第 3 章　C/C++语言基础

在第 1 章和第 2 章中介绍了 Visual Studio 2010 的开发环境和基本应用程序的创建。在 Visual Studio 2010 中创建了应用程序后，就需要了解 C++语言的语法和规则。只有深入了解语法规则及语法细节，才能开发出正确高效的程序。本章将详细讲述 Visual C++ 2010 的开发语言——C/C++的语言基础。

3.1　对标准 C 的扩展——C++

每种开发语言都有自己规定的结构和语法，只有编写的程序的结构和语法符合规定，相应的编译器才能正确处理。实质上，C 语言的编写就是数据定义和函数调用的组合。根据数据的特性，C 语言支持多种数据类型的定义，而对数据的操作则在函数调用中完成。程序入口是 main()函数，在 main()函数中调用其他功能函数。因此，C 语言是面向过程的开发语言。

C++是从 C 语言基础上发展而来的面向对象的编程语言，是对 C 语言的扩展，在保留了 C 语言的基本风貌的基础上，修正了 C 语言的弊端。C++语言主要在以下几个方面对 C 语言进行了扩展。

- C++语言的语法并不是全新的，这为原来的 C 语言开发人员从面向过程的开发语言过渡到面向对象的开发语言，提供了一个快速的转型过程。已有的 C 代码在 C++环境中仍然可以使用，只需要使用 C++编译器重新编译，并修正本来隐藏的错误就可以了。
- C++语言是更完善的 C 语言。C++语言是对 C 语言的扩展，不仅保留了良好的 C 语言习惯，并且修正了部分 C 语言的漏洞。如 C++语言对函数的声明做了强制规定，使得编译器可以检查函数的调用，减少错误发生的可能；C++语言加入了引用技术，使得函数调用者可以处理函数参数和返回的地址；C++语言引入了函数重载技术，使不同函数可以使用相同的函数名；C++语言引入了对命名空间的支持，扩大了函数的定义范围；并且提供了更完善的类型检查和编译时处理等。
- C++语言与 C 语言的运行效率基本一样。据不完全统计，相同条件下，使用 C++语言编写的面向对象的程序效率与 C 语言编写的程序相差在±10%左右。而且 C++语言的一些性能还可以调整程序的运行效率。
- C++语言是面向对象的，C 语言是面向过程的。因此，C++语言是用问题空间的概念描述问题的解决方法，而 C 语言是用解空间的概念描述问题的解决方法。所以，C++语言编写的程序比 C 语言编写的程序更容易理解。容易理解带来的好处就是

易于维护。通常维护工作是占用系统开销比较大的部分，因此 C++语言编写的程序的维护开销要比 C 语言编写的程序的维护开销要小。

- C++语言扩展了 C 语言对库的支持。使用库复用已有的代码可以大大提高开发效率，因此 C++语言也对 C 语言库的支持做了升级，它将库转换为类，当程序引入一个库，便向程序中引入一个新类，使得程序原有代码与引入的库浑然一体，风格一致，从而使得开发人员对库的使用更方便。
- C++语言引入了异常处理。这一点是对 C 语言的补充，因为 C 语言基本没有错误处理机制，C 程序对错误的处理，全靠开发人员自己实现。C++语言引入了异常处理，减少了开发人员对错误处理的程序的编写，并且增强了程序的健壮性。
- C++语言对复杂程序的支持比 C 语言要好。当程序非常复杂时，用于处理的变量和函数会非常多，比较容易发生命名冲突。因此，C++语言引入了命名空间机制，有了命名空间的限制，使用的变量和函数就可以无限制的增加。从而可以支持复杂程序的编写。据不完全统计，当 C 语言代码超过 50000 行时，命名冲突就成为问题，从而阻碍程序的开发。

C++语言由两种文件组成，即以.h 为扩展名的头文件和以.cpp 为扩展名的源文件，分别存放各元素的声明和数据、函数及类的定义。

3.2　C++语法元素

C++语法元素包括符号、注释、标识符、关键字、标点符号和操作符。本节同时还讲述了如何进行元素的声明和定义。

3.2.1　最小的元素——符号

C++符号是 C++程序中解析器可以识别的最小的元素。C++解析器可以识别多种符号，包括标识符、关键字、常数、操作符、标点和其他分隔符等。这些符号组合起来，就成为程序指令。符号被“空白”分隔开。空白可以是一个或多个下列元素的组合。

- 空格：当按下 Space 键时，输入的就是空格。
- 水平 Tab 键：此键根据系统定义，可以连续输入几个空格，一般是 4 个空格或 8 个空格。
- 换行：表示在编辑器中光标另起一行。
- 回车：当按下 Enter 键时，输入的就是回车。
- 注释：是用于描述代码的作用，方便开发人员标记程序的功能。

每个处理单元使用输入流处理，解析器使用从左到右的方向扫描输入流，创建更长的符号并从中分隔符号。例如代码如下：

```
a = i+++j;                    //自增一语句的使用示例
```

开发人员可能想实现下面两条语句中的一条：

```
a = i + (++j)
```

```
a = (i++) + j                    //编译器会按照此种方法解析上面的自增语句示例
```

因为解析器分析输入流时，使用从左到右的方向分析，所以，它会采用第二种解释方法。

3.2.2　注释规范

注释是写在程序代码中用于标记代码功能的符号，但是编译器在编译时，会将注释作为空格处理。虽然编译器在编译时忽略注释内容，但是它对程序开发来说非常重要，也是衡量程序质量的一个重要指标。注释的主要作用是注释代码，提供编写准确、适当的注释，对程序员和整个开发团队来说都非常重要，为后期维护和代码共享提供方便。C++支持两种注释方式——单行注释和块注释。

- 单行注释：以两个反斜杠开头，后面加注释内容。此注释方式表示//后一直到行尾的内容全部为注释。
- 块注释：以/*开始，以*/结束，其中的内容全部为注释。

下面代码说明了两种注释的使用：

```
int a=5;                                    //定义整型变量 a，初始化为 5
/*定义整型变量 b，
初始化为 6*/
int b=6;
```

从上面的例子可以看出，在注释出现跨行时，最好使用块注释。当注释比较简短，一行足以显示时，使用单行注释比较简单。需要注意的是，注释是不支持嵌套的，例如：

```
/* 目的：注释整块代码
   问题：每行后的嵌套注释代码是无效的
   char a = 'A';                        /* 初始化字符 */
   cout << "a: " << a << "\n";          /* 打印字符 */
*/
```

上面代码是不能编译成功的，因为编译器在编译时，会为第一个/*查找与它匹配的第一个*/，即第一行的/*与第三行的*/匹配为一对。而第四行的/*与*/匹配为一对，第五行的*/没有匹配的注释符，因此，系统会提示编译错误。在使用单行注释要注意，不允许单行注释后跟行继续符，例如：

```
void main()
{
   printf( "This is a number %d",  //\
           5 );                    //此处使用单行注释会出现错误
}
```

上面的代码编译器进行编译时会提示错误，会将注释符后的行继续符下一行的内容作为空格进行编译，即“5);”会被忽略，因此，编译器会报语法错误。编译的代码如下所示，因此要注意单行注释后不要使用行继续符\。

```
void main()
{
```

```
    printf( "This is a number %d",
}
```

3.2.3　标识符命名规范

C++标识符，是系统预留的用于描述系统使用的元素的名称，由大小写的 26 个英文字母、0～9 之间的 10 个数字以及下划线组成，并且第一个元素必须是字母（大写或小写都可以）或者下划线。标识符是区别大小写的，如 hDevie 变量与 HDevice 变量是不同的。在C++中下列元素需要使用标识符来表示。

- 对象或变量名：在内存中占据一部分空间，C++为它定义一个名称，在程序中使用对象名或变量名就可以直接访问存储空间中的值。如 int a;，语句中的 a 就是变量名。
- 类、结构或联合体名称：实质上是复杂类型的名称的标识符，用于标识不同种类的复杂类型。如 class Student 中的 Student 就是类名。
- 类型名称：表示简单类型的名称的标识符。如 int a 语句中的 int 为整型类型的标识符。
- 类、结构、联合体或枚举的成员：表示在类、结构、联合体或枚举中定义的成员变量的标识符。例如如果在 Student 类中定义 age 变量，则 age 就是类的成员标识符。
- 函数或类成员函数：表示函数名称的标识符。例如如果在 Student 类中定义 CheckIn() 函数，则 CheckIn 就是类的成员函数的标识符。
- typedef 名称：表示类型重定义的标识符。
- 标签名称：表示 C++中用于标记 goto 语句可以跳转到的语句，此处主要用作语句指示。
- 宏名称和宏参数：使用#define 定义的宏的名称和参数。

在 C++中，不能使用关键字作为标识符。但是标识符中可以包含关键字。如 int 是一个非法的标识符，但是 pint 是合法的标识符。在 VC 中，标识符的最大长度为 247。

C++中在全局范围内预留以两个连续的下划线开头或者一个下划线后跟着一个大写字母的标识符，在文件范围内预留一个下划线后跟着一个小写字母的标识符。尽量不要使用这些形式的标识符，以避免与现在或将来预留的标识符冲突。

3.2.4　C++预定义的关键字

在 3.2.3 小节讲过，C++中不能使用关键字作为标识符，而实际上关键字是预定义的具有特殊意义的标识符。C++中预定义的关键字如表 3-1 所示。

表 3-1　C++关键字

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

续表

except	explicit	extern	false
finally	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	type_info
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	while		

在 VC 中，以两个下划线开头的标识符是为编译器预留的。因此，在对 C++的实现中，在特定关键字前加两个下划线作为特定的 C++关键字。如表 3-2 是微软指定的 C++关键字。

表 3-2　微软指定的C++关键字

allocate	dllexport	__inline	__leave	property	__try
__asm	dllimport	__int8	__multiple_inheritance	selectany	uuid
__based	__except	__int16	naked	__single_inheritance	__uuidof
__cdecl	__fastcall	__int32	nothrow	__stdcall	__virtual_inheritance
__declspec	__finally	__int64		thread3	

其中，__asm 替换标准 C++中的 asm。而 allocat、dllexport、dllimport、naked、nothrow、property、selectany、thread 和 uuid 关键字是在使用__declspec 时才有效。默认情况下，VC 支持微软的 C++扩展，但是在编译时指定/Za 命令行选项（ANSI-Compatible）可以关闭此支持。当微软扩展打开时，用户可以在程序中使用上表中列出的关键字。在 ANSI 方式下，需要在这些关键字前加上双下划线标注。为了向后兼容，除了__except、__finally、__leave 和__try 关键字外，同时支持单下划线的关键字和 cdecl 关键字。

3.2.5　标点符号

C++中的标点符号是有语法的，并且对编译器来说，是具有语义的，但是标点符号本身没有语义。有些标点符号，不管是单独的还是组合的，也是 C++操作符或是对预处理器有语义。操作符主要有：

```
! % ^ & * ( ) - + = { } | ~ [ ] \ ; ' : " < > ? , . / #
```

其中，符号 [ ]、( )和 { }必须是成对出现的。

3.2.6　操作符

C++语言包括了所有 C 语言操作符，并且增加了一些 C++特有的操作符。C++操作符分为一元操作符、二元操作符和三元操作符。操作符在进行运算时，严格按照操作符优先

权的顺序进行运算，具有高优先权的运算符先于低优先权的运算符运算，操作符在进行操作时，有“方向性”。处在同一级别的操作符具有相同的优先级，此时，执行顺序为从左向右运算。通过小括号，可以改变操作符的运算顺序。表 3-3 列出了 C++支持的操作符。执行顺序是按照从高到低的优先权依次执行的。

表 3-3　C++操作符

操　作　符	名　　称	方　　向
::	范围确定符	无
::	全局符	无
[ ]	数组下标	从左向右
( )	函数调用	从左向右
( )	转换	无
.	对象的成员选择	从左向右
–>	指针的成员选择	从左向右
++	自增一后缀	无
– –	自减一后缀	无
new	分配对象	无
delete	删除对象	无
delete[ ]	删除对象	无
++	自增一前缀	无
– –	自减一前缀	无
*	乘	无
&	地址符	无
+	加	无
–	减	无
!	逻辑否	无
~	位与	无
sizeof	对象大小	无
sizeof ()	类型大小	无
typeid()	类型名称	无
(type)	类型转换	从右向左
const_cast	类型转换	无
dynamic_cast	类型转换	无
reinterpret_cast	类型转换	无
static_cast	类型转换	无
*	类成员的应用指针	从左向右
–>*	类成员的指针	从左向右
*	乘	从左向右
/	除	从左向右
%	取模	从左向右
+	加	从左向右
–	减	从左向右

续表

操　作　符	名　　称	方　　向
<<	左转换	从左向右
>>	右转换	从左向右
<	小于	从左向右
>	大于	从左向右
<=	小于等于	从左向右
>=	大于等于	从左向右
==	等于	从左向右
!=	不等于	从左向右
&	位与	从左向右
^	位异或	从左向右
|	位或	从左向右
&&	逻辑与	从左向右
||	逻辑或	从左向右
e1?e2:e3	条件	从右向左
=	赋值	从右向左
*=	乘赋值	从右向左
/=	除赋值	从右向左
%=	模赋值	从右向左
+=	加赋值	从右向左
– =	减赋值	从右向左
<<=	左转换赋值	从右向左
>>=	右转换赋值	从右向左
&=	位与赋值	从右向左
|=	位或赋值	从右向左
^=	位异或赋值	从右向左
,	逗号	从左向右

3.2.7　声明与定义

C++使用声明告诉编译器程序中定义了哪些程序元素或对象，而定义则说明了元素所执行的代码或数据。对象在使用前必须先声明。一条声明语句可以声明一个或多个对象。通常程序中需要多次使用声明。要使用多声明，则多声明中定义的元素类型必须相同。除了下面的情况，声明也可以作为定义来使用。

- 声明是一个函数原型，即声明函数，但是没有函数体。
- 包括 extern 标识符时，但是对象和变量没有初始化，或者函数没有函数体。此时表示在当前处理单元中不用进行定义，给定的是外部的连接名称。
- 在类声明中的静态数据成员。因为静态类数据成员是被类的所有对象共享的不连续的变量，因此，必须在类声明的外部定义和初始化。
- 没有定义的类名称的声明，如 class T。
- 使用 typedef 关键字声明的类型。

下面的代码给出了声明也作为定义的用法：

```
int i;                                      //声明和定义整型 int 变量 i
int j = 10;                                 //声明和定义整型 int 变量 j
enum suits { Spades = 1, Clubs, Hearts, Diamonds };    //声明枚举类型
class CheckBox : public Control             //声明继承自 Control 的 CheckBox 类
{
public:
    Boolean IsChecked();                    //判断是否选中的函数声明
    virtual int ChangeState() = 0;          //选择状态变量的函数声明
};
```

下面的代码给出了声明不作为定义的用法：

```
extern int i;                                                  //声明外部整型变量 i
char *strchr( const char *Str, const char Target );    //声明 strchr()函数
```

定义是对象或变量、函数、类或枚举型的唯一说明。因为定义必须是唯一的，所以一个给定的程序元素只能包含一个定义。在声明和定义之间是多对一的关系，同一定义的元素，可以在程序多处声明。有如下两种情况，程序元素可以被声明但是不用定义。

- 函数声明了，但是未被其他函数调用或使用此函数地址的表达式。
- 类仅被使用，但是不需要知道其定义。这也是需要将声明放在.h 头文件中的原因。因此类必须声明。代码如下：

```
class WindowCounter;          //声明引用的外部类 WindowCounter，此类没有定义
class Window                  //声明类 Window
{
   static WindowCounter windowCounter;     //不需要 WindowCounter 定义
};
```

3.3　常量和变量

在编写程序的过程中，需要使用“一些可以区分的实体”存储开发过程中的值。这就产生了常量和变量。C++中使用“一串有意义的字符的组合”（即标识符）标识常量和变量，这样在程序设计中，就可以使用这些常量和变量存取需要的值。对于常量和变量的定义及使用 C++有其语法和规定。本节就介绍有关常量和变量的使用。

3.3.1　定义常量

常量顾名思义就是固定的量，在有效范围内值是不可以变化的。在 C++中使用 const 关键字定义常量，表示标识符表示的值是常量，告诉编译器，不允许程序代码修改它的值。语法如下：

```
//定义一个数据类型为 datatype，名称为 name，取值为 value 的常量
const datatype name=value;
```

上述语法表示定义一个数据类型为 datatype，名称为 name，取值为 value 的常量。如

下代码定义一个数据类型为整型，常量名为 fee 并且取值为 2 的常量：

```
const int fee=2;                          //定义一个整型的常量 fee，值为 2
```

在程序中，使用下列代码是错误的：

```
fee = 10;                                 //错误：常量值是不可以修改的
fee ++;                                   //错误：常量值是不可以修改的
```

C++中常量分为整型常数、字符常数、浮点常数和字符串常数 4 种类型。

1．整型常数

整型常数表示没有小数部分或指数部分的数据常数，代表固定的整数。整型常数可以是十进制数（有效数字位为 0～9）、八进制数（有效数字位为 0～7）或十六进制数（有效数字位为 0～9，A～F）。其可以是有符号数，也可以是无符号数。可以是 long 类型，也可以是 short 类型的。可以是以 0 开头的数字代表八进制数，以 0x 或 0X 开头的数字代表十六进制数，以 u 或 U 结束的数字表示无符号数，以 l 或 L 结束的数字表示长整数，以 i64 结束的数字表示 64 位整数。如下代码显示了整型常量的定义方式。

```
const int a = 347;                        //十进制常量，值为 347
const int c = 0365;                       //八进制的 365
const int d= 0x55ff;                      //十六进制常量
const int e= 0X55FF;                      //十六进制常量，与 d 的值相等
const unsigned uVal = 256u;               //无符号数
const long lVal = 0x7FFFFEL;              //十六进制形式的长整型
const unsigned long ulVal = 076342ul;     //八进制形式的无符号长整型
```

2．字符常数

C++字符常数是字符集的一个或多个成员，使用单引号引起来。VC 使用 ASCII 字符集。字符常数有 3 种形式——标准字符常数、多字符常数和宽字符常数。

```
const char ch = 'y';                      //标准字符常数
const int mbch = 'xy';                    //指定依赖于系统的多字符常数
const wchar_t wcch = L'xy';               //指定宽字符常数
```

此处 mbch 是一个 int 类型的。如果将它声明为 char，则其中的 y 将会被忽略。一个多字符常数可以包含 4 个字符，指定超过 4 个字符的字符常数时，会发生错误。因为一个字符是 8 位的，而 int 是 32 位的，因此，只能代表 4 个字符。

微软 C++支持标准、多字符和宽字符常数。使用宽字符常数可以指定扩展的可执行字符集的成员。标准字符常量使用类型 char，多字符常数使用类型 int，宽字符常数使用类型 wchar_t，这 3 种类型分别在 STDDEF.H、STDLIB.H 和 STRING.H 文件中定义，其中宽字符函数在 STDLIB.H 文件中定义。

标准字符常数和宽字符常数的定义方式的不同在于，需要在宽字符常数取值前加上字母 L。例如代码如下：

```
const char schar = 'a';                   //标准字符常数
const wchar_t wchar = L'\x81\x19';        //宽字符常数
```

从上面的代码中可以看出，在定义 wchar 宽字符常数时使用\x81 的形式，这里用到了转义符反斜线。在 C++中，使用反斜线加上指定码值代表特殊的字符。如表 3-4 所示列出了 C++中常用的转义字符。

表 3-4　C++常用转义字符

字　　符	ASCII 表示方法	ASCII 值	转 义 序 列
换行	NL (LF)	10 or 0x0a	\n
水平 Tab 键	HT	9	\t
垂直 Tab	VT	11 or 0x0b	\v
退格键	BS	8	\b
回车	CR	13 or 0x0d	\r
进制符	FF	12 or 0x0c	\f
反斜线	\	92 or 0x5c	\\
问号	?	63 or 0x3f	\?
单引号	'	39 or 0x27	\'
双引号	"	34 or 0x22	\"
八进制数	ooo	—	\ooo
十六进制数	hhh	—	\xhhh
Null 字符	NUL	0	\0

如果在反斜杠后的字符不是合法的换码字符，各种 C++处理各不相同，在 VC 中，会报 unrecognized character escape sequence 警告，表示不能识别的转义字符序列。

3．浮点型常数

浮点型常数用于表示包含小数部分的常数，默认类型为 double 型。浮点型常数包含小数点，也可以包含指数。浮点常数中的 e 或 E 后面部分的内容为指数部分，表示浮点数的数量级，数值为 10 的次幂，在常数前面的+或–表示符号，在常数后面的 f（或 F）、l（或 L）分别表示其类型为 float 或 long。下面是浮点型常数的例子。

```
const double = 18.46            //浮点型常数
const double = 38.              //浮点型常数
const double = 18.46e0          //浮点型常数，18.46 的 10 的 0 次幂
const double = 18.46e1          //浮点型常数，18.46 的 10 的 1 次幂
```

4．字符串常数

字符串常数是包含 0 个或多个字符集中的字符的字符串，使用双引号引起来。字符串常数是一个以 NULL 结束的字符序列。字符串的连接可以采用多种方式实现，例如代码如下：

```
char szStr[] = "12" "34";           //定义字符数组
char szStr[] = "1234";              //定义字符数组与上一行的作用相同
cout << "今天是星期一 "
       "并且是中秋节 "
       "也就是八月十五";             //输出内容，为一组字符串
```

```
cout << "今天是星期一  \
         并且是中秋节  \
         也就是八月十五.";              //输出内容，为字符串
```

3.3.2　常量成员函数

除了常数类型的常量，还可以将类的成员函数定义为常量，即常量成员函数。在函数体内不可以修改任何数据成员，也不可以调用任何不是常量的成员函数。语法是在函数定义后加上 const 关键字，格式如下：

```
returntype class:functionname(param1,param2,……)  const
{
    //body—函数体
}
```

其中，returntype 表示返回类型，class 表示成员函数所属的类，functionname 表示成员函数的函数名称，param1,param2 等表示常量成员函数的参数，const 表示此成员函数是常量成员函数，{}中是成员函数的函数体。注意，在 C++中定义常量成员函数时，要在声明和定义后都加上 const 关键字，示例代码如下：

```
class Date
{
public:
   int getMonth() const;             //获取当前月的取值的只读函数
   void setMonth( int mn );          //写函数，不能定义为 const
private:
   int month;                        //存放月的变量
};
int Date::getMonth() const           //获取当前类中的月变量的值
{
   return month;                     //只是返回月变量的值，没有修改任何内容
}
void Date::setMonth( int mn )        //设置类中月变量的值
{
   month = mn;                       //修改数据成员
}
```

3.3.3　定义变量

与常量不同的是，变量在定义后，可以根据程序的需要对值进行修改。变量的定义方法与常量的定义方法类似，只是去掉了 const 关键字，其语法如下：

```
//定义一个数据类型为 datatype，名称为 name，取值为 value 的变量
datatype name=value;
```

以下代码定义了一个变量名称为 balance 的整型变量，并为其分配初始值 0。

```
int balance = 0;
```

3.3.4　代码的有效范围——作用域

每个 C++变量只能在程序的一定范围内使用，此范围称为变量的作用域。除了静态对象外，作用域可以确定变量的生命期。当调用类的构造函数和析构函数、变量在作用域内初始化时，作用域还确定变量的可见性。

1．作用域类型

C++中共分 5 种作用域，如下所述。

（1）本地作用域。在代码块中声明的变量只能在声明块中声明语句后访问。比如具有函数块中作用域的函数形参具有本地作用域，即在函数体内声明的变量只能在函数体内使用。例如代码如下：

```
{
   int i;                  //定义本地作用域的变量
}
```

上面代码中变量 i 在大括号内声明，因此，i 的作用域为本地作用域，只能在大括号内的代码中使用。

（2）函数作用域。只有标签属于此作用域的变量。标签可以在定义它的函数内任意地方使用，函数外部就不能使用。

（3）文件作用域。在代码块或类的外部声明的变量具有文件作用域。可以在处理单元中声明点后的任何地方访问。文件作用域对象既可以是静态对象也可以是非静态对象，其中具有文件作用域的非静态变量通常称为全局变量。

（4）类作用域。类的成员变量具有类作用域。只能通过使用对象的成员选择操作符（.或->）或类对象的指针成员操作符（*或->*）访问。而非静态类成员数据可以看作类对象的本地作用域。例如代码如下：

```
class Point          //定义代表点的类
{
   int x;            //点的 X 坐标，具有类作用域，也可以看作类对象的本地作用域
   int y;            //点的 Y 坐标，具有类作用域，也可以看作类对象的本地作用域
};
```

在上面的代码中，Point 类的 x 和 y 成员的作用域可以看作 Point 类的类作用域。

（5）原型作用域。函数原型中声明的变量只在原型声明范围内有效。以下代码声明了 strcpy()函数原型，其中变量 szDest 和 szSource 的作用域是在函数原型声明的范围内。

```
char *strcpy( char *szDest, const char *szSource );    //strcpy()函数原型
```

2．理解作用域

虽然作用域的概念是隐形的，但是对于它理解不透，会在程序中出现错误或隐藏的问题。下面对作用域的理解做些说明。

（1）变量的声明点为声明后和初始化前的程序点；枚举类型的声明点是定义了标识符，

但是还没有初始化前。例如代码如下：

```
double dVar = 7.0;          //定义 double 类型的变量，并初始化为 7.0
void main()
{
   double dVar = dVar;      //将全局变量的值赋值给本地作用域的变量
}
```

在上面代码中，声明点在初始化之前，则本地 dVar 应该初始化成全局变量 dVar 的值，即 7.0。枚举值的处理方式是相同的。例如代码如下：

```
//定义 4 个值分别为 1、2、3、4 的常量
const int Spades = 1, Clubs = 2, Hearts = 3, Diamonds = 4;
enum Suits                  //定义 Suits 枚举类型
{
   Spades = Spades,         //错误
   Clubs,                   //错误
   Hearts,                  //错误
   Diamonds                 //错误
};
```

上面代码定义了常量 Spades、Clubs、Hearts 和 Diamonds，这些常量的作用域为全局作用域，因此在枚举值 Suits 中使用这些常量定义枚举值是错误的。因此，在编写代码时，即使作用域不相同，也应该尽量避免名称重复。在 C++中，提供了一种隐藏名称的方法。使用这个方法可以将变量的作用域限制在一定范围内。例如代码如下：

```
Test()          //Test()函数
{
        int i = 0;          //定义函数作用域的 int 类型的变量 i，并初始化值为 0
        cout << "i=" << i << "\n"   //输出 i 的值
        {
            int i = 7, j = 9;       //定义局部作用域的变量 i 和变量 j
            cout <<"i="<<i <<"\n" <<"j=" << j <<"\n";
                                    //输出局部作用域的 i 值和 j 值
        }
        cout <<"i = " <<i << "\n"; //输出函数作用域的 i 值
}
```

在上面代码中，程序在函数中使用了一对大括号，将代码包括起来，在大括号内的变量 i 的作用域为大括号内，其值不会影响大括号外的变量 i 的作用域，运行结果如下：

```
i = 0
i = 7
j = 9
i = 0
```

（2）当在文件中声明具有文件作用域的变量或函数与块中定义的变量或函数名称相同时，可以通过使用作用域确定操作符（::）访问文件作用域的名称。例如代码如下：

```
#include <iostream.h>
int i = 7;                                          //定义文件作用域的变量 i
void main()
{
  int i = 5;                                        //定义块作用域的变量 i
  cout << "块作用域 i 的值为: " << i << "\n";        //输出块作用域 i 的值
```

```
   cout << "文件块作用域 i 的值为: " << ::i << "\n";   //输出文件作用域 i 的值
}
```

上面代码运行的结果为：

```
块作用域 i 的值为:5
文件块作用域 i 的值为: 7
```

（3）在同一作用域中，当函数与变量的名称相同时，通过在名称前加上前缀 class 表示访问的是类对象。例如代码如下：

```
class Account                                          //在文件范围内声明类 Account
{
public:
    Account( double InitialBalance )  { balance = InitialBalance; }
   double GetBalance()  { return balance; }    //返回账户中的余额值
private:
   double balance;                                     //存放账户余额的变量
};
double Account = 15.37;                                //隐藏类名 Account
void main()
{
    class Account Checking( Account );          //限定 Account 作为类名
    cout << "账户余额为: " << Checking.GetBalance() << "\n";
}
```

上面代码定义了类名为 Account 的类和变量名为 Account 的全局变量。在 main()函数的第一行中，定义了变量名为 Checking 的对象。通过在 Account 前加上 class 前缀，表示定义的是 Account 类的对象变量，而括号中的 Account 表示的是全局变量 Account，类型为 double 类型。下面代码显示了使用 class 关键字声明 Account 类型的指针的方法。

```
//定义 Account 指针的类变量
class Account *Checking = new class Account( Account );
```

3.4　数 据 类 型

C++中包含 3 种数据类型：基本数据类型、派生数据类型和 C++类。基本数据类型主要指内置在语言中的数据类型，如 int、char、float 等。派生数据类型指从基本数据类型派生而来的新类型。本节主要介绍 C++中的基本数据类型和派生数据类型。因为类是 C++中一个比较重要的概念，所以将在第 4 章中详细讲解 C++类。

3.4.1　基本数据类型

C++中基本数据类型指内置在语言中的数据类型，分为 3 类，分别是定点类型、浮点类型和空类型（void）。定点类型指在固定长度的存储空间内准确的存储一个数值；浮点类型指使用近似值表示一个数值，其有可能带有小数部分；空类型指空值，任何变量都不可以定义为 void，主要作用是表示函数不返回任何值或者无类型或任意类型的数据。表 3-5 列出了 C++中的基本数据类型以及 VC 中存储相应数据类型使用的长度。

表 3-5　C++基本数据类型

种类	数据类型	说　　明	长度
定点类型	char	char 类型表示字符型。在 VC 中，表示 ASCII 字符。char 类型分为 unsigned char 和 signed char 两种，分别表示无符号字符型和有符号字符型。默认情况下，char 类型是指 signed char 类型。实际上，在编译时，将字符型按照整型处理	1 字节
	short	short 类型（又称为 simply short 或者是 short int）是长度大于 char 类型，小于 int 类型的整型，又称为短整型。short 类型分为 signed short 和 unsigned short 两种，分别表示有符号短整型和无符号短整型。short 类型也就是 signed short 类型	2 字节
	int	int 类型是长度大于 short 类型，并且小于 long 类型的整型，int 类型分为 signed int 和 unsigned int，分别表示有符号整型和无符号整型。int 类型也就是 signed int 类型	4 字节
	__intn	定长整型。其中，n 是以比特为单位的整型长度。n 的取值可以是 8、16、32 和 64。也就是说可以使用它定义 8 位、16 位、32 位或者是 64 位的整型，而不需要使用 short、int、long 等类型。这样可以避免不同的 C++标准环境有可能为这些类型分配的存储空间大小不相同。根据 n 的取值，长度不相同，长度是 n/8 个字节。如 8 位整型，长度为 1 个字节	不定
	long	long 类型（或者是 long int）是长度大于 int 类型的整型。long 类型分为 signed long 和 unsigned long 两种，分别表示有符号长整型和无符号整型。long 类型也就是 signed long 类型	4 字节
浮点类型	float	float 是长度最小的浮点类型	4 字节
	double	double 是长度大于 float 类型，并且小于 long double 的浮点类型	8 字节
	long double	long double 是长度等于 double 类型的浮点类型	8 字节
空类型	void	void 表示空值	

表 3-5 列出的定点数中默认情况下，不带 signed 和 unsigned 标识的类型分别表示对应的有无符号类型，如 int 表示 signed int 类型。需要注意的是，当使用/J 编译选项时，默认的 char 类型为无符号字符型，即 unsigned char，其他类型的默认类型没有变化，仍然是有符号类型。

__intn 类型的数据类型与具有相同长度的数据类型是等同的。在 Visual Studio 2010 环境下，__int8 类型与 char 类型等同；__int16 类型与 short 类型等同；__int32 类型与 int 类型和 long 类型等同。

注意：C++中各种基本数据类型所占的存储空间根据 C++实现的不同而不同，表 3-5 第四列列出的基本数据类型长度是指在 Visual Studio 2010 下各种类型数据所占的存储空间。

3.4.2　数据类型的转换方式

C++中有多种数据类型，但在实际使用时，经常会遇到需要在不同数据类型之间转换

数据的情况，这时，就需要对数据进行类型转换。C++中进行数据类型转换的语法格式为：

```
数据类型转换表达式=(类型名称)转换表达式
```

其中，转换表达式表示要进行数据类型转换的表达式，而类型名称表示要将转换表达式转换成的类型。类型转换为对象提供了在适当情况下显式地将它转换成其他类型的方法。当完成数据转换后，编译器会将转换表达式作为类型名称指定的类型处理。数据转换可以在任何可度量数据类型之间进行转换，而且显式数据类型转换的规则与隐式数据类型转换的规则是相同的。具体参看如下代码：

```
void printTypeCast()                    //数据类型转换示例
{
    double x = 57.98;                   //定义 double 类型的变量
    cout << " x=" << x << "\n";         //输出 double 类型变量值
    int y = (int)x;                     //将 double 类型的变量值转换为 int 类型的值，
                                        //并存入 y
    cout << "y=" << y << "\n";          //输出 int 类型的 y 的变量值
}
```

上面代码将 double 类型的 x 转换成 int 类型，并存储在 y 中输出到界面上。从 double 型转换成 int 型时，会将 double 型数据的小数部分后的内容去掉。运行结果如下：

```
x=57.98
y=57
```

3.4.3　数组

派生数据类型又可以分为直接派生数据类型和组合派生数据类型。其中，直接派生数据类型指从基本数据类型直接派生而来的数据类型，包括数组、函数、指针和引用等。这些类型所指向的核心数据是基本数据类型，因此称为直接派生数据类型。组合派生数据类型指将基本数据类型组合而成的新的数据类型，其中不止包含一种基本数据类型，因此称为组合派生数据类型，包括类、结构体和共用体。

数组是包含指定数目的特定类型的变量或对象的集合。如一个由整型派生而来的数组，是一个整型数组。下面的代码定义了一个具有 10 个 int 变量的数组和一个具有 5 个 SampleClass 类对象的数组。

```
int         ArrayOfInt[10];         //定义具有 10 个 int 类型元素的数组
SampleClass aSampleClass[5];        //定义具有 5 个 SampleClass 类型元素的数组
```

在 C++中使用数组元素访问操作符（[]）访问数组元素。使用方法是在表达式后加上[]后，并加上表示数组对象元素在数组中的位置的下标。其语法为：

```
数组标识符[下标表达式]
```

其中，下标表达式表示要访问的元素在数组中的位置，它必须是可整型化的表达式。要注意的是，C++中下标值是基于 0 起始的，即数据中的第一个元素的下标为 0，第二个元素的下标为 1，依次类推。数组标识符表示要获取的数组元素所在数组的指针值。上面所说的都是一维数组，数组还可以是多维的。多维数组是一个具有数组元素的数组。因此，

三维数组也就可以看作是一个具有二维数组的数组。其语法格式为：

```
数组标识符[下标表达式 1] [下标表达式 2]...
```

下标表达式是按照从左向右的运算方向。首先计算最左边的下标“数组标识符[下标表达式 1]”，地址生成一个指针表达式。然后再向此指针表达式添加[下标表达式 2]，形成一个新的指针表达式。依此类推，直到最后一个下标表达式添加上。当运算完成后，如果最后得到的指针指向的数据不是数组类型时，就可以使用间接访问操作符（*）获取该值。例如代码如下：

```
int nYearsMonthsDays[20][12][16];          //三维数组的示例
```

上面代码定义了一个三维数组，表示 20 年间的任何一天。第一维表示 20 年中的一年，第二维表示指定年份中的某月，第三维表示指定年份的指定月份的某天。

3.4.4　结构体

C++中的结构体与类相同，区别在于结构体的所有成员数据和函数默认情况下都是具有公开权限的，并且默认是公开继承的。C++中结构体的定义语法如下：

```
struct struct_name      //结构体名称
{
    //body 结构体定义
};
```

其中，struct_name 表示定义的结构体的名称，并且在//body 的位置定义结构体的成员变量，示例如下：

```
struct MyDateTime       //日期时间结构体
{
    int year;           //年
    int month;          //月
    int day;            //日
    int hour;           //时
    int minute;         //分
    int second;         //秒
};
```

上面代码定义了表示日期时间的结构体，结构体的成员变量 year、month、day、hour、minute、second 分别用于存储日期时间的年、月、日、时、分、秒。

3.4.5　共用体

C++提供了一种共享内存的存储方法，称为共用体。它是在同一内存空间中可以定义多种数据元素，而在同一时间只能包含一种数据元素的类型，数据元素的类型可以是简单数据类型，也可以是数组或类等复杂数据类型。共用体的成员代表共用体包含的数据种类。共用体的存储空间是其成员列表中占用空间最大的成员的存储空间大小。其语法格式如下：

```
union 共用体名称 { 成员列表 } ;
```

其中，共用体名称指定了共用体类型的类型名。成员列表中可以定义共用体成员数据，也可以定义共用体成员函数。共用体成员数据可以是各种数据类型，但是不能定义为具有构造函数和析构函数的类、不能定义为具有重载赋值操作符的类、不能定义为静态数据成员。共用体的成员函数与类中的函数是相同的，可以是一般的成员函数，也可以是特殊函数，如构造函数和析构函数，但是都不能定义为虚函数。共用体不具有派生性和继承性。下面代码演示了如何使用共用体，工程名为 SampleUnion。

```
01  #include <stdio.h>
02  #include <string.h>
03  #include <stdlib.h>
04  #include <LIMITS.H>
05
06  enum NumType                        //声明一个枚举类型来描述要输出的类型
07  {
08      INTEGER_INT,                    //整型类型
09      INTEGER_LONG,                   //长整型类型
10      INTEGER_DOUBLE                  //double 类型
11  };
12  union NumValue                      //声明一个包含下面 3 种类型的共用体
13  {
14      int        iValue;              //int 类型值
15      long       lValue;              //long 类型值
16      double     dValue;              //double 类型值
17  };
18
19  void main( int argc, char *argv[] )
20  {
21      int count = argc - 1;                          //计算输入的参数个数
22      NumValue *Values = new NumValue[count];        //存放值的共用体
23      NumType *Types = new NumType[count];           //存放类型的数组
24      for( int i = 1; i < argc; ++i )                //循环处理每个参数
25      {
26          //判断输入参数中是否包含小数点
27          if( strchr( argv[i], '.' ) != 0 )
28          {
29              //为 dValue 成员赋值,并记录类型
30              Values[i].dValue = atof( argv[i] );
31              //记录数组的成员的类型为 double 型
32              Types[i] = INTEGER_DOUBLE;
33          }
34          else                                       //不是 floating 类型
35          {
36              if (( atol( argv[i] ) >  INT_MAX )||(atol(argv[i]) <  0))
37              {
38                  //如果数据大于 int 类型的最大值，则将其存储在 lValue 成员中
39                  //并记录类型
40                  //将值转换成长整型
41                  Values[i].lValue = atol( argv[i] );
42                  //记录数组的成员的类型为长整型
43                  Types[i] = INTEGER_LONG;
44              }
45              else
46              {
47                  //否则，将其存储在 iValue 成员中,并记录类型
```

```
48                  Values[i].iValue = atoi( argv[i] );//将值转换成整型
49                  //记录数组的成员的类型为整型
50                  Types[i] = INTEGER_INT;
51              }
52          }
53          switch( Types[i] )          //根据类型种类，将种类信息和值信息输出
54          {
55          case INTEGER_INT:           //如果数据为整型，则输出整型值
56              printf( "数据类型为 Integer，值为%d\n", Values[i].iValue );
57              break;
58          case INTEGER_LONG:          //如果数据为长整型，则输出长整型值
59              printf( "数据类型为 Long，值为%d\n", Values[i].lValue );
60              break;
61          case INTEGER_DOUBLE:        //如果数据为 double 型，则输出 double 值
62              printf( "数据类型为 Double，值为%f\n", Values[i].dValue );
63              break;
64          }
65      }
66      system("pause");
67  }
```

在上面代码中，NumValue 共用体有 3 个成员，分别是 iValue、lValue 和 dValue，其分别是整型、长整型和双精度类型，根据具体为成员的赋值决定其类型。而其 3 个成员共享同一块内存。其内存分配如图 3-1 所示。

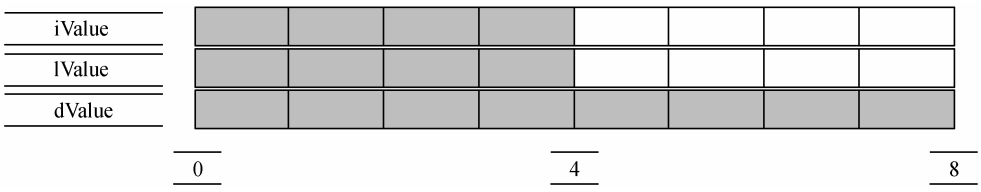

图 3-1　共用体的内存分配

上面代码首先取出通过命令行传入的参数个数，然后依次处理每个参数。在处理每个参数时，首先根据是否含有小数点判断输入的是否是浮点型。如果是浮点型，则将输入的参数值存入共用体的浮点型成员 dValue 中；如果不是浮点型，则根据转换后的值是否大于最大的整型值 INT_MAX 判断是整型还是长整型。根据结果，分别存入 iValue 和 lValue 成员变量中。最后根据判断的类型，输出类型信息和值。代码的运行结果如图 3-2 所示。

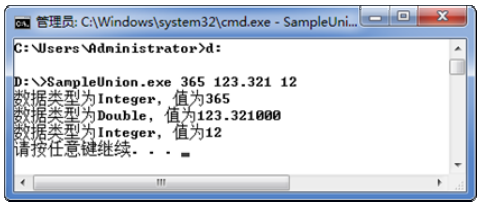

图 3-2　共用体示例的运行结果

3.4.6　匿名共用体

为了简化操作，C++还提供了一种特殊的共同体——匿名共用体，即没有声明共用体

名称的共用体。其语法格式为：

```
union { 成员列表 } ;
```

虽然匿名共用体与共用体一样，是成员共享同一块内存。但是匿名共用体不是声明类型，而是声明对象。因此，在匿名共用体中声明的名称不能与在其相同范围内声明的其他名称冲突。在匿名共用体中声明的名称可以直接使用，就像没有使用 union 声明的变量一样，只是其中的变量共享同一块内存。要使得共用体变成匿名共用体，除了要符合共用体的要求外，不能定义为静态，也不能包含成员函数。下面的代码用匿名共用体改写了上小节的程序。

```
01  #include <stdio.h>
02  #include <string.h>
03  #include <stdlib.h>
04  #include <LIMITS.H>
05
06  struct NumForm                          //表示数值的匿名共用体
07  {
08      enum NumType                        //声明一个枚举类型，用于描述要输出的类型
09      {
10          INTEGER_INT,                    //整型类型
11          INTEGER_LONG,                   //长整型类型
12          INTEGER_DOUBLE                  //double 类型
13      };
14      NumType type;                       //值的类型
15      union                               //声明一个包含下面 3 种类型的共用体
16      {
17          int        iValue;              //int 类型值
18          long       lValue;              //long 类型值
19          double     dValue;              //double 类型值
20      };
21      void print();                       //打印信息的函数
22  };
23  void NumForm::print()                   //根据数据类型，打印相应的信息
24  {
25     switch( type )                       //判断类型
26     {
27      case INTEGER_INT:                   //如果是整型，则输出整型值
28          printf( "数据类型为 Integer，其值为%d\n", iValue );
29          break;
30      case INTEGER_LONG:                  //如果是长整型，则输出长整型值
31          printf( "数据类型为 Long，其值为%d\n", lValue );
32          break;
33      case INTEGER_DOUBLE:                //如果是 double 型，则输出 double 值
34          printf( "数据类型为 Double，其值为%f\n", dValue );
35          break;
36     }
37  }
38
39  void main( int argc, char *argv[] )
40  {
41      int count = argc - 1;                          //计算输入的参数个数
42      NumForm *Values = new NumForm[count];          //存放输入的参数信息
43      for( int i = 1; i < argc; ++i )                //循环处理每个参数
44      {
```

```
45          //floating 类型。为 dValue 成员赋值，并记录类型
46          if( strchr( argv[i], '.' ) != 0 )
47          {
48              Values[i].dValue = atof( argv[i] );//转换成 float 类型
49              Values[i].type = NumForm::NumType::INTEGER_DOUBLE;
50          }
51          else                                        //不是 floating 类型
52          {
53          {   //如果数据大于 int 类型的最大值
54              //则将其存储在 lValue 成员中，并记录类型
55              if (( atol( argv[i] ) >  INT_MAX )||(atol( argv[i] )<0))
56              Values[i].lValue = atol( argv[i] );    //转换成 long 类型
57              Values[i].type = NumForm::NumType::INTEGER_LONG;
58              }
59              else
60              {   //否则，将其存储在 iValue 成员中，并记录类型
61                  Values[i].iValue = atoi( argv[i] );//转换成 int 类型
62                  Values[i].type = NumForm::NumType::INTEGER_INT;
63              }
64          }
65          Values[i].print();                              //打印数值
66      }
67  }
```

从上面的代码中可以看出，在 NumForm 的 NumForm::print()成员函数中，对共用体的 3 个数据成员的访问就像声明数据成员一样，唯一区别就是共用体的 3 个数据成员共享同一块的内存。

3.4.7　枚举类型

枚举类型是用户子定义类型，包含一组命名的常数即枚举成员。默认情况下，第一个枚举成员的值为 0，每个连续的枚举成员比上一个枚举成员大 1，除非显式地为枚举成员指定值。枚举成员的取值可以重复。每个枚举成员的名称被作为一个常数，在 enum 定义的范围内必须唯一。枚举成员可以转换为整型值，但是，将整型值转换为枚举成员时需要显式转换，并且当枚举成员的取值重复时，结果是不确定的。C++中使用 enum 关键字指定枚举类型，语法格式为：

```
enum 枚举类型名称;              //定义枚举类型
```

在 C++中，在类中定义的枚举成员只能由类的成员函数访问，除非在它前面冠上类名（如，类名::枚举成员）。用户可以使用相同的语法直接访问类型名（即，类名::类型名）。如以下代码所示，使用 enum 关键字定义枚举类型。

```
01  enum Days
02  {
03     saturday,                //saturday = 0 默认值
04     sunday = 0,              //sunday = 0
05     monday,                  //monday = 1
06     tuesday,                 //tuesday = 2
07     wednesday,               //依次类推
08     thursday,
09     friday
```

```
10  } today;                    //定义 Days 类型的变量 today
11  int  tuesday;               //错误，重复定义 tuesday
12  enum Days yesterday;        //在 C 和 C++中都合法
13  Days tomorrow;              //只在 C++中合法
14  yesterday = monday;         //为 yesterday 变量赋值为 Monday
15  int i = tuesday;            //有效，i = 2
16  yesterday = 0;              //错误，因为没有进行类型转换
17  yesterday = (Days)0;        //有效，但是结果不确定
18                              //不确定为 saturday 还是 sunday
```

上面代码演示了定义枚举类型 Days，其中存放每星期的星期数，并示范了如何使用枚举类型。

3.4.8　用 typedef 定义类型

C++不仅提供了丰富多样的类型，还提供了定义类型别名的关键字 typedef。typedef 可以为基本类型和派生类型定义别名。代码如下：

```
typedef unsigned char   BYTE;   //定义长度为 8 比特的无符号字符类型的别名为 BYTE
typedef BYTE *          PBYTE;  //定义 BYTE 指针的别名为 PBYTE
BYTE                    by;     //声明一个类型为 BYTE 的变量
PBYTE                   pbBy;   //声明一个指向 BYTE 类型的指针变量
```

由上面的代码可以看出，typedef 简化了数据类型的使用。对于比较长的数据类型，可以为其定义简短有意义的数据类型别名，这样方便使用。同时，当开发平台更换，根据需要对数据类型的使用发生变化时，只需要修改使用 typedef 的定义语句，就可以方便地实现数据类型的平台移植，而不需要修改程序中所有使用数据类型的地方。使用 typedef 不仅可以定义类型的别名，而且还可以定义函数类型的别名。代码如下：

```
void func1();                   //func1()函数原型
void func2();                   //func2()函数原型
typedef void (*PVFN)();         //定义 PVFN 为指向函数的指针，返回值为 void
PVFN pvfn[] = { func1, func2 };//声明函数指针数组
(*pvfn[1])();                   //执行函数指针数据中的第二个函数，即 func2()
```

上述代码定义了 func1()函数和 func2()函数，使用 typedef 关键字定义了函数类型的别名 PVFN，并定义了函数指针数组，初始化成员为 func1 和 func2，最后调用了函数指针数组中的第二个函数 func2()。

3.4.9　位域

类和结构可以包含占用比整型类型还小的存储空间的成员，这些数据成员称为位字段或位域。其语法格式如下：

```
数据成员声明 : 占用的位数
```

其中，数据成员声明部分表示在程序中访问时使用的名称，必须是整型类型。占用的位数部分指定此成员在结构中占用的位数。代码如下：

```
struct Date
{
   unsigned nWeekDay  : 3;      //取值范围 0～7  （3 位）
   unsigned nMonthDay : 6;      //取值范围 0～31  （6 位）
   unsigned nMonth    : 5;      //取值范围 0～12  （5 位）
   unsigned nYear     : 8;      //取值范围 0～100 （8 位）
};
```

上面代码定义了时间结构，使用位域成员确定成员的存储。在位域定义中，如果定义的位数超过了其定义的类型的长度，则系统会自动分配新的存储单元作为后面的位域存储空间，而其类型与定义的类型是相同的。位域的存储空间的分配如图 3-3 所示。

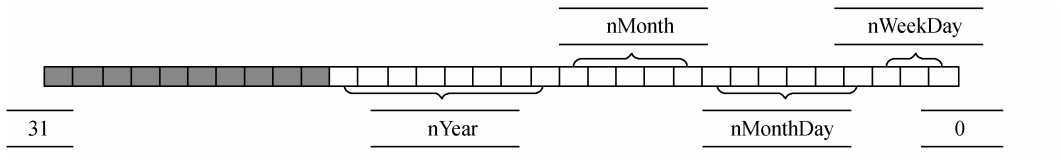

图 3-3　位域的内存分配

注意：在 Visual C++中，位域内存的分配是从低字节到高字节的。

如果忽略位字段的名称，则会填充数据剩下的位数，其后定义的成员会从新的存储空间开始重新分配。如下代码所示，声明了一个未命名的长度为 0 的字段：

```
struct Date
{
   unsigned nWeekDay  : 3;      //取值范围 0～7（3b）
   unsigned nMonthDay : 6;      //取值范围 0～31（6b）
   unsigned           : 0;      //强制对齐到下一个存储空间
   unsigned nMonth    : 5;      //取值范围 0～12（5b）
   unsigned nYear     : 8;      //取值范围 0～100（8b）
};
```

其中在 Date 结构中定义了一个长度为 0 的字段，这会使后面的字段从新的存储单元开始存储，如图 3-4 所示，nMonth 和 nYear 会从新的整型空间开始存储。

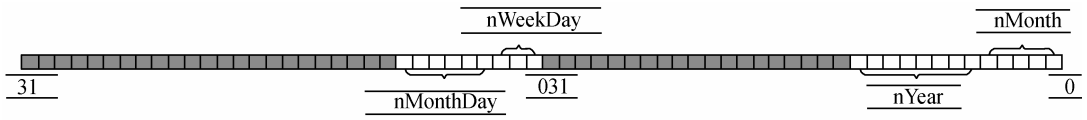

图 3-4　位域的扩展应用

位域可以充分利用存储空间，节约空间，提高程序运行效率。它在通信协议的解析等对存储位要求严格的情况下非常有用。但是在位域上进行操作时，不能操作位域的地址，也不能初始化位域的引用。

3.5　运算符和表达式

C++中变量是用来存储数据的，而运算符是用于操作数据的。使用运算符的语句组成

了表达式。C++语言中表达式的形式多种多样，但是其核心就是使用运算符操作变量或常量对象中的数据，完成预期功能。本节将介绍 C++中支持的运算符及其使用方法和注意事项。

3.5.1　算术运算符

C++提供了一组算术运算符，用于完成算术运算。其语法格式为：

```
左操作数 算术运算符 右操作数
```

上面的左操作数和右操作数是要进行算数运算的操作数，必须是可度量的数据类型。而算术运算符与现实世界中的算术运算的含义是相同的，如加法、减法、乘法、除法、取模等。系统支持的算术运算符如表 3-6 所示。

表 3-6　算术运算符

算术运算符	功　　能
+	加号运算符将两个操作数相加。两个操作数可以都是整型或浮点型，或者一个是指针型，一个是整型
–	减号运算符从第一操作数中减去第二个操作数。两个操作数可以都是整型或浮点型，或者一个是指针型，一个是整型
*	乘号运算符将两个操作数相乘。两个操作数可以都是整型或浮点型。使用乘号操作符时，得到的乘积会转换成结果代表的类型，但是它不处理溢出或下溢情况，如果存放结果的变量类型不够乘积结果的值，可能会丢失数据
/	除号运算符使用第一个操作数除以第二个操作数
%	取模运算符计算第一个操作数除以第二个操作数的余数

具体的使用方法，如以下代码所示。

```
void printMath()                                //算术运算符示例
{
    int a=7, b=8;                               //定义整型变量 a 和变量 b
    cout << "a+b=" << (a + b) << "\n";          //输出 a+b 的结果
    cout << "a-b=" << (a - b) << "\n";          //输出 a-b 的结果
    cout << "a*b=" << (a * b) << "\n";          //输出 a*b 的结果
    cout << "a/b=" << (a / b) << "\n";          //输出 a/b 的结果
    cout << "a%b=" << (a % b) << "\n";          //输出 a%b 的结果
}
```

上面代码依次演示了加法、减法、乘法、除法和取模运算符的使用。程序运行结果如下：

```
a+b=15
a-b=-1
a*b=56
a/b=0
a%b=7
```

3.5.2　赋值运算符

C++提供了一组赋值运算符，赋值运算符都是二元运算符，可以使用适当的赋值运算

符，执行运算后再赋值。使用赋值运算符的表达式称为赋值表达式。赋值运算符会将执行完运算的右操作数的值分配给运算符左边的表达式。因此，赋值操作的左操作数必须是可修改的。赋值后，赋值表达式具有左操作数的值。表 3-7 中列出了 C++支持的赋值运算符。

表 3-7　赋值运算符

赋值运算符	功　　能
=	简单赋值运算符，将右操作数赋值给左操作数
*=	乘赋值运算符，将右操作数和左操作数的乘积赋值给左操作数
/=	除赋值运算符，将左操作数除以右操作数的商赋值给左操作数
%=	取余赋值运算符，将左操作数除以右操作数的余数赋值给左操作数
+=	加赋值运算符，将左操作数和右操作数的和赋值给左操作数
–=	减赋值运算符，将左操作数减去右操作数的差赋值给左操作数
<<=	左移赋值运算符，将左操作数的二进制位向左移动右操作数个位，并赋值给左操作数
>>=	右移赋值运算符，将左操作数的二进制位向右移动右操作数个位，并赋值给左操作数
&=	位与赋值运算符，将左操作数和右操作数的与结果赋值给左操作数
|=	位或赋值运算符，将左操作数和右操作数的或结果赋值给左操作数
^=	位异或赋值运算符，将左操作数和右操作数的异或结果赋值给左操作数

具体的使用，参看如下代码：

```
void printAssignment()                                  //赋值运算符示例
{
    int a = 2, b = 3, c = 6, d = 9, e, f = 4;           //定义整型变量 a、b、c、d、
                                                        //e、f
    cout << "(e=d) e=" << (e=d, e) << "\n";             //输出(e=d, e)结果
    cout << "(a*=b) a=" << (a*=b, a) << "\n";           //输出(a*=b, a)结果
    cout << "(c/=b) c=" << (c/=b, c) << "\n";           //输出(c/=b, c)结果
    cout << "(a%=f) a=" << (a%=f, a) << "\n";           //输出(a%=f, a)结果
    cout << "(c+=d) c=" << (c+=d, c) << "\n";           //输出(c+=d, c)结果
    cout << "(c-=d) c=" << (c-=d, c) << "\n";           //输出(c-=d, c)结果
    cout << "(c<<=b) c=" << (c<<=b, c) << "\n";         //输出(c<<=b, c)结果
    cout << "(c>>=b) c=" << (c>>=b, c) << "\n";         //输出(c>>=b, c)结果
    cout << "(b&=d) b=" << (b&=d, b) << "\n";           //输出(b&=d, b)结果
    cout << "(c|=d) c=" << (c|=d, c) << "\n";           //输出(c|=d, c)结果
    cout << "(b^=c) b=" << (b^=c, b) << "\n";           //输出(b^=c, b)结果
}
```

上面代码依次演示了如何使用各种赋值运算符。其中还用到了逗号运算符，在后面会介绍。代码运行结果如下：

```
(e=d) e=9
(a*=b) a=6
(c/=b) c=2
(a%=f) a=2
(c+=d) c=11
(c-=d) c=2
(c<<=b) c=16
(c>>=b) c=2
(b&=d) b=1
(c|=d) c=11
(b^=c) b=10
```

3.5.3　关系运算符

C++提供了一组二进制关系和相等运算符，用于比较两个操作数之间的指定关系是否成立。如果关系成立，则关系表达式返回 1；否则，关系表达式返回 0。关系表达式的返回值类型为 int 类型。表 3-8 中列出了 C++支持的关系运算符。

表 3-8　关系运算符

关系运算符	功　　能
<	小于运算符。判断左操作数是否小于右操作数
<=	小于等于运算符。判断左操作数是否小于或等于右操作数
>	大于运算符。判断左操作数是否大于右操作数
>=	大于等于运算符。判断左操作数是否大于或等于右操作数
==	等于运算符。判断左操作数是否等于右操作数
!=	不等于运算符。判断左操作数是否不等于右操作数

下面代码显示了关系运算符的使用：

```
void printRelation()                                    //关系运算符示例
{
    int a = 1, b = 2, c = 2, d = 4, e = 5;      //定义整型变量 a、b、c、d、e
    cout << "(a<b)=" << (a<b) << "\n";          //输出(a<b)结果
    cout << "(c<=b)=" << (c<=b) << "\n";        //输出(c<=b)结果
    cout << "(d>e)=" << (d>e) << "\n";          //输出(d>e)结果
    cout << "(e>=d)=" << (e>=d) << "\n";        //输出(e>=d)结果
    cout << "(b==c)=" << (b==c) << "\n";        //输出(b==c)结果
    cout << "(b!=c)=" << (b!=c) << "\n";        //输出(b!=c)结果
}
```

上面代码演示了关系运算符的使用，运行结果如下：

```
(a<b)=1
(c<=b)=1
(d>e)=0
(e>=d)=1
(b==c)=1
(b!=c)=0
```

在使用关系运算符时，要注意等于运算符“==”与赋值运算符“=”的使用，等于运算符是判断两个操作数是否相等，而赋值运算符是将右操作数的值赋值给左操作数。示例代码如下：

```
if (a ==b)       //判断 a 和 b 是否相等
{}
和
if (a = b)       //将 b 赋值给变量 a，并判断变量 a 的值是否大于 0
{}
```

虽然上面两条 if 语句都没有语法错误，但是含义是完全不同的。第一条表示判断变量 a 和变量 b 的值是否相等，第二条语句表示将变量 b 的值赋值给变量 a。有时会由于笔误，

将第一条语句写成第二条语句的形式，这样会导致程序的逻辑错误。

3.5.4　逻辑运算符

C++中有 3 种逻辑运算符，分别是逻辑与、逻辑或和逻辑非。除了逻辑非是一元运算符外，逻辑与和逻辑或都是二元运算符。逻辑与运算符符号为“&&”，用于运算两个操作数之间的与关系。当两个操作数都为非 0 时，则结果值为 1；否则，结果值为 0。运算方向为从左向右。代码如下：

```
void printLogicalAnd()              //逻辑与运算符的示例
{
    int nCount=5, nPrice=2;         //定义数量和单价变量，并分别赋值为 5 和 2
    if ((nCount > 0)&&(nPrice > 0))
        //输出货物总价
        cout << "货物总价" << nCount*nPrice<< "元\n";
    else
        //如果数量或单价为 0，则提示输入的数据无效
        cout << "数据无效" << "\n";
}
```

上面代码使用逻辑与判断货物数量和货物单价的取值是否为有效范围。如果有效，计算货物总价；如果无效，则输出提示信息。运算结果为：

```
货物总价 10 元
```

逻辑或运算符符号为“||”，用于运算两个操作数之间的或关系。当两个操作数中的任何一个操作数为非 0 时，则结果值为 1；否则，结果值为 0。运算方向为从左向右。代码如下：

```
void printLogicalOr()                   //逻辑或运算符的示例
{
    int age=1000;                       //定义年龄变量，并初始化 1000
    if ((age > 120) || (age < 0))       //判断年龄值是否不在合理范围从 0～120
        cout << "年龄值无效" << "\n";
    else
        cout << "年龄值有效" << "\n";//如果年龄值有效，则提示年龄值有效
}
```

上面代码使用逻辑或判断年龄的取值是否为有效范围，此处定义年龄的有效取值范围为 0～120 岁，然后输出提示信息。运算结果为：

```
年龄值无效
```

逻辑非运算符符号为“!”，用于单个操作数的否运算。当操作数为 0 时，结果为 1；否则，结果值为 0。运算结果为 int 类型，此运算符的操作数必须为整数、浮点型或指针类型。代码如下：

```
void printLogicalNot()                  //逻辑非运算符的示例
{
    int balance=5;                      //定义余额变量，并初始化 5
    if (!balance)
```

```
        cout << "账户余额为 0 元" << "\n";     //判断余额是否为 0，并输出
    else
        cout << "账户余额不为 0 元" << "\n";   //余额不为 0，输出结果
}
```

上面代码使用逻辑非判断账户余额是否为 0，然后输出提示信息。运算结果为：

```
账户余额不为 0 元
```

3.5.5　位运算符

C++提供了对位进行操作的运算符，使用位运算符可以减少程序占用的空间，加快程序运行速度。位运算符主要有位与运算符、位或运算符、位异或运算符、位左移运算符、位右移运算符和反码运算符，如表 3-9 所示。

表 3-9　位运算符

位运算符	功　　能
&	位与运算符是二目运算符，比较第一个操作数的每位和第二个操作数相应的位。如果都为 1，则结果值相应的位也为 1；否则，结果值相应的位为 0
|	位或运算符是二目运算符，比较第一个操作数的每位和第二个操作数相应的位。如果其中有一个位为 1，则结果值相应的位也为 1；否则，结果值相应的位为 0
^	位异或运算符是二目运算符，比较第一个操作数的每位和第二个操作数相应的位。如果两个中一个为 0，另一个为 1，则结果值相应的位也为 1；否则，结果值相应的位为 0
<<	左移运算符是二目运算符，结果是将第一个操作数的各位向左移动第二个操作数指定的位数后的值。在移动过程中，原来左边多出的位数丢掉，右边的位数使用 0 补齐
>>	右移运算符是二目运算符，结果是将第一个操作数的各位向右移动第二个操作数指定的位数后的值。在移动过程中，原来右边多出的位数丢掉，左边的位数使用 0 补齐
~	二进制反码运算符是一目运算符，也称为补码或位非运算符，得出操作数的补码。操作数必须是整型类型的，并且运算后的结果值的类型与操作数相同。当操作数的某位为 1 时，结果值对应的位为 0；当操作数的某位为 0 时，结果值对应的位为 1

位运算符的使用，代码如下：

```
void printBitOperator()                          //位运算符示例
{
    int a=3, b=5, result;                        //0000 0011、0000 0101
    result = a & b;                              //位与运算符 0000 0001=1
    cout << "(a & b)=" << result << "\n";        //输出位与运算结果
    result = a | b;                              //位或运算符 0000 0111=7
    cout << "(a | b)=" << result << "\n";        //输出位或运算结果
    result = a ^ b;                              //异或运算符 0000 0101=6
    cout << "(a ^ b)=" << result << "\n";        //输出异或运算结果
    result = a << 2;                             //左移运算符 0000 1100 = 12
    cout << "(a << 2)=" << result << "\n";       //输出左移运算结果
    result = a >> 3;                             //右移运算符 0000 0000 = 0
    cout << "(a >> 3)=" << result << "\n";       //输出右移运算结果
    unsigned short c = 0xBBBB;                   //补码运算符 1011 1011 1011 1011
    c = ~c;                                      //0100 0100 0100 0100 = 17476
```

```
    cout << "(~c)=" << c << "\n";           //输出补码运算结果
}
```

代码运算结果如下：

```
(a & b)=1
(a | b)=7
(a ^ b)=6
(a << 2)=12
(a >> 3)=0
(~c)=17476
```

3.5.6　三目运算符

C++中只有一个三目运算符，即条件运算符（?：）。它是为了简化条件语句的编写而提供的运算符，其语法格式为：

```
逻辑表达式 ? 表达式 : 条件表达式
```

其中，逻辑表达式必须是整型、浮点型或指针类型，其后加上条件运算符，表示运算符会计算逻辑表达式的值是否为 true（即非 0）。如果是 true，则结果值为表达式代表的值；如果是 false（即 0），则结果值取条件表达式的值。这两个也称为结果表达式，加上冒号作为分隔符。无论是表达式还是条件表达式都可以是可计算的表达式，但是这两者不能同时是可计算的表达式。条件运算符的功能等同于如下 if 条件表达式：

```
if (逻辑表达式)
{
    结果 = 表达式;
}
else
{
    结果 =条件表达式;
}
```

如下代码显示了条件运算符的使用方法：

```
void printConditional()                              //条件运算符
{
    int nBalance = 20, nAssign = 1, result;          //定义变量值
    result = (nBalance <= 0) ? 0 : nAssign ;         //判断余额是否为 0
    cout << "result=" << result << "\n";             //输出结果
}
```

上例使用三目条件运算符判断 nBalance 变量的值是否小于等于 0。如果是，则返回结果值为 0；否则，返回结果值为 nAssign 的值 1。上述代码等价于如下 if 条件语句：

```
if(nBalance <= 0)                          //判断 nBalance 变量是否小于等于 0
{
    result = 0;                            //赋值 result 为 0
}
else
{
    result = nAssign;                      //否则赋值 result 为 nAssign 变量的值
}
```

3.5.7　增 1 和减 1 运算符

增 1 运算符会将表达式的值增 1，减 1 运算符会将表达式的值减 1，分别是++和– –。其中，当增 1 或减 1 操作符出现在操作数前面时，称为前增 1 或前减 1 运算符，此时，会先将一元表达式的值增 1 或减 1 后，再使用该值。当增 1 或减 1 操作符出现在操作数后面时，称为后增 1 或后减 1 运算符，此时，会先使用表达式的值，然后将该值增 1 或减 1。后增 1 或后减 1 运算符的优先权比前增 1 或前减 1 运算符的优先权高。

```
++操作数                    //前增 1 运算符
--操作数                    //前减 1 运算符
操作数++                    //后增 1 运算符
操作数--                    //后减 1 运算符
```

无论是前增 1 或前减 1 运算符，还是后增 1 或后减 1 运算符，操作数必须是整型、浮点型或指针类型，并且是可以修改值的表达式。请参看下面的例子。

```
void printBeforeIncrement()         //前增 1 运算符的示例
{
    int a=1, b=2, result;           //定义变量 a、b 和 result
    result = (a) + (++b);           //result 现在的取值为 4
    cout << result << "\n";         //输出 result 值
}
void printBeforeDecrement()         //后增 1 运算符的示例
{
    int a=1, b=2, result;           //定义变量 a、b 和 result
    result = (a) + (b++);           //result 现在的取值为 3
    cout << result << "\n";         //输出 result 值
}
```

在上例中，printBeforeIncrement()函数中使用前增 1 运算符，会将 b 的取值增 1 变成 3 后再与 a 的值 1 相加，即结果为 4；而 printBeforeDecrement()函数中使用后增 1 运算符，会先将 b 的取值 2 和 a 的取值 1 相加后，再将 b 的取值增 1，即结果为 3。程序运行结果为：

```
4
3
```

3.5.8　逗号运算符

逗号运算符即连续赋值运算符，符号为逗号（,）。它是二元运算符，从左向右计算两个操作数。逗号运算符左边的操作数表示空表达式。运算的结果值与右操作符具有相同的类型。每个操作数可以是任何类型的数据。逗号运算符不能在操作数之间完成类型转换，它不能处理左值。在第一个操作数后有个顺序点，表示左操作数的所有求值会在右操作数求值开始之前完成。

逗号运算符通常用在上下文环境中只允许一个表达式时，计算两个或多个表达式。在有些情况下，它可以作为分隔符使用。所以要注意，逗号作为分隔符使用和作为逗号运算符使用是完全不同的。例如代码如下：

```
void TestFunction(int x, int y, int z)
{
    cout << "x=" << x << "\n";      //输出 x 值
    cout << "y=" << y << "\n";      //输出 y 值
    cout << "z=" << z << "\n";      //输出 z 值
}
void printComma()                   //打印运算结果值函数
{
    int a = 50, b = 0, c = 99;      //分别定义变量 a、b 和 c，这里使用逗号运算符
                                    //作为分隔符
    TestFunction(a, (b=47, b-7), c);   //输出 a、(b=47, b-7)和 c 的值
}
int main(int argc, char* argv[])    //程序主函数
{
    printComma();                   //执行 printComma()函数
    return 0;                       //返回
}
```

在上面代码中，TestFunction()函数具有 3 个参数。在 printComma()中调用 TestFunction()函数打印 3 个值。其中，第二个参数使用了逗号运算符，因此会顺序执行 b=47 语句和 b–7 语句，并将运算结果作为第二个参数传入。运算结果如下：

```
x=50
y=40
z=99
```

3.5.9　sizeof 运算符

sizeof 关键字用于计算表达式表示的变量或类型的存储字节数。此关键字返回一个 size_t 类型的值，计算的表达式是标识符或类型转换表达式。当计算结构类型或变量时，sizeof 返回实际大小，其中也包括为对齐而插入的字节。当计算静态数组的大小时，sizeof 返回整个数组的大小。sizeof 运算符不能返回动态分配的数组或外部数组的大小。例如代码如下：

```
struct align_struct
{
   char ch;                                 //字符型
   int i;                                   //整型
};
void printSizeof()                          //sizeof 示例
{
    cout << "sizeof(int)=" << sizeof( int ) << "\n";
                                            //输出 int 类型的长度
    cout << "sizeof(align_struct)=" << sizeof( align_struct ) << "\n";
                                            //输出 align_struct 类型的长度
    int  array[] = { 11, 22, 33, 44 }; //定义整型数组
    cout << "sizeof( array )=" << sizeof( array ) << "\n";
                                            //输出数组长度
    cout << "sizeof( array[0] )=" << sizeof( array[0] ) << "\n";
                                            //输出数组元素长度
    cout << "count=" << sizeof( array ) / sizeof( array[0] ) << "\n";
                                            //输出数组个数
```

```
}
```

在上面代码中，首先输出 int 类型的大小，然后输出自定义结构 align_struct 的大小，接着输出定义的整个数组的大小和单个数组元素的大小，最后根据整个数组大小除以单个数组元素大小的方式计算数组个数，此方法是比较常用的计算数组包含的元素个数的方法。运行结果如下：

```
sizeof(int)=4
sizeof(align_struct)=8
sizeof( array )=16
sizeof( array[0] )=4
count=4
```

注意：其中的 sizeof(align_struct)的运行结果与/Zp 选项有关，所以根据用户的设置不同，可能其结果值与此处不相同。

3.5.10　new 和 delete

new 关键字用于为指定类型的对象从空闲存储区中分配内存，并返回指向对象的对应类型的指针。如果失败，则 new 操作返回 0。用户可以通过编写自定义异常处理程序修改默认的操作，并将函数名作为参数调用运行时库函数_set_new_handler。其语法格式为：

```
[::] new [定位符] 类型名称 [初始值]
[::] new [定位符] (类型名称) [初始值]
```

如果重写对象的 new 方法，则定位符可以用于传递外加的参数。类型名称指定要分配的空间存储的变量的类型。如果类型是复杂类型，则可以通过使用括号强制绑定顺序。初始值用于提供初始化对象使用的值。不能为数组指定初始值，只有当类具有默认的构造函数时，new 操作符才可以创建数组对象。请参看下面的代码：

```
int *pint = new int;                    //创建指向 int 类型的指针
char *pchar = new char( 'X' );          //创建指向 char 类型的指针
Date *pdate = new Date( 2, 18, 2013 ); //创建指向 Date 类型的指针
char *pstr = new char[sizeof( str )];  //创建指向字符数组的类型的指针
char (*pchar)[10] = new char[x][10];   //创建指向二维字符数组的类型的指针
```

在上面代码中，第一条语句为整型分配了内存。第二条语句为字符类型分配了存储空间，并初始化为字符 X。第三条语句为日期类型分配了存储空间，使用类的带有 3 个参数的构造函数为其赋值为 2013 年 2 月 18 日，并将结果返回给日期类型的指针。第四条语句为字符数组分配存储空间，并将其指针分配给一个字符指针。第五条语句为大小为 x*10 的二维数组分配存储空间。当为多维数组分配空间时，除了第一维的维数外，其他维数必须为常数表达式。

如果使用没有任何外部参数的运算符，当构造函数抛出异常错误时，编译器会生成调用 delete 操作符的代码。如果使用占位符格式的 new 运算符或使用带外部参数的 new 运算符，当构造函数抛出异常错误，编译器不会生成调用 delete 操作符的代码。因此，此种情况下的异常发生后的内存释放工作应该由开发人员人工处理，否则就会出现 C++的典型程

序漏洞——内存泄露。代码如下：

```
MyClass* p1 = new MyClass(99);      //p1 指向的堆栈内存会被 delete 释放
//此调用，因为使用了带定位符的构造函数，因此，会产生内存泄露
MyClass* p2 = new(__FILE__, __LINE__) MyClass(99); //创建 MyClass 类
```

使用 delete 运算符可以释放由 new 运算符分配的内存空间。其语法格式为：

```
[::] delete 指针
[::] delete [ ] 指针
```

delete 关键字释放内存块。上面的指针参数必须是先前由 new 操作符分配的内存块的指针。如果指针指向数组，则当调用 delete 时，需要在指针前加上一对空中括号。代码如下：

```
delete pint;        //删除指向整型的指针
delete pchar;       //删除指向字符型的指针
delete pdate;       //删除指向日期型的指针
delete pstr[];      //删除指向字符串型的指针
delete pchar[];     //删除指向二维字符数组型的指针
```

上面代码依次释放前面申请分配的存储空间。

3.5.11　范围确定符

在 C++中，在变量名前加上范围确定符“::”前缀，即两个冒号，可以告诉编译器使用全局变量而不使用本地变量。即使代码上下文环境是嵌套的本地作用域，范围确定符也不能提供访问下个外层范围的变量，只能提供对全局变量的访问。示例代码如下：

```
#include <iostream.h>                                   //范围确定符的示例
int pages = 800;                                        //全局变量
void printPages()
{
    int pages = 100;                                    //本地变量
    cout << "全局变量 pages=" << ::pages<< '\n'; //打印全局变量
    cout << "本地变量 pages=" << pages<< '\n';   //打印本地变量
}
int main()                                              //程序主函数
{
    printPages();                                       //调用 printPages()函数
    return 0;                                           //返回
}
```

在上面的例子中，有两个名为 pages 的变量。第一个是全局变量，值为 800 页。第二个为 printPages()函数的本地变量。两个冒号告诉编译器使用全局 pages 变量而不是本地 pages 变量。运行结果如下：

```
全局变量 pages=800
本地变量 pages=100
```

3.5.12　类成员访问符

C++提供访问结构体、联合体和类的成员的操作符（.和->），其语法格式为：

```
表达式.成员选择标识符              //第一种方式
表达式->成员选择标识符             //第二种方式
```

其中，表达式表示类型为结构体、联合体或类等复合对象的变量或指针。第一种方式中，表达式代表的是对象变量；第二种方式中，表达式代表的是指针。成员选择标识符表示表达式类型的成员函数名称。操作的结果值是标识符的返回值。

实际上，如果表达式在前面包含间接访问操作符“*”应用的指针值，使用“->”成员选择符的表达式是使用“.”成员选择符的表达式的简写版本。因此，当表达式是指针类型时，下面两种成员选择表达式的作用是相同的：

```
表达式->成员选择标识符                //指针成员访问符
(*表达式).成员选择标识符              //对象成员访问符
```

具体使用代码如下：

```
struct MyDate {                    //成员选择操作符示例
   int  nYear;                     //年
   int  nMonth;                    //月
   int  nDay;                      //日
};
void printMemberSelect()           //打印选择的成员函数
{
    MyDate tmpDates;               //定义 MyDate 结构的变量
    tmpDates.nYear = 2008;         //为对象的 nYear 变量赋值
    cout << "年=" << (&tmpDates)->nYear << "\n";   //输出 nYear 变量的值
}
```

在上面的代码中，tmpDates.nYear 与(&tmpDates)->nYear 表达的含义是完全相同的，区别在于一个是用于赋值，一个是用于取值。其运行结果如下：

```
年=2008
```

3.5.13　成员指针操作符

C++提供成员指针操作符.和->来访问类的成员指针。这两个操作符都是二元操作符，组合第一个操作数和第二个操作数访问类成员，其语法格式为：

```
类表达式. 成员表达式                //第一种情况
类表达式-> 成员表达式               //第二种情况
```

- 第一种情况：类表达式必须是类类型的对象，成员表达式必须是成员指针类型。
- 第二种情况：类表达式必须是指向类类型对象的指针，成员表达式必须是成员指针类型。

下面是使用这两种成员操作符的示例，代码如下：

```
void printClassAccess()                    //成员访问符示例
{
    CMyDate myDate1;                       //定义 CMyDate 类型的变量
    myDate1.nDay = 17;                     //为 myDate1 变量的 nDay 成员赋值为 17
    CMyDate* myDate2 = new CMyDate();      //定义 CMyDate 类型的指针变量
    myDate2->nDay = 18;                    //为 myDate2 变量的 nDay 成员赋值为 18
    //输出变量的值
    cout << "myDate1.nDay=" << myDate1.nDay << ";myDate2->nDay="
    << myDate2->nDay << ".";
}
```

3.6 控 制 语 句

所有的编程语言都是通过语句执行指令，而现实世界中语句间存在 3 种逻辑结构：选择结构、循环结构和跳转结构。所以，在 C++语言中提供了表达式语句、选择语句、循环语句和跳转语句。本节将介绍 C++中有关控制语句的语法。

3.6.1 表达式语句、空语句和复合语句

表达式语句可以计算表达式的值，结果不会控制程序的跳转，也不会重复执行。在表达式语句中的所有表达式都会被计算，并且在执行完下条语句前会完成计算。常用的表达式有赋值语句和函数调用。如下代码所示，其中两条语句都是表达式语句。

```
int z = 2*3 + 5;
int a = min(x,y);
```

C++也支持空语句，就是一个没有任何表达式的表达式语句，由一个分号构成，常用于在重复语句中作占位符，或是在复合语句或函数的结尾处放置标签的语句。下面的代码显示了如何使用空语句。

```
char *strcpy( char *Dest, const char *Source ) //复制字符串函数
{
    char *DestStart = Dest;                //定义字符指针，指向 Dest
    while( *Dest++ = *Source++ )           //将数据从源字符串中复制到目的字符串中，
                                           //直到到达字符串尾
   ;                                       //空语句
   return DestStart;                       //返回结果字符指针
}
```

复合语句也称为代码段（程序块），由放在一对大括号中的 0 个或多个语句组成。复合语句可以用在任何可以使用单条语句的地方。在使用复合语句时，要注意在复合语句中的声明语句，其作用范围仅在定义的复合语句中有效。例如，if 语句后的一对大括号中包含的就是复合语句，其中的变量 a 仅在 if 语句的复合语句中有效。例如代码如下：

```
if( Amount > 100 )                 //判断 Amount 变量的值是否大于 100
{
    int a = 30;                    //定义整型变量 a 的值为 30
    cout << a;                     //输出 a 的值
```

```
}
else
    Balance -= Amount;          //在余额中减去 Amount 的值
```

3.6.2　选择语句

选择语句提供了有条件地执行代码段的方法。C++中提供了两种选择语句：if 语句和 switch 语句。

1．if语句

if 语句根据求得的括号内表达式的值确定执行的代码块。其语法格式为：

```
if ( 表达式 )  语句 1
if ( 表达式 )  语句 1  else 语句 2
```

其中，表达式必须是算术类型或指针类型，或者是明确定义了转换成算术或指针类型的方法的类。当表达式的计算结果为非 0 值，即 true 时，执行语句 1 代码段；否则，跳过或执行语句 2 代码段，如图 3-5 所示。

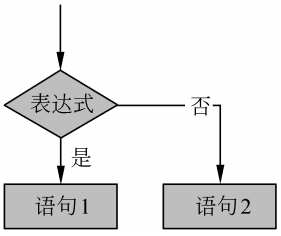

图 3-5　if 语句的执行流程图

当 if 语句中出现嵌套时，if…else 语句中的 else 语句与前面最近的还没有相应 else 语句的 if 语句进行匹配。如下代码显示了 if 语句的嵌套情况。

```
if( condition1 == true )
    if( condition2 == true )
        cout << "条件 1 为真; 条件 2 为真\n";
    else
        cout << "条件 1 为真; 条件 2 为假\n";
else
   cout << "条件 1 为假\n";
```

虽然，使用 if…else 语句的嵌套规则可以确定配对情况，但是建议在编写程序时，养成良好的编程习惯，使用大括号{}将 if 和 else 子句括起来，这样，可以清晰地显示 if 条件语句的逻辑关系。以下是将上面的代码用大括号改写后的代码。

```
if( condition1 == true )                //如果条件 1 为 true
{
    if( condition2 == true )            //如果条件 2 为 true
    {
        cout << "条件 1 为真; 条件 2 为真\n";
    }
    else                                //如果条件 2 不为 true
    {
        cout << "条件 1 为真; 条件 2 为假\n ";
    }
}
else                                    //如果条件 1 不为 true
{
    cout << "条件 1 为假\n";
}
```

2．switch语句

switch 语句根据表达式的取值在多个代码段中选择要执行的代码段。语句流程如图 3-6 所示。

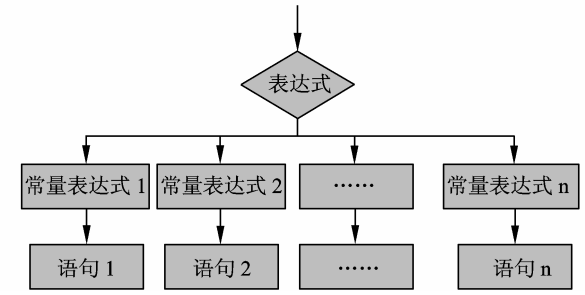

图 3-6　switch 语句流程图

从图 3-6 中可以看出，switch 语句会判断表达式的值，依次判断与各个常量表达式是否相同，与哪个常量表达式相同，就执行相应的语句。其语法格式为：

```
switch ( 表达式 )     //判断表达式的值
{
case 常量表达式 1:     //判断表达式的值是否与常量表达式 1 相同，如果是，则执行语句 1
        语句 1
        break;
case 常量表达式 2:     //判断表达式的值是否与常量表达式 2 相同，如果是，则执行语句 2
        语句 2
        break;
case 常量表达式 n:     //判断表达式的值是否与常量表达式 n 相同，如果是，则执行语句 n
        语句 n
        break;
default:
        语句 n+1
}
```

其中控制代码执行的表达式的值必须使用括号括起来，并且它必须是一个可整型化的类型或者是一个可以明确转换成整型化的类。也就是说，表达式的值必须可以“比较”。switch 语句体根据控制表达式的值、case 标签的值和是否存在 default 标签，确定是无条件地跳转到 switch 语句体还是略过 switch 语句体。其中，case 标签和 default 标签都是可以忽略的。case 标签可以定义多条，每条标签后的取值不能有重复，而 default 是默认情况下要执行的代码，所以最多定义一次。表 3-10 列出了 switch 语句在各种情况下的执行方式。

表 3-10　switch语句的执行方式

条　　件	执 行 方 式
如果转换的 case 标签值与控制表达式的值匹配	程序跳转到 case 标签后的代码段
没有 case 标签值与控制表达式的值匹配，但是 default 标签存在	程序跳转到 default 标签后的代码段
没有 case 标签值与控制表达式的值匹配，同时 default 标签也不存在	程序跳转到 switch 语句后的代码段

在 switch 语句的可能执行的地方，也就是所有可能执行的路径的地方，可以使用带初

始化的定义。在这些地方声明的定义只具有本地作用域，即只能在定义的范围内使用。代码如下：

```
switch( tolower( *argv[1] ) )
{
    //此处程序不可能执行到，声明是无效的
    char szInput[] = "Please Enter Command: ";

    case 'x' :
    {
        //szInput[]的声明有效，是本地范围的变量
        char szInput[] = "Please Enter Command: ";
        cout << szInput << "x\n";  //输出 szInput 数组的值
    }
        break;
    case 'y' :
        cout << szInput << "y\n";  //szInput 未定义
        break;
    default:
       cout << szInput << "既不是 x 也不是 y\n";   //szInput 未定义
       break;
}
```

在上面的代码中，case 'x' 标签下定义的 szInput 变量在本 case 标签下是有效的；第二条 case 标签和 default 标签下的代码段中，调用 szInput 是错误的，因为已经超过了 szInput 的作用范围。

switch 语句与 if 语句一样，也是支持嵌套的，并且 case 标签和 default 标签也是与最近的 switch 语句匹配的。下面的代码是 Windows 消息循环的处理代码段。switch 语句首先判断 msg 的类型，只有当它为 WM_COMMAND 时，才会根据 wParam 参数处理命令，而其中又使用 switch 语句判断命令类型，分别处理 IDM_F_NEW 和 IDM_F_OPEN 命令。代码如下：

```
switch ( msg )              //判断消息类型，进行分类处理
{
case WM_COMMAND:            //Windows 命令，处理多个命令
   switch ( wParam )
   {
   case IDM_F_NEW:          //“新建”菜单命令
      break;
   case IDM_F_OPEN:         //“打开”菜单命令
      break;
   ...                      //此处代码省略
   }
case WM_CREATE:             //创建窗体
   ...                      //此处代码省略
   break;
case WM_PAINT:              //窗体需要重绘
   ...                      //此处代码省略
   break;
default:
   return DefWindowProc( hWnd, Message, wParam, lParam );
                            //处理对话框处理过程
}
```

switch 语句不会在 case 标签和 default 标签中终止语句的运行，要停止 switch 语句的继续运行，则需要在适当的位置加入 break 语句。程序在执行到 break 语句时，会跳转到 switch 语句后的代码。下面的代码说明了 break 语句的使用。

```
BOOL fClosing = false;                    //定义是否关闭对话框的变量
...                                       //此处代码省略
switch ( wParam )                         //判断消息的 wParam 参数
{
case IDM_F_CLOSE:                         //“关闭”菜单命令
   fClosing = true;
case IDM_F_SAVE:                          //“保存”菜单命令
   if( document->IsDirty() )              //判断文档是否修改过
      if( document->Name() == "UNTITLED" )
                                          //如果文档是未命名的新文档，则另存为
         FileSaveAs( document );          //另存文档
      else
         FileSave( document );            //保存文档
   if( fClosing )
      document->Close();  //如果要关闭文档
   break;
}
```

在上面的代码中，当执行 IDM_F_SAVE 命令时，fClosing 为 false，则程序会保存文件，但不会关闭文件；当执行 IDM_F_CLOSE 命令时，程序先将 fClosing 赋值为 true，然后继续执行到 IDM_F_SAVE 标签，保存文档后，关闭文档。这样巧妙的设计，既按照设计完成了功能，又减少了重复代码的工作量。所以在实际编程时，应该巧妙地利用语法原有的特性。

3.6.3　循环语句

循环语句，又称重复语句。它会使语句执行 0 次或多次，直到循环条件不满足。如果循环执行的语句是复合语句，则会按照顺序执行，除非在语句中使用 break 或 continue 等跳转语句，跳转语句在 3.6.4 小节会介绍。

C++中有 3 种循环语句，分别是 while、do 和 for。每种循环语句会重复执行语句，直到终止表达式计算的结果为 false 或者遇到 break 语句强制终止。其中，终止表达式必须是整型化的类型或者是可以明确地转化为整型化的类的表达式，而重复语句的执行语句部分不能是声明语句，但是可以是包含声明语句的复合语句。同时，循环语句也是支持嵌套的，嵌套规则与选择语句相同。循环语句的执行方式如表 3-11 所示。

表 3-11　C++ 循环语句的执行方式

语　　句	何 时 求 值	初　始　化	增　　量
while	循环开始	无	无
do	循环结束	无	无
for	循环开始	有	有

1．while循环语句

while 循环语句的语法格式为：

```
while ( 表达式 ) 语句
```

while 语句是在每次执行语句前判断终止条件是否为 true。如果为 false，则退出循环语句；如果为 true，则继续执行循环语句，因此 while 循环有可能执行多次，也有可能一次也不执行。其流程图如图 3-7 所示。

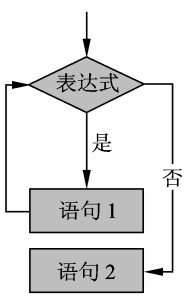

图 3-7　while 语句的执行流程图

从图 3-7 中可以看出，程序首先判断 while 后的表达式的值是否为 true，如果为 true，则执行语句 1，执行完后会继续判断表达式，直到表达式的值为 false，则会执行语句 2。如下代码显示了 while 语句的使用。

```
char *trim( char *szSource )                          //去掉字符串头尾的内容
{
   char *pszEOS;                                      //定义字符指针
   pszEOS = szSource + strlen( szSource ) - 1;        //设置开始指针在字符串尾
   while( pszEOS >= szSource && *pszEOS == ' ' )      //循环处理字符串中的字符
      *pszEOS-- = '\0';                               //去掉空格字符
   return szSource;                                   //返回处理后的字符串
}
```

上面代码的功能是去掉字符串尾部的空格。代码首先将指针指向要处理的字符串的最后一个字符的地址处，然后开始执行循环，去掉尾部的空格，直到指针到达字符串头，或者最后一个字符不是空格，即退出循环。对于此函数，如果字符串尾部没有空格，则循环会退出。

2．do循环语句

do 循环语句的语法格式如下：

```
do 语句 while (表达式) ;
```

do 语句重复的执行语句，直到指定的终止条件表达式计算结果为 0 时，即退出循环。终止条件的判断是在每次循环语句执行完一次循环后执行，因此 do 循环语句至少要执行一次。其流程图如图 3-8 所示。

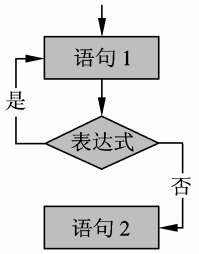

图 3-8　do 语句的执行流程图

从图 3-8 中可以看出，程序首先执行语句 1，然后判断 while 后的表达式的值是否为 true。如果为 true，则继续执行语句 1；否则会执行语句 2。如下代码显示了 do 语句的使用。

```
void WaitKey( char ASCIICode )      //等待用户输入指定字符的函数
{
   char chTemp;                     //定义字符变量，用于存放用户输入的字符
   do
```

```
    {
        chTemp = _getch();              //接收用户输入的字符
    }
    while( chTemp != ASCIICode );       //循环条件为判断接收到的字符是否为指定字符
}
```

上面的函数实现了等待用户按下指定按键的功能。它会提示用户输入字符，直到输入的字符与要求的字符相同，则会退出 do 循环。此功能也可以使用 while 循环语句实现，但是实现起来就比较繁琐，没有 do 循环语句简明。可以看出，在至少需要执行一次的循环语句中使用 do 循环语句比较适合，否则还是使用 while 语句比较好。如下代码使用 while 语句改写了上面的代码。

```
void WaitKey( char ASCIICode )      //等待用户输入指定字符的函数
{
    char chTemp;                    //定义字符变量，用于存放用户输入的字符
    chTemp = _getch();              //接收用户输入的字符
    while( chTemp != ASCIICode )    //循环条件为判断接收到的字符是否为指定字符
    {
        chTemp = _getch();          //接收用户输入的字符
    }
}
```

3．for循环语句

for 循环语句可以分为 3 部分，其语法格式如下：

```
for （表达式 1; 表达式 2; 表达式 3）语句
```

其中，表达式 1 部分用于初始化循环参数，是在执行 for 循环语句前，首先执行的语句。表达式 2 部分是整型化的表达式或可以转换成整型化的类。在每次执行循环语句前，判断表达式 2 的值是否为 true，决定是否继续执行。表达式 3 部分用于处理循环计数，每次执行循环语句完成后执行此部分，执行完后进入下一次循环。如图 3-9 显示了 for 循环的执行流程。

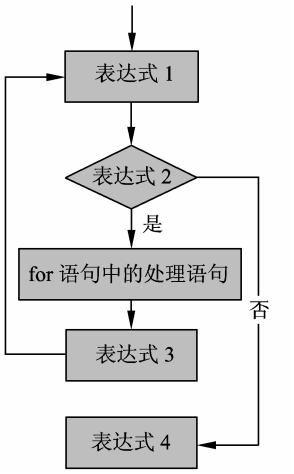

图 3-9　for 语句的执行流程图

for 初始化语句通常用于声明和初始化循环索引变量，判断语句用于测试循环终止条件，循环处理计数语句用于增加循环计数。在 for 循环中，这 3 条语句任何一条都是可选的。其中，for 初始化语句可以是声明语句或表达式语句，也可以是空语句，并且可以包含多条，其间以逗号分开。注意任何在 for 初始化语句中声明的对象，都是在 for 本地作用域内有效。因为 for 语句和 while 语句都是循环语句，因此，二者之间是可以互相转换的。如下面的 for 语句与 while 语句就具有相同的作用。

```
for(表达式 1; 表达式 2; 表达式 3)
{
    //语句
}
```

与

```
表达式 1;
while(表达式 2)
{
    //语句
    表达式 3;
}
```

而下面的 for 循环语句与 while 循环语句都能够完成无限循环的功能。

```
for( ; ; )
{
    //要执行的语句
}
```

与

```
while( 1 )
{
    //要执行的语句
}
```

虽然通常情况下，for 语句的 3 个语句分别用于初始化、测试终止条件和递增计数，但是也可以不这样用，如下代码，作用是打印 1～100 之间的整数，其 for 语句的执行语句为空语句，在 for 语句的表达式语句中完成打印。

```
void main()
{
    for( int i = 0; i < 100; cout << ++i << endl )      //执行 for 循环
        ;
}
```

3.6.4　跳转语句

跳转语句控制代码直接跳转到指定的代码处。C++中有 4 条跳转语句，分别是 break 语句、continue 语句、return 语句和 goto 语句。

1．break语句

break 语句用于退出循环语句或 switch 选择语句，它会控制程序直接跳转到紧跟在循环语句或 switch 语句后的代码语句上。break 语句仅会终止离它最近的循环语句或 switch 语句。在循环语句中，break 语句用于提前退出循环语句。而在 switch 语句中，break 语句用于终止代码段，通常用在 case 标签前。如下代码显示了如何在 for 循环语句中使用 break 语句终止循环语句的执行。

```
for( ; ; )                  //没有终止条件
{
    if( List->AtEnd() )
        break;              //判断是否到达链表尾，如果到达链表结尾，则退出 for 循环
    List->Next();           //取下一个链表元素
}
```

2．continue语句

continue 语句强制代码跳转到离它最近的循环继续语句中，即强制开始一次新的循环。因此，continue 语句只能依赖于循环语句。在 for 循环语句中，执行 continue 语句，也就是执行一次“表达式 2”，然后执行“表达式 3”。

```
//获取输入的合法字符在字符串中的位置
int GetLegalChar( char *szLegalString )
{
    char *pch;                    //定义指向字符的指针
    do
    {
        char ch = _getch();       //获取用户输入的字符
         //判断输入的字符是否在传入的字符串中存在
        if( (pch = strchr( szLegalString, ch )) == NULL )
            continue;             //如果没有则使用 continue 语句，重新执行循环
        return (pch - szLegalString);//返回合法字符在字符串中的位置
    } while( 1 );                 //执行循环条件
    return 0;                     //返回
}
```

3．return语句

return 语句使函数直接返回到调用它的函数中去，对于主函数来说，return 语句会使程序退出。可以使用表达式，将 return 值传回给调用函数。但是对于函数类型为 void，构造函数和析构函数来说，不能通过 return 语句指定返回表达式。其他类型的函数，必须使用 return 语句指定返回值。在一个函数中可以多次使用 return 语句。如果指定了表达式，则会转换成函数声明的返回值类型。

4．goto语句

goto 语句可以无条件地将程序跳转到标签标识的代码段中。要使用 goto 语句，必须在要跳转到的代码前使用标签标注出来。声明方法为在程序源代码前使用标识符，在标识符后加一个冒号。这样，在代码中就可以使用 goto 加标识符，从而直接跳转到指定的代码部分。如下代码所示。

```
for( p = 0; p < NUM_PATHS; ++p )        //使用 for 循环处理文件
{
    NumFiles = FillArray( pFileArray, pszFNames )  //使用文件填充数组
    for( i = 0; i < NumFiles; ++i )     //使用 for 循环打开文件
    {
        if( (pFileArray[i] = fopen( pszFNames[i], "r" )) == NULL )
                                        //打开文件
            goto FileOpenError;         //打开文件失败跳转到 FileOpenError 处
                                        //处理打开的文件
    }
}
FileOpenError:
    cerr << "打开文件错误，中断处理。\n" );
```

在上面的代码中，当打开文件失败时，程序会直接跳转到最后的 FileOpenError 标签的

代码处，打印出错误信息。标签不能单独出现，其后必须要加上语句，如果没有需要处理的语句，则可以加上一条空语句。标签必须在当前函数内，作用范围也仅限于当前函数。它在同一函数内不能重复声明，但是在不同的函数内可以使用相同的标签名称。

3.7　函　　数

函数是 C++语言的核心组成部分，是完成一定功能的语句和变量的组合。根据函数作用的范围和功能的不同，函数分为很多种，如全局函数、类的构造函数、类的析构函数和内联函数等。本节将主要介绍函数的使用方法及需要注意的问题。

3.7.1　函数的定义和调用

函数是完成特定功能的代码段，需要使用函数名称标识代码段，可以不使用参数，也可以使用多个给定类型的参数；返回值可以是指定类型也可以不返回任何类型，此时返回值为 void。函数定义的语法为：

```
可选的说明符 函数名称 (参数列表) 其他函数限定符
{
    //函数体
}
```

其中，可以在可选的说明符中指定函数的返回类型。函数名称指定了函数的名字，不能与相同作用域中的其他标识符重名。参数列表是可选部分，函数可以不指定任何参数，参数列表中各个参数之间使用逗号分隔开，每个参数需要指定参数的类型。其他函数限定符可以是 const 关键字或 volatile 关键字，分别表示常量函数和变化函数。需要注意的是，函数在定义之前，需要先声明，声明的语法就是上面中的第一行，并在行尾加上分号即可。以下代码是一个函数定义的示例。

```
int Add(int a, int b )          //加法函数定义
{
    return a+b;                 //返回两个参数的和
}
```

上面的 Add()函数定义了实现将两数相加的功能，其中 Add 为函数名称。第一个 int 表示函数的返回类型为整型 int。小括号里面表示此函数具有两个参数，都是 int，分别为参数 a 和参数 b。在函数体中，编写代码返回整型参数 a 和整型参数 b 的和。

函数调用是调用函数执行的表达式，其语法为：

```
函数标识符 ([参数列表])
```

其中，函数标识符是指要调用的函数名称或是指向函数的指针值。参数列表是函数调用中的可选部分，是传入函数的参数值的表达式集合。当忽略参数列表时，表示函数的参数部分为空。参数列表可以包含一个或多个参数，各个参数之间用逗号分隔开，传入的参数类型必须与函数定义中相应位置的参数类型相同。

函数调用表达式具有返回值，类型与函数的返回值类型相同。函数不能返回数组类型

的对象。如果函数的返回值类型为 void，也就是说，函数声明为不返回值的形式，则函数调用表达式也具有 void 类型。

```
int c = Add(2 , 3); //函数调用示例
```

上面代码表示创建整型变量 c，存储整型值 2 和整型值 3 的和。

3.7.2　带默认形参值的函数

有时候，函数会具有一些参数，这些参数通常情况下取值是确定的，只需要在特殊情况下修改传入的参数。这时，就需要使用带默认形参值的函数。使用参数默认值，必须保证指定的默认参数值是有效取值。其语法格式为：

```
可选的说明符 函数名称 (参数列表，参数类型 形参名称=形参值，...) 其他函数限定符
{
    //函数体
}
```

带默认值参数的函数的定义与普通函数的定义方式是相同的。区别在于，需要在函数定义中具有默认值的参数后加上等号（=）及其默认值。以下代码为带默认值参数的函数的定义方法。

```
int AddYear(int old, int increment = 1)              //为时间值增加一年
{
    return old + increment;
}
```

上面的代码定义了 AddYear()函数，其返回值为 int，表示处理后的年份的取值。此函数有两个参数，第一个参数是要进行年处理的原来的年份值，第二个参数为要在基础年的基础上增加的年数，由于默认情况下每次增加一年，所以，此参数的默认值为 1。

调用带默认参数值的函数的方法与调用普通函数的方法相同。区别在于，它可以使用两种方式调用函数，一种方式是按照参数列表中的参数依次输入各个参数；另一种方式是只传入普通参数，不为带有默认值的参数赋值，此种情况下，对应参数会采用参数的默认值进行处理。代码如下：

```
//带默认参数的函数调用示例，不使用默认参数值，结果 c=2010
int c = AddYear(2008 , 2);
//带默认参数的函数调用示例，使用默认参数值，结果 d=2009
int d = AddYear(2008);
```

上面的代码显示了如何使用带默认值参数的函数的用法，第一种调用方式不使用默认参数值，函数返回结果是两个参数都起作用的结果，结果为 2010；第二种调用方式使用参数默认值，函数返回结果为第一个参数值与第二个参数的默认值的计算结果，结果为 2009。使用带默认形参值的函数需要注意以下几个问题。

- 带有默认值的参数必须放在函数参数列表中的结尾处，并且在调用函数时，传入的参数值，也是从左到右赋值的。以下代码就是不可用的：

```
int AddYear(int old=2008, int increment){}
```

- 带默认参数值的函数，其中的参数默认值不仅可以是常量，还可以指定参数表达式，如可以使用函数返回的值。下例是创建滚动条的函数声明，使用 Win32 API 函数 GetSystemMetrics()函数来获取高度值。

```
BOOL CreateHScrollBar(HWND hWnd,  short nHeight=GetSystemMetrics(SM_
CYHSCROLL ) );
```

3.7.3　函数的递归调用

递归函数是指在函数内部调用函数自身的函数。理解递归函数最好的例子就是使用阶乘的概念，阶乘就是计算从 1 到给定数之间的所有整数的乘积。递归是一种非常重要的技术，但是递归的使用要尤其注意不能出现无限递归的情况。当函数无法获取确定的结果，或者无法计算到结束点时，就会出现无限递归，从而导致无限循环。当出现这种情况时，程序的逻辑结构一定是有问题的。因此，在设计递归函数时要尤其小心。下面是实现阶乘计算函数的代码。

```
01  #include <iostream>
02  using namespace std;
03
04  long factorial(int number)      //递归函数的调用，功能是计算指定整数的阶乘
05  {
06      if (number < 0)
07          return -1;      //如果传入的参数小于 0，则无法计算阶乘
08      if ((number == 0) || (number == 1))
09          return 1;       //如果传入的参数等于 0 或 1，则阶乘为 1
10      else                //否则，递归调用，直到计算完成
11          return (number * factorial(number - 1));   //递归调用阶乘函数
12  }
13  void printRecuFunction()                            //使用递归函数的示例
14  {
15      int a;                                           //定义整型变量 a
16      cout << "递归示例，输入要计算阶乘的数：";          //输出提示信息
17      cin >> a;                                        //接收用户输入
18      cout << a << "的阶乘 " << factorial(a) << "\n";//输出计算的阶乘结果
19
20  }
21  int main()                                           //程序入口函数
22  {
23      printRecuFunction();               //执行 printRecuFunction()函数
24      getchar();                         //阻止控制台程序自动退出
25      getchar();
26      return 0;                          //返回
27  }
```

在上面代码中，计算阶乘值的 factorial()函数就是一个递归函数，因为，在函数中调用了函数本身 factorial，用于与计算比当前函数值小 1 的数的阶乘，直到计算到 1 的阶乘。此处需注意，在 factorial()函数中，首先判断了传入的参数是否为负数，如果是负数，则返回错误。这是因为如果不判断参数是否为负数，则计算–5 的阶乘时，会首先计算–6 的阶乘，而计算–6 的阶乘时，首先计算–7 的阶乘……这样下去，就会进入无限递归的情况，程序永

远执行不完。运行结果如图 3-10 所示。

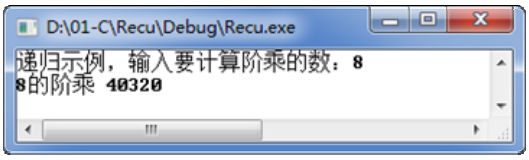

图 3-10　递归函数运行结果

上面的结果显示了如何调用计算阶乘的函数，当输入要计算的值 8 时，计算结果为 40320。

3.7.4　内联函数

inline 关键字用于指定内联函数，告诉编译器使用函数体的代码代替函数调用部分。此种替换方式也就是“内联展开”，也称为内联。内联展开通过增加代码大小来减轻函数调用的系统开销。inline 关键字告诉编译器优先进行内联展开。然而，编译器可以创建函数的单独实例，并创建标准的调用链接代替插入内联代码，从而适当减少内联的代码量。以下代码是内联函数的例子。

```
inline int max( int a , int b )          //取两个值中较大的一个
{
    if( a > b )
        return a;                        //如果 a 大于 b，则返回 a 的值
    else
        return b;                        //否则，返回 b 的值
}
```

上面的代码是取两个数中较大的一个的函数，此函数通过 inline 关键字定义为内联函数。除了可以指定一般函数为内联函数外，类的成员函数也可以声明为内联函数。有以下两种方法可以声明类的成员函数为内联函数。

- 显式指定内联函数：在类的成员函数的定义前面加上 inline 关键字。
- 隐式指定内联函数：在类的成员函数的声明中，直接加上函数体的内容。

下面代码显示了指定类的成员函数为内联函数的方法。

```
class MyClass                                  //自定义 MyClass 类
{
public:
    int min()                                  //隐式内联
    {
        if( a < b )
            return a;
        else
            return b;
    }
    int max();                                 //取较大值函数
private:
    int a;                                     //整型变量值 a
    int b;                                     //整型变量值 b
```

```
};
inline int MyClass::max()                    //显式内联，取较大值函数
{
    if( a > b )
        return a;                            //如果 a 大于 b，则返回 a 值
    else
        return b;                            //否则返回 b 值
}
```

在上面的代码中，MyClass 类使用显式方式定义 max()函数为内联函数，使用隐式方式定义 min()函数为内联函数。

3.7.5　函数的重载

C++允许在同一作用域内指定相同函数名的多种函数形式，此技术称为函数重载。重载函数允许程序员根据参数类型和个数提供函数的不同语义。对程序员来说，函数重载就是定义相同函数名的多个函数原型，但是重载函数的返回值类型必须相同。

- 可以使用参数个数进行函数重载，即函数名称和返回值相同，但是参数的个数不同。
- 可以使用参数类型进行函数重载，即函数名称、返回值和参数个数都相同，但是参数的类型不同。
- 可以使用是否设置参数的默认值进行函数重载，即 3.7.2 小节中使用的 AddYear()函数的例子。
- 可以使用 const 和 volatile 关键字重载函数。

以下代码中定义的重载函数可以实现在多种数据类型中获取较大值的功能。

```
void max(int a, int b, int& c);
void max(double a, double b, double& c);
void max(char a, char b, char& c);
```

3.8　指针和引用

为了提高存储效率，C++中提供了指针和引用两种类型。这两种类型也是 C++数据类型中的重点和难点。在学习本节内容时，一定要透彻理解指针和引用的概念，在运用时要能够融会贯通、举一反三。

3.8.1　指针和指针变量

不管在 C 还是 C++中，指针类型都是一个难点和重点。C++中的指针指向内存中一个变量或对象的存储空间。因此，指针的定义分为两种情况，一种是指向给定类型的变量的指针，一种是指向类成员的指针。

指针变量指定变量指向的对象的类型。对象可以是全局的、本地的或者动态分配的。

指针类型指定了指针指向的对象类型，可以是基本类型、结构或联合体。指针变量也可以指向函数、数据和其他指针。使用指向给定类型的函数的指针，可以在程序运行时确定指定对象选择的函数。下面的代码是有关指针声明的几个例子。

```
char *szNameStr;                //指向 char 类型的指针
int *iCount;                    //指向 int 类型的指针
int const *x;                   //声明指针变量 x，指向一个常数值
int *const y = &a;              //声明常指针变量 y，指向一个 int 值
int *const volatile z = &b;     //声明固定值的常指针变量
```

在上面语句中，第一条语句声明了一个指向 char 类型的指针。第二条语句声明了一个指向 int 类型的指针。第三条语句声明了指针变量 x，可以修改其指向不同的 int 类型值，但是指向的 int 值是不能修改的。第四条语句声明的指针变量 y 是常指针，可以修改其指向的 int 类型的值，但是不能修改其指向其他的 int 值，即只能指向对象 a 的地址。第五条语句声明了指针变量 z，程序既不能修改其指向的值，也不能修改指针本身的取值。

3.8.2　&和*运算符

地址操作符符号为&，是一目运算符，用于获取操作数的地址。地址操作符的操作数可以是函数定义或是表示对象的变量，但不能是位字段，也不能是使用 register 声明的存储类关键字。地址操作符应用于基本类型变量、结构变量、类变量或共用体变量。代码如下：

```
void printAddressOf()           //地址操作符的示例
{
    int *pPtr;                  //定义整型指针变量 pPtr
    int nArray[5];              //定义整型数组
    pPtr = &nArray[2];          //赋值指针变量指向整型数组的第二个元素
    cout << pPtr << "\n";       //输出指针变量的值
    cout << &nArray << "\n";    //输出数组的地址值
}
```

上面的代码使用地址操作符获取 nArray 数组的第三个元素的地址，并将其存储在 pPtr 指针变量中，然后将其输出，并输出 nArray 数组的起始地址。运算结果为：

```
0x0012FEE0
0x0012FED8
```

从上面的运行结果可以看出，因为数组元素是整型，所以每个元素占用 4 个字节。数组起始地址为 0x0012FED8，也就是第一元素的存储地址，则第二个元素的存储地址为 0x0012FEDC，因此，第三个元素的存储地址为 0x0012FEE0，如图 3-11 所示。

注意：此处使用的地址值是根据内存情况由系统分配的，当每次测试时，结果值可能都不相同，这里仅是举例说明。

指针操作符符号为*，也称为间接访问操作符，是一目运算符，通过指针间接地获取操作数的取值。它的操作数必须是指针值，运算结果为操作数指向的地址中存放的值，代码如下：

```
void printIndirection()                    //间接操作符的示例
{
    int nTest;                             //定义整型变量 nTest
    int *pTest;                            //定义指针
    pTest = &nTest;                        //为指针变量分配值为 nTest 的地址
    *pTest = 15;                           //为 pTest 指向的存储区赋值
    cout << nTest << "\n";                 //输出 nTest 的值
}
```

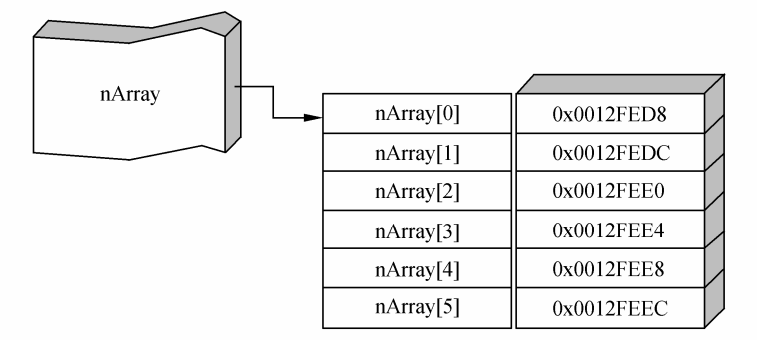

图 3-11　数组地址示意图

上面代码定义了两个变量，一个是整型类型的变量 nTest，一个是指向整型变量的指针类型 pTest。首先将整型变量的地址赋值给整型指针变量，此时，pTest 中存储的是 nTest 的地址。然后使用*间接操作符将 pTest 指向的地址中的值赋值为 15，最后输出 nTest 的值。运算结果为：

```
15
```

从上面的运行结果可以看出，*运算符可以看作获取操作数中存储的地址值中的实际值的运算符。虽然&和*符号的含义简单，但是只有深刻理解其本质，才能处理各种复杂的组合情况。

3.8.3　指针和数组

C++中允许指针指向数组。以下代码用来声明指向数组的指针。

```
int *student[10];              //声明指针数组，数组中的每个元素为指向整型的指针
int (*student )[10];           //声明一个指针，指向一个具有 10 个元素的数组
```

上面这两条声明的变量是不同的，第一条是声明了一个指针数组，它是一个具有 10 个元素的数组，其中每个元素都是一个指向 int 类型的指针。第二条声明了一个指针，它指向一个具有 10 个元素的数组。所以，要注意当指针遇到数组时的处理。

3.8.4　指针和结构体

C++中允许指针指向结构体，并且在结构体、共用体或枚举类型定义之前，可以使用标记声明指向结构体、共用体或枚举类型的指针。对于指针变量编译器不需要预先确定指

向的结构体或占用存储空间大小的共用体，因此使用指针定义结构体效率相对较高。代码如下：

```
struct collection *next, *previous;     //使用 collection 标记定义指针
struct collection                       //collection 结构的定义
{
    char *student;                      //存放学生名称的字符串
    int age;                            //存放学生年龄
    struct collection *next;            //指向下一个学生结构
}students;
```

上面代码声明了名为 next 和 previous 的两个指针变量，均指向结构体 collection。这条声明可以在 collection 结构定义之前声明，当定义结构后，则在声明中结构也就可见了。变量 students 是 collection 结构类型的变量。collection 结构具有 3 个成员，第一个数据成员是指向 char 类型的 student 指针，用于存储学生代码；第二个数据成员是指向 int 类型的 age 指针，用于存储学生年龄；第三个数据成员是指向另外一个 collection 结构的指针，用于链接下一条学生记录。

3.8.5　函数的指针传递

在函数中可以像使用其他类型一样，将指针作为函数参数传递。示例代码如下：

```
struct student                          //学生结构
{
    char name[50];                      //存放学生名称的字符串
    int age;                            //存放学生年龄
};
void printStudent(student* stu)         //输出学生信息
{
    cout << "姓名=" << stu->name << ";年龄=" << stu->age <<".";
                                        //输出学生姓名和年龄
}
void TestPassPoint()                    //测试指针传递
{
    student* stu = new student();       //创建指向 student 结构的变量
    sprintf(stu->name, "%s", "张三");   //为 student 结构的 name 分量赋值为张三
    stu->age = 10;                      //为 student 结构的 age 分量赋值为 10
    printStudent(stu);                  //将 stu 指针传入 printStudent 参数，
                                        //输出学生信息
}
```

上面代码定义了 student 结构体和打印学生信息的 printStudent()函数。在 TestPassPoint()测试函数中，定义了 student 类型的变量并赋值，将其传入 printStudent()函数中，从而输出定义的学生信息。

3.8.6　引用及函数的引用传递

引用类型（&）是一个 16 位或 32 位的存放对象指针数，但是它的行为更像对象。引用声明的格式如下：

```
声明标识符 & [限定符列表] 引用名称 ;
```

其中，声明标识符列出了要定义的数据类型信息，限定符列表是一个可选的包含引用定义的限定符选项，引用名称是定义的引用的名称。在 C++中，任何地址都可以转换为指针类型的对象，也可以转换成对应的引用类型。如任何地址都可以转换为 char*的对象，也可以转换成 char&类型。以下代码显示了引用类型的使用。

```
void AddOne(int& a)                    //引用类型示例
{
    a++;                               //将传入的变量的值自增 1
}
void printRefrence()                   //打印通过引用传递的值
{
    int x= 99;                         //定义整型变量 x，赋值为 99
    cout << "x=" << x << "\n";         //输出 x 的值
    AddOne(x);                         //通过调用带有引用参数的函数改变变量的值
    cout << "AddOne(x) x=" << x << "\n";   //输出改变值后的 x 的值
}
```

在 AddOne()函数的定义中，int 是声明标识符，&是引用操作符，x 是定义的引用名称。在 printRefrence()函数中，定义了 int 类型的 x，并将其传入 AddOne()函数中，在 AddOne()函数中，将传入的变量 a 的值自增 1。在 AddOne()函数中修改的 a 的值，就是修改了 printRefrence()函数的 x 值。因此，&操作符与地址操作符的功能是相同的，它传入的是变量的引用，而不是一般函数的形参。函数的运行结果如下：

```
x=99
AddOne(x) x=100
```

3.9 预　处　理

C++程序进行编译前，会首先检索程序代码，将其中的预处理指令进行转换，然后将转换后的代码提交给 C++编译器，编译器再进行程序编译，这个预处理指令转换的过程称为预处理。在 C++中预处理主要包括 3 种，分别是宏定义、文件包含和条件编译。本节将分别介绍这 3 种预处理方式。

3.9.1 宏定义

C++中，预处理通过预处理指令完成，所有的预处理指令都是以#开头。宏定义的作用是为常用的对象分配一个有意义的名称，指令是#define，语法格式如下：

```
#define 宏名称 宏内容
```

宏定义指令定义的内容会在预处理时，将所有使用宏名称的地方都使用宏内容代替。但是当宏名称出现在注释中或包含在其他标识符中时，预处理器不会替换宏内容。在宏内容中可以包括一系列标记，如关键字、常数、完整语句和空格等。宏名称中可以使用参数，但是在使用宏时，传入的参数必须与宏定义中指定的参数个数一致。在定义具有参数的宏

时，最好使用小括号分隔宏名称和宏参数，使得代码清晰易懂。以下代码为宏定义的示例。

```
#define LENGTH  100          //定义名称为 LENGTH 的宏，值为 100
#define ADD(a,b) (a + b)     //定义名称为 ADD 的宏，作用是将 a 和 b 相加
#define min(a,b) \
   (if (a>b) ? b : a )       //定义名称为 min 的宏，作用是获取两个值中较小的一个
```

上面的代码定义了 3 个宏，分别是 LENGTH、ADD 和 min。其中，LENGTH 宏代表一个常量，ADD 宏代表一个加法表达式，min 宏代表一个关系运算表达式。在定义 min 宏时，使用反斜线作为续行符。

在 C++中，可以使用#if defined 和#ifdef 指令判断是否定义过指定的宏。使用#undef 指令可以关闭已经使用#define 指令定义的宏。其语法格式为：

```
#undef 宏名称
```

其中，宏名称表示要取消的宏的名称。以下代码会取消对 LENGTH 宏的定义，在后面代码中如果使用 LENGTH 宏将会产生错误。

```
#undef LENGTH
```

3.9.2　文件包含

在 C++程序中，一个系统可以由多个程序模块组成，每个程序模块由多个文件组成，这就会出现在一个文件中引用其他文件中的内容的情况，此时需要使用#include 文件包含预处理。#include 指令告诉预处理器在使用此指令的文件中包含指定文件中的代码。

文件包含支持嵌套，即被包含的文件中也可以使用#include 指令包含其他文件。可以将用到的常数和宏定义放置到文件中，然后使用#include 指令将这些定义添加到任何需要使用这些常数和宏的源文件中。#include 指令对于组织外部变量和复杂的数据类型来说非常有用。使用#include 指令使得在多处使用的定义可以只在一处定义。#include 指令的语法格式如下：

```
#include "文件名"
#include <文件名>
```

其中，文件名表示要加入内容所在的文件名，可以指定文件所在路径。文件名的语法格式根据操作系统的不同会有差别。#include 的两种语法格式的功能是相同的，区别仅在于当没有指定完整的文件路径时，预处理器查找头文件的路径有所不同。

- 当使用第一种语法指定文件包含时，会首先从包括#include 语句的文件所在的路径下查找，然后从其他包含此文件的文件所在的路径查找，再从/I 编译选项指定的路径中查找，最后会在 INCLUDE 环境变量中指定的路径中查找。
- 当使用第二种语法指定文件包含时，会先从/I 编译选项指定的路径中查找，然后在 INCLUDE 环境变量中指定的路径中查找。

以下代码是#include 指令的使用代码。

```
#include <stdio.h>
#include "Sample.h"
```

上面的代码包含 stdio.h 文件和 Sample.h 文件。其中，stdio.h 文件会在/I 编译选项指定的路径和 INCLUDE 环境变量中指定的路径中查找，而 Sample.h 会首先从文件本地路径中查找。

3.9.3　条件编译

C++支持条件预编译指令控制源文件编译的部分。条件编译指令有#if、#elif、#else 和#endif 指令。如果编译指令#if 后的表达式为非 0 值，则其后的代码会进行编译，直到#endif 指令结束处。在源文件中，每个#if 指令必须与#endif 指令配对。在#if 和#endif 之间可以使用多个#elif 指令判断多种条件，但是其中最多只能有一条#else 指令。另外，如果使用#else 指令，则必须是#endif 指令前的最后一条指令。条件编译指令是支持嵌套的，每个嵌套的#else、#elif 和#endif 指令与距离其最近的#if 指令配对。

通常情况下，条件编译会与#defined 配对使用，判断指定的宏名称是否已经定义，确定是否编译指定代码。代码如下：

```
#if defined(DB_SQLSERVER)        //如果定义了 DB_SQLSERVER
   connectSQLServer();           //连接 SQLServer 数据库
#elif defined(DB_ACCESS)         //如果定义了 DB_ACCESS
   connectACCESS();              //连接 Access 数据库
#else                            //如果没有定义数据库类型
   printerror();                 //输出错误信息
#endif
```

在上面代码中，首先使用#if defined(DB_SQLSERVER)语句判断是否定义了 DB_SQLSERVER 宏。如果定义了该宏，会使用 connectSQLServer()函数连接 SQLServer 数据库；如果没有定义 DB_SQLSERVER 宏，则继续判断是否定义了 DB_ACCESS 宏。如果定义了 DB_ACCESS 宏，则会使用 connectACCESS()函数连接 Access 数据库，否则，会报告错误。

3.10　文 件 操 作

文件是操作系统组织数据和操作的基本元素。计算机从最初的科学计算向前迈进的第一步就是文件系统。文件将数据和操作分单元存储，可以写入数据、读取数据等。因此，对文件的操作也主要分为打开文件、读文件、写文件和关闭文件。本节将介绍有关文件的操作。

3.10.1　打开文件

要对文件进行读写，首先需要打开文件，使用 fopen()函数和_wfopen()函数可以打开文件。其原型为：

```
FILE *fopen( const char *filename,      //表示要打开的文件名称
             const char *mode );        //打开文件时的访问权限类型
```

```
FILE *_wfopen( const wchar_t *filename, const wchar_t *mode );
```

其中，mode 参数的有效取值如下。

- r：读取文件打开。如果文件不存在或查找不到，则 fopen()函数调用失败。
- w：打开空文件写入。如果指定的文件存在，则内容会被删除。
- a：打开文件，从文件尾开始添加内容，在向文件中写入新数据前不删除 EOF 标记。如果文件不存在，则创建文件。
- r+：打开文件，既可以从文件中读取，也可以向文件中写入，但是文件必须存在。
- w+：打开空文件，既可以从文件中读取，也可以向文件中写入，如果文件存在，则会删除其中的数据。
- a+：打开文件，既可以从文件中读取，也可以向文件中写入。在新数据被写入文件之前会移除 EOF 标记。当写入操作完成时，会恢复 EOF 标记。如果文件不存在，则会创建新文件。

FILE 表示返回的打开的文件指针。如果返回值是 NULL 指针，则表示在打开文件时发生错误了。fopen()函数和_wfopen()函数的区别在于_wfopen()函数是 fopen()函数的多字符集版本。

3.10.2　从文件读取数据

从文件或数据流中读取数据使用 fread()函数，其原型为：

```
size_t fread(
    void *buffer,                //指定存储数据的本地存储空间的指针
    size_t size,                 //指定存储区的大小
    size_t count,                //指定要读取的数据项的最大个数
    FILE *stream                 //指向 FILE 结构的指针，是使用 fopen()函数
                                 //打开的文件句柄
 );
```

fread()函数会从文件流 stream 中读取 size 参数和 count 参数之间较小数目的数据，存储到 buffer 参数指定的缓冲区中。函数的返回值是实际读取的数据长度，当读取的过程发生错误或已经读到文件结尾处时，返回值可能小于参数 count 指定的值。使用 feof()函数和 ferror()函数可以区分这两种情况。如果参数 size 或参数 count 为 0，则函数返回 0，并且不会读取数据。

3.10.3　向文件写入数据

向文件中写数据使用 fwrite()函数，其原型为：

```
size_t fwrite(
    const void *buffer,      //指定存放要写入的数据的存储空间的指针
    size_t size,             //指定存储区的大小
    size_t count,            //指定要写入的数据项的最大个数
    FILE *stream             //指向 FILE 结构的指针，即使用 fopen()函数
                             //打开的文件句柄
 );
```

fwrite()函数会将 buffer 参数指定缓冲区中的数据写入文件流 stream 中，写入的数据个数是 size 参数和 count 参数之间较小的数目。函数返回实际写入的数据长度，当写数据发生错误时，则返回值可能小于参数 count 指定的值。如果参数 size 或参数 count 为 0，则函数返回 0，并且不会向文件流中写任何数据。

3.10.4　关闭文件

C++中使用 fclose()函数或_fcloseall()函数关闭文件。其中 fclose()函数关闭文件流，_fcloseall()函数关闭所有打开的文件流。_fcloseall()函数还可以关闭和删除 tmpfile()函数创建的临时文件。当文件流关闭时，会释放系统分配的缓冲区。函数原型为：

```
int fclose( FILE *stream );          //指向 FILE 结构的指针
int _fcloseall( void );
```

如果成功关闭文件流，则 fclose()函数返回 0。_fcloseall()函数用于返回关闭的文件流的个数。当关闭文件流发生错误时，函数会返回 EOF。

3.10.5　文件操作示例

从上面讲解的内容可以看出文件操作的过程是这样的，先根据需求使用合适的权限打开文件，判断是否成功打开文件，并对文件进行读写，文件使用完后，需要关闭文件句柄。以下代码是文件操作的示例。

```
01  #include <stdio.h>
02  #include <process.h>
03
04  FILE *stream, *stream1, *stream2;
05
06  void main()
07  {
08      int numclosed;
09      char list[30];               //存放从文件中读取的数据
10      int  i, numread, numwritten;    //读取的数目，写入的数目
11
12      //打开文件 data 进行读，如果文件不存在，则失败
13      if( (stream1  = fopen( "data", "r" )) == NULL )
14          printf( "打开文件'data'进行读失败\n" );
15      else
16          printf( "打开文件'data'进行读\n" );
17
18      //打开文件 data2 进行写操作
19      if( (stream2 = fopen( "data2", "w+" )) == NULL )
20          printf( "打开文件'data2'进行写失败\n" );
21      else
22          printf( "打开文件'data2'进行写\n" );
23
24      //使用文本模式打开文件，对文件进行写操作
25      if( (stream = fopen( "fread.out", "w+t" )) != NULL )
26      {
27          //向文件流中写入 25 个字符
28          for ( i = 0; i < 25; i++ )
```

```
29              list[i] = (char)('z' - i);
30
31          numwritten = fwrite( list, sizeof( char ), 25, stream );
32          printf( "写入 %d 个字符\n", numwritten );
33          fclose( stream );
34      }
35      else
36          printf( "打开文件 fread.out 时，发生错误，无法写数据到文件中\n" );
37
38      if( (stream = fopen( "fread.out", "r+t" )) != NULL )
39      {
40          //从文件中读取 25 个字符
41          numread = fread( list, sizeof( char ), 25, stream );
42          printf( "读取的数据个数 = %d\n", numread );
43          printf( "读取的内容为 = %.25s\n", list );
44          fclose( stream );
45      }
46      else
47          printf( "打开文件 fread.out 时，发生错误，无法从文件中读取数据\n" );
48
49      //关闭文件
50      if( fclose( stream2 ) )
51          printf( "关闭文件'data2'失败\n" );
52
53      //关闭其他打开的文件
54      numclosed = _fcloseall( );
55      printf( "使用函数_fcloseall 关闭的文件数目为 : %u\n", numclosed );
56      system("pause");
57  }
```

上面代码演示了操作文件的方法，包括打开文件、读文件、写文件和关闭文件。其中，对文件句柄 stream1 和 stream2 没有进行读写操作，对文件句柄 stream 进行了读写操作。首先向文件中写入 25 个字母，然后再从文件中读取这 25 个字母，最后关闭文件。代码运行结果如图 3-12 所示。

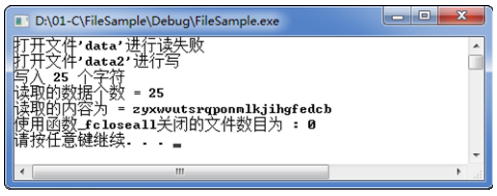

图 3-12　代码运行结果

3.11　本 章 小 结

本章主要讲解了 C++的基本语法，在与 C 语言语法对比的基础上，讲述了 C++语言特有的特性。从基本的变量和常量，到控制语句的语法以及函数的概念，C++在继承 C 的基础上做了适当调整。C++中的指针和引用是 C++语法中的一个难点。预处理的理解也是熟悉 C++语言必不可少的。最后以文件操作为例，讲述了在 C++中操作文件的方法，并结合实例进行说明。第 4 章将讲解 C++语言中的面向对象程序设计的知识。

3.12　习　　题

1．定义一个结构体 Person，用来记录人的姓名、性别、年龄、身份证号码和年收入。

【思路】依据结构体各成员存放的数据的情况，来决定所使用的数据类型。

【示例代码】

```
struct Person
{
    char name[9];
    char sex[3];
    short age;
    char identityCard[19];
    float income;
};
```

2．有一个式子：(x + 303 * y – 64)/ z 。x、y、z 分别为 int 型的变量，编程实现：从用户处获取各个变量的值，默认赋值 1，代入表达式中计算并输出计算结果。

【思路】通过 cin 获取用户的输入，代入表达式中求值，用 cout 输出。如图 3-13 所示，为编程实现效果的一种方式。

3．声明一个 short 类型的数组 Num，包含 10 个数据成员，成员依条件赋值，条件是：3*n+1（n 是大于 0 小于 11 的整数）。然后通过指针 pNum 来修改数组中下标为偶数的成员，统一设置为 0，最后输出所有的数组成员。

【思路】按规则为数组赋值，然后用指针指向数组并按照规则修改数组成员，最后循环输出所有的数组成员。如图 3-14 所示为演示的一种输出效果。

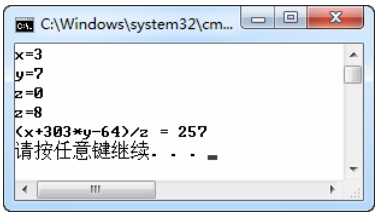

图 3-13　程序运行效果

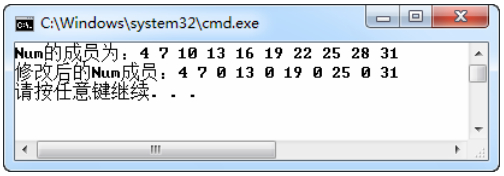

图 3-14　数组输出效果

4．依据用户输入的完整文件名（包含路径），打开并读取文本文件内容，然后显示出来。

【思路】使用 fopen()打开文件、fread()读取文件、feof()判断是否到达文件末尾。如图 3-15 所示为读取文件内容的效果。

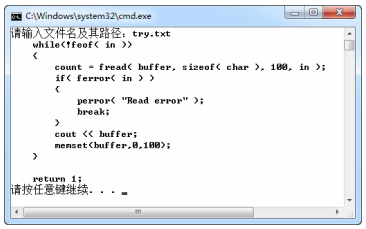

图 3-15　读取文件内容效果