是一个表达式，因而几乎可以出现在程序的任何地方，由于赋值运算符的等级较低，因此这时的赋值表达式两边应加上圆括号。例如：

```
a = 7 + (b = 8)                    // 赋值表达式值为 15，a 值为 15，b 值为 8
a = (c = 7) + ( b = 8)             // 赋值表达式值为 15，a 值为 15，c 值为 7，b 值为 8
( a = 6 ) = (c = 7) + ( b = 8)     // 赋值表达式值为 15，a 值为 15，c 值为 7，b 值为 8
```

要注意上面最后一个表达式的运算次序：由于圆括号运算符的优先级在该表达式中是最高的，因此先运算(a = 6)、(c = 7)和(b = 8)，究竟这 3 个表达式谁先运算，取决于编译器。由于这三个表达式都是赋值表达式，其结果分别为它们的左值 a、c 和 b，因此整个表达式等效于 a = c + b，结果为 a=15、b=8、c=7，整个表达式的结果是左值 a。

需要说明的是，当赋值表达式出现在 cout 中时，需将赋值表达式两边加上圆括号。例如：

```
cout<<(a=6)<<endl;                 // 输出结果为 6
cout<<(a=6) + (b=5)<<endl;         // 输出结果为 11
```

1.3.3 数据类型转换

在进行表达式运算时，往往要遇到混合数据类型的运算问题。例如一个整数和一个实数相加就是一个混合数据类型的运算。C++采用两种方法对其数据类型进行转换，一种是自动转换，另一种是强制转换。

1. 自动转换

自动转换是将数据类型按从低到高的顺序自动进行转换，如图 1.6 所示，箭头的方向表示转换的方向。由于这种转换不会丢失有效的数据位，因而是安全的。

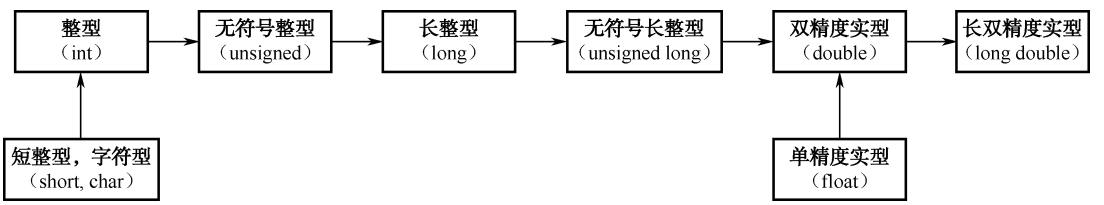


图 1.6 数据类型转换的顺序