◇ 第一章 引言
- 课前索引
- 第一节 计算机语言及其发展
- 第二节 程序设计方法
- 第三节 C++语言的特点
- 第四节 Visual C++6 开发环境简介
- 本章小结
- 课后习题
◇ 第二章 C++语言基础
- 课前索引
- 第一节 简单的 C++程序
- 第二节 标识符和关键字
- 第三节 基本数据类型
- 第四节 变量
- 第五节 常量
- 第六节 枚举类型
- 第七节 输入和输出
- 本章小结
- 课后习题
◇ 第三章 运算符和表达式
- 课前索引
- 第一节 C++运算符
- 第二节 优先级和结合性
- 第三节 类型转换
- 第四节 表达式
- 本章小结
- 课后习题
◇ 第四章 流程控制语句
- 课前索引
- 第一节 if 语句
- 第二节 switch 语句
- 第三节 循环语句
- 第四节 break 和 continue 语句
- 第五节 程序举例
- 本章小结
- 课后习题
◇ 第五章 函数
- 课前索引
- 第一节 函数的定义
- 第二节 函数的说明
- 第三节 函数的调用
- 第四节 inline 函数

- 第五节　函数的递归调用
- 第六节　变量的作用域与存储期
- 第七节　函数的重载
- 第八节　程序举例
- 本章小结
- 课后习题

◇ 第六章 数组与字符串

- 课前索引
- 第一节　一维数组
- 第二节　二维数组
- 第三节　字符数组
- 第四节　数组作函数参数
- 第五节　程序举例
- 本章小结
- 课后习题

◇ 第七章 指针与引用

- 课前索引
- 第一节　指针变量的定义与使用
- 第二节　动态内存分配
- 第三节　指针运算
- 第四节　指针与常量
- 第五节　指针作函数参数
- 第六节　指针与数组
- 第七节　函数指针
- 第八节　引用
- 第九节　Typedef
- 第十节　程序举例
- 本章小结
- 课后习题

◇ 第八章 结构与链表

- 课前索引
- 第一节　结构及使用
- 第二节　结构变量作为函数参数
- 第三节　结构数组
- 第四节　指向结构的指针
- 第五节　链表
- 第六节　程序举例
- 本章小结
- 课后习题

◇ 第九章 编译预处理

- 课前索引
- 第一节　宏定义

- 第二节　文件包含
- 第三节　条件编译
- 本章小结
- 课后习题

◇ 第十章 类与对象

- 课前索引
- 第一节　类与对象概述
- 第二节　构造函数与析构函数
- 第三节　复制构造函数
- 第四节　类作用域
- 第五节　const 成员函数
- 第六节　静态成员
- 第七节　友元
- 本章小结
- 课后习题

◇ 第十一章 继承与多态

- 课前索引
- 第一节　基类和派生类
- 第二节　虚函数与动态联编
- 第三节　抽象类
- 第四节　虚析构函数
- 第五节　设计继承
- 第六节　程序举例
- 本章小结
- 课后习题

◇ 第十二章 输入输出流

- 课前索引
  - 第一节　输入输出流类
  - 第二节　文件流
  - 第三节　字节流类
  - 第四节　流错误处理
  - 本章小结
  - 课后习题
- 【学习目标】

  本章介绍 C++语言的一些基本知识，包括程序的基本结构、标识符、数据类型、变量、常量、基本的输入和输出方法等。通过本章的学习，应该能够编写简单的 C++程序。

  【重点和难点】

  本章内容比较简单，没有难于理解的概念。重点内容包括：

  ◇ C++语言中的变量和常量的概念和使用；

  ◇ C++语言中输入输出库的使用。

  【学习方法指导】

  本章内容简单，但确是重要的基础知识，对基本概念应加强记忆与理解，为以后的学习打下坚实的基础。

【知识点】

标识符；关键字；变量；整型变量；浮点型变量

字符型变量；布尔型变量；字符串；常量；枚举变量

标准输出；标准输入；printf 输出；scanf 输入

首先看下面这个简单的 C++程序（为了方便起见，程序的每一行都加上了行号）。

程序 2-1：

```
1．//simple.cpp 是一个简单的 C++程序
2．#include <iostream.h>
3．void main(void)
4．{
5. int i;
6．  cout<<"Hello C++!";
7．  cout<<"请输入一个整数：\n";
8．  cin>>i;
9．  cout<<"您输入的整数是：";
10．  cout <<i;
11．  cout <<'\n';
12．}
```

请学习者自己键入这段程序，并自己编译、连接、运行该程序，输入不同的整数，看能否得到预期的结果。注意，程序中左边的数字是为了讲解的方便添加的，将该程序键入到开发环境的编辑窗口时，一定要略去这些数字。

我们对这个程序逐行加以解释：

（1）第一行是 C++语言的注释。其中，"//"是 C++语言的一种注释符号，自"//"开始，一直到本行结束，所有内容都会被当作注释对待。C++注释也可以写成下面的形式：

/*注释内容*/

即夹在"/*"与"*/"号间的部分是要注释的内容，例如，本句可以改为：

/*simple.cpp 是一个简单的 C++程序*/

我们进行程序设计时，这两种注释形式都会经常用到。它们的区别在于：前者只能注释一行内容，而后者可以注释多行内容。它可以用在程序的任何地方，编译器在编译时将这些信息忽略，注释就是用来帮助阅读和理解程序的信息，也可使用注释来帮助用户调试程序。

（2）第 2 行使用预处理指令#include 将头文件 iostream.h 包含到程序中来， iostream.h 是标准的 C++头文件，它包含了输入和输出的定义。

（3）第 3 行定义了一个称为 main 的函数。

◇ 一个函数有 0 个或多个参数，它们出现在函数名后的一对圆括号中。

◇ 括号中的 void 表示 main 没有参数。

◇ 一个函数可以有一个返回类型，它在函数名的左边。

◇ main 函数左边的返回类型为 void，表示没有返回值。

◇ C++程序的执行总是从 main 函数开始的。

（4）第 4 行是一个花括号，是 main 函数体开始的标记。

（5）第 5 行是一个语句。

◇ 一个语句可能是定义或声明一个变量，也可能是得到一个数值的计算步骤。

◇ 一个语句用分号(;)结尾，C/C++用分号来分隔语句。

◇ 这个语句定义了一个整型变量 i。

◇ 一个变量能够保存一种特定类型的数据，整型变量能够保存一个整型数据。

（6）第 6 行也是一个语句。

◇ 这个语句将字符串"Hello C++!"发送到 cout 输出流。

◇ 一个字符串是一个用双引号包围的字符系列。

（7）第 7 行也是一个语句。

◇ 这个语句将字符串"请输入一个整数："发送到 cout 输出流。

◇ 字符串的最后一个字符（\n）是一个换行符。

◇ 流是执行输入和输出的对象。

◇ cout 是 C++标准的输出流，标准输出通常是指计算机屏幕。

◇ 符号<<是一个输出运算符，带一个输出流作为它的左操作数，一个表达式作为它的右操作数。后者被发送到前者，字符串"请输入一个整数：\n"发送到 cout 的效果是把字符串打印到计算机屏幕上。

（8）第 8 行也是一个语句。

◇ 这个语句将 cin 输入流抽取到变量 i。

◇ cin 是 C++标准的输入流，标准输入通常是指计算机键盘。

◇ 符号>>是一个输入运算符，带一个输入流作为它的左操作数，一个变量作为它的右操作数。前者被抽取到后者，cin 输入流抽取到变量 i 的效果是将键盘的输入值复制到变量 i 中。

（9）第 9、10、11 行分别是在屏幕上打印"您输入的整数是："、变量 i 和和换行符。这三行实际上可以综合为一个语句：

```
cout<<"您输入的整数是：" <<i <<'\n';
```

它的效果与使用三个语句是一样的。

（10）第 12 行的花括号是 main 函数体结束的标记。

运用第一章介绍的知识，我们在 Visual C++6 的开发环境里，编译、连接、运行该程序，可以得到下面的提示：

```
Hello C++!请输入一个整数：
```

假定我们输入整数 5，即：

```
5↙
```

↙表示键入了"ENTER"键（或称为回车键），则得到运行结果：

```
您输入的整数是：5
```

通过这个实例，我们对 C++程序的结构、语句、变量、函数、预处理指令、输入和输出等，已有了一个初步的印象，在后面的章节中，我们还将作更详细的介绍。

由于我们定义 main( )函数的返回类型为 void，所以最后就不用返回值了。如果我们定义 main 的返回类型的 int，则要返回一个整型值：

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

要注意的是 C/C++是区分大小写的，不能随便更改，cout 是 C++标准的输出流，而 Cout 不是，如果没有预先定义，编译器就不认识这个 Cout。大多数 C＋＋命令使用小写字母，大多数常量使用大写字母，而大多数变量是大小写混合使用。

标识符是一个字符序列，用来标识变量、函数、数据类型等。任何程序都离不开标识符，也就是说，不可能有没有标识符的 C++程序。在程序 2-1 中，include、void、main、int、i、cin、cout 等都是标识符。

标识符可以由大写字母、小写字母、下划线（_）和数字 0~9 组成，但必须是以大写字母、小写字母或下划线（_）开头。在 C++语言程序中，大写字母和小写字母不能混用，比如 Name 和 name 就代表两个不同的标识符。在表 2-1 中，我们给出了几个正确和不正确的标识符实例。

表 2-1：正确和不正确的标识符实例

正确	不正确
smart	5smart（不能数字开头）
_decision	bomb?（有非法字符？）
key_board	key.board（有非法字符.）

标识符的命名规则：

1. 所有标识符必须由一个字母(a~z 或 A~Z)或下划线(_)开头；
2. 标识符的其它部分可以用字母、下划线或数字(0~9)组成；
3. 大小写字母表示不同意义, 即代表不同的标识符，如前面的 cout 和 Cout；

在定义标识符时，虽然语法上允许用下划线开头，但是，我们最好避免定义用下划线开头的标识符，因为编译器常常定义一些下划线开头的标识符。

C++没有限制一个标识符中字符的个数，但是，大多数的编译器都会有限制。不过，我们在定义标识符时，通常并不用担心标识符中字符数会不会超过编译器的限制，因为编译器限制的数字很大（例如 255）。

一个写得好的程序，标识符应该尽量有意义。 比如表示年可以用 year，表示长度可以用 length，表示累加和可以用 sum 等，这样的标识符本身就增加了程序的可读性，使程序更加清晰易懂。

C++语言的标识符经常用在以下情况中：

1. 标识对象或变量的名字
2. 类、结构和联合的成员
3. 函数或类的成员函数
4. 自定义类型名
5. 标识宏的名字
6. 宏的参数

在 C++中，有一些预定义的标识符，称之为关键字，也称之为保留字。例如，程序 2-1 中的 int、void 都是关键字。可见，关键字是一种特殊的标识符。关键字具有特定的含义，不能对它们再定义。例如，int、void 在 C++中被预定义为特定的数据类型，我们不能把它们再定义为变量的标识符。C++的关键字很多，不仅仅程序 2-1 中见到的 int 和 void 两个。标准 C++中预定义了 63 个关键字，参见表 2-2。另外，还定义了 11 个运算符关键字，它们是：and、and_eq、bitand、bitor、compl、not、not_eq、or、or_eq、xor ?、xor_eq。在我们后面要学习的内容中，并没有涉及到 C++的所有关键字，但会逐步介绍最重要和最常用的一些关键字。

另外，有些标识符虽然不是关键字，但 C++语言总是以固定的形式用于专门的地方，也不能把它们当作一般标识符使用，以免造成混乱。这样的标识符有 include、define 等，我们在后面的学习中会逐渐遇到。

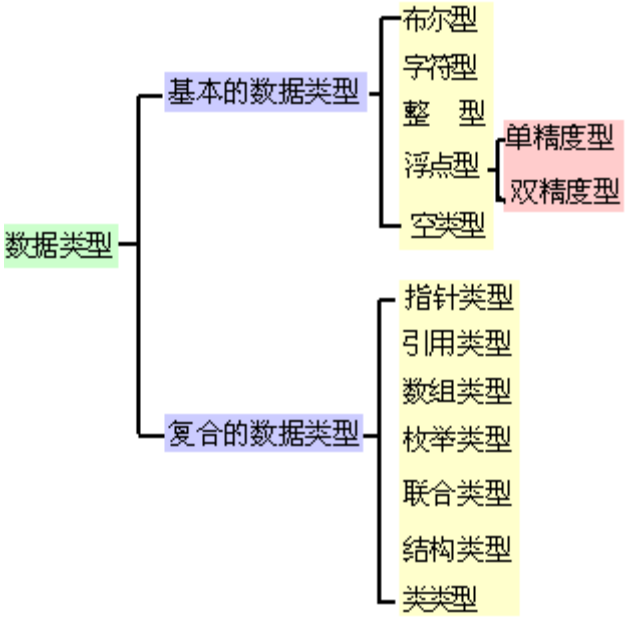
2-2 C++关键字

asm	default	float	operator	static_cast	union
auto	delete	for	private	struct	unsigned
bool	do	friend	protected	switch	using
break	double	goto	public	template	virtual
case	dynamic_cast	if	register	this	void

catch	else	inline	reinterpret_cast	throw	volatile
char	enum	int	return	true	wchar_t
class	explicit	long	short	try	while
const	export	mutable	signed	typedef	
const_cast	extern	namespace	sizeof	typeid	
Continue	false	new	static	typename	

在 C++中，数据具有不同的类型，类型定义了变量可存储的数值范围以及可进行的操作变量是用于内存中保存数据的，每个变量都必须有确定的数据类型，C++语言的数据类型如图 2-1 所示。

在图 2-1 中，把数据类型划分为基本的数据类型和复合的数据类型，我们也可把数据类型分为内置的类型和用户定义的类型两大类，用户定义的类型在使用以前，必须先定义，包括：结构、类、枚举和联合类型；内置的类型是指直接被 C++提供的类型，也就是说，是除用户定义的类型以外的其它类型。



从语法上来说，void 类型也是基本的类型，但是，它不是一个完整的类型，只能作为更复杂类型的一部分。没有 void 类型的变量，它或者用于指定一个函数（在第五章介绍）没有返回值，或者作为指针（在第七章介绍）类型，表示该指针指向未知类型的变量，例如：

```
void x; //错误，没有 void 变量
void f( ); //函数 f 不返回值
void *pv; //指针 pv 指向未知类型的变量
```

布尔型、字符型、整型和浮点型也称为算术类型；枚举、指针、数组、引用、结构、联合和类称之为复合数据类型，它们能通过其它数据类型进行构造。

在基本的数据类型 char、 int 前可以添加修饰符，以改变基本类型的意义，可用的修饰符有 long、short、signed 和 unsigned 四种，另外，双精度型前可以加 long 修饰符。基本的数据类型及其表示范围，可参见表 2-3。

表 2-3 基本的数据类型及其表示范围

类型名	类型	字节	表示范围
char	字符型	1	-128 ~127

unsigned char	无符号字符型	1	0 ~255
signed char	有符号字符型（与字符型相同）	1	-128 ~127
int	整型	*	与机器有关
unsigned int	无符号整型	*	与机器有关
signed int	有符号整型（与整型相同）	*	与机器有关
short int	短整型	2	-32,768~ 32,767
unsigned short int	无符号短整型	2	0~65,535
signed short int	有符号短整型（与短整型相同）	2	-32,768~ 32,767
long int	长整型	4	-2,147,483,648 ~2,147,483,647
signed long int	有符号长整型（与长整型相同）	4	-2,147,483,648 ~ 2,147,483,647
unsigned long int	无符号长整型	4	0~4,294,967,295
float	浮点型	4	3.4E +/- 38 (7 位有效数字)
double	双精度型	8	1.7E +/- 308 (15 位有效数字)
long double	长双精度型	10	1.2E +/- 4932 (19 位有效数字)

unsigned 和 signed 只用于修饰 char 和 int，且 signed 修饰词可以省略。当用 unsigned 修饰词时，后面的类型说明符可以省略。例如：

signed int n; //与"int n;"等价

signed char ch; //与"char ch;"等价

unsigned int n; //与"unsigned n;"等价

unsigned char ch; //与"unsigned ch;"等价

short 只用于修饰 int，且用 short 修饰时，int 可以省略，即：

short int n; //与"short n;"等价

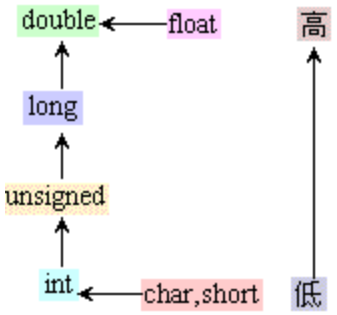
long 只能修饰 int 和 double。当用 long 修饰 int 时，int 可以省略，即：

long int n; //与"long n;"等价

int 和 unsigned int 类型占用一个机器一个字（word）的字节。在 16 位操作系统上，它们占用 2 个字节；在 32 位操作系统上，它们占用 4 个字节。

各种数据类型精度由低到高排列如图 2-2：

图 2-2



如果一个双目运算符两边的操作数类型不同，先要将它们转换为相同的类型，即较低类型转换为较高类型，然后再参加运算。所谓类型的高低，跟所占的存储空间大小有直接关系，所占存储空间越大的类型，级别越高。

图中横向的箭头表示必须的转换，如两个 float 型数参加运算，虽然它们类型相同，但仍要先转换成 double 型再进行运算，结果亦为 double 型。纵向箭头表示当运算符两边的操作数为不同类型时的转换，如一个 long 型数据与一个 int 型数据一起运算，需要先将 int 型数据转换为 long 型，然后两者再进行运算，结果为 long 型。所有这些转换都是由系统自动进行的，使用时只需了解结果的类型即可。

程序 2-2

```
#include <iostream.h>
void main(void)
{
  char a = 'x';
  int b = 3,f = 2;
  float c = 2.5678;
  double d = 5.2345
  long e = 32L;
  cout<<a - b + d / c - e * f<<endl;
}
```

下面我们来分析一下这段程序：

（1） 进行 d / c 运算时，要将 c 转换成 double 型，运算的中间结果为 double 型；

（2） 进行 e * f 运算时，将 f 转换为 long 型，运算的中间结果为 long 型；

（3） 进行 a - b 运算时，将 a 转换为 int 型（数值为 ASC II 码值 120），运算的中间结果为 int 型；

（4） 当（3）的中间结果与（1）的中间结果运算时，将（3）的中间结果转换为 double 型，运算的中间结果为 double 型；

（5） 当（4）的中间结果与（2）的中间结果运算时，将（2）的中间结果转换为 double 型，得出最后结果。

于是，程序最后的运行结果为 55.038515。

如果数据是从较高类型转换成较低类型，这意味着类型的精度或表示范围降低，可能造成数据丢失。下面两个表分别列出了有符号数之间和无符号数之间的转换：

1． 有符号数的转换

有符号数的转换中，如果从较低类型转换到较高类型，将进行符号扩展，例如一个值从 short int（16

位）转换到 long 类型，如果这个数是正数，则最高位为 0，从 16 位扩展到 32 位时，扩展的高 16 位用 0 填充，即将符号位 0 进行扩展，这样扩展后的 32 位整数和原来的整数值是一样的。如果该数为负数，则最高位为 1，从 16 位扩展到 32 位时，扩展的高 16 位用 1 填充，即将符号位 1 进行扩展，这样扩展后的 32 位整数和原来的整数值是也是一样的。

如果从较高类型转换到较低类型，将抛弃高位，直接将低位复制过来，例如一个值从 int（假定为 32 位）转换到 short int 型（16 位），系统将抛弃高 16 位，取低 16 位的值作为转换后的值。

浮点数类型和整数类型转换比较复杂，因为它们的内部表示方式不同，转换时它们不是简单的符号位扩展或者高位截断，它们首先需要进行内部表示方式的转换。左表是有符号数类型转换的所有情况。

表 2-4

从	到	方法
char	short	符号位扩展
char	long	符号位扩展
char	unsigned char	最高位失去符号位意义，变为数据位
char	unsigned short	符号位扩展到 short;然后从 short 转到 unsigned short
char	unsigned long	符号位扩展到 long; 然后从 long 转到 unsigned long
char	float	符号位扩展到 long; 然后从 long 转到 float
char	double	符号位扩展到 long; 然后从 long 转到 double
char	long double	符号位扩展到 long; 然后从 long 转到 long double
short	char	保留低位字节
short	long	符号位扩展
short	unsigned char	保留低位字节
short	unsigned short	最高位失去符号位意义，变为数据位
short	unsigned long	符号位扩展到 long; 然后从 long 转到 unsigned double
short	float	符号位扩展到 long; 然后从 long 转到 float
short	double	符号位扩展到 long; 然后从 long 转到 double
short	long double	符号位扩展到 long; 然后从 long 转到 double
long	char	保留低位字节
long	short	保留低位字节
long	unsigned char	保留低位字节
long	unsigned short	保留低位字节
long	unsigned long	最高位失去符号位意义，变为数据位
long	Float	使用单精度浮点数表示。可能丢失精度。
long	double	使用双精度浮点数表示。可能丢失精度。
long	long double	使用双精度浮点数表示。可能丢失精度。

2. 无符号数的转换

无符号数转换相对简单一些，它没有符号位，当低级类型向高级类型转换时，只需要将高位补 0，高级类型向低级类型转换同有符号数。左表是无符号数类型转换的所有情况

从	到	方法

unsigned char	char	最高位作为符号位
unsigned char	short	0 扩展
unsigned char	long	0 扩展
unsigned char	unsigned short	0 扩展
unsigned char	unsigned long	0 扩展
unsigned char	float	转换到 long; 再从 long 转换到 float
unsigned char	double	转换到 long; 再从 long 转换到 double
unsigned char	long double	转换到 long; 再从 long 转换到 double
unsigned short	char	保留低位字节
unsigned short	short	最高位作为符号位
unsigned short	long	0 扩展
unsigned short	unsigned char	保留低位字节
unsigned short	unsigned long	0 扩展
unsigned short	float	转换到 long; 再从 long 转换到 float
unsigned short	double	转换到 long; 再从 long 转换到 double
unsigned short	long double	转换到 long; 再从 long 转换到 double
unsigned long	char	保留低位字节
unsigned long	short	保留低位字节
unsigned long	long	最高位作为符号位
unsigned long	unsigned char	保留低位字节
unsigned long	unsigned short	保留低位字节
unsigned long	float	转换到 long; 再从 long 转换到 float
unsigned long	double	Convert directly to double
unsigned long	long double	转换到 long; 再从 long 转换到 double

变量就是机器一个内存位置的符号名，在该内存位置可以保存数据，并可通过符号名进行访问。变量有三个特征：

◇ 每一个变量有一个名字，其命名规则与标识符相同。

◇ 每一个变量有一个类型。

◇ 每一个变量保存一个值。如果需要变量保存某一个值，就把该值赋给变量。

为了提高程序的可读性，给变量命名时，应该注意使用有意义的名字，以帮助描述变量所保存的值。最好是一开始就坚持使用小写字母。例如：要用一个变量保存工资总额，可以使用 total_payroll，也可以使用 XYZ34 作变量名，它们都是合法的名字。但使用前者比使用后者就更好，因为从变量名，就能大致知道它保存了什么样的值，便于记忆。

在使用一个变量之前，必须先定义。变量的定义的一般格式如下：

数据类型 变量名 1[= 初始值 1], 变量名 2[= 初始值 2], ...;

"数据类型"是指 C++有效的数据类型，如 int、double、long 等。方括号中的内容是可选的，它是在变量定义时，给变量赋初值。下面是几个变量定义的实例：

long n; //定义了一个长整型变量 n，未赋初值

double a, b = 0.5; //定义双精度型变量 a、b，a 未赋初值，b 的初值为 0.5

下面是一个使用变量的程序实例：

程序 2-3：

```
#include <iostream.h>
int main (void)
{
  1.  int workDays = 5;
  2.  float workHours, payRate, weeklyPay;
  3.  workHours = 7.5; payRate = 38.55;
  4.  weeklyPay = workDays * workHours * payRate;
  5.  cout << "Weekly Pay = " << weeklyPay << '\n';
}
```

第一行定义了一个整型变量 workDays，并初始化为 5，表示一周的工作天数。

第二行定义了三个实型变量，分别表示每天工作的小时数、每小时应支付的薪水及每周应支付的薪水。

第三行是两个赋值语句。7.5 赋给变量 workHours，38.55 赋给变量 payRate。

第四行是计算每周的工资，即三个变量 workDays、workHours、payRate 的积，*是乘运算符，结果保存在变量 weeklyPay 变量中。

第五行输出字符串"Weekly Pay = "、变量 weeklyPay 的值和换行符。

本程序的运行结果如下：

```
Weekly Pay = 1445.625
```

如果我们定义一个变量时，没有给它赋初值，它的值是不定的。例如，在上面的第四行执行前，weeklyPay 的值是不定的。

变量第一次赋值称之为初始化，变量在使用之前应当先初始化。

2.4.1 整型

整型变量可用 short、int、long 定义，区别在于 int 占用机器的字节数比 short 多或一样，而 long 占用的字节数比 int 多或一样。例如：

short age = 20;

int salary = 65000;

long price = 4500000;

缺省的情况下，整型变量假定为有符号的，但是，使用 unsigned 关键字，也可以把整型变量定义为无符号的。当然，定义整型变量时，也可以使用 signed 关键字，但是多余的。

unsigned short age = 20;

unsigned int salary = 65000;

unsigned long price = 4500000;

一个整型数，例如 1984，总是被假定为 int 类型，除非有 L 或 l 后缀，才被处理为 long int 类型。同样，一个整型数也可以加 U 或 u 后缀，指定为 unsigned 类型。例如表 2-6：

整型数能被表示成 10 进制、8 进制和 16 进制。如果一个整型数有前缀 0，则表示是 8 进制数，有前缀 0x 或 0X，表示是 16 进制数。例如：

92 // 十进制

0134 // 8 进制

0x5C // 16 进制

8 进制数只能使用数字 0~7，16 进制数可用 0~7 及 A~F (或 a~f)

表 2-6

1984L	1984l	1984U	1984u	1984LU	1984lu

整型是常用的一种数据类型，但是，它的大小是不固定的，这是由操作系统决定的。在计算机中任何信息都是以二进制的形式存储的，二进制数每一位是 0 或 1，八位组成一个字节（byte），两个字节组成一个字（word），四个字节组成一个双字（dword）。

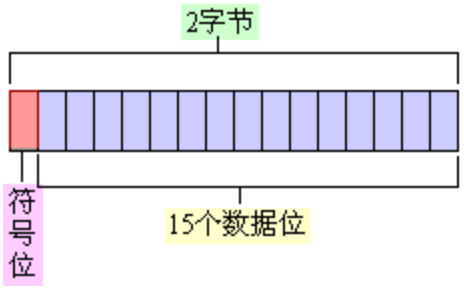
整数能存储的最大值是由计算机给它分配的存储空间的大小决定的，而整数所占的存储空间因不同的计算机而异。例如，某种计算机可能用 16 位（两个字节）来存储一个整数，而另一种计算机则可能用 32 位（四个字节）来存储。

当一个整数存储在计算机中时，左起第一位叫做第 0 位，即符号位，它用来表示该数是正数或负数。如果是负数，符号位被置 1，正数则为 0。

两个字节能表示的整数范围是-32768～32767， 四个字节能表示的整数范围则增加到-2147463648~2147483647。

由于整数所占的存储空间在不同的计算机上是不同的，所以在代码移植时容易出现问题。

图 2-3 2 字节如何表示整型值



2.4.2 浮点型

浮点型变量可用 float 或 double 定义，后者在机器内存中占用的字节数更多，所以有效位数更多，表示的数字范围也更大。例如：

float interestRate＝0.06; double pi＝3.141592654;

一个实型数，如 0.06 总是假定为 double 型，除非有 F 或 f 后缀，才被处理为 float 型。若有 L 或 l 后缀，则被处理为 long double 型。例如表 2-7：

实型数也可以表示成指数形式。例如：0.002164 可以写成：

2.164E-3 或 2.164e-3

E 或 e 代表指数。

表 2-7

1984L	1984l	1984U	1984u	1984LU	1984lu

浮点型分为单精度浮点型和双精度浮点型，就是所谓的 float 和 double。下表列出了它们的主要数据：

表 2-8

类型	字节数	有效位（十进制）	指数长度	尾数长度
float	4	6－7	8bits	23bits
double	8	15－16	11bits	52bits

浮点数是由符号位、指数位和尾数位构成的，上表已列出浮点型的指数位长度和尾数位长度。以单精度浮点数为例：它是由 1 个符号位、8 位指数位和 23 位尾数组成，共 32 位，4 个字节。

2.4.3 字符型

字符变量用 char 定义，一个字符变量仅占用一个字节，存放该字符的编码。该编码是一个数字，并依赖于所使用的编码系统（即与机器相关）。最普通的编码是 ASCII（American Standard Code for Information Interchange），即美国信息交换标准编码。例如：字符 A 的 ASCII 码为 65，字符 a 的 ASCII 码为 97。

```
char ch = 'A';
```

象整型数一样，字符变量也可被指定为 signed 或 unsigned。大多数机器上，char 与 signed char 意义相同。有些机器上，char 可能与 unsigned char 意义相同。一个有符号字符变量可以保存-128 ~127 之间的整数，而一个无符号字符变量可以保存 0 ~255 之间的整数，它们均可用于表示小整数，象整型数一样赋值：

```
signed char offset = -88;
unsigned char row = 2, column = 26;
```

一个字符是用一对单引号包围起来，例如：'A'。C++中还有一些不能打印的特殊字符，称之为转义字符。例如 2-1：

字符也可以用它们的编码指定，转义字符通常用三位 8 进制数表示，例如 2-2：（假定为 ASCII 码）：

C++的字符由下列字符组成。

1、大小写英文字母

a~z，A~Z

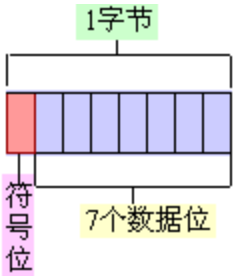
2、数字字符

0~9

3、特殊符号

空格 ！ # % ^ & * _ - + =

图 2-6 2 字节如何表示字符型值



C++语言把字符型变量当作一个较小的整型量，可以象整型量一样使用它。下面举一个例子说明：

程序 2-6

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

```
void main()
{
   char c1 = 'a';
   char c2 = 'b';
   char c3,c4;
   c3 = c1 - ('a' - 'A');
   c4 = c2 - ('a' - 'A');
   cout<<c3<<c4<<endl;
}
```

运行结果为：

```
A B
```

程序中的'a' - 'A'是大小写字母之间的差值，其值为 32。所以我们也可以把程序写成：

```
c3 = c1 - 32;
c4 = c2 - 32;
```

如果我们想看一看 c3、c4 中到底存储的值是多少，可以这样：

```
cout<<(int)c3<<(int)c4<<endl;
```

其运行结果为：

```
65 66
```

注意：在内存中，字符数据以 ASCII 码存储，即以整数表示，'0'和 0 是不同的。

例 2-1：

```
'\n' // 换行
'\r' // 回车
'\t' // 水平 tab
'\v' // 垂直 tab
'\b' // 退格
'\f' // 进纸
'\'' // 单引号 (')
'\"' // 双引号 (")
'\\' // 反斜杠 (\)
```

例 2-2：

```
'\12' //换行(10 进制编码 = 10)
'\11' // 水平 tab (10 进制编码= 9)
'\101' // 'A' (10 进制编码= 65)
'\0' // null (10 进制编码= 0)
```

2.4.4 字符串

字符串是一个连续的字符系列，有一个'\0'字符结尾。假定有一个字符串为"HELLO"，它在内存中的存储，参见图 2-2。

一个字符串用一对双引号包围起来，例如："HELLO"，编译器在每一字符串的结尾增加'\0' 结尾符。字符串可以由任意字符组成，例如：

```
"Name\tAddress\tTelephone" // tab-分隔字符
```

"ASCII character 65: \101" // 'A' 用'101'指定

一个长字符串可以占两行或多行，但在最后一行之前的各行应用反斜杠结尾，例如：

"Example to show \
the use of backslash for \
writing a long string"

上面的字符串与下面的单行字符串等价：

"Example to show the use of backslash for writing a long string"

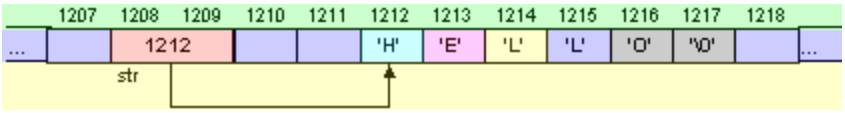
需要注意的是：

（1） 不要把字符与字符串混淆了。例如："A"与'A'不等价，前者由两个字节组成（字符'A'与字符'\0'），而后者只有一个字节。

（2） 最短的字符串是空字符串("")，它仅由一个结尾符'\0'组成。

（3） C++中，没有字符串类型，而是用字符数组来处理字符串，这将在第六章中介绍。

图 2-7 在内存中的字符串



字符串常量与字符常量不同，字符串常量是用一对双引号括起来的零个或多个 ASCII 字符的序列，并以 NULL（ASCII 码值为 0）结束。例如：

例 2-3：

"This is a character string."
"CHINA"
"0123456789"
"a"
"$10000.00"
" "（引号中有一个空格）
""（引号中什么也没有）
"\n"（引号中有一个转义字符）

字符串常量在内存中存储时，系统自动为每个字符串常量的尾部加一个字符\0，用以标识这字符串的结束，如字符串

"CHINA"

在内存中的形式是：

表 2-9

C	N	I	N	A	\O

了解这一点，我们就可以区分字符串"a"和字符'a'有何不同了。它们在内存中的形式分别为：

2.4.5 布尔型

布尔类型变量用 bool 关键字定义，其值为 true 或 false，true 和 false 都是 C++的关键字。整型值也可赋给 bool 变量，但会被隐式转换成 true 或 false。转换规则如下：整型值非 0 时，转换为 true，整型值为 0 时，转换为 false。

同样，布尔值也可赋值给整型变量，但要进行隐式转换，转换规则为：布尔值为 true 时，转换为 1，布尔值为 false 时，转换为 0。

布尔值也可插入输出流 cout，当布尔值为 true 时，在屏幕上打印 1，当布尔值为 bool 时，在屏幕上打印 0。例如 2-5。

C 没有 bool 这个数据类型，C++中定义了 bool 类型，它只有两个值：true 和 false。所有的条件表达式返回类型都是 bool 型，比如 n!=0（!=是"不等于"运算符，我们在本章后面要介绍）根据不同的 n，返回 true 或 false。true 和 false 有如下关系：

!true==false;

!false==true;

！是取反运算符，我们在本章后面要介绍。我们看一个简单的例子："if (exp1) statement1;"。这个语句的意思是：如果表达式 exp1 的值是 true，就执行 statement1，否则就不执行。

如果对一个 bool 类型变量使用前缀或后缀自增（＋＋）运算符，不管这个变量是 false 还是 true，这个变量将变为 true。但是不能对 bool 类型变量使用前缀或后缀自减（——）运算符（有关自增、自减运算符的内容在本章的后面介绍）。如：

例 2-4：

```
#include "iostream.h"
int main(int argc, char* argv[])
{
   bool b;
   b=false;
   cout<<b++;
   cout<<b++;
   cout<<b;
   return 0;
}
```

程序运行结果为：

011

一个右值为 bool 型变量将被自动转换成整型，且 false 被转换成 0，true 被转换成 1。所以上面程序中输出的是 011，代表的意思是 false、true、true。

C++中还有一种比较特殊的类型，它用于定义函数返回类型，或者用于定义"一般类型"的指针，即该指针可以指向任意类型数据（被 const 和 volatile 修饰的变量除外），但是不能定义空类型的变量。例如，

函数声明：void func();

指针定义：void *p;

例 2-5：

```
cout << "A true value: ";
cout << true;
cout << endl;
```

```
cout <<"A false value: ";
cout <<false;
cout <<endl;
```

输出结果为：

A true value: 1

A false value: 0

在进行程序设计时，常常需要常量，例如圆周率 π＝3.1416。在需要常量的地方，直接使用常量的数值的方法非常不好，例如：

```
//计算圆的面积
s＝3.1416*r*r;
...
```

如果我们需要提高计算精度，将 π 的值改为 3.1415927 进行计算，我们就不得不将程序中所有的 π 值从 3.1416 改为 3.1415927，这不仅繁琐，更重要的是很容易出错。

C++允许定义符号常量，定义常量的一般形式为：

```
const 类型 名字 = 值;
```

其中，"类型"是指常量的类型，如 short、long、double 等，"名字" 是指常量的名字，而"值"是指赋给常量的、合适的数据类型的数值。 参看例 2-7。

下面给出一个有常量定义的实例程序，这个程序是打印给定半径的圆的面积和周长。

例 2-6：

```
void main()
{
  const double PI＝3.1415926535898; //定义圆周率常量 PI
  double radius; //定义圆半径变量
  double area; //定义圆面积变量
  double circumference; //定义圆周长变量
  cout <<"Enter radius : ";
  cin >>radius;
  area = PI*radius*radius;
  circumference ＝2.0*PI*radius;
  cout <<"Area of circle of radius " << radius << " is "
  << area << " \n";
  cout <<"and its circumference is " << circumference
  << "
}
```

运行该程序，并输入半径值 10.0：

```
Enter radius : 10.0↙
```

则输出结果为：

```
Area of circle of radius 10.0 is 314.159
and its circumference is 62.8319
```

常量就是在程序运行过程中其值不发生变化的量。常量可分为直接常量与符号常量。直接常量就是平常我们所说的常数，例如：

`r` // r 为字符型直接常量

3.1415926 // 3.1415926 为双精度直接常量

符号常量就是用一个标识符代表某个常量。符号常量可用关键字 const 声明，其格式如下：

const 数据类型 常量名＝常数值；

例如：

const int a= 1234; //定义 a 为整型常量其值为 1234

const char b=`a` //定义 b 为字符型常量其值为 a

在程序设计中，尽量使用符号常量来代替常数，这是一种好的编程习惯，这样可以增加程序的可读性、可维护性。例如，在数值计算中，我们会经常遇到一些常量，比如圆周率。如果把它定义成符号常量，当需要更改常量值的时候，只需要更改符号常量的定义语句即可。

也可以使用预处理语句定义符号常量，例如我们用：

#define PI 3.14159

定义符号常量 PI，然后在程序中使用 PI，跟使用常数 3.14159 的效果是一样的。编译器在编译时，把符号 PI 替换成 3.14159，当需要修改 PI 的值时，只需要修改上面的语句即可。但是我们不推荐这样做：因为预定义符号与符号常量不同，在编译时使用常数替代了所有的预定义符号，这样在代码中相应位置实际都是常数。程序中过多的常数会导致程序代码量变大，而且在多个源文件中定义了同样的符号，会产生符号重定义的问题。使用常量优于＃define 宏，优点在于可指定类型信息。

例 2-7：

```
const double PI = 3.1415927; //圆周率 π
const long number = 49L;
const char TAB = '\t';
const char QUERY = '?';
const double SOMENUM = 1.235E75;
```

我们常常需要一些整型常量表示某一个的数据范围。例如，假定需要表示汽车的颜色：

```
const int cRED = 0;
const int cBLUE = 1;
…
int auto_colour;
auto_colour = cBLUE;
```

上面的程序语句是用整型变量 auto_colour 表示颜色，下面的语句也是合法的：

```
auto_colour = -1;
…
```

显然，-1 并不是所需要的颜色值。如果能够限定 auto_colour 在限定的整数范围（即规定的几种颜色集合）内取值就比较好，C++中的枚举类型能够让我们做到这一点。枚举类型是一种用户定义的数据类型，其一般定义形式为：

```
enum 枚举类型名
{
  标识符[=整型常数],
  标识符[=整型常数],
  ...
  标识符[=整型常数],
};
```

"枚举类型名"右边花括号中的内容称之为枚举表，枚举表中的每一项称之为枚举成员，枚举成员是常

量，也就是说，枚举表是整型常量的集合。枚举成员之间用逗号隔开，方括号中的"整型常数"是枚举成员的初值。

如果不给枚举成员赋初值，即省掉了标识符后的"=整型常数"时，则编译器为每一个枚举成员给一个不同的整型值，第一个成员为 0，第二个为 1，等等。当枚举类型中的某个成员赋值后，其后的成员则按依次加 1 的规则确定其值。下面是一个枚举类型定义实例：

enum Colour { eRED, eBLUE, eYELLOW, eGREEN,

eSILVERGREY,eBURGUNDY };

这样就定义了一个称之为 Colour 的枚举类型，编译器给枚举成员 eRED，…，eBURGUNDY 的值分别为 0，…，5。 定义了枚举类型，就可以定义枚举类型的变量，其值限定为某一个枚举成员。例如：

Colour auto_colour;

…

auto_colour = eBURGUNDY;

Colour 类型变量 auto_colour 只能取 eRED，…，eBURGUNDY 六个值中的一个，下面的语句将会出现编译错误：

auto_colour = 4;

我们也可以在定义枚举类型时，为枚举成员指定初值，例如：

enum Colour { eRED=5, eBLUE, eYELLOW, Egreen=30,

Esilvergrey=40,eBURGUNDY };

这时，eBLUE=6、Eyellow=7、Eburgundy=41。

由上述几种基本数据类型通过各种组合可以得到复合数据类型，这些数据类型在以后章节会有详细讲解。

如果一个变量只有几种可能的值，就可以把它定义为枚举类型。"枚举"，顾名思义，就是把这种类型数据可取的值一一列举出来。

例如，可以定义一个表示日期的枚举类型 weekday：

enum weekday{sun,mon,tru,wed,thu,fri,sat};

这里 enum 是关键字，用于定义枚举数据类型，weekday 是枚举类型的名字，枚举元素用标识符表示。

接下来我们可以用 weekday 来定义变量，例如：

weekday today,nextday;

C++语言也允许在定义枚举类型的同时定义枚举变量，例如：

enum weekday{sun,mon,tru,wed,thu,fri,sat}today,nextday;

这样，变量 today 和 nextday 就具有 weekday 类型，它们的取值只能是 sun，mon，…，sta，例如可有：

today = sun;

nextday = mon;

if(today == sat)

nextday = sun //如果 today 是 sat 的话，那么 nextday 赋值为 sun

C++编译器对枚举元素实际上是按整型常数处理。所以，就象前面所说的那样，当遇到枚举元素列表时，编译程序就把其中第一个标识符赋 0 值，第二、三、…个标识符依次赋 1、2、…。因此，当枚举元素赋给枚举变量时，该变量实际得到一个整数值。例如：

today = sun;

是将 0 赋给 today，而不是将字符串"sun"赋给 today。赋值后：

cout<<today;

语句的输出结果是 0。

我们也可以在枚举类型定义时指定枚举元素的值，例如：

enum weekday{sun = 7,mon = 1,tue,wed,thu,fri,sat};

这时 sun 的值是 7，mon 的值是 1，而 tue 以后各元素的值，从 mon 的值开始，每次递增 1，即 tue 的值为 2，wed 的值为 3，…。如果不写 mon = 1，则 mon 的值为 8，tue 的值为 9，依此类推。

由于枚举元素是常量，所以我们在程序中不能对它们进行赋值，如"sun = 0；mon = 1；"将产生错误。

既然枚举元素就是整型值，那么它有什么存在的必要呢？至少有两个原因：一个是用标识符表示数值增加了程序的可读性，例如：

if(today == sat)

next = sun

就比

if(today == 6)

nextday = 0;

清楚多了；另一个重要的原因是：它限制了变量取值的范围，如现在 today 只能取 sun~sat 中的值。

注意：enum 在 C++和 C 中使用的不同。下面的 C 语言语句定义了名为 day 的变量：

enum Days { Monday, Tuesday, Wednesday, Thursday, Friday };

enum Days day;

在 C++中，定义为：

enum Days { Monday, Tuesday, Wednesday, Thursday, Friday };

Days day;

C++中定义了枚举类型数据后，程序可以把它当作一种类型使用，不需在类型名 Days 前加 enum 关键字。

C/C++语言本身并不带输入和输出（即 I/O）功能，而是提供了输入输出库，也称为 I/O 库。通过 I/O 库，我们可以完成输入和输出的操作。大多数 C 程序使用一种称为 stdio（标准 I/O）的 I/O 库，该库也能够在 C++中使用。但是，在 C++程序中，一种称为 iostream（I/O 流库）的 I/O 库用得更多。

在 C++中，I/O 使用了流的概念-字符（或字节）流。每一个 I/O 设备传送和接收一系列的字节，称之为流。输入操作可以看成是字节从一个设备流入内存，而输出操作可以看成是字节从内存流出到一个设备。流是输入输出设备的另一个名字-一个文件、屏幕、键盘等。要使用 C++标准的 I/O 流库的功能，必须包括两个头文件：

#include<iostream.h>

#include<iomanip.h>

iostream.h 文件提供基本的输入输出功能，iomanip.h 文件提供格式化的功能。通过包含 iostream 流库，内存中就创建了一些用于处理输入和输出操作的对象。标准的输出流（通常是屏幕）称为 cout，标准的输入流（通常是键盘）称为 cin。

输出变量 d 的值到标准输出设备的语法形式如下：

cout << d;

注意：（<<）是双小于号，不是左移操作符，它是一种输出操作符，指出程序哪个流发送数据。

本语句表示传送 d 的值到标准的输出设备（由 cout 表示）。理解这个语句的一种方法是调用了函数 operator<<，d 是函数参数（关于函数调用，在第五章学习）。明确地说，为了输出 d 的值到 cout，可以使用语句：

cout.operator<<(d);

对于 C 语言的程序中，使用的是 stdio（标准 I/O）库，在这个库中不仅定义了面向控制台（显示器和键盘）的输入输出，还分别定义了文件输入输出函数和面向内存的输入输出函数。在 C++程序中，我们最常使用的是 iostream（I/O 流库），它是基于面向对象的，可以实现 stdio 库的所有功能，通过它统一了标准 I/O、文件和存储块的函数接口，使得对所有设备的操作看上去都一样，隐藏了内部实现。与标准 C 输入输出库的各种各样的函数相比，输入输出流更容易、更安全、更有效。为了保证兼容性，I/O 流类和 C 标准输入输出函数库是可以共同使用的。

使用 stdio 库，需要在程序中包含 stdio.h 头文件，若使用 iostream 库，需要包含 iostream.h 头文件，如果要带参数的格式化输入输出，还需要包含 iomanip.h 头文件。

使用 stdio 库的标准输入输出，要使用 scanf 和 printf 函数，scanf 函数是从标准输入（键盘）设备读入，printf 是向标准输出设备（显示器）输出。

iostream 库在 iostream.h 中预定义了四个全局的流对象：cout、cerr、clog 和 cin，用于标准输出和输入，cout 和 cin 在程序设计中会经常用到。cout 流对象控制向控制台（显示器）的标准输出，cin 控制从控制台（键盘）输入。

2.7.1 标准输出

输出内置的数据类型到标准的输出设备，用<<运算符和 cout 输出流。例如：

```
cout << d; // 输出 d
cout << d << endl; // 输出 d 并换行
cout << "This is the value of d : " << d << endl; // 输出字符串、d 并换行
```

一个语句中多次使用<<等价于<<单独使用多次，即最后一个语句等价于：

```
cout << "This is the value of d : " ;
cout << d;
cout << endl;
```

符号 endl 的功能是换行，并清除输出缓冲区。变量的值以缺省的格式打印，改变打印格式方法在后面介绍。

C 语言的转义字符在 C++中仍然有效，所以在 iostream 中，我们可以使用转义字符，比如要达到回车的效果，我们可以使用：

```
cout<<endl;
```

也可以使用：

```
cout<<'\n';
```

如果实现简单的格式化输出，我们可以使用制表符：

```
cout<<"name\tage\tsex"<<endl;
```

2.7.2 标准输入

标准输入的用法与标准输出类似，使用>>运算符和 cin 输入流。语句：

```
cin >> d;
```

是从标准输入读（或抽取）一个值到变量 d，并与语句中的数据类型匹配。例如：如果 d 是整型数，上面的命令读数字，直至没有遇到数字为止；如果 d 是浮点数，该命令读数字、小数点、指数，直至没有遇到合适的字符为止。

假如把输入和输出语句结合在一起使用，可能会发生问题。我们编写程序时，常常先用一个输出语句打印提示，然后要求用户输入数据。对于一些 C++编译器，在输入语句抽取数据前，应先输出一个换行符。例如 2-9：

否则，可能达不到目的

经验：如何记<<和>>:cout 是 c 出，出了就小（<<）了； cin 是 c 进，进则收入，就大(>>)

注意：cin 流不使用指针引用变量，不应给 cin 传递一个指向变量的指针，如果这样做 cin 将返回错误。例如，下面的程序 cin_err.cpp ,当编译它时将返回六个错误：

例 2-8：

```
#include <iostream.h>
void main(void)
```

```
{
  int age;
  float salary;
  char name[128];
  cout << "Enter your first name age salary: ";
  cin >> &name >> &age >> &salary;
  cout << name << " " << age << " " << salary;
}
```

例 2-9：

```
cout << " Enter the value of n : " << endl;
cin >> n;
```

而不要写成：

```
cout << " Enter the value of n : ";
cin >> n;
```

2.7.3 格式化

有两种方法设置变量的输出格式：一种方法是直接设置输出流的格式状态，另一种方法是通过输入输出操纵符。

如果选择前者，需要设置与输出流相关的标志，此后，任何数据输出到该流，将按照标志决定的格式输出；如果选择后者，可将格式化的命令嵌入到输入输出语句中。

前者也许更好，因为，我们常常为所有的输出选择一种格式，也就是说，常常不需要大量的、不同的格式。与输出流格式状态相关的标志有许多，我们仅讨论最常用的两种：精度（precision）和宽度（width）。精度是指小数点后的数字位数，宽度是指变量输出的总位数。设置精度和宽度的形式如下：

```
cout.precision(4);
cout.width(10);
```

用下面的语句，我们可以得到当前的精度和宽度：

```
n = cout.precision();
m = cout.width();
```

一旦设置了 precision，在下一次设置以前，原设置值保持不变。width 有所不同，它是数据输出的最小位数，如果宽度不够，则会分配更多的位数。也就是说，当 width 位数不够时，输出的数据不会被截断。而且输出操作完成后，width 会恢复为缺省值。还有一些标志可以设置浮点数的固定或指数表示、输出值的对齐方法（右、左或中间）、是否显示尾部 0，这些标志用成员函数 setf 设置。例如，假定要使用标准输出，指数表示、左对齐及显示尾部 0，可以使用下面的语句：

```
cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::showpoint, ios::showpoint);
cout.setf(ios::scientific, ios::floatfield);
```

使用 precision 域可以控制输出的精度。对于整数，precision 域用来控制输出结果的长度，如果整数值长度小于设置的精度，则输出结果的前面补 0。对于浮点数，precision 域用于控制小数点后面的位数。

例如：

```
printf("%+10.5f",f);
```

输出浮点数 f，带符号，宽度是 10 位（包括小数点），小数位是 5 位。

控制符 L 或 l 可以将整数设置为长整型输出，h 可以将整数作为短整型输出。

2.7.4 printf 和 scanf 输出输入

printf 和 scanf 是标准的 C 输出输入语句，在 C 语言标准的头文件 stdio.h 中定义。但在 C++中，它们也能够被使用。

一、 printf 输出

printf 语句的一般形式如下：

printf("格式字符串", 表达式, ...);

该语句将"表达式"按照"格式字符串"给定的格式，显示在屏幕上。"格式字符串"包括文本字符、转义字符和格式说明符。

如果我们只要打印简单的信息，并不需要包括"表达式"，例如，要打印信息："今天是星期二"，可以使用下面的 printf 语句：

printf("Today is Thursday");

这个语句执行的结果为：

```
Today is ThursdayPress any key to continue
```

需要注意的是：输出的信息与系统提示连在一起。为了解决这个问题，可以使用转义字符，常用的转义字符参见 2.4.3 节的内容。

C++使用操纵符（在<iomanip.h>和<iostream.h>中定义），而 C 提供了一系列的说明符，参见表 2-10。这些说明符用于输出，也可用于输入。

表 2-10：

说明符	类型
%wd	int 或 char
	w = 宽度
%w.df	double w = 总的宽度（包括小数点） d = 小数位数
%wc	char 或 int w = 宽度
%ws	char * (字符串)格式 w = 宽度
%wu	unsigned int w = 宽度
%wo	八进制
%wx	十六进制
%m.dddddde±xx %m.ddddddE±xx %m.dddddg±xx %m.ddddddG±xx	指数记数法

下面，我们给出一些格式实例，假定有下面的变量说明：

表 2-11：

变量说明
int x = 98;

```
float y = 1.34567889;
char letter = 'A';
char word[] = "Hello";
```

下表给出了输出这些变量的缺省的格式：

表 2-12：

语句	printf("Default formats: %d %f %c %s\n", x,y,letter,word);
结果	Default formats: 98 1.345679 A Hello

我们也可以改变缺省的格式。需要注意的是：输出结果是右对齐的，也就是说，当输出结果的宽度小于指定的宽度时，空格填充到左边：

表 2-13：

格式说明	
语句	printf("Width specified: %5d %10.4f %5c %10s\n", x,y,letter,word);
结果	Width specified: 98 1.3457 A Hello

我们也可以打印变量的内存地址信息，但要使用 unsigned 格式说明符：

表 2-14：

打印内存地址	
语句	printf("Address of x: %u Address of y: %u\n", &x, &y);
结果	Address of x: 4026528400 Address of y: 4026528396

表达式按照格式字符串指定的格式输出。当格式字符串中没有格式说明时，printf 函数就相当于输出一个普通的字符串。 例如：

printf("this is a string.");

格式说明符总是以一个百分号（%）引导，当 printf 函数遇到第一个格式说明符时，用第一个表达式的值来替换它，遇到第二个格式说明符时，就用下一个表达式的值替换，依此类推。如果表达式个数多余格式说明符个数，则多余的参数就被抛弃。格式说明符的格式是：

%[flags][width][.precision][{h|l|L}]type

每个域是由一个字符或者一个数组成，方括号扩起的域是可选项，最基本的格式是：%type。

type 域字符：

表 2-19：

转换字符	类型	含义
C	int	输出一个字符
C	int	同上

D	int	输出有符号十进制整数
I	int	同上
O	int	输出无符号八进制整数
U	int	无符号十进制整数
X	int	无符号十六进制整数
X	int	同上
e	double	以科学计数法输出浮点数
E	double	同上
F	double	输出十进制浮点数
G	double	等效于%e 和%f，取两者之中较短的
G	double	同上
P	void 指针	输出指针地址，格式如下：xxxx:yyyy ，xxxx 是段地址，yyyy 是偏移量，它们都是十六进制数
S	char*	输出字符串

例如:

printf("%d%x", n, n)

是将 n 分别按 10 进制和 16 进制整数输出。

我们可以用 flag 任选符来改变输出的格式，

表 2-20：

Flag	含义	缺省
-	左对齐	右对齐
+	如果结果是有符号数，则显示符号	只显示负号
0	在结果前面填充 0	无填充
空格(' ')	如果结果是有符号数且为正，则在前面填充一个空格	不填充空格

例如：

printf("%+d",n);

输出整数 n 时显示符号（包括正号）。

可以使用 width 域设置输出的宽度，如果输出的宽度小于设置的宽度，用空格填充，如果大于设置的宽度，

则设置的宽度将被忽略，而不会将结果裁剪。

例如：

printf("%+10d", n);

输出整数 n 时显示符号，且宽度为 10 位，如 n 不够 10 位，则用空格填充，否则按实际位数显示。

二、 scanf 输入

scanf 语句的一般格式如下：

scanf("格式字符串", 地址，…);

scanf 语句用"格式字符串"控制键盘读入的方式。"格式字符串"中一般只包括格式说明符，它们与 printf 语句中的一样，而"地址"是指变量在内存中的位置。我们看看下面的程序实例，它定义了一个 int、float、char 和 char []变量，并使用了格式说明符：

程序 2-4：

```
#include <stdio.h>
#define MAX_WORD 20
void main()
{
   int x;
   float a;
   char ch, empty;
   char word[MAX_WORD];

   printf("Enter an integer: ");
   scanf("%d", &x);

   printf("Enter a float: ");
   scanf("%f", &a);

   fflush(stdin); // 清空输入缓冲区
   printf("Enter a character: ");
   scanf("%c", &ch);

   printf("Enter a string: ");
   scanf("%s", word);

   printf("Your integer was: %d\n", x);
   printf("Your float was: %f\n", a);
   printf("Your character was: %c\n", ch);
   printf("Your word was: %s\n", word);
}
```

这个程序有几个有趣的特征，总结如下：

（1） 地址。整型、浮点型和字符型变量需要使用地址运算符&，而字符串不需要，因为数组名就是数

组在内存中的起始地址，有关数组和字符串的内容，我们在第六章详细介绍。

（2） fflush() 函数。我们在读字符之前，调用 fflush() 函数，这个函数清空标准的输入设备缓冲区（stdin）。如果要删除缓冲区中回车换行符，就必须清空缓冲区。因为它们也是字符，如果不在读入之前清除它们，就不能正确地读入字符。

（3） 字符串。我们输入字符串"hello there"，但读到空格时，scanf()就停止了，仅"hello"存入了字符串。下面是说明这种现象的一个实例，运行该程序，并输入。参看例 2-10：

如果需要读取带空格的字符串，可以使用函数 gets 来做这件事情。gets 的作用是读取一行字符到一个字符串变量中。它会读取空格，直到遇到'\n'或'\0'时才返回。

例 2-10：

Enter an integer: 10↙

Enter a float: 1.23456↙

Enter a character: c↙

Enter a string: hello there↙

运行结果为：

Your integer was: 10

Your float was: 1.234560

Your character was: c

Your word was: hello

格式说明符的形式是：

%[*][width][{h|l|L}] type

scanf 的格式说明符与 printf 的类似，最简单的形式如下：

%type

type 域标识要读取的数据类型，它与 printf 函数的字符是一样的。Width 是读取的最大字符数，例如，数据为"1.234456678"，我们使用：

double f;

scanf("%5lf",&f);

那么它只会读取 5 个字符，即 1.234 到变量 f 中。

注意：用 scanf 读取字符串的时候，如果字符串中存在空格，那么它将在空格处将该字符串裁掉，只取前面部分。scanf 在遇到空格，制表符时会认为数据已经读取完毕，并将读取的数据转换成相应类型，存储到地址所在的存储空间中。

【本章小结】

本章讲述了 C++语言的基本知识。

◇ 在 C++语言中，语句、变量、函数、预处理指令、输入和输出等，是重要的概念，应该在编程实践中逐渐掌握这些概念和它们的应用。

◇ 标识符是用来标识变量、函数、数据类型等的字符序列。C++中的标符可以由大写字母、小写字母、下划线（_）和数字 0~9 组成，但必须是以大写字母、小写字母或下划线（_）开头。C++语言中预定义了一些标识符，称之为关键字，它们不能被再定义。

◇ 布尔型、字符型、整型、浮点型和空类型是基本数据类型。指针、数组、引用、结构和类可以通过基本数据类型进行构造，称之为复合数据类型。

◇ 变量就是机器一个内存位置的符号名，在该内存位置可以保存数据，并可通过符号名进行访

问。为了提高程序的可读性，给变量命名时，应该注意使用有意义的名字。变量第一次赋值称之为初始化，变量在使用之前应当先声明。

◇ 常量是在程序运行过程中，其值不能改变的量。

◇ C++语言本身没有输入输出功能，而是通过输入输出库完成 I/O 操作。C 程序使用的 stdio（标准 I/O）I/O 库也能够在 C++中使用；另外 C++语言还提供了一种称之为 iostream（I/O 流库）的 I/O 库。

【学习目标】

熟练掌握 C++各种运算符的使用方法。能熟练运用运算符组成需要的表达式。正确使用操作数的类型转换和运算符的优先级顺序和结合性。

【重点和难点】

自增、自减运算符的运用；各个运算符的优先级顺序及其结合性；操作数的类型转换。

【学习方法指导】

要在学习中注意区分运算符、表达式和语句的不同；不同类型的操作数赋值时，尽量进行显式转换，隐式转换容易犯错误；优先级和结合性也容易犯错误，一般说来，表达式中加上圆括号，比较保险，这也可以提高程序的可读性。

【知识点】

算术运算符；自增、自减运算符；关系运算符；逻辑运算符；位运算符

赋值运算符；求字节数运算符；逗号表达式；短路表达式；显式类型转换；隐式类型转换；优先级与结合性

所谓运算符就是指运算的符号，例如加运算符（+）、乘运算符（*）、取地址运算符（&）等。表达式与运算符密不可分，它由运算符与操作数组合而成，并由运算符指定对操作数要进行的运算，一个表达式的运算结果是一个值。本章中，我们将介绍 C++语言基本的运算符、表达式，各种运算符的优先级、结合性，以及不同数据类型的值之间的转换。

C++语言中的运算符是可以让 C++语言编译器能够识别的具有运算意义的符号。编译器把这些符号及其组成的表达式翻译成相应的机器代码，就可以由计算机运行得出正确的结果。

其实，就象我们日常生活当中许多东西的名字，如冰箱、电视机等分别代表不同功能的电器设备一样，运算符就是代表 C++语言中的各个运算功能的名字，这些名字是由制定 C++语言规范的人员确定的。下面给出由运算符组成的表达式的例子：

下面是几个表达式例子。如：

100+200-300*200+1000/20+100%10

a || b && c || d

a = b + c + d*e

a += b++

c -= d--

只要是按 C++语法写出的表达式，编译器就能够解释其中的运算符和由运算符、操作数组成的表达式的意义。

C++提供的运算符有以下几种：算术运算符、关系运算符、逻辑运算符、位运算符、条件运算符、赋值运算符、逗号运算符、sizeof 运算符及其它运算符(这是按功能分的)。不同的运算符，需要指定的操作数的个数并不相同。根据运算符需要的操作数的个数，可将其分为三种：单目运算符（一个操作数）、双目运算符（两个个操作数）和三目运算符（三个操作数）。下面，我们介绍几种基本的 C++运算符。

3.1.1 算术运算符

C++提供 5 种基本的算术运算符，参见表 3-1。

表 3-1 5 种基本的算术运算符

运算符	名字	实例
+	加	12 + 4.9 // 得出 16.9
-	减	3.98 - 4 // 得出 -0.02
*	乘	2 * 3.4 // 得出 6.8
/	除	9 / 2.0 // 得出 4.5
%	取余	13 % 3 // 得出 1

这 5 个算术运算符都是双目运算符。除%运算符外，其它算术运算符的两个操作数可以是整型（short int、int、long int、unsigned short int、unsigned int 或 unsigned long int 型）和实型（float 或 double 型）的混合类型，运算结果的数据类型是：两个操作数的数据类型中，具有较高级别的数据类型（数据类型的级别，请参见 3.3 节）。例如，一个 int 型操作数和一个 float 型操作数的运算结果是 float 型；一个 float 型操作数和一个 double 型操作数的运算结果是 double 型。

当除运算符（/）的两个操作数均为整数时，所得的结果总是被圆整化为整数。例如：

9 / 2 // 得出 4, 不是 4.5!

-9 / 2 // 得出 -5, 不是-4!

在程序设计中，我们有时可以利用整数除法获得所需要的结果，但也容易产生错误。如果两个操作数是整数，要获得实数除法，我们应当将两个或任一个整型操作数强制转换为实型数，例如：

```
int cost = 100;
int volume = 80;
double unitPrice = cost / (double) volume; // 得出 1.25
```

执行除法运算时，如果除数为零,程序运行时，会产生一个被零除的错误。

取余运算符（%）的两个操作数都必须是整型数，运算结果是整除后的余数。例如 13%3 的结果是 1。

算术运算的结果可能太大，而不能存储在一个指定的变量中，这种情形称之为溢出。例如：

```
unsigned char k = 10 * 92; // 溢出: 920 > 255
```

3.1.2 自增、自减运算符

＋＋是自增运算符，——自减运算符，这两个运算符都是单目运算符，且功能相近，都是将数值变量的值加 1 或减 1，用户只能将这类操作符应用于变量而不能应用于常量。要替代下列代码

```
valuel=valuel+1;
```

可使用

```
++valuel;
```

或

```
valuel++;
```

这里前一种方式称为前缀方式，后一种称为后缀方式，其目的都是使 valuel 加 1。二者的区别是：前缀式先将操作数增 1（或减 1），然后取操作数的新值参与表达式的运算。后缀是先将操作数增 1（或减 1）之前的值参与表达式的运算，到表达式的值被引用之后再做加 1（或减 1）运算。

参见表 3-3。

我们可以看出, 自加和自减运算符可在变量名前，也可在变量名后，即都可以用于前缀和后缀的形式，但含义并不相同。对于前缀的形式，变量先作自加或自减运算，然后将运算结果用于表达式中；而对于后缀的形式， 变量的值先在表达式中参与运算，然后再作自加或自减运算。

可以作自加或自减的变量类型也可以是实型，但是，实型变量自加或自减运算的用处不大，我们很少使用。通常，我们对整型变量作自加或自减运算。以后，我们还会看到，指针变量也可以作自加或自减运算。

3.1.2 自增、自减运算符

＋＋是自增运算符，——自减运算符，这两个运算符都是单目运算符，且功能相近，都是将数值变量的值加 1 或减 1，用户只能将这类操作符应用于变量而不能应用于常量。要替代下列代码

```
valuel=valuel+1;
```

可使用

```
++valuel;
```

或

```
valuel++;
```

这里前一种方式称为前缀方式，后一种称为后缀方式，其目的都是使 valuel 加 1。二者的区别是：前缀式先将操作数增 1（或减 1），然后取操作数的新值参与表达式的运算。后缀是先将操作数增 1（或减 1）之前的值参与表达式的运算，到表达式的值被引用之后再做加 1（或减 1）运算。

参见表 3-3。

我们可以看出, 自加和自减运算符可在变量名前，也可在变量名后，即都可以用于前缀和后缀的形式，但含义并不相同。对于前缀的形式，变量先作自加或自减运算，然后将运算结果用于表达式中；

而对于后缀的形式， 变量的值先在表达式中参与运算，然后再作自加或自减运算。

可以作自加或自减的变量类型也可以是实型，但是，实型变量自加或自减运算的用处不大，我们很少使用。通常，我们对整型变量作自加或自减运算。以后，我们还会看到，指针变量也可以作自加或自减运算。

3.1.3 关系运算符

C++提供 6 种关系运算符，用于数值之间的比较，表达式的值或为 1（表示 true），或为 0（表示 false），参见表 3-3。

应该注意：<= 和 >= 运算符不能写成=< 和 =>，=< 和 =>是无效的运算符。关系运算符的操作数应当是一个数值，字符是有效的操作数，因为它们是用数值来表示的。例如（假定采用 ASCII 编码）：

'A' < 'F' // 得出 1 (它等价于 65 < 70)

字符串不应当用关系运算符比较，因为被比较的不是字符串的内容本身，而是字符串的地址。例如：

"HELLO" < "BYE"

引起"HELLO"的地址与"BYE"的地址进行比较。由于字符串的地址是由编译器决定的，所以，表达式的结果或为 0，或为 1，并不确定。我们以后会看到：可以用 C++的库函数 strcmp，比较两个字符串。

表 3-4 关系运算符

运算符	名字	实例
= =	等于	5 = = 5 // 得出 1
! =	不等于	5 != 5 // 得出 0
<	小于	5 < 5.5 // 得出 1
< =	小于或等于	5 <= 5 // 得出 1
>	大于	5 > 5.5 // 得出 0
> =	大于或等于	6.3 >= 5 // 得出 1

由关系运算符组成的关系表达式的值是逻辑型的，即 bool 型。在 C++中常常将逻辑真用非 0 表示（一般为 1），逻辑假用 0 表示。

其中:

'=='用于判断其前后数值或表达式的结果 a 和 b 是否相等。如 a 为 2001，b 为 667*3，则表示 a 等于 b 是真的。

'!='用于判断其前后数值或表达式的结果 a 和 b 是否不相等。如 a 为 2002，b 为 667*3，则表示 a 不等于 b 是真的。

'<'用于判断其前后数值或表达式的结果 a 是否小于 b。如 a 为 2002，b 为 667*3，则表示 a 小于 b 是假的。

'<='用于判断其前后数值或表达式的结果 a 是否小于或等于 b。 如 a 为 2002，b 为 667*3，则表示 a 小于或等于 b 是假的。

'>'用于判断其前后数值或表达式的结果 a 是否大于 b。如 a 为 2002，b 为 667*3，则表示 a 大于 b 是真的。

'>='用于判断其前后数值或表达式的结果 a 是否大于或等于 b。 如 a 为 2002，b 为 667*3，则表示 a 大于或等于 b 是真的。

关系运算符组成的表达式，一般是和逻辑运算符组合用在条件表达式中的。条件表达式用于程序的分支处理。

注意：

1．关系运算符两边的数值结果必须是类型相同的，否则会容易出现关系判断的错误。如：

char a = '0xa2';

a > '0xa1' 的结果是 a 小于'0xa1'。因为 a 是符号整数，而'0xa1'是无符号整数。给 a 赋值后其实是小于 0 的。

2．字符串不应当用关系运算符比较，因为被比较的不是字符串的内容本身，而是字符串的地址。例如：

"Welcome" < "Beijing"

引起"Welcome"的地址与"Beijing"的地址进行比较。由于字符串的地址是由编译器决定的，所以，表达式的结果或为假，或为真，并不确定。我们以后会看到：可以用 C++的库函数 strcmp，比较两个字符串。

3.1.4 逻辑运算符

C++提供了 3 种逻辑运算符，参见表 3-5。如同关系运算符，用逻辑运算符组成的表达式的值或为 1（表示 true），或为 0（表示 false）。

逻辑非（！）是单目运算符，它将操作数的逻辑值取反。即：如果操作数是非零，它使表达式的值为 0；如果操作数是 0，它使表达式的值为 1。

逻辑与（&&）与逻辑或（||）的含义，参见表 3-6。

下面是一些有效的逻辑表达式：

```
!20 // 得出 0
10 && 5 // 得出 1
10 || 5.5 // 得出 1
10 && 0 // 得出 0
```

表 3-5 逻辑运算符

运算符	名字	实例
!	逻辑非	!(5 = = 5) // 得出 0
&&	逻辑与	5 < 6 && 6 < 6 // 得出 0
||	逻辑或	5 < 6 || 6 < 5 // 得出 1

表 3-6 逻辑与（&&）和逻辑或（||）运算

运算符	操作数 1	操作数 2	表达式的值
逻辑与（&&）	true	true	true
	false	true	false

	true	false	false
	false	false	false
逻辑或（‖）	true	true	true
	false	true	true
	true	false	true
	false	false	false

表 3-5 逻辑运算符

运算符	名字	实例
!	逻辑非	!(5 = = 5) // 得出 0
&&	逻辑与	5 < 6 && 6 < 6 // 得出 0
‖	逻辑或	5 < 6 ‖ 6 < 5 // 得出 1

表 3-6 逻辑与（&&）和逻辑或（‖）运算

运算符	操作数 1	操作数 2	表达式的值
逻辑与（&&）	true	true	true
	false	true	false
	true	false	false
	false	false	false
逻辑或（‖）	true	true	true
	false	true	true
	true	false	true
	false	false	false

3.1.5 位运算符

C++提供 6 种位运算符，可以进行二进制位的运算，参见表 3-7。

位运算符要求操作数是整型数，并按二进制位的顺序来处理它们。取反运算符是单目运算符，其它位运算符是双目运算符。取反运算符（~）将操作数的二进制位逐位取反。逐位与运算符（&）比较两个操作数对应的二进制位，当两个二进制位均为 1 时，该位的结果取 1，否则取 0。逐位或运算符（|）比较两个操作数对应的二进制位，当两个二进制位均为 0 时，该位的结果取 0，否则取 1。逐位异或运算符（^）比较两个操作数对应的二进制位，当两个二进制位均为 1 或均为 0 时，该位的结果取 0，否则取 1。

逐位左移运算符（<<）和逐位右移运算符（>>）均有一个正整数 n 作为右操作数，将左操作数的每一个二进制位左移或右移 n 位，空缺的位设置为 0 或 1。对于无符号整数或有符号整数，如果符号位为 0（即为正数），空缺位设置为 0；如果符号位为 1（即为负数），空缺位是设置为 0 还是设置为 1，要取决于所用的计算机系统。

表 3-7 位运算符

运算符	名字	实例
~	取反	~'\011' // 得出 '\366'
&	逐位与	'\011' & '\027' // 得出'\001'
|	逐位或	'\011' | '\027' // 得出'\037'
^	逐位异或	'\011' ^ '\027' // 得出'\036'
<<	逐位左移	'\011' << 2 // 得出'\044'
>>	逐位右移	'\011' >> 2 // 得出'\002'

位操作运算符是用来进行二进制位运算的运算符。它分为两类：逻辑位运算符和移位运算符。

1、逻辑位运算符

（1）单目逻辑位运算符：~（按位求反）

作用是将各个二进制位由 1 变 0，由 0 变 1。

（2）双目逻辑运算符：&（按位与），|（按位或），^（按位异或）

其中优先级&高于^，而^高于|。

& ：两个都为 1 时结果为 1。

| ：两个其中有一个为 1 则结果为 1。

^ ：两个不同则结果为 1，否则为 0。

2、移位运算符

移位运算符有两个，<<（左移）、>>（右移）。两个 都是双目运算符。

例. 指出下面表达式的功能。

(p&0377)|(k&0xff)<<8

答：将整数 p 的低字节作为作为结果的低字节，k 的低字节作为结果的高字节拼成一个新的整数。

针对两个操作数 011 和 027，表 3-8 给出各个位运算符的计算实例，运算结果在表 3-7 中。为了避免符号位的麻烦（它与机器有关），我们采用无符号的整数。

unsigned char x = '\011';

unsigned char y = '\027'

表 3-8 位运算符如何运算

实例	10 进制值	二进制值							
x	011	0	0	0	0	1	0	0	1
y	027	0	0	0	1	0	1	1	1
~x	366	1	1	1	1	0	1	1	0
x & y	001	0	0	0	0	0	0	0	1
x | y	037	0	0	0	1	1	1	1	1
x ^ y	036	0	0	0	1	1	1	1	0
x << 2	044	0	0	1	0	0	1	0	0
x >> 2	002	0	0	0	0	0	0	1	0

位运算符要求操作数是整型数，并按二进制位的顺序来处理它们。位运算符除取反运算符'~'为

单目运算符外，其余的都是双目运算符，这可以从上面的表格中的表达式例子中看出来。具体说来：

'&'为逐位与运算符，用于把两个操作数对应的二进制位进行逻辑与操作，逻辑与的操作上一节已经作了介绍，这里不再赘述。所以 0x011&0x001 其实就是二进制数 0000000000010001B 和二进制数 0000000000000001B 每一位都进行逻辑与运算，其结果必然是 0000000000000001B 即 0x001。

'|'为逐位或运算符，用于把两个操作数对应的二进制位进行逻辑或操作。所以，根据逻辑或的运算规则，表达式 0x011|0x001 的结果必然是 0x011。有兴趣的同学可以自己把相应的二进制位进行逐个的逻辑或运算，看看结果是不是 0x011.。

'^'为逐位异或运算符，用于把两个操作数对应的二进制位进行异或操作。因此，0x011^0x001 的结果是 0000000000010000B，即 0x010。

'<<'为逐位左移运算符，用于将左操作数的每一个二进制位左移右操作数位，空缺的位设置为 0 或 1。对于无符号整数或有符号整数，但符号位为 0（即为正数），空缺位设置为 0；如果符号位为 1（即为负数），空缺位是设置为 0，还是设置为 1，要取决于所用的计算机系统。因此，0x011<<2 结果是 0x044。

'>>'为逐位右移运算符，将左操作数的每一个二进制位右移右操作数位，空缺的位设置为 0 或 1，和逐位左移运算符一致。因此，0x011>>2 的结果是 0x004

具体的运算的进行还可以参考表 3-7 位运算符如何运算。

3.1.6 赋值运算符

=是赋值运算符，它的作用是将一个表达式的值赋给一个左值。一个表达式或者是一个左值，或者是一个右值。所谓左值是指一个能用于赋值运算左边的表达式。左值必须能够被修改，不能是常量。我们现在是用变量作左值，以后还可以看到，指针和引用也可以作左值。例如：

```
int a, b, c;
a=3;
b=4;
c = ( a + b )*（2*a - b） // 得出 14
```

赋值运算符可与算术运算符和位运算符组合，产生许多变种，参见表 3-9（实例中假定 n 是一个整型变量）。

赋值运算本身也是一个表达式，即赋值表达式，该表达式的值是存储在赋值运算符左边变量中的值。一个赋值表达式能作为另一个赋值表达式的右操作数，多个赋值运算符能够连接在一个表达式中。例如：

```
int m, n, p;
m = n = p = 100; // 即: n = (m = (p = 100));
m = (n = p = 100) + 2; // 即: m = (n = (p = 100)) + 2;
m += n = p = 10; // 即: m = m + (n = p = 10);
```

表 3-9 赋值运算符

运算符	实例	等价于
=	n = 25	
+ =	n += 25	n = n + 25
- =	n -= 25	n = n - 25
* =	n *= 25	n = n * 25
/ =	n /= 25	n = n / 25
% =	n %= 25	n = n % 25

& =	n &= 0xF2F2	n = n & 0xF2F2
| =	n |= 0xF2F2	n = n | 0xF2F2
^ =	n ^= 0xF2F2	n = n ^ 0xF2F2
<< =	n <<= 4	n = n << 4
>> =	n >>= 4	n = n >> 4

赋值运算符分为两种：一种是简单的最常用的=运算符；二是复合的赋值运算符，又称为带有运算的赋值运算符，也叫赋值缩写。

例如：i=i+j；可表示为 i+=j；这里+=是复合赋值运算符。

同样的共有 10 种这样的运算符，它们是：

+= 加赋值

-= 减赋值

*= 乘赋值

/= 除赋值

%= 求余赋值

&= 按位与赋值

|= 按位或赋值

^= 按位异或赋值

<<= 左移位赋值

>>= 右移位赋值

<> 当右操作数又是一个赋值表达式时，形成多重赋值表达式。例如：

i=j=0; //结果 i、j 的值都为 0

注意：多重赋值表达式不能出现在变量说明中。例如：

int i=j=0;

是非法的。

例：有变量说明

int a=2,b;

指出下面表达式运算后 a 和 b 的结果。

b+=b=++a;

答：a 为 3，b 为 6。

<> 有时将赋值运算与比较运算结合在一起形成嵌入赋值。例如：

while ((ch=getchar())!='\n');

这条语句的含义是：等待用户按下回车键后程序向下执行。

需要注意的是： x *= y+8 等价于 x=x*(y+8)，不等价于 x = x*y+8。

同样：z &= y-x 等价于 z = z & (y-x)，而不等价于 z = z &y -x。

3.1.7 求字节数运算符

C++提供了一个有用的运算符 sizeof，它是一个单目运算符，用于计算表达式或数据类型的字节数，其运算结果与不同的编译器和机器相关。当编写用于进行文件输入/输出操作或给动态列表分配内存的程序时，用户将发现，如能知道程序给这些特定变量所分配内存的大小将会很方便。 例如 3-1：

当我们进行算术运算时，如果运算结果超出变量所能表达的数据范围时，就会发生溢出。而利用 sizeof

运算符计算变量所占的字节数，也就是说，可以算出变量的数据范围，从而可以避免可能出现的错误。

例 3-1

```
#include <iostream.h>
int main (void)
{
   cout << "char size = " << sizeof(char) << " bytes\n";
   cout << "char* size = " << sizeof(char*) << " bytes\n";
   cout << "short size = " << sizeof(short) << " bytes\n";
   cout << "int size = " << sizeof(int) << " bytes\n";
   cout << "long size = " << sizeof(long) << " bytes\n";
   cout << "float size = " << sizeof(float) << " bytes\n";
   cout << "double size = " << sizeof(double) << " bytes\n";
   cout << "1.55 size = " << sizeof(1.55) << " bytes\n";
   cout << "1.55L size = " << sizeof(1.55L) << " bytes\n";
   cout << "HELLO size = " << sizeof("HELLO") << " bytes\n";
}
```

我们运行这个程序，产生下面的结果（在作者的 PC 机上）：

```
char size = 1 bytes
char* size = 2 bytes
short size = 2 bytes
int size = 2 bytes
long size = 4 bytes
float size = 4 bytes
double size = 8 bytes
1.55 size = 8 bytes
1.55L size = 10 bytes
HELLO size = 6 bytes
```

表 3-10

类型	运算	运算符
求字节数运算符	求字节大小	sizeof

下表给出了常用数据类型的字节数。具体的例子可参考书中给出的例子。当我们进行算术运算时，如果运算结果超出变量所能表达的数据范围时，就会发生溢出。如果我们能够利用 sizeof 运算符计算变量所占的字节数，就可算出变量的数据范围，从而可以避免可能出现的错误。

下表给出了常用数据类型的字节数。具体的例子可参考书中给出的例子。

表 3-11

数据类型	占用字节数

Char	1
Char *	4
Short	2
Int	4（VC 6.0）2 (VC 1.5x)
Long	4
Float	4
Double	8

3.1.8 条件运算符

条件运算符是 C++中唯一的三目运算符，也称为三元运算符，它有三个操作数：

操作数 1？操作数 2：操作数 3

条件运算符又可以称为'?'号运算符。操作数 1 一般是条件表达式，若表达式成立，即为真，则整个表达式的值为操作数 2，否则为操作数 3。表中的例子：若 a>=b， 则例子的结果为 a ，否则就为 b。

例如、cout <<('A'<=ch && ch<='Z')? ('a'+ch-'A')：ch

结果是输出一个小写字母。

如果第一个操作数非零，表达式的值是操作数 2，否则表达式的值取操作数 3。例如：

```
int m = 1, n = 2;
int min = (m < n ? m : n); // min  取  1
```

由于条件运算本身是一个表达式，即条件表达式，它可以作为另一个条件表达式的操作数。也就是说，条件表达式是可以嵌套的。例如：

```
int m = 1, n = 2, p =3;
int min = (m < n ? (m < p ? m : p)
                 : (n < p ? n : p));
```

再看看其它的例子：

```
int a=10,b=20;
int min = (a>=b? a: b);
```

则 min 取值为 20。

由条件运算符组成的条件表达式，可以作为另一个条件表达式的操作数，即条件表达式是可以嵌套的，如：

```
int a=10,b=20,c=30;
int min=(a>=b ?) (b<=c ? b: c): (a<=c ? a : c) //  结果为 10。
```

3.1.9 逗号运算符

多个表达式可以用逗号组合成一个表达式，即逗号表达式。

逗号运算符带两个操作数，返回值是右操作数。

逗号表达式的一般形式是："表达式 1，表达式 2，……，表达式 n"，它的值是取表达式 n 的值。

逗号运算符的用途仅在于解决只能出现一个表达式的地方却要出现多个表达式的问题。

例如：

d1，d2，d3，d4

这里 d1、d2、d3、d4 都是一个表达式。整个表达式的值由最后一个表达式的值决定。计算顺序是从左至右依次计算各个表达式的值，最后计算的表达式的值和类型便是整个表达式的值和类型。

例 3-2

```
int m, n, min;
int mCount = 0, nCount = 0;
// ...
min = (m < n ? mCount++, m : nCount++, n);
```

当 m 小于 n 时，计算 mCount++，m 存储在 min 中。否则，计算 nCount++，n 存储在 min 中。

表 3-12

类型	运算	运算符	例子
逗号运算符	逗号运算符	,	a=10,11,12

表中例子的结果是 a 为 12。

除了上面介绍的一些常用的基本运算符之外，C++还有一些比较特殊的运算符如下：

表 3-13

类型	运算符	例子
全局变量或全局函数	: :（全局）	: : GetSystemDirectory
类中的域变量或函数	: :（类域）	CWnd::FromHandle
括号及函数调用	( )	(a+b)*(a-b)
指针指向的结构或类种的域变量	- >	(CWnd *wnd)-> FromHandle
结构或类中的域变量	.	(CWnd wnd). FromHandle
数组下标运算符	[]	nYearsMonthsDays[10][12][366]
内存分配运算符	new	new CWnd
内存释放运算符	delete	delete (CWnd *wnd)

当不同的运算符混合运算时，运算顺序是根据运算符的优先级而定的，优先级高的运算符先运算，优先级低的运算符后运算。在一个表达式中，如果各运算符有相同的优先级，运算顺序是从左向右，还是从右向左，是由运算符的结合性确定的。所谓结合性是指运算符可以和左边的表达式结合，也可以与右边的表达式结合。C++运算符的优先级和结合性参见表 3-14。

在表 3-14 中，"单"表示是单目运算符，"双" 表示是双目运算符，"三" 表示是三目运算符。表中有一些运算符，我们还没有介绍，以后我们会在有关的内容中学习。

表 3-14 C++运算符的优先级和结合性

优先级	运算符	种类	结合性
最高	: :（全局）	单	从右到左
最高	: :（类域）	双	从左到右

	( )(括号及函数调用)　　- > . .[ ]	双	从左到右
	+ ++ ! * new sizeof - -- ~ & delete (类型)	单	从右到左
	- > *　　. *	双	从左到右
	+ / %	双	从左到右
	+ —	双	从左到右
	<< >>	双	从左到右
	< < = > > =	双	从左到右
	= =　　! =	双	从左到右
	&	双	从左到右
	^	双	从左到右
	|	双	从左到右
	& &	双	从左到右
	||	双	从左到右
	? :	三	从左到右
	=	双	从右到左
最低	,	双	从左到右

每种运算符都有一个优先级，优先级是用来标志运算符在表达式中的运算顺序的。优先级高的先做运算，优先级低的后做运算，优先级相同的右结合性决定计算顺序。

大多数运算符都是按从左到右计算，只有三类运算符的结合性是从右到左。他们是：单目、三目和赋值。这一点一定要记住。

3.3 类型转换

C++中，一种数据类型能够被转化为另一种数据类型。例如 3-3：

从上面实例我们可以看到：类型标识符能被用于类型运算符。类型运算符是单目运算符，并位于其操作数的左边的圆括号里，称之显式类型转换，或强制类型转换。显式类型转换的一般形式为：

（类型名）（表达式）

需要注意的是： (int)(x+y)是将（x+y）转换为 int 型，而（int）x+y 是将 x 转换为 int 型后再与 y 相加。显式类型转换只是得到一个所需类型的中间变量，原来变量的类型并不发生变化。

例 3-3

```
(int) 3.14 // 3.14  转换成整型数 3
(long) 3.14 // 3.14 转换成长整型数 3L
(double) 2 // 2 转换成双精度数 2.0
(char) 122 // 122 转换成 Unicode 码为 122 的字符
(unsigned short) 3.14 // 3.14 转换成无符号整型数 3
```

如果类型标识符仅为一个单词，显式类型转换的形式也可以写成：类型表示符（表达式）。例如：

```
int(3.14) //  等价于(int) 3.14
```

在一般情况下，各种数据都按默认类型参与运算， 当两个不同的数据类型（整型、字符型、实型）进行运算时，数据将进行自动类型转换，转换后再参与运算。

例：指出下面每个语句的执行结果。

例 3-4

```
char ch='c';
int a，b=13；
float x=2.0；
double y；
a=ch+5；  //a=104，a 先转化为 int 型，再参与运算。
x=b/2/x；  //x=3.0，先作整除运算，然后再转换成 double 与 x 运算。
x=b/x/2; //x=3.25，b 先转换成 float 型与 x 做除法，同时 2 也转化成 float 型，然后做除法运算。
y=x/b；  //x=0.153846153846154，x 和 b 分别转化成 double 然后做除法运算。
```

又例如：

例 3-5

```
double d = 1; // d 的值 1.0
int i = 10.5; // i 的值 10
i = i + d; // 等价于: i = int(double(i) + d)
```

上例中，表达式 i = i + d 的变量 i 与 d 的类型不同，使得 i 首先被转换为 double 型（提升），再与 d 相加，其结果为一 double 型数。由于该 double 型数与赋值语句左边 i 的类型不匹配，所以在它被赋给 i 之前，再被转换为 int 型（下降）。

上面的自动类型转换也称为隐式转换。

操作数的类型是有级别的，上表中从下到上，"条件"栏中每一行的数据类型的级别由低到高，且该数据类型是表示两个操作数中，具有较高级别的数据类型。当两个操作数进行运算时，需要将另一个操作数也提升为相应的数据类型。

类型转换是程序设计过程中经常碰到的，需要把一种类型的数据转换为另一种类型表示，然后才能一起在表达式中参加运算。

下面先看一看下表中的 C++中的类型转换的例子

表 3-16

例子	转换前类型	转换后类型	结果
(int) 3.5	双精度浮点型 double	整型 int	3
(long) -3.5	双精度浮点型 double	长整型 long	-4
(double)2	整型 int	双精度浮点型 double	2.0
(char)122	整型 int	字符型 char	'0x7A'
(float) 2	整型 int	单精度浮点型 float	2.0F
(unsigned short)3.5	双精度浮点型 double	无符号短整型 unsigned short	3
int （3.5）	双精度浮点型 double	整型 int	3

而同学们更需要注意的是当不同类型的数值在一个表达式中混合运算时，C++进行的隐式类型转换。如：

例 3-6

```
double d = 2; // d 的值 2.0
int i = 3.5; // i 的值 3
i = i - d; // 等价于: i = int(double(i) - d) 值为 1
```

其中，表达式 i = i - d 的变量 i 与 d 的类型不同，使得 i 首先被转换为 double 型（提升），再与 d 相加，其结果为一 double 型数。由于该 double 型数与赋值语句左边 i 的类型不匹配，所以在它被赋给 i 之前，再被转换为 int 型（下降）。

在显式类型转换中，应该特别注意从较高级别的类型转换为较低级别的类型时，容易引起数据的丢失。如：

(unsigned char) 900

的结果是 132。

有时，一个数据经过几次类型变换以后再变换回原来的数据类型就可能不是原来的数值了。如；

例 3-7

```
short ishort = 1000;
unsigned char uichar;
uichar = ishort;
ishort = uichar;
```

最后的结果是 ishort=232，相当于 1000%256。在赋值表达式"uichar = ishort;"的执行中，把 ishort 的值 1000 进行类型转换（从 short 转换为 unsigned char），得到值 232。最后表达式"ishort = uichar;"在把 232 转换为 short 类型，赋值给变量 ishort。

3.4.1 表达式的种类

表达式是由运算符和操作数组成的式子。运算符可以是前面讲过的那些。操作数包含了常量、变量、函数和其他一些命名的标识符。最见大的表达式是常量和变量。

C++中由于运算符很丰富，因此表达式的种类也很多。常见的表达式有如下六种：

已知 int a；

·算术表达式。例如，a+5.2/3.0-9%5

·关系表达式。例如，'m'>='x'

·逻辑表达式。例如，！ a && 8 || 7

·条件表达式。例如，a>4 ? ++a:--a

·赋值表达式。例如，a=7

·逗号表达式。例如，a+5,a=7,a+=4

注意：

1、在表达式中，连续出现两个运算符时，最好用空格符分隔。如：

a+++b; （注意：在 visual c++中这种写法是错误的，编译将不能通过）

系统将默认为是 a++ +b，因系统将按尽量取大的原则来分割多个运算符。如果想执行 a 加++b，则应写成

a+ ++b；

2、在写表达式中，有时记不清楚运算符的优先级时，可使用括号来确定运算符组合。

3.4.2 表达式的值和类型

任何表达式经过计算都应有一个确定的值和类型。在计算一个表达式的值时，应注意下述两点：

1、先确定运算符的功能。在 C++中，有些运算符相同但功能不同，因此要先确定其功能。例如运算符：*， &， -。它们有时是单目运算符，有时是双目运算符，在计算前要分清楚。

2、确定计算顺序。

一个表达式的计算顺序是由运算符的优先级和结合性来决定的。优先级高的先做，优先级地的后做。在优先级相同的情况下，右结合性决定。多数情况下，由左至右。少数情况下，由右至左。

表达式的类型由运算符的种类和操作符的类型决定。

表达式的求值方法和确定类型的方法如下：

1、算术表达式

算术表达式是由算术运算符和位操作运算符组成的表达式，其表达式的值是一个数值。表达式的类型具体地由运算符合操作数决定。参见例 3-8

例 3-8 分析下面程序的输出结果

```
#include <iostream.h>
void main( )
{
   int a,b, m=3,n=4;
   a=7*2+-3%5-4/3;//-3%5=-3，4/3=1
   b=m++ - --n;
   cout<<a<<"\t"<<b<<"\t"<<m<<"\t"<<n<<endl;
}
```

执行结果如下：

10 0 4 3

2、关系表达式

由关系运算符组成的表达式为关系表达式。关系表达式的运算结果为逻辑型，常用在条件语句和循环语句中。

参见例 3-9

例 3-9 分析下面程序的执行结果

```
#include <iostream.h>
void main( )
{
   char x='m',y='n';
   int n;
```

```
    n=x<y;
    cout <<n<<endl;
    n=x==y-1;
    cout<<n<<endl;
    n=('y'!='Y')+(5<3)+(y-x==1);
    cout <<n<<endl;
}
```

程序执行结果为：

```
1
1
2
```

通过上面的程序可以看出：关系运算的结果为'真'时值等于 1，结果为假时值等于 0。

3、逻辑表达式

由逻辑运算符组成的表达式称为逻辑表达式。逻辑表达式的值为逻辑型，结果为 1 和 0。

在由&& 和||运算符组成的逻辑表达式中，C++规定：只对能够确定整个表达式值所需要的最少数目的子表达式进行计算。也就是说，当计算出一个子表达式的之后便可确定整个逻辑表达式的值时，后面的子表达式就不需要再计算了，整个表达式的值就是该子表达式的值。这种表达式也称为短路表达式。参看例 3-10

例 3-10

int a=3, b=0; 问下面表达式运算后 a 和 b 的值是多少？

(1) ! a && a+b && a++

(2) !a||a++||b++

分析：第一题是一个由&&组成的逻辑表达式，从左至右计算三个子表达式，只要有一个为 0 就不再计算其他子表达式。当计算！a 的值为 0 时，便可确定整个表达式的值为 0，因此后面的子表达式就不再计算了。所以 a 的值为 3，b 的值为 0。

第二题：这是一个由||组成的逻辑表达式，从左至右计算三个子表达式，只要有一个结果为真则不再计算后面的子表达式。第一个子表达式为!a 结果为 0，再计算 a++结果为 4，所以就不再计算后面的子表达式。所以结果 a 为 4，b 为 0。

4、条件表达式

由三目运算符？：组成的表达式为条件表达式。例如 a>b? x=4:x=9; 条件表达式的值取决于？前面的表达式的值，该表达式的值为非 0 时，整个表达式的值为'：'前面的表达式的值，否则为'：'后面的表达式的值。

参见例 3-11

例 3-11 分析下面程序的执行结果

```
#include <iostream.h>
void main( )
{
   int a=3,b=4,c;
```

```
    c=a>b? ++a:++b;
    cout<<a<<","<<b<<","<<c<<endl;
    c=a-b?a-3?b:b-a:a;
    cout<<a<<","<<b<<","<<c<<endl;
}
```

该程序执行的结果为：

3 5 5

3 5 2

5、赋值表达式

由赋值运算符组成的表达式为赋值表达式。赋值运算符除了"="之外还有十个符合运算符，这是赋值和运算相结合的运算符。。

赋值运算符的结合性是由右至左，因此，C++程序中允许出现连赋值的情况。

参看例 3-12

例 3-12

例如下面的赋值是合法的。

```
int a,b,c,d;
a=b=c=d=5/2;
```

这里先计算 5/2 结果为 2，再赋值给 d，结果 d=5/2 表达式的值为 2，再将这个值赋给 c，以此类推，结果 a、b、c、d 的值均为 2。

再计算复合赋值运算符表达式中，首先计算右值表达式的值后再与左值运算。例如：

```
int a=3，b=4；
a*=b+1；
```

这里先计算 b+1 等于 5，再与 a 相乘赋值给 a，结果等于 15。

赋值还可以嵌入到比较运算表达式中，例如：

```
x=func( );
if (x==3)
{
   .....
}
```

可以将赋值放在 if 的条件表达式中，如：

```
if ((x=func( ))==3 )
{...
}
```

这里，比较运算符==的优先级高于赋值运算符，所以应将赋值运算加上括号。

6、逗号表达式

逗号表达式是用逗号将若干个表达式连起来组成的表达式。该表达式的值是组成逗号表达式的若干个表达式中的最后一个表达式的值，类型也是最后一个表达式的类型。

逗号表达式的计算顺序是自左至右。

参见右例 3-13

例 3-13 分析下面程序的执行结果

```
#include <iostream.h>
void main( )
{
   int a,b,c;
   a=1,b=2,c=a+b+3;

cout<<a<<','<<b<<','<<c<<endl;
   c=(a++,a+=b,a-b);

cout<<a<<','<<b<<','<<c<<endl;
}
```

输出结果如下：

1，2，6

4，2，2

【本章小结】

介绍了 C++基本的各种运算符构成(算术运算符、关系运算符、逻辑运算符、位运算符、条件运算符、赋值运算符、逗号运算符几其它运算符)以及它们的优先级和结合性。同时介绍了由运算符组成的各种表达式、表达式中操作数类型的显式及隐式转换。

自增、自减运算符，前缀式先将操作数增 1（或减 1），然后取操作数的新值参与表达式的运算。后缀是先将操作数增 1（或减 1）之前的值参与表达式的运算，到表达式的值被引用之后再做加 1（或减 1）运算。

关系运算符两边的数值结果必须是类型相同的。

在实现优先级与实际需要不相符时，需要使用括号来改变。

参加运算的两个操作数类型不同时，C++将自动作隐式类型转换，但有时要作强制类形转换。

表达式和语句的一个重要区别是：表达式具有值，而语句是没有值的并且语句末尾要加分号。

【学习目标】

熟练掌握 C++各种流程控制语句的用法，包括分支控制语句：if-else 语句，多分支控制语句：switch 语句，三种循环语句：for 循环、while 循环和 do-while 循环语句，会用流程控制语句构造程序。

【重点和难点】

各种流程控制语句的的嵌套使用，包括 if-else 语句的嵌套，循环语句的嵌套，以及分支语句与循环语句的嵌套等。循环语句和 break， continue 语句的组合使用。

【学习方法指导】

很少程序是顺序执行的，要改变程序的执行流程，就要用流程控制语句。一定要熟练掌握各种流程控制语句的应用。可以用各种流程控制语句编写一些小程序，并运行，仔细体会它们的用法与区别。

【知识点】

if 语句；if-else 语句；switch 语句；while 语句；do-while 语句；for 语句；break 语句；continue 语句

语句是构造程序最基本的单位，程序运行的过程就是执行程序语句的过程。程序语句执行的次序称之为流程控制（或控制流程）。我们前面介绍的实例程序中，程序语句都是顺序执行的。我们是否能够改变程序顺序执行的流程呢？例如，当满足某个条件时，执行某种运算或重复执行某种运算。答案是肯定的，流程控制语句就是用于修改程序的执行路径。

本章我们介绍的 C++流程控制语句包括：

◇ if 语句，也称为分支语句。

◇ switch 语句，也称之为多分支语句。

◇ 循环语句。

◇ break 和 continue 语句，这两个语句与 switch 语句或循环语句配合使用。

◇ goto 语句。

现实生活中的流程是多种多样的，如汽车在道路上行驶，既要顺序地应道路前进，碰到交叉路口时，驾驶员就需要判断是转弯还是直走，在环路上是继续前进，还是需要从一个出口出去等等。又比如，在生产线上的零件的流动过程，应该顺序地从一个工序流向下一个工序，但当检测不合格时，就需要从这道工序中退出，或继续在这道工序中再加工直到检测通过为止。因此，现实生活中的流程可以概括为下面几个方面：

1. 流程分支，一个流程到达一定点时，需要分为两个或多个分支继续进行。
2. 流程循环，或者可以说是重复不停地进行同一工作。

为此，C++流程控制语句相应地设置了 if 语句、switch 语句、循环语句、break 和 continue 语句

4.1 if 语句

if 语句的一般形式如下：

```
if (表达式)
    语句;
```

"表达式"是所谓给定的条件，if 语句首先计算"表达式"的值，如果结果为非 0，则"语句"被执行，否则不执行该"语句"。

if 语句的一个变种是要求指定两个语句。当给定的条件满足时，执行一个语句；当条件不满足时，执行另一个语句。这也被称为 if-else 语句，其一般形式如下：

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

考虑到两个复合语句中的共同部分，我们可以将上面的语句简化成如例 4-2 的形式：

例 4-2

```
if (balance > 0)
interest = balance * creditRate;
```

```
else
interest = balance * debitRate;
balance += interest;
```

我们还可以使用条件表达式将上述语句进一步简化为：

```
interest = balance * (balance > 0 ? creditRate : debitRate);
balance += interest;
```

或:

```
balance += balance * (balance > 0 ? creditRate : debitRate);
```

同样考虑到两个复合语句中的共同部分，我们可以作如下简化：

```
把点从原始点表中取出;
    if （点不被多边形遮挡）
    {
      把点加入到显示点表中;
    }
    else
    {
      把点加入到非显示点表中;
    }
```

也可以使用上一章的条件运算符将上述语句进一步简化为：

```
把点从原始点表中取出;
点不被多边形遮挡? 把点加入到显示点表中: 把点加入到显示点表中;
```

if 语句能够被嵌套，也就是说，一个 if 语句能够出现在另一个 if 语句里。

例 4-3

```
if (callHour > 6) {
  if (callDuration <= 5)
    charge = callDuration * tarrif1;
  else
    charge = 5 * tarrif1 + (callDuration - 5) * tarrif2;
}
else
  charge = flatFee;
```

现实生活中的各种条件是很复杂的，在一定的条件下，又需要满足其它的条件才能确定相应的动作。为此，C++ 提供了 if 语句的嵌套功能，也即一个 if 语句能够出现在另一个 if 语句或 if-else 语句里。例如：

例 4-4

```
if (callHour > 6) {
  if (callDuration <= 5)
    charge = callDuration * tarrif1;
```

```
    else
      charge = 5 * tarrif1 + (callDuration - 5) * tarrif2;
  }
  else
    charge = flatFee;
```

另一例：

例 4-5

```
if ( cos( value ) < 0 ) // value  的 cos 函数小于零
{
  if ( value > 0 ) //  若 value  大于零
    value = value - PI;
  else
    value = value + PI;
}
```

嵌套的 if 语句的常用的形式也涉及到 else 部分由另一个 if-else 语句组成。

使用 if 语句的嵌套形式时，需要要注意的是：else 语句都是与离它最近的 if 语句配对。

例如，下面的 if 语句：

例 4-8

```
if ( x < 0 ) y = -1;
else if ( x == 0 ) y = 0;
      else y = 1;
```

我们可以把它修改为：

```
if ( x >= 0)
      if ( x > 0 ) y = 1;
      else y = 0;
else y = -1;
```

4.2 switch 语句

switch 语句的一般形式如下：

```
switch (表达式) {
case 常量表达式 1:
      语句;
...
case 常量表达式 n:
      语句;
default:
      语句;
}
```

switch 语句的执行过程是这样的：首先计算"表达式"的值，然后，其结果值依次与每一个常量表达式的值进行匹配（常量表达式的值的类型必须与"表达式"的值的类型相同）。如果匹配成功，则执行该常量表达式后的语句系列。当遇到 break 时，则立即结束 switch 语句的执行，否则，顺序执行到花括号中的最后一条语句。default 情形是可选的，如果没有常量表达式的值与"表达式"的值匹配，则执行 default 后的语句系列。需要注意的是："表达式"的值的类型必须是字符型或整型。

我们下面看一个分析双目算术运算的例子 4-9。双目算术运算符存在字符型变量 operator 中，两个操作数分别存在变量 operand1 和 operand2，switch 语句执行相应的运算，并将结果存在变量 result 中。

我们主要使用三种形式的循环语句：while 语句、do-while 语句和 for 语句，下面分别介绍。

4.3.1 while 语句

while 语句的一般形式如下：

```
while (表达式)
语句;
```

while 语句的执行过程如下：首先计算"表达式"（称之为循环条件）的值，如果其结果值非 0，则执行"语句"（称之为循环体）。这个过程重复进行，称之为循环，直至"表达式"的值为 0 时，才结束循环。

4.3.2 do-while 语句

do-while 语句类似于 while 语句，但是它先执行循环体，然后检查循环条件。do-while 语句的一般形式为：

```
do
    语句;
while (表达式);
```

如果"表达式"的值非 0，循环继续进行，否则，循环终止。

与 while 语句相比，我们使用 do-while 语句要少一些。但是，对于有些情况，循环体至少要执行一次，do-while 语句就很有用。例如，假定我们要重复读一个值，并打印它的平方，终止的条件是该值为 0。我们可以用 do-while 语句表示为：

```
do {
    cin >> n;
    cout << n * n << '\n';
    }
while (n != 0);
```

4.3.3 for 语句

for 语句的一般形式如下：

```
for (表达式 1; 表达式 2; 表达式 3)
语句;
```

for 语句执行过程如下：首先计算"表达式 1"（循环初值），且仅计算一次。每一次循环之前计算"表达式 2"（循环条件），如果其结果非 0，则执行"语句"（循环体），并计算"表达式 3"（循环增量）。否则，循环终止。for 循环与下面的 while 循环等价：

```
表达式 1;
```

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

下面我们再通过另一个例子来进一步学习 for 循环语句，for 循环语句经常用在有确定次数的循环语句中。假定我们要打印图 4-8 所示的乘法表：

图 4-8 乘法表

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

在前面我们已经看到：break 语句常和 switch 语句配合使用。break 语句和 continue 语句也与循环语句配合使用，并对循环语句的执行起着重要的作用。且 break 语句只能用在 switch 语句和循环语句中，continue 语句只能用在循环语句中。下面我们分别介绍 break 语句和 continue 语句。

根据程序的目的，有时需要程序在满足另一个特定条件时立即终止循环，程序继续执行循环体后面的语句，break 语句可实现此功能 continue 语句实现的功能是，根据程序的目的，有时需要程序在满足另一个特定条件时跳出本次循环。

4.4.1 break 语句

break 语句用在循环语句的循环体内的作用是终止当前的循环语句。例如 4-15：

例 4-15

```
// 无 break 语句
int sum = 0, number;
cin >> number;
while (number != 0) {
   sum += number;
```

```
    cin >>
number;
}
// 有 break
语句
int sum =
0, number;
while (1) {
    cin >>
number;
    if
(number
== 0)

break;
    sum +=
number;
}
```

这两段程序的产生的效果是一样的。需要注意的是：break 语句只是跳出当前的循环语句，对于嵌套的循环语句，break 语句的功能是从内层循环跳到外层循环。例如 4-16：

例 4-16

```
int i = 0, j,
sum = 0;
while (i <
5) {
    for ( j =
0; j < 5;
j++) {

sum + = i
```

```
+ j;
        if
( j == i)
break;
  }
  i++;
}
```

本例中 break 语句执行后，程序立即终止 for 循环语句，并转向 for 循环语句的下一个语句，即 while 循环体中的 i++语句，继续执行 while 循环语句。

终止当前的循环语句意思是在多重循环中，break 只能退出它所在的那重循环；只能退出内重循环到外重循环中，要退出外重循环还要在外重循环中使用 break。

例 4-15

```
// 无 break 语句
int sum = 0, number;
cin >> number;
while (number != 0) {
  sum += number;
```

```
    cin >>
number;
}
// 有 break
语句
int sum =
0, number;
while (1) {
    cin >>
number;
    if
(number
== 0)

break;
    sum +=
number;
}
```

这两段程序的产生的效果是一样的。需要注意的是：break 语句只是跳出当前的循环语句，对于嵌套的循环语句，break 语句的功能是从内层循环跳到外层循环。例如 4-16：

例 4-16

```
int i = 0, j,
sum = 0;
while (i <
5) {
    for ( j =
0; j < 5;
j++) {

sum + = i
```

```
        + j;
            if
        ( j == i)
        break;
          }
          i++;
        }
```

本例中 break 语句执行后，程序立即终止 for 循环语句，并转向 for 循环语句的下一个语句，即 while 循环体中的 i++语句，继续执行 while 循环语句。

终止当前的循环语句意思是在多重循环中，break 只能退出它所在的那重循环；只能退出内重循环到外重循环中，要退出外重循环还要在外重循环中使用 break。

4.4.2 continue 语句

continue 语句的功能与 break 语句不同，它是结束当前循环语句的当前循环，而执行下一次循环。在循环体中，continue 语句执行之后，其后的语句均不再执行。

在 while 和 do-while 循环语句中，下一次循环是从判断循环条件开始，在 for 循环语句中，下一次循环是是从先计算第三个表达式，再判断循环条件开始的。例如：重复读入一些整数，当该整数为负时忽略，否则，处理该整数，而该整数为 0 时，程序执行终止。这段代码可以表示为程序段 4-1：

4.4.3 goto 语句

goto 语句，它使程序执行分支转移到被称为"称号"（label）的目的地。例如 4-17，用 goto 语句来显示 1～100 的数字：

例 4-17

```
#include <iostream.h>
void main()
{
   int count=1;
   label:
   cout << count++;
   if(count <= 100)
      goto label;
}
```

使用 goto 语句时，标号的位置必须在当前函数内。也就是说，不能使用 goto 从 main 转移到另一个函数的标号上，或反过来。

注意：现代程序设计方法主张尽可能地限制 goto 语句的使用，可使用 if,if-else 和 while 这样地结构来代替它，增强代码的可读性。

编写一个程序，模拟具有加、减、乘、除四种功能的简单计算器，如左图 4-10 所示。

该程序具体要求如下：

1. 运行程序，显示当前值及命令提示 command>。
2. 通过键入字符'+'、 '-'、'*'或 '/'（分别表示加、减、乘或除运算符）响应程序的提示 command>，能够支持加、减、乘、除运算。
3. 当一个运算符输入后，程序便要求输入同当前值一起运算的第二个数。若给定了第二个数，程序便执行指定的运算，显示新的当前值。
4. 在程序提示 command>后可接受字符'C'作为清除当前值的命令，即将当前值设置为 0。
5. 在程序提示 command>后可接受字符'Q'命令，终止程序的运行。

【本章小结】

本章介绍了 C++的流程控制语句，包括分支语句：if、if-else 语句，多分支 switch 语句；循环语句：for、 while、 do-while 语句；break、 continue 语句。本章的后面还给出了一个综合运用流程控制语句实例。

if 语句与 switch 语句可以互换使用。对于多分支的情形，我们常用 switch 语句，如用 if-else 语句，会使程序显得复杂，可读性较差。

三种循环语句也可以互换使用。对于有确定次数的循环，我们常用 for 语句，要注意 while 循环语句与 do-while 的差别：do-while 循环语句至少要执行一次循环体。我们编程时，常常是循环可能一次也不执行，此时就不能用 do-while 循环语句，而要用 while 循环语句或 for 循环语句。

【学习目标】

◇ 掌握函数的概念、定义和调用方法。

◇ 理解 C++函数参数传递的方法：值传递和引用传递。掌握函数参数值传递的过程，并能够灵活运用。后面的章节中，将进一步学习引用传递。

◇ 理解内联函数的概念、作用，会定义内联函数。

◇ 理解函数重载的概念、作用，能够熟练地定义和运用重载的函数。

◇ 理解递归的概念，并能运用递归的方法解决一些实际问题。

◇ 理解变量的作用域与生存期的概念，能够理解全局变量、局部变量、静态变量的概念和用法。

【重点难点】

本章的重点是要掌握函数的定义、声明、调用的方法，函数参数传递方式、值传递的过程和运用，局部变量、全局变量和静态变量的概念与运用。函数的调用过程，带缺省参数的函数的定义和调用，递归的概念是本章的难点。

【学习方法指导】

C++语言支持结构化的程序设计，结构化的程序就是由函数组成的，所以，要把函数掌握好。本章内容和概念较多，有些内容有难度，需要多练、多体会和思考，才能完全掌握。一定要把函数参数的传递过程搞清楚，这也有助于对指针、引用及函数参数引用传递等概念的理解。递归的概念较难理解，可以跟踪一些简单的递归函数的执行过程，加深对递归概念的理解。

【知 识 点】

函数定义；函数声明；实参；形参；值传递；引用传递；内联函数；递归调用；

局部变量；全局变量；静态变量；函数重载

语句是构造程序的最基本单位。当我们用程序语句编写的程序越来越大，越来越复杂的时候，为了使程序更简洁、可读性更好、更便于复用，以及更便于维护，就有必要将把它分成若干个模块，每个模块完成一项任务。在 C++中，这些模块就是一个个的函数。函数也是 C++语言构造程序的重要的基本单位。**什么是模块化的程序设计方法：**

人们在求解一个复杂的问题的时候，通常采用逐步分解、分而治之的方法。也就是把一个大问题分解为几个比较容易求解的小问题，然后分别求解。程序员在设计一个复杂的应用程序时，往往也是把整个程序划分为若干个功能较为单一的程序模块，然后分别予以实现，最后再把所有的程序模块象搭积木一样搭起来，这种在程序设计中分而治之的策略，被称为模块化程序设计方法。

在 C++中，这些模块就是一个个的函数。我们可以用一个比方来更加生动地说明函数到底是什么：

假设您因为工作压力太大，感到有些疲倦，决定去大连的海滨去度假休息一周。可是由于您太忙，不能亲自安排一切行程，您便选择了一家旅行社来帮您做这些事情。该旅行社通过电话为您预订车票和旅馆住宿，车站将车票发给旅行社，而旅馆也将必要的预定单据发给旅行社。然后旅行社把机票和这些单据发给你本人，这样，您足不出户，所要求的一切就已经都准备得妥妥当当了。

本来您有任务要执行（买票和定旅馆），但是您把这些任务委托给旅行社了。旅行社再通过有关途径取得相关的票和单据。实际上旅行社如何完成这些工作对您而言是不可见的。您知道并关心的只是您传递给旅行社的信息（你的旅行日程、目的地）和旅行社返回给您的东西（票和单据），这里旅行社所做的工作就相当于一个函数。函数的本质有两点：

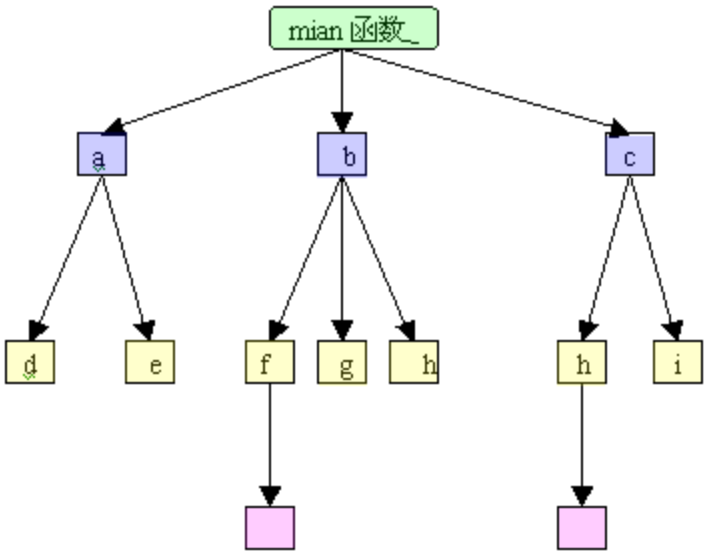
（1） 函数由能完成特定任务的独立程序代码块组成，如有必要，也可调用其它函数，来产生最终的输出。

（2） 函数内部工作对程序的其余部分是不可见的。

一个 C++程序里包含一个主函数（即 main 函数）和若干个其它函数。由主函数调用其它函数，其它函数之间也可以互相调用。

我们用下图来说明程序，函数以及语句的关系：

 **图 5-1**



函数定义的一般形式如下：

返回类型 函数名（参数表列）

{

语句系列

return 合适类型数值

}

函数的定义包括以下几个部分：

1. 函数名
2. 参数表列
3. 返回类型
4. 函数体

1. 函数名： 一个符合 C++语法要求的标识符，定义函数名与定义变量名的规则是一样的，但应尽量避免用下划线开头，因为编译器常常定义一些下划线开头的变量或函数。函数名应尽可能反映函数的功能，它常常由几个单词组成。

如 VC 中的按下鼠标左键的响应函数为：OnLButtonDown，这样就较好的反映了函数的功能。在上例中，我们可以将"买票"和"订旅馆"作为函数名。

2. 参数表列：0 个或多个变量，用于向函数传送数值或从函数带回数值，每一个参数都有自己的类型，它不同于变量定义，因为几个变量可以定义在一起，例如?quot;int i，j，k;"。如果参数表列中参数个数为 0，我们称之为无参函数，无参函数可以定义为：

"返回类型 函数名( ){…}"或"返回类型 函数名(void){…}"

旅行日程和目的地就是我们传送给上例中的函数的数值，当然如果你对日程和目的地都没有什么特殊的要求，那也可以完全交给旅行社一手包办，这样便相当于一个无参函数了。

3. 返回类型：指定函数用 return 返回的函数值的类型，如果函数没有返回值，返回类型应为 void。

上例中返回值就是旅行社将要交给你的票和单据，而返回值的类型则可以看成是票和单据的类别。

4. 函数体：花括号中的语句称为函数体，一个函数的功能，通过函数体中的语句来完成。

上例中的函数体即为旅行社为买票和订旅馆进行的具体操作。

下面我们看一个具体的函数定义的例子：

例 5-1

```
int GetIntegerInRange(int low, int high)
{
   int res;
   do {
      cout << "Enter an integer in the range " << low
      << " ... " << high << " :";
      cin >> res;
      } while (! ((res >= low) && (res <= high)));
return res;
}
```

int GetIntegerInRange(int low, int high)行定义函数名为 GetIntegerInRange，返回类型为 int，且指定函数需要两个整型值，并通过 low 和 high 引用。在函数体内定义了整型变量 res，并通过 return 语句返回该变量的值。(注意)变量 res 的类型必须与函数的返回类型一致，否则，系统以函数返回类型为准，而对 res 的数值进行强制类型转换，使其与函数的返回类型保持一致。

return 语句的返回值通常是一个表达式，例如 5-2：

一个函数也可以有多个 return 语句，在函数的执行过程中，遇到任一个 return 语句，将立即停止执行，并返回到调用函数。例如 5-3：

例 5-2

```
return res;
return n % MAXSIZE;
return 3.5 + x*(6.1 + x*(9.7 + 19.2*x));
```

上面的 return 语句也可以写成下面的形式：

```
return (res);
return (n % MAXSIZE);
return (3.5 + x*(6.1 + x*(9.7 + 19.2*x)));
```

即对返回值的表达式加一对圆括号。不过，这圆括号是多余的，可以省略掉。

例 5-3

```
double absolute_value(double number)
{
   if (number >= 0)
      return number;
   else
      return 0 - number;
}
```

这里我们用一个很简单的函数定义的例子 5-4 来进一步加以说

明

例 5-4

```
int Inc(int a)
{
  a++;
  return a;
}
```

这个函数的功能为增 1，其中，int Inc(int a)行定义了：函数名-Inc，参数-整型数 a，返回类型-整型，这说明该函数中用 return 语句返回的值应该是整型值。在花括号中的即为函数体，这个函数比较短，其函数体只有两行语句：

```
a++;
return a;
```

这两行语句完成 a 自增 1 的功能，返回的是 a 增 1 后的值，返回值与返回类型是匹配的。下面是一个不带有参数和返回值的函数。

例 5-5

```
#include <iostream.h>
void Hello()
{
  cout<<" Hello,world"<<endl;
}
```

这里 Hello 函数是一个无参函数，当被其它函数调用时，输出 Hello world 字符串。

如果一个函数调用另一个函数，在调用函数中必须对被调用函数进行说明，函数说明类似于函数定义，不过没有实现代码，函数说明的一般形式如下：

返回类型 函数名(参数表列);

函数说明是一个语句，所以要以分号结束。C++中，函数说明也称为函数原型，下面是函数说明的例子：

```
void PrintStats(int num, double ave, double std_dev);
int GetIntegerInRange(int low, int high);
```

一些函数的说明，我们放在头文件中，也有一些函数的说明放在文件的头部，而函数的定义放在该文件后面。我们知道，C++应用程序是从 main 函数开始执行，main 函数在文件中的位置并没有特别的要求，它可以在文件的头部、中部或尾部。下面我们举一个简单的应用程序例子 5-6，看看如何进行函数的说明。

例 5-6

```
#include <iostream.h>
int Power (int base, unsigned int exponent)
{
  int result = 1;
```

```
    for (int i = 0; i < exponent; ++i)
        result *= base;
    return result;
}
void main (void)
{
    cout << "2 ^ 8 = " << Power(2,8) << '\n';
}
```

上例可以表示为:

```
#include <iostream.h>
int Power (int base, unsigned int exponent);
void main (void)
{ … }
int Power (int base, unsigned int exponent)
{ … }
```

或表示为：

```
#include <iostream.h>
void main (void)
{
    int Power (int base, unsigned int exponent);
    cout << "2 ^ 8 = " << Power(2,8) << '\n';
}
int Power (int base, unsigned int exponent)
{ … }
```

在 C++中说明函数时，可以定义缺省参数，例如：

void ShowMessage(char*Text, int Length = -1, int Color = 0);

调用该函数时，可以不必传送参数 Length 和 Color 的值，此时，编译器将 Length 的缺省值-1 和 Color 的缺省值 0 传递到函数。

注意：如果定义某个缺省参数，必须定义所有后续参数的缺省值。例如，下面的函数说明是错误的：

void ShowMessage(char*Text, int Length = -1, int Color);

因为，参数 Length 定义了缺省值，所以其后的参数 Color 也必须有缺省值。

我们定义了一个函数，完成某种功能，就要涉及到调用这个函数，调用一个函数之前必须对该函数进行说明。函数调用由函数名和函数调用运算符（）组成，（）内有 0 个或多个逗号分隔的参数（称为实参）。每一个参数是一个表达式，且参数的个数与参数的类型要与被调函数定义的参数（称为形参）个数和类型匹配。

当被调函数执行时，首先计算实参表达式，并将结果值传送给形参，然后执行函数体，返回的返回值被传送到调用函数。

如果函数调用后有返回值，调用表达式可以用在表达式中，而无参函数的调用是一个单独的语句。下面是函数调用的一个实例 5-7：

例 5-7

```
#include<iostream.h> //C++库函数说明
int area(int length, int width); // area 函数说明
void main() //主函数，程序从该函数开始执行
{
   int this_length, this_width;
   cout << "Enter the length: "; //调用 cout 和 cin 库函数
   cin >> this_length;
   cout << "Enter the width: ";
   cin >> this_width;
   cout << "\n";
   cout << "The area of a " << this_length << "x" << this_width;
   cout << " rectangle is " << area(this_length, this_width); //调用 area 函数
}
int area(int length, int width) // area 函数定义
{
   int number;
   number = length * width;
   return number;
}
```

对于无参函数的调用形式应该是：

函数名( )

例如，对于函数：

```
int TPSOKTSNK();
```

可以这样调用：

```
if(TPSOKTSNW()) {
   语句系列
}
```

如果调用形式写成：

```
if(TPSOKTSNW) {
   语句系列
}
```

它仍然符合 C++语法，但意义发生了变化。它是检查函数 TPSOKTNSW()是否有地址，有关地址的概念，我们在后面要讲到。如果该程序连接成功，则 TPSOKTNSW()有地址，if 语句的条件为真。

例 5-8

```
#include <iostream.h>
int max(int a,int b)
{
   if(a>b)return a;
   else return b;
```

```
}
void main()
{
   int max(int a,int b);
   int x,y,z;
   cout<<"input two numbers"<<endl;
   cin>>x>>y;
   z=max(x,y);
   cout<<"maxmum="<<z;
}
```

程序的第 2 行至第 6 行为 max 函数定义。进入主函数后，因为准备调用 max 函数，故先对 max 函数进行说明(程序第 9 行)。可以看出函数说明与函数定义中的函数头部分相同，但是末尾要加分号。程序第 13 行为调用 max 函数，并把实参 x、y 传送给 max 的形参 a、b。max 函数执行的结果（a 或 b)，返回给调用函数 main 中的变量 z，且 main 函数将 z 值输出。

5.3.1 带缺省参数的函数的调用

函数说明时可以带缺省的参数，我们还是看 5.2 节的例子：

```
void ShowMessage(char*Text, int Length = -1, int Color = 0);
```

调用 ShowMessage 函数时，可以指定一个、二个或三个参数：

```
ShowMessage("Helloo");
ShowMessage("Helloo", 5);
ShowMessage("Helloo", 5, 8);
```

注意：说明缺省参数类似，如果调用函数时省略缺省参数，必须省略所有后续参数，例如，下面的调用是错误的：

```
ShowMessage("Helloo", , 8);
```

5.3.2 参数传递

我们已经知道：函数调用时，要将调用函数中实参传送给被调函数中的形参。在 C++中，有两种参数传递方式，即值传递和地址传递（或称引用传递）。地址传递我们在后面介绍，这里介绍值传递的方法。

函数在被调用以前，形参变量并不占内存单元，当函数被调用时，形参变量分配有存储单元，并将相应的实参变量存储单元的值复制到形参变量单元。所以，被调函数在执行过程中，如果形参变量的值发生了变化，并不影响实参变量的值。例如 5-9：

例 5-9

```
#include <iostream.h>
void Foo (int num)
{
   num = 0;
   cout << "num = " << num << '\n';
```

```
}
int main (void)
{
   int x = 10;
   Foo(x);
   cout << "x = " << x << '\n';
   return 0;
}
```

函数 Foo 有一个形参变量 num，当它在 main 函数中被调用时，实参变量 x 的值传送到 num，且 x 和 num 在不同的内存单元中。在函数 Foo 中，虽然 num 被设置为 0，但并不影响 x 的值，程序运行结果如下：

num = 0;

x = 10;

值传递也可以称之为"赋值调用"，这种方式是把实参的值复制到函数的形式参数中，函数中的形式参数的任何变化都不会影响到实参变量的值。如例 5-10：

例 5-10

```
#include <iostream.h>
void Foo (int first, int second, int third)
{
   cout << "Original function values "
   << "first = " << first
   << "second = " << second
   << "third = " << third
   <<endl;
   first += 50;
   second += 50;
   third += 50;
   cout << "Ending function values "
   << "first = " << first
   << "second = " << second
   << "third = " << third
   <<endl;
}
void main (void)
{
   int x = 1, y = 2, z = 3;
   Foo(x,y,z);
   cout << "Ending values in main "
   << "first = " << x
```

```
    << "second = " << y
    << "third = " << z
    <<endl;
}
```

编译并执行，屏幕将显示如下结果：

Original function values first = 1 second = 2 third = 3

Ending function values first = 51 second = 52 third = 53

Ending values in main first = 1 second = 2 third = 3

可以看到，函数对变量所作的改变只在函数内部有效。

5.3.3 函数调用过程

C++函数调用是基于栈存储结构来实现的。当函数被调用时，为函数形参、返回值和其它变量等在栈中分配内存空间，当函数返回时，这些内存空间又被释放，以便于再利用。例如 5-11，假定 main 函数调用函数 Solve，而函数 Solve 又调用函数 Normalize

例 5-11

```
int Normalize (void)
{ //... }
int Solve (void)
{ //...
   Normalize();
   //...
}
int main (void)
{
   //...
   Solve();
   //...
}
```

main、Solve 和 Normalize 函数在栈中的内存分布如下：

图 5-1 执行 Normalize 时的栈内存分布

应该注意到：调用一个函数，涉及到为该函数在栈中分配内存和释放内存的开销。对于大多数的函数而言，同函数的实际计算量相比可以忽略不计。

例 5-12

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

```
void reverse(int n)
{
   int num;
   if(n<=1)
   {
      cin>>num;
      cout<<"Reverse numbers"<<endl;
      cout<<num<<endl;
   }
   else
   {
      cin>>num;
      reverse(n-1);
      cout<<num<<endl;
   }
}
void main()
{
   int i;
   i=5;
   cout<<"Enter number"<<endl;
   reverse(i);
}
```

5.4 inline 函数

我们看下面的函数，函数体中只有一行语句：

```
double Average(double total, int number)
{
   return total/number;
}
```

定义这么简单的函数有必要吗？实际上，它还是有一些优点的：第一，它使程序更可读；第二，它使这段代码可以重复使用。但是，它也有缺点：当它被频繁地调用的时候，由于调用函数的开销，会对应用程序的性能有损失。例如，Average 在一个循环语句中重复调用几千次，会降低程序的执行效率。

那么，有办法避免函数调用的开销吗？对于上面的函数，我么可以把它定义为内联函数的形式：

```
inline double Average(double total, int number)
{
   return total/number;
}
```

函数的引入可以减少程序的目标代码，实现程序代码的共享。但是，函数调用也需要一些时间和空间方面的开销，因为调用函数实际上将程序执行流程转移到被调函数中，被调函数的程序代码执行完后，

再返回到调用的地方。这种调用操作要求调用前保护现场并记忆执行的地址，返回后恢复现场，并按原来保存的地址继续执行。对于较长的函数这种开销可以忽略不计，但是对于一些函数体代码很短，但又被频繁地调用的函数，就不能忽视这种开销。引入内联函数正是为了解决这个问题，提高程序的运行效率。

在程序编译时，编译器将程序中出现的内联函数的调用表达式用内联函数的函数体来进行替换。由于在编译时将函数体中的代码替代到程序中，因此会增加目标程序代码量，进而增加空间开销，而在时间开销上不象函数调用时那么大，可见它是以目标代码的增加为代价来换取时间的节省。

上面的内联函数同它的非内联函数相比，仅仅是多了一个关键字 inline，它们在功能上并没有区别：前者也是有两个形参，一个 double 型，一个 int 型，返回值是 double 型，且两个形参相除后所得的商作为结果返回。但是，编译器对这两个函数的调用过程的处理是不同的。对于内联函数的调用，编译器是将其函数体放在调用的地方，没有非内联函数调用时的栈内存的创建和释放开销。但是，所执行的计算是完全相同的。

使用内联函数时应注意以下几个问题：

（1） 在一个文件中定义的内联函数不能在另一个文件中使用。它们通常放在头文件中共享。

（2） 内联函数应该简洁，只有几个语句，如果语句较多，不适合于定义为内联函数。

（3） 内联函数体中，不能有循环语句、if 语句或 switch 语句，否则，函数定义时即使有 inline 关键字，编译器也会把该函数作为非内联函数处理。

（4） 内联函数要在函数被调用之前声明。例如下面的代码将内联函数放在函数调用之后声明，不能起到预期的效果。

例 5-12

```
#include <iostream.h>
#include <conio.h>
int add(int x, int y)
void main()
{
   cout << "3 + 5 = " << add(3, 5) << endl;
   cout << "4 + 23 = " << add(4, 23) << endl;
   cout << "34 + 45 = " << add(34, 45) << endl;
   getch( );
}
inline int add(int x, int y)
{
   return (x + y);
}
```

所谓递归是指函数能够调用自身，它是一种通用的编程技术，为我们解决某些问题提供了极大的方便。下面我们来看一个阶乘的例子，对于一个整数 n，其阶乘定义为：

$n! = 1 \times 2 \times 3 \times \ldots \times n$

$0! = 1$

阶乘也可以采用如下的递归定义：

$$n!=\begin{cases}1 & \text{if } n=0\\ nx(n-1)! & \text{if } n>0\end{cases}$$

使用第一个定义，我们计算 5 的阶乘：

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

下面用第二个定义计算 5 的阶乘：

5! = 5 * 4!

= 5 * 4 * 3!

= 5 * 4 * 3 * 2!

= 5 * 4 * 3 * 2 * 1!

= 5 * 4 * 3 * 2 * 1 * 0!

= 5 * 4 * 3 * 2 * 1 * 1

= 120

下面是用第一个定义计算阶乘的 C++函数，它是用循环来实现的：

```
int Factorial(int n)
{
   int result = 1;
   for (int i = 1; i <= n; i++) {
         result *= i;
      }
   return result;
}
```

下面再看看用阶乘的第二个定义计算阶乘的 C++函数：

```
int RecursiveFactorial(int n) {
   if (n == 0)
      return 1;
   else return n * RecursiveFactorial(n - 1);
}
```

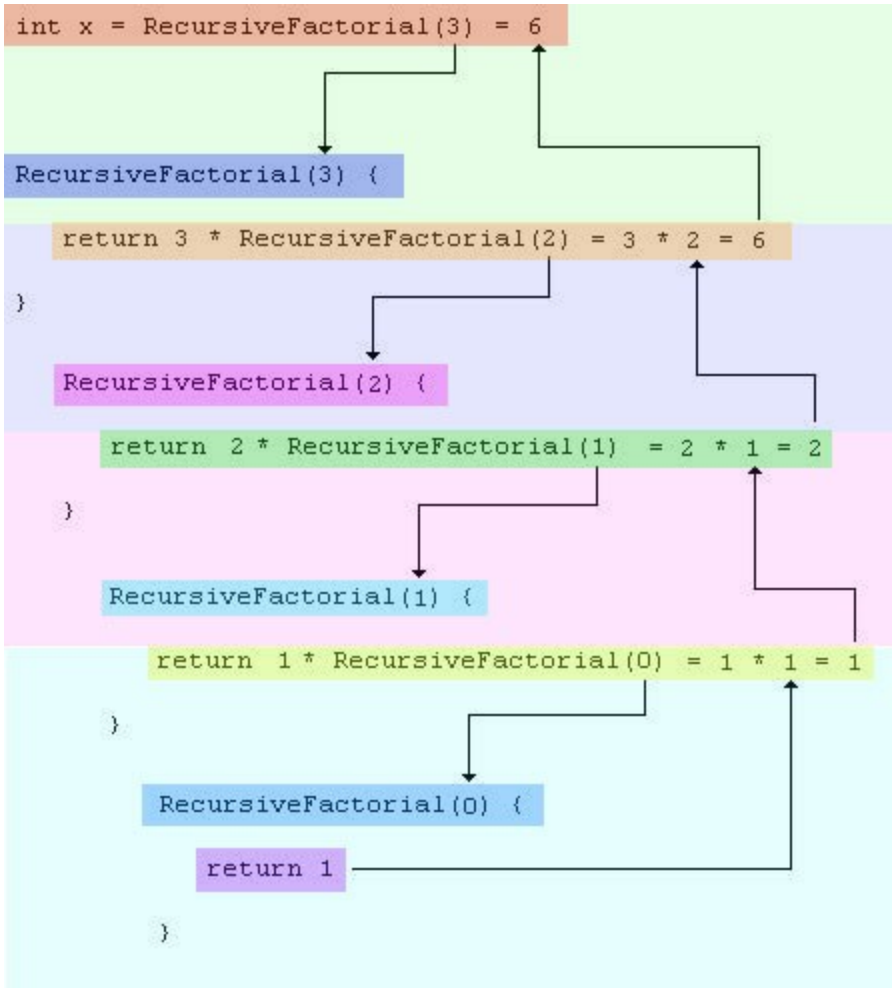
RecursiveFactorial 函数的结构具有一定的通用性，所有递归调用的函数的结构都是一样的：

（1） 要有递归终止条件检查，RecursiveFactorial 递归终止的条件是 n=0。

（2） 如果不满足递归终止的条件，则返回涉及递归调用的表达式。在 RecursiveFactorial 函数中，返回的表达式是：n * RecursiveFactorial(n - 1)。

图 5-8 给出了函数 RecursiveFactorial 计算 3!的过程：

图 5-8 3！的计算过程



一个函数在它的函数体内调用它自身称为递归调用， 这种函数称为递归函数，C ++允许函数的递归调用。在递归调用中，递归函数反复调用其自身，每调用一次就进入新的一层。例如：

```
int f (int x)
{
  int y = f(x);
  //递归调用 f()函数
  return y;
}
```

这个函数就是一个递归函数。 但是运行该函数将无休止地调用其自身，这显然是不正确的，会导致堆栈溢出。如果在 main( )函数中调用该函数：

```
void main()
{
  int i = 4;
  int j = f(i);
}
```

用 Visual C++编译、运行该程序，会出现图 5-9 的错误信息。

为了防止递归调用无终止地进行， 必须在函数内有递归调用终止的条件，满足某种条件后就不再作递归调用，然后逐层返回。

图 5-9

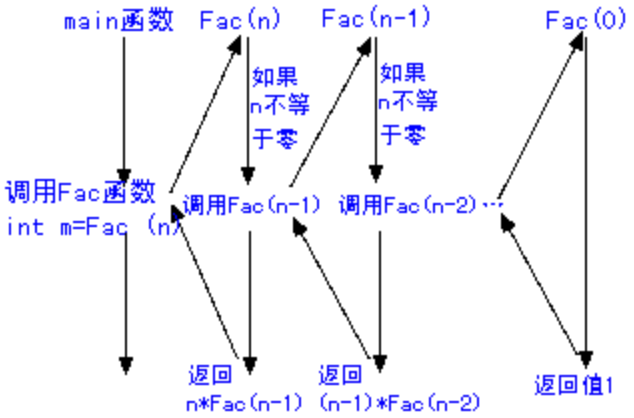
例如，我们用函数递归来编写计算阶乘的 C++函数：

例 5-15

```
int Fac(int n)
{
  if (n == 0)
       return 1;
  else return n *Fac(n - 1);
}
```

程序中给出的函数 Fac 是一个递归函数。主函数调用 Fac 后即进入函数 Fac 执行，如果 n=0 时函数停止执行，否则就递归调用 Fac 函数自身。由于每次递归调用的实参为 n-1，即把 n-1 的值赋予形参 n，最后当 n-1 的值为 0 时，将使递归终止，并可逐层返回。

图 5-10



输出 m 的值，结束

下面我们再举例说明该过程：设执行本程序时输入 5， 即求 5 的阶乘。在主函数中的调用即为 Fac (5)，进入 Fac 函数后，由于 n = 5，不等于 0 或 1，故应执行 n*Fac (n-1)，即 5*Fac (5-1)。该语句对 Fac 作递归调用即 Fac (4)。 如此逐次递归展开，进行五次递归调用后，Fac 函数形参值变为 0，故不再继续递归调用而开始逐层返回到主调函数。Fac(0)的函数返回值为 1，Fac(1)的函数返回值为 1*1=1，ff(2)的返回值为 1*2=2，ff(3)的返回值为 2*3=6，ff(4) 的返回值为

6*4=24，最后返回值 ff(5)为 24*5=120。

应该指出：递归是一种编程技术，并不是用于计算阶乘的。用非递归的方法计算阶乘效率更高，我们用递归来计算阶乘，是因为它的递归调用易于实现和理解。

我们再看 an 递归定义的例子：

$$a^n = \begin{cases} 1 & (n=0时) \\ a^{n/2} \times a^{n/2} & (n为偶数时) \\ a \times a^{n/2} \times a^{n/2} & (n为奇数时) \end{cases}$$

这个定义的 C++函数实现如下：

例 5-16

```
double Power(double base, int expo) {
  if (expo == 0) {
      return 1.0; // x0 = 1
  }
  else {
      double semi = Power(base,expo/2);
      if (expo % 2 == 0) { //  偶数幂
          return semi * semi;
      }
      else { // 奇数幂
          return base*semi*semi;
          }
       }
}
```

注意：编写递归的函数时，一定注意要有递归调用终止的条件，且每调用一次就向调用终止更靠近一步。这可确保递归调用能够正常结束，而不至于导致系统内存耗尽而崩溃。递归函数总比功能相同的非递归函数慢，因为每次调用函数时都有调用开销。任何用递归编写的函数都可使用循环结构代替，以消除递归。

在 C++中，变量有效的范围（称为变量的作用域）和被存储的时间（称为变量的存储期或生存期）都是不同的。如果按变量的作用域来分类的话，变量可以分为局部变量和全局变量；如果按变量的存储期来分类的话，变量可以分为外部变量、静态变量、自动变量、寄存器变量。下面对各种类型的变量分别进行介绍。

5.6.1 作用域的概念

在 C++中变量、函数和类（我们在后面要介绍类）的作用域是不同的，在函数和类外定义的变量，具有全局的作用域，这些变量称之为全局变量。同样，在类外定义的函数，我们称之为全局函数。

在讨论函数形参变量时曾经提到：形参变量只在被调用期间才分配内存单元，调用结束则立即释放。这一点表明形参变量只有在函数内才是有效的， 离开该函数就不能再使用了。不仅对于形参变量，C++中所有的变量都有自己的有效范围。变量有效的范围，称之为变量的作用域。变量说明的方式不同，其作用域也不同。C++语言中的变量，按作用域范围可分为两种，即局部变量和全局变量。

例如：

```
int year = 1994; //  全局变量
int Max (int, int); //  全局函数
int main (void) //  全局函数
{
  //...
}
```

未被初始化的全局变量，系统自动初始化为 0。

把一些 C++语句，我们由一对花括号括起来，称之为语句块或块，在块中定义的变量作用域在块内，称之为局部作用域，在局部作用域中定义的变量称之为局部变量。例如，在函数和在一个复合语句中定义的变量都是局部变量。函数的形参的作用域在函数内，也是局部变量。局部变量只在局部作用域内有效，我们称之为可见，离开了其所在的局部作用域便无效，或称之为不可见。

在同一个作用域内，变量不能同名，否则，程序编译时，编译器会给出变量重复定义的错误。不同的作用域内，变量同名不会出现语法问题，但可能会使某些变量不能访问。我们看程序 5-1 一个局部作用域嵌套的情形：

程序 5-1

```
int xyz;// xyz 全局变量
void Foo (int xyz)// xyz  是  Foo 函数中的局部变量
{
  if (xyz > 0) {
    double xyz;// xyz 是 if 语句块中的局部变量
    //...
  }
}
```

上面的程序段中有三个作用域，全局作用域及两个局部作用域，且 if 语句块作用域嵌套在 Foo 函数作用域内。在 if 语句块内，访问不到 Foo 函数中的 xyz 变量，这说明内部作用域会覆盖外部作用域。需要注意的是：运用全局运算符::，我们可以在局部作用域中访问到全局变量。例如，在上面的 if 语句块中，我们增加一条语句：

```
double t = ::xyz;
```

就可以把全局变量 xyz 的值赋给局部变量 t。

5.6.2 局部变量和全局变量

我们已经知道了什么叫全局变量，什么叫局部变量。一个应用程序可能包含多个源文件，而一个源文件可能包含多个函数。一般说来，全局变量的作用范围是定义点起至文件结束为止，局部变量的作用范围是从定义点起至该局部变量所在块的尾部为止。变量的存储期也限制在其作用域内。例如：全局变量的存储期与应用程序的生存期相同，局部变量是在进入作用域时创建，而在退出作用域时被销毁。

一、局部变量

由于局部变量的存储期是是在其所在的局部作用域内，其内存也是由系统自动分配的，所以，它

也被称为自动变量。可以用一个关键字 auto 显式指定一个变量是自动变量。例如：

```
void Foo (void)
{
  auto int xyz; //  等价于 int xyz;
  //...
}
```

一般说来，我们很少使用 auto 关键字。因为 C++中，局部变量默认为自动变量。

我们已经知道，变量是存放在内存中。如果程序对一些频繁使用的变量（如循环变量），要求更高的访问效率，可以把这些变量保存在寄存器中，保存在寄存器中的变量，我们也称之为寄存器变量。

寄存器变量也是局部变量，定义寄存器变量需要用 register 关键字，例如：

```
for (register int i = 0; i < n; ++i)
sum += i;
```

需要注意的是：有时我们用 register 关键字定义了寄存器变量，编译器也可能不把该变量放在寄存器中，而放在内存中。因为机器的寄存器个数是有限，当我们申请寄器存放变量时，可能所有的寄存器都在被使用中。

局部变量也称为内部变量。局部变量可以在函数内或者是复合语句内定义，其作用域仅限于被定义的函数内或复合语句内。

例 5-17

```
int f1(int a)
//函数 f1
{
  int b,c;
  ……
}
//a,b,c 作用域
int f2(int x)
//函数 f2
{
  int y,z;
}
//x,y,z 作用域
void main()
{
  int m,n;
}
//m,n 作用域
```

在函数 f1 内定义了三个变量，a 为形参，b、c 为一般变量。在 f1 的范围内 a、b、c 有效，或者说 a、b、c 变量的作用域限于 f1 内。同理，x、y、z 的作用域限于 f2 内， m、n 的作用域限于 main 函数内。

主函数中定义的变量也只能在主函数中使用，不能在其它函数中使用。同时，主函数中也不能使用其它函数中定义的变量。因为主函数也是一个函数，它与其它函数是平行关系（不过 main 函数只能被系统调

用，而不能被其它的函数调用）。

允许在不同的函数中使用相同的变量名，因为它们分配不同的存储单元，互不干扰，也不会发生混淆。

二、全局变量

我们知道：全局变量是在函数和类外定义的，所以也称之为外部变量。全局变量一旦定义，从定义点开始至文件结束的所有函数都可以使用该变量。

一般情况下，我们把全局变量的定义放在引用它的所有函数之前。但是，如果在全局变量定义点之前的函数要引用该全局变量或另一个源文件中的函数要引用该全局变量，需要在函数内对要引用的全局变量加 extern 说明。例如 5-18：

例 5-18

```
#include <stdio.h>
int max(int x, int y);
main()
{
   extern int a, b; //全局变量说明，而非定义
   printf("%d\n", max(a, b));
   return 0;
}
int a = 13, b = -8;
int max(int x, int y)
{
   int z;
   z = x > y ? x : y;
   return z;
}
```

注意：用 extern 说明全局变量的时候，不能给初值。例如：

extern int size= 10; // 不再是说明!

因为这会使得 size 变成变量的定义，而不是说明，编译器会为它分配内存。如果别的地方定义了全局变量 size，该程序在编译时，编译器会给出变量重复定义的错误。

使用全局变量，在我们编程中，有时会来一些方便。但是，它也有许多副作用：在程序的整个执行过程中始终占用内存空间，使程序的可读性、通用性和可移植性降低等，建议不在必要时，不使用全局变量。

5.6.3 静态变量

如果在变量的定义前加上 static 关键字，就定义了静态变量。例如 5-21：

在上面的程序段中，我们定义了一个静态全局变量 shortestRoute 和一个静态局部变量 count。静态变量与全局变量具有相同的存储期，它们均与应用程序的生存期相同。但是，静态的全局变量只能在定义该全局变量的文件中访问，静态的局部变量只能在定义该局部变量的局部作用域中访问。

静态变量这种特性是有用的，如果我们需要某些全局变量只在本文件中访问，就可以把它们定义为静态的，这也可以减少了不同文件中定义的同名的全局变量而发生冲突的可能性，从而提高了程序的可移植性。

static 关键字不仅可以放在变量的定义前，也可以放在函数的定义前。在全局函数的定义前加上了 static 关键字，就称为静态全局函数。例如 5-22：

在同一源文件中，允许全局变量和局部变量同名。在局部变量的作用域内，全局变量不可见，不过，使用全局运算符::可以访问到全局变量。

例 5-21

```
static int shortestRoute; // 静态全局变量
void Error (char *message)
{
  static int count = 0; // 静态局部变量
  ...
}
```

当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该静态变量的源文件内有效，而在其它源文件中不能使用它。

例 5-22

```
static int FindNextRoute (void) // 仅在本文件中访问
{
  //...
}
```

与静态全局变量的特性相似，静态全局函数也只能在定义该全局函数的文件中访问。

静态局部变量的特性也是很有用的。例如，假定我们在一个函数中定义了一个局部变量，需要该局部变量在函数退出时并不释放，下一次进入该函数时，局部变量原来的值还存在，我们就可以把该局部变量定义为静态的。

例 5-23

```
#include <iostream.h>
void func()
{
  static int i = 0;
  cout << " i = "<<++i << endl;
}
void main()
{
  for(int x = 0; x < 10; x++)
  func();
}
```

本程序的运行结果为：

i =1 i =6

i =2 i =7

i=3 i =8

i=4 i =9

i=5 i =10

类似于全局变量，静态局部变量自动初始化为 0。

5.6.4 变量的存储

除寄存器变量存储在寄存器中外，其它类型变量均存储在内存中。变量在内存中的分布区域参见图 5-3，它是 C++程序内存分布的一个总体结构，实现细节依赖于不同的编译器、机器结构和操作系统。

程序代码：存放应用程序的可执行代码。

全局数据等：存放应用程序的全局数据和静态数据，它们的生命期与应用程序的生命期一相同，即从应用程序开始运行时起至停止运行时止。全局数据与静态数据的内存空间均是在编译时分配的。

栈：局部变量存放在栈中，它们的内存空间是在程序运行时动态分配的。

堆：C++中，对内存用 new 命令创建（我们在后面要介绍），用 delete 命令释放。

C++中，当有一组函数完成相似功能时，函数名允许重复使用，编译器根据参数表中参数的个数或类型（不能根据形参变量名）来判断调用哪一个函数，这就是函数的重载。

重载函数只要其参数表中参数个数或类型不同，就视为不同的函数。例如 5-26：

例 5-26

```
#include <stdio.h>
void show(int val)
{
   printf("Integer: %d\n", val);
}
void show(double val)
{
   printf("Double: %lf\n", val);
}
void show(char *val)
{
   printf("String: %s\n", val);
}
int main()
{
   show(12);
   show(3.1415);
   show("Hello World\n!");
   return (0);
}
```

上面的程序段定义了三个有相同名字的函数 show，但形参不同，分别为 int、double 和 char *类型(这

是指向字符的指针类型，有关指针类型，在第七章作详细介绍)。

定义重载的函数时， 我们应该注意以下几个问题：

（1） 避免函数名字相同，但功能完全不同的情形。例如上面的重载函数 show 的功能就是相关的，它们均是向屏幕打印信息。

（2） 函数的形参变量名不同不能作为函数重载的依据。

（3） C++中不允许几个函数名相同、形参个数和类型也相同，仅仅是返回值不同的情形，否则，程序编译时会出现函数重复定义的错误。这是因为我们编程时常常忽略返回值，例如下面的代码段：

printf("Hello World!\n");

就没有函数 printf 函数返回值的信息（顺便提及：printf 函数返回值是一个整数，表示打印出的字符的个数。这个返回值，我们实际上从来不用。)。如果两个重载的 printf 函数仅仅是返回值不同，编译器便不能区分它们。

（4） 函数重载有时可能会产生意想不到的结果。例如：

show(0);

在上面给定的三个函数中，0 可以被解释为一个空指针 NULL，即(char *)0，也可以解释为一个整数 0。C++选择调用有整型参数的 show 函数，这也可能不是你所期望的结果。

（5） 调用重载的函数时，如果实参类型与形参类型不匹配，编译器会自动进行类型转换。如果转换后仍然不能匹配到重载的函数，则会产生一个编译错误。例如 5-27：

例 5-27

```
#include <stdio.h>
void add(int val)
{
   printf("Int:%d\n", val)
}
void add(long val)
{
   printf("Int:%ld\n", val)
}
void main()
{
   add(3.2); //调用发生歧异
}
```

编译器会由于无法决定将 3.2 转换成 int 还是 long 而产生调用发生歧异错误。

所谓函数重载是指同一个函数名可以对应着多个函数的实现。例如，可以给函数名 add()定义多个函数实现，该函数的功能是求和，即求两个操作数的和。其中，一个函数实现是求两个 int 型数之和，另一个实现是求两个浮点型数之和。每种实现对应着一个函数体，这些函数的名字相同，但是函数参数的个数或类型不同，这就是函数重载的概念。

当调用多个同名的重载函数时，要求能够唯一地确定应执行哪一个函数，这是通过函数参数的个数和类型来区分的。所以，重载的函数要求参数个数或者参数类型上不同，否则会出现编译错误。下面是一个函数重载的例子，"int add(int i,int j)"函数实现整数的加法，"double add(double i,double j)"

函数实现双精度浮点数的加法，它们的名字相同，仅仅是参数的类型不同，min 函数的功能是在若干个整数中，求最小的整数，"int min(int a,int b)"、"int min(int a,int b,int c)"和"int min(int a,int b,int c,int d)"参数类型相同，仅仅是参数个数不同：

给定一个正数 c，编写一个函数，计算 c 开 n 次方的正实根（n 为大于 1 的正整数）。

算法：

程序：

算法：

要计算 $s=\sqrt[n]{c}$，相当于求方程：

$$x^n - c = 0$$

的正实根。解决这个问题，我们可以使用牛顿迭代公式。

对于一般的线性方程 $f(x)=0$，用牛顿迭代法求解的方法是：给定一个初值 x_0，用下面的迭代公式：

$$x_{k+1} = x_k - f(x_k)/f'(x_k), \quad k = 0,1,\Lambda$$

得到一个迭代序列，当 $\dfrac{|x_k - x_{k-1}|}{|x_k|} < \varepsilon$ $x_1 = x_0 - f(x_0)/f'(x_0)$ (给定的精度)时，我们便认为 x_k 为方程 $f(x)=0$ 的根，并停止迭代。

牛顿迭代法有很明显的几何意义（参见右图 5-12），当我们选定 x_0 以后，过 $(x_0, f(x_0))$ 作 $f(x)$ 的切线，其切线方程为：

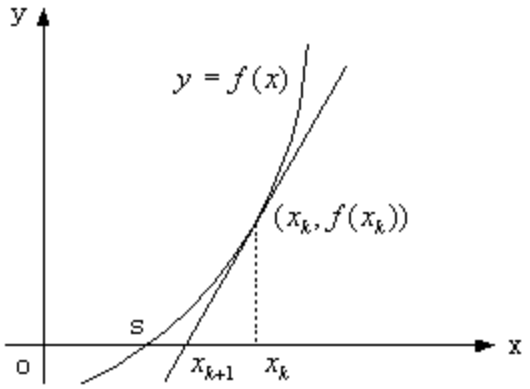
$$y - f(x_0) = f'(x_0)(x - x_0)$$

求此切线方程与 x 轴的交点，即得：

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

正因为牛顿法有这一明显得几何意义，所以也叫切线法。

图 5-12



本题中，因为 $f(x)=x^n-c$ ， $f'(x)=nx^{n-1}$ ，故牛顿迭代公式为：

$$x_{k+1}=x_k-\frac{x_k^n-c}{nx_k^{n-1}}=\frac{(n-1)x_k}{n}+\frac{c}{nx_k^{n-1}},k=0,1,\Lambda$$

在区间（0, ∞）内 $f(x)$ 单调增大， $f'(x)>0,f''(x)=n(n-1)x^{n-2}>0$ ，因此，对于任取的 $x_0\geq s$ 都收敛

【本章小结】

本章学习了函数的定义和调用方法。在调用函数时，一定要在调用之前对被调用函数进行声明。如果是外部函数还要加 extern 关键字，如果要限制函数的作用域在本文件之中，则要加 static 关键字进行限定。inline 关键字用于定义内联函数，正确地使用内联函数可以提高程序的运行效率。

函数参数值传递的方法一定要搞清楚。实参和形参占用不同的存储单元，形参值的变化不会影响到实参的值。这与函数参数的引用传递不同。

函数重载为我们编写程序提供了很大的方便。需要注意的是：重载的函数是指函数的名字相同，但至少参数个数或类型不能相同，如果参数个数和类型都相同，仅仅是返回类型不同，是不行的。

递归的概念较难理解。编写递归的函数时，一定注意要有递归调用终止条件，且每调用一次就向调用终止更靠近一步。这可确保递归调用能够正常结束，而不至于导致系统内存耗尽而崩溃。

变量的作用域和生存期是很重要的概念，要掌握全局变量、局部变量、静态变量的用法。滥用全局变量是造成名字冲突、程序错误的原因之一，因此，应尽量少用全局变量。在同一个作用域内，变量不能同名，否则，程序编译时，编译器会给出变量重复定义的错误。不同的作用域内，变量同名不会出现语法问题，但会出现变量不可见的问题。

【学习目标】

◇ 掌握数组的定义与使用方法，学会利用数组编写程序。

◇ 掌握数组元素的初始化方法。

◇ 掌握字符数组处理字符串的技巧。

◇ 熟悉字符串处理函数的功能及使用方法。

◇ 掌握数组作为函数参数的使用方法。

【重点与难点】

1.数组与字符串的灵活使用是本章的重点。

2．字符串在内存中的存储方式是难点。

3．字符串处理函数的使用也是难点。

【学习方法指导】

数组是一种常用的、重要的线性数据结构，一定要掌握好。学习中，要注意区分数组的赋值与初始化的方法不同之处。C++中没有定义字符串数据类型，而用字符数组存储字符串，要注意区分字符串数组与其它非字符串数组的不同之处。C++中有一些操作字符串的标准库函数，应该学会运用这些库函数。

【知识点】

数组； 数组的定义；一维数组；一维数组的存储；一维数组的初始化；二维数组，二维数组的存储；二维数组的初始化；字符数组；字符串处理函数；数组作函数参数

在 C++中，数组是一种集合数据类型，它由许多元素组成，每一个元素都有相同的数据类型，在内存中占用相同大小的存储单元，且在内存中连续存放。每一个数组有一个名字，数组中的每一个元素有一个序号（或称下标）表示元素在数组中的位置，我们正是通过下标来识别数组中的每一个元素。

数组有一维的，也有多维的，数组也有大小，数组的维数和大小是在定义数组时就确定的，程序运行的时候不能改变。

数组适合于表示由许多相似项组成的组合数据，例如，人名表、世界城市和它们的温度表、银行每月的帐目处理等。

本章中，我们将介绍一维数组、二维数组和字符数组的定义、引用和它们的初始化。

数组主要有以下特点：

（1） 每个数组元素的数据类型相同，且可以是任何合法的数据类型；

（2） 数组可以是一维的、二维的，甚至更高维的，例如，下面的语句就定义了一个三维数组 ARRAY：

INT ARRAY[3][3][3];

（3） 数组同其它变量一样，在使用前必须定义；

（4） 数组各个元素按顺序排列，其位置由下标确定；

（5） 数组元素可以同其它变量一样使用。

注意：数组的维数和大小是在定义数组时就确定的，程序运行的时候不能改变。

数组就是一些数据组合在一起的一个有序序列。例如，某个年级中学生的分数，家庭中所有成员的年龄以及某个机构中雇员的姓名。这些例子都是将数据组合在一个特定的类别中（姓名、分数、年龄），组中的每个成员赋予相同或不同的值，例如这个班的学生的分数可能是 85、92、97、88，家庭成员的年龄可能是 35 岁、38 岁、6 岁和 8 岁，机构中的雇员的名字可能是玛丽、约翰、爱丽丝，所有这些都可以用数组来表示。

数组是同类型数据的有序集合。一个数组中的所有数据都来自同一类型，如一个正文行可以看成是一个由字符组成的字符数组，一个向量可以看成是由整数或浮点数组成的数组，一个矩阵可以看作是由向量组成的数组。

每一个数组有一个名字，数组中的每一个元素有一个序号（或称下标）表示元素在数组中的位置，我们正是通过下标来识别数组中的每一个元素。例如，对于上面的例子，我们可以用数组来表示班级的分数，并把这个数组取名为 MARKS。假设这个班只有 4 个人，则该数组包含 4 个元素，表示如下：

MARKS[0]＝85；

MARKS[1]＝92；

MARKS[2]＝95；

MARKS[3]＝88；

需要注意的是：方括号里的数字是数组的下标，即该元素在数组中的序号，数组下标是从 0 开始计数的。例如，元素 MARKS[2]在数组中的下标或序号为 2，但它是数组中的第三个元素。

上面的四个语句用来给数组的每个元素赋值，但在给数组元素赋值之前，一定要先定义该数组（记住：所有变量在使用之前都必须定义），下面是该数组的定义：

INT MARKS[4];

数字 4 表示这个数组中有 4 个元素，从第 0 号元素开始，到第 3 号元素结束。

我们继续下一个例子。我们定义一个家庭成员的年龄数组，该数组表示如下：

INT AGE[4]; //定义有 4 个元素的数组 AGE

AGE[0] = 35; //给每个数组元素赋值

AGE[1] = 38;

AGE[2] = 6;

AGE[3] = 8;

最后，我们看一下某机构的雇员姓名的数组，该数组被声明和赋值如下：

CHAR NAMES[3][6] = {"MARY", "JOHN", "ALICE"}; //定义数组 NAMES 并初始化

数组 NAME 是个二维数组，二维数组可以看成是元素是一维数组的一维数组。例如，NAME 数组可以认为是有三个元素的一维数组，而每个数组元素又是有 6 个元素的一维数组。

6.1.1 一维数组的定义

数组的定义与变量的定义类似，一维数组的一般定义形式为：

类型说明符 数组名[常量表达式]；

"类型说明符"指定数组元素的类型，"数组名"的命名规则与变量一样，方括号中的"常量表达式"的值表示数组元素的个数，它必须是一个整数。例如，北京最近一百年来每年的平均温度值可保存在一个数组中，该数组可定义为：

float annual_temp[100];

这个定义会使得编译器分配 100 个连续的 float 变量的内存空间。数组元素的个数在编译时就必须固定，且最好定义为一个常量。这样，当数组元素的个数需要改变时，只要改变那个常量即可：

const int NE = 100;

float annual_temp[NE];

数组元素的下标从零开始计数，在 annual_temp 数组中，第一个元素是 annual_temp[0]，第二个元素是 annual_temp[1]，余类推，最后一个元素是 annual_temp[NE-1]。

在实际中，我们经常要处理一组有关系的数据，比如一组学生的成绩，我们可以定义它们为整数，如：

int grade0, grade1, grade2 grade3, grade4;

当学生的成绩较多时，比如说 100 个，这样定义很不方便，而且各变量之间也没有制约关系，不能保证它们是一组相关联的变量。对于这种情况，用数组就比较方便：

int grade[100];

grade 数组有 100 个元素，可以用它来存放 100 个学生成绩，这样表示既清楚又方便。

例如：

int a[10];

float f[50];

char name[20];

都是正确形式的数组定义。各数组元素在内存中是连续存放的，例如：

int a[10];

定义了一个由 10 个元素组成的整型数组 a，它在内存中存放的形式如下所示：

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

其中每一个元素都相当一个整型变量，可以存放一个整型数值。与 10 个整型变量不同之处在于：数组元素是按顺序排列的，这在处理数据时会带来许多方便。

从定义中可以看出，一维数组名是一个标识符，其后方括号中的常量表示数组元素的个数。C++语言中数组元素的下标总是从 0 开始的，因此，数组 a 的元素下标范围为：0~9。同样，对于数组 f：

float f[50];

其下标范围为：0~49。。

象基本类型的变量一样，数组也必须先定义后使用。下面我们举一个例子：

例 6-1

```
#include <iostream.h>
void main(void)
{
   int a[5],sum; //定义数组 a 和变量 sum
   a[0] = 3;
   a[1] = 1;
   a[2] = 7;
   a[3] = 4;
   a[4] = 8;
   sum = a[0] + a[1] + a[2] + a[3] + a[4];
   cout<<"sum = "<<sum<<endl;
}
```

上述程序的运行结果为：

sum = 23;

这里我们看到了数组元素的使用，一个数组元素是由数组名和下标值来标识的，如 a[0]、a[1]等。数组元素的一般引用形式为：

数组名[下标表达式]

虽然数组定义和数组元素的引用形式类似，但它们含义不同：

int a[5];

是定义有 5 个元素的数组 a。而表达式：

a[4]

表示数组中下标为 4 的元素。

在前面的例子中，数组元素象普通变量一样使用。由于数组元素排列的规律性，实际中，我们常用循环的办法来操作。

下面的例子可以求一组学生的平均成绩：

例 6-2

```
#define num 5 //预定义 num 的值为 5
#include <stdio.h>
void main(void)
{
   int grade[num],i,total;
   float average;
   total = 0;
   printf("Please input %d score\n",num);
   for(i = 0;i <num;i ++)
   {
      printf("grade[%d] = ",i);
      scanf("%d",&grade[i]);
      total = total + grade[i];
   }
   average = (float)total/num;
   printf("average = %0.2f\n",average);
}
```

这一段程序的运行结果为：

```
please input 5 scores
grade[0] = 90
grade[1] = 65
grade[2] = 97
grade[3] = 57
grade[4] = 78
average = 77.40
```

可以看到：这个程序从键盘上接收 5 个成绩，求平均成绩，for 循环控制变量 i 从 0 变化到 4，printf 语句每次显示出要接收的下标变量 grade[0]～grade[4]，scanf 接收相应的值，再把它们加到 total 中。

要注意，在这里 scanf 不能一次接收整个数组的值，如写成：

```
scanf("%d",&grade);
```

是错误的，要想对数组中所有元素赋值，就要通过循环。

引入数组后，许多需要循环处理的问题就变得更方便解决了，下面再来看一下求菲波那契数列的问题，我们使用数组来解决。

例 6-3

```
#include <stdio.h>
void main(void)
{
   const int NUM = 20;
   const int COL = 5;
   int f[NUM + 1],i;
   f[1] = 0;
   f[2] = 1;
   for(i = 3; i<= NUM;i++)
   f[i] = f[i - 2] + f[i - 1];
   for(i = 1;i <= NUM;i++)
   {
      printf("%10d",f[i]);
      if(i % COL == 0)
         printf("\n");
   }
}
```

运行结果为：

0 1 1 2 3

5 8 13 21 34

55 89 144 233 377

610 987 1597 2584 4181

程序用 fn-2 和 fn-1 求 fn，写成

f[n] = f[n - 2] + f[n - 1];

显得非常自然，容易书写，也容易阅读理解。

程序中定义数组元素个数为 NUM + 1（21 个），使用时没有用 f[0]，而只用了 f[1]~f[20]，这样和数学公式表达相一致，处理也更方便。

有了数组，许多问题的处理变得容易起来，但 C++的数组也有一个令人很头疼的地方，就是它不作数组下标是否越界的判断，如上例中的第二个 for 语句写成：

for(i = 1;i <= NUM+1;i++)

循环体中的 f[i]在最后一次将是 f[21]，而 f 的下标范围为 0～20，f[21]已经不是数组 f 的元素了。

6.1.2 一维数组的使用

C++中，数组元素的下标总是从 0 开始，如果一个数组有 n 个元素，则第一个元素的下标是 0，最后一个元素的下标是 n-1。

访问数组元素的方法是通过数组名及数组名后的方括号中的下标。例如，假定我们要设置上面定义的数组 annual_temp 的第 15 个元素值为 11.5，则代码为：

annual_temp[14] = 11.5;

如果，企图访问一个不存在的数组元素，会导致一个严重的错误。例如：

annual_temp[100]= 12.3; //错误，该数组的最大下标为 99！

能使用标识符的地方，就能引用数组元素。下面是引用数组元素的一些例子，并假定在引用前已有如下的一些定义：

```
const int NE = 100,
N = 50;
int i, j, count[N];
float annual_temp[NE];
float sum, av1, av2;
```

能够用 cin 函数将一个值直接读到数个数组元素中：

```
cin >> count[i];
```

数组元素能够进行算术运算：

```
count[i] = count[i] + 5;
count[i] += 5;
```

数组元素可以用在逻辑表达式中：

```
if (annual_temp[j] < 10.0)
cout << "It was cold this year "
<< endl;
```

处理数组元素时，通常使用循环语句。例如 annual_temp 的数组元素的值可用下面的语句获得：

```
for (i = 0; i < NE; i++)
cin >> annual_temp[i];
```

下面的代码可以算出数组前 10 个元素均值：

```
sum = 0.0;
for (i = 0; i <10; i++)
sum += annual_temp[i];
av1 = sum / 10;
```

假定 k 是一个整型变量或常量，但 k<=NE。假定要计算数组中最后 k 个元素的平均值，代码如下：

```
sum = 0.0;
for (i = NE - k; i < NE; i++)
sum += annual_temp[i];
av2 = sum / k;
```

注意：C++不检查引用一个数组元素时的下标值是否在数组要求的下标值范围内。例如，给 annual_temp[200]赋值并不会有语法错误，但是，程序运行的结果是难以预料的。它可能导致将数值赋给一个内存单元，但在程序分配的内存空间之外；它也可能导致程序的运行被操作系统终止；它也可能导致将数值赋给一个内存单元，该内存单元在程序分配的内存空间之内，但改变了在该内存单元的其它变量的值或覆盖了在该内存单元的程序代码。同样，读一个值从 annual_temp[200]也会产生类似的问题。引用数组元素时，下标超出了允许的范围，称之为下标溢出。下标溢出是一个很容易犯的错误，它常常使程序出现一些很奇怪和难以查找的错误，我们在编程时要十分小心。

数组在内存中如何存储

对于数组定义"int marks[4];"，编译器在内存中为 marks 保留 4 个整数内存空间，假定存放一个整数要占用两个字节，存放 marks 数组内存的起始地址为 5000，则 marks 四个元素在内存中的状态如下图：

表 6-1

内存地址	内容
5000	85
5002	92
5004	95
5006	88

假定 marks 是一个 float 型数组，每个 float 型数在内存中占 4 个字节，则 marks 四个元素在内存中的状态如下图：

表 6-2

内存地址	内容
5000	85.0
5004	92.0
5008	95.0
5012	88.0

对于数组定义"char marks[5] = "Mary";"，假定每个字符占一个内存字节，存放 marks 数组内存的起始地址为 5000，则 marks 在内存中的状态如下图：

表 6-3

内存地址	内容
5000	'M'
5001	'a'
5002	'r'
5003	'y'
5004	'\0'

前面我们已经定义了名为 name 的二维数组：

```
char names[3][6] = {"Mary", "John", "Alice"};
```

假定每个字符占一个内存字节，存放 names 数组内存的起始地址为 5000，则 names 在内存中的状态如下图：

表 6-4

5000	5001	5002	5003	5004	5005
'M'	'a'	'r'	'y'	'\0'	
5006	5007	5008	5009	5010	5011
'J'	'o'	'h'	'n'	'\0'	
5012	5013	5014	5015	5016	5017
'A'	'l'	'i'	'c'	'e'	'\0

注意：内存地址 5005 和 5011 的内容是空的，这是因为 name[]的第一个元素和第二个元素都只有 4 个字符长，因此只占用 5 个连续的内存地址空间。也要注意：每个字符串以一个空字符（'\0'）结束。

一维数组应用举例：

1． 筛法求素数（只能被 1 和自身整除的正整数是素数）。

用筛法求素数的基本思想是：把从 1 开始的、某一范围内的正整数从小到大顺序排列， 1 不是素数，首先把它筛掉。剩下的数中选择最小的数是素数，然后去掉它的倍数。依次类推，直到筛子为空时结束。如有：

1 2 3 4 5 6 7 8 9 10

11 12 13 14 15 16 17 18 19 20

21 22 23 24 25 26 27 28 29 30

1 不是素数，去掉。剩下的数中 2 最小，是素数，去掉 2 的倍数，余下的数是：

3 5 7 9 11 13 15 17 19 21 23 25 27 29

剩下的数中 3 最小，是素数，去掉 3 的倍数，如此下去直到所有的数都被筛完，求出的素数为：

2 3 5 7 11 13 17 19 23 29

写程序时，采用一个数组，用数组的下标表示自然数，如果一个数不在筛中就将其对应的元素值赋 0，如果仍在筛中，则那个元素值为 1，程序如下：

例 6-4

```
#define RANGE 200
#include <stdio.h>
void main(void)
{
  int sieve[RANGE + 1];
  int i,j,count;
  for(i = 0;i <= RANGE;i ++)
    sieve[i] = 1;//初始化
    sieve[0] = sieve[1] = 0;//0 和 1 不是素数
    count = 0;
  for(i = 2;i <= RANGE;i ++)
    if(sieve[i] == 1)//i 是素数
    {
```

```
            printf("%5d",i);//输出素数
            count ++;
            if(count % 8 == 0)//每行输出 8 个值
               printf("\n");
            for(j = i;j <= RANGE;j += i)
               sieve[j] = 0;//筛去 i 的倍数
         }
      printf("\n");
   }
```

运行结果：

2 3 5 7 11 13 17 19

23 29 31 37 41 43 47 53

59 61 67 71 73 79 83 89

97 101 103 107 109 113 127 131

137 139 149 151 157 163 167 173

179 181 191 193 197 199

2．求杨辉三角形。

所谓杨辉三角形就是二次项的系数：

1

1 2

1 2 1

1 3 3 1

1 4 6 4 1

……

我们编一个程序来求解它们，并以上面的形式输出。杨辉三角形的头一行是容易生成的，只需简单地赋值就可以了。那么对于任意一行呢？我们可以看到，第一列和最后一列总是 1，而其它数是上面一行中本列和前一列元素之和，如第 5 行的生成过程是：

4 = 3 + 1

6 = 3 + 3

4 = 1 + 3

我们可以利用两个数组，用存放在前一个数组中的数据生成新的一行放在另一个数组中，通过来回交换就可生成一行行的数据。我们也可以只用一个数组，每次倒着生成（即从右向左生成）数组中的各元素，如

yanghui[4] = yanghui[4] + yanghui[3];

yanghui[3] = yanghui[3] + yanghui[2];

yanghui[2] = yanghui[2] + yanghui[1];

本程序采用后一种方法，程序如下：

例 6-5

```
#define LASTROW 10
#include <stdio.h>
void main(void)
{
   int yanghui[LASTROW + 1],row,col;
   yanghui[0] = 1;
   printf("%4d\n",yanghui[0]);//输出第一行
   //由前一行生成新的一行
   for(row = 1;row <= LASTROW;row ++)
   {
      yanghui[row] = 1;
      for(col = row - 1;col > 0;col --)
         yanghui[col] = yanghui[col] + yanghui[col - 1];
      for(col = 0;col <= row;col ++)
         printf("%4d",yanghui[col]);//输出一行
      printf("\n");
   }
}
```

运行结果：

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

6.1.3 一维数组的初始化

数组也能够在定义时就给出数组元素的初值，称之为初始化，数组的初始化与简单变量的初始化类似。初值放在一对花括号中，各初值之间用逗号隔开。例如：

```
int primes[] = {1, 2, 3, 5, 7, 11, 13};
```

在上面的定义中，我们没有显式说明数组元素的个数，编译器会根据初值的个数和类型，分配足够的内存空间。本例中，primes 将分配 7 个整型元素的内存空间。再看下面的例子：

```
int primes[10] = {1, 2, 3, 5, 7};
```

数组元素的个数有 10 个，但初值只有 5 个。此时，只有前 5 个数组元素赋给了初值。

维数组的初始化程序举例：

例 6-6

```
#include <iostream.h>
void main(void)
{
   int marks[4] = {82,92,95,88};
   cout<<"Second element is:"<<marks[1]<<endl;
}
```

该程序的输出是：

Second element is 92

注意：

<> 这个数组在定义时就被初始化了；

<> 用花括号将初始化的值括起来；

<> 用 marks[1]引用数组的第二个元素，这是因为数组下标是从 0 开始的。

初始化时也可只对部分元素赋值，例如：

```
int a[10] = {9,8,7,6,5};
```

把值 9、8、7、6、5 赋给 a 的前 5 个元素，后 5 个元素系统自动赋值为 0，情况如下所示：

如果对数组的全部元素都初始化，则可以不指定数组长度，例如：

```
int a[10] = {9,8,7,6,5,4,3,2,1,0};
```

也可以写成：

```
int a[] = {9,8,7,6,5,4,3,2,1,0};
```

这时由初值的个数决定数组的长度。

6.2.1 二维数组的定义

二维数组相当于一个矩阵，它的定义方法与一维数组的定义类似。二维数组定义的一般形式为：

类型说明符 数组名[常量表达式 1][常量表达式 2]；

"类型说明符"指定数组元素的类型，"数组名"是 C++允许的标识符，"常量表达式 1"指定数组元素的行数，"常量表达式 2" 指定数组元素的列数。例如，我们要表示北京、天津、上海三个城市一年四季的平均气温，参见表 6-5。

可以定义一个二维数组来保存气温值：

```
float seasonTemp[3][4];
```

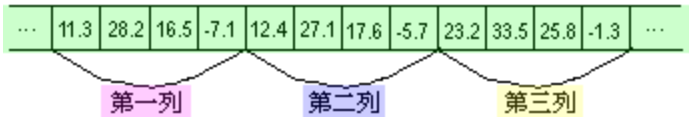
这个数组在内存中占用 12 个连续的 float 元素的存储单元，如图 6-2。

我们可以看到：C++中，二维数组在内存中的存放仍然是一维的，且各个元素按行顺序存放。

表 6-5 三个城市各季节的平均气温

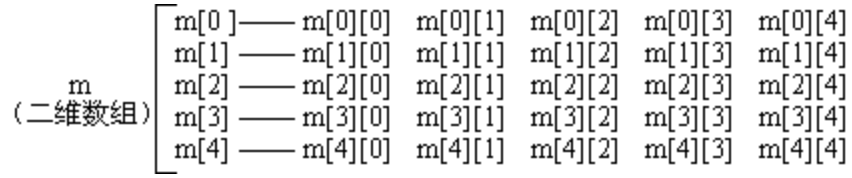
	春	夏	秋	冬
北京	11.3	28.2	16.5	-7.1
天津	12.4	27.1	17.6	-5.7
上海	23.2	33.5	25.8	-1.3

图 6-2 气温值在内存中的存储



C++语言把二维数组看成是一个一维数组，该一维数组的各个元素又是一个一维数组。二维数组 m 可看成是有 5 个元素的一维数组，而每一个元素又是有 5 个元素的一维数组。如下图 6-3 所示：

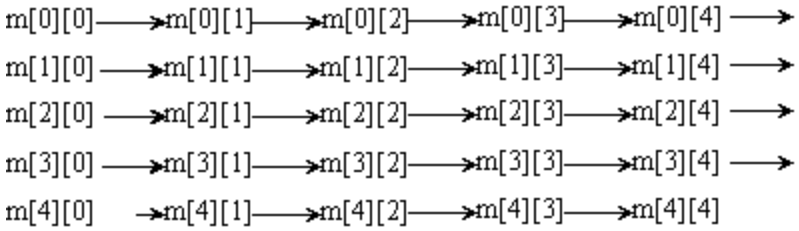
图 6-3



由这种形式我们可以了解更高维的数组。一个三维数组实际上可以看作是元素为二维数组的数组，依此类推。

我们可以看到：在 C++语言中，二维数组在内存中的存放仍然是一维的，而且各个元素是按行顺序存放的，即在内存中是先放第一行的元素，再放第二行的元素，…，下图给出了二维数组 m 的排列顺序：

图 6-4



6.2.2 二维数组的初始化

二维数组的初始化是分行进行的。

例如，我们对上面定义的二维数组 seasonTemp 可用下面的语句初始化：

```
float seasonTemp[3][4] = {
   {11.3, 28.2, 16.5, -7.1},
   {12.4, 27.1, 17.6, -5.7},
   {23.2, 33.5, 25.8, -1.3}
};
```

我们已经知道,二维数组在内存中是一维的，且由各行元素顺序存放得到。所以，seasonTemp 也可以采用下面的语句初始化：

```
int seasonTemp[3][4] = {
   11.3, 28.2, 16.5, -7.1, 12.4, 27.1, 17.6, -5.7, 23.2, 33.5, 25.8, -1.3
```

};

但是，按行初始化更直观、更通用，它能够初始化某一行中的部分元素。例如：

```
int seasonTemp[3][4] = {{26}, {24}, {28}};
```

上面的语句仅初始化数组每一行的第一个元素，而其它元素未给初值。

我们定义并初始化一个二维数组时，可以省略"常量表达式 1"，例如：

```
int seasonTemp[][4] = {
   {26, 34, 22, 17},
   {24, 32, 19, 13},
   {28, 38, 25, 20}
};
```

如果省略"常量表达式 1"最好采用按行初始化的方式，否则，容易出错。编译器根据初值的行数来确定数组的行数。

注意："常量表达式 2"一定不能省略，否则程序编译时会出错。因为二维数组元素在存取时，需要在一维和二维之间转换，如果省略"常量表达式 2"，则编译器无法计算出它们之间的对应关系。

6.2.3 二维数组的使用

类似于一维数组的访问，访问二维数组元素可通过两个[]运算符，第一个[]运算符指定元素的行下标，第二个[]运算符指定元素的列下标。二维数组的行下标和列下标均是从 0 开始的。如果有一个有 m 行、n 列的二维数组，它的第一行第一列元素的行和列下标分别是 0 和 0，最后一行最后一列元素的行和列下标分别是 m-1 和 n-1。

处理二维数组也类似于一维数组，通常用一个二重循环语句。例如下面的程序就是用一个二重循环语句在数组 seasonTemp 中查找最高温度：

例 6-7

```
const int rows = 3;
const int columns =4;
int seasonTemp[rows][columns] = {
   {26, 34, 22, 17},
   {24, 32, 19, 13},
   {28, 38, 25, 20}
};
int HighestTemp (int temp[rows][columns])
{
   int highest = 0;
   for (register i = 0; i < rows; ++i)
   for (register j = 0; j < columns; ++j)
         if (temp[i][j] > highest)
            highest =temp[i][j];
return highest;
}
```

二维数组应用举例：

1．矩阵的转置。

矩阵的转置就是把矩阵的行和列互换，比如一个 3*4 的矩阵

1 2 3 4

5 6 7 8

9 10 11 12

转置后变成 4*3 的矩阵

1 5 9

2 6 10

3 7 11

4 8 12

相应的程序可以写为：

例 6-8

```
#include <iostream.h>
#define ROW 3
#define COL 4
void main(void)
{
  int a[ROW][COL],b[COL][ROW];
  int i,j;
  cout<<"Please input element of the matrix a（ ";
  cout<<ROW<<"×"<<COL<<")";
  for(i = 0;i < ROW;i ++) //输入矩阵 a
       for(j = 0;j < COL;j ++)
       cin>>a[i][j];
  for(i = 0;i < ROW;i ++) //转置
    for(j = 0;j < COL;j ++)
  b[j][i] = a[i][j];
  cout<<"Matrix b:"<<endl;
  for(i = 0;i < COL;i ++) //输出矩阵 b
  {
  for(j = 0;j < ROW;j ++)
    cout<<b[i][j]<<" ";
  cout<<endl;
  }
}
```

运行结果：

Please input elements of the matrix a(3×4)

1 2 3 4

5 6 7 8

9 10 11 12

Matrix b:

1 5 9

2 6 10

3 7 11

4 8 12

在上面的程序中，由于矩阵的行列不同，必须使用两个数组，对于 n×n 矩阵，我们可以只用一个数组，在数组中进行行列元素的互换。程序如下：

例 6-9

```
#define ROW 4
#include <stdio.h>
void main(void)
{
   int sm[ROW][ROW],i,j,temp;
   printf("Input elements of a matrix (%d×%d):\n",ROW,ROW);
   for(i = 0;i < ROW;i ++)
      for(j = 0;j < ROW;j ++)
         scanf("%d",&sm[i][j]);
   for(i = 0;i < ROW - 1;i ++) //转置
      for(j = i + 1;j < ROW;j ++)
         {
            temp = sm[i][j];
            sm[i][j] = sm[j][i];
            sm[j][i] = temp;
         }
   printf("The matrix has been transposed:\n");
   for(i = 0;i < ROW;i ++)
   {
      for(j = 0;j < ROW;j ++)
         printf("%5d",sm[i][j]);
      printf("\n");
   }
}
```

运行结果：

Input elements of a matrix (4×4):

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

The matrix has been transposed:

1 5 9 13

2 6 10 14

3 7 11 15

4 8 12 16

例 6-10

```
#define MAX 15
#include <stdio.h>
void main(void)
{
   int m,mm,i,j,k,ni,nj;
   int magic[MAX][MAX];
   printf("Enter the number you wanted\n");
   scanf("%d",&m);
   for(i = 0;i < m;i ++) //初始化
      for(j = 0;j < m;j ++)
         magic[i][j]  = 0;
   if((m > 0)&&(m % 2 != 0))//奇数阶
   {
      mm = m * m;
      i = 0;//第一个值的位置
      j = m / 2;
      for(k = 1;k <= mm;k ++)
      {
         magic[i][j]  = k;
         //求右上方方格的坐标
         if(i == 0)//最上一行
            ni = m - 1;//下一个位置在最下一行
         else
            ni = i - 1;
         if(j == m - 1)//最右端
            nj = 0;//下一个位置在最左端
         else
            nj = j + 1;
         //判断右上方方格是否已有数
         if(magic[ni][nj]  == 0)//右上方无值
         {
            i = ni;
            j = nj;
         }
         else //右上方方格已填上数
```

```
            i ++;
        }
        for(i = 0;i < m;i ++)
        {
            for(j = 0;j < m;j ++)
                printf("%4d",magic[i][j]);
                printf("\n");
        }
    }
else //m<=0 或 m_是偶数
printf("Error in input data.\n");
}
```

以输入值 5 为例，程序的运行结果为：

```
Enter the number you wanted
5
17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
11 18 25 2 9
```

当数组中的元素都是由一个个字符组成时，便称之为字符数组。字符数组并无什么特别之处，它的的重要性在于： C++中，用一个一维的字符数组表示字符串。数组的每一个元素保存字符串的一个字符，并附加一个空字符，表示为`\0'，添加在字符串的末尾，以识别字符的结束。所以，如果一个字符串有 n 个字符，则至少需要有 n+1 个元素的字符数组来保存它。例如，一个字符`a'仅需要一个字符变量就可以保存，而字符串"a"需要有两个元素的字符数组来保存，一个元素存字符`a'，另一个元素存空字符`\0'。

6.3.1 字符数组的输出

字符串可以用 printf 或 cout 函数输出，例如：

```
cout << "The string s1 is " << s1 << endl;
printf("The string s1 is %s\n", s1) ;
```

上面两个语句的输出结果相同，均为：

```
The string s1 is example
```

面是一个字符数组的定义实例：

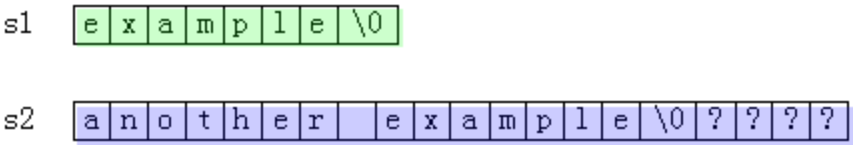
```
char s1[10];
```

数组 s1 能够保存字符数最多为 9 个的字符串。象其它数组的初始化一样，字符数组也能在定义时初始化。不过字符数组并不需要一个字符一个字符地赋初值，可用整个字符串常量来使字符数组初始化。例如：

```
char s1[] = "example";
char s2[20] = "another example";
```

上面两个语句保存下面两个字符串：

图 6-6



在第一种情形下，数组分配 8 个字符的空间，分别保存字符串的 7 个字符和一个结尾符`\0'。第二种情形申请了 20 个字符的空间，但仅 16 个字符位置被占用，即 15 个字符串字符和一个结尾符`\0'。

注意：字符串的长度并不包括结尾符。

我们定义字符数组时，通常采用第一种形式，即不显式指定字符串的长度，而由编译器自行确定字符数组的长度。

6.3.2 字符数组的输入

字符串（字符数组）通常用 scanf 和 cin 函数输入，下面分别介绍。

一、用 scanf 函数输入字符串

scanf 可以一次输入一个或多个字符串，如果一次输入多个字符串，字符串之间用逗号隔开，用"ENTER"结束输入。

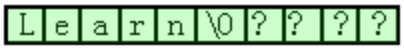
例如，假定我们已经定义：

```
char s1[10], s2[10];
```

执行语句：

```
scanf("%s", s1);
```

输入字符串"Learn C++"，数组 s1 中的元素如下图：

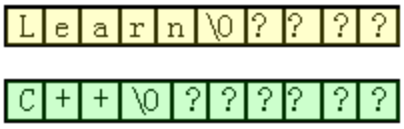


即将空格前的字符送入数组 s1 中，忽略了空格后的字符。

如果执行语句：

```
scanf("%s%s", s1, s2);
```

输入字符串"Learn C++"，数组 s1 和 s2 中的元素如下图：



这里要注意用 scanf 函数输入字符与输入字符串的不同。例如：

```
char str1,str2;
char s1[10],s2[10];
scanf ("%c%c",&str1,&str2);
scanf ("%s%s",s1,s2);
```

二、用 cin 输入字符串

用 cin 输入字符串的作用与 scanf 类似，但更方便。例如，假定 s1 定义还与前面一样，执行下面的语句：

```
cin >> s1;
```

输入输入字符串"Learn C++"，s1 的值与用 scanf 的结果是一样的。如果一个字符串中有空格，我们就需要在空格的地方将字符串分成二个或多个字符串输入。例如，某人名字为"Ian Aitchison"，我们可以用下面的语句输入：

```
char firstname [12], surname[12];
cout << "Enter name ";
cin >> firstname;
cin >> surname;
cout << "The name entered was "
<< firstname << " "
<< surname;
```

执行上面的语句，输入"Ian Aitchison"，输出结果如下：

```
The name entered was Ian Aitchison
```

上面的程序段我们也可以修改为，效果是一样的：

```
char firstname [12], surname[12];
cout << "Enter name ";
cin >> firstname >> surname;
cout << "The name entered was "
<< firstname << " "
<< surname;
```

字符数组应用举例：

1. 求字符串的长度
2. 从键盘上读一个输入行并输出
3. 把两个字符串连接起来
4. 把一个数字字符串转换为相应的整数

字符数组应用举例：

1. 求字符串的长度

例 6-11

```
#define MAXLEN 80
#include <stdio.h>
void main(void)
{
   char str[MAXLEN + 1];
   int l;
   printf("Input a string:");
   scanf("%s",str);
   l = 0;
   while(str[l] != '\0')
   l ++;
   printf("The length of this string is %d\n",l);
}
```

运行结果：

Input a string:world

The length of this string is 5

本程序的 scanf 输入使用了%s 格式，它用来输入一个字符串，输入项这时用数组名，并且前面不带&，如果写成：

scanf("%s",&str);

反而错了。C++编译器对数组名的处理是：把它作为存放数组的内存的起始地址。由于数组名本身是地址，因此也就不需要用求地址运算符&了。

scanf 接收一个字符串并自动为它加上一个结束符标识'\0'，因此我们可以利用'\0'来判断字符串的长度。

2. 从键盘上读一个输入行并输出

在 scanf 中使用%s 或利用 cin 可以读入一个字符串，当遇到空格符时就认为字符串结束了，因此不能用它们来读入一个输入行，因为一行中可能有空格。我们的程序要逐个检查输入的字符，只有遇到换行符'\n'时才停止读入，程序如下：

例 6-12

```
#define MAXLEN 80
#include <stdio.h>
void main(void)
{
   char line[MAXLEN + 1],c;
   int i;
   i = 0;
   while((c = getchar()) != '\n')
   line[i ++] = c;
   line[i] = '\0';
   printf("%s\n",line);
}
```

运行该程序，输入"How are you"，则屏幕打印：

How are you

这个程序用 getchar 逐个读入字符，当不是换行符时，就把读入的字符送到字符数组 line 中。i++使字符数组每得到一个字符后，就将下标后移一位，准备存放下一个字符，当遇到'\n'时，不把'\n'送进数组 line，而是在最后加一个字符串结束标志'\0'，构成一个字符串。printf 使用'%s'格式输出，它们执行是这样的：按字符数组名 line 找到该数组的起始地址，然后逐个输出数组中的字符，直至遇到'\n'为止。

3. 把两个字符串连接起来

例 6-13

```
#define LENGTH 40
#include <iostream.h>
void main(void)
```

```
{
    char str1[LENGTH + 1],str2[LENGTH + 1];
    char result[2 * LENGTH + 1];
    int len1,len2;
    cout<<"Input the first string:"<<endl;
    cin>>str1;
    cout<<"Input the second string."<<endl;
    cin>>str2;
    len1 = 0;
    while(str1[len1] != '\0')
    {
      result[len1] = str1[len1];
      len1 ++;
    }
    len2 = 0;
    while(str2[len2] != '\0')
    {
      result[len1] = str2[len2];
      len1 ++;
      len2 ++;
    }
    result[len1] = '\0';
    cout<<result<<endl;
}
```

运行该程序并输入：

Input the first string:

Good↙

Input the second string:

bye↙

运行结果为：

Goodbye

程序中第一个循环把 str1 的内容送到 result 中，但没有送'\0'，从第一个字符串的末尾位置开始，第二个循环把 str2 送到 result 中，同样没有送'\0'，因此在最后我们为新的字符串加一个'\0'表示字符串的结束，最后用 printf 输出这个字符串。

4. 把一个数字字符串转换为相应的整数

我们可以把由数字组成的字符串转换成对应的整数值，一个数字字符转为整数可用它的 ASCⅡ码值减去字符 0 的 ASCⅡ码值求得，如：

I = c - '0';

程序如下：

例 6-14

```
#include <iostream.h>
#include <stdio.h>
void main(void)
{
   char s[10];
   int i,n,sign;
   cout<<"Input a numberic string"<<endl;
   cin>>s;
   i = 0;
   sign = 1;
   if(s[i] == '+' || s[i] == '-')//符号
      sign = (s[i ++] == '+')?1:-1;
   for(n = 0;s[i] >= '0' && s[i] <= '9';i ++)
      n = n * 10 + s[i] - '0';
   n = sign * n;
   cout<<n<<endl;
}
```

运行该程序并输入：

Input a numberic string

123↙

运行结果为：

123

本程序允许数字串中带有正负号，语句

sign = (s[i ++] == '+')?1:-1;

用到条件运算符，它相当于

sign = (s[i ++] == '+')

sign = 1;

else

sign = -1;

6.3.3 字符串处理函数

C++中有一些操作字符串的标准的库函数，头文件 string.h 包含所有字符串处理函数的说明。常用的一些函数如下：

```
strcpy(char destination[], const char source[]);
strncpy(char destination[], const char source[], int numchars);
strcat(char target[], const char source[]);
strncat(char target[], const char source[], int numchars);
int strcmp(const char firststring[], const char secondstring);
strlen( const char string[] );
```

它们的功能分别为：

strcpy：将字符串 source 拷贝到字符串 destination 中。

strcpy 函数应用举例

原型：strcpy(char destination[], const char source[]);

功能：将字符串 source 拷贝到字符串 destination 中

例程：

例 6-15

```
#include <iostream.h>
#include <string.h>
void main(void)
{
  char str1[10]={"Tsinghua "};
  char str2[10]={"Computer"};
  cout<<strcpy(str1,str2)<<endl;
}
```

运行结果：

Computer

注意：在定义数组时，字符数组 1 的字符串长度必须大于或等于字符串 2 的字符串长度。不能用赋值语句将一个字符串常量或字符数组直接赋给一个字符数组。所有字符串处理函数都包含在头文件 string.h 中。

strncpy：将字符串 source 中前 numchars 个字符拷贝到字符串 destination 中。

strncpy 函数应用举例

原型：strncpy(char destination[], const char source[], int numchars);

功能：将字符串 source 中前 numchars 个字符拷贝到字符串 destination 中

例程：

例 6-16

```
#include <iostream.h>
#include <string.h>
void main(void)
{
  char str1[10]={"Tsinghua "};
  char str2[10]={"Computer"};
  cout<<strncpy(str1,str2,3)<<endl;
}
```

运行结果：

Comnghua

strcat:将字符串 source 接到字符串 target 的后面。

strcat 函数应用举例

原型：strcat(char target[], const char source[]);

功能：将字符串 source 接到字符串 target 的后面

例程：

例 6-17

```
#include <iostream.h>
#include <string.h>
void main(void)
{
  char str1[]={"Tsinghua "};
  char str2[]={"Computer"};
  cout<<strcpy(str1,str2)<<endl;
}
```

运行结果：

Tsinghua Computer

注意：在定义字符数组 1 的长度时应该考虑字符数组 2 的长度，因为连接后新字符串的长度为两个字符串长度之和。进行字符串连接后，字符串 1 的结尾符将自动被去掉，在结尾串末尾保留新字符串后面一个结尾符。

strncat:将字符串 source 的前 numchars 个字符接到字符串 target 的后面。

strncat 函数应用举例：

原型：strncat(char target[], const char source[], int numchars);

功能：将字符串 source 的前 numchars 个字符接到字符串 target 的后面

例程：

例 6-18

```
#include <iostream.h>
#include <string.h>
void main(void)
{
  char str1[]={"Tsinghua "};
  char str2[]={"Computer"};
  cout<<strcpy(str1,str2)<<endl;
}
```

运行结果：

Tsinghua Com

strcmp: 比较两个字符串 firststring 和 secondstring。

strcmp 函数应用举例

原型：int strcmp(const char firststring[], const char secondstring);

功能：比较两个字符串 firststring 和 secondstring

例程：

例 6-19

```
#include <iostream.h>
#include <string.h>
void main(void)
{
   char buf1[] = "aaa";
   char buf2[] = "bbb";
   char buf3[] = "ccc";
   int ptr;
   ptr = strcmp(buf2,buf1);
   if(ptr > 0)
      cout<<"Buffer 2 is greater than buffer 1"<<endl;
   else
      cout<<"Buffer 2 is less than buffer 1"<<endl;
   ptr = strcmp(buf2,buf3);
   if(ptr > 0)
      cout<<"Buffer 2 is greater than buffer 3"<<endl;
   else
      cout<<"Buffer 2 is less than buffer 3"<<endl;
}
```

运行结果：

```
Buffer 2 is less than buffer 1
Buffer 2 is greater than buffer 3
```

strlen：统计字符串 string 中字符的个数。

strlen 函数应用举例

原型：strlen(const char string[]);

功能：统计字符串 string 中字符的个数

例程：

例 6-20

```
#include <iostream.h>
#include <string.h>
void main(void)
{
   char str[] = {"Tsinghua Computer"};
   cout<<"The length of the string is "<<strlen(str)<<endl;
```

```
}
```

运行结果：

```
The length of the string is 17
```

注意：strlen 函数的功能是计算字符串的实际长度，不包括'\0'在内。另外，strlen 函数也可以直接测试字符串常量的长度，如：strlen("Welcome")。

同其它变量一样，数组也可以作函数的参数。数组元素只能作函数实参，且同其它变量的用法没有区别。例如 6-20：

我们已经知道，数组元素要通过数组名和相应的下标一个个地引用，而数组名可以作函数的实参和形参。当数组作为函数参数时，调用函数中的实参数组只是传送该数组在内存中的首地址，即调用函数通知被调函数在内存中的什么地方找到该数组。在前面我们已经知道了函数参数的值传递方式，调用函数向被调函数传递数据地址的方式，称之为函数参数的引用传递。

对于函数参数的引用传递，除传送数组名外，调用函数还必须通知被调函数：数组有多少个元素。所以，有数组参数的函数原型的一般形式为：

类型说明符 函数名(数组参数, 数组元素个数)

数组参数并不指定数组元素的个数，即[]中是空的，没有数字。例如 6-21：

我们已经知道：函数参数的引用传递不同于值传递。值传递时是把实参的值复制到形参，实参和形参占用不同的存储单元，形参值的改变不会影响到实参。而数组作为函数参数传递时，是引用传递方式，即把实参数组在内存中的首地址传给了形参，被调函数可以通过该地址，找到实参数组中的各个元素。这就意味着：在被调函数中，如果改变了形参数组某元素的值，在被调函数中，实参数组对应元素的值也会发生相应的改变。

例 6-21

```
#include <iostream.h>
int add(int x, int n)
{
   return (x + n);
}
void main()
{
   int a[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
   int i;
   for( i = 0; i<10; i++)
      a[i]=add(a[i], i); //数组元素作函数参数
   for(i = 0; i<10; i++)
      cout << a[i] << endl;
}
```

例 6-22

```
double mean(double data_array[], int numelements)
```

```
{
    double sum = 0.0;
    for(int i=0; i<numelements; i++)
        sum += data_array[i];
    return sum / numelements;
}
```

下面我们看看应该怎样调用上面的 mean 函数：

例 6-23

```
int main()
{
    double heights[100];
    double h;
    int n = 0;
    cout << "Enter heights:" << endl;
    cin >> h;
    while(h > 0.0) {
        heights[n] = h;
        n++;
        cin >> h;
    }
    double m = mean(heights, n);
    cout << "The mean is " << m << endl;
    return 0;
}
```

传值调用与传址调用的区别在于，使用传值调用时，函数接收的是函数值的一份拷贝；另一方面，使用传址调用时，函数接收的是变量的内存地址。因此，函数能改变存放于指定内存单元（也就是变量值）的值，所作的改变在函数结束后继续保留下来。

函数参数的引用传递是有用的。例如当数组作函数参数时，仅仅传送数组在内存中的首地址，避免了复制每一个数组元素，可以节省机器的内存和运行时间。另外，由于函数中 return 语句只能带回一个返回值，如果被调函数中有多个返回值，我们常常通过数组带回。但对引用传递应该注意：被调函数中对形参数据的不恰当的改变，会破坏调用函数中的实参数据。

1. 顺序查找
2. 把一个字符串反序后输出

数组作为函数参数举例：

1. 顺序查找

查找是计算机中经常要遇到的一种操作，其含义是在一组数据中查找到待查数据的位置。当一组数据无序时，一般采用顺序查找。顺序查找是把给定的值与这组数据中的每个值顺序比较，如果找到，就输出这个值的位置，如果找不到则报告没有找到。下面是顺序查找的程序：

例 6-24

```
#include <stdio.h>
#define SIZE 10
//求给定值 key 的位置，找到则返回其下标，找不到返回-1
int seq_search(int v[],int n,int key) //v[]：数组参数  n：数组元素个数  key：待查的值
{
   int i;
   for(i = 0;i < n;i ++)
      if(key == v[i])
         return i;
   return -1; //没找到
}

void main(void)
{
   int d[SIZE],KEY,i,index;
   printf("Input %d numbers:\n",SIZE);
   for(i = 0;i < SIZE;i ++)
      scanf("%d",&d[i]);
   printf("Input a key you want to search:\n");
   scanf("%d",&KEY);
   index = seq_search(d,SIZE,KEY);
   if(index >= 0)
      printf("The index of the key is %d.\n",index);
   else
      printf("Not found.\n");
}
```

运行该程序并输入：

Input 10 numbers:

1 2 3 4 5 6 7 8 9 0↙

Input a key you want to search:

6↙

输出结果为:

The index of the key is 5

本程序中使用了一个数组作为参数，函数 seq_search 中形参 v 没有指定数组的长度，要处理的元素的个数由其后所跟的参数 n 决定。

2. 把一个字符串反序后输出

例 6-25

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

```
#include <string.h>
#define LENGTH 80

//反序一个字符串
void reverse(char s[])
{
   char c;
   int i,j;
   j = strlen(s) - 1;
   for(i = 0;i < j;i ++)
   {
      c = s[i];
      s[i] = s[j];
      s[j] = c;
      j --;
   }
}

void main(void)
{
   char str[LENGTH + 1];
   cout<<"Input a string:";
   cin>>str;
   reverse(str);
   cout<<"The string is reversed:"<<str<<endl;
}
```

运行该程序并输入：

Input a string:abcd↙

运行结果为：

The string is reversed:dbca

这个程序中，函数 reverse 没有指定数组元素个数的参数，这是因为字符串总是以一个'\0'结束，通过'\0'就可以得到字符数组中元素的个数，也就是字符串的长度，这里我们可以看到使用字符串结束标识的好处。

二维数组也可作为函数参数，第一维的大小可以省略，而第二维的大小不能省略。下面是一个二维数组作为函数参数的例子：

例 6-26

```
#include <stdio.h>
void main()
{
   void add(int x[][4], int n, int num);
```

```
    void print(int x[][4], int n);
    int a[3][4]={{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    int number=10;
    add(a, 3, number);
    print(a, 3);
}
void add(int x[][4], int n, int num)
{
    for(int i = 0; i<n; i++)
    for(int j = 0; j<4; j++)
    x[i][j] = x[i][j]+num;
}
void print(int x[][4], int n)
{
    for(int i = 0; i<n; i++)
    for(int j = 0; j<4; j++)
    printf("%4d\n", x[i][j]);
}
```

【本章小结】

本章主要介绍了数组与字符串的基本知识。 数组是同类型变量组成的集合，访问数组中特定的元素通过下标运算符。数组由连续存储单元组成，其起始地址对应于数组的第一个元素。一维数组和二维数组都很常用，一定要熟练掌握。

C++中没有字符串类型，字符串是用一维字符数组来存储的。它与其它一维数组不同之处在于：除存放字符串中的各个字符外，存放字符串最后一个字符数组元素的下一个数组元素是字符"\0"，它是字符串结尾的标记。另外，还介绍了 C++语言提供的几个常用的字符串处理函数，供学习者参考使用。

数组越界是一个严重的错误，使用数组时一定不能越界，最好在使用时，对数组是否越界进行检查。

最后，我们还介绍了数组作为函数参数时的用法。数组名本身是地址，使得数组作为函数参数来传递从空间利用上显得合理。当数组传给函数时，处理的是数组的实际的值，这一点在使用的时要格外注意。

【学习目标】

◇ 掌握指针和引用的的基本概念。包括指针和引用的定义方法与初始化等。

◇ 掌握内存的动态申请与释放的方法。

◇ 掌握指针和引用作为函数的参数的用法。

◇ 搞清楚引用传递与值传递的区别。

◇ 搞清楚指针与数组下标的关系。

【重点与难点】

重点：指针与引用的基本概念，指针与引用作为函数参数的传递，引用传递与值传递的区别。

难点：对初学者而言，本章内容总的来说较难掌握。尤其是函数的指针，内存空间的释放，指针与引用作为函数的参数及引用作为函数的返回值等。

【学习方法指导】

指针与引用在 C++中应用非常广泛，但它较难掌握，易于犯错误，且不易查找。在学习过程中，一定要把基本概念搞得很清楚，这样才尽可能地减少编程错误。例如：如何定义变量与函数的指针，它们有什么不同，如何进行指针的初始化，否则，会有什么危险，对定义的指针要不要申请内存，何时申请内存，申请内存何时释放，不释放又有什么后果等。指针与引用经常作为函数的参数进行传递，搞清楚参数传递的过程，并比较引用传递与值传递的区别。

【知识点】

指针；指针变量；new；delete；main 函数的参数；函数指针；引用；typedef

在 C++中，一种比较重要、也较难掌握的一种数据类型，就是指针类型。所谓指针就是在内存中的地址，它可能是变量的地址，也可能是函数的入口地址。如果指针变量存储的地址是变量的地址，我们称该指针为变量的指针（或变量指针）；如果指针变量存储的地址是函数的入口地址，我们称该指针为函数的指针（或函数指针）。

注意：指针变量与变量指针的含义不同：指针变量也简称为指针，是指它是一个变量，且该变量是指针类型的；而变量指针是指它是一个变量，该变量是指针类型的，且它存放另一个变量的地址。

我们知道：局部变量等的内存空间是编译器在栈中自动分配的，它的作用域和生存期局限于所定义的程序块中。我们也可以在堆中用 new 运算符自己申请内存，该内存的生存期由程序员自己控制。指针的一个重要用途就是：可以匿名访问通过 new 运算符在堆中分配的内存。

一般说来，变量有两种方法访问：直接通过变量名访问或通过指针间接访问。以前我们介绍的程序中，对变量的访问大多是通过变量名访问的，变量也可以通过指针间接访问，即通过变量的指针而找到变量的值，这是我们下面要学习的内容。此外，我们还要学习函数的指针。

引用为变量提供了一个别名，变量通过引用访问与通过变量名访问是完全等价的。引用提供了与指针相同的能力，但比指针更为直观，更易于理解。

首先，我们需要明确一下指针的概念。

指针是一种数据类型，具有指针类型的变量称为指针变量。实际上，可以把指针变量（也简称为指针）看成一种特殊的变量，它用来存放某种类型变量的地址。一个指针存放了某个变量的地址值，就称这个指针指向了被存放地址的变量。简单地说，指针就是内存地址，它的值表示被存储的数据的所在的地址，而不是被存储的内容。

为了进一步说清楚指针的含义，需要明白数据在机器中是如何存储和访问的。我们知道，内存是按字节（8 位）排列的存储空间，每个字节有一个编号，称之为内存地址，就象一个大楼里各个房间有一个编号一样。内存中存放的数据包括各种类型的数、地址，还有程序的指令代码，等等。保存在内存中的变量一般占几个字节，我们称之为内存单元，一个内存单元保存一个变量的值。不同的数据类型在机器内存中所占的内存单元的大小一般是不一样的，例如，整型数占两个字节，浮点数占 4 个字节等。但是，在同一个机器上，相同的数据类型占有相同大小的存储单元，而在不同的机器系统里，即使相同的数据类型所占的存储单元也可能是不一样的，例如，16 位机器上，一个整型数占两个字节，而在 32 位机器上，一个整型数占 4 个字节。

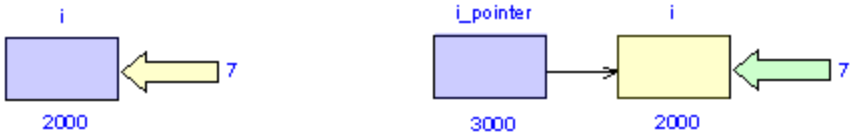
为了访问某个单元中的数据，就必须知道该单元在内存中的地址。这跟我们的实际生活很类似，比如说，当我们要找某一个人的时候，就必须知道他的当前地址，否则，就无法达到目的。

在这里，务必弄清楚一个内存单元的地址与内存单元的内容的区别。

打个比方，为了打开一个 A 抽屉，有两种办法，一种是将 A 抽屉的钥匙带在身上，需要时直接找出该钥匙打开抽屉，取出所需的东西，另一种办法是，为安全起见，将该 A 抽屉的钥匙放在另一个抽屉 B 中锁起来。如果需要打开 A 抽屉，就需要先找出 B 抽屉中的钥匙，打开 B 抽屉，取出 A 抽屉的钥匙，再打开 A 抽屉，取出 A 抽屉中之物，这就是"间接访问"。指针变量相当于 B 抽屉，B 抽屉中的东西相当于地址（如果把钥匙比喻成地址，不甚确切），A 抽屉中的东西，相当于存储单元的内容。

直接访问与间接访问的区别参见下图。为了表示将数值 7 送到变量的存储单元中，有两种方法：（1）直接将 7 送到变量 i 所占的单元中，参见图 7-1(a)。（2）将 7 送到变量 i-pointer 所"指向"的存储单元中，参见图 7-1(b)。

图 7-1



(a)　　(b)

所谓"指向"就是通过地址来体现的。i_pointer 中的值为 2000，就是 i 的地址，这样就在 i_pointer 和 i 之间建立起一种联系，即通过 i_pointer 就能知道 i 的地址，从而找到变量 i 的内存单元。上图中以箭头→表示这种"指向"关系。

既然指针变量的值是一个地址，那么这个地址不仅可以是变量的地址，也可以是函数的地址。在一个指针变量中存放一个数组或一个函数的首地址有何意义呢？因为数组元素或函数代码都是连续存放的。通过访问指针变量取得了数组或函数存储单元的首地址，也就找到了该数组或函数。这样一来，凡是出现数组、函数的地方都可以用一个指针变量来操作。这样做，将会使程序的概念十分清楚，程序本身也精练、高效。

7.1.1 指针变量的定义

我们已经知道，指针类型的变量是用来存放内存地址的。定义了指针类型的变量，就可以在该变量中存放其它变量的地址。如果我们将变量 v 的地址存放在指针变量 p 中，就可以通过 p 访问到 v，我们也说，指针 p 指向变量 v。指针的定义方法是在它所指的变量的类型后面加一个"*"。下面是指针变量定义的例子：

```
int *ptr1;
char *ptr2;
```

这个定义说明：ptr1 和 ptr2 均保存变量的地址，且 ptr1 指向整型变量，ptr2 指向字符变量。定义指针变量时应该注意：

```
int *ptr1;
int* ptr1;
```

是等价的。严格地说，*是属于变量名的。例如：

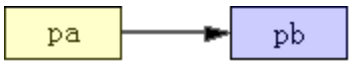
```
int* pa, pb;
```

pa 和 pb 分别是属于什么类型？pa 是一个指向整型变量的指针，而 pb 是一个整型变量。也就是说：*应该是属于变量名的。根据这个定义，我们可以写出下面的语句：

pa= & pb;

这个语句是给指针变量赋值。&称为地址运算符，它是单目运算符，有一个变量作为它的右操作数，其功能是获取变量的地址。该语句执行后，pb 的地址就被赋给了 pa，即 pa 指向 pb，图 7-2 给出了它们的示意图。

图 7-2 简单的整型指针

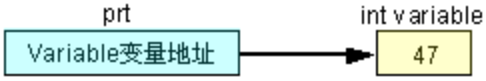


下面的语句

int *ptr;

定义了一个指向整型数据的指针变量，该指针变量的变量名是 ptr。该定义在内存中有如下含义：在内存中有一个存储单元 ptr，它里面存放了另外一个整型数据所在内存的地址。如下图所示：

图 7-3



下面都是指针定义的例子：

float *pf;//定义了一个指向 float 型变量的指针 pf

char *pc;//定义了一个指向 char 型变量的指针 pc

char (*pch)[10];//定义了一个指向 10 个 char 元素组成的数组的指针 pch

int (*pi)();//定义了一个返回值为 int 型的函数的指针 pi

double **pd;//定义了一个指向指针的指针 pd，被指向的指针指向

//一个 double 型变量

在定义指针变量时要注意两点：

（1）变量名前面的"*"，表示该变量为指针变量，但"*"不是变量名的一部分。

（2）一个指针变量只能指向同一个类型的变量。如前面定义的 pf 只能指向浮点变量，不能时而指向一个浮点变量，时而又指向一个字符变量。

在定义了一个指针后，系统会为指针分配内存单元。各种类型的指针被分配的内存单元上大小是相同的，因为每个指针都存放的是内存地址的值，所需要的存储空间当然相同。

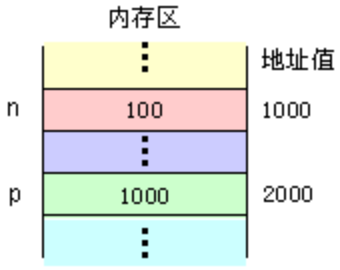
下面通过实例说明指针的含义：

int n = 100;

int *p = &n;

这里首先定义了一个 int 型变量 n，并初始化为 100，然后定义了一个指针变量 p，它指向该 int 型变量。变量 n 的地址是通过取地址运算"&"得到的，并赋给指针变量 p。也就是说，&n 就表示 int 型变量 n 的地址，并把它作为 p 的初值。这样，p 就成了指向变量 n 的指针，如下图所示：

图 7-4



我们假设变量 n 的地址是 1000，指针变量 p 的地址是 2000，n 的值为 100（已知）。由于语句"*p=&n;"是把变量 n 的地址赋给了指针 p，所以指针 p 的数据值为 1000（n 的地址）。

7.1.2 指针变量的使用

假定 pa 指向 pb（如图 7-2），下面的表达式：

*pa

是获取 pa 指向的变量，即为 pb。*称为间接运算符。它是单目运算符，有一个变量作为它的右操作数，其功能是获取指向变量的值。

*pa 也能作为左值，即给 pa 指向的内容赋值。例如：

*pa=10；

在本例中，它与：

pb=10；

的效果是一样的。

C++编译器能够检查数据类型，如果把一个变量赋给一个类型不匹配的数据，可能会出现错误，指针也不例外。例如，如果 ptr1 和 ptr2 的定义如前，下面的语句就会出现编译错误：

ptr2=ptr1;

如果我们把 ptr1 强制转换成 char*类型，再赋给 ptr2，就可以了：

ptr2 = (char*) ptr1;

如果指针类型是 void*类型，则可以与任意数据类型的匹配。例如 7-1：

void 指针在被使用之前，必须转换为正确的类型。例如：

int i = 99;

void *vp = &i;

而下面的语句会产生一个编译错误：

*vp = 3;

如果我们没有让指针变量赋值，指针指向的内容并没有意义。在 C++中，有几个头文件定义了一个常量 NULL（它的值为 0），表示指针不指向任何内存单元。我们可以把 NULL 常量赋给任意类型的指针变量，初始化指针变量。例如：

int *ptr1=NULL;

char *ptr2=NULL;

NULL 常用于基于指针的数据结构（例如链表）的末尾（参见第八章），处理这样的数据结构通常是用循环语句。遇到 NULL 指针时，循环停止。

注意：全局指针变量被自动初始化为 NULL，局部指针变量的初值是随机的。我们编程错误常常出现在没有给指针赋初值。未初始化的指针可能是一个非法的地址，导致程序运行时出现"segmentation fault", "bus error", "system error 2" 等错误，而使程序运行终止。

图 7-2 简单的整型指针

例 7-1

```
void main()
{
   void *vp;
   char c; int i;
   float f; double d;
   // 任意类型的地址能被赋给指针 vp
   vp = &c; vp = &i;
   vp = &f; vp = &d;
}
```

定义了一个指针之后，必须对它初始化之后才能使用，否则将可能会造成系统出错，这一点与其它变量类似。对指针初始化的时候，是要将内存中的一个合法地址赋给它。指针变量中只能存放地址（指针），不能将一个整型量（或任何其它非地址类型的数据）赋给一个指针变量。

变量、数组元素、结构成员等变量的地址都是用运算符&取得的，如：

int a, b[5];

变量 a 的地址可通过表达式&a 得到，数组元素 b[5]的地址可通过表达式&b[5]得到。数组名是的该数组所占内存空间的首地址，同样，函数名也是函数的入口地址，例如：

char str[10];

double max(double []);

在上面的定义中，字符数组 str 的首地址就是数组名 str；函数 max(double [])的入口地址就是 max。

指针运算符有两个：

（1）运算符&

它是一个单目操作符，即只有一个操作数，它返回的是操作数的存储单元地址。

如：abc_addr=&abc;

表示将变量 abc 的地址赋给变量 abc_addr。这里，abc_addr 必须是指针变量。

（2）运算符*

也是一个单目操作符，它返回的是操作数（指针变量）所指的地址的内容。

如：*abc_addr 表示获取指针变量 abc_addr 所指的地址的内容。

设有指向整型变量的指针变量 p，如要把整型变量 a 的地址赋予 p 可以有下面两种方式：

(1) 指针变量初始化的方法

```
int a;
int *p = &a; //定义时初始化
```

(2) 赋值语句的方法

```
int a;
int *p;
p = &a;
```

不允许把一个数赋予指针变量，例如：

```
int *p;
p=1000; //错误：不能将一个整型数赋给指针变量
```

被赋值的指针变量前不能再加"*"说明符，如写为"*p=&a;"，也是错误的。

我们再看下面的例子:

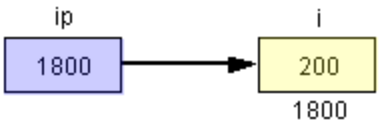
```
int i=200, x;
int *ip;
```

定义了两个整型变量 i、x 及一个指向整型数的指针变量 ip。i、x 中可存放整数，而 ip 中只能存放整型变量的地址。我们可以把 i 的地址赋给 ip：

```
ip=&i;
```

此时指针变量 ip 指向整型变量 i，假设变量 i 的地址为 1800，这个赋值可形象理解为下图所示的联系。

图 7-5



以后我们便可以通过指针变量 ip 间接访问变量 i，例如:

```
x=*ip;
```

运算符*访问以 ip 为地址的存贮单元，而 ip 中存放的是变量 i 的地址。因此，*ip 访问的是地址为 1800 的存贮单元(因为存贮单元中存放的是整数，实际上是从 1800 开始的两个字节)，它就是 i 所占用的存贮单元，所以上面的赋值表达式等价于：

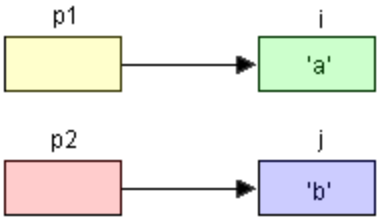
```
x=i;
```

另外，指针变量和一般变量一样，存放在它们之中的值是可以改变的。也就是说，可以它们可以指向不同的变量。例如：

```
int i,j,*p1,*p2;
i='a';
j='b';
p1=&i;
p2=&j;
```

则建立了如下图所示的联系:

图 7-6

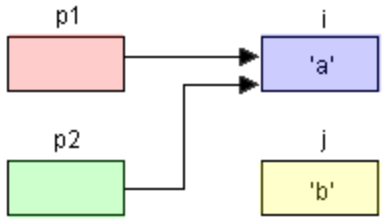


这时赋值表达式:

p2=p1

就使 p2 与 p1 指向同一变量 i。此时，*p2 就等价于 i，而不是 j，参见下图:

图 7-7

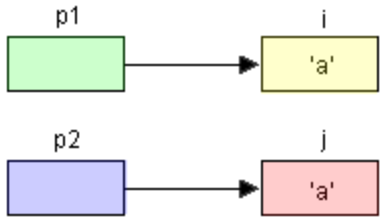


如果执行如下表达式:

*p2=*p1;

则表示把 p1 指向的内容赋给 p2 所指的存储单元, 此时的存储状态参见下图：

图 7-8



通过指针访问它所指向的一个变量，是所谓间接访问。它比直接访问一个变量更费时间，而且不直观。因为，通过指针访问一个变量，取决于指针的值(即指向)。例如，"*p2=*p1;"实际上就是"j＝i;"，前者不仅速度慢，而且目的不明。但是，使用指针变量的优点也是明显的。由于指针是变量，我们可以通过改变它们的指向，间接访问不同的变量，给程序设计带来灵活性，也使得程序代码编写得更为简洁和有效。

指针变量可出现在表达式中, 例如，对于：

int x,y,*px=&x;

指针变量 px 指向整数 x，则*px 可出现在 x 能出现的任何地方。例如:

y=*px+5;/*表示把 x 的内容加 5，并赋给 y*/

y=++*px;/*px 的内容加上 1 之后赋给 y，++*px 相当于++(*px)*/

y=*px++;/*相当于 y=*px;px++;*/ 通常情况下，C++要求指针变量的数据类型和该指针所指向的数据类型一致。因此，在给指针变量赋值的时候一定要注意类型匹配，必要的时候可以使用类型强制转换。

应用指针运算符&和*的例子:

例 7-2

```
#include <stdio.h>
void main( )
{
   int *abc_addr,abc,val;
   abc=67;
```

```
    abc_addr=&abc;
    val=*abc_addr;
    printf("abc_addr=%x\n",abc_addr);
    printf("val=%d",val);
    return;
}
```

运行上述程序后，得到的结果是：

abc-addr=FFF4

val=67

需要注意的是：在不同系统上运行本程序时，得到的内存地址会不相同。 一个 NULL 指针，在 C++中表示不指向内存中任何数据。NULL 是 C++中几个头文件中定义的一个宏，它的值为 0，常常作为初始值赋给指针变量。

对于全局指针变量，它被自动初始化为 0，即 NULL。但是，作为局部变量的指针，如果不被初始化，它的值是不确定的。它可能指向任何地方，也有可能指向非法地址而导致程序出错，定义一个指针变量时，就给改变量初始化是一个好的编程习惯。 下面对"&"和"*"运算符 再做些说明：

（1）*（&abc）=abc，"&"和"*"两个运算符的优先级别相同，但按自右向左方向结合，因此先进?amp;abc 的运算，再进行*运算，相互抵消，结果为 abc。

（2）&(*pointer_1)=&a

（3）*(pointer_1)++相当于 a++。注意圆括号是必要的，如果没有圆括号，就成为了*(pointer-1++),这时先按 pointer_1 的原值进行*运算，得到 a 的值，然后使 pointer_1 的值改变，这样 pointer_1 便不再指向 a 了。

在程序运行过程中，堆内存能够被动态地分配，new 和 delete 两个运算符分别用于堆内存的分配和释放。

注意：malloc,free 和 new ,delete 的不同，前者是函数而后者是运算符。new 和 delete 都是单目运算符，new 的操作数是一个数据类型，返回为该类型的变量分配的内存块的指针。例如：

```
int *ptr＝new int;
char *str＝new char[10];
```

上面的两个语句分别分配了存放一个 int 型变量的内存块及能够存放 10 个字符的内存块，它实际上就是一个字符数组。

在堆中分配的内存的生存期是由程序员自己控制的，例如 7-3：

当 Foo 返回时，局部变量 str 被释放，但它所指的内存还在，直到被程序员显式释放为止。

delete 运算符用于释放由 new 运算符分配的内存。delete 的操作数是指针，释放该指针所指向的内存。例如：

```
delete ptr;// 释放 ptr 指向的内存块
delete [] str;// 释放 str 数组
```

注意：当被释放的内存块是数组时，需要添加[]。如果指针指向的内存不是用 new 申请的堆内存（例如，该内存在栈中），而用 delete 释放时，则会产生一个严重的运行错误。如果指针为空（指针值为 0 或 NULL）时，它不指向任何内存单元，释放没有意义，不过，这不会导致程序出错。

下面我们看一个内存动态申请的一个例子 7-4：

函数 CopyOf 的功能是复制字符串，需要注意的是：该函数的名字前有一个*，表示该函数返回值是一

个指针。形参是一个指向复制的字符串的指针，函数的返回值是指向已复制的字符串的指针。

（1） string.h 是标准的字符串头文件，它包含了许多字符串操作函数的说明，除 strcpy、strlen 外，还有 strcmp、strcat 等。

（2） strlen 函数是统计字符串中字符的个数（不包括结尾符'\0'），我们为字符串申请内存时，除要有存放各字符的内存单元外，还要有一个存放结尾符的单元。

（3） 本例中，strcpy 函数是把字符串 str 复制到 copy，包括结尾符。

我们知道：局部变量的内存单元是被自动分配和释放的，而用 new 申请的堆内存需要用 delete 显式释放。当我们对申请的堆内存不再需要时，就应及时释放。因为内存资源是有限的，如果在程序运行中，申请了许多大的内存块而又没有释放，则有可能使内存资源耗尽。如果程序中存在未被释放的、无用的内存块，我们称之为有内存泄露。内存泄露会导致程序性能降低，甚至崩溃。

在本例中，CopyOf 返回指向申请的内存块的指针，在调用函数中，不再需要该内存时，就应及时释放。

顺便提及：函数的返回值也可以是一个指针，如本例的 CopyOf 函数。带回指针值的函数的一般定义形式是：

数据类型 *函数名(参数表列);

但是，带回指针值的函数，不能将具有局部作用域的变量的地址返回。例如 7-5,定义的函数是错误的：

这是因为 value 是 GetInt 函数局部变量，当 GetInt 函数返回后，value 被释放。存放 value 变量的原内存单元的值是随机的，在调用函数中，并不能根据返回的指针，取回所期望的值。

例 7-3

```
void Foo (void)
{
   char *str = new char[10];
   //...
}
```

例 7-4

```
#include <string.h>
char *CopyOf(const char *str)
{
   char *copy = new char[strlen(str) + 1];
   strcpy(copy, str);
   return copy;
}
```

例 7-5

```
int *GetInt(char *str)
{
   int value = 20;
   …
```

```
    return &value
}
```

一般来说，程序中的变量可以分为全局变量和局部变量。全局变量是存放在内存中固定的内存单元内，且是在编译时分配的，它们的生命期与应用程序相同。所有的函数都可以使用它们；局部变量是存放在栈里，只有当声明它们的函数被调用的时候才存在，也就是说，内存空间是在运行时才分配的，当该函数返回后，这些变量也就不再存在。假如定义了全局变量：

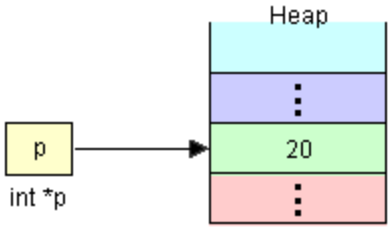
int x;

double score[50];

这两个语句被编译的时候，编译器就分别为 x 分配可存放一个整型数的内存单元，为 score 分配可存放 50 个 double 型数的内存单元。

对于有些变量，它们所需的内存空间无法准确预测，而是在程序运行的时候才能具体确定，比如大小不能事先确定的数组。这类变量的内存空间需要动态分配，它们是存放在称之为堆的内存块中，当然，它们的在堆中存放它们的内存单元地址也只有在分配后才知道。下图给出了指针 int *p 与堆内存的关系（假定已在堆中分配了内存单元，并给该单元存放了值为 20 的整数）：

图 7-10



在 C++中，用户的应用程序可以在堆中申请内存。但是，与局部变量不同，申请的堆内存不会自动释放。因此，如果申请了堆内存，在用完之后一定要释放，否则造成内存泄露。在 C++中，分别用 new 和 delete 来分配和释放内存。

运算符 new 从堆的空闲内存中为程序分配所需内存，并且返回新分配的内存地址。new 后面的操作数是我们希望的、在内存中要存放的数据的类型，C++根据这种数据类型来分配内存空间。例如：

char *string;

string=new char[size+1];

上面的语句通过 new 命令分配了大小为 size+1 个字节的内存空间（size 为一个 int 变量），用于存放一个字符串，该内存空间的首地址为 string。然后，程序就可以用 string 访问这段内存。例如：

cout<<string;

表示输出由 string 寻址的字符串。

delete 用于释放由 new 动态申请的内存。

另外，在进行堆内存分配的时候，我们还可以检测 new 操作是否成功。如果 new 不能成功分配所要求的内存空间，则返回 NULL，否则返回内存单元的首地址。在用完 new 操作之后，通常用下面的语句来进行检测：

```
p=new int;
if(p==NULL)
error("out of memory");
```

在 C++中，指针也能与整数作加减运算，即让指针变量加一个整数或减一个整数。但指针运算与整数

的运算并不相同，它与指针所指向的变量的大小有关。例如：

```
char *str = "HELLO";
int nums[] = {10, 20, 30, 40};
int *ptr = &nums[0]; //  指向 nums 数组第一个元素
```

假定一个 int 型变量占用的内存空间是 4 个字节。在上例中，str++使 str 移动一个字符（一个字节），指向"HELLO"的第二个字符；而 ptr++使 ptr 移动一个 int 型数（即 4 个字节），指向数组的第二个元素，如图 7-11。

所以，"HELLO"的元素可以通过*str、 *(str + 1)、 *(str + 2)等引用，nums 的元素可以用*ptr、 *(ptr + 1)、*(ptr + 2)、*(ptr + 3)等引用。

在 C++中，两个相同类型的指针也允许作减运算。例如：

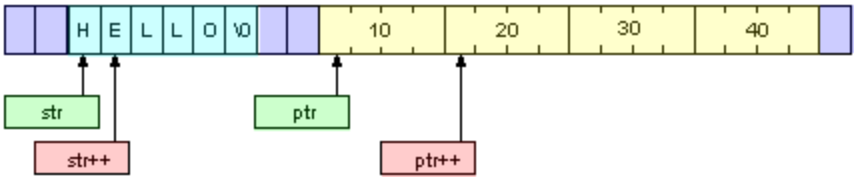
```
int *ptr1 = &nums[1];
int *ptr2 = &nums[3];
int n = ptr2 - ptr1; // n 变为 2
```

在处理数组元素时，指针运算是非常方便的。例如：

```
void CopyString (char *dest, char *src)
{
while (*dest++ = *src++);
}
```

这个循环是将 src 指向的内容赋给 dest 指向的内容，然后，两个指针分别加 1。当 src 的结尾符被赋给 dest 时，条件表达式变为 0，即为假，循环结束。

图 7-11 指针运算



指针是一种特殊的变量，它所允许的运算有下面几种：赋值运算、算术运算和关系运算。

（1） 指针的赋值运算

可以将一个变量的地址赋给一个指针，也可以将一个数组的首地址或一个函数的入口地址赋给指针，但所赋的地址值必须与指针的类型匹配。另外，相同类型的指针之间也可以相互赋值。例如：

```
int a,*p,*q;
p=&a;
q=p;
```

这样就使 q 与 p 指向的同一个变量 a，即 p 和 q 都是指向变量 a 的指针。

另外，为了安全起见，可以将 NULL（即 0）赋给暂时不用的指针，使它不指向任何变量，该指针称为空指针。

（2）指针的算术运算

由于指针存放的都是内存地址，所以指针的算术运算都是整数运算。

一个指针可以加上或减去一个整数值，包括加 1 和减 1。根据 C++地址运算规则，一个指针变量加（减）

一个整数并不是简单地将其地址量加（减）一个整数。而是根据其所指的数据类型的长度，计算出指针最后指向的位置。例如，p+i 实际指向的地址是：

p+i*m

其中 m 是数据存储所需的字节数，一般情况下，字符型数据 m=1，整型数据 m=2，浮点型数据 m=4。例如，下面的语句说明了一个 int 型指针变量 p 进行算术运算的情况。

```
int *p; //p=3000
p++; //p=3002
p--; //p=2FFE
```

因为一个整数在内存中占两个字节的空间。p++操作是使指针 p 指向下一个整型数据，同理可知，p--操作是使指针 p 指向前一个整型数据。

此外，如果两个指针所指的数据类型相同，在某些情况下，这两个指针可以相减。例如，指向同一个数组的不同元素的两个指针可以相减，其差便是这两个指针之间相隔元素的个数。又例如，在一个字符串里面，让指向字符串尾的指针和指向字符串首的指针相减，就可以得到这个字符串的长度。

（3）指针的关系运算

在某些情况下，两个指针可以相比较，但要求这两个指针指向相同类型的数据。指针间的关系运算包括：>、>=、<、<=、==、!= 。例如，比较两个指向相同数据类型的指针，如果它们相等，就说明它们指向同一个地址（即同一个数据）。

如：if(p1==p2) printf("two pointrs are equal.\n");

指向不同数据类型的指针之间进行关系运算是没有意义的。但是，一个指针可以和 NULL（0）作相等或不等的关系运算，用来判断该指针是否为空。

我们在定义指针时，如果在*的右边加一个 const 修饰符，则定义了一个常量指针，即指针值是不能修改的：

```
int d =1;
int* const p =&d;
```

p 是一个常量指针，它指向一个整型变量。p 本身不能修改，但它所指向的内容却可以修改：

```
*p =2;
```

我们也可以用定义常量指针指向常量，以下两种形式都是合法的：

```
int d =1;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
```

现在，指针和变量都不能改变。

注意：定义指针常量时必须初始化。下面的常量指针定义是错误的：

```
const int* const x;
```

同其它变量一样，指针也可以作函数的参数。我们以下面的 swap 函数为例，该函数的功能是交换两个整型变量的值：

```
void swap(int* pa, int* pb)
{
int temp = *pa;
*pa = *pb;
*pb = temp;
```

}

注意：被交换的值是 pa、pb 指向的内容，不是 pa、pb 本身。

调用 swap 函数，会影响到实参的值。例如：

```
float x=10, y = 20, z = 0;
swap (&x, &y);
cout<< x <<", "<< y << endl;
```

输出结果为：20, 10。这是因为当 swap 被调用时，会创建两个临时指针变量 pa、pb，并分别被初始化为实参 x、y 的地址，即相当于：

```
int *pa= &x, *pb= &y;
```

函数中参与运算的值不是 pa、pb 本身，而是它们指向的内容，也就是实参 x、y 的值（*pa 与 x、*pa 与 y 占用相同的内存单元）。所以在 swap 函数中改变了*pa，也就是改变了实参 x，改变了*pb，也就是改变了实参 y。交换了*pa 与*pb，也就是交换了 x、y 的值。

从上面我们可以看到：在调用函数中，是把实参的指针传送给形参，即传送&x、&y，这是函数参数的引用传递。但是，作为指针本身，仍然是函数参数的值传递的方式。因为在 swap 函数中创建的临时指针，在函数返回时被释放，它不能影响调用函数中的实参指针（即地址）值。

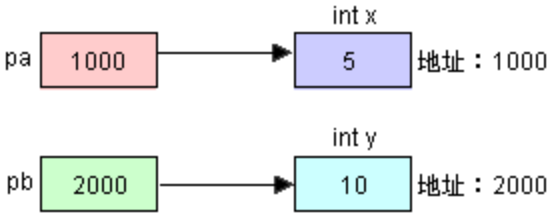
函数的参数不仅可以是整型、实型、字符型等数据，还可以是指针。它的作用是将一个变量的地址传送到另一个函数中。

我们知道，函数调用的的值传递方式中，形参和实参均占用不同的存储单元，形参值的变化不会影响实参的值，也就是说，值传递方式不能带回参数值。我们也知道，一个函数用 return 只能返回一个函数值，如果要求函数返回多个值，一个可行的方法是将变量的地址作为函数的参数进行传递，也就是把指针作为函数的参数。

对于 swap 函数，定义了两个整型指针 pa、pb 作为形参，在函数内部，通过对 pa、pb 的操作，改变它们所指的变量的值。在调用的时候，swap(&x,&y)，分别把整型变量 x、y 的地址作为参数传给 pa、pb，在 swap 函数内部，通过对 x、y 地址的引用，交换的是该地址所指向的数据，即交换了 x、y 的值。

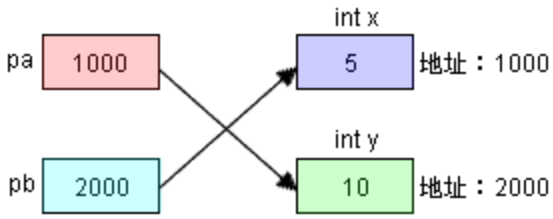
更清晰的表达可以从下面的图中看出：

图 7-13



执行交换后，

图 7-14



而*pa 和*pb 的交换，实际上是对地址 1000 和地址 2000 内存放的数据进行交换。所以，执行完 swap 函数以后，地址 1000 和地址 2000 内的数据做了交换，临时指针变量 pa、pb 被释放。

下面再举一例，输入 a、b、c3 个整数，按大小顺序输出，要求 a 为最小，c 为最大。

例 7-6

```
void swap(int *pt1,int *pt2)
{
   int temp;
   temp=*pt1;
   *pt1=*pt2;
   *pt2=temp;
}
void exchange(int  *q1,int *q2,int *q3)
{
   if(*q1<*q2) swap(q1,q2);
   if(*q1<*q3 swap(q1,q3);
   if(*q2<*q3) swap(q2,q3);
}

void main()
{
   int a,b,c,*p1,*p2,*p3;
   scanf("%d,%d,%d",&a,&b,&c);
   p1=&a;p2=&b;p3=&c;
   exchange(p1,p2,p3);
   printf("\n%d,%d,%d \n",a,b,c);
}
```

拿 exchange 函数中第一条 if 语句分析，如果 a>b，调用函数 swap，其中&a 和&b 是实参，swap 函数通过中间变量 temp 交换*pt1 和*pt2。因为:

pt1=(&a)=a

pt2=(&b)=b

实际上，通过中间变量 temp 交换 a 和 b。

在 C++中，指针与数组是密不可分的。数组名本身就是指针（地址），是数组元素在内存中的首地址，

数组元素可用下标访问，也可以用指针访问。指针本身也可以定义成数组，称之为指针数组。下面，我们将分别介绍。

首先明确一下数组和指针的关系。

数组是具有相同类型的一组变量的有序集合，数组元素存放在一段连续的内存区域里，每个元素占用的内存单元大小相同，数组名就是数组所占存储区域的首地址，也就是指向该数组第一个元素的地址（指针）。

在 C++中，指针和数组是紧密相关的两种数据类型，它们计算地址的方法相同。数组的元素可以用下标表示，也可以用指针表示。假定一个指针变量指向数组，就能用该指针访问数组里的元素，用指针访问数组元素与用数组下标访问数组元素的效果是一样的。

下面的程序可以说明如何使用指针访问数组元素：

例 7-7

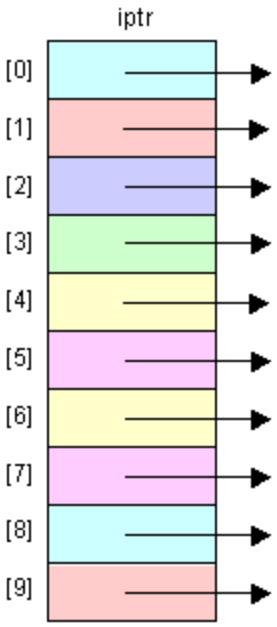
```
#include <iostream.h>
#define SIZE 10
void main()
{
   int array[SIZE];
   int *p=array;
   int i ;
   for(i=0;i<SIZE;i++)
      array[i]=i;
   for(i=0;i<SIZE;i++)
      cout<<*p++<<endl;
   p=&array[8];
   cout<<"array[8]="<<*p<<endl;
   cout<<"array[8]="<<array[8]<<endl;
}
```

这个程序首先声明了一个大小为 SIZE 的整型数组 array 和一个整型指针 p，并且把数组的首地址赋给 p。接着，对各数组元素赋值，并用指针 p 输出该数组元素。*p 实现了对 p 所指向的数组元素的访问，p++使 p 指向下一个数组元素，*p++就是访问下一个数组元素。最后，将 array[8]的地址赋给 p，并输出了 array[8]和*p，它们的结果是一样的，都是对数组 array 第 9 个元素的访问。

7.6.1 指针数组

所谓指针数组就是指数组的每一个元素都是指针，左面是一个定义指针数组的例子：

图 7-15 有 10 个整型指针的数组



数组元素也可以是指针类型，数组元素为指针的数组称之为指针数组。指针数组是一种很有用的数据结构，它使得数组元素可以指向不同的内存块，实现对不同大小的的内存块的数据统一管理。指针数组的一般定义形式为：

类型标识符 *指针数组[元素个数];

编译时，根据数组的大小为指针数组分配相应的内存空间。例如：

int *p[4];

定义了有 4 个指针元素的指针数组 p，每个指针元素指向一个整型变量。给数组元素赋值时，是为每个元素赋一个整型变量的地址。指针数组的应用举例如下：

例 7-8

```
#include <stdio.h>
void main( )
{
   int a[2][3]={1,3,5,7,9,11};
   int *pa[2], i, j;
   pa[0]=a[0];
   pa[1]=a[1];
   for(i=0;i<2; i++)
      for(j=0;j<3;j++,pa[i]++)
         printf("a[%d][%d]=%d\n",i,j,*pa[i]);
   return;
}
```

程序中，pa[0]=&a[0]表示将数组 a 第 1 行元素的首地址赋给指针数组 pa[0]，所以也可以写成："pa[0]=&a[0][0]；"，使用了表达式：pa[i]++来修改指针数组 pa 所指向的数据。例如，当 i=0 时，通过 pa[i]++便得到数组元素 a[0][0]、a[0][1]和 a[0][2]的地址。

7.6.2 用指针访问数组元素

我们已经知道：数组名就是数组在内存中的首地址，即数组中第一个元素的地址。我们可以通过下标访问数组元素，也可以通过指针访问数组元素。对于下面的数组：

int ara[5] = {10, 20, 30, 40, 50};

如果打印 ara[0]和*ara 均输出结果 10，即数组 ara 第一个元素的值。如果要打印 ara 的第三个元素，可用下面的语句：

cout << *(ara+2) ; // 即打印 ara[2]

对于数组 ara：

ara+0 指向 ara[0]

ara+1 指向 ara[1]

ara+2 指向 ara[2]

ara+3 指向 ara[3]

ara+4 指向 ara[4]

虽然数组名是指针，但它是常量指针，数组名的值是不能改变的。

我们也可以用指针访问二维数组元素，不过比一维数组元素的访问麻烦一些。由于二维数组元素在内存中是一维的，且按行存放。所以用指针访问二维数组的关键是如何计算出某个二维数组元素在内存中的地址。例如，对于一个 m 行、n 列的二维数组 num，num[i][j]（i≤m，j≤n）在内存中的地址应为：num+i×n+j。

例 7-8：假定我们将一年四季的平均气温存放在一个二维数组 temp 中，例如，第三年的春、夏、秋、冬四季的平均气温分别为 15、30、18、-1，则有：

temp[2][0]=15；temp[2][1]=30；temp[2][2]=18；temp[2][3]=-1；

第五年的春、夏、秋、冬四季的平均气温分别为 16、29、17、0，则有：

temp[2][0]=16；temp[2][1]=29；temp[2][2]=17；temp[2][3]=0；

需要查找一年四季中的最高的平均气温，函数如下：

例 7-9

```
int HighestTemp (const int *temp, const int rows, const int columns)
{
   int highest = 0;
   for (register i = 0; i < rows; ++i)
      for (register j = 0; j < columns; ++j)
         if (*(temp + i * columns + j) > highest)
            highest = *(temp + i * columns + j);
   return highest;
}
```

本例中，函数的形参是一个 int 型指针和另外两个整型变量，以指定数组各维的大小。三个形参的 const 修饰符表示：temp 指向的内容和 rows、columns 的值在函数中是不能修改的。用指针作函数形参同用二维数组作函数形参相比，更加灵活，因为二维数组作函数形参时，不能省略第二维大小的说明。

表达式*(temp + i * columns + j)是取数组的第 i 行、第 j 列的元素，等价于 temp[i][j]。

由于二维数组在内存中的存储是一维的， HighestTemp 函数实际上能够简化为用大小为 row * column 的一维数组来处理：

例 7-10

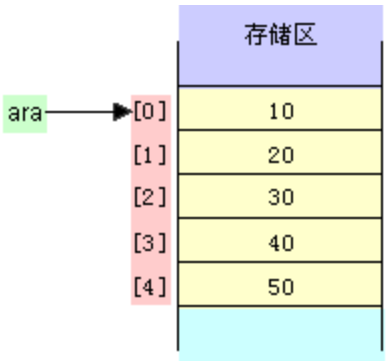
```
int HighestTemp (const int *temp, const int rows, const int columns)
{
  int highest = 0;
  for (register i = 0; i < rows * columns; ++i)
    if (*(temp + i) > highest)
      highest = *(temp + i); return highest;
}
```

前面已经提到过，C++中数组和指针是密切相关的。可以用指针访问数组的元素，这是因为数组名实际上就是指针，它指向数组的第一个元素。上面的例子实际上就是用指针访问数组元素，下面就这个问题，我们作进一步的说明。对于数组：

int ara[5] = {10, 20, 30, 40, 50};

在内存中是这样存储的：

图 7-16



由于数组名是数组的首地址，它是一个常量，所以不能改变。但是，对数组名可以在表达式中参与算术运算，例如：ara 指向 ara[0]，表达式 ara+1 指向 ara[1]，…

指针可以象数组那样使用下标，这进一步表明了指针和数组的密切关系。例如，p 是一个数组的指针，p[k]则指向该数组的第 k+1 个元素，所以，p[k]与*(p+k)是等价的。

此外，也可以用指针对多维数组进行访问。以二维数组为例，

int m[3][3];

可以看成是以 m[0]、m[1]、m[2]为首地址的三个一维数组，每个一维数组具有三个整型元素。如果定义了下面的指针数组：

int *pm[0];

pm[0]=m[0];

pm[1]=m[1];

pm[2]=m[2];

那么，指针数组的三个元素（即三个指针）分别指向三个一维数组。这样，就可以通过这三个指针对二维数组进行访问。例如："m[2][0]=*(pm[2]);m[2][1]=*(pm[2]+1)"。

根据以上叙述，引用一个数组元素可以用：

（1） 下标法。即用 a[i]形式访问数组元素，在前面介绍数组时都是采用这种方法。

7（2） 指针法。即采用*(a+i)或*(p+i)形式，用间接访问的方法来访问数组元素，其中 a 是数组名，p 是指向数组的指针变量，其初值为 p=a。

下面三个程序段均以输出数组中的全部元素为例，说明它们的用法：

程序段 7-1

```
void main() //下标法
{
   int a[10],i;
   for(i=0;i<10;i++)
      a[i]=i;
   for(i=0;i<10;i++)
      printf("a[%d]=%d\n",i,a[i]);
}
```

程序段 7-2

```
void main()
{
   int a[10],i;
   for(i=0;i<10;i++)
      *(a+i)=i;
   for(i=0;i<10;i++)
      printf("a[%d]=%d\n",i,*(a+i));
}
```

程序段 7-3

```
void main() //指针法（用指针变量指向元素）
{
   int a[10],i,*p;
   p=a;
   for(i=0;i<10;i++)
      *(p+i)=i;
   for(i=0;i<10;i++)
      printf("a[%d]=%d\n",i,*(p+i));
}
```

因为，多维数组在内存中也是按照一维顺序存放的，所以访问的时候，也可以通过对指针作算术运算的方法对数组元素进行访问。

7.6.3 main 函数的参数

到目前为止，我们编写的程序中，main 函数都没有参数。实际上 main 函数也可以带参数，它们用于接收命令行参数。main 函数原型为：

返回类型 main(int argc, char *argv[])

其中，argc 是参数个数（文件名也是参数），argv 是一个指向字符串的指针数组。

例 7-11

```
//
command.cpp
#include
<iostream.h>
void
main( int
argc, char
*argv[] )
{
   int i;
   for( i = 1; i
< argc; i++ )
      cout <<
argv[i] << "
";
   cout <<
endl;
}
```

command.cpp 文件中的 main 函数的功能是：打印从命令行输入的、除文件名外的其它字符串。假如我们将文件 command.cpp 编译、连接，生成执行文件 command.exe，在命令行提示符下键入：

command I am learning C++

则 argc=4，argv[0]="I"， argv[1]="am" ，argv[2]="learning" ，argv[3]="C++"。

输出结果为：

I am learning C++

由于 main 函数不能被其它函数调用，只能被系统调用，因此不可能在程序内部取得实际值。那么，在何处把实参值赋予 main 函数的形参呢？实际上，main 函数的参数值是从操作系统命令行上获得的。当我们要运行一个可执行文件时，在 DOS 提示符下键入文件名，再输入实际参数，便可把这些实参传送到 main 的形参中去。

DOS 提示符下命令行参数的一般形式为：

C:\>可执行文件名 参数 参数……

需要注意的是：main 的两个形参和命令行中的参数在位置上不是一一对应的。因为，main 的形参只有二个，而命令行中的参数个数原则上未加限制。argc 参数表示了命令行中参数的个数(文件名本身也算一个参数)，argc 的值是在输入命令行时由系统按实际参数的个数自动赋予的，而其它参数的标识符被存放在指针数组 argv 中。

例如，如果运行程序时，键入命令：

c:\path\program -c filename.txt

则命令行参数共有 3 个，即 argc=3，它们分别是 c:\path\program、-c、filename.txt。标识这三个参数的字符串的地址分别存放在 argv 数组的 argv[0]、argv[1]和 argv[2]中。

在 C++中，可以将函数地址保存在函数指针中，然后用该指针间接调用函数。例如：

```
int (*Compare)(const char*, const char*);
```

该语句定义了一个函数名为 Compare 的函数指针，它能用于保存任何有两个常量字符形参、返回整型值的函数的地址。例如，Compare 能指向 C++标准的字符串比较函数库中的函数 strcmp：

```
Compare = &strcmp; // Compare 指向 strcmp 函数
```

&运算符可以省略：

```
Compare = strcmp; // Compare 指向 strcmp 函数
```

函数指针也能在定义时初始化：

```
int (*Compare)(const char*, const char*) = strcmp;
```

当把函数地址赋给函数指针时必须匹配。上面的定义是有效的，因为函数 strcmp 的原型与 Compare 匹配：

```
int strcmp(const char*, const char*);
```

给定了上面的定义后，strcmp 能被直接调用，也能通过 Compare 被间接调用，下面的三个调用是等价的：

```
strcmp("Tom", "Tim"); // 直接调用
(*Compare)("Tom", "Tim"); // 间接调用
Compare("Tom", "Tim"); // 间接调用
```

函数指针通常作为另一个函数的参数，我们这里以折半查找函数作为例子。假定对一个已经由小到大排序（按字符串中字符的 ASCII 码或 Unicode 码的大小排序）的字符串数组（称为查找表），查找它的元素中是否存在一个元素与另一个字符串（称为查找项）相等，函数的代码如下：

例 7-12

```
1 int BinSearch (char *item, char *table[],
2 int n, int (*Compare)(const char*, const char*))
{
   int bot = 0;
   int top = n - 1;
   int mid, cmp;
3 while (bot <= top)
   {
4 mid = (bot + top) / 2;
5 if ((cmp = Compare(item,table[mid])) == 0)
6     return mid; // 返回查出的元素的在数组中的序号
   else if (cmp < 0)
          top = mid - 1; // 查找数组的下半部
       else
7          bot = mid + 1; // 查找数组的上半部
```

```
    }
8 return -1; // 未查到
}
```

本例中，查找表是数组 table，查找项是变量 item，在 table 中查找是否含有 item 的元素。Compare 是一个函数指针，用于比较 item 与 table 中的元素是否相等。函数体主要是一个 while 循环语句，每循环一次，对 table 元素的查找范围就减小一半。当 item 与 table 中的某一个元素匹配成功或查找范围的两端（bot 和 top）产生冲突（即 bot>top）时，查找结束。

item 每次与位于 table 中间的元素进行比较，如果匹配成功，即它们相等，则返回后者在 table 中的序号；如果 item 比中间元素小，则查找 table 的下半部；如果 item 比中间元素大，则查找 table 的上半部；如果 table 中没有与 item 相等的元素，则函数返回-1。

下面给出了调用 BinSearch 函数的例子：

```
char *cities[] = {"Beijing", "Shanghai", "Tianjin", "Zhongqing"};
cout << BinSearch("Tianjin  ", cities, 4, strcmp) << '\n';
```

上述调用的输出结果是 2。

在C++语言中，一个函数的代码总是占用一段连续的内存区，而函数名就是该函数所占内存区的首地址。我们可以把函数的这个首地址(或称之为入口地址)赋予一个指针变量，也就是说，使该指针变量指向该函数。然后，通过指针变量就可以调用这个函数。我们把这种指向函数的指针变量称之为"函数指针变量"。如果一个指针 p 存放了一个函数的入口地址，我们也说指针 p 指向了该函数。

函数指针的声明方法是：在变量名前加一个"*"，并且将"*"和变量名用圆括号括起来，还要指明该变量指向的函数的参数类型、个数和函数的返回类型。即：

类型说明符 （*指针变量名）（参数表）；

其中"类型说明符"表示被指函数的返回值的类型。"(* 指针变量名)"中"*"后面的变量是被定义的函数指针变量名。后面括号中的"参数表"表示被指函数所带的参数。例如：

```
float (* funptr)(int x,int y);
```

这里定义了一个指向函数的指针变量 funptr，所指函数带两个整型参数，返回类型为 float。下面定义函数指针方法是不对的：

```
float * funptr(int x,int y);
```

实际上，这是一个函数的声明，funptr 函数有两个整型参数，返回类型为指向 float 变量的指针。

和指向变量的指针一样，使用函数指针之前，一定要对该指针进行初始化，使它对可执行程序寻址。给函数指针初始化时，要注意指针和被指向的函数在函数参数的个数、类型及函数的返回值要匹配。

下面的例子用来说明用指针形式实现对函数调用的方法。

例 7-13

```
int max(int a,int b)
{
  if(a>b)return a;
  else return b;
}
void main()
{
```

```
    int max(int a,int b);
    int(*pmax)( int a,int b);
    int x,y,z;
    pmax=max;
    printf("input two numbers:\n");
    scanf("%d%d",&x,&y);
    z=(*pmax)(x,y);
    printf("maxmum=%d",z);
}
```

从上述程序可以看出用，函数指针变量形式调用函数的步骤如下：

（1） 定义函数指针变量，如后一程序中第 9 行中的" int (*pmax)(int a,int b);"，定义 pmax 为指向函数的指针变量。

（2） 被调函数的入口地址(函数名)赋予该函数指针变量，如程序中第 11 行 pmax=max;

（3） 用函数指针变量形式调用函数，如程序第 14 行 z=(*pmax)(x,y);

调用函数的一般形式为：

（* 指针变量名）（参数表）；

注意:

（1） 函数指针变量不能进行算术运算，这与数组指针变量是不同的。数组指针变量加减一个整数可使指针移动指向后面或前面的数组元素，而函数指针的移动是毫无意义的。

（2） 函数调用中"(*指针变量名)"的圆括号不可少，其中的*不应该理解为求值运算，在此处它只是定义指针变量的一种符号。

引用是一个变量的别名，除用&取代*外，定义引用的方法与定义指针类似。例如：

```
double num1 = 3.14;
double &num2 = num1; // num 是 num2 的引用
```

定义 num2 为 num1 的引用，它并没有复制 num1，而只是 num1 的别名。也就是说，它们是相同的变量。例如，如果执行下面的语句：

```
num1 = 0.16;
```

则 num1 和 num2 的值均为 0.16。

不同于变量的定义：可以先定义，后初始化。正如上面看到的：引用必须在定义时初始化。例如，下面的定义是错误的：

```
double &num3; // 非法：引用没有初始化
num3 = num1;
```

引用可用常量来初始化，此时，常量会被复制，引用与其复制值保持一致：

```
int &n = 1; // n 取 1 的复制值
```

为什么用常量初始化引用时要被复制，我们看下面的例子：

```
int &x = 1;
++x;
int y = x + 1;
```

第一行的 1 和第三行的 1 可能占用相同的存储单元(大多数编译器将两个 1 分配在内存中相同的地方)，虽然我们期望 y 的值为 3，但它的结果可能是 4，这是由于第二行的++x 运算后，常量 1 的值变成了 2。如

果我们使 x 取 1 的复制值，可保证 x 的改变不会影响到常量的值。

其实，引用作为另一个变量的别名用处不是很大，除非变量名很长。引用最重要的用处是作函数的参数。我们知道，函数参数传递有值传递和引用传递两种方式。用引用作为函数，是引用传递方式。为了比较值传递和引用传递的区别，我们仍然以交换两个变量值的函数作为例子：

例 7-14

```
1 void Swap1 (int x, int y) //值传递
{
   int temp = x;
   x = y;
   y = temp;
}
2 void Swap2 (int *x, int *y) //引用传递（指针）
{
   int temp = *x; *x = *y;
   *y = temp;
}
3 void Swap3 (int &x, int &y) //引用传递
{
   int temp = x;
   x = y;
   y = temp;
}
```

在上面的三个函数中，虽然 Swap1 交换了 x 和 y，但并不影响传入该函数的实参，因为实参传给形参时被复制，实参和形参分别占用不同的存储单元。

Swap2 使用指针作为参数克服了 Swap1 的问题，当实参传给形参时，指针本身被复制，而函数中交换的是指针指向的内容。当 Swap2 返回后，两个实参可以达到交换的目的。

Swap3 通过使用引用参数克服了 Swap1 的问题，形参是对应实参的别名，当形参交换以后，实参也就交换了。

下面的 main 函数说明调用三个函数时的区别：

例 7-15

```
void main (void)
{
   int i = 10, j = 20;
   Swap1(i, j);
   cout << i << ", " << j << '\n';
   Swap2(&i, &j);
   cout << i << ", " << j << '\n';
   Swap3(i, j);
```

```
    cout <<i <<", " <<j <<'\n';
}
```

main 函数运行结果如下：

10, 20

20, 10

10, 20

我们看到函数 Swap3 与 Swap2 的效果一样，都达到了交换的目的，但 Swap3 更直观，调用它的方法与调用 Swap1 的方法是一样的。但是，引用作为函数参数，调用时可能会出现歧义，例如：

```
void fn(int s)
{
  …
}
void fn(int& t)
{
  …
}
void main()
{
  int a=5;
  fn(a); //匹配哪一个函数?
}
```

当以引用方式传递函数参数时，我们常使用 const 关键字。例如：

```
void f1(const int i) {
  i++; // 非法，编译错误
}
```

这可以避免在函数中修改了不该修改的参数，有助于提高程序的可靠性。

使用引用的规则：

1. 初始化后，程序不能改变引用的值。
2. 不能创建指向引用的指针。
3. 不能比较两个引用的值，可比较被引用变量的值。
4. 不能使引用的值加，减和改变，但对被引用变量的值可以。
5. 不能对 void 进行引用。
6. 不能建立引用数组。
7. 没有引用的引用。
8. 有空指针，无空引用。

引用实际上是一种隐式指针，它为变量建立一个别名。引用定义一般形式如下：

类型 &变量＝变量;

这里的"类型"可以是基本的数据类型，也可以是用户自己定义的类型，赋值号左边的"变量"是引用变量名，它是赋值号右边"变量"的别名。

引用变量只是它所引用的变量的别名，对它的操作与对原来变量的操作具有相同的作用。举例如下：

int i=0;

int &ref=i;

ref=2;

上面的语句首先定义了一个整型变量 i 和一个引用变量 ref，它是变量 i 的别名。那么对 i 的操作和对 ref 的操作的结果完全一样。语句"ref=2;"把 2 赋值给 ref，实际上也改变了 i 的值。还有一点需要注意的是：引用必须在定义的时候初始化，如果先定义，后赋值编译时会出错。引用除了用变量初始化外，还可以用常量来初始化。 引用可以作为函数参数进行传递，称为引用传递。书中用交换两个变量值的三个函数为例，清楚地说明了值传递、指针传递、引用传递三种方式是如何工作的。

下面再举一个例子说明引用的使用：

例 7-16

```
#include <iostream.h>
int main()
{
   int a;
   int &ref=a;
   a=10;
   cout<<a<<"---"<<ref<<endl;
   a=100;
   cout<<a<<"---"<<ref<<endl;
   int b=20;
   ref=b; //把 b 的值赋给 a
   cout<<a<<"---"<<ref<<endl;
   ref--;
   cout<<a<<"---"<<ref<<endl;
   return 0;
}
```

程序运行输出结果如下：

10---10

100---100

20---20

19---19

程序中定义了一个引用变量 ref，它实际上是整型变量 a 的一个别名。对 ref 的任何操作等价于对 a 的操作。

C++不仅提供了丰富的数据类型，而且还允许由用户自己定义类型说明符，也就是说允许由用户为数据类型取"别名"，该功能是通过关键字 typedef 来完成的。例如，有整型量 a、b：

int a,b;

int 是整型变量的类型说明符，int 实际上是 integer 的简写。为了增加程序的可读性，可把整型说明符用 typedef 定义为：

typedef int INTEGER

以后就可用 INTEGER 来代替 int 作整型变量的类型说明了，例如：

INTEGER a,b;

它等效于：

int a,b;

用 typedef 定义数组、指针、结构等类型将带来很大的方便，不仅使程序书写简单而且使意义更为明确，因而增强了可读性。 C++中是用关键字 typedef 定义一个标识符来代表一种数据类型，该标识符可以象其它基本类型的标识符一样使用。在用 typedef 进行类型定义时，其语法和变量定义很相似。typedef 定义的一般形式为：

typedef 原类型名 新类型名

例如：

typedef int * intptr;

定义 intptr 为一个指向整型的指针类型。变量定义语句：

intptr p;

等价于：

int * p;

需要注意的是：typedef 定义的类型只是 C++已有类型的别名，而不是新类型。有时也可用宏定义来代替 typedef 的功能，但是宏定义是由预处理完成的，而 typedef 则是在编译时完成的，后者更为灵活方便。

typedef 可以为数据类型定义一个符号名，就象引用是一个变量的别名一样。它的主要用途是简化复杂的类型说明，改进程序的可读性。下面是几个例子：

typedef char *String;

Typedef char Name[12];

typedef unsigned int uint;

这些定义的效果是 String 变成 char*的别名，Name 变成有 12 个元素的字符数组的别名，uint 变成 unsigned int 的别名。

String str; // 等价于 char *str;

Name name; // 等价于 char name[12];

uint n; // 等价于 unsigned int n;

前面复杂的的 Compare 说明，用 typedef 简化后，可读性就好多了：

```
typedef int (*Compare)(const char*, const char*);
int BinSearch (char *item, char *table[], int n, Compare comp)
{
  //...
  if ((cmp = comp(item, table[mid])) == 0)
  return mid;
  //...
}
```

typedef 使得 Compare 作为一个新的类型名，它表示一个有给定原型（有两个常量字符指针参数，返回值为 int 型）的函数指针类型。

7.10.1 字符串处理实例

从键盘上输入两个字符串：

◇ 对两个字符串分别由小到大排序

◇ 将它们合并，合并后的字符串按由小到大排序，并删去相同的字符

◇ 显示排序和合并后的字符串。

算法：

根据题意，对字符串的处理分为三步：第一步是从键盘上输入两个字符串；第二步是将两个字符串分别排序；第三步是将字符串合并；第四步是显示处理结果。

第一步和第四步好办，关键是第二步和第三步的处理，下面分别加以说明。

字符串排序是指将一个字符串中各个字符按照 ASCII 码值的大小排序。例如，字符串"Beijing"由小到大的排序结果应该是："Bejiign"。排序算法很多，第二个例子，我们就要介绍快速排序算法。这里，我们使用简单的冒泡排序算法：即将字符串中的每一个字符一个个进行比较，找出最小的字符，然后，再在剩下的字符中找最小的字符，…。例如：字符?quot;Beijing"的排序过程如下：

第一次将字符"Beijing"中的每一个字符：'B'、'e'、'i'、'j'、'i'、'n'、'g'进行比较，找到最小的字符'B'。

第二次在剩下的字符 'e'、'i'、'j'、'i'、'n'、'g'中，找到最小的字符'e'。

第三次在剩下的字符 'i'、'j'、'i'、'n'、'g'中，找到最小的字符'j'。

…

第三步是合并字符串，合并后的字符串仍然由小到大排序。由于待合并的两个字符串已经排好序。假定两个排好序的字符串分别为 A 和 B，合并后的字符串为 C，要使待合并后的字符串仍然由小到大排序，可采取下述步骤：

（1） 从前往后取 A 中的字符，并按从前往后的顺序与 B 中的字符比较，若 A 中的字符较小，则将该字符存入 C，并移到 A 的下一个字符，继续与 B 中的字符比较。

（2） 若 A 中的字符较大，则将 B 中的字符存入 C，并移到 B 的下一个字符，继续与 A 中的字符比较。

（3） 若 A 与 B 中的字符相等，则将 A 或 B 中的字符存入 C，并将 A 和 B 均移到下一个字符。

（4） 若 A 或 B 字符串到达末尾，则将 B 或 A 的剩余部分加到字符串 C 中。

需要注意的是：A、B 和 C 三个字符串均可以用字符数组来表示，C 数组的长度不能小于 A、B 两数组的长度之和。另外，判别字符串是否结尾的方法是：从 A 或 B 中取出的字符是否为'\0'，所有字符串都是以'\0'结尾的。

程序：

根据上面的分析，我们可以写出下面的程序。

例 7-17

```
#include <stdio.h>
void strmerge(char *a,char *b,char *c)//将字符串 a,b 合并到字符串 c 中
{
  char t,*w;
  w=c;
  while((*a!='\0')&&(*b!='\0'))
  {
  //找到字符串 a,b 当前字符中较小的字符
```

```
    if(*a<*b)
    {
       t=*a;
       a++;
    }
    else if(*a>*b)
    {
       t=*b;
       b++;
    }
    else //字符串 a,b 当前字符相等
    {
       t=*a;
       a++;
       b++;
    }
    if(*w=='\0')//开始，可直接赋值
       *w=t;
    else if(t!=*w)
    //如果 a,b 中较小的当前字符与 c 中当前字符不等，才赋值
       *(++w)=t;
}
if(*a!='\0') //如果字符串 a 还没有结束，则将 a 的剩余部分赋给 c
while(*a!='\0')
    if(*a!=*w)
    {
       *(++w)=*a;
       a++;
    }
    else
       a++;
if(*b!='\0') //如果字符串 b 还没有结束，则将 b 的剩余部分赋给 c
while(*b!='\0')
    if(*b!=*w)
    {
       *(++w)=*b;
       b++;
    }
    else
       b++;
       *(++w)='\0';
```

```
    }

void strsort(char *s)//将字符串 s 中的字符排序
{
    int i,j,n;
    char t,*w;
    w=s;
    for(n=0;*w!='\0';n++) //得到字符串长度 n
        w++;
    for(i=0;i<n-1;i++) //对字符串 s 进行排序，按字母先后顺序
        for(j=i+1;j<n;j++)
            if(s[i]>s[j])
                {
                    t=s[i];
                    s[i]=s[j];
                    s[j]=t;
                }
    }

void main()
{
    char s1[100],s2[100],s3[100];
    printf("\nPlease input the first string:");
    scanf("%s",s1);
    printf("\nPlease input the second string:");
    scanf("%s",s2);
    strsort(s1);//将字符串 s1 排序
    strsort(s2); //将字符串 s2 排序
    printf("%s\n",s1);
    printf("%s\n",s2);
    s3[0]='\0'; //字符串 s3 的第一个字符先置'\0'结束标志
    strmerge(s1,s2,s3);//将 s1 和 s2 合并，按照字母顺序排列，
    //且要删去相同字符，存入 s3 中
    printf("%s",s3);
}
```

该程序定义了两个子函数 strsort 和 strmerge。它们分别实现了将一个字符串按字母顺序排序和将两个字符串合并排序，并删去相同字符。在主函数里，先输入两个字符串 s1 和 s2，然后调用 strsort 函数对它们分别排序，然后调用 strmerge 函数将 s1 和 s2 合并，将合并后的字符串赋给字符串 s3，最后输出字符串 s3。

运行该程序，并输入：

Please input the first string: China↙

Please input the second string: Beijing↙

输出结果为：

Cahin

Begiijn

BCaeghijn

例 7-18

```
#include <stdio.h>
void strmerge(char *a,char *b,char *c)//将字符串 a,b 合并到字符串 c 中
{
  char t,*w;
  w=c;
  while((*a!='\0')&&(*b!='\0'))
  {
    //找到字符串 a,b 当前字符中较小的字符
    if(*a<*b)
    {
      t=*a;
      a++;
    }
    else if(*a>*b)
        {
          t=*b;
          b++;
        }
        else
        {
          t=*a;
          a++;
          b++;
        }
    if(*w=='\0')//开始，可直接赋值
      *w=t;
    else if(t!=*w)//如果 a,b 中较小的当前字符与 c 中当前字符不等，才赋值
      *(++w)=t; //将指针 w 向后移动一位后赋值
  }
  if(*a!='\0') //如果字符串 a 还没有结束，则将 a 的剩余部分赋给 c
    while(*a!='\0')
      if(*a!=*w)
      //判断字符串 a 中当前字符与 w 所指字符是否相等，如不等才赋值
      {
```

```
            *(++w)=*a; //赋值前要将 w 指针向后移动一位
            a++;//赋值后将指针 a 向后移动一位
        }
        else
            a++;
    if(*b!='\0') //如果字符串 b 还没有结束，则将 b 的剩余部分赋给 c
      while(*b!='\0')
        if(*b!=*w)
        {
            *(++w)=*b;
            b++;
        }
        else
            b++;
    *(++w)='\0'; //最后给字符串 c 的结尾加上'\0'标识
}
void strsort(char *s)//将字符串 s 中的字符排序
{
    int i,j,n;
    char t,*w;
    w=s; //字符指针 w 指向字符串 s 的首地址
    for(n=0;*w!='\0';n++)
    //将指针 w 向后移动，直到字符串结束，从而得到字符串长度 n
      w++;
    for(i=0;i<n-1;i++) //对字符串 s 进行排序，按字母先后顺序
      for(j=i+1;j<n;j++)
        if(s[i]>s[j])
        //如果前面的字符码大于后面的字符码，则需要交换
        {
            t=s[i]; //用字符变量 t 作中间变量，从而实现两个数组元素的交换
            s[i]=s[j];
            s[j]=t;
        }
}
void main()
{
    char s1[100],s2[100],s3[100];
    printf("\nPlease input the first string:");
    scanf("%s",s1);
    printf("\nPlease input the second string:");
    scanf("%s",s2);
    strsort(s1);//将字符串 s1 排序
```

```
    strsort(s2); //将字符串 s2 排序
    printf("%s\n",s1);
    printf("%s\n",s2);
    s3[0]='\0'; //字符串 s3 的第一个字符先置'\0'结束标志
    strmerge(s1,s2,s3);
    //将 s1 和 s2 合并，按照字母顺序排列，且要删去相同字符，存入 s3 中
    printf("%s",s3);
    return 0;
}
```

由于这里用字符数组存储字符串，所以字符串的首地址即字符数组的数组名，函数中都用到了字符串的指针（即字符数组的数组名）作参数。这样，可以在函数中直接对字符串进行操作，字符串指针的移动正是对字符数组元素的依次访问。由于字符数组的末尾要用标志'\0'，所以，程序中根据该表示来判断字符串是否结束。

函数 strsort 首先用循环的方法，得到字符串的长度，然后根据这个长度对字符串（字符数组）进行排序。排序的时候，是直接对字符数组下标访问的办法来访问每一个字符。

函数 strmerge 则是首先从两个待合并的字符串的第一个字符开始，寻找其中较小的字符放到字符变量 t 中，然后把该字符赋给第三个字符串所在的单元。由于两个待合并字符串已经是排好序的了，这样每次比较的、当前最小字符就是两个字符串中的最小字符。如果其中一个字符串结束了，就继续把另一个字符串的剩余部分按照字符不相同原则赋给第三个字符串。最后，第三个字符串加上结束标志'\0'。

本程序通过指针传递函数的参数。当需要把实参的值带回时，我们常常用指针或引用作为函数的参数，我们一定要掌握好，还要理解函数参数值传递与引用传递的区别。下面我们再举一个字符串处理的例子，问题的描述为：.通过运用指针将一个字符串反向，仍然用指针作为函数的参数，程序代码如下：

例 7-19

```
#include <stdio.h>
void revstr(char *s) //将字符串 s 反向
{
  char *p=s,c;
  while(*p) //找到串结束标记'\0'
    p++;
  p--;
  //指针回退一个字符，保证反向后的字符串有串结束标记'\0',
  //指针 p 指向串中最后
  //一个字符
  while(s<p)//当串前面的指针 s 小于串后面的指针 p 的时候，进行循环
  {
    c=*s; //交换两个指针所指向的字符，先将指针 s 指向的字符存入变量 c
    *s++=*p;//把指针 p 指向的字符赋给指针 s 指向的字符，
            // 然后将指针 s 向后移动 1 位
    *p--=c;//将 c 中存放的字符赋给指针 p 指向的地址，
            //然后将指针 p 向前移动 1 位
  }
```

```
}
void main()
{
   char a[50];
   printf("Please input the string:");
   scanf("%s",a); //输入字符串
   revstr(a);//将该字符串反向
   printf("%s",a);//输出反向后的字符串
}
```

由于数组名就是该数组的首地址，给指向该数组的指针，所以在输入函数 printf 和反向函数 revstr 中都直接使用数组名 a 作为参数。实际上是以字符指针作为参数传递的。

其中函数 revstr 实现了将一个字符串反向的功能。具体做法是用两个字符指针 s 和 p 分别指向字符串的第一个和最后一个字符，利用指针运算?quot;*"实现对所指字符的访问。通过中间变量 c 的转移，使 s 和 p 指向的两个字符交换，然后将 s 和 p 分别向字符串中间移动，再进行相同的操作。直到把所有的字符都进行了交换，从而实现了对原字符串对反向。

【本章小结】

指针和引用是 C++语言的重要内容之一，也较难掌握。本章主要学习了指针的含义、指针和地址、指针变量和指针运算、指针和函数参数、指针和数组、指针和引用等内容。

指针类型的变量是用来存放内存地址的。定义指向变量的指针变量时，应在它所指的变量的类型后面加一个"*"。指针指向数组时，常常把指针与整数作加减运算，即让指针变量加一个整数或减一个整数，但这与整数的运算并不相同，指针变量加 1，是将指针变量指向下一个元素，余类推。

在 C++语言中，函数名是该函数代码所占内存区的首地址，我们可以把函数的这个首地址(或称之为入口地址)赋予一个指针变量，此指针是所谓函数指针。然后，可用该指针调用函数，这有助于提高程序设计的灵活性。需要注意的是，函数指针与变量指针的定义方法并不相同。

在 C++中，内存的分配和释放用 new 和 delete 两个运算符。它们可以使得程序员能够根据需要，申请和释放内存，而不依赖于内存的自动分配和释放，可以大大提高编程地灵活性。但内存的申请和释放也很容易犯错误，例如，未对指针变量进行初始化或分配内存，便进行各种运算，内存的重复释放和未释放，导致内存泄露等，要十分小心。

指针和引用可以作为函数的参数进行传递，并可以提高程序的运行效率（不必复制指针指向的内容，从而接节省了内存和运行时间），要注意比较和区分引用传递与值传递的区别，指针与引用作为函数的参数与数组作为函数参数的异同，如何利用指针和引用带回函数的返回值等。

引用是指针的另一种形式，把指针搞清楚了，学习引用也不会有什么困难。但是，引用和指针的定义方法并不同，特别是，引用作为函数返回值时，很易出错，要十分小心。一般说来，引用比指针要直观，在编程时，应尽量使用引用，这有助于提高程序的可读性。

【学习目标】

通过本章的学习，掌握结构数据类型的定义和使用方法，例如，结构数组，结构变量和指针作为函数的参数等，理解链表的概念，会用结构建立链表，并能够对链表进行插入、删除等操作。

【重点与难点】

重点：结构数据类型的概念、作用和特点；结构类型变量的定义、初始化和使用方法；结构变量和指针作为函数参数。

难点：链表的概念、建立，以及链表的插入、删除等操作。

【学习方法指导】

结构数据类型与我们以前学习的数据类型，有很大的不同，要注意区分它们的异同点。例如结构变量与数组的存储有什么不同，结构变量的成员与数组元素分别如何进行访问，它们分别是如何进行初始化的等。

链表综合了结构、指针等内容，初学者可能会感到比较困难。必须把结构和指针等基本概念搞清楚，否则，不能正确理解链表结点之间指针的连接关系。为了使指针正确地指向链表结点，建立或对链表进行操作时，画出链表结构图非常有帮助。

【知识点】

结构；结构变量；数据成员；结构变量数据成员的访问方式；结构变量作参数；结构数组；指向结构的指针；链表的创建；链表的遍历；链表的查找；链表的插入；链表的删除

我们进行程序设计时，很少仅用到一些简单的数据类型，常常要将一些数据类型组合在一起。例如，假定要记录一批小孩的身高、重量、年龄和性别。可以用数组来处理：

```
const int nMaxChildren = 1000;
double heights[nMaxChildren];
double weights[nMaxChildren];
int years[nMaxChildren];
int months[nMaxChildren];
char gender[nMaxChildren];
```

如果我们要查找某一个小孩的身高、体重等，可以通过读取有相同下标的数组元素得到，如 heights[5], weights[5]等。但是，一个小孩到底有哪些相关的数据项，不是很明显。况且，不能排除要将某些数组排序，如按身高排序，在这种情况下，要获取某一个小孩的相关数据项将十分困难。

C++中，提供了结构（struct）数据类型，它能够识别一组相关的数据。

我们已经知道数组，它用来保存线性数据，但是它有许多缺点：

◇ 数组的大小是固定的，在程序运行期间是不能改变的。我们在定义数组时必须足够大，保证程序运行时不会溢出。但是，这也常常导致大量数组存储单元的浪费。

◇ 数组需要一块连续的内存空间。但当应用程序较大，数组需要的内存块也较大时，这可能会降低程序的效率。

◇ 如果在数组元素中，有较多的插入操作，则被插元素后的元素需要向后移位，这也浪费机器的运行时间。

链表也是表示线性数据最有用的方法之一，用链表保存线性数据，可以克服数组的问题。使用链表数据结构需要结构和指针作为基础，本章中，我们将学习结构和链表的有关知识。

以前章节，我们所使用的数据（整型、字符、浮点型）都是 C++预先定义的基本数据类型，这些数据类型的存储方法和运算规则是由语言本身规定，它们与机器硬件有更直接的关系。但仅用这些基本数据类型还难以描述现实世界的各种各样的客观对象之间的关系。

在实际问题中，一组数据往往具有不同的数据类型。例如，在学生登记表中，姓名应为字符型；学号可为整型或字符型；年龄应为整型；性别应为字符型；成绩可为整型或实型。由于数组还必须要求元素为相同类型，显然不能用一个数组来存放这一组数据。因为数组中各元素的类型和长度都必须一致，以便于编译系统处理。

我们可用结构数据类型来解决这个问题。为了满足程序设计的需要，C++允许我们自己定义数据类型，称之为自定义数据类型，结构是自定义数据类型中的一种，它可将多种数据类型组合在一起使用。

在定义 C++结构时，允许指定一组数据变量：

```
struct Child {
    double height;
    double weight;
    int years;
    int months;
    char gender;
};
```

这样，我们就定义了一个结构类型 Child，struct 是关键字。Child 类型包含两个 double、两个 int 和一个 char 域（或成员），它可以与 C++的基本数据类型一样地使用。（注意）结构类型定义也是一个语句，所以结尾必须有分号（;），否则，会产生编译错误。

同标准的 char、long、double 等类型一样，定义结构类型以后，便可以定义结构类型的变量，结构类型的变量在内存中占的字节数　　是其各个成员在内存中占的字节数的总和。例如：

```
Child cute;
Child kid;
Child brat;
```

也可以定义结构类型数组：

```
Child surveyset[nMaxChildren];
```

定义结构变量时，可以同时初始化它的数据成员：

```
Child example = {
    125.0, 32.4, 13, 2, 'f'
};
```

结构变量也能够在定义结构类型时定义:

```
struct Rectangle {
    int left, top;
    int width, height;
} r1, r2, r3;
```

在定义 Rectangle 类型时，同时定义了三个 Rectangle 变量：r1、r2 和 r3。不过，我们应尽可能避免这种定义方式，把类型定义和变量定义分开，否则它很容易产生错误。

访问结构变量数据成员的方式为：

结构变量名.成员名

例如，我们要检查 brat 的身高是否超过某一个限度：

```
if(brat.height > h_limit)
…
```

同样，如果要设置 cute 的体重：

```
cute.weight = 35.4;
```

在大多数的语句和表达式中，是要访问结构的数据成员。但是，在 C++中，整个结构变量的赋值也是允许的：

```
Child Tallest;
```

```
Tallest.height = 0;
for(int i = 0; i < NumChildren; i++)
if(surveyset[i].height > Tallest.height)
Tallest = surveyset[i];
```

编译器处理这样的赋值，是通过复制一个位置的内存块到另一个位置。

一个结构变量也可以是另一个结构类型的成员。例如，在例 8-1 程序中，结构 employee 的一个成员类型是 room，room 也是一个结构，它有数据成员 number、floor 和 capacity：

例 8-1

```
#include <iostream.h>
const int MAXIMUM_NAME_LENGTH= 50;
typedef char name_string[MAXIMUM_NAME_LENGTH];
struct room
{
   int number;
   int floor;
   int capacity;
};
struct employee
{
   name_string name;
   int age;
   double salary;
   room location;
};
int main()
{
   int place;
   employee first_employee;
   room room_one, room_two;
   room_one.number = 1;
   room_one.floor = 0;
   room_one.capacity =6;
   room_two.number = 2;
   room_two.floor = 3;
   room_two.capacity = 4;
   cout << "Please enter the employee's last name: ";
   cin >> first_employee.name;
   cout << "Please enter the employee's age: ";
   cin >> first_employee.age;
   cout << "Please enter the employee's current salary: ";
```

```
    cin >> first_employee.salary;
    cout << "Put in room 1 or in room 2? (type '1' or '2'): ";
    cin >> place;
    if (place == 1)
      first_employee.location = room_one;
    if (place == 2)
      first_employee.location = room_two;
    cout << endl << first_employee.name << " is now on floor ";
    cout << first_employee.location.floor << "." << endl; return 0;
}
```

上例中的表达式 first_employee.location.floor 是读取 first_employee 变量 location 成员的 floor 成员。假定运行该程序，并给出下面的输入：

Please enter the employee's last name: Miller

Please enter the employee's age: 37

Please enter the employee's current salary: 110.76

Put in room 1 or in room 2? (type '1' or '2'): 1

则输出结果如下：

Miller is now on floor 0.

上面的程序中有两个赋值语句，均为将一个 room 结构变量复制到另一个结构变量（包括三个成员）：

```
...
first_employee.location = room_one;
...
first_employee.location = room_two;
```

1．结构的定义

结构与数组不同，是不同数据类型的数据集合。结构中的不同类型的数据都是有关联的，它们被作为一个整体来看待。如同在调用函数之前要先定义函数一样,结构作为一种自定义的数据类型，在使用它之前也必须先定义。

结构类型定义的一般形式是：

```
struct 结构名
{
  数据类型标识符 1 变量名 1;
  数据类型标识符 2 变量名 2;
  ……
  数据类型标识符 n 变量名 n;
};
```

结构定义以关键字 struct 开头，"结构名"必须是 C++的有效标识符，花括号中间的部分是数据成员说明列表，它是由变量说明语句构成的一个语句序列。需要注意的是：一个结构内至少要有一个成员，每个成员也称为结构的一个域，成员的类型可以是基本数据类型，也可以是非基本数据类型。例如：

```
struct example
```

```
{
  int a;
  float b;
  double c;
  example * ptr;
}
```

在这个定义里面，example 是结构名，"int a;float b;double c;example *ptr;"四条语句组成了数据成员说明列表。即结构类型 example 中有四个成员，它们分别是整型变量 a、单精度浮点型变量 b、双精度浮点型变量 c 和指向结构 example 的指针变量 ptr。

结构类型比一般基本数据类型可以更加灵活、方便地表示实际程序设计中的复杂数据，而这种类型的使用方法与基本数据类型相似。

2.结构变量的说明

有了结构类型的定义之后，就可以定义这种类型的变量，要定义一个结构类型的变量，可以采用以下三种方法。

（1）先定义结构类型，再定义结构类型变量。

这种方式定义结构变量的一般形式是：

结构类型名 变量名 1，变量名 2，…，变量名 n；

我们还以结构类型 example 为例，下面的语句定义了两个 example 型变量 x、y：

```
example x,y;
```

在 C 语言中，定义结构类型的变量，除要结构类型名外，还要有关键字 struct。上面的语句在 C 中的定义应为：

```
struct example x,y;
```

而在 C++中，则允许不带关键字 struct。

（2）在定义结构类型的同时定义结构变量。

其一般形式为：

```
struct 结构名
{
  成员列表
}变量名列表;
```

按照这种形式，（1）中的结构变量定义可以写为：

```
struct example
{
  int a;
  float b;
  double c;
  example * ptr;
}x,y;
```

（3）直接说明结构变量

其一般形式为：

```
struct
{
  成员列表
```

}变量名列表;

这种方法与（2）中方法的区别在于：省去了结构名，而直接给出结构变量。由于它不便于使用，我们很少用这种方法定义结构变量。　　我们再看下面的例子：

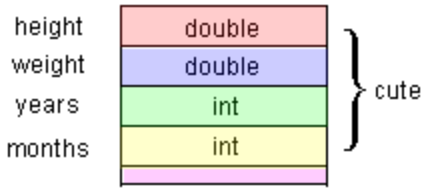
例 8-2

```
struct date
{
   int month;
   int day;
   int year;
};

struct{
   int num;
   char name[20];
   char sex;
   date birthday;
   float score;
}boy1,boy2;
```

首先定义一个称之为 date 结构类型，它由 month(月)、day(日)、year(年) 三个成员组成。 然后，又定义了一个无名结构及该结构的两个变量 boy1、boy2，它包括 num、name、sex 、birthday、score 五个成员。需要注意的是：后一个结构的成员包括 date 结构变量 birthday，这说明一个结构的成员也可以又是另一个结构的变量，即可构成嵌套的结构。此外，结构中的成员名可与程序中其它变量同名，它们互不干扰。

结构类型变量在内存中所占的存储空间是各个成员在内存中所占空间的总和。对于 Child 结构的变量 cute，具有成员"double height;double weight;int years;int months;char gender;"它在内存中所占的空间就是个两个 double、两个 int 和一个 char 类型所占的存储空间的总和。如下图所示：

图 8-1 结构 Child 型变量 cute 的内存分配示意



3．结构变量成员的访问

结构由不同的成员类型组成，通常参加运算的是结构变量中的各个成员。对结构变量的使用，包括赋值、输入、输出等运算，一般都是通过结构变量的成员来实现的。对结构变量中成员的引用形式为：

结构变量名.成员名

这样的表达式也称之为成员选择表达式，其中圆点"."称为成员运算符，它的运算优先级很高（参

见第三章）。例如 brat.height 表示对结构变量 brat 的成员 height 的引用。

对于给成员赋值语句 cute.weight=35.4，系统的操作顺序是先找到结构变量 cute，然后从 cute 所在的存储空间找到成员 weight，把 35.4 存入到 weight 分配的存储空间。由于成员运算符"."的结合性是从左到右的，因此可以引用成员类型较为复杂的结构。对于嵌套结构类型，即一个结构的成员本身是一个结构类型，可以通过嵌套引用的方法对最低级的成员进行引用，这在例 8-1 程序中可以看出来。

结构可以整体赋值，即将一个结构变量作为一个整体赋给另一个同类型的结构变量。例如："Tallest = surveyset[i];"赋值的结果是把结构变量 surveyset[i]的所有成员的值赋给结构变量 Tallest 相应的成员，该赋值等价于：

Tallest.height= surveyset[i].height;

Tallest.weight= surveyset[i].weight;

Tallest.years= surveyset[i].years;

Tallest.months= surveyset[i].months;

Tallest.gender= surveyset[i].gender;

另外，我们还可以通过结构指针来访问结构变量的成员，此时用"箭头运算符"，形式为：

指向结构的指针变量名→成员名

这在后面要讲到。

4．结构变量的初始化

结构变量可以在定义时初始化，例如：

```
struct stu /*定义结构*/
{
   int num;
   char *name;
   char sex;
   float score;
}boy2,boy1={102,"Zhangping",'M',78.5};
```

定义了结构类型 stu 的同时，也定义了结构变量 boy1 和 boy2，并对 boy1 作了初始化。

例 8-3

```
#include <iostream.h>
const int MAXIMUM_NAME_LENGTH=50;
typedef char name_string[MAXIMUM_NAME_LENGTH];
//定义最大长度为 50 的 name_string  字符串
struct room //定义结构类型 room,包括三个整型成员
{
   int number;
   int floor;
   int capacity;
};
struct employee //定义结构类型 employee,包括成员
                //name,age,salary,location
{
```

```
    name_string name;
    int age;
    double salary;
    room location;
};
int main() //主函数
{
    int place;
    employee first_employee;//声明一个 employee 型结构变量 first_employee
    room room_one, room_two; //声明 room 型结构变量 room_one 和 room_two
    room_one.number = 1; //对 room_one 各成员进行赋值
    room_one.floor = 0;
    room_one.capacity = 6;
    room_two.number = 2; //对 room_two 各成员进行赋值
    room_two.floor = 3;
    room_two.capacity = 4;
    cout << "Please enter the employee's last name: ";
    cin >> first_employee.name;
    //输入 first_employee 的成员 name(长度不超过 50 的字符串)
    cout << "Please enter the employee's age: ";
    cin >> first_employee.age; //输入 first_employee 的成员 age
    cout << "Please enter the employee's current salary: ";
    cin >> first_employee.salary;//输入 first_employee 的成员 salary
    cout << "Put in room 1 or in room 2? (type '1' or '2'): ";
    cin >> place; //输入 1 或 2，选择 room
    if (place == 1)
        first_employee.location = room_one;
    if (place == 2)
        first_employee.location = room_two;
    cout << endl << first_employee.name << " is now on floor ";
    cout << first_employee.location.floor << "." << endl;
    //输出 first_employee.name 所在的 floor
    return 0;
}
```

这个程序比较简单，主要是应用结构数据类型来表示和处理实际问题。其中用到了结构类型的定义、结构变量的定义、结构变量的 3 种引用和赋值方式。完全体现了结构类型从定义到引用的方法和规则，下面结合该程序总结一下结构类型的定义和引用赋值。

程序首先定义了常量 MAXIMUM_NAME_LENGTH 和最大长度为 MAXIMUM_NAME_LENGTH 的字符数组 name_string。然后定义了两个结构类型 room 和 employee，其中 employee 中有一个成员是 room 结构类型的，即 location。有了 room 和 emplyee 两种结构类型，就可以方便地定义这两种结构类型的变量。

在主函数中，首先定义了 employee 型结构变量 first_employee 和 room 型结构变量 room_one 和 room_two。然后对 room_one 和 room_two 赋值，用到了成员运算符"."，例如：room_one.capacity=6。

因为结构的最低级成员就是内置的数据类型，可以象内置数据类型的变量一样进行各种运算和输入输出。因此，程序用输入的方式对 employee 的成员（name、age、salary）进行赋值。并且根据输入 1 或 2 来确定 employee.location 的值。输入为 1 时，用 room_one 赋值，输入为 2 时，用 room_two 赋值。这里也用到了结构变量的整体赋值，例如：

first_employee.location=room_one

最后，对 first_employee 的数据进行输出，包括 name 和 location.floor。由于 first_employee.location 是结构 room 的成员，所以对 location 的成员 floor 要嵌套引用，即 first_emplyee.location.floor。

对于程序的输出结果，如果选择 1 作为 place 的输入，那么 first_employee.location 就是 room_one，最后输出的 first_emplyee.location.floor 实际上是 room_one.floor。由于一开始结构变量 room_one 就已被赋值，其中 room_one.floor 的值为 0，所以最后的输出为：

Miller is now on floor 0

同其它变量一样，结构类型的变量也可以作为函数的参数。例 8-4 的程序是例 8-1 的不同形式：

例 8-4

```
#include<iostream.h>
const int MAXIMUM_NAME_LENGTH=50;
typedef char name_string[MAXIMUM_NAME_LENGTH];
struct employee
{
   name_string name;
   int age;
   double salary;
};
void input_details(employee& this_employee);
void display(employee this_employee);
int main()
{
   employee first_employee;
   input_details(first_employee);
   display(first_employee);
   return 0;
}
void input_details(employee& this_employee)
{
   cout << "Please enter the employee's last name: ";
   cin >> this_employee.name;
   cout << "Please enter the employee's age: ";
   cin >> this_employee.age;
```

```
    cout << "Please enter the employee's current salary: ";
    cin >> this_employee.salary;
}
void display(employee this_employee)
{
    cout << endl;
    cout << this_employee.name << " has age " << this_employee.age;
    cout << " and salary " << this_employee.salary << ".\n\n";
}
```

下面是给出的输入和输出的例子：

```
Please enter the employee's last name: Miller
Please enter the employee's age: 37
Please enter the employee's current salary: 89.45
Miller has age 37 and salary 89.45.
```

函数的返回值也可以是结构变量。例如，在例 8-2 的 main 函数中，可以将语句：

```
input_details(first_employee);
```

替换为：

```
first_employee = input_details();
```

并将 input_details(...)函数实现作如下修改：

```
employee input_details()
{
    employee this_employee;
    cout << "Please enter the employee's last name: ";
    cin >> this_employee.name;
    cout << "Please enter the employee's age: ";
    cin >> this_employee.age;
    cout << "Please enter the employee's current salary: ";
    cin >> this_employee.salary;
    return this_employee;
}
```

函数参数有值传递和引用传递两种方式。我们从例 8-2 也可以看到：结构变量作为函数参数时可以这两种方式中的任一种方式传递，也可以作为函数的返回值。结构变量作为函数参数传递和作为函数结果返回时，坚持下面的基本规则：

◇ 对于小的结构数据，可以采用传值的方法。

◇ 对于大的结构数据，采用传地址或引用。因为占用内存大的结构变量，在复制时，对时间和空间的开销都较大。

◇ 如果需要修改结构变量数据，要传地址或引用，而传值是不能修改变量数据的。

◇ 如果要返回函数的一个局部变量的值，用 return 返回，不必考虑变量数据的大小。虽然有复制的开销，但不能返回局部变量的指针或引用，不过，可以返回用 new 申请内存的指针。

◇ 如果不需要改变所传递的参数或函数的返回结果，可以用 const 修饰形参（当形参为引用或指针时）或用 const 修饰返回类型。

结构作为函数参数传递的情况有两种，即传递结构的成员和传递整个结构。实际上，两种传递的参数都可以被看作是内置数据类型的变量或数组。对于结构的成员作为参数传递的情况例 8-2 没有用到，下面再举一简单的例子说明：

```
double addsalary(double x)
{
   x=x+x*0.05;
   return x;
}
main()
{
   ……
   first_employee.salary=addsalary(first_employee.salary);
   ……
}
```

还是利用例 8-2，程序其余部分相同，只是先定义函数 addsalary()，然后在主函数内调用函数 addsalary()，以结构变量 first_employee 的成员 salary 作为实参。我们可以看到：由于成员 salary 是 double 型的，对它的使用与对其它 double 型的变量的使用是一样的。

对于整个结构作为函数参数的情形，例 8-2 给出了很清楚的示例。我们可以看到：在主函数中分别调用了子函数 input_details()和 display()，都是以整个结构变量 first_employee 作为参数传递的。细心的读者可能会发现，两个函数虽然都是以整个结构变量为参数，但是传递的方式不同。前者 input_details()是以引用的形式传递结构变量 this_employee 的（即&this_employee），后者 display()是传值的方式传递结构变量 this_employee 的。两种传递方式根本的不同在于：前一种方式可能会改变实参结构变量的数据，后一种对结构变量的使用，不影响实参结构变量的值。

有关用结构作为参数传递的时候，到底是采用引用方式还是传值方式，右边给出了一些基本规则。

在 8.1 节我们已经知道，结构可以定义为数组，结构数组的初始化也与其它数组的初始化方法类似。下面以 8.1 节中定义的结构 Rectangle 为例，说明结构数组初始化的方法。

```
Rectangle r[3] = {{0, 0, 10, 10}, {10, 5, 10, 10}, {20, 20, 15, 10}};
```

我们也可以用结构变量初始化结构数组元素，例如：

```
Rectangle r[3];
Rectangle temp = {0, 0, 0};
for(int i = 0; i<3; i++)
r[i] = t;
```

但是，下面的初始化是错误的：

```
Rectangle temp = {0, 0, 0};
Rectangle r[3] = {t, t, t};
```

定义数组 r 时，也可以不指定数组元素的个数：

```
Rectangle r[ ] = {{…}, {…}, {…}};
```

编译器根据给出的初值的个数来确定数组元素的个数

下面我们再举一个结构数组的例子，它是例 8-2 修改版本，允许输入 10 个雇员的信息并存储在结构数组中。

例 8-5

```
#include <iostream.h>
const int MAXIMUM_NAME_LENGTH= 50;
const int NUMBER_OF_EMPLOYEES= 10;
typedef char name_string[MAXIMUM_NAME_LENGTH];
struct employee
{
   name_string name;
   int age;
   double salary;
};
void input_details(employee& this_employee);
void display(employee this_employee);
int main()
{
   employee employee_list[NUMBER_OF_EMPLOYEES];
   int count;
   for (count = 0 ; count < NUMBER_OF_EMPLOYEES; count++)
   {
      cout << "EMPLOYEE NUMBER " << (count +1) << ":\n";
      input_details(employee_list[count]);
      cout << endl;
   }
   for (count = 0 ; count < NUMBER_OF_EMPLOYEES; count++)
      display(employee_list[count]);
      return 0;
   }
void input_details(employee& this_employee)
{
   cout << "Please enter the employee's last name: ";
   cin >> this_employee.name;
   cout << "Please enter the employee's age: ";
   cin >> this_employee.age;
   cout << "Please enter the employee's current salary: ";
   cin >> this_employee.salary;
}
void display(employee this_employee)
{
   cout << this_employee.name << " has age " << this_employee.age;
   cout << " and salary " << this_employee.salary << ".\n";
}
```

从给出的程序可以看出：对结构数组的操作实际上是对每个数组元素进行操作，即对每个结构变量进行赋值和引用。需要注意的是：不能把一个结构数组元素整体进行输入输出，而相同结构类型的数组元素和变量之间可以相互赋值。

在上一节里，我们定义了结构数组。实际上，我们常常要为结构动态地分配内存，这就需要指向结构类型变量的指针，定义指向结构类型变量指针的方法与定义其它类型（如 int 型、float 型）变量指针的方法类似。我们知道：结构类型变量访问其成员要用" . "运算符，而结构类型指针访问结构成员要用"->"运算符，它的语法为：

指针变量->成员名

下面举一个例子加以说明。

例 8-7

```
#include <iostream.h>
#include <string.h>
struct Person_Info {
   char name[10];
   int age;
   long ssn;
};
void main (void)
{
   Person_Info* p1;
   p1 = new Person_Info; // 申请内存
   p1->age = 32; // 给 age 成员赋值
   strcpy(p1->name,"Bobbie"); // 给 name 成员赋值
   // 打印 p1 指向的内容
   cout << "Name: " << p1->name << endl; // 打印 name 成员
   cout << "Age: " << p1->age << endl; // 打印 age 成员
   delete p1; // 释放申请的内存
}
```

该程序的运行结果为:

Name：Bobbie

Age：32

一个指针变量当用来指向一个结构变量时，便称之为结构指针变量。结构指针变量中的值是所指向的结构变量的首地址。通过结构指针便可访问该结构变量，这与数组指针和函数指针的情况是相同的。

定义结构指针的一般形式为：

结构名 *指针变量名；

如 Child *fp 就定义了一个指向结构 Child 的指针 fp。给指针变量 fp 赋值是把结构变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。

例如，下面的语句定义了指向结构类型 employee 的指针 p，并且把结构变量 first_employee 的地址赋给了 p：

employee * p;

p=&first_employee;

表达式*p 就是对变量 first_employee 的值。

结构类型和结构变量是两个不同的概念，不能混淆。结构类型就象 int、double 等类型一样，是一种数据类型，编译系统并不对它分配内存空间。只有定义了结构类型的变量时，才对该变量分配存储空间。因此 p=&employee 这种写法是错误的，不可能去取一个结构类型的首地址。有了结构指针变量，就能方便地访问结构变量的各个成员。

如果要引用 first_employee 的成员 age，可以用下面的表达式：

（*p）.age

因为运算符"*"的优先级低于运算符"."的优先级，所以要对*p 加括号，表示取指针所指向的变量 first_employee 的值。此外，C＋＋还提供了一种结构类型指针访问结构成员的方法，就是使用运算符"－>"：

指针变量—>成员名

还用上面的例子，指向变量 first_employee 指针 p 要访问该变量的成员 age，可以用下面的表达式：

p－>age

我们再来看看下面的例子：

例 8-8

```
#include <stdio.h>
struct stu
{
   int num;
   char *name;
   char sex;
   float score;
} boy1={102,"Zhangping",'M',78.5},*pstu;
void main()
{
   pstu=&boy1;
   printf("Number=%d\nName=%s\n",boy1.num,boy1.name);
   printf("Sex=%c\nScore=%f\n\n",boy1.sex,boy1.score);
   printf("Number=%d\nName=%s\n",(*pstu).num,(*pstu).name);
   printf("Sex=%c\nScore=%f\n\n",(*pstu).sex,(*pstu).score);
   printf("Number=%d\nName=%s\n",pstu->num,pstu->name);
   printf("Sex=%c\nScore=%f\n\n",pstu->sex,pstu->score);
}
```

本程序的运行结果如下：

Number=102

Name= Zhang ping

Sex=M

Score=78.500000

Number=102

Name= Zhang ping

Sex=M

Score=78.500000

Number=102

Name= Zhang ping

Sex=M

Score=78.500000

本例定义了一个结构 stu 后，再定义了一个 stu 类型结构变量 boy1 并作了初始化赋值，一个指向 stu 结构类型的指针变量 pstu。在 main 函数中，使 pstu 指向 boy1，并在 printf 语句内用三种形式输出 boy1 的各个成员值：

结构变量.成员名

(*结构指针变量).成员名

结构指针变量->成员名

从输出结果，我们可以看到：这三种访问结构成员的方法是完全等效的。

指针变量也可以指向一个结构数组，这时结构指针变量的值是整个结构数组的首地址。结构指针变量也可指向结构数组的某一个元素，这时结构指针变量的值是该结构数组元素的首地址。

设 ps 为指向结构数组的指针变量，则 ps 也指向该结构数组的下标为 0 的元素，ps+1 指向下标为 1 的元素，ps+i 则指向下标为 I 的元素，这与其它类型的数组的情况并无不同。

结构指针的算术运算遵循指针运算的一般规则，例如：

employee list[100];

employee *pp=list;

employee 是一个结构类型，list 是大小为 100 的结构数组，结构指针 pp 存储 数组 list 的首地址，即 list[0]。表达式 pp+1 指向元素 list[1]，表达式 pp+n 的指向数组元素 list[n]。

下面是一个应用关于指向结构数组的指针的例子：

例 8-9

```
struct stu
{
   int num;
   char *name;
   char sex;
   float score;
}boy[5]={
   {101,"Zhou ping",'M',45},
   {102,"Zhang ping",'M',62.5},
   {103,"Liou fang",'F',92.5},
   {104,"Cheng ling",'F',87},
   {105,"Wang ming",'M',58},
```

```
};
void main()
{
   stu *ps;
   printf("No\tName\t\t\tSex\tScore\t\n");
   for(ps=boy;ps<boy+5;ps++)
      printf("%d\t%s\t\t%c\t%f\t\n",ps->num,ps->name,ps->sex,ps->score);
}
```

在该程序中，定义了 stu 结构类型、stu 数组 boy，并作了初始化赋值。在 main 函数内定义 ps 为指向 stu 变量的指针。在循环语句 for 的表达式 1 中，ps 指向 boy 变量，然后循环 5 次，输出 boy 数组中各成员值。

应该注意的是：一个结构指针变量虽然可以用来访问结构变量或结构数组元素的成员，但是，不能使它指向一个成员，因为结构变量与结构变量中的成员类型不同。下面的赋值是错误的：

```
ps=&boy[1].sex;
```

而只能是：

```
ps=boy;//赋予数组首地址
```

或者是：

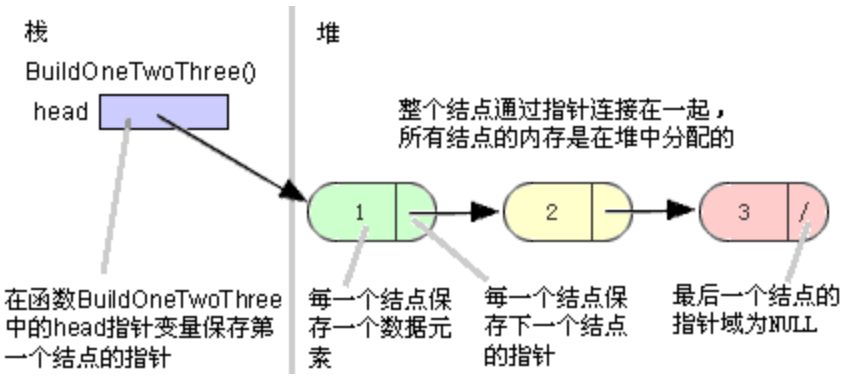
```
ps=&boy[0];//赋予 0 号元素首地址
```

例 8-3 给出了指向结构的指针的定义和用法。程序开始定义了结构类型 Person_Infor，然后在主程序中定义了指向结构的指针 p1，并用"p1=new Person_Infor;"语句动态地给 p1 分配了内存。这样，便可以给该指针指向的结构变量的内存单元进行赋值。对内存单元的访问使用了"->"运算符，如 p1->age=32，而"strcpy(p1->name,"Bobbie");"是利用了字符串的拷贝函数 strcpy( )对 p1->name 进行赋值。然后通过 p1->name 和 p1->age 实现对 p1 所指的 Person_Infor 的两个成员进行访问。最后，释放为指针所指结构分配的内存，这对于动态分配内存的变量来说是非常重要的，否则，会引起内存泄露。

我们已经知道：数组是一种数据结构，用来保存线性数据。它并不象数组那样，需要一个连续的内存块。使用链表时，每一个数组元素动态地分配内存单元，并通过指针把他们联系在一起，链表元素我们也称之为链表结点。

每一个结点包含两个域：数据域和指针域。数据域保存数据，指针域连接该结点到下一个结点。每一个结点用 new 在堆中申请内存，所以在用 delete 释放申请的内存以前，结点的内存一直存在。链表的前端有一个指向第一个结点的指针，最后一个结点的指针域不指向别的结点。假定我们要建一个有三个元素 1、2、3 的链表，该链表的示意图如左：

 **图 8-2**



上图表明：该链表由函数 BuildOneTwoThree()创建，head 指针指向链表的第一个结点。第一个结点包含一个指向第二个结点的指针，第二个结点包含一个指向第三个结点的指针，第三个结点（即最后一个结点）的指针域设置为空（即 NULL）。从 head 开始，通过各结点指针能够访问到链表的每一个元素。图左边的 head 是一个普通的局部指针变量，保存在栈中，图右边的表结点是在堆中分配的。

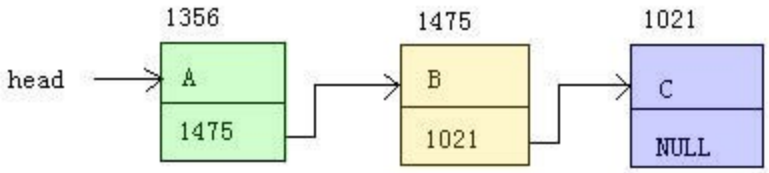
如果表结点个数为 0，我们称之为空表，此时，head 指针为 NULL。

在某一类数据集合中，依据数据成员之间的关系的不同，可把数据结构分为两大类：线性结构和非线性结构。线性结构中各个数据成员一次排列在一个线性序列中，如数组和链表等；非线性结构中，每个数据成员可能与零个或多个其它数据成员之间发生关系，如树和图。

用数组来存放数据的时候，需要事先确定数组的长度，并分配连续的内存空间。因而不能动态地分配内存，不能改变数组的大小。而链表是一种非连续存放的重要的线性数据结构，它可以对存储空间进行动态的分配。比如我们要对一个公司员工数据进行记录，由于公司的人事可能会经常地变动，涉及到员工人数的增减。如果用数组来存放员工的记录，当员工人数发生变化的时候，数组的大小很难做到相应的变化。如果人数减少，数组会有剩余空间，但人数增加，原来的数组大小可能不够，就必须重新开数组。这样就会产生浪费空间或增加维护负担等问题。如果改用链表来存放员工的数据，就不会产生这些问题，因为链表对存储空间的分配是动态的。

链表是由一些列的结点组成，每一个结点包含数据和指向下一结点的指针。每一个结点占用一块存储单元，当要在链表中增加一个结点时，可动态地为该结点分配一个存储单元；当要在链表中删除一个结点时，也可释放该结点的存储单元。下图是一个简单的链表示意图：

图 8-3



该链表由 3 个结点组成，其中每个结点都分为两个域，一个是数据域（即图中的大写字母 A、B、C），存放各种实际的数据，如学号 num、姓名 name、性别 sex 和成绩 score 等。另一个域为指针域，存放下一结点的首地址。比如第一个结点的指针域存放的地址是 1475，就是第二个结点的首地址。链表中的每一个结点都是同一种结构类型。

另外，每个链表都有一个头指针 head，它存放的是链表头结点的首地址。我们就可以通过链表的 head 指针找到链表的头结点，然后通过该结点的指针域找到一个结点的首地址，从而可以找到该链表的所有结

点。需要注意的是：最后一个结点的指针域为 NULL，据此，我们可以判断一个链表是否结束。下面是一个学生结构的实例：

```
struct stu
{
  int num;
  int score;
  stu *next;
}
```

前两个成员组成数据域，后一个成员 next 是指针域，它是一个指向 stu 类型结构的指针变量。

对链表的主要操作有以下几种：

1. 建立链表；
2. 结构的查找与输出；
3. 插入一个结点；
4. 删除一个结点；

下面对每一种操作具体进行讲解。

8.5.1 链表的创建

为了创建图 8-3 所示链表，我们需要定义下面的结构：

```
struct node {
  int data;
  node* next;
};
```

创建图 8-3 所示链表的函数 BuildOneTwoThree()代码如下：

例 8-10

```
node* BuildOneTwoThree()
{
  node* head = NULL;
  node* second = NULL;
  node* third = NULL;
  head = new node; // 在堆中申请三个结点
  second = new node;
  third = new node;
  head->data = 1; // 设置第一个结点
  head->next = second;
  second->data = 2; // 设置第二个结点
  second->next = third;
  third->data = 3; // 设置第三个结点
  third->next = NULL;
  return head;
}
```

BuildOneTwoThree()仅仅是创建一个仅有三个结点的链表实例，不是创建链表的一般方法。最好的方法是有一个函数，能够向链表中增加新的结点。假定该函数为 Push()，则其代码如下：

```
void Push(node** headRef, int data)
{
  node* newNode = new node;
  newNode->data = data;
  newNode->next = *headRef; // *headRef 是实际表头
  *headRef = newNode;
}
```

下面是调用该函数建立链表的一个例子:

```
void main()
{
  node* head = NULL; //空表
  Push(&head, 3); // 注意&运算符
  Push(&head, 2);
  Push(&head, 1);
  Push(&head, 13);
  // 表 head 为 {13, 1, 2, 3}
}
```

需要注意的是：如果 Push()函数写成下面的形式是错误的：

```
void Push(node* head, int data)
{
  node* newNode = new node;
  newNode->data = data;
  newNode->next = head;
  head = newNode;
}
```

这是因为 newNode 是建在栈中的局部指针变量，函数 Push()返回后，该指针变量被释放，不能通过 head 参数带回到调用函数，要把 head 带回到调用函数，要把它定义为指向指针的指针。其实，为了带回 head 参数，我们也可以把 head 参数定义为对指针的引用：

```
void Push(node*& head, int data)
{
  node* newNode = new node;
  newNode->data = data;
  newNode->next = head; // 在 head 前不需要额外的*
  head = newNode;
}
```

调用 Push()函数建立链表的函数则为：

```
void main()
{
  node* head = NULL;
  Push(head, 3); //在 head 前不需要额外的&
```

```
    Push(head, 2);
    Push(head, 1);
    Push(head, 13);
    // 链表 head 为 {13, 1, 2, 3}
}
```

调用 Push()函数时应该注意：在调用函数中，如果传递的链表是一个空链表（链表创建之前），应将其头指针设置为空（NULL）。

上面我们已经看到，当我们传入 1、2、3、4、5 个结点时，链表中结点的次序是 5、4、3、2、1，即新增加的结点放在原链表的头上。那么，我们能不能建立一个链表，使新增加的结点放在原链表的末尾？其实，只要我们把调用函数修改为下面的形式，就可以达到目的：

例 8-11

```
void main()
{
   node* head = NULL;
   node* tail;
   int i;

   Push(&head, 1);
   tail = head;

   for (i=2; i<6; i++) {
      Push(&(tail->next), i); // 在 tail->next 增加结点
      tail = tail->next; // advance tail to point to last node
   }
   // 链表 head 为{1, 2, 3, 4, 5}
}
```

该函数调用了形参为指向指针的指针的 Push()函数。

在创建链表之前，先要定义一个结构类型作为链表的结点类型。函数 BuildOneTwoThree()创建链表的过程是：首先定义三个指向结点的指针 head、second、third，并赋初值为 NULL。然后，开辟三个结点的存储空间，每一个指针指向各存储空间的首地址。接下来，给每一个结点的数据域和指针域赋值，使得 head 结点的指针 next 指向 second 结点，second 结点指向 third 结点，third 结点的指针 next 赋值为 NULL。最后返回头结点 head。这样就得到了三个连在一起的结点 head、second、third 组成的链表。

函数"Push(node **headRef,int data)"实现了向已有链表中添加结点的功能。首先定义一个指向结点的指针，并分配一个结点的存储空间。然后对新结点赋值，使其指针域指向原链表的表头结点，并且以新的结点作为新的表头。该函数实际上是向表头添加结点的办法实现链表结点的插入，然后在主函数中调用该函数就可以创建链表了。 接下来，书中给出了函数"Push(node *&headRef,int data)"的定义和用该函数实现链表创建的方法，它是用对结点指针的引用作为参数的。如果直接用指针作参数，不能将修改后的指针值带回到调用函数。

最后，给出了一个从链表尾插入结点的方法，创建链表的函数。具体做法是：首先建立一个头结点 head，

然后每次以当前结点的指针域为参数，创建并加入下一个结点，其结果就是每次在链表尾加入一个新的结点，并让该结点的指针域为 NULL。

例 8-12

```
void main()
{
   node* head = NULL;
   node* tail;
   int i;
   Push(&head, 1);//创建头结点
   tail = head; //使 tail 和 head 指向同一个结点

   for (i=2; i<6; i++) {
      Push(&(tail->next), i); //  在 tail->next 增加结点
      tail = tail->next; // advance tail to point to last node，
      //tail 指向当前结点的下一个结点,实际为 NULL
   }
   //  链表 head 为{1, 2, 3, 4, 5}
}
```

8.5.2 链表的遍历和查找

下面我们看一个简单的链表操作函数 Length()，该函数计算链表中结点的个数。

```
//给定链表的头指针，计算并返回链表结点的个数
int Length(node* head)
{
   node* current = head;
   int count = 0;
   while (current != NULL) {
   count++;
   current = current->next;
}
return count;
}
```

计算链表结点个数时，该函数使用了循环语句，这也是对链表操作中常用的方法。上面的函数是用 while 循环语句，也可以用下面的 for 循环语句替代：

```
for (current = head; current != NULL; current = current->next)
{ … }
```

Length()函数有下面几个特点：

（1） current 是一个局部指针变量，它最初与 head 指向相同的结点。当函数返回后，它被自动释放，但链表结点依然在堆内存中。

（2） 能够适应空链表的情形，此时 current=head=NULL，循环体中的语句不会执行。

（3） 循环体中的语句"current = current->next;"，使局部指针 current 移向链表的下一个结点。当到达最后一个结点时，执行 "current = current->next; "语句后，current 为 NULL。

下面的程序是调用 Length()函数的实例，先调用 BuildOneTwoThree()函数创建链表，然后调用 Length()函数计算链表结点的个数：

```
void LengthTest()
{
  node* myList = BuildOneTwoThree();
  int len = Length(myList);
          // len 的结果为 3
}
```

我们可以用与 Length()函数类似的方法查找链表中的某一个结点。

```
//给定链表的头指针和待查结点数据，返回查到的结点的指针
node* Find(node* head，int data)
{
  node* current = head;
  while (current!=NULL)
  if(current->data == data) break;
  else current = current->next;
  return current;
}
```

Find()函数的功能是：输入链表头结点 head 和待查结点数据 data，如果某一个结点的数据与 data 相等，则返回该结点的指针；如果查完每一个结点也没有找到数据与 data 相等的结点，则返回空指针。

需要注意的是：Find 函数也可以写成下面的形式。

```
//给定链表的头指针和待查结点数据，返回查到的结点的指针
node* Find(node* head，int data)
{
  node* current = head;
  while (current!=NULL&& current->data == data)
  current = current->next;
  return current;
}
```

但把 while 的条件"current!=NULL&& current->data == data"写成" current->data == data &&current!=NULL"的形式，则 Find 函数可能会出现运行错误。这是因为：如果链表的最后一个结点，仍然不是要找的结点，则到下一次循环时 current 为 NULL，再进行条件判断时，前者 current!=NULL 为真，而不再作 current->data == data 的判断，循环便结束；而后者在作 current->data == data 判断时就会导致程序崩溃

由于链表具有的特殊结构，我们在对链表中的结点进行访问和查找的时候，必须从链表的头结点开始，按照链表结点指针域所指的顺序逐个查找结点，直到找到为止。

首先，我们来看看计算链表结点个数的函数 Length()是如何实现的：

```
int Length(node* head) //以指向链表头结点的指针作为参数
{
  node* current = head; //定义一个结点指针，并使之指向链表头结点
  int count = 0; //计数器变量，初值为 0
```

```
    while (current != NULL) { //如果未到链表尾，继续循环
    count++; //计数器加 1
    current = current->next; //指针 current 指向下一个结点
  }
  return count; //返回计数值，即链表长度
  }
```

该函数就利用了链表结点的特点，即各结点通过指针依次链接起来，从头结点开始，使一个结点的指针依次指向下一个结点，同时计数器加 1。这样，就实现了对链表结点个数的统计。Length( )函数对链表结点依次访问的过程，实际上就是对链表的一次遍历，得到所需要的信息。针对链表的这种访问方式，可以打个简单的比方：比如有一队小朋友手拉手站着，现在我们要对小朋友的人数进行统计。首先设定人数值为 0，这时，我们只需要找出站在队首的小朋友，通过他们互相拉着的手找到下一个小朋友，同时每找到下一个小朋友就对计数器加 1，直到找到最后一个小朋友为止，也就得到了该队小朋友的总人数。

为了对链表的一个结点数据进行修改或者需要读取某个结点的一些数据，就要用到链表的查找。我们可以参考链表遍历的方法，对链表的结点逐个访问，判断该结点的数据是否是我们要找的数据值。若是，则查找成功，返回指向该结点的指针；若不是，则继续判断下一个结点，直到到达链表尾为止。若查找失败，则返回 NULL：

```
  node* Find(node* head，int data) //以链表头指针和待插数据为参数
  {
    node* current = head; //定义指针变量，指向头指针
    while (current!=NULL) //如果没有到达链表尾，则继续查找
    if(current->data == data) break;
    //如果该结点数据等于待查数据，则查找成功，跳出循环
    else current = current->next; //否则继续指向下一个结点
    return current; //返回当前指针
  }
```

8.5.3 链表的插入和删除

链表和数组都是保存线性数据，前者的优点在于能够方便地进行数据元素的插入和删除，链表结点的插入和删除操作是链表很常用的操作。在一个链表中插入一个结点或删除一个结点，需要知道待插入结点位置及前一个结点位置的指针，然后改变链表中结点的链接关系。

链表的两个指针可以根据实际情况确定，例如，假定有一个链表，其头指针为 head，结点数据是由小到大排序的。现要在该链表中插入一个结点 position，继续保持链表的排序关系，两个指针就可以通过搜索原链表结点得到，其代码如左：

例 8-13

```
void Insert(node* &head，node* position)
{
  node * before, *current;
  before = current = head;
  //搜索链表，确定指针 before 和 current
  while (current!=NULL)
    if(current->data >= position ->data) break;
    else{
```

```
            before = current;
            current = current->next;
        }
    //插入结点 position
        if(current == head){ //插在链表头
            position ->next = head;
            head = position;
        }
        else{ //插在链表中间
            before ->next = position;
            position ->next = current;
        }
}
```

在链表中插入结点，就是改变链表相关结点之间的链接关系（参见图 8-3、图 8-4），但要处理结点插在链表头的情形。表头指针 head 定义为引用，是为了 Insert()函数返回后,能将 head 值带回到调用函数。

与链表中插入一个结点类似，要在链表中删除一个结点，也要先搜索原链表，找到要删除的结点指针及前一个结点的指针。假定有一个链表，其头指针为 head。现要在该链表中删除一个数据域为 data 的结点，其代码如下：

例 8-14

```
void Delete(node* &head，int data)
{
  node * before, *current;
  before = current = head;
  //搜索链表，确定指针 before 和 current
  while (current!=NULL)
    if(current->data == data) break;
    else{
      before = current;
      current = current->next;
    }
    //删除数据域为 data 的结点
    if(current!=0){
      if(current == head){ //删除链表头结点
        head = current ->next;
        delete current;
        }
      else{ //删除链表中间结点
        before ->next = current ->next;
        delete current;
```

```
        }
    }
}
```

与在链表中插入结点类似，要删除链表中的某一个结点，也是改变链表相关结点之间的链接关系（参见图 8-5、图 8-6），但要处理结点插在链表头的情形。表头指针 head 定义为引用，也是为了 Delete() 函数返回后，能将 head 值带回到调用函数。

需要注意的是：链表结点删除后，应用 delete 释放该结点的内存，否则造成该结点内存泄露。如果链表为空或没有找到要删除的结点时，原链表保持不变。

链表能够方便地实现结点的插入和删除操作，这也是链表结构具有动态分配存储空间的体现，也是它优于数组的地方之一。还是举小朋友排队的例子来说明链表的插入和删除是怎样实现的。在这个比喻里面，每一个小朋友相当于一个结点，一个小朋友的手拉着另一个小朋友的手，相当于一个结点的指针域指向下一个结点。假设现有一对按大小个排好队的小朋友，又来一个小朋友需要加入该队列。这时候，就需要从原来队列里面的第一个小朋友开始，按照他们的身高找到新来小朋友应该站的位置（前一个小朋友的身高比他矮，后一个小朋友的身高比他高）。然后，把这两个小朋友的手分开，让前一个小朋友的手该拉着新来小朋友的一只手，新来小朋友的另一只手拉着后一个小朋友的一只手。这样，新来的小朋友就被插入到这个队伍里面了，并且这个队伍的小朋友还是按照身高顺序排列的。特别地，如果新来小朋友最矮，他只需要站在队伍的开头，并且让他的一只手拉着原来站在对头的小朋友的手就行了。

实际链表的插入操作也就可以类似地实现了，具体程序如下：

例 8-15

```
void Insert(node* &head，node* position)
{
  node * before, *current; //定义两个结点型指针 before,current
  before = current = head; //为这两个指针赋值为链表头指针

  //搜索链表，确定指针 before 和 current
  while (current!=NULL)//如果没有到链表尾
  {
    if(current->data >= position ->data) break;
    //因为链表数据从小到大排序，所以如果当前结点的数据
    // 大于待插结点数据，就认为以找到插入位置，跳出循环
    else{//否则，继续查找
      before = current;//before 指向 current 所指结点
      current = current->next;
      //current 指向 current 所指结点的下一个结点
    }
  }
  //插入结点 position
  if(current == head){ //插在链表头
    position ->next = head;//待插结点指针指向原来的表头
```

```
        head = position;//新的表头指针指向待插结点
    }
    else{ //插在链表中间
        before ->next = position; //前一结点指向待插结点
        position ->next = current; //待插结点指向当前结点
    }
}
```

类似地，我们也可以对链表结点进行删除操作，方法就是：首先遍历链表，找到被删结点。如果该结点位于表头，则将表头指针指向该结点的下一个结点；如果该结点位于其它位置，则将该结点的前一结点直接指向该结点的下一结点。另外，需要注意的就是：如果该结点被删除，则要用 delete 操作释放该结点所占用的内存空间，否则会造成内存泄漏。

例 8-16

```
void Delete(node* &head，int data)
//删除表头指针为 head 的链表中数据为 data 的结点
{
    node * before, *current;
    before = current = head;
    //搜索链表，确定指针 before 和 current
    while (current!=NULL)
    if(current->data == data) break;//找到被删结点，跳出循环
    else{//继续查找
        before = current;
        current = current->next;
    }
    //删除数据域为 data 的结点
    if(current!=0){
        if(current == head){ //删除链表头结点
            head = current ->next; //表头指针指向头结点的下一个结点
            delete current; //释放被删结点
            }
        else{ //删除链表中间结点
            before ->next = current ->next;
            //被删结点的前一结点直接指向被删结点的下一结点
            delete current;//释放被删结点
        }
    }
}
```

定义一个多边形结构：

```
struct polygon
```

```
{
  int n;
  int *x，*y；
  polygon *next；
}
```

1. 建立该结构的链表；
2. 显示链表的各个结点数据；
3. 编写一个函数，删除链表中的所有结点。

前面已经介绍了链表的创建、遍历和删除的方法，解决本题应该并不困难。要显示链表各结点的数据，就是要把各结点找到，然后把该结点的的每一个 x、y 坐标打印出来。不过，与我们前面介绍的链表的也有不同的地方：就是该链表的结点数据是指针。要在链表结点中存入数据，必须先动态分配存储数据的内存单元；要删除链表中的各个结点，必须先释放结点数据的内存单元，否则会造成内存泄露。

程序：

根据上面的分析，我们不难写出下面的程序代码。其中，create 函数是创建链表，每输入一个结点的数据，就把该结点加入到链表当中，它返回创建的链表的头指针，结点数据包括：多边形顶点数，各顶点的纵横坐标，当多边形顶点数为 0 时，链表创建结束。disp 函数显示结点的数据，del 函数删除链表的各个结点。需要注意的是：要先释放结点数据内存，再删除结点，如果在释放结点数据内存单元之前删除结点，则无法找到结点数据内存单元的地址，也就无法释放数据的内存单元。

例 8-17

```
#include "iostream.h"
#include "iomanip.h"
struct polygon
{
  int n;
  int *x;
  int *y;
  polygon *next;
};
void Push(polygon*& head, int n)
{
  polygon* newNode = new polygon;
  newNode = new polygon;
  newNode->next=NULL;
  newNode->x = new int[n];
  newNode->y = new int[n];
  newNode->n=n;
  for(int i=0;i<=n-1;i++){
    cout<<"请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：";
    cin>>newNode->x[i]>>newNode->y[i];
  }
```

```
    newNode->next = head; //  在 head 前不需要额外的*
    head = newNode;
}
polygon *create()
{
    polygon* head = NULL;
    polygon* tail;
    int n;
    cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
    cin>>n;
    if(n==0) return head;
    Push(head, n);
    tail = head;
    cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
    cin>>n;
    while(n!=0)
    {
        Push(tail->next, n); //  在 tail->next 增加结点
        tail = tail->next; // advance tail to point to last node
        cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
        cin>>n;
    }
    return head;
}
void disp(polygon *head)
{
    int i,No=1;
    cout<<setw(10)<<"x"<<setw(6)<<"y"<<endl;
    while(head!=NULL)
    {
        cout<<"第"<<No<<"结点："<<endl;
        for(i=0;i<=head->n-1;i++)
        cout<<setw(10)<<head->x[i]<<setw(6)<<head->y[i]<<endl;
        No++;
        head=head->next;
    }//Match while statement
}
void del(polygon *head)
{
    polygon *p;
    while(head!=NULL)
    {
```

```
        p=head;
        head=head->next;
        delete p->x;delete p->y;
        delete p;
        }//Match while statement
    }
    void main()
    {
      polygon *head;
      head=create();
      disp(head);
      del(head);
    }
```

编译、运行上面的程序，并输入：

请输入多边形顶点的个数(顶点个数为 0 时结束)：3↙

请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：23 56↙

请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：34 12↙

请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：78 45↙

请输入多边形顶点的个数(顶点个数为 0 时结束)：3↙

请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：26 90↙

请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：38 41↙

请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：29 65↙

请输入多边形各顶点 x、y 坐标，坐标值之间用空格分隔：314 78↙

运行结果如下：

x y

第 1 个结点：

23 56

34 12

78 45

第 2 个结点：

78 45

38 41

29 65

314 78

我们也可以把 create 函数改写成下面的形式，不难验证，其效果与上面的 create 相同。

```
polygon *create()
{
  polygon* head = NULL;
  polygon* tail;
  int n;
  cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
  cin>>n;
```

```
if(n==0) return head;
Push(head, n);
tail = head;
cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
cin>>n;
while(n!=0)
{
   Push(tail->next, n); //  在 tail->next 增加结点
   tail = tail->next; // advance tail to point to last node
   cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
   cin>>n;
}
return head;
```

}**程序举例**

书中的实例程序定义了一个多边形结点类型，并实现了多边形链表的创建和删除，下面给出了该程序的详细的注释。

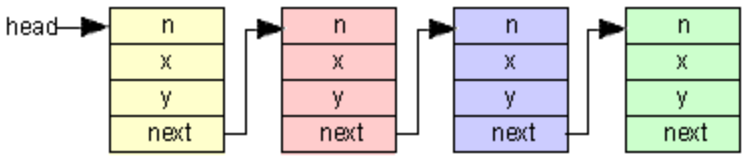
例 8-18

```
#include "iostream.h"
struct polygon//定义结构 polygon，作为链表结点类型
{
   int n; //多边形顶点数
   int *x; //多边形各顶点的横坐标数组
   int *y;//多边形各顶点的纵坐标数组
   polygon *next; //指向结构 polygon 的指针
};
polygon *create() //创建链表
{
   struct polygon *p1,*p2,*head;
   int i,n;
   head=NULL;
   scanf("%d",&n);
   if(n==0) return head;
   while(n!=0) //如果输入的 n 不为 0，创建各多边形结点，组成链表
   {
      p1=new polygon;//创建一个结点的空间
      p1->next=NULL;
      p1->x=new int[n];
      //开 n 个 int 型的内存空间，存放各定点横坐标
      p1->y=new int[n];
      //开 n 个 int 型的内存空间，存放各定点纵坐标
      p1->n=n;
```

```
    for(i=0;i<=n-1;i++)
    scanf("%d%d", &(p1->x[i]),(&p1->y[i]));//输入各点坐标
    if(head==NULL) head=p2=p1;
    else{ //将链表尾和新结点链接起来
    p2->next=p1;
    p2=p1;
    }
    scanf("%d",&n);//输入 n 值
}//Match while statement
return head;//返回链表头指针
}
void del(head)//将整个链表删除
struct polygon *head;
{
    struct polygon *p;
    while(head!=NULL)
    {
        p=head;
        head=head->next;//链表头指针指向下一个结点
        free(p->x);free(p->y);//先释放各点坐标所占空间
        free(p);//再释放整个结点所占空间
    }//Match while statement
}
void main()
{
    struct polygon *head;//声明一个指向结构 polygon 的指针变量 head
    head=create();//创建多边形链表，返回链表头指针 head
    del(head);//删除该多边形链表
}
```

这样的结点组成的链表可以表示成下面的图形：

图 8-7



链表创建函数 creat 首先需要用户输入一个多边形的顶点数 n，只要 n 不为 0，就创建一个 n 边形的结点。让用户输入各顶点坐标，并且把该结点链接到原链表的末尾。最后返回链表头指针 head。

链表删除函数 del 则实现了从链表头开始，逐个删除链表的结点，从而将整个链表删除。判断删除结束的标识为链表头指针为空，因为每删除一个结点，就将该链表的头指针指向下一个结点，直到全部链表

结点被删除。另外，每次删除一个链表结点的时候，要先释放该链表结点中多边形顶点坐标所占的内存空间，再释放整个链表结点所占的地址空间。

书中提供了两个 create 函数，另一个 create 函数的注释如下：

例 8-19

```
void Push(polygon*& head, int n)
{
  polygon* newNode = new polygon;
  newNode = new polygon; //开一个新结点
  newNode->next=NULL; //新结点指针域初始化为 0
  newNode->x = new int[n];//为新结点的数据分配内存
  newNode->y = new int[n];
  newNode->n=n;
  for(int i=0;i<=n-1;i++){//输入个结点的坐标
    cout<<"请输入多边形顶点 x、y 坐标，坐标值之间用空格分隔：";
    cin>>newNode->x[i]>>newNode->y[i];
  }
  newNode->next = head; //  新结点插入链表头部
  head = newNode; //修改头指针
}
polygon *create()
{
  polygon* head = NULL;
  polygon* tail;
  int n;
  cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
  cin>>n;
  if(n==0) return head;
  Push(head, n);//创建链表头结点
  tail = head;//链表尾指针指向链表头
  cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
  cin>>n;
  while(n!=0)//创建新的链表结点，直至 n 为 0 为止
  {
    Push(tail->next, n); //  在 tail->next 增加结点
    tail = tail->next; //尾结点后移
    cout<<"请输入多边形顶点的个数(顶点个数为 0 时结束)：";
    cin>>n;
  }
  return head;
}
```

结构是建立链表的基础，为了更好地掌握链表的用法，这里再举一个结构的例子。问题的描述是：说明一个结构类型，包含用户的姓名（6 个字符的字符串）和电话号码（8 个字符的字符串）。该函数读入 N 位用户的数据存入结构数组中，并且实现了对数组数据按字典顺序排序和输出所有数据。

我们直接写出程序代码如下，详细的注释已夹在程序代码中间：

例 8-20

```
#include "stdio.h"
#include "string.h"
#define N 5
struct user //定义结构类型 user
{
   char name[7];//长度为 7 的字符数组 name
   char tel[9]; //长度为 9 的字符数组 tel
};
void getstr(struct user *p,int n)
//输入 n 个用户的数据，存入 p 指向的结构体数组中
{
   int i;
   printf("\nPlease input %d's users' data:",n);
   for(i=0;i<n;i++)
   scanf("%s%s",p[i].name,p[i].tel);//输入每个用户的数据
}
void sortstr(struct user *p,int n)
//对 p 所指向的结构数组各元素进行排序,以 name 作为关键字
{
   int i,j,k;
   struct user t;
   for(j=0;j<n-1;j++) //选择法排序
   {
      k=j;
      for(i=j+1;i<n;i++)
      if(strcmp(p[i].name,p[k].name)<0)
      //如果后面的元素的用户名小于前面元素的用户名，则需要交换。
      k=i;
      strcpy(t.name,p[k].name);//用字符串拷贝函数交换数组元素的成员值
      strcpy(t.tel,p[k].tel);
      strcpy(p[k].name,p[j].name);
      strcpy(p[k].tel,p[j].tel);
      strcpy(p[j].name,t.name);
      strcpy(p[j].tel,t.tel);
   }
```

```
}
void outstr(struct user *p,int n) //输出 p 指向的结构数组的各元素
{
   int i;
   for(i=0;i<n;i++)
   printf("%s %s\n",p[i].name,p[i].tel);
}
void main()
{
   user w[N];//声明结构 user 的数组 w
   getstr(w,N);//读如 N 位用户的数据，放入数组 w 中
   sortstr(w,N);//将 w 数组中 N 位用户的数据进行排序
   outstr(w,N);//输出这 N 位用户的数据
}
```

这里需要说明的是，由于字符串总是以'\0'作为结束标志，因而为了使 name 中能存放 6 个字符，就应当将其长度定义成 7；同样的，tel 的长度应定义为 9。

程序中用到的 N 是一个符号常量，可在程序开头使用 define 命令给出具体定义。

由于在主函数中调用的三个函数都提供了相同的实参（结构数组 w 和符号常量 N），因而在定义三个函数的时候相应的形参也都应是一致的。当一维数组的数组名作实参时，对应的形参可定义为相同类型的一维数组或类型相同的指针变量。程序中采取后一种方式，即结构类型 user 的指针变量 p。

调用函数时，通过虚实结合，使指针变量 p 指向结构数组 w 的首地址，整型变量 n 接收符号常量 N 的值。在 getstr 函数中，读入 n 位用户数据是通过 for 循环实现的，并且通过指针变量 p 来引用每位用户的姓名域和电话域，引用的方式为：p[i].name 和 p[i].tel。所以读入 n 位用户数据的过程可以用 for 语句实现如下：

```
for(i=0;i<n;i++)
scanf("%s%s",p[i].name,p[i].tel);
```

在 sortstr 函数中，采用了选择排序的方法对 n 位用户的 name 字符串进行排序。这里要注意到，由于 p[i].name 是一个字符串，所以比较两个字符串（p[i].name 和 p[k].name）的大小的时候，一定要用 strcmp 函数，而不能写成 p[i].name<p[k].name。同样，当进行字符串赋值的时候，也应该使用 strcpy 函数来实现，而不能直接写成的 p[i].name=p[k].name。的形式。

对于 outstr 函数，则是简单地用 for 语句将数组每个元素输出，包括每个 user 结构的 name 域和 tel 域。

【本章小结】

结构是自定义数据类型中的一种，它可将多种数据类型组合在一起使用，方便了程序对一些复杂数据的处理。在程序中，定义结构变量以前，必须先进行结构类型的定义。定义一个结构类型的变量，我们介绍了三种不同的方法。

访问结构数据的成员，可以用成员运算符有两个：". "和"->"。如果是结构变量，用". "运算符，如果是结构指针，用"->"运算符。

同样，结构型数据也能作为函数参数，且有值传递和引用传递两种。结构型数据，我们通常采取引用传递方式，这可以避免结构数据复制所带来的开销。结构也可定义为数组，结构数组的初始化与其它类型数组的初始化方法也类似。

在介绍了结构数据类型后，我们介绍了链表，链表综合了结构和指针的有关内容。初学者对链表的学习可能感到困难。但是，如果搞清楚了链表，有助于加深对指针、结构等内容的理解和掌握。

我们用 C++进行编程的时候，可以在源程序中包括一些编译命令，以告诉编译器对源程序如何进行编译。这些命令包括：宏定义、文件包含和条件编译，由于这些命令是在程序编译的时候被执行的，也就是说，在源程序编译以前，先处理这些编译命令，所以，我们也把它们称之为编译预处理，本章将对这方面的内容加以介绍。

实际上，编译预处理命令不能算是 C++语言的一部分，但它扩展了 C++程序设计的能力，合理地使用编译预处理功能，可以使得编写的程序便于阅读、修改、 移植和调试。

预处理命令共同的语法规则如下：

◇ 所有的预处理命令在程序中都是以"#"来引导 如"#include "stdio.h""。

◇ 每一条预处理命令必须单独占用一行，如"#include "stdio.h" #include <stdlib.h>" 是不允许的。

◇ 预处理命令后不加分号，如"#include "stdio.h";" 是非法的。

◇ 预处理命令一行写不下，可以续行，但需要加续行符"\"。

下面我们对宏定义、文件包含和条件编译三种预处理命令的用法分别进行介绍。

C++提供的编译预处理的功能，是它与其它许多编程语言的重要区别之一。它允许在源程序中使用几种特殊的命令（它们不是一般的 C++语句）。编译系统对程序进行编译之前，先对程序中这些特殊的命令进行"预处理"。然后再进行编译处理，以得到目标代码。

编译预处理命令扩展了 C++程序设计的能力，合理地使用编译预处理功能，可以使得编写的程序便于阅读、修改、 移植和调试。

C++提供的编译预处理功能主要有以下三种：

（一） 宏定义

（二） 文件包含

（三） 条件编译

宏定义命令将一个标识符定义为一个字符串。如：

#define CUBE_THREE 3*3*3

#表示这是一条预处理命令，define 为关键字，CUBE_THREE（标识符）称为宏名，也简称为宏，3*3*3 是被定义的字符串。这样， CUBE_THREE 就代表字符串 3*3*3。当源程序被编译的时候，遇到标识符 CUBE_THREE，均以字符串 3*3*3 进行替换。由于宏定义命令是用于字符串的替换，我们也常把宏名用指定的字符串进行替换的过程，称之为宏替换。

下面是宏定义的一个例子：

例 9-1

```
#include <iostream.h>
//将系统输入输出流头文件包含进来，如果没/有这一行语句，
//用 VC 编译器进行编译的//话，会出现下面的错误提示：
//error C2065: 'cout' : undeclared identifier
#define CUBE_THREE 3*3*3
//用简单宏定义定义了一个符号常量"CUBE_THREE"
void main()
```

```
{
  int a;
  a=CUBE_THREE;
  //将该宏的值赋给 a，实际上是将该宏所定义的符号常量的值赋给 a，
  //因此在编译运//行到该语句的时候，"a=CUBE_THREE;"等价于"a=3*3*3;"
  cout<<"a is"<<a<<endl;
  //输出 a 的值，观察宏替换产生的结果
}
```

编译并运行该程序，结果如下：

a is 27

因为经过宏替换后，语句"a=CUBE_THREE"等价于"a=3*3*3;"。

宏替换的功能为我们编程时可以带来一些方便。因为，如果在程序中，某一个常量出现较多，就可以为该常量定义一个宏。这样，假定我们要修改该常量时，就不必在程序中对该常量用手工一个个地查找、修改，而只要修改其宏定义即可。例如，如果在一个程序里，3 的立方出现的次数比较多，我们就可以把它定义成上面的宏的形式。假定现在需要把 3 的立方修改成 3 的 4 次方，只要修改宏定义即可：

#define CUBE_THREE 3*3*3*3

宏定义命令通常有两种格式：一种是简单的宏定义，另一种是带参数的宏定义。

9.1.1 简单的宏定义

上面的实例就是一个简单的宏定义，简单宏定义的一般形式如下：

#define <宏名> <字符串>

其中， define 是宏定义命令的关键字，<宏名>是一个标识符，<字符串>可以是常数、表达式、格式串等。

在程序被编译的时侯，如果遇到宏名，先将宏名用指定的字符串替换，然后再进行编译。

下面是另一个包含简单的宏定义的程序实例：

例 9-2

```
#include<iostream.h>
#define SIDECAR 3
void main ( )
{
  int CUBEBULK,CUBEAREA;
  cout<< "The cube's sidecar is"<<SIDECAR<<endl;
  CUBEBULK=SIDECAR* SIDECAR* SIDECAR;
  CUBEAREA=6* SIDECAR* SIDECAR;
  cout<< "The cube's bulk is"<<CUBEBULK<<endl;
  cout<< "The cube's area is"<<CUBEAREA<<endl;
}
```

经过宏替换以后，main 函数变为：

```
void main（）
```

```
{
  int CUBEBULK,CUBEAREA;
  cout<< "The cube's sidecar is"<<3<<endl;
  CUBEBULK=3*3*3;
  CUBEAREA=6* 3*3;
  cout<< "The cube's bulk is"<<CUBEBULK<<endl;
  cout<< "The cube's area is"<<CUBEAREA<<endl;
}
```

该程序执行后，输出结果如下：

The cube's sidecar is 3

The cube's bulk is 27

The cube's area is 54

我们已经看到：简单宏定义只是定义了一个符号常量。在 C++中，const 也是用来定义符号常量的。例如：

const double PI=3.14159265;

#define PI 3.14159265

上面两行的效果是一样的，都是将标识符 PI 定义为 3.14159265。但是，这两种定义符号常量的方法还是有区别的：

（1）const 产生的符号是具有类型的，在

const double PI=3.14159265;

中，PI 是一个 double 型的常量，而#define 命令定义的符号则不具有类型，它仅仅被另一个字符串替换，而不管内容是否正确。例如：

#define PI 3.14159,265

这样的简单宏定义在预编译的时候不会产生错误，因为系统仅仅将后面的"3.14159,265"视为一个字符串而不是 double 型的常量。但是，如果把它作为一个 double 型的常量，参与表达式的运算，则会出现编译错误，例如：

double r = 10.0;

double d = PI*r*r;

后一个语句经宏替换后，则变为：

double d = 3.14159,265*r*r;

显然是不对的。

（2）const 可以定义一个局部常量，在某一个函数体内用 const 定义的常量是局部常量，其作用域仅限于该函数体。而用#define 定义的常量则不一样，即使在某个函数体内，但它的作用域并不仅限于该函数体，而是从定义点开始，直到整个文件结束为止，除非在此过程中使用#undef 取消其定义。

（3）使用 const 定义常量是一个说明语句，以分号结束；而用#define 定义常量是一个预处理命令，不能用分号结束。

在 C++中，我们一般用 const 定义符号常量。很显然，用 const 定义常量比用 define 定义常量更好。

在使用宏定义时应注意的是：

（a） 在书写#define 命令时，注意<宏名>和<字符串>之间用空格分开，而不是用等号连接。

（b） 使用#define 定义的标识符不是变量，它只用作宏替换，因此不占有内存。

（c） 习惯上用大写字母表示<宏名>，这只是一种习惯的约定，其目的是为了与变量名区分，因为变量名通常用小写字母。

如果某一个标识符被定义为宏名后，在取消该宏定义之前，不允许重新对它进行宏定义。取消宏定义使用如下命令：

#undef<标识符>

其中，undef 是关键字。该命令的功能是取消对<标识符>已有的宏定义。被取消了宏定义的标识符，可以对它重新进行定义。

宏定义可以嵌套，已被定义的标识符可以用来定义新的标识符。例如：

#define PI 3.14159265

#define R 10

#define AREA (PI*R*R)

简单的宏定义将一个标识符定义为一个字符串，源程序中的该标识符均以指定的字符串来代替。前面已经说过，预处理命令不同于一般 C++语句。因此预处理命令后通常不加分号。这并不是说所有的预处理命令后都不能有分号出现。由于宏定义只是用宏名对一个字符串进行简单的替换，因此如果在宏定义命令后加了分号，将会连同分号一起进行置换。如：

#define PI 3.14159265;

则此时的 PI 所代替的字符串是"3.14159265;"而不是我们所期望的"3.14159265"。

例 9-3

```
#include<iostream.h>
//将系统输入输出流头文件包含进来
#define PI 3.14159265
//用简单宏定义定义了一个符号常量 PI
void main( )
{
  double r, l, s,v;
  cout<< "Input radius:";
  cin >>r;
  1=2*PI*r;
  s=PI * r*r;
  v=4.0/3.0 * PI *r*r*r;
  //上面语句中的 PI，被符号常量/3.14159265 替换
  cout<< "1="<<1<< "\n"<< "s="<<s<< "\n"<< "v="<<v<<end1;
  //分别输出 l、s、v 的值，观察宏替换产生的结果
}
```

该程序中 main 函数被替换后的结果如下：

```
void main( )
{
  double r, l, s,v;
  cout<< "Input radius:";
  cin>>r;
```

```
    l=2*3.14159265 *r;
    v=4.0/3.0 *3.14159265 *r* r*r;
    cout<< "l="<<l<< "\n"<< "s="<<s<< "\n"<< "v="<<v<<end1;
}
```

运行该程序，输入：

Input radius：5↙

输出结果为：

l=31.4159

s=78.5398

v=523.599

简单的宏定义和 const 语句都可以用来定义符号常量，但是两者又有许多不同之处：

（1）#define 命令定义的符号不具有类型，它仅仅是被另一个字符串替换罢了，而 const 产生的符号常量是具有类型的。例如：

const int SIZE=80;

说明 SIZE 是一个 int 型的常量，而#define 命令仅产生文本替换。

（2）两者定义的符号常量的作用域是不同的。在函数体内用 const 定义的常量是局部常量，其作用域仅限于该函数体。而用#define 定义的常量的作用域是从定义命令之后到本源文件结束，可以用"#undef"命令终止宏定义的作用域。下图可形象的说明用#define 定义的常量的作用域：

```
#define A 10
function()
{
…… A 的有效范围
……
……
}
#undef A
function2()
……
……
```

A 的有效范围

（3）const 语句必须以分号结束，而用#define 定义常量是一个预处理命令，结尾一般没有分号。

事实上，用 const 定义常量比用 define 定义常量要好。C 语言中没有用 const 定义常量的功能，常量要用 define 定义。C++为了与 C 语言兼容，也允许用 define 定义常量。

下面是一些比较常见的宏定义错误：

★#define A=10

不应该用等号连接，而应该用空格分开，正确写法为：

#define A 10

思考：#define A=10 的效果是什么？

★#define A 10;

预处理不用分号结束，这样定义的结果是 A 所定义的符号常量不是整数 10，而是内容为"10;"的一个字符串。正确的写法为：

#define A 10

★#define A10 #define B20

每条预处理命令必须单独占用一行，正确的写法应该是：

#define A 10

#define B 20

宏名一旦定义，就可以嵌套使用，成为其它宏定义的一部分。例如，下面代码定义了 ONE，TWO 以及 THREE 的值：

#define ONE 1

#define TWO ONE+ONE

#define THREE ONE+TWO

9.1.2 带参数的宏定义

带参数的宏定义的一般形式如下：

#define <宏名>（<参数表>） <宏体>

其中， <宏名>是一个标识符，<参数表>中的参数可以是一个，也可以是多个，视具体情况而定，当有多个参数的时候，每个参数之间用逗号分隔。<宏体>是被替换用的字符串，宏体中的字符串是由参数表中的各个参数组成的表达式。例如：

#define SUB(a,b) a-b

如果在程序中出现如下语句：

result=SUB(2, 3)

则被替换为：

result=2-3；

如果程序中出现如下语句：

result= SUB（x+1, y+2）;

则被替换为：

result=x+1-y+2;

在这样的宏替换过程中，其实只是将参数表中的参数代入到宏体的表达式中去，上述例子中，即是将表达式中的 a 和 b 分别用 2 和 3 代入。

我们可以发现：带参的宏定义与函数类似。如果我们把宏定义时出现的参数视为形参，而在程序中引用宏定义时出现的参数视为实参。那么上例中的 a 和 b 就是形参，而 2 和 3 以及 x+1 和 y+2 都为实参。在宏替换时，就是用实参来替换<宏体>中的形参。

下面是带参数的宏定义的例子：

例 9-4

```
#include <iostream.h>
#define SUB(a,b) a-b
void main ()
{
   int x,y,result;
   x=2;
   y=3;
```

```
    result=SUB(x+2,y-1);
    cout<<"result="<<result<<end1;
}
```

该程序经宏替换后，main 函数变为：

```
void main ()
{
    int x,y,result;
    result=x+2-y-1;
    cout<<"result="<<result<<end1;
}
```

执行该程序后，输出结果如下：

result=0

虽然带参数的宏定义和带参数的函数很相似，但它们还是有本质上的区别：

（1） 在带参的宏定义中，形式参数不是变量，只是一个符号，不分配内存单元，不必定义其形式参数的类型。发生宏"调用"时，只是将实参替换形参。而在函数中，形参和实参是完全独立的变量，它们均有自己的作用域。当调用发生时，实参传递给形参的过程（值传递或引用传递）完全不同于简单的宏替换。

（2） 在宏定义中的形参是标识符，而宏调用中的实参可以是表达式。如：

例 9-5

```
#include <iostream.h>
#define SQ(y) (y)*(y)
void main()
{
    int x, result;
    cout<<"input a number: "<<endl;
    cin>>x;
    sq=SQ(x+1);
    cout<<"result="<<endl<<result;
}
```

上例中"#defint SQ(y) (y)*(y)"一行为宏定义，形参为 y。程序中宏调用时实参为 x+1，是一个表达式，在宏展开时，用 x+1 代换 y，再用(y)*(y) 代换 SQ，得到如下语句："result=(a+1)*(a+1); "，这与函数的调用是不同的， 函数调用时要把实参表达式的值求出来再赋予形参。而宏代换中，对实参表达式不作计算直接地照原样替换。

在使用带参数的宏定义时需要注意的是：

（1）带参数的宏定义的<宏体>应写在一行上，如果需要写在多行上时，在每行结束时，使用续行符 "\" 结束，并在该符号后按下回车键，最后一行除外。

（2）在书写带参数的宏定义时，<宏名>与左括号之间不能出现空格，否则空格右边的部分都作为宏体。例如：

#define ADD (x,y) x+y

将会把"（x,y）x+y"的一个整体作为被定义的字符串。

（3）定义带参数的宏时，宏体中与参数名相同的字符串适当地加上圆括号是十分重要的，这样能够避免可能产生的错误。例如,对于宏定义：

#define SQ(x) x*x

当程序中出现下列语句：

m=SQ(a+b);

替换结果为：

m=a+b*a+b;

这可能不是我们期望的结果，如果需要下面的替换结果：

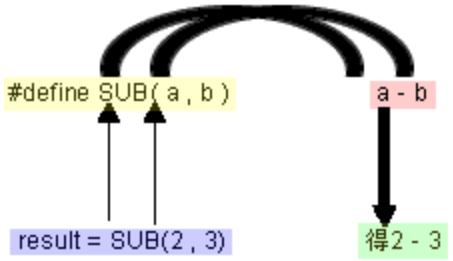
m=(a+b)*(a+b);

应将宏定义修改为：

#define SQ(x) (x)*(x)

对于带参的宏定义展开置换的方法是：在程序中如果有带实参的宏（如"SUB（2,3）"），则按"#define"命令行中指定的字符串从左到右进行置换。如果串中包含宏中的形参（如 a、b），则将程序语句中相应的实参（可以是常量、变量或者表达式）代替形参，如果宏定义中的字符串中的字符不是参数字符（如 a-b 中的-号），则保留。这样就形成了置换的字符串。如下图所示：

图 9-1



例 9-6

```
#include <iostream.h>
//将系统输入输出流头文件包含进来
#define SUB(a,b) a-b
//用带参数的宏定义定义了一个宏 SUB(a,b)
void main ()
{
   int x,y,result;
   x=2;
   y=3;
   result=SUB(x+2,y-1);
   //调用该宏
   cout<< "result="<<result<<end1;
}
//输出 result 的值，观察宏替换产生的结果
```

带参数的宏定义和带参数的函数在形式上很相似，却有本质上的区别：

（1）在带参的宏定义中，宏"调用"只是将实参替换形参。而在函数中，形参和实参是完全独立的变量，当调用发生时，实参的值传递给形参。

从发生的时间来说，宏"调用"是在编译时发生的，函数调用是在程序运行时发生的；宏"调用"不存在内存单元分配的问题，而函数调用时，会给形参变量分配内存单元，然后，复制实参的值，函数调用结束后，形参变量占用的内存单元被释放。

（2）在宏定义中的形参是标识符，而宏调用中的实参可以是表达式。如：

例 9-7

```
#include <iostream.h>
//将系统输入输出流头文件包含进来
#define SQ(y) (y)*(y)
//用带参数的宏定义定义了一个宏 SQ(y)
void main()
{
   int x, result;
   cout<<"input a number: "<<endl;
   cin>>x;
   sq=SQ(x+1);
   //调用该宏
   cout<<"result="<<endl<<result;
   //输出 result 的值，观察宏替换产生的结果
}
```

对于上边的例程而言，第二行

"#define SQ(y) (y)*(y)"

中的"(y)*(y)"的括号是必需的，因为对带参数的宏的展开只是将语句中的宏名后面括号内的实参字符串代替#define 命令行中的形参。如果不加括号，即：

"#define SQ(y) y*y"

则后面的语句：

"sq=SQ(x+1);"

把实参"x+1"代替"y*y"中的形参"y"成为：

"sq=x+1*x+1;"

这显然与程序设计原来所希望得到的结果"(x+1)*(x+1)"不符，因此必须要在 y 的外面加上括

文件包含命令格式如下：

#include <文件名>

或者

#include "文件名"

其中，include 是关键字，文件名是指被包含的文件全名。在前面我们已多次用此命令包含过库函数的头文件。例如："#include "stdio.h""。

文件包含命令的功能是把指定的文件插入该命令行位置，从而把指定的文件和当前的源程序文件连成

一个源文件。在 C++的程序设计中，文件包含命令用得很多。我们常将符号常量、类及其它类型的定义放到一个.h 文件中（即文件扩展名为 h 的文件，称之为头文件）。在其它文件的开头用包含命令包含该头文件，这样就可以减少重复劳动，节省时间，并减少出错的可能性。头文件可以是自己编写的，也可以是系统提供的。例如，前面程序中已出现的 iostream.h 便是一个系统提供的、有关输入输出操作信息的头文件。

上面我们看到，文件包含命令的格式有两种，一种将文件名以尖括号（<>）括起，另一种是将文件名以双引号（""）括起。这两种格式的用法略有区别：使用前一种格式，在编译的时候将会在指定的目录下查找此头文件，而使用后一种格式，在编译的时侯会首先在当前的源文件目录中查找该头文件，若找不到才会到系统的指定目录下去查找。

在定义和使用文件包含时还应注意以下几点：

（1）一条文件包含命令只能包含一个文件，若想包含多个文件须用多条文件包含命令。例如：

```
#include<iostream.h>
#include<stdio.h>
…
```

（2）文件包含命令可以嵌套使用，即在一个被包含的文件中可以包含另一个文件。例如，定义一个头文件，其名字为 headfilel.h，该文件内容如下：

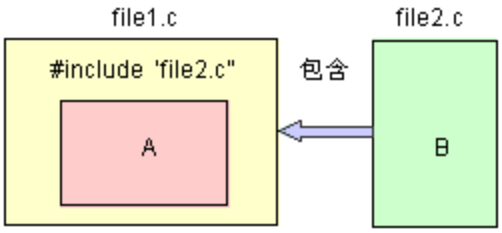
```
#include "headfile2.h"
#include "headfile3.h"
…
```

这里即嵌套使用了文件包含命令。

文件包含是指一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。

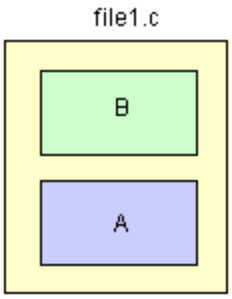
下图是一个文件包含的例子：

图 9-2



上图表示了"文件包含"的含意，在 file1.c 中有一个#include "file2.c"的命令，该文件中的其它内容用 A 表示，file2.c 的文件内容用 B 表示。在编译预处理的时候，需要对#include 命令进行处理，即将 file2.c 的全部内容复制、插入到#include "file2.c"命令处，也即 file2.c 被包含到 file1.c 中，得到的结果如下图所示：

图 9-3



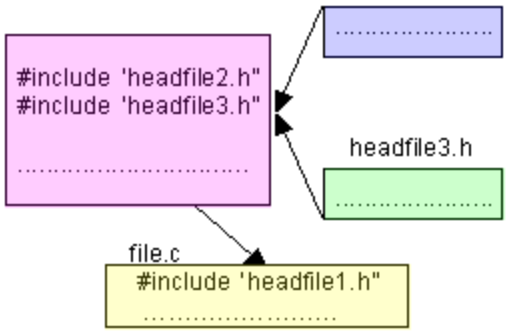
file1.c 将作为一个整体的源文件单位进行编译。

"文件包含"命令在 C++程序中经常可以见到，它可以节省程序设计人员的重复劳动。例如，我们在编写一个程序解决物理问题的时候，经常会碰到一些固定的符号常量（如 g=9.8，pi=3.1415928，c=300000000，…），可以将这些常量用宏来定义，并把这些宏定义命令放在一个文件中。编写程序时，可以用#include 命令将该文件包含到程序中来。

当一个文件 file.c 包含头文件 headfile1.h，而 headfile1.h 又用到了 headfile2.h 和 headfile3.h 的内容，这时应注意不要在 file.c 文件中，把这三个文件.h 同时包含进来：

headfile2.h

图 9-4



如果在 file.c 文件中用三个 include 命令分别包含 headfile2.h、headfile3.h 和 headfile1.h：

#include "headfile2.h"

#include "headfile3.h"

#include "headfile1.h"

则 headfile2.h、headfile3.h 文件中必须有条件编译命令，否则 file.c 文件会出现编译错误

条件编译命令可以使得编译器按不同的条件去编译程序不同的部分，产生不同的目标代码文件。也就是说，通过条件编译命令，某些程序代码要在满足一定条件下才被编译，否则将不被编译。

常用的条件编译命令有如下三种格式：

9.3.1 格式一

#ifdef 标识符

程序段 1

#else

程序段 2

#endif

其中，ifdef、else 和 endif 都是关键字。程序段 1 和 程序段 2 是由若干预处理命令和语句组成的。它的功能是，如果标识符已被 #define 命令定义过，则对程序段 1 进行编译；否则对程序段 2 进行编译。本格式中的#else 也可以没有：

```
#ifdef 标识符
  程序段
#endif
```

下面是使用这种条件编译命令的一个例子：

例 9-8

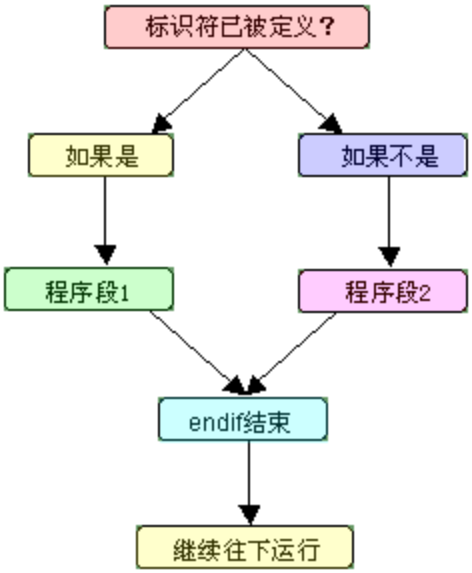
```
#include <iostream.h>
#define TIME
void main()
{
  #ifdef TIME
    cout<<"Now begin to work"<<endl;
  #else
    cout<<"You can have a rest"<<endl;
  #endif
}
```

在此程序中因为加入了条件编译预处理命令，因此要根据 TIME 是否被#define 语句定义过，来决定编译哪一个 cout 语句。如果定义过，则编译"cout<<"Now begin to work"<<endl;"语句，否则则编译"cout<<"You can have a rest"<<endl;"语句。本例 TIME 已经定义过，所以输出的结果为：

```
Now begin to work
```

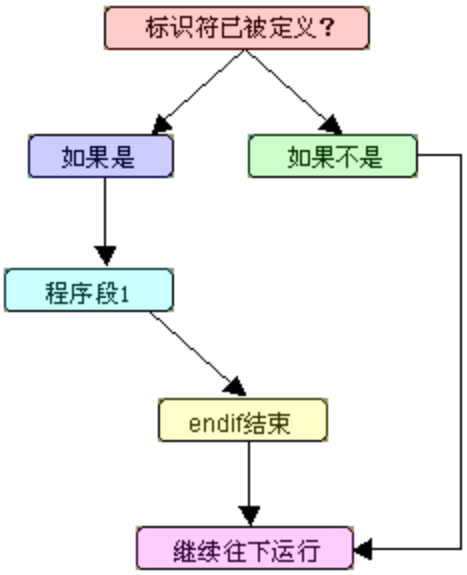
格式一的流程图表示如下：

图 9-5



本格式中的#else 也可以没有：如果没有"#else"，则流程图如下：

图 9-6



例 9-9

```
#include <iostream.h>
//将系统输入输出流头文件包含进来
#define TIME
//用简单宏定义定义了一个符号常量 TIME
void main()
{
  #ifdef TIME
```

```
    //判断 TIME 是否已经被#define 命令定义过
      cout<<"Now begin to work"<<endl;
      //如果已经定义过，则输出"Now begin to //work"
    #else
    //如果没有定义过，则输出"You can have a //rest"
      cout<<"You can have a rest"<<endl;
    #endif
}
```

9.3.2 格式二

```
#ifndef 标识符
  程序段 1
#else
  程序段 2
#endif
```

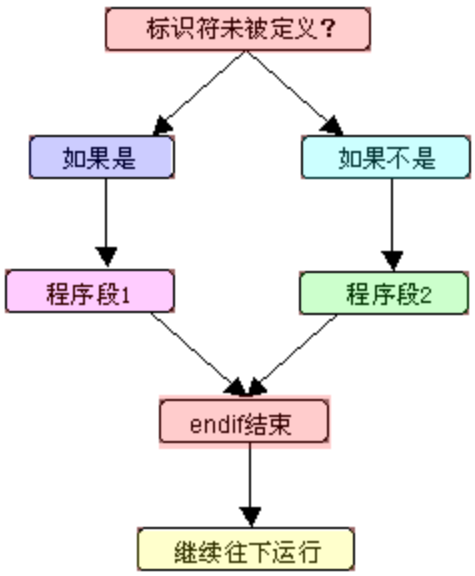
格式二和格式一形式上的的区别在于 ifdef 关键字换成了 ifndef 关键字，其功能是：如果标识符未被 #define 命令定义过，则对程序段 1 进行编译， 否则对程序段 2 进行编译，这与格式一的功能正好相反。例如：

```
#ifndef NULL
#define NULL ((void *)0)
#endif
```

本段代码能够保证符号 NULL 只有一次定义为((void *)0

格式二的流程图表示如下：

图 9-7



9.3.3 格式三

```
#if 常量表达式
   程序段 1
#else
   程序段 2
#endif
```

if、else 和 endif 是关键字。程序段 1、程序段 2 都是由若干条预处理命令和语句组成。它的功能是：如常量表达式的值为真(true)，则对程序段 1 进行编译，否则对程序段 2 进行编译。因此可以使程序在不同条件下，完成不同的功能。

举例如下：

例 9-10

```
#include <iostream.h
#define R 1
main()
{
   float c,r,s;
   cout<<"input a number: "<<endl;
   cin>>c;
   #if R
   r=3.14159*c*c;
   cout<<"area of round is:"<<r<<endl;
   #else
   s=c*c;
   cout<<"area of square is"<<s<<endl;
   #endif
}
```

在这个例子中，如果常量表达式 R 为真，则编译语句：

```
r=3.14159*c*c;
cout<<"area of round is:"<<r<<endl;
```

否则则编译语句：

```
s=c*c;
cout<<"area of square is"<<s<<endl;
```

【本章小结】

编译预处理命令并不遵循 C++的语法，而是具有自己独特的语法结构。编译预处理命令可以控制编译器的行为，在有些情况下非常有用。

C++提供的编译预处理功能主要有以下三种：宏定义、文件包含和条件编译，这些命令是在程序编译前被执行的，也就是说，在源程序编译以前，编译器先执行这些命令。

宏定义命令将一个标识符定义为一个字符串。简单的宏定义，是在程序被编译的时侯，如果遇到宏名，先将宏名用指定的字符串替换，然后再进行编译。带参数的宏定义和带参数的函数很相似，

我们要注意它们的区别。

文件包含命令的功能是把指定的文件插入到该命令行位置，从而把指定的文件和当前的源程序文件连成一个源文件。我们常将符号常量、类（第十章介绍）及其它类型的定义放到头文件中，在其它文件的开头用包含命令包含该头文件。头文件可以是自己编写的，也可以是系统提供的。

条件编译命令可以使得编译器按不同的条件去编译程序不同的部分，产生不同的目标代码文件。这样可以避免文件重复包含带来的问题，也可以减少编译的代码量，提高程序的运行效率。

【学习目标】

理解面向对象程序设计的基本思想和基本概念，理解面向对象的程序与结构化程序的不同之处。能用面向对象的程序设计方法，编写基于对象的程序。

【重点与难点】

本章重点包括

◇ 类与对象的概念和定义、类成员的访问权限；

◇ 构造函数（包括复制构造函数）和析构函数的概念和用法；

◇ this 指针的用法

◇ 类作用域的概念；

◇ 静态数据成员和静态成员函数的概念和用法；

◇ 友元概念和用法。

本章的难点包括：

◇ 复制构造函数的概念和使用；

◇ const 成员函数的概念和使用；

◇ 静态成员的概念和使用；

◇ 友元的概念和使用。

【学习方法指导】

面向对象的程序与结构化的程序有很大的不同，本章学习中，一定要理解和领会面向对象程序设计的基本思想和基本概念，否则，会感到很困难。例如：面向对象程序的结构如何，如何封装，类成员的访问权限是如何控制的，构造函数与析构函数的作用是什么，有什么特点，为什么需要复制构造函数，它又有什么特点，this 指针是什么，如何使用，静态成员、友元的作用与特点等。这些概念和用法要通过多编程、多体会才能逐渐掌握。

【知识点】

类；对象；数据成员；成员函数；public；private；protected；this 指针；构造函数；析构函数；复制构造函数；类作用域；const 成员函数；静态成员；友元

C++语言是当今应用最广泛的程序设计语言，它与 C 语言兼容，既支持面向对象的程序设计，也支持面向对象的程序设计方法。在前面的章节中，我们编写的程序是由一个个函数组成的，可以说是结构化的程序。从本章开始，我们编写的程序是由对象组成的，也就是说，将要学习用 C++语言进行面向对象的程序设计。

什么叫类，什么叫对象？我们已经知道什么叫变量。假定我们在 main 函数中定义了一个整型变量 nInteger：

```
void main()
```

```
{
  int nInteger;
  …
}
```

则在 main 函数中为 nInteger 分配栈内存，保存变量 nInteger 的值，并在 main 返回时，释放该内存。在面向对象的程序设计中，nInteger 也称之为对象。所谓对象就是一个内存区，它存储某种类型的数值，变量就是有名的对象。对象除可以用上述定义的方法来创建外，也可以用 new 表达式创建，也可能是应用程序运行时临时创建的，例如，在函数调用和返回时，均会创建临时对象。

对象是有类型的，例如，我们上面定义的 nInteger 对象就是整型的。一个类型可以定义许多对象，一个对象有一个确定的类型，可以这么说：int 型变量是 int 类型的实例。以后，我们也常说：对象是类的实例，那么 int 是不是一个类呢？

实际上，我们所说的类，并非指 C++中的那些基本的数据类型。C++中引入了 class 关键字来定义类，它也是一种数据类型。类是 C++支持面向对象的程序设计的基础，它支持数据的封装、隐藏等。类与我们前面学习过的结构类似，实际上 C++中也可以用 struct 关键字来定义类（虽然很少使用）。

我们前面学习的结构中，只有数据成员。实际上，类中除可以定义数据成员外，还可以定义对这些数据成员（或对象）操作的函数，也正是这些函数限制了对对象的操作，即不能对对象进行这些操作函数之外的其它操作，类的成员也有不同的访问权限。下面，我们将要介绍怎样定义类及类的成员。

我们的周围是一个真实的世界，不论在何处，我们所见到的东西都可以看成是对象。人、动物、工厂、汽车、植物、建筑物、割草机、计算机等等都是对象，现实世界是由对象组成的。

对象多种多样，各种对象的属性也不相同。有的对象有固定的形状，有的对象没有固定的形状，有的对象有生命，有的对象没有生命，有的对象可见，有的对象不可见，有的对象会飞，有的对象会跑，有的对象很高级，而有的对象很原始，…。各个对象也有自己的行为，例如：球的滚动、弹跳和缩小，婴儿的啼哭、睡眠、走路和眨眼，汽车的加速、刹车和转弯，等等。但是，各个对象可能也有一些共同之处，至少它们都是现实世界的组成部分。

人们是通用过研究对象的属性和观察它们的行为而认识对象的。我们可以把对象分成很多类，每一大类中又可分成若干小类，也就是说，类是分层的。同一类的对象具有许多相同的属性和行为，不同类的对象可具有许多相同的属性和类似的行为，例如：婴儿和成人，人和猩猩，小汽车和卡车、四轮马车、冰鞋等等都有共同之处，类是对对象的抽象。

在 C++中，就是用类来描述对象的，类是对现实世界的抽象得到的。例如，在真实世界中，同是人类的张三和李四，有许多共同点，但肯定也有许多不同点。当用 C++描述时，相同类的对象具有相同的属性和行为，它把对象分为两个部分：数据（相当于属性）和对数据的操作（相当于行为）。我们刻画张三和李四的数据可能用姓名、性别、年龄、职业、住址等，而对数据的操作可能是读或设置它们他们的名字、年龄等。

从程序设计的观点来说，类就是数据类型，是用户定义的数据类型。这种类型的使用虽然与 C++内置的数据类型类似，但是也有很大的区别。例如，C++内置的浮点类型并不针对任何具体问题，仅仅与机器的存储单元相对应，而类是用户根据具体问题的需要而定义的，也就是说，类与具体问题相适应。我们可以通过定义所需要的类，来扩展程序设计语言解决问题的能力。

当我们把现实世界分解为一个个的对象，解决现实世界问题的计算机程序也与此相对应，由一个个对象组成，这些程序就称为面向对象的程序，编写面向对象程序的过程就称为面向对象的程序设计（Object-Oriented Programming，简称为 OOP）。OOP 技术能够将许多现实的问题归纳成为一个简单解，支持 OOP 的语言也很多，C++是应用最广泛的、支持 OOP 的语言，第一个成功的支持 OOP 的语言是 Smalltalk。

面向对象的程序设计（OOP）使用软件的方法模拟真实世界的对象，它利用了类的关系，即同一对象（如同一类运载工具）具有相同的特点；还利用了继承甚至多重继承的关系，即新建的对象类是通过继承现有类的特点而派生出来的，但是又包含了其自身特有的特点，如子女有父母的许多特点，但是矮个子父母的子女也可能是高个子。

面向对象的程序设计（OOP）使程序设计过程更自然和直观。也就是说，面向对象的程序设计模拟了真实世界的对象（它们的属性和行为）。OOP 还模拟了对象之间的通信，就像人们之间互送消息一样（如军官命令部队立正），对象也是通过消息进行通信的。

10.1.1 类的定义

类定义的一般形式如下：

```
class Name
{
  细节
};
```

类的定义由头和体两个部分组成。类头由关键字 class 开头，然后是类名，其命名规则与一般标识符的命名规则一致，有时可能有附加的命名规则，例如美国微软公司的 MFC 类库中的所有类均是以大写字母'C'开头的。类体包括所有的细节，并放在一对花括号中。类的定义也是一个语句，所以要有分号结尾，否则，会产生难以理解的编译错误。

类体定义类的成员，它支持两种类型的成员：

1. 数据成员，它们指定了该类对象的内部表示。
2. 成员函数，它们指定该类的操作。

类成员有三种不同的访问权限：

1. 公有（public）成员可以在类外访问。
2. 私有（private）成员只能被该类的成员函数访问。
3. 保护（protected）成员只能被该类的成员函数或派生类（有关基类和派生类的概念我们在下一章介绍）的成员函数访问。

数据成员通常是私有的，成员函数通常有一部分是公有的，一部分是私有的。公有的成员函数可在类外被访问，也称之为类的接口。我们可以为各个数据成员和成员函数指定合适的访问权限，类定义常有下面的形式：

```
class Name {
  public:
    类的公有接口
  private:
    私有的成员函数
    私有的数据成员定义
};
```

私有的成员与公有的成员的先后次序无关紧要。不过公有的接口函数放在前面更好，因为，有时我们可能只想知道怎样使用一个类的对象，那只要知道类的公有接口就行了，不必阅读 private 关键字以下的部分。

下面是一个简单类的定义实例，该类定义了一个二维坐标点。

例 10-1

```
1.class Point
{
2. public:
3. int GetX();
4. int GetY();
5. void SetPt (int, int);
6. void OffsetPt (int, int);
7. private:
8. int xVal, yVal;
9. }
```

表 10-1 注释

1.	用 class 关键字定义名为 Point 的类，类体包含在花括号中
2.	public 关键字以下的类成员为公有成员
3-4.	两个公有成员函数的原型，无形参，返回类型 int
5-6.	两个公有成员函数的原型，带两个整型参数，返回类型为 void
5.	private 关键字以下的类成员为私有成员
6.	定义两个私有的数据成员 xVal 和 yVal

如果未指定类成员的访问权限，默认访问权限是私有的，且数据成员和成员函数出现的顺序也没有关联。例 10-1 也可写成下面的形式：

```
class Point {
   int xVal, yVal;
   public:
   int GetX();
   int GetY();
   void SetPt (int, int);
   void OffsetPt (int, int);
};
```

"声明"向计算机介绍名字，这个名字是什么意思。而"定义"为这个名字分配存储空间。无论涉及到变量时还是函数时含义都一样。无论在哪种情况下，编译器都在"定义"处分配存储空间。对于变量，编译器确定这个变量占多少存储单元，并在内存中产生存放它们的空间。对于函数，编译器产生代码，并为之分配存储空间。

定义也可以是声明。如果该编译器还没有看到过名字 A，程序员定义 int A，则编译器马上为这个名字分配存储地址。

声明常常使用 extern 关键字。如果我们只是声明变量而不是定义它，则要求使用 extern。对于函数声明，extern 是可选的，不带函数体的函数名连同参数表或返回值，自动地作为一个声明。

函数原型包括关于参数类型和返回值的全部信息。"int f(float,char);"是一个函数原型，因为它不仅介绍 f 这个函数的名字，而且告诉编译器这个函数有什么样的参数和返回值，使得编译器能对参数和返回值做适

当的处理。下面是一些声明的例子：

```
extern int i; //声明但未定义
extern float f(float); //函数声明
float b; //声明和定义
float f(float a) //定义
{
  return a + 1.0;
}
int i; //定义
int h(int x) //声明和定义
{
  return x + 1;
}
void main(void)
{
  b = 1.0;
  i = 2;
  f(b);
  h(i);
}
```

在函数声明时，参数名可给出也可不给出。而在定义时，它们是必需的。

C++的类机制

这里我们讨论 C++的关键概念之一：类机制。我们不仅要学会如何声明、定义和使用类，而且，最重要的是，要知道用类代替结构的原因，而且要知道使用类究竟能带来什么好处。

首先，我们回顾一下已经学习过的结构：结构是一种用户定义的数据类型，它将合法的数据类型的变量聚集到一起。下面是一个简单的结构声明：

```
struct family
{
  char *husband;
  char *wife;
  char *son;
  char *daughter;
};
family Anderson;
```

结构 family 由四个指向字符数组的指针变量组成，这些变量分别为 husband、wife、son 和 daughter。Anderson 是结构类型 family 的一个变量，也可以在声明时初始化 Anderson 的成员，并通过指针来存取：

```
Anderson.husband = "John Anderson";
Anderson.wife = "Mary Anderson";
Anderson.son = "Joey Anderson";
Anderson.daughter = "Marla Anderson";
```

通过指针来访问 Anderson 的成员，可用如下程序来说明：

例 10-2

```
#include <iostream.h>
struct family
{
   char *husband;
   char *wife;
   char *son;
   char *daughter;
};
//声明 Anderson 为结构类型 family 的一个变量
family Anderson =
{
   "John Anderson","Mary Anderson",
   "Joey Anderson","Marla Anderson"
};
//ptr 是 family 类型的指针
family *ptr;
void main(void)
{
   //ptr 指向 family 的第一个成员
   ptr=&Anderson;
   cout<<"husband is "<<ptr->husband<<endl
   <<"wife is "<<ptr->wife<<endl
   <<"son is "<<ptr->son<<endl
   <<"daughter is "<<ptr->daughter<<endl;
}
```

编译运行该程序会得到如下输出结果：

husband is John Anderson
wife is Mary John Anderson
son is Joey Anderson
daughter is Marla Anderson

在本程序中声明了一个名为 family 的结构，定义 Anderson 为这个结构类型的变量，并且给其中各成员赋以初始化值，接着，声明了这个类型的指针 ptr。在 main 中，指针 ptr 指向存储结构变量 Anderson 的内存地址。通过指向相应成员的指针，由 cout 语句来打印输出 Anderson 的每个结构成员。

要注意在声明 Anderson 和 ptr 时，关键字 struct 被省略掉了。在 C++中，这个关键字是可选的，但在声明结构类型时，这个关键字则是必不可少的。

再来看一下这个 main 函数，就会发现它分成了两个部分。第一部分初始化指针 ptr，使之指向结构变量 Anderson，第二部分只是简单地输出每个成员的内容。下面我们修改这个 main 函数，建立两个函数来实现这个功能。

程序段 10-1

```
family *ptr;
family *initialize(family  *ptr)
{
   ptr=&Anderson;
   return ptr;
}
void output(family *ptr)
{
   cout<<"husband is "<<ptr->husband<<endl
   <<"wife is "<<ptr->wife<<endl
   <<"son is "<<ptr->son<<endl
   <<"daughter is "<<ptr->daughter<<endl;
}
void main(void)
{
   ptr=initialize(ptr);
   output(ptr);
}
```

现在假定有 15 个附加函数组成的 1000 行程序代码，要加到这一源始程序中，这些增加的函数由 main 来调用：

例 10-3

```
#include <iostream.h>
struct family
{
   char *husband;
   char *wife;
   char *son;
   char *daughter;
};
family Anderson =
{
   "John Anderson","Mary Anderson",
   "Joey Anderson","Marla Anderson"
};
…
family *ptr;
//函数 initialize 的代码保持不变
family *initialize(family  *ptr)
```

```
{
  …
}
//增加的 15 个函数的函数原型放在这里
//函数 output 的代码保持不变
void output(family *ptr)
{
  cout<<"husband is "<<ptr->husband<<endl
  <<"wife is "<<ptr->wife<<endl
  <<"son is "<<ptr->son<<endl
  <<"daughter is "<<ptr->daughter<<endl;
}
void main(void)
{
  …
}
```

假定该程序的输出结果如下：

husband is Mark Davis

wife is Jennifer Davis

son is Michael Davis

daughter is Maria Davis

这个输出结果是由于在程序的某个过程中指针 ptr 错误地指向结构类型 faimly 的其它的变量（也许是 Davis），它的结构成员已初始化为 Davis 的成员了。这对于作为程序员来说意味着要在这 1000 行语句的程序中，查找究竟是在哪个函数中将指针 ptr 设为指向 family 类型的变量 Davis（而不是 Anderson）的内存地址。可是这样的事情通常会使程序员既伤脑筋又浪费时间，C++的类就能有效地解决

在 C++中，有一个类机制允许指定一个独立的对象集合，该对象集合由组成该类的对象以及这些对象所允许的操作组成。让我们来修改一下这个程序的最初版本，用类（class）来代替结构，看一看类是如何运作的：

例 10-4

```
#include <iostream.h>
class family
{
  private:
    char *husband;
    char *wife;
    char *son;
    char *daughter;
    family *ptr;//ptr 在类 family 内部声明
  public:
```

```
        void initialize(void);
        void output(family *ptr);
};
//Anderson 是类 family 的对象
family Anderson;
//initialize 被类名限定
void family::initialize(void)
{
   Anderson.ptr=&Anderson;
   Anderson.ptr->husband="John Anderson";
   Anderson.ptr->wife="Mary Anderson";
   Anderson.ptr->son="Joey Anderson";
   Anderson.ptr->daughter="Marla Anderson";
   Anderson.output(Anderson.ptr);
}
//output 被类名限定
void family::output(family *ptr)
{
   cout<<"husband is "<<ptr->husband<<endl
   <<"wife is "<<ptr->wife<<endl
   <<"son is "<<ptr->son<<endl
   <<"daughter is "<<ptr->daughter<<endl;
}
void main(void)
{
   //initialize 被对象名限定
   //注意圆点（.）操作符
   Anderson.initialize();
}
```

该程序产生如下输出结果：

```
husband is Mark Anderson
wife is Jennifer Anderson
son is Michael Anderson
daughter is Maria Anderson
```

现在让我们一步步地分析这个程序，理解为什么要用类来代替结构，以及为什么要这样来声明类。

我们将 family 结构改成了 family 类，类定义用 class 关键字。类的成员包括数据和操作（函数），它们有不同的访问权限。在关键字 private 的后面声明的 husband、wife、son、daughter 和 ptr 是私有的。ptr 定义为指向 family 的指针，编译器自然会将 family 理解为类名，因为已经在类声明的开始处指定过了。

下面是关键字 public 后跟一个冒号，然后是成员函数声明。由于这些函数声明为 public，它们可以在整个程序范围内的任何地方被访问。

在 main 中，对成员函数 initialize 作了调用。该函数有一个前缀 Anderson，以及一个圆点（.），再回顾一下结构成员的引用方式。

```
struct family
{
   char *husband;
   char *wife;
   char *son;
   char *daughter;
};
family Anderson;
```

结构成员前面带有这种结构类型的变量名加上一个圆点运算符。

可以用同样的方法来引用类成员。initialize 是类 family 的成员函数，Anderson 则是该类的一个对象。所以，可用下面的方式调用函数 initialize：

```
Anderson.initialize();
```

再来看看 initialize 函数的定义：

```
void family::initialize(void)
{
   Anderson.ptr = &Anderson;
   Anderson.ptr->husband = "John Anderson";
   Anderson.ptr->wife = "Mary Anderson";
   Anderson.ptr->son = "Joey Anderson";
   Anderson.ptr->daughter = "Marla Anderson";
   Anderson.output(Anderson.ptr);
}
```

函数名前面是类名加上作用域分隔符::。在我们的程序中很可能还有其它的函数以 initialize 为名，然而编译器却能理解这里只不过是类 family 的一个成员函数，因为在它的前面已经标明了类的名称，换句话说，该函数被类名所限定。让我们继续分析下列代码：

```
Anderson.ptr = &Anderson;
Anderson.ptr->husband = "John Anderson";
Anderson.ptr->wife = "Mary Anderson";
Anderson.ptr->son = "Joey Anderson";
Anderson.ptr->daughter = "Marla Anderson";
```

类成员 ptr 赋值为类对象 Anderson 内存地址，然后初始化其余的成员。最后，调用成员函数 output。

从上面的对比讲述可以看出，传统的 struct 编程使得数据和操作分开，这是造成上述 output( )输出错误的根本原因。而类和对象的概念，将数据和在此数据上执行的操作封装在一起，使得 Anderson 的 output( ) 操作只能输出 Anderson 家庭的数据信息，而不再会产生输出另外一个家庭（比如 Davis 家庭）信息的错误。

类成员的数据类型

类成员列表可以由任何合法的 C++数据类型组成。

可以包含通常的基本类型：

```
class primary
{
```

```
    int a;//整型
    char b;//字符型
    float c;//浮点型
    double d;//双精度浮点型
}
//class_1 是类 primary 的一个对象
primary class_1;
```

可以包含结构类型：

```
class structure_1
{
    //成员表中包含结构类型 family
    struct family;
}
//Anderson 是类 structure_1 的对象
structure_1 Anderson;
```

可以包含任何合法类型的指针：

```
class pointer_1
{
    //ptr 是结构类型 family 的指针
    struct family;
    family *ptr;
}
//Anderson 是类 pointer_1 的一个对象
pointer-1 Anderson;
```

另外，还可以包含类对象。

类成员的访问说明符

类成员可以是公有的（public）、私有的（private）或保护的（protected），类成员通过这些访问说明符控制其访问权限。

在 C++中，struct 也可以定义类，与用 class 定义类不同的是：缺省访问说明符的成员访问权限是 public（公有的），在下面的声明中：

```
struct family
{
    char *husband;
    char *wife;
    char *son;
    char *daughter;
};
family Anderson; //Anderson 是结构类型 family 的变量
```

在程序中的任何地方都可以通过变量（对象）访问结构（类）成员 husband、wife、son 和 daughter。用 class 关键字定义的类，缺省访问说明符的成员访问权限是 private（私有的），即只能被该类的成员函数访问。所有 public（公有）成员都必须显式说明：

```
class family
{
  public:
  char *husband;
  char *wife;
  char *son;
  char *daughter;
};
family Anderson; //Anderson 是类 family 的对象
```

一个访问说明符在遇到另一个访问说明符之前，对其后的所有的成员的声明均有效。同一访问说明符在类中也可以多次使用

10.1.2 类成员函数的定义

类的成员函数通常在类外定义，一般形式如下:

```
返回类型 类名::函数名(形参表)
{
  函数体
}
```

双冒号::是域运算符,它主要用于类的成员函数的定义

例如，例 10-1 的成员函数 Point 类的成员函数可定义为：

例 10-5

```
1. void Point::SetPt (int x, int y)
2. {
3.     xVal = x;
4.    yVal = y;
5. }
6. void Point::OffsetPt (int x, int y)
7. {
8.     xVal += x;
9.     yVal += y;
10. }
11. int Point:: GetX ()
12. {
13.    return xVal;
14. }
15. int Point:: GetY ()
16. {
17.    return yVal;
18. }
```

表 10-2 注释

1.	类成员函数的定义类似于普通的全局函数的定义，但函数名前有类名和一个双冒号（即域运算符），用于指定 SetPt 是 Point 类的成员
3-4.	成员函数 SetPt 能够访问私有成员 xVal 和 yVal，非该类成员函数不能访问

正象全局函数可以定义为内联函数一样，类的成员函数也可以定义为内联的。例如，类 Point 的两个成员函数都很短（仅两个语句），如果定义为内联函数，可以提高执行效率。定义内联的成员函数只要在该成员函数定义前插入 inline 关键字即可：

```
inline void Point::SetPt (int x,int y)
{
   xVal = x;
   yVal = y;
}
```

定义内联函数一个更容易的方法是：将函数定义放在类的定义内。

```
class Point {
   int xVal, yVal;
public:
   int GetX (){return xVal ;}
   int GetY (){return yVal;}
   void SetPt (int x,int y) { xVal = x; yVal = y; }
   void OffsetPt (int x,int y) { xVal += x; yVal += y; }
};
```

需要注意的是：由于函数体在类内，所以函数原型后不需要分号，且所有的函数参数必须有名的。

类成员函数可以是公有的，也可以是私有的。公有成员函数可在类外调用，而私有成员函数只能被该类的其它成员函数调用。例如：

例 10-6

```
#include <iostream.h>
#include <iomanip.h>
class SalesPerson
{
   public:
   SalesPerson(); //构造函数
   void setSales(); //由用户提供售货源
   void printAnnualSales();
   private:
   double sales[13]; //12 个月的售货源
   double totalAnnualSales(); //私有成员函数
};
//构造函数对数组的初始化
```

```
SalesPerson::SalesPerson()
{
   for(int i = 0;i <= 12;i ++)
      sales[i] = 0.0;
}
//设置十二月售货额的函数
void SalesPerson::setSales()
{
   for(int i = 1;i <= 12;i ++)
   {
      cout<<"Enter sales amount for month "
         <<i<<":";
      cin>>sales[i];
   }
}
//设置年度总售货额的私有成员函数
double SalesPerson::totalAnnualSales()
{
   double total = 0.0;
   for(int i = 1;i <= 12;i ++)
      total += sales[i];
   return total;
}
//打印年度总售货额
void SalesPerson::printAnnualSales()
{
   cout<<setprecision(2)
      <<setiosflags(ios::fixed|ios::showpoint)
      <<"\nThe total annual sales are: $"
      <<totalAnnualSales()<<endl;
}
int main(int argc, char* argv[])
{
   SalesPerson s;
   s.setSales();
   s.printAnnualSales();
   return 0;
}
```

运行上述程序，并输入：

Enter sales amount for month 1:5414.76↙

Enter sales amount for month 2:4292.38↙

Enter sales amount for month 3:4589.83↙

Enter sales amount for month 4:5534.03↙

Enter sales amount for month 5:4376.34↙

Enter sales amount for month 6:5698.45↙

Enter sales amount for month 7:4439.22↙

Enter sales amount for month 8:5893.57↙

Enter sales amount for month 9:4906.67↙

Enter sales amount for month 10:5123.45↙

Enter sales amount for month 11:4024.97↙

Enter sales amount for month 12:5923.92↙

运行结果如下：

The total annual sales are: $60120.59

类 SalesPerson 有一个记录十二个月售货额的数组，该数组被构造函数初始化为零。公有成员函数 setSales 可为数组设置用户提供的值，printAnnualSales 用来打印十二个月的收获总额（将售货额转换成美元形式输出）。下面对该程序再作一些解释。函数调用：

setprecision(2)

是将输出结果设置为保留两个小数位（如 23.47 美元），它是 "带参数的流操纵算子"的调用。凡是使用这类调用的程序都必须包含下面的预处理指令：

#include <iomanip>

下面一行语句：

setiosflag(ios::fixed | ios::showpoint);

的作用是以固定格式输出结果，showpoint 选项是要求显示小数点尾部的零，如显示出 47.00 不能显示为 47. 或 47。在 C++中，如果不设定这一选项，这样的数据将只被显示成 47。

内联函数和编译器

对于非内联函数，编译器在它的符号表里放入函数信号（即包括名字，参数个数、类型及函数的返回类型）。而对于内联函数，是将函数的整个代码放入符号表，这样可以省去函数调用的一些开销。

内联函数不能太复杂，假如语句较多，或有选择等控制语句，编译器仍然按非内联函数处理。

10.1.3 使用对象

定义了类以后，就可以定义类类型的变量(或对象)，例如：

```
void DrawLine(Point& p1, Point& p2)
{
   Point MidPoint;
…
}
```

函数 DrawLine 中定义了一个 Point 对象 MidPoint。

象结构一样，类类型的变量也能够作为函数参数，以值或引用的方式传递，也可以作为函数的返回值，以及在赋值语句中被复制。

我们已经知道：结构变量通过"."运算符访问其数据成员，对象数据要通过其成员函数进行修改或读出。

```
Point thePoint;
…
```

```
thePoint.SetPt(5, 10);
…
if(thePoint.GetX() < 0) …
```

可以看到，调用成员函数语法与结构变量访问其数据成员的语法相同：

对象名. 成员函数名 (实参表)

成员函数也可以通过指向对象的指针，调用形式为：

指向对象的指针->成员函数名 (实参表)

注意：

（1） 对私有数据成员的访问只能通过成员函数，下面的语句是非法的：

```
thePoint.xVal = 5; // 非法
```

（2） 不要混淆了类与对象的概念。类是用户定义的数据类型（不占内存），对象是类的实例（占内存单元），例如：

```
Point pt1, pt2, pt3;
```

定义了三个 Point 对象 pt1、pt2 和 pt3。

动态对象创建

有时我们知道程序中需要创建多少对象，但是多数情况下，我们不能预知所需对象的确切数量。比如公路交通管理系统必须同时处理多少辆汽车？一个三维建筑设计系统需要处理多少个模型？解决这些编程问题的方法，是要能够支持在运行时动态创建和销毁对象。

一个对象被动态创建时，依次发生两件事情：

1. 为对象分配内存；
2. 调用构造函数来初始化这块内存。

同样，一个对象被动态销毁时，按照顺序发生了下面两件事情：

1. 调用析构函数清除对象；
2. 释放对象的内存；

C++提供了两个运算符 new 和 delete，分别用来完成动态对象的创建和销毁。当我们用 new 创建一个对象时，就在堆里为对象分配内存并调用相应的构造函数。new 返回一个指向刚刚创建的对象的指针；当我们用 delete 销毁一个对象时，就调用相应的析构函数，释放掉分配的堆内存，delete 运算符的操作数是指向对象的指针。需要注意的是：用 new 创建的对象必须用 delete 销毁，否则，会出现内存泄漏。

举个例子说明利用 new 和 delete 动态创建和销毁对象的过程：

例 10-7

```
#include <iostream.h>
class Tree
{
  public:
  Tree(int height)
  {
    cout<<"tree object is creating"<<endl;
    this->height = height;
  }
  ~Tree()
```

```
    {
      cout<<"tree object is deleting"<<endl;
    }
    void display()
    {
      cout<<"this tree is "<<height<<" meters high"<<endl;
    }
  private:
    int height;
};
void main()
{
  Tree* tree = new Tree (100);
  tree->display();
  delete tree;
}
```

程序的输出结果如下：

```
tree object is creating
this tree is 100 meters high
tree object is deleting
```

main( )函数中的第一个语句，是用 new 运算符动态创建一个 Tree 类对象，new 后面括号中的 100，实际上是 new 创建对象时，传给构造函数的参数。main( )函数的第二个语句是调用对象的显示函数，打印出的结果显示树高为 100 米，可见 new 操作符确实调用了类的构造函数 display。main( )函数的最后一个语句用 delete 运算符销毁用 new 创建的对象。对象一旦被销毁后，就不再存在。如果继续访问对象 tree 的数据成员或成员函数，则程序会产生错误。

10.1.4 this 指针

C++中，定义了一个 this 指针，用它指向调用非静态成员函数的对象。也就是说，this 指针仅能在类的成员函数中访问，它指向调用该函数的对象，在后面我们要介绍的静态成员函数没有 this 指针。

当一个非静态的成员函数被一个对象调用时，对象的地址作为一个隐含的参数传给被调用的函数。例如：

```
myDate.setMonth( 3 );
```

可被解释为:

```
setMonth( &myDate, 3 );
```

下面的成员函数 setMonth，可用两种方法实现：

```
void Date::setMonth( int mn )//使用隐含的 this 指针
{
  month = mn;
```

```
}
void Date::setMonth( int mn )//显式使用 this 指针
{
   this->month = mn;
}
```

虽然显式使用 this 指针的情况并不是很多，但是，this 指针有时是有用的。例如，下面的赋值是不允许的：

```
void Date::setMonth( int month)
{
   month = month;
}
```

但可以用 this 指针来解决：

```
void Date::setMonth( int month)
{
   this->month = month;
}
```

类的每个成员函数都访问特殊的指针－this。This 指针保留了激活成员函数的对象地址（也就是说，this 总是指向目标对象）。This 指针仅仅在成员函数内部是合法的，而且名称 this 是 C++的保留字。

每个成员函数所收到的第一个参数就是 this 指针。程序员没有必要明确定义这个 this 指针，但是，它总是存在的。This 指针通常是每个成员函数（非静态）隐含的第一个参数。编译器在每个成员函数的声明中插入这个隐含参数。当成员函数使用类的成员的绝对名称的任何时候，它隐式使用 this 指针。编译器在引用成员函数内部的每个表达式中插入 this 指针（如果用户并没有这样做的话）。

10.1.5 一个实例

例 8-8：Set 是一个集合类实例。集合中的元素是无序的，且不能重复。为了简便起见，限制该集合类的元素为整型，且元素个数是有限的。

例 10-8-a

```
1. #include <iostream.h>
2. const maxCard  = 100;
3. enum bool {false, true};
4. class Set  {
5. public:
6. void EmptySet (void) { card = 0; }
7. bool Member (const int);
8. void AddElem (const int);
9. void RmvElem  (const int);
10. void Copy (Set&);
11. bool Equal (Set&);
12. void Intersect (Set&, Set&);
13. void Union (Set&, Set&);
```

```
14. void Print (void);
15. private:
16. int elems[maxCard]; // 集合元素
17. int card; // 集合元素个数
18. };
```

表 10-3 注释

2.	MaxCard 表示集合中的最大元素个数
6.	EmptySet 通过设置集合中的元素个数为 0，清除集合中的内容
7.	Member 检查一个给定的数是否是集合中的元素
8.	AddElem 增加一个新的元素到集合中，如果该元素已经在集合中，则集合保持不变，否则，在不使集合溢出的前提下，插入该元素
9.	RmvElem 删除集合中的一个元素
10.	Copy 复制一个集合到另一个集合
11.	Equal 检查两个集合是否相等，两个集合相等的条件是：两个集合所包含的元素是完全相同的
12.	Intersect 比较两个集合，生成一个新的集合，且该集合的元素同时包含在另外两各集合中。例如，集合{2,5,3}和{7,5,2}通过 Intersect 运算后，生成的集合为{2,5}
13.	Union 比较两个集合，生成一个新的集合，且该集合的元素包含另两个集合的全部元素。例如，集合{2,5,3} 和 {7,5,2}通过 union 运算后，生成的集合为{2,5,3,7}
14.	Print 打印一个集合。例如，包含元素 5, 2, 和 10 的集合打印为{5,2,10}
16.	elems 数组表示集合元素
17.	card 表示集合中元素的个数

在类外定义的成员函数也称为类的实现，Set 类的实现如下：

例 10-8-b

```
bool Set::Member (const int elem)
{
   for (register i = 0; i < card; ++i)
      if (elems[i] == elem)
         return true;
   return false;
}

void Set::AddElem (const int elem)
{
   if (Member(elem))
      return;
   if (card < maxCard)
```

```
        elems[card++] = elem;
    else
    cout << "Set overflow\n";
}

void Set::RmvElem  (const int elem)
{
    for (register i = 0; i < card; ++i)
        if (elems[i] == elem)  {
            for (; i < card-1; ++i) // shift elements left
                elems[i] = elems[i+1];
            --card;
        }
}

void Set::Copy (Set &set)
{
    for (register i = 0; i < card; ++i)
        set.elems[i] = elems[i];
    set.card = card;
}

bool Set::Equal (Set &set)
{
    if (card != set.card)
        return false;
    for (register i = 0; i < card; ++i)
        if (!set.Member(elems[i]))
            return false;
    return true;
}

void Set::Intersect (Set &set, Set &res)
{
    res.card = 0;
    for (register i = 0; i < card; ++i)
        if (set.Member(elems[i]))
            res.elems[res.card++] = elems[i];
}

void Set::Union (Set &set, Set &res)
{
```

```
    set.Copy(res);
    for (register i = 0; i < card; ++i)
        res.AddElem(elems[i]);
}

void Set::Print (void)
{
    cout << "{";
    for (int i = 0; i < card-1; ++i)
        cout << elems[i] << ",";
    if (card > 0) // no comma after the last element
        cout << elems[card-1];
    cout << "}\n";
}
```

下面的 main 函数创建了三个 Set 对象，并调用了它的一些成员函数：

例 10-8-c

```
void main (void)
{
    Set s1, s2, s3;

    s1.EmptySet(); s2.EmptySet(); s3.EmptySet();
    s1.AddElem(10); s1.AddElem(20); s1.AddElem(30); s1.AddElem(40);
    s2.AddElem(30); s2.AddElem(50); s2.AddElem(10); s2.AddElem(60);

    cout << "s1 = "; s1.Print();
    cout << "s2 = "; s2.Print();

    s2.RmvElem(50); cout << "s2 - {50} = "; s2.Print();
    if (s1.Member(20)) cout << "20 is in s1\n";
    s1.Intersect(s2,s3); cout << "s1 intsec s2 = "; s3.Print();
    s1.Union(s2,s3); cout << "s1 union s2 = "; s3.Print();
    if (!s1.Equal(s2)) cout << "s1 /= s2\n";
}
```

运行该程序，输出结果如下：

```
s1 = {10,20,30,40}
s2 = {30,50,10,60}
s2 - {50} = {30,10,60}
20 is in s1
s1 intsec s2 = {10,30}
s1 union s2 = {30,10,60,20,40}
s1 /= s2
```

在 C++中，有两种特殊的成员函数，即是构造函数和析构函数，下面分别予以介绍。

10.2.1 构造函数

变量应该被初始化，我们已经知道了简单变量的初始化、数组的初始化、结构和结构数组的初始化。对象也需要初始化，应该怎样初始化一个对象呢？

C++中定义了一种特殊的初始化函数，称之为构造函数。当对象被创建时，构造函数自动被调用。构造函数有一些独特的地方：函数的名字与类名相同，它也没有返回类型和返回值

例如：

```
class Point {
  public:
  Point(int x = 0, int y = 0);
  …
};
```

类 Point 的构造函数有两个参数，它们是赋给 xVal、yVal 的初始值，该构造函数的原型也为实参指定了缺省值 0。

```
Point::Point(double x, double y)
{
  xVal = x;
  yVal = y;
}
```

我们也可以把该构造函数定义为内联函数的形式：

```
class Point {
  Point (int x=0, int y=0) {xVal = x; yVal = y;} //  构造函数
  …
};
```

如果不采用 Point 构造函数，我们将不得不调用 SetPt 函数初始化。可见，使用 Point 构造函数方便了编程，简化了程序代码。

现在，我们可以定义 Point 类对象并立即初始化它们：

```
Point pt1(10, 20); // xVal 和 yVal 的初值分别为 10 和 20
Point pt2; // xVal 和 yVal 的初值均为 0
```

构造函数也可以重载：

```
class Point {
  int xVal, yVal;
public:
  Point (int x, int y) { xVal = x; yVal = y; }
  Point (float, float); //极坐标
  Point (void) { xVal = yVal = 0; }
  …
};

Point::Point (float len, float angle) //极坐标
{
```

```
    xVal = (int) (len * cos(angle));
    yVal = (int) (len * sin(angle));
}
```

创建 Point 对象时，可以使用这三种构造函数中的任一个。

```
Point pt1(10, 20); // 笛卡儿坐标
Point pt2(60.3, 3.14); // 极坐标
Point pt3; // 原点
```

我们也可以为 Set 类定义构造函数：

```
class Set {
  public:
    Set (void) { card = 0; }
    //...
};
```

这样，当创建对象时，便不再需要调用 EmptySet 初始化，构造函数可确保每一个对象的初始状态为空。

Set 类的最大元素个数是固定的，即为 maxCard。其实，我们可以将 elems 定义为整型指针，而不是整型数组，就可以控制 Set 类的最大元素个数。将 maxCard 变成 Set 类的一个成员（对不同的集合对象，maxCard 可能便不再相同），并在构造函数中定义一个形参指定所希望的的尺寸，构造函数只要简单地分配一个动态数组，并初始化 maxCard 和 card 即可：

```
class Set
{
public:
    Set (const int size);
    //...
private:
    int *elems; // 集合元素
    int maxCard; // 集合最大尺寸
    int card; // 集合元素个数
};

Set::Set (const int size)
{
    elems = new int[size];
    maxCard = size;
    card = 0;
}
```

现在便可以定义不同的最大尺寸：

```
Set ages(10), heights(20), primes(100);
```

注意：构造函数是在对象创建时被调用，对象何时创建与对象的作用域有关。例如，全局对象是在程序开始执行时创建，自动对象是当进入其作用域时创建，动态对象则是在使用 new 运算符时创建。

当用 new 运算符创建动态数组时，如：

```
elems = new int[size];
```

调用构造函数的顺序依次是 elems[0]、elems[1]、…、elems[size-1]。由于 new 的调用格式是类型后面跟[元素个数]，不能再跟构造函数的参数，所以数组元素的初始化只能调用无参的构造函数，如果没有无参的构造函数，则会出现编译错误

用构造函数确保初始化

无论用什么方法使用类对象，都要求用户必须正确地初始化。在 C＋＋中，可以通过一个叫做构造函数的特殊函数来保证每个对象正确地初始化。对象被创建时，构造函数被自动调用。所以，在我们使用一个对象之前，它的初始化就已经完成了。构造函数的特点是没有类型、没有返回值、名字与类名相同。下面是一个简单的类，它显式定义了构造函数：

```
class X
{
   int i;
public:
   X(); //构造函数
};
```

当用该类定义一个对象时：

```
void f()
{
   X a;
   //…
}
```

定义对象 a 与定义一个整型变量的方法并没有什么区别：为这个对象分配内存。但是，与定义一个整型变量也有不同的地方：当对象 a 创建时，会自动调用其构造函数。a 的构造函数没有参数，实际上，构造函数也可以有参数。例如：Tree 类的构造函数有一个整型参数，用以指定树的高度，创建 Tree 类对象的方法如下：

```
Tree t(12);// 12 英尺高的树
```

Tree (int)的构造函数带一个整型参数，创建 Tree 对象时，必须指定该参数，用下面的语句创建 Tree 对象是错误的：

```
Tree t;
```

综上所述，构造函数是一个有着特殊名字，在对象创建时被自动调用的一种函数，它的功能就是完成类的初

10.2.2 成员初始化表

一、常量成员

二、引用成员

三、类对象成

我们已经知道，类中数据成员可以通过构造函数来初始化，例如：

程序段 10-2

```
class Image {
public:
```

```
    Image(const int w, const int h);
private:
    int width;
    int height;
    //...
};
Image::Image  (const int w, const int h)
{
    width = w;
    height = h;
    //...
}
```

其实，类 Image 中的 width 和 height 的初始化也可以通过成员初始化表来初始化，例如：

程序段 10-3

```
class Image {
public:
    Image(const int w, const int h);
private:
    int width;
    int height;
    //...
};
Image::Image  (const int w, const int h) : width(w), height(h)
{
    //...
}
```

这个定义的效果是 width 被 w 初始化，height 被 w 初始化。用成员初始化表初始化类成员与用构造函数初始化类成员的区别在于：前者的初始化是在构造函数被执行以前进行的。

成员初始化表可用于初始化类的任意数据成员（后面要介绍的静态数据成员除外），它被放在构造函数的头和体之间，并用冒号将它与构造函数的头分隔开。它由逗号分隔的数据成员表组成，初值放在一对圆括号中

一、常量成员

类成员也可以定义为常量，例如：

```
class Image
{
    const int width;
    const int height;
```

```
  //...
};
```

但是，常量数据成员不能用如下的方法被初始化：

```
class Image {
  const int width = 256; //  非法!
  const int height = 168; //  非法!
  //...
};
```

初始化常量数据成员的正确方法是用成员初始化表：

程序段 10-4

```
class Image {
  public:
    Image(const int w, const int h);
  private:
    const int width;
    const int height;
    //...
};
Image::Image  (const int w, const int h) : width(w), height(h)
{
  //...
}
```

常量数据成员不能用于定义数组成员的尺寸，例如：

程序段 10-5

```
class Set {
  public:
    Set (void) : maxCard(10) { card = 0; }
    //...
  private:
    const maxCard;
    int elems[maxCard]; //非法!
    int card;
};
```

这是因为编译时，常量 maxCard 还没有确定的值，构造函数要到程序运行以后才会被调用。

二、引用成员

类数据成员也可以被定义为引用，例如：

```
class Image {
  int width;
```

```
    int height;
    int &widthRef;
    //...
};
```

正象常量数据成员一样，引用数据成员也不能用下面的方法初始化：

```
class Image {
    int width;
    int height;
    int &widthRef = width; //  非法!
    //...
};
```

初始化引用数据成员的正确方法是使用初始化表：

程序段 10-6

```
class Image {
    public:
        Image(const int w, const int h);
    private:
        int width;
        int height;
        int &widthRef;
        //...
};
Image::Image  (const int w, const int h) : widthRef(width)
{
    //...
}
```

这使得 widthRef 引用 width。

三、类对象成员

类数据成员也可以是另一个类的对象。例如，一个矩形数据可用两个点数据成员，分别表示矩形的左上角点和右下角点：

```
class Rectangle {
    public:
    Rectangle (int left, int top, int right, int bottom);
    //...
private:
    Point topLeft;
    Point botRight;
};
```

矩形的构造函数也应该初始化两个类对象成员，假定 Point 类有构造函数，矩形两个数据成员 topLeft

和 botRight 可放在 Rectangle 构造函数的初始化表中初始化：

```
Rectangle::Rectangle  (int left, int top, int right, int bottom)
   : topLeft(left,top), botRight(right,bottom)
{
}
```

如果 Point 构造函数没有参数，或所有参数都有缺省的参数，上面的成员初始化表可以省略，但是，构造函数仍然被隐含地调用。

初始化化的顺序如下：首先 topLeft 的构造函数被调用，接着调用 botRight 的构造函数，最后调用 Rectangle 的构造函数。

TopLeft 初始化在 botRight 之前的原因不是因为：在成员初始化表中，TopLeft 位于 botRight 之前，而是因为在 Rectangle 类的定义中，TopLeft 位于 botRight 之前。所以，如果构造函数的定义改成下面的形式，并不会影响到构造函数调用的顺序。

```
Rectangle::Rectangle  (int left, int top, int right, int bottom)
   : botRight(right,bottom), topLeft(left,top)
{
}
```

在与构造函数赋值相比，成员初始化列表占用较少的开销，因此具有较高的效率。更重要的是成员初始化列表是用来初始化类的 const 或引用类型数据成员的唯一方法，它们不能采用赋值操作。

10.2.3 析构函数

我们已经知道，当对象创建时，会自动调用构造函数进行初始化。当对象销毁时，也会自动调用析构函数进行一些清理工作。与构造函数类似的是：析构函数也与类同名，但在名字前有一个~符号，析构函数也没有返回类型和返回值。但析构函数不带参数，不能重载，所以析构函数只有一个。

若一个对象中有指针数据成员，该指针数据成员指向某一个内存块。在对象销毁前，往往通过析构函数释放该指针指向的内存块。例如，Set 类中 elems 指针指向一个动态数组，我们应该给 Set 类再定义一个析构函数，使 elems 指向的内存块能够在析构函数中被释放：

```
class Set
{
public:
   Set (const int size);
   ~Set(void) {delete elems;} // 析构函数
   //...
private:
   int *elems; // 集合元素
   int maxCard; // 集合最大尺寸
   int card; // 集合元素个数
};
```

下面我们通过一个例子，分析析构函数是如何工作的：

```
void Foo (void)
{
   Set s(10);
   //...
```

}

当函数 Foo 被调用时，创建对象 s，并调用其构造函数，为 s.elems 分配内存及初始化其它对象成员。在 Foo 函数返回以前，s 的析构函数被调用，释放 s.elems 指向的内存区。

注意：对象的析构函数在对象销毁前被调用，对象何时销毁也与其作用域有关。例如，全局对象是在程序运行结束时销毁，自动对象是在离开其作用域时销毁，而动态对象则是在使用 delete 运算符时销毁。析构函数的调用顺序与构造函数的调用顺序相

用析构函数确保对象的清除

我们常常不会忽略初始化的重要性，却很少想到清除的重要性。实际上，清除也很重要，例如，我们在堆中申请了一些内存，如果不在用完后就释放，就会造成内存泄露，它会导致应用程序运行效率降低，甚至崩溃，我们不可掉以轻心。在 C++中，提供了析构函数，保证对象清除工作的自动执行。

析构函数与构造函数不同点是：析构函数的功能与构造函数相反，也不带任何参数，所以，它不能被重载。当对象超出其作用域被销毁时，析构函数会被自动调用。

下面我们仍用 Tree 类例子来说明构造函数和析构函数的调用情况。

例 10-9

```
#include <iostream.h>
class Tree
{
    int height;
  public:
    Tree (int initialHeight); //Constructor
    ~Tree (); //Destructor
    void grow(int years);
    void printsize();
};
Tree:: Tree (int initialHeight)
{
  height = initialHeight;
}
Tree::~ Tree ()
{
  cout<<"inside tree destructor"<<endl;
  printsize();
}
void Tree::grow(int years)
{
  height += years;
}
void Tree::printsize()
{
  cout<<"tree height is "<<height<<endl;
```

```
}
void main()
{
   cout<<"before opening brace"<<endl;
   {
      Tree t(12);
      cout<<"after tree creation"<<endl;
      t.printsize();
      t.grow(4);
      cout<<"before closing brace"<<endl;
   }
   cout<<"after closing brace"<<endl;
}
```

下面是上面程序的输出结果：

```
before opening brace
after tree creation
tree height is 12
before closing brace
inside tree destructor
tree height is 16
after closing brace
```

我们可以看到，析构函数在包含它的右括号处调用。

关于返回值

构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为 void 的函数是不同的。后者虽然也不返回任何值，但有返回类型，而构造函数和析构函数根本就没有返回类型。

10.2.4 缺省的构造函数和析构函数

前面我们介绍了构造函数和析构函数的特点、功能及应用。当用户未显式定义构造函数和析构函数时，编译器会隐式定义一个内联的、公有的构造函数和析构函数。

缺省的构造函数执行创建一个对象所需要的一些初始化操作，但它并不涉及用户定义的数据成员或申请的内存的初始化。

例 10-10

```
class C
{
   private:
   int n;
```

```
    char *p;
    public:
    virtual ~C() {}
};
void f()
{
    C obj; //隐式定义的构造函数被调用
}
```

类 C 没有显式定义构造函数，一个隐式缺省的构造函数被定义。在函数 f 中，定义对象 obj 时，调用该构造函数，但它并不初始化数据成员 n 和 p，也不为后者分配内存。

同样，缺省的的析构函数也不涉及释放用户申请的内存的释放等清理工作。

对一个简单变量的初始化方法是用一个常量或变量初始化另一个变量，例如：

```
int m = 80;
int n = m;
```

我们已经会用构造函数初始化对象，那么我们能不能象简单变量的初始化一样，直接用一个对象来初始化另一个对象呢？答案是肯定的。我们以前面定义的 Point 类为例：

```
Point pt1(15, 25);
Point pt2 = pt1;
```

后一个语句也可以写成：

```
Point pt2( pt1);
```

它是用 pt1 初始化 pt2，此时，pt2 各个成员的值与 pt1 各个成员的值相同，也就是说，pt1 各个成员的值被复制到 pt2 相应的成员当中。在这个初始化过程当中，实际上调用了一个复制构造函数。当我们没有显式定义一个复制构造函数时，编译器会隐式定义一个缺省的复制构造函数，它是一个内联的、公有的成员，它具有下面的原型形式：

```
Point:: Point (const Point &);
```

可见，复制构造函数与构造函数的不同之处在于形参，前者的形参是 Point 对象的引用，其功能是将一个对象的每一个成员复制到另一个对象对应的成员当中。

虽然没有必要，我们也可以为 Point 类显式定义一个复制构造函数：

```
Point:: Point (const Point &pt)
{
    xVal=pt. xVal;
    yVal=pt. yVal;
}
```

如果一个类中有指针成员，使用缺省的复制构造函数初始化对象就会出现问题。为了说明存在的问题，我们假定对象 A 与对象 B 是相同的类，有一个指针成员，指向对象 C。当用对象 B 初始化对象 A 时，缺省的复制构造函数将 B 中每一个成员的值复制到 A 的对应的成员当中，但并没有复制对象 C。也就是说，对象 A 和对象 B 中的指针成员均指向对象 C，参见图 10-1。

实际上，我们希望对象 C 也被复制，得到 C 的对象副本 D。否则，当对象 A 和 B 销毁时，会对对象 C 的内存区重复释放，而导致错误。为了使对象 C 也被复制，就必须显式定义复制构造函数。下面我们以 string 类为例说明，如何定义这个复制构造函数。

例 10-11

```
class String
{
  public:
    String(); //构造函数
    String(const String &s); //复制构造函数
    ~String(); //析构函数

    // 接口函数
    void set(char const *data);
    char const *get(void);

  private:
    char *str;//数据成员 ptr 指向分配的字符串
};

String ::String(const String &s)
{
  str = new char[strlen(s.str) + 1];
  strcpy(str, s.str);
}
```

我们也常用无名对象初始化另一个对象，例如：

```
Point pt = Point(10, 20);
```

类名直接调用构造函数就生成了一个无名对象，上式用左边的无名对象初始化右边的 pt 对象。

构造函数被调用通常发生在以下三种情况，第一种情况就是我们上面看到的：用一个对象初始化另一个对象时；第二种情况是当对象作函数参数，实参传给形参时；第三种情况是程序运行过程中创建其它临时对象时。下面我们再举一个例子，就第二种情况和第三种情况进行说明：

```
Point foo(Point pt)
{
  …
  return   pt;
}
void main()
{
  Point pt1 = Point(10, 20);
  Point pt2;
  …
```

```
    pt2=foo(pt);
    …
}
```

在 main 函数中调用 foo 函数时，实参 pt 传给形参 pt，将实参 pt 复制给形参 pt，要调用复制构造函数，当函数 foo 返回时，要创建一个 pt 的临时对象，此时也要调用复制构造函数。

缺省的复制构造函数

在类的定义中，如果没有显式定义复制构造函数，C++编译器会自动地定义一个缺省的复制构造函数。下面是使用复制构造函数的一个例子：

例 10-12

```
#include <iostream.h>
#include <string.h>
class withCC
{
  public:
  withCC(){}
  withCC(const withCC&)
  {
    cout<<"withCC(withCC&)"<<endl;
  }
};

class woCC
{
  enum{bsz = 100};
  char buf[bsz];
public:
  woCC(const char* msg = 0)
  {
    memset(buf, 0, bsz);
    if(msg) strncpy(buf, msg, bsz);
  }
  void print(const char* msg = 0)const
  {
    if(msg) cout<<msg<<":";
    cout<<buf<<endl;
  }
};

class composite
{
```

```
    withCC WITHCC;
    woCC WOCC;
  public:
    composite() : WOCC("composite()"){}
    void print(const char* msg = 0)
    {
      WOCC.print(msg);
    }
};

void main()
{
  composite c;
  c.print("contents of c");
  cout<<"calling composite copy-constructor"<<endl;
  composite c2 = c;
  c2.print("contents of c2");
}
```

类 withCC 有一个复制构造函数，类 woCC 和类 composite 都没有显式定义复制构造函数。如果在类中没有显式定义复制构造函数，则编译器将自动地创建一个缺省的构造函数。不过在这种情况下，这个构造函数什么也不作。

类 composite 既含有 withCC 类的成员对象又含有 woCC 类的成员对象，它使用无参的构造函数创建 withCC 类的对象 WITHCC（注意内嵌的对象 WOCC 的初始化方法）。

在 main()函数中，语句：

```
composite c2 = c;
```

通过对象 C 初始化对象 c2，缺省的复制构造函数被调用。

最好的方法是创建自己的复制构造函数而不要指望编译器创建，这样就能保证程序在我们自己的控制之下。

我们已经学习了局部作用域和全局作用域，下面介绍类作用域，所有的类成员是属于类作用域的。

类本身可被定义在三种作用域内：

1. 全局作用域。这是所谓全局类，绝大多数的 C++类是定义在该作用域中，我们在前面定义的所有类都是在全局作用域中。
2. 在另一个类的作用域中。这是所谓嵌套类，即一个类包含在另一个类中。
3. 在一个块的局部作用域中。这是所谓局部类，该类完全被块包含。

例如：

```
int fork (void); //  全局 fork
class Process {
  int fork (void);
  //...
};
```

成员函数 fork 隐藏了全局函数 fork，前者能通过单目域运算符调用后者：

```
int Process::fork (void)
{
  int pid = ::fork(); // 使用全局 fork
  //...
}
```

下面举一个嵌套类的例子。

例 10-13

```
class Rectangle {
  public:
  Rectangle (int, int, int, int);
  //..
private:
  class Point {
    public:
      Point (int, int);
    private:
      int x, y;
  };
  Point topLeft, botRight;
};
```

类 Point 嵌套在 Rectangle 类中，Point 成员函数可定义为内联的，也可在全局作用域中，但后者对成员函数名需要更多的限制，例如：

```
Rectangle::Point::Point (int x, int y)
{
  //...
}
```

同样，在类域外访问嵌套类需要限制类名，例如：

```
Rectangle::Point pt(1,1);
```

下面我们再看一个局部类的例子：

例 10-14

```
void Render (Image &image)
{
  class ColorTable {
  public:
    ColorTable(void) { /* ... */ }
    AddEntry(int r, int g, int b) { /* ... */ }
    //...
  };
```

```
    ColorTable colors;
    //...
}
```

类 ColorTable 是在函数 Render 中的局部类。与嵌套类不同的是：局部类不能在其局部作用域外访问，例如：

ColorTable ct; // 非法，没有定义 ColorTable 类!

局部类必须完全定义在局部作用域内。所以，它的所有成员函数必须是内联的，这就决定了局部类的成员函数都是很简单的。

一些成员函数改变对象，一些成员函数不改变对象

例如：

```
int Point::GetY()
{
  return yVal;
}
```

这个函数被调用时，不改变 Point 对象，而下面的函数改变 Point 对象：

```
void Point:: SetPt (int x, int y)
{
  xVal=x;
  yVal=y;
}
```

为了使成员函数的意义更加清楚，我们可在不改变对象的成员函数的函数原型中加上 const 说明：

例 10-15

```
class Point
{
  public:
    int GetX() const;
    int GetY() const;

    void SetPt (int, int);
    void OffsetPt (int, int);
  private:
    int xVal, yVal;
};
```

const 成员函数应该在函数原型说明和函数定义中都增加 const 限定：

例 10-16

```
int Point::GetY() const
```

```
{
    return yVal;
}

class Set {
public:
    Set (void){ card = 0; }
    bool Member(const int) const;
    void AddElem(const int);
    //...
};

bool Set::Member (const int elem) const
{
    //...
}
```

非常量成员函数不能被常量成员对象调用，因为它可能企图修改常量的数据成员：

```
const Set s;
s.AddElem(10); // 非法: AddElem 不是常量成员函数
s.Member(10); // 正确
```

但构造函数和析构函数对这个规则例外，它们从不定义为常量成员，但可被常量对象调用（被自动调用）。它们也能给常量的数据成员赋值，除非数据成员本身是常量。

为什么需要 const 成员函数？

我们定义的类的成员函数中，常常有一些成员函数不改变类的数据成员，也就是说，这些函数是"只读"函数，而有一些函数要修改类数据成员的值。如果把不改变数据成员的函数都加上 const 关键字进行标识，显然，可提高程序的可读性。其实，它还能提高程序的可靠性，已定义成 const 的成员函数，一旦企图修改数据成员的值，则编译器按错误处理。

const 成员函数和 const 对象

实际上，const 成员函数还有另外一项作用，即常量对象相关。对于内置的数据类型，我们可以定义它们的常量，用户自定义的类也一样，可以定义它们的常量对象。例如，定义一个整型常量的方法为：

```
const int i=1 ;
```

同样，也可以定义常量对象，假定有一个类 classA，定义该类的常量对象的方法为：

```
const classA a(2);
```

这里，a 是类 classA 的一个 const 对象，"2"传给它的构造函数参数。const 对象的数据成员在对象寿命期内不能改变。但是，如何保证该类的数据成员不被改变呢？

为了确保 const 对象的数据成员不会被改变，在 C++中，const 对象只能调用 const 成员函数。如果一个成员函数实际上没有对数据成员作任何形式的修改，但是它没有被 const 关键字限定的，也不能被常量对象调用。下面通过一个例子来说明这个问题：

例 10-17

```
class C
{
   int X;
public:
   int GetX()
   {
      return X;
   }
   void SetX(int X)
   {
      this->X = X;
   }
};
void main()
{
   const C constC;
   cout<<constC.GetX();
}
```

如果我们编译上面的程序代码，编译器会出现错误提示：constC 是个常量对象，它只能调用 const 成员函数。虽然 GetX( )函数实际上并没有改变数据成员 X，由于没有 const 关键字限定，所以仍旧不能被 constC 对象调用。如果我们将上述加粗的代码：

```
int GetX()
```

改写成：

```
int GetX()const
```

再重新编译，就没有问题了。

const 成员函数的使用

const 成员函数表示该成员函数只能读类数据成员，而不能修改类成员数据。定义 const 成员函数时，把 const 关键字放在函数的参数表和函数体之间。有人可能会问：为什么不将 const 放在函数声明前呢？因为这样做意味着函数的返回值是常量，意义完全不同。下面是定义 const 成员函数的一个实例：

```
class X
{
   int i;
   public:
   int f() const;
};
```

关键字 const 必须用同样的方式重复出现在函数实现里，否则编译器会把它看成一个不同的函数：

```
int X::f() const
{
   return i;
}
```

如果 f( )试图用任何方式改变 i 或调用另一个非 const 成员函数，编译器将给出错误信息。任何不修改成员数据的函数都应该声明为 const 函数，这样有助于提高程序的可读性和可靠性。

前面我们已经看到：每一个类对象有其公有或私有的数据成员，每一个 public 或 private 函数可以访问其数据成员。
有时，可能需要一个或多个公共的数据成员，能够被类的所有对象共享。在 C++中，我们可以定义静态（static）的数据成员和成员函数。

10.6.1 静态数据成员

要定义静态数据成员，只要在数据成员的定义前增加 static 关键字。静态数据成员不同于非静态的数据成员，一个类的静态数据成员仅创建和初始化一次，且在程序开始执行的时候创建，然后被该类的所有对象共享；而非静态的数据成员则随着对象的创建而多次创建和初始化。

下面是静态数据成员定义的实例：

例 10-18

```
class Test
{
  public:
    static int public_int;
  private:
    static int private_int;
};

void main()
{
  Test::public_int = 145; //  正确
  Test::private_int = 12; //  错误，不能访问私有的数据成员
}
```

从上例我们可以看到：静态数据成员的访问方式是：
类名::静态数据成员名
但是,不能直接访问私有的数据成员。其实，上面的程序段只是为了介绍如何访问静态数据成员，不能通过编译器的的编译和连接。

一、私有的静态数据成员

为了说明私有的静态数据成员的使用，考虑下面的程序段：

```
class Directory
{
  public:
  ...
  private:
    // 数据成员
    static char path[];
```

```
};
```

数据成员 path[]是一个私有的静态变量，在程序执行过程中，仅一个 Directory::path[]存在，即使有多个 Directory 类的对象。静态的数据成员能够被类的成员函数访问，但不能在构造函数中初始化。这是因为静态数据成员在构造函数被调用之前就已经存在了。静态数据成员可以在定义时初始化，且必须在类和所有的成员函数之外，与全局变量初始化的方法一样。例如，类 Directory 的数据成员的定义与初始化方法如下：

```
// 静态数据成员的定义与初始化
char Directory::path[200] = "/usr/local";
// 无参的构造函数
Directory::Directory()
{
  ...
}
```

在类中的静态数据成员的定义，只是说明该类有这么一个数据成员，并没有为该数据成员分配内存。就象非静态数据成员是在创建对象时分配内存一样，静态数据成员是在初始化时分配内存的。所以，在定义静态的数组成员时，可以省略它的尺寸（数组元素的个数），但在初始化时，必须确定数组的尺寸。

二、公有的静态数据成员

数据成员也可以是公有的，不过我们很少定义公有的数据成员，因为它破坏了数据隐藏的原则。如果 Directory 类的成员 path[]定义为公有的，则该成员可在类外的任一地方访问：

```
void main()
{
  strcpy(Directory::path, "/usr/local/pub"); //修改 path 的值
  …
}
```

path 仍然必须先初始化:

```
char Directory::path[200];
```

为什么需要静态数据成员？

类的静态数据成员拥有一块单独的存储区，而不管我们创建了多少个该类的对象。也就是说，静态数据成员被类的所有对象共享，它是属于类，而不是属于对象的。所有对象的静态数据成员都共享一块静态存储空间，可以节省内存，也为对象之间提供了一种互相通信的方法。静态数据成员的作用域在类内，也有 public（公有）、private（私有）或者 protected（保护）三种访问权限。

静态数据成员的存储空间分配

对于非静态数据成员而言，有多少个对象，就有多少个不同的内存单元，它们分布在各个对象的存储空间中。静态数据成员不同于非静态数据成员，不管有多少个对象，它们都共享相同的内存单元。所以，静态数据成员不应该属于任何一个对象，说它属于类更确切。

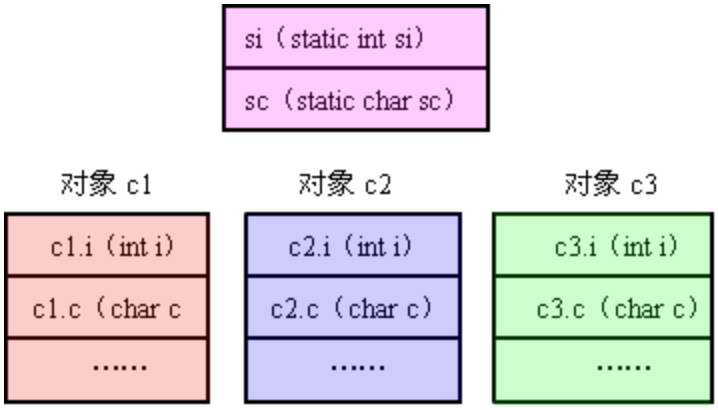
我们用一个例子来说明这种存储空间的关系， 对于类：

```
class C
{
  static int si;
  static int sc;
```

```
    int i;
    char c;
    //……
};
```

如果我们创建了类 C 的三个对象 c1、c2 和 c3，那么它们的数据成员的存储关系如图 10-2 所示：

图 10-2



静态数据成员的初始化

静态数据成员的初始化必须在类外，例如：

```
class A
{
   static int i;
   public:
   //……
};
```

静态数据成员 i 的初始化方法为："int A::i=1;"。可见：它与全局变量的初始化的方法的不同之处在于，有类名和作用域分隔符指定 i 的范围。

我们还可以定义静态的数组成员，例如：

例 10-19

```
class Values
{
   static const int size;
   static const float table[4];
   static char letters[5];
};
const int Values::size = 100;
const float Values::table[4] = {1.1, 2.2, 3.3, 4.4};
```

```
char Values::letters[5] = {'a', 'b', 'c', 'd', 'e'};
void main(){}
```

本例创建了静态的成员变量 size，静态的数组成员 table 和 letters。静态的数组成员的初始化方法与数组的初始化类似，但要有类名和作用域分隔符限制。

成员常量

我们先看下面一个实例：

```
class A
{
  const size = 100; //illegal
  int array[size]; //illegal
};
```

类 A 是想定义一个常量数组，但这个类的定义是错误的。因为定义类常量数据成员时，不能同时进行初始化。定义类时，只是说明类有那些数据成员，而不涉及到内存单元的分配，类数据成员存储单元的分配是在对象初始化时进行的。

我们也可以把常量数据成员定义成静态的，同静态成员变量初始化一样，静态成员常量初始化也在类外，例如：

例 10-20

```
class A
{
  static const int size;
  int array[size];
  public:
  // ...
};
const int A::size = 100; //definition
```

嵌套类和局部类中静态成员变量的使用

可以很容易地把一个静态数据成员放在一个嵌套类中，它是非嵌套类中静态数据成员情况的扩展，只将它的范围进行另一个级别的指定就可以了。然而，在局部类（如在函数内部定义的类）中不能有静态数据成员。例如：

例 10-21

```
// Nested class CAN have static data members:
class outer
{
  class inner
  {
    static int i; //OK
  };
};
```

```
int outer::inner::i = 100;

// Local class CANNOT have static data members:
void f()
{
  class local
  {
    static int i; //error
  };
}
void main(){}
```

10.6.2 静态成员函数

除静态数据成员外，C++也允许定义 static 成员函数。静态的数据成员被所有的对象共享，也就是说，静态数据成员不属于对象，而是属于类的。与静态数据成员类似，静态成员函数也是属于类的。

注意：静态成员函数仅能访问静态的数据成员，不能访问非静态的数据成员，也不能访问非静态的成员函数，这是由于静态的成员函数没有 this 指针。类似于静态的数据成员，公有的、静态的成员函数在类外的调用方式为：

类名::成员函数名(实参表)

不过,也允许用对象调用静态的成员函数。

下面举一个例子说明静态成员函数的调用方法:

例 10-22

```
class Directory
{
  public:
    ...
    // 静态公有的函数
    static void setpath(char const *newpath);

  private:
    // 静态字符串
    static char path[];
};

//静态数据成员的初始化
char Directory::path[199] = "/usr/local";

// 静态函数
void Directory::setpath(char const *newpath)
{
  strncpy(path, newpath, 199);
```

```
}

// 使用实例
void main()
{
  // 通过类名调用 setpath()成员函数
  Directory::setpath("/etc");

  //通过象调用 setpath()成员函数
  Directory dir;
  dir.setpath("/etc");
}
```

在上面的例子中，setpath()始一个公有静态的成员函数，C++也允许私有静态的成员函数，它只能被该类的其它成员函数调用，不能在类外被调用。

需要注意的是：静态的成员函数仅能访问静态的变量和静态的成员函数，不能访问非静态的数据成员和非静态的成员函数。

静态成员函数

我们也可以创建一个静态的成员函数。同静态数据成员类似，它也是属于类，而不是属于某一个对象的。

静态成员函数不能访问非静态的数据成员，它只能访问静态数据成员，也只能调用其它的静态成员函数。原因是：当前对象的地址（this）是被隐含地传递到被调用的函数的。但一个静态成员函数没有 this 指针，所以它无法访问非静态的成员函数。

下面是一个应用静态数据成员和静态成员函数的例子：

例 10-23

```
class X
{
  int i;
  static int j;
public:
  X(int I = 0):i(I)
  {
    //非静态成员函数可以访问
    //静态成员变量和成员函数
    j = i;
  }

  int val()const
  {
    return i;
```

```
    }

    static int incr()
    {
      //i++;          //Error:  静态成员函数
                      //不能访问非静态成员变量
      return ++j; //OK
    }

    static int f()
    {
      //val();          //Error:  静态成员函数
                        //不能访问非静态成员函数
      return incr(); //OK
    }
};

int X::j = 0;

void main()
{
    X x;
    X* pX = &x;
    x.f();
    pX -> f();
    X::f();
}
```

由于静态成员函数是属于类的，所以，静态成员函数的调用方法为：

类名::静态成员函数名();

但也允许用对象或指向对象的指针调用。例如，在 main()中，静态成员函数 f 分别被对象 x 和指向 x 对象的指针 pX 调用。

下面是一个有趣的例子：类 egg 有一个静态数据成员 E 和静态成员函数 instance，instance 的返回类型是指向 egg 对象的指针，类的构造函数是私有的。显然，我们不能创建该类的对象。但是该类确有一个对象存在，我们可以访问那个对象，也可以通过该对象访问其成员函数：

例 10-24

```
#include <iostream.h>
class egg
{
```

```
    static egg  E;
    int i;
    egg(int  I):i(I){}
public:
    static egg*  instance()
    {
      return &E;
    }
    int val()
    {
      return i;
    }
};

egg egg::E(100);

void main()
{
    //egg  x(1); //Error:  不能调用私有的构造函数
    //因此，不能创建新的对象
    cout<<egg::instance()->val()<<endl;
}
```

E 的初始化出现在类的声明完成后，所以编译器已有足够的信息为对象分配空间并调用构造函数。

我们知道：类的公有成员能够在类外访问，私有的成员只能被类的其它成员函数访问。在C++中，可以定义友元，如果某一个函数定义为类的友元，则该函数就可以访问该类的私有成员，下面举例说明友元函数的使用。

假定我们已经定义了两个集合类变量，一个为整型数集合，另一个为实型数集合：

程序段 10-7

```
class IntSet {
public:
    //...
private:
    int elems[maxCard];
    int card;
};

class RealSet  {
public:
    //...
```

```
private:
  float elems[maxCard];
  int card;
};
```

我们还要定义一个函数 SetToReal，其功能是将整型数集合转换为实型数集合。SetToReal 可以定义为 IntSet 类的成员函数。

程序段 10-8

```
void IntSet::SetToReal (RealSet &set)
{
  set.EmptySet();
  for (register i = 0; i < card; ++i)
    set.AddElem((float) elems[i]);
}
```

这个函数中，整型数集合中的每一个元素作为实参，调用实型数集合中的 AddElem 成员函数的开销较大。如果我们将 SetToReal 函数定义为 RealSet 类的友元，使得 SetToReal 函数能够访问 RealSet 类的私有成员，则可提高程序运行的效率。

程序段 10-9

```
class RealSet {
  //...
  friend void IntSet::SetToReal (RealSet&);
};

void IntSet::SetToReal (RealSet &set)
{
  set.card = card;
  for (register i = 0; i < card; ++i)
    set.elems[i] = (float) elems[i];
}
```

从上面我们可以看出：定义友元需用 friend 关键字，SetToReal 函数定义为类 RealSet 的友元后，就能够直接访问 RealSet 类的私有数据成员 elems，避免了函数调用的开销，从而提高了程序运行的效率。

其实，我们也可以把一个类定义为另一个类的友元，例如：

```
class A;
class B {
  //...
  friend class A;
};
```

类 A 是类 B 的友元，类 A 中的所有成员函数都能访问类 B 的私有成员。

全局函数也可以作为类的友元。例如，假定我们将 SetToReal 定义为全局函数，并定义为 IntSet 和 RealSet 两个类的友元：

程序段 10-10

```
class IntSet {
  //...
  friend void SetToReal (IntSet&, RealSet&);
};

class RealSet  {
  //...
  friend void SetToReal (IntSet&, RealSet&);
};

void SetToReal (IntSet &iSet, RealSet &rSet)
{
  rSet.card = iSet.card;
  for (int i = 0; i < iSet.card; ++i)
    rSet.elems[i] = (float) iSet.elems[i];
}
```

注意：虽然友元定义出现在类内，但它并不是类的成员。友元在类内定义的位置也是任意的，不管是出现在类的 private、pretected 或 public 部分，含义均相同。

虽然友元为我们进行程序设计提供了一定的方便性，但是面向对象的程序设计要求类的接口与类的实现分开，对对象的访问通过其接口函数进行。如果直接访问对象的私有成员，就破坏了面向对象程序的信息隐藏和封装特性，虽然提供了一些方便，但有可能是得不偿失的，所以，我们要慎用友元。

C＋＋的存取控制

C＋＋有三个关键字，用于设置类成员的访问权限，它们是：public、private 和 protected。

◇ public 意味着在其后声明的所有成员在类外可以访问；

◇ private 关键字则意味着，除了该类的成员函数之外，类外不能访问这些成员。如果试图访问私有成员，就会产生编译错误。类中缺省访问权限说明的的成员声明是 private 的；

◇ protected 和 private 区别在于：protected 成员可以被派生类访问，而 private 成员则不能被派生类访问。

友元的使用

类的私有成员只能在类内访问，那么有没有可能让类外的成员访问？其实，在 C++中，友元可以访问类的私有成员。友元不是类的成员，但必须在类的内部声明。因为"谁是我的朋友"必须由"我自己决定"。绝对不能允许下面的现象发生：李四说，"嘿，我是张三的朋友"，然后李四就开始使用张三的一切，包括它的私有财产。李四究竟是不是张三的朋友，必须由张三自己说了算。

通过 friend 关键字，可以把一个全局函数声明为友元，也可以把另一个类中的成员函数，甚至整个类都声明为友元，请看下面的例子：

例 10-25

```
class X; //Declaration
class Y
{
   void f(X*);
};
class X
{
private:
   int i;
public:
   void initialize();
   friend void g(X*, int); //Global friend
   friend void Y::f(X*); //class member friend
   friend class Z; //Entire class is a friend
   friend void h();
};
void X::initialize()
{
   i = 0;
}
void g(X* x, int i)
{
   x->i = i;
}
void Y::f(X* x)
{
   x->i = 100;
}
class Z
{
private:
   int j;
public:
   void initialize();
   void g(X* x);
};
void Z::initialize()
{
```

```
    j = 99;
}
void Z::g(X* x)
{
    x->i += j;
}
void h()
{
    X x;
    x.i = 200; //直接数据操作
}
void main()
{
    X x;
    Z z;
    z.g(&x);
}
```

类 Y 有一个成员函数 f()，它要访问 X 对象的私有数据成员 i，需将 Y::f(X*)声明为类 X 的友元。这里有一个难题，因为 C＋＋的编译器要求在使用任一变量之前必须声明，所以，类 Y 必须在它的成员 Y::f(X*)被声明为类 X 的一个友元之前声明。但声明 Y::f(X*)之前，又必须声明类 X。

解决的办法是：首先声明类 X，但是这个类 X 的声明并不完整，能行吗？在本例中是可以的，这是由于 Y::f(X*)的参数是指向 X 对象的指针，所以编译器允许在声明 Y::f(X*)之前做一个不完全的类型指定。仅仅是告诉编译器，有一个叫 X 的类。这样，在类 X 中，Y::f(X*)就可以成功地声明为一个友元函数。

再来看看其它两个友元函数，第一个声明将一个全局函数 g()作为一个友元，但 g()在这之前并没有定义，这表明 friend 可以同时声明全局函数和友元。同样，friend 可以同时声明类 Z 及类 Z 作为类 Y 的友元。

嵌套友元

对于嵌套类，也不能访问私有成员。嵌套类要访问私有成员也必须定义为友元。请看下面的例子：

例 10-26

```
#define SZ 20
class holder
{
private:
      int a[SZ];
  public:
      void initialize();
      class pointer
      {
```

```
    private:
        holder *h;
        int* p;
    public:
        void initialize(holder* H);
        //move around in the array.
        void next();
        void previous();
        void top();
        void end();
        //access values.
        int read();
        void set(int i);
    };
    friend holder::pointer;
};
```

类holder包含一个整型数组和一个嵌套类pointer。通过定义pointer为holder的友元，使得pointer成员可以访问 holder 类的私有成员：

friend holder : :pointer ;

关于友元的使用

如果一个函数被某一个类声明为 friend，就意味着它不是这个类的成员函数，但却可以访问该类的私有成员，而且它必须被列在类的定义中。虽然友元这种机制为我们进行程序设计提供了一定的方便性，但是，面向对象的程序设计要求类的接口与类的实现分开，对对象的访问通过其接口函数进行。如果直接访问对象的私有成员，就破坏了面向对象程序的信息隐藏和封装特性，虽然提供了一些方便，但有可能是得不偿失的，所以，我们要慎用友元。

【本章小结】

本章讲述了 C++语言中面向对象编程的基本概念基本方法。

在 C++语言中，通过 class 关键字可以定义类，类的成员包括数据成员和成员函数两种。用户定义了新的类之后，就可以定义该类的对象。C++中还定义了一个 this 指针，它仅能在类的成员函数中访问，它指向该成员函数所在的对象，即当前对象。

在 C++中，有两种特殊的成员函数，即构造函数和析构函数。它们分别负责对象的初始化和清除工作。复制构造函数的形参是本类对象的引用，它是用一个对象来初始化另一个对象。如果编程者没有显式定义构造函数（包括复制构造函数），C++编译器就隐式定义缺省的构造函数。

为了实现对象的常量化，C++引入了 const 函数的概念。const 函数不改变对象的数据成员，也不能调用非 const 函数。常量对象只能调用 const 函数；但构造函数和析构函数对这个规则例外，它们从不定义为常量成员，但可被常量对象调用（被自动调用）。

在 C++中，为了实现类的所有对象对一个或多个类成员的共享，可以定义静态数据成员和静态成员函数。一个类的静态数据成员仅创建和初始化一次，且在程序开始执行的时候创建，然后被该类的所有对象共享；而非静态的数据成员则随着对象的创建而多次创建和初始化。与静态数据成员

类似，静态成员函数也是属于类的。静态成员函数仅能访问静态的数据成员，不能访问非静态的数据成员，也不能访问非静态的成员函数，这是由于静态的成员函数没有 this 指针。

一般来说，类的公有成员能够在类外访问，私有的成员只能被类的其它成员函数访问。但是通过 C++中提供的友元概念，可以实现类的私有成员的访问。虽然友元为我们进行程序设计提供了一定的方便性，但是面向对象的程序设计要求类的接口与类的实现分开，对对象的访问通过其接口函数进行。如果直接访问对象的私有成员，就破坏了面向对象程序的信息隐藏和封装特性，虽然提供了一些方便，但有可能是得不偿失的，所以，我们要慎用友元。

【学习目标】

◇ 理解基类和派生类的概念

◇ 掌握继承的概念和用法

◇ 理解虚函数、纯虚函数和抽象基类的概念和用法

◇ 理解多态的概念和作用，掌握多态的实现方法

◇ 理解虚析构函数的概念和作用，掌握虚析构函数的用法

【重点和难点】

重点：继承的概念和使用；虚函数的概念和使用；如何用虚函数实现多态。

难点：多继承；多态的实现；抽象类的使用；虚析构函数的使用。

【学习方法指导】

本章的内容较难理解。学习本章以前，应当把第十章中有关面向对象的基本概念搞清楚。要把握派生类和基类的相同点与不同点，基类的哪些成员可以继承，又有哪些不能继承，如何在派生类中覆盖基类的成员函数。要理解虚函数的概念和用法，它是理解和应用多态的关键。

【知识点】

基类；派生类；继承；多重继承；访问权限；函数的隐藏；函数的覆盖；虚函数的定义；虚函数的使用；多态的实现；抽象类；虚析构函数

上一章，我们学习了类，类是进行面向对象程序设计的基础。它能够定义数据和对数据的操作，并通过不同的访问权限，将类的接口和内部的实现分开，支持信息的封装和隐藏。面向对象程序设计的其它重要特征还包括：继承和多态。支持程序代码的复用是面向对象程序设计的主要目标之一，而支持程序代码复用的最重要的方法之一就是继承。继承能够从一个类派生出另一个类，前者称之为基类或父类，后者称之为派生类或子类。派生类能够继承基类的功能，也能改变或增加它的功能。

在面向对象的程序设计中，多态的功能强大，但也较难掌握。它允许指向基类对象的指针指向派生类的对象。基类和派生类中可有名字和参数完全相同的函数，但他们的功能并不相同。当程序运行时，如果用基类指针调用成员函数，能够根据该指针所指向的对象的类型自行确定是调用基类的成员函数还是调用派生类的成员函数。如果没有多态的功能，我们就不得不用条件语句，确定是调用哪一个类的成员函数。本章中，我们将介绍继承和多态的用法。

代码复用是 C++最重要的性能之一，它是通过类继承机制来实现的。通过类继承，我们可以复用基类的代码，并可以在继承类中增加新代码或者覆盖基类的成员函数，为基类成员函数赋予新的意义，实现最大限度的代码复用。

11.1.1 继承的定义及工作方式

我们可以用一个简单的例子解释什么叫继承。假定我们要处理 2 维空间中的点，定义了一个称之为 twoD 的 2 维空间点类：

程序段 11-1

```
class twoD
{
protected:
   double x, y;// x 和 y 坐标
public:
   twoD(double i, double j):x(i), y(j){}
   //内联构造函数
   //内联成员函数
   void setX(double NewX){x = NewX;}
   void setY(double NewY){y = NewY;}
   double getX() const {return x;}
   double getY() const {return y;}
};
```

假定后来又要处理 3 维空间点的情形，一个直接的方法是再定义一个 3 维空间点类 threeD：

程序段 11-2

```
class threeD
{
protected:
   double x, y, z;// x 、y 和 z 坐标
public:
   twoD(double i, double j, double k):x(i), y(j), z(k){}
   //内联构造函数
   //内联成员函数
   void setX(double NewX){x = NewX;}
   void setY(double NewY){y = NewY;}
   void setZ(double NewZ){z = NewZ;}
   double getX() const {return x;}
   double getY() const {return y;}
   double getZ() const {return z;}
};
```

threeD 类中，标记为兰色的地方是 threeD 类比 twoD 类多出的部分。实际上 threeD 类仅比 twoD 类多一个成员变量 z 及两个成员函数 setZ()和 getZ()。也就是说，threeD 类增加了一点新的代码到 twoD 类的定义之中。

在 threeD 类中编写了一部分与 twoD 类中重复的代码，如果使用继承，则可以简化 threeD 类的代码。继承的一般形式如下：

```
class 派生类:访问权限 基类
{
   …
```

```
}
```

访问权限是访问控制说明符，它可以是 public、private 或 protected。如果使用继承，我们可以将 threeD 类的定义改写为：

```
// 使用继承定义 threeD 类
class threeD:public twoD
{
private:
   double z;
public:
   // 内联的构造函数
   // 基类的构造函数不被继承
   // threeD 类的构造函数复用了 twoD 类的构造函数，并通过
   //成员初始化表将值传送到 twoD 的构造函数
   threeD(double i, double j, double k):twoD(i,j){z = k;}
   void setZ(double NewZ){z = NewZ;}
   double getZ() {return z;}
};
```

上例中，twoD 称为基类，threeD 称为派生类。应该注意到：派生类 threeD 中，setX()、setY()、getX(、getY()函数没有再定义，因为这些函数是可从基类 twoD 继承，就好象在 threeD 类中定义了这些函数一样。

注意：twoD 的构造函数用 threeD 的构造函数的初始化表中，说明基类的数据成员先初始化。基类的构造函数和析构函数不能被派生类继承。每一个类都有自己的构造函数和析构函数，如果用户没有显式定义，则编译器会隐式定义缺省的构造函数和析构函数。

下面是一个可以编译和运行的完整的程序：

例 11-1

```
// 文件名：Points.cpp
// 功能：演示继承的思想
//
#include <iostream.h>
class twoD
{
protected:
   double x, y;
public:
   twoD(double i, double j):x(i), y(j){}

   void setX(double NewX){x = NewX;}
   void setY(double NewY){y = NewY;}
   double getX() const {return x;}
   double getY() const {return y;}
```

```
};

class threeD:public twoD
{
private:
   double z;
public:
   threeD(double i, double j, double k):twoD(i,j){z = k;}
   void setZ(double NewZ){z = NewZ;}
   double getZ() {return z;}
};
void main()
{
   twoD Obj2(1, 1);
   threeD Obj3(1, 2, 3);
   cout << "Coordinates for 3d object is: " <<endl;
   cout << Obj3.getX() << ", " << Obj3.getY() <<", " <<
   Obj3.getZ()<< endl;
}
```

运行结果如下：

Coordinates for 3d object is:

1, 2, 3

派生类是与基类有一定联系的，基类是描述一个事物的一般特征，而派生类有比基类更丰富的属性和行为。比如正文中的例子，基类是一个二维点的模型，派生类从一个二维点模型中派生，添加了第三维向量和相应函数，生成了一个三维点模型。继承的一般形式如下：

```
class  派生类:访问权限  基类
{
   …
}
```

如果需要，派生类可以从多个基类继承，也就是多重继承，这将在后面章节中讲解。通过继承，派生类自动得到了除基类私有成员以外的其它所有数据成员和成员函数，在派生类中可以直接访问，从而实现了代码的复用。

从概念上讲，基类和派生类的关系与类和对象的关系有着根本的不同。比如说，我们定义了猫科动物类：

```
class felid
{
   //类中定义了猫科动物的基本特征
};
```

现在由于某种需要，我们想对猫这个动物进行定义。猫是猫科动物的一种，很自然，我们就想到从猫科动物这个类里面派生：

```
class cat: public felid
{
   char name[20] //名字
   char color[80] //毛色
   cat();
   cat(char* name, char* color);
}
```

在类 cat 中，示意性地定义了两个变量。我们知道，对象是类的一个实例，而在这里，cat 还是一个类，一个更加具体的类，不是 felid 类的对象，这一点要分辨清楚。我们可以定义猫的一个对象（实例）：

```
cat mycat("Jack","bronze");
```

mycat 才是一只实实在在的猫对象。

派生类对象生成时，要调用构造函数进行初始化。编译器的调用过程是先调用基类的构造函数，对派生类中的基类数据进行初始化，然后再调用派生类自己的构造函数，对派生类的数据进行初始化工作。当然，在派生类中也可以更改基类的数据，只要它有访问权限。

基类数据的初始化要通过基类的构造函数，而且，它要在派生类数据之前初始化，所以基类构造函数在派生类构造函数的初始化表中调用：

派生类名 (参数表 1)：基类名（参数表 2）

其中"参数表 1"是派生类构造函数的参数，"参数表 2"是基类构造函数的参数。通常情况下，参数表 2 中的参数是参数表 1 的一部分。也就是说，用户应该提供给派生类所有需要的参数，包括派生类和基类的。事实上也是这样，派生类继承了基类的成员变量，就相当于是自己的一部分，当然有责任对基类的变量进行初始化，只不过对于基类成员的初始化要借助于基类的构造函数而已。如果派生类构造函数没有显式调用基类的构造函数，编译器也会先调用基类的缺省参数的构造函数，对基类数据进行初始化。如果派生类自己也没有显式定义构造函数，那么编译器会为派生类定义一个缺省的构造函数，在生成派生类对象时，仍然先调用基类的构造函数。所以，派生类没有定义构造函数的话，必须保证基类有缺省参数的构造函数。

下面的情况是错误的：

程序段 11-3

```
class A
{
public:
   A(int n):a(n){}
   int a;
};
class B
{
public:
   int b;
};
```

A 是 B 的基类，B 是 A 的派生类，如果一个类定义了构造函数，那么编译器就不再为它生成缺省构造函数，这样 A 就不存在缺省构造函数。当程序中定义了一个类 B 的对象：

B b;

编译器会报告"no appropriate default constructor available"这样的错误，虽然编译器会为类 B 生成缺省的构造函数。在这种情况下，必须编写类 B 的构造函数，并在初始化表中调用基类的构造函数，进行初始化，例如：

B(int i,int j): A(i) {b=j;}

如果类中有另一个类的对象作为它的数据成员，该对象称为内嵌的对象。内嵌的对象应该先构造，所以，对它的初始化也应在初始化表中。例如：

程序段 11-4

```
class B
{
public:
   B(int I, int j):a(i),b(j){}
   A a;
   int b;
}
```

调用内嵌对象的构造函数与调用基类构造函数不同的是：调用基类构造函数用的是基类构造函数名，而这里使用的是内嵌的对象名。

上面我们讨论了派生类的构造函数的调用，也有必要讲解一下析构函数的调用。析构函数在对象被销毁时调用，对于派生类对象来说，基类析构函数和派生类构造函数也要分别调用，不过我们不再需要进行显式的析构函数调用，编译器能够为我们做好这件事。因为对于任何类型，只有一个析构函数，并且它并不带任何参数。需要注意的是：析构函数次序与构造函数调用次序正好相反。

下面我们举个例子来说明构造函数和析构函数的调用顺序。

例 11-2

```
#include "iostream.h"
#define CLASS(ID) class ID{\
public:\
ID(int) {cout<<#ID<<" created"<<endl;}\
~ID() {cout<<#ID<<" destroyed"<<endl;}\
};
CLASS(base);
CLASS(member1);
CLASS(member2);
CLASS(member3);
CLASS(member4);

class derived1:public base
{
   member1 mem1;
   member2 mem2;
```

```
public:
   derived1(int i) :mem2(i), mem1(i) ,base(i){
      cout<<"derived1 created"<<endl;
   }
   ~derived1() {
      cout<<"derived1 destroyed"<<endl;
   }
};

class derived2 : public derived1
{
   member3 mem3;
   member4 mem4;
public:
   derived2(int i) :derived1(i), mem3(i), mem4(i) {
      cout<<"derived2 created"<<endl;
   }
   ~derived2() {
      cout<<"derived2 destroyed"<<endl;
   }
};

void main()
{
   derived2 d(1);
}
```

在程序中，我们用下面的语句定义类：

```
#define CLASS(ID) class ID{\
public:\
ID(int) {cout<<#ID<<" created"<<endl;}\
~ID() {cout<<#ID<<" destroyed"<<endl;}\
};
```

CLASS（base）；

实际上就是：

```
class base{
public:
base(int) {cout<<"base"<<" created"<<endl; }
~base() {cout<<"base"<<" destroyed"<<endl;}
};
```

程序运行结果为：

base created

member1 created

member2 created

derived1 created

member3 created

member4 created

derived2 created

derived2 destroyed

member4 destroyed

member3 destroyed

derived1 destroyed

member2 destroyed

member1 destroyed

base destroyed

除了构造函数和析构函数不能被继承外，operater =() 也不能被继承，因为它完成和构造函数类似的工作。

11.1.2 访问控制

一、类内访问说明符

我们定义 twoD 和 threeD 类的时候，变量成员和成员函数前面有访问说明符：public、private 或 proteced，它们控制变量成员和成员函数在类内和类外如何访问。所谓类内访问是指用类的成员函数进行访问，而类外访问是指用对象或指向对象的指针进行访问。

当一个类的成员定义为 public，就能够在类外访问，包括它的派生类。

当一个成员定义为 private，它仅能在类内访问，不能被它的派生类访问。

当一个成员定义为 proteced，它仅能在类内访问，但是能被它的派生类访问。

当一个成员没有指定访问说明符时，默认为 private。

二、继承访问说明符

在定义派生类时，访问说明符也能出现在基类的前面，它控制基类的变量成员和成员函数在派生类中的访问方法。当访问说明符为 public 时，称为公有继承。同样地，当问说明符为 protected 时，称为保护继承，而当问说明符为 private 时，称为私有继承。

公有继承时，基类的公有成员，变为派生类的公有成员，基类的保护成员，变为派生类的保护成员。

保护继承时，基类的公有和保护成员，均变为派生类的保护成员。

私有继承时，基类的公有和保护成员，均变为派生类的私有成员。

有关派生类的访问权限参见表 11-1。

表 11-1 派生类访问权限

访问权限	示例	注释
public	class B: public A{…}	A 中公有成员对 B 也是公有的，A 中保护成员对 B

		也是保护的，并可被 B 访问，而 A 中私有成员不能被 B 访问
private	class B: private A{…}	A 中公有成员和保护成员可被 B 访问，但它们是私有的，A 中私有成员不能被 B 访问
protected	class B: protected A{…}	A 中公有成员和保护成员可被 B 访问，但它们是保护的，A 中私有成员不能被 B 访问

虽然 C++语法上，能够定义公有继承、保护继承和私有继承。但是，我们通常使用公有继承，保护继承与私有继承的用处不大。

下面，我们再举一个公有继承的例子：

例 11-3

```
class A
{
private:
   int x;
protected:
   double w;
public:
   A(int v, double z): x(v), w(z) {}
   ~A() {}
   double f();
   double g();
};

class B : public A {
private:
   double u;
public:
   B(int x, double z, double y): A(x,z), u(y) {}
   ~B() {}
   double g(); //  重新定义 A 的成员函数 g
   double h(); //  增加一个新的接口函数 h
};
```

我们应该注意到：A 的构造函数被用在 B 的构造函数的初始化表中。B 的公有接口包括函数 f（从 A 继承）、函数 g（从 A 继承，但被重新定义）和函数 h（在 B 中新增加的）。B 的成员函数能够使用 A 的 public 成员（即函数 f 和 g）和保护成员（即数据成员 w），但是，不能访问 A 的数据成员 x，因为它是私有的。

例如，B 的成员函数 g 的中，调用 A 的成员函数 g 和数据成员 w：

```
double B::g()
{
```

```
    double res1 = A::g();
    // 调用 A 中的函数 g
    double res2 = h();
    // 调用 B 中的函数 h
    w = w+res1+res2+u+f();
    // 修改 w
    return res1;
}
```

在 B 的成员函数中调用 A 的成员函数时，要用域运算符::，这是因为 A 中的成员函数 g 与 B 中的成员函数 g 的原型完全相同，如果不加域运算符，虽不会有语法错误，但编译器便会调用 B 中 g 成员函数，而与编程意图不符。

一、类内访问说明符

二、继承访问说明符

一、类内访问说明符

例 11-4

```
#include "iostream.h"
class A
{
   int a;
   //没有指定访问说明符，默认为 private
public:
   A(int i,int j):a(i),b(j){}
   //类内访问，可以访问公有变量和私有变量
   int b;
   int geta();
};

void main()
{
   A a(1,2);
   cout <<a.a;
   cout <<a.b; //错误，不能访问类的私有成员
   cout<<a.geta();
}
```

二、继承访问说明符

这里要明确几点：

1. 基类的私有成员不能被派生类访问。
2. 公有继承时，派生类原样继承基类的公有成员和保护成员，它们的属性在派生类中没有改变。
3. 保护继承时，基类的公有成员和保护成员，变成派生类的保护成员，它只能在派生类内被访问，但能够被该派生类的派生类访问。

4． 私有继承时，基类的公有成员和保护成员，变成派生类的私有成员。

在一般情况下，我们都是用 public 继承，这样使得基类的接口也是派生类的接口。当我们想隐藏这个类的基类部分的功能时，我们可以用 private 继承。当私有继承时，基类的所有公有和保护成员都变成派生类的私有成员，如果希望它们中任何一个成为 public，只要用派生类的 public 选项声明它们的名字即可。

二、继承访问说明符

例 11-5

```
#include "iostream.h"
class A
{
public:
   void a() { cout<<"a"<<endl; }
   void b() { cout<<"b"<<endl; }
};
class B :private A
{
public:
   A::a;
};

void main()
{
   B b;
   b.a();
   b.b(); //错误，函数 b 是类 b 的私有成员
}
```

成员函数 a 和 b 是从类 A 私有继承到类 b 的，但在类 b 中我们将基类中的成员函数 a 声明为 public，这样 a 在类 b 中的类型就是 public 的，在类外可以被访问，而成员函数 b 是 private 的，如果在类外访问编译器会出现错误提示。

11.1.3 函数的隐藏与覆盖

我们已经知道了函数的重载是怎么回事，重载的函数名字相同，但它们的参数个数和类型不同。函数的隐藏和覆盖，与函数的重载不同，它们只是在继承的情况下才存在。

如果在派生类中定义了一个与基类同名的函数，也就是说为基类的成员函数提供了一个新的定义，有两种情况：

◇ 在派生类中的定义与在基类中的定义有完全相同的信号（signature）（即参数个数与类型均相同）和返回类型，对于普通成员函数，这便称之为重定义；而对于虚成员函数（在本章的后面介绍），则称之为覆盖。

◇ 在派生类中，改变了成员函数参数表与返回类型。此时会出现什么情况？

我们看看下面的实例。

例 11-6

```
// 在派生类中，隐藏重载的函数名
#include <iostream.h>
class Base
{
public:
   int f() const {
      cout << "Base::f()\n";
      return 1;
   }
   int f(char *) const { return 1; }
   void g() {}
};

class Derived1 : public Base
{
public:
   void g() const {}
};

class Derived2 : public Base
{
public:
   // 函数 f 被重定义
   int f() const {
      cout << "Derived2::f()\n";
      return 2;
   }
};

class Derived3 : public Base
{
public:
   // 改变函数 f 的返回类型
   void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base
{
public:
   // 改变函数 f 的参数表
```

```
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }
};

void main()
{
    char s[]="hello";
    Derived1 d1;
    int x = d1.f();
    d1.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(s); // 基类带 char *参数的成员函数 f 被隐藏
    Derived3 d3;
    //! x = d3.f(); // 基类带返回 int 类型的成员函数 f 被隐藏
    Derived4 d4;
    //! x = d4.f(); // 基类带成员函数 f 被隐藏
    x = d4.f(1);
}
```

在派生类 Derived1 中，重定义了基类 Base 的成员函数 g。运行该程序，我们可以看到：在派生类 Derived1 中，基类 Base 重载的函数 f 可以被使用。但是，在派生类 Derived2 中，重定义了基类 Base 两个重载的成员函数 f 中的一个，导致第二个重载的函数也不能用。在派生类 Derived3 中，改变了返回类型，基类 Base 的两个重载的成员函数 f 都不能用，派生类 Derived4 也表明：改变参数表也隐藏基类 Base 的两个重载的成员函数 f。

在派生类重定义了基类重载的成员函数，通常会自动隐藏基类与该函数同名的其它重载的成员函数。关于函数覆盖应该注意以下两点：

（1） 在派生类中覆盖了基类的成员函数后，基类的成员函数被隐藏。但是，基类的成员函数如果需要的话，仍然可以显式调用。例如，派生类 B 中成员函数 g 覆盖了基类成员函数 g，但基类成员函数 g 仍然可以通过域运算符显式调用：

```
double B::g()
{
    double res1 = A::g(); // 调用 A 中的函数 g
    …
}
```

（2） 在类外，基类的被覆盖的成员函数，也可以通过域运算符访问。例如：

```
Derived3 d3;
int x = d3. Base::f();
```

在派生类中重定义基类的成员函数，会隐藏基类的该成员函数，例如：

例 11-7

```
class A
{
public:
   int a(int i) {return i;}
   float a(float i) {return i;}
};

class B: public A
{
public:
   void a(char* str) {}
};

void main()
{
   B b;
   b.a(1); //error;
   b.a(1.0); //error
   b.a("hehe");
}
```

上面程序中，类 A 的成员函数 a 被重载了两次，但是由于类 B 中隐藏了基类 A 的成员函数 a，所以 B 的对象无法直接看到基类的成员函数 a，当然用户可以通过域运算符显式调用。

```
b.a::a(1);
b.a::a(1.0);
```

11.1.4 不合适的继承

继承和派生有一个基本的原则，就是基类对象能够使用的地方，也能用同样的方法使用派生类的对象。

下面是一个破坏了这个原则的实例，BoundedStack 时从 Stack 派生的，Stack 类支持基本的栈操作，即压入一个元素到栈中和从栈中弹出一个元素，如例 11-8。

在该类中，隐含 push 成员函数总能压入一个新的元素到栈中。有时，我们可能需要只能压入固定元素个数的栈。为了复用 Statck 类的实现，从 Statck 类派生了一个 BoundedStack 类，如例 11-9。

但是，这破坏了上面提到的基本的原则：基类对象能够使用的地方，派生类对象也能够被使用。例如，假定 BoundedStack 的 capacity 成员设置为 5，即栈中最多只能有 5 个元素，则压栈次数超过 5，则会出错。

之所以从 Statck 派生 BoundedStack，只是为了复用 Statck 类的代码，其实，我们也可以用组合的方法复用代码，所谓组合是指用一个类的对象作为另一个类的数据成员，如例 11-9。

现在我们对这个原则可能还理解不深，在学习了多态的概念后，对这个原则会有更深的理解。

例 11-8

```
class Stack
{
private:
   ...
public:
   // 未给出构造函数
   int size(); // 返回当前栈元素的个数
   int pop();
   void push(int item);
};
```

例 11-9

```
class BoundedStack: public Stack
{
private:
   int capacity;
public:
   // 许多成员函数未给出
   void push(int item)
   {
      if (size()< capacity) Stack::push(item);
      else <error!!>
   }
};
```

例 11-10

```
class BoundedStack
{
private:
   Stack _myStack;
   int capacity;
public:
   // 省略了许多成员函数
   void push(int item)
   {
      if (_myStack.size()<capacity) _myStack.push(item);
      else <error!!!>
   }
}
```

继承和派生有一个基本的原则，就是基类对象能够使用的地方，也能用同样的方法使用派生类的对象。

当某一个类 A 希望拥有另一个类 B 的性能，但又不希望类 B 的接口成为类 A 的接口，或者类 B 是类 A 的一个组成部分，就使用组合，而不要使用继承。例如，一台电脑可以由一个主机，一个显示器和一个键盘组成。用类描述可以是：

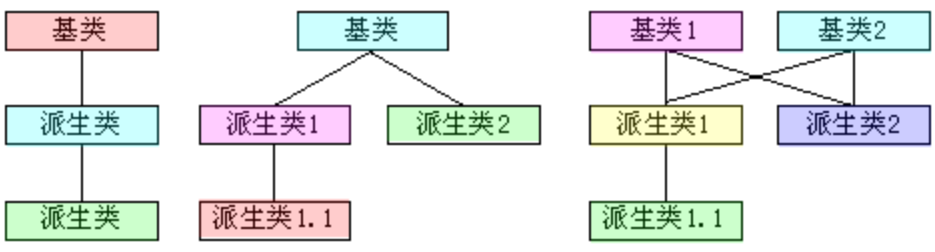
程序段 11-5

```
class mainframe
{
   ……
};

class monitor
{
   ……
};

class keyboard
{
   ……
};

class computer
{
public:
   mainframe mf;
   monitor con;
   keyboard kb;
   ……
};
```

11.1.5 多继承

我们前面介绍的内容中，一个派生类继承一个基类，我们称之为单继承。C++也支持多继承，即一个派生类继承多个基类，参见图 11-1

图 11-1 基类和派生类



多继承与单继承很类似。

例 11-11

```
class X
{
public:
   X(int n);
   ~X();
   //…
};
class Y
{
public:
   Y(double d);
   ~Y();
   //…
};
class Z : public X, public Y {
public:
   Z(int n, double d);
   ~Z();
   //…
};
```

派生类 Z 继承了两个基类 X 和 Y 所有的公有和保护成员,由于两个基类有带参的构造函数，派生类构造函数应该在成员初始化表中调用它们：

```
Z ::Z(int n, double d):X(n), Y(d)
{
   …
}
```

基类构造函数的调用顺序与在定义派生类时指定的顺序相同，与它们在成员初始化表中出现的顺序无关。本例中，X 的构造函数在 Y 的构造函数之前调用，即使我们把 Z 的构造函数改成下面的形式，也仍然如此：

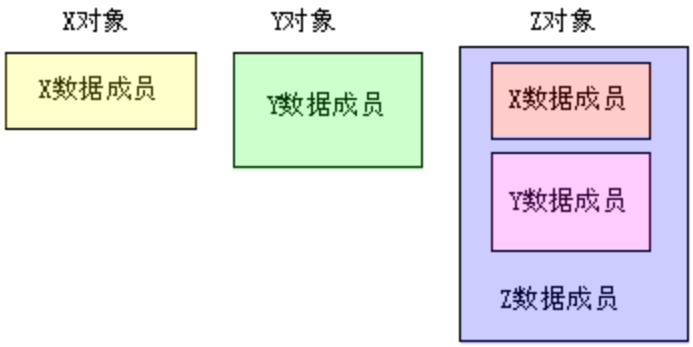
```
Z ::Z(int n, double d): Y(d), X(n)
{
   …
}
```

析构函数的调用顺序与构造函数的调用顺序相反，类 X、Y、Z 析构函数的调用顺序为：~Z、~Y、~X。

派生类对象包含每一个基类对象的成员，图 11-2 表示了 Z 对象与它的基类对象之间的关系：

在多继承时，C++并没有限制基类的个数，但不能有相同的基类，例如 11-12：

图 11-2 基类与派生类对象



例 11-12

```
class Z : public X, public Y, public X //  非法: X 出现两次
{
   //...
};
```

多继承类成员的引用比单继承复杂。例如 11-13，假定 Z 的基类 X 和 Y 均有成员函数 H。我们也可以在 Z 类中定义 H 成员，并调用基类的 H 成员，例如 11-14。

例 11-13

```
class X
{
public:
   //…
   void H (int part);
};
class Y
{
public:
   //…
   void H (int part);
};
```

派生类 Z 将继承这些成员函数，使得下面的调用发生歧义：

Z zObj;

ZObj. H (0);

编译器并不知道是要调用 X::H，还是 Y::H，我们可以通过显式调用来解决这个问题：

ZObj. Y::H (0);

例 11-14

```
class Z: public X, public Y
```

```
{
public:
   //…
   void H (int part);
};
void Z::H (int part)
{
   X::H (part);
   Y::H (part);
}
```

当继承基类时，派生类对象就包含了每一个基类对象的成员。假定以类 X 和类 Y 为基类派生出类 Z，类 Z 就会同时包含类 X 的和类 Y 的数据成员，如图 11-2 表示。

如果类 X 和类 Y 都是从相同的基类 A 派生的，那么从类层次上看，就成了一个菱形的结构，如图 11-3 表示。

图 11-2 基类与派生类对象

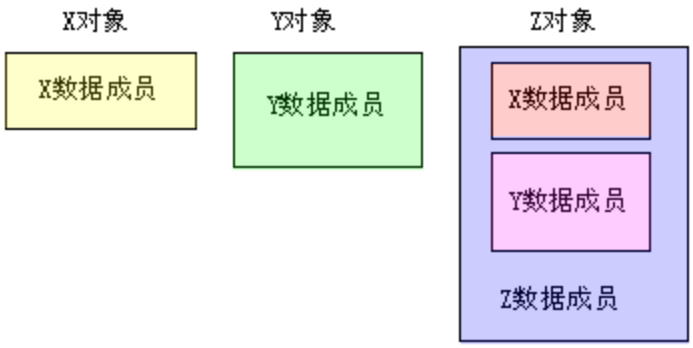
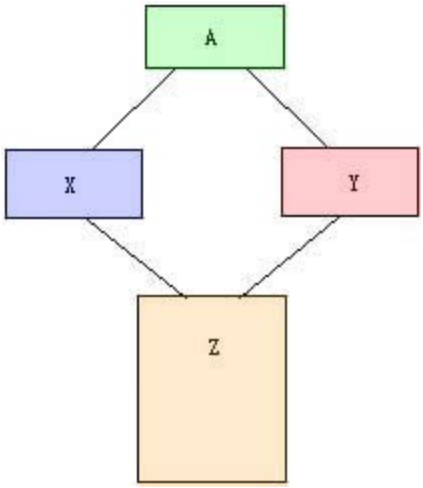


图 11-3



没有菱形的情况，多继承并没有什么麻烦。一旦出现菱形的情况，事情就变得复杂起来。首先，对于 A 类的数据成员来说，它在 Z 中是重叠的，即在 Z 中它有两个副本。这不仅增加了存储空间，更严重的是产生了二义性。

类 X 和类 Y 都实现了一个 func1()函数，当 Z 从 X 和 Y 继承时，就会导致一个冲突：当我们使用基类指针调用虚函数 func1()时，编译器就不知道它调用的是类 X 还是类 Y 的成员，这样编译器就会给出一个错误信息："Z::func1' is ambiguous"。正确的作法是必须在类 Z 中重定义 func1()函数，以消除二义性。

如果继承出现菱形的情况，还有很多地方会出现类似的问题。所以我们建议尽量不使用多重继承。

通过下面的例子我们可以看到：

例 11-15

```
#include
"iostream.h"
class A
{
public:
   virtual
void
func1(){}
};

class
X:public A
{
public:
   virtual
void
func1(){}
};
```

```
class
Y:public A
{
public:
   virtual
void
func1(){}
};

class
Z:public
X,public Y
{
   …
};

void main()
{
   A*
z=new Z();

z->func1();
//error
}
```

11.1.6 基类和派生类之间的转换

假定我们定义了两个对象，一个是基类对象，另一个是派生类对象：

twoD two(15.0, 15.0);

threeD three(20.0, 20.0, 30.0);

派生类对象 three 同时也是基类对象 two，派生类对象允许给基类对象赋值：

two = three;

对象 two 比对象 three 少一个数据成员 z，这个赋值的效果是对象 two 收到对象 three 的 x 和 y 坐标值。

反过来，用基类对象对派生类对象赋值是不行的。例如：

three = two;

由于对象 two 没有 z 数据成员，使得对象 three 的数据成员 z 的值是不定的。所以，C++编译器不允许基类对象对派生类对象赋值。

派生类对象可以给基类对象赋值，派生类对象的地址也可以赋给指向基类对象的指针，例如：

threeD three(20.0, 20.0, 30.0);

twoD *pTwo;

下面的赋值是合法的：

pTwo = &three;

由于指针 pTwo 是指向 twoD 对象，在这个赋值过程中，派生类类型被隐式转换成 twoD。所以，pTwo 只能调用 twoD 的成员函数，需要注意的是：我们不能把一个指向派生类对象的指针指向基类的对象。

引用与指针类似。例如，如果一个函数 A 定义有 twoD 引用参数，用函数 B 调该函数 A 时，实参可以是从 twoD 派生的类的对象。在函数 A 内，可以访问 twoD 类的成员。

下面我们通过一个例子说明基类指针如何访问对象的成员函数。假定我们在 twoD 类和 threeD 类分别增加一个成员函数 getSum：

例 11-16

```
class twoD
{
  …
  double getSum() const {return x+y;}
};

class threeD:public twoD
{
  …
  double getSum() const {return x+y+z;}
};
```

再采用如下方法调用 getSum 函数：

```
threeD three(20.0, 20.0, 30.0);
twoD *pTwo = &three;
double d = pTwo-> getSum();
```

d 的值为为 40.0，而不是 70.0。虽然 twoD 类指针指向 threeD 类对象 three，但是 pTwo -> getSum() 调用的是 twoD 类的成员函数，而不是 threeD 类的成员函数。可见，通过指针调用成员函数时，到底是调用基类的成员函数，还是调用派生类的成员函数，是由指针的类型决定的，而不是由对象决定的。

我们也可以对指针的类型进行强制转换，例如：

```
threeD three(20.0, 20.0, 30.0);
twoD *pTwo = &three;
threeD *pThree=(threeD *)pTwo;
double d = pThree->getSum();
```

上段代码的倒数第二行将 twoD 指针 pTwo 强制转换后赋给 threeD 指针 pThree，若直接赋值会出现编译错误，因为 twoD 指针不能隐式转换成 threeD 指针。

如果指针 pTwo 确实指向 threeD 对象，pThree->getSum()正常调用 threeD 类的成员函数 getSum，否则，可能出现难以预料的结果。

继承表示一种关系，即"新类是老类的一个类型"，这意味着在基类的所有函数在派生类中也是可用的，twoD 里面的函数，在 threeD 里面也有。ThreeD 对象不仅仅是 threeD 类型，同时也是 twoD 类型。例如：

```
threeD three(20.0, 20.0, 30.0);
twoD *pTwo;
pTwo = &three;
```

这种将派生类对象、指针或引用转变为基类的对象、指针或引用的过程称为向上映射。向上映射不需要作显式的说明，或强制的转换。因为向上映射是安全的，它是从特殊的类型转换为一般的类型，出现的最坏情况是失去类成员，而不会增加类成员。

由于向上映射是将特殊的类型转换成了一般的类型，对象的类型信息就损失掉了。比如，上面的对象three，编译器只把指针 pTwo 当成一个 twoD 类型指针处理，也就是说，不知道它现在指向的是一个 threeD 类型的对象。当调用 getSum（）成员函数时：

pTwo->getSum();

编译器会调用基类版本"twoD::getSum()"，而不是调用"threeD::getSum()"，如果想解决这个问题，需要利用下一节虚函数的知识。

多态性是面向对象设计语言的基本特征。仅仅是将数据和函数捆绑在一起，进行类的封装，使用一些简单的继承，还不能算是真正应用了面向对象的设计思想。多态性是面向对象的精髓，也是难点。在 C++中，多态性是通过虚函数来实现的。

11.2.1 为什么需要虚函数

为了说明虚函数的作用，我们先看一个程序实例：

例 11-17

```
#include <iostream.h>
class vehicle
{
   int wheels;
   float weight;
public:
   void message(void)  {cout << "Vehicle  message\n";}
};

class car : public vehicle
{
   int passenger_load;
public:
   void message(void)  {cout << "Car message\n";}
};

class truck : public vehicle
{
   int passenger_load;
   float payload;
public:
   int passengers(void) {return passenger_load;}
};

class boat : public vehicle
```

```
{
    int passenger_load;
public:
    int passengers(void) {return passenger_load;}
    void message(void)  {cout << "Boat message\n";}
};

void main()
{
    vehicle  *unicycle;
    car  *sedan;
    truck semi;
    boat sailboat;

    unicycle = new vehicle;
    unicycle-> message(); //输出 Vehicle  message
    delete unicycle;

    unicycle = new car;
    unicycle -> message(); //输出 Vehicle  message

    sedan = (car  *) unicycle;
    sedan -> message(); //输出 Car message
    delete sedan;

    semi.message();  //输出 Vehicle  message
    sailboat.message(); //输出 Boat  message
}
```

该程序的运行结果，我们已经标注在程序之中。从 11.1.4 节的知识，我们可以也很容易分析出它的运行结果。因为指针的类型决定调用那一个成员函数，所以，一个 vehicle*调用 vehicle 成员函数，即使它指向派生类的对象。同样，一个 car *也调用 car 的成员函数。我们把这称为早期联编或静态联编，因为指针要调用那一个函数是在编译时就确定的。

那么，当 vehicle*指向派生类对象时，我们能不能通过该指针来调用派生类的成员函数呢？在 C++中，我们是可以作到的，这要用到 C++的多态特性。也就是说，基类指针是调用基类的成员函数，还是调用派生类的成员函数，不是由指针的类型决定的，而是由指针指向的对象的类型决定的。

多态也称为动态联编或迟后联编，因为到底调用哪一个函数，在编译时不能确定，而要推迟到运行时确定。也就是说，要等到程序运行时，确定了指针所指向的对象的类型时，才能够确定。在 C++中，动态联编是通过虚函数来实现的。

我们知道，函数调用是通过相应的函数名来实现的。对于源程序进行编译后，存放在内存中的可执行程序，函数实际上是一段机器代码，它是通过首地址进行标识和调用的。例如，假定定义一个函数：

```
void func()
```

```
{
  //…
};
```

我们可以用下面的语句调用这个函数：

```
func(); //调用 func 函数
```

这是在源程序中调用函数的方法，它是用函数名操作的。

下面我们看看在可执行程序中函数调用是怎么操作的，我们用汇编语言来说明，因为汇编语言和机器语言（计算机可以直接执行的语言）是一一对应的。

在可执行程序中，函数调用使用下面的方法：

```
call [xxxxx]
```

xxxxx 代表存放函数代码内存空间的首地址。

call 是汇编语句中的一条指令，意思是调用一个函数。实际操作过程是：保存当前地址、保护现场，跳转到 xxxxx 地址执行。正是基于这个原因，在 C/C++中的函数名是一个指针，该指针指向该函数段代码在内存中的首地址。

如何将源程序中的函数调用和函数体（也就是在内存中该函数的机器代码）联系起来呢？这件工作是由编译器和连接程序来完成的。

在 C/C++语言中，函数调用在程序运行之前就已经和函数体（函数的首地址）联系起来。编译器把函数体翻译成机器代码，并记录了函数的首地址。在对函数调用的源程序段进行编译的时候，编译器知道这个函数名的首地址在那里（它可以从生成的标识符表中查到这个函数名对应的首地址），然后将这个首地址替换函数名，一并翻译成机器码。这种编译方法称为早期或静态联编。

那么，当 vehicle*指向派生类对象时，我们能不能通过该指针来调用派生类的成员函数呢？从这种编译方法来看，是不可能的。因为编译器只会寻找 vehicle*的成员函数。如何实现这个功能：当用基类指针调用成员函数时，是调用基类的成员函数，还是调用派生类的成员函数，不由指针的类型决定，而由指针指向的对象的类型决定呢？也就是说，如果基类指针指向基类对象，就调用基类的成员函数，如果基类指针指向派生类对象，就调用派生类的成员函数。这就要用到另外一种方法，称为动态联编或迟后联编。到底调用哪一个函数，在编译时不能确定，而要推迟到运行时确定。在 C++中，动态联编是通过虚函数来实现的。下面我们先介绍虚函数，然后讨论动态联编实现的原理。

针对教材中举出的例子，我们看一下编译器的编译过程：

例 11-18

```
#include <iostream.h>
//如果定义了 vehicle 类的对象，编译器为该对象分配 8 个字节的存储空间
class vehicle
{
  int wheels; //占用 4 个字节
  float weight; //占用 4 个字节
public:
  void message(void) {cout << "Vehicle  message\n";}
  //该函数不在类中分配空间，编译器另外分配空间给该函数，并把函数首
  //地址记录到一个标识符表中
};
//如果定义了 car 类的对象，编译器为该对象分配 12 个字节的存储空间
```

```
class car : public vehicle //基类占用 8 个字节
{
   int passenger_load; //占用 4 个字节
public:
   void message(void) {cout << "Car message\n";}
   //该函数不在类中分配空间，编译器另外分配空间给该函数，并把函数首
   //地址记录到标识符表中
};
//如果定义了 truck 类的对象，编译器为该对象分配 16 个字节的存储空间
class truck : public vehicle //基类占用 8 个字节
{
   int passenger_load; //占用 4 个字节
   float payload;//占用 4 个字节
public:
   int passengers(void) {return passenger_load;}
   //该函数不在类中分配空间，编译器另外分配空间给该函数，并把函数首
   //地址记录到标识符表中
};
//如果定义了 boat 类的对象，编译器为该对象分配 12 个字节的存储空间
class boat : public vehicle //基类占用 8 个字节
{
   int passenger_load; //占用 4 个字节
public:
   int passengers(void) {return passenger_load;}
   void message(void) {cout << "Boat message\n";}
   //这两个函数都不在类中分配空间，编译器另外分配空间给该函数，并把
   //函数首地址记录到标识符表中
};
void main()
{
   vehicle *unicycle;
   car *sedan;
   truck semi;
   boat sailboat;
   unicycle = new vehicle;
   unicycle-> message(); //输出 Vehicle message
   /*
   当编译器编译到上面语句的时候，会参照该对象指针所属的类名 vehicle 和函数名
message 去标识符表中查找该成员函数的首地址，查找到 vehicle 类的成员函数 message 的
首地址，并使用 call xxxx 语句替换该函数调用语句。*/
   delete unicycle;
   unicycle = new car;
```

```
unicycle -> message(); //输出 Vehicle message
/*
  当编译器编译到上面这句的时候，会参照该对象指针所属的类名 vehicle 和函数名 message 去标识符表中查找该成员函数的首地址，它查到的也是 vehicle 类的成员函数 message 的首地址，然后使用 call xxxxx 语句替换该函数调用语句。*/
sedan = (car *) unicycle;
sedan -> message(); //输出 Car message
/*
  当编译器编译到上面这句的时候，会参照该对象指针所属的类名 car 和函数名 message 去标识符表中查找该成员函数的首地址，它查到的是 car 类的成员函数 message 的首地址，然后使用 call xxxxx 语句替换该函数调用语句。
*/
delete sedan;
semi.message(); //输出 Vehicle message
/*
  当编译器编译到上面这句的时候，会参照该对象指针所属的类名 car 和函数名 message 去标识符表中查找该成员函数的首地址，在基类 vehicle 中它查到了成员函数 message 的首地址，然后使用 call xxxxx 语句替换该函数调用语句。
*/
sailboat.message(); //输出 Boat message
/*
  当编译器编译到上面这句的时候，会参照该对象指针所属的类名 boat 和函数名 message 去标识符表中查找该成员函数的首地址，它查到的是 boat 类的成员函数 message 的首地址，然后使用 call xxxxx 语句替换该函数调用语句。
*/
}
```

11.2.2 虚函数的定义与使用

虚函数的定义很简单，只要在成员函数原型前加一个关键字 virtual 即可。如果一个基类的成员函数定义为虚函数，那么，它在所有派生类中也保持为虚函数，即使在派生类中省略了 virtual 关键字。

还是上面的例子,我们把基类的成员函数定义为虚函数,即:

例 11-19

```
class vehicle
{
  …
  virtual void message(void) {cout << "Vehicle message\n";}
};
```

并将 main 函数修改为:

```
void main()
{
   vehicle *unicycle;

   unicycle = new vehicle;
   unicycle->message();
   delete unicycle;

   unicycle = new car;
   unicycle->message();
   delete unicycle;

   unicycle = new truck;
   unicycle->message();
   delete unicycle;

   unicycle = new boat;
   unicycle->message();
   delete unicycle;
}
```

该程序的运行结果为:

```
Vehicle message
Car message
Vehicle message
Boat message
```

下面我们对这个程序作一个简单的分析。基类 vehicle 的成员函数 message 被定义为虚函数，虽然其派生类中的 message 成员函数定义时，没有 virtual 关键字，但都是虚函数。从上面的结果我们确实看到：如果派生类中有与基类对应的方法，并且基类指针指向派生类的对象，那么基类指针调用的方法是派生类的方法。

我们应该注意到：main 函数中有四个相同的语句"unicycle->message();"，但是它们的结果并不相同。哪一个类的 message 成员函数被调用，不是在编译时确定的，而是根据运行时 unicycle 指针指向的对象的类型确定的。由于类 truck 没有覆盖基类的 message 成员函数，系统调用基类的 message 成员函数。

需要注意的是：要达到动态联编的效果，基类和派生类的对应函数不仅名字相同，而且返回类型、参数个数和类型也必须相同。

程序段 11-6

```
class vehicle
{
   …
   virtual void message(void) {cout << "Vehicle message\n";}
};
```

```
class car : public vehicle
{
   int passenger_load;
public:
   （virtual）void message(void) {cout << "Car message\n";}
};
```

在这个例子里，我们在成员函数 message 原型前面加上 virtual，使这个函数变为虚函数。在派生类中与基类的虚函数有相同的信号（signature）（即参数个数与类型均相同）的函数也都变为虚函数，不管它的前面有没有加上 virtual。之所以这么说是因为下面的情况中，并不是所有覆盖的函数都变成虚函数。

程序段 11-7

```
class car : public vehicle
{
   int passenger_load;
public:
   （virtual）void message(void) {cout << "Car message\n";}
   void message(int  n) {}
   //这个函数会覆盖基类的 message 函数，但是它不是虚函数
};
```

还有一个有意思的情况：

程序段 11-8

```
class car : public vehicle
{
   int passenger_load;
private:
   （virtual）void message(void) {cout << "Car message\n";}
};
```

我们把派生类的虚函数定义为私有，就会出现下面的情况：

```
car mycar;
vehicle*  v=&mycar;
mycar.message()  //error
v->message() //right
```

因为 message()是 car 类的私有成员，所以在类外应该访问不到它，正象"mycar.message()"会出现编译错误一样。但是，如果我们把它定义成虚函数，就可以通过基类指针访问到了。

对于"v->message();"，虽然 v 是 vehicle 类型指针，但是，它调用的是虚函数，所以实际上调用的是 car 类的 message()函数。

使用虚函数，我们可以获得良好的可扩展性，在一个设计比较好的面向对象程序中，大多数函数都是与基类的接口进行通信。因为使用基类接口时，调用基类接口的程序不需改变就可以适应新类。如果用户想添加新功能，他就可以从基类继承并添加相应的新功能。例如，上面 vehicle 类的例子中已经实现了 car、truck、boat 三种交通工具，然后我们定义一个"很复杂"的函数：

```
void getmessage(vehicle*  v)
{
   v->message();
}
```

若干天后，我们又想在这个程序中加入 bus 的功能，可以从基类 vehicle 派生出 bus 类：

```
class bus : public vehicle
{
   int passenger_load;
public:
   void message(void)  {cout << "bus message\n";}
};
```

对于上面的 getmessage()函数，由于它"很复杂"，我们不想改变它，那么这个函数可以适应我们添加的新功能吗？答案是肯定的。我们不需要改变函数的任何一部分，就可以适应这个新类。

```
car mycar;
getmessage(&car);  //这是原来的功能
bus mybus;
getmessage(&bus);  //这是新功能，同样的用法。
```

运行结果是：

```
car message
bus message
```

11.2.3 多态的实现

这一部分，我们简要地介绍一下在 C++中多态是怎样实现的。

多态的基本思想是：在编译时，C++编译器不知道调用哪一个函数，而要到运行时确定。这意味着应把函数的入口地址保存在某一个地方，以便于在调用前查询，而存储函数入口地址的地方也应能被相关的对象访问。例如，一个 Vehicle * unicycle 指针指向 car 对象，然后，unicycle->message()调用 car 的成员函数，这个函数的入口地址由 unicycle 指向的对象决定。

在 C++中，一般实现方法如下：包含虚函数的对象，增加了一个隐含的数据成员，且是它的第一个数据成员，该数据成员指向一个指针数组，而指针数组存储对象的虚函数地址，需要说明的是这个实现与具体的编译器有关。

某一个类的虚函数地址表被该类的所有对象共享，甚至有可能两个类共享同一个虚函数地址表。内存开销包括：

◇ 每一个对象增加了一个额外的数据成员。

◇ 每一个类有一个指针表，用于存储该类各虚函数的地址。

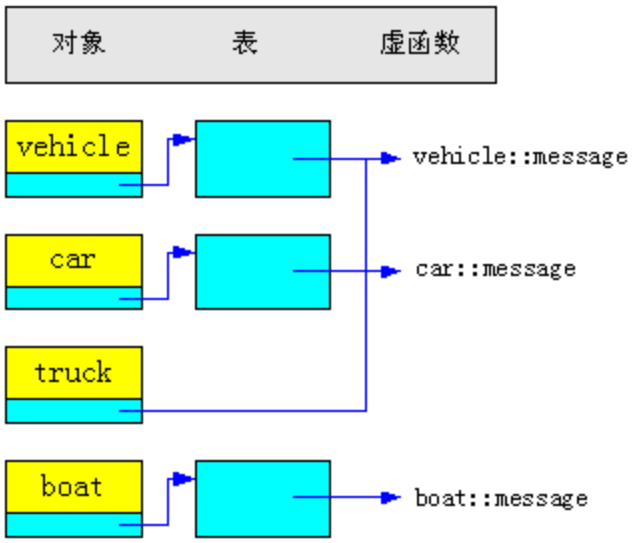
所以，unicycle->message()的调用过程是：首先检查 unicycle 指向的对象的隐含的数据成员，在我们前面所举的例子中，该数据成员指向的指针表只有一个元素，即 message 函数的入口地址，被调用的函数根据指针表确定。

有虚函数的的对象的内部组织，我们可以用 11-4 的示意图来说明：

正象我们在图 11-4 中看到的，有虚函数的所有对象均有一个隐含的指针数据成员，且指向存放虚函数

入口地址的指针表。类 Vehicle 对象与 truck 对象共用一个表，而 car 和 boat 有自己的 message 函数，所以，它们需要自己的虚函数指针表。

图 11-4 有虚函数的对象的内部组织



virtual 将一个成员函数说明为虚函数，对于编译器来讲，它的作用是告诉编译器，这个类含有虚函数，对于这个函数不使用静态联编，而是使用动态联编机制。编译器就会按照动态联编的方案进行一系列的工作。

对于每个包含虚函数的类，编译器都为其创建一个表（称之为 VTABLE 表）。在 VTABLE 表中放置的是每个类自己的虚函数地址，在每个包含虚函数的类中放置了一个指针（VPTR），指向 VTABLE 表。通过基类指针调用虚函数时，编译器会在函数调用的地方插入一段特定的代码。这段代码的作用就是得到 VPTR，找到 VTABLE，并在 VTABLE 表中找到相应的虚函数地址，然后进行调用。

下面这个例子可以让我们看到 VPTR 的用法：

例 11-20

```
#include "iostream.h"
//没有虚函数的类
class A
{
public:
   int a;
   A():a(0){}
};
//有一个虚函数的类
class B
{
```

```
public:
    int a;
    B():a(0){}
    virtual void func(){}
};
//有两个虚函数的类
class C
{
public:
    int a;
    C():a(0){}
    virtual void func(){}
    virtual void func2(){}
};

void main()
{
    cout<<"没有虚函数的类大小是"<<sizeof(A)<<endl;
    cout<<"有一个虚函数的类大小是"<<sizeof(B)<<endl;
    cout<<"有两个虚函数的类大小是"<<sizeof(C)<<endl;
    A a;
    B b;
    C c;
    int* tmp;
    tmp=(int*)&b;
    //打印对象 b 中两个整型单元中的值
    cout<<(*tmp)<<endl;
    cout<<(*(tmp+1))<<endl;
    //改变 b 中成员 a 的值
    b.a=1;
    //再次打印对象 b 两个整型单元中的值
    cout<<(*tmp)<<endl;
    cout<<(*(tmp+1))<<endl;
}
```

程序运行结果为：

```
没有虚函数的类大小是 4
有一个虚函数的类大小是 8
有两个虚函数的类大小是 8
4358196
0
4358196
```

1

main 函数分为两部分，从第一部分可以很清楚的看到没有虚函数的类的长度是 4，就是所有成员变量的长度，带一个或多个虚函数的类的长度是 8，是所有成员变量加上一个 VPTR 的长度，因为这只是指向 VTABLE 表的指针，所以带几个虚函数对这个指针是不起作用的，它们只影响 VTABLE 表的长度。

从第二部分可以看到，B 类对象 b 的大小是 8 个字节，即两个整型的大小，从我们改变它的成员变量 a 的值前后两个整型单元的数值，我们可以看到，第一个整型单元的值没有改变，第二个整型单元的值从 0 变成 1，这正是成员变量 a 的位置。第一个整型单元是什么呢？这就是 VPTR 所在的位置。第二部分告诉我们 VPTR 的位置是在这个类的开始，并且永远处在类的开始位置。将来定义了这个类的对象以后，该对象指针的位置就是这个 VPTR 的位置。

这里可能有个疑问，如果一个类里面没有成员变量，那怎么办？它的对象的长度岂不是要变成 0，一个 0 长度的数据的地址是什么？即使分配给它一个地址，当给下一个对象分配空间时，岂不是要与这个对象的地址相同。为了避免这个问题，编译器强制这个类的对象的长度非 0，如果这个类没有任何成员变量，也没有虚函数，即理论上它的对象的长度是 0，但是，编译器会向这个对象中插入一个"哑"成员。当类中仅有虚函数时，该对象中仅有一个 VPTR。如果将上面程序的所有成员变量全注释掉，再重新编译运行程序，我们就会看到这种情况。

程序运行结果是：

没有虚函数的类大小是 1

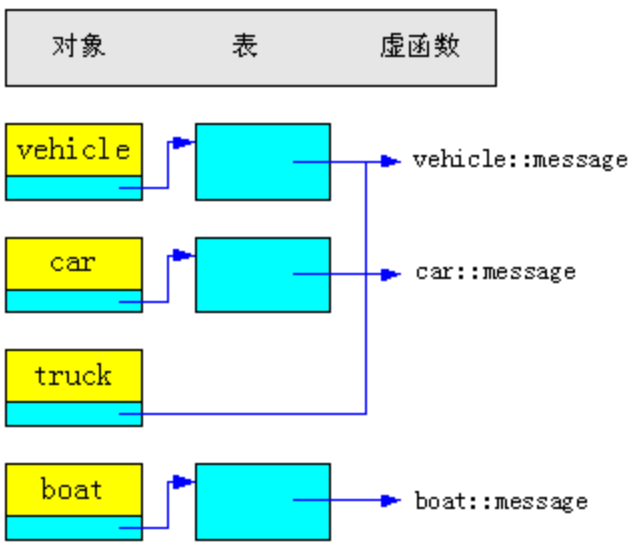
有一个虚函数的类大小是 4

有两个虚函数的类大小是 4

我们已经清楚 VPTR，下面我们来讨论 VTABLE 的构造。

每个包含虚函数的类都有自己的 VTABLE 表，VTABLE 表中存放各自类的虚函数地址，对于没有重定义的虚函数使用基类函数的地址。不管怎样，在每个类中总要有全体虚函数的地址，并且虚函数地址在各自表中的顺序必须一致，因为在函数查找时，只是根据 VPTR 加上一个偏移量来确定某个虚函数，如果 VTABLE 表的虚函数不全，或者顺序不对，就不可能查到正确的函数地址。

图 11-4 有虚函数的对象的内部组织



但是如果派生类中有自己的虚函数，那么 VTABLE 将是什么结构呢？我们举个例子来说明：

例 11-21

```
class A
{
public:
   int a;
   A():a(0){}
   virtual void func1(){}
   virtual void func2(){}
};

class B ： public A
{
public:
   int a;
   B():a(0){}
   virtual void func1(){}
   virtual void func2(){}
   virtual void func3(){}
};

class C ： public B
{
public:
   int a;
   C():a(0){}
   virtual void func1(){}
   virtual void func3(){}
};
```

这三个类的 VTABLE 表是下面的情况：

表 11-2

类 A	类 B	类 C
A::func1	B::func1	C::func1
A::func2	B::func2	B::func2
	B::func3	C::func3

从这个表中可以看到，派生类中和基类都有的虚函数，在各自的 VTABLE 表中位置必须一样，如果派生类中没有重定义该虚函数，则它就使用基类的虚函数地址。派生类自己定义虚函数依次添加在 VTABLE 表中。

当编译器对类的定义进行编译时，如果类中含有虚函数，它就采用动态联编机制，生成这样一个VTABLE 表，记录这个类的虚函数的地址。编译器在类的构造函数开头部分秘密添加了一段代码（如果用户没有显式定义构造函数，编译器会生成缺省构造函数），这段代码对用户是不可见的，它的作用是：初始化 VPTR，使它指向该类的 VTABLE。这样，当生成该类对象时，所做的第一件事就是将 VPTR 赋值，使它指向对应的 VTABLE 表。

现在有了 VTALBE 表，生成了该类对象，并已经设置了 VPTR，使它指向 VTABLE，那么当使用基类指针调用虚函数时，到底是怎么实现的呢？

对于上面的基类 B 和派生类 C，我们看下面的程序段：

```
B* pb=new C();
pb->func3();
```

pb 是 B 型指针，但指向的是 C 类对象，并调用虚函数 func3()。我们看一下这个函数调用在 VC6.0 编译器中编译出的代码（汇编语言的代码跟具体编译器有关，"；"后面部分是汇编语言的注释）：

```
mov edx, DWORD PTR _pb$[ebp]
；将 this 指针取出到 edx 中
mov eax, DWORD PTR [edx]
；将 VPTR 取出到 eax 中
mov esi, esp
；与本文无关，不用管它
mov ecx, DWORD PTR _pb$[ebp]
；将 this 指针作为参数传给函数
call DWORD PTR [eax+8]
；到 VTABLE 中查找，并进行调用
```

在这段代码中，寄存器 edx 存放的是对象的首地址，它对应该对象的 this 指针，因为每个成员函数调用都有个隐含参数，就是该对象的 this 指针。第一句是将 this 指针取出。因为 VPTR 保存在对象的首部，正是 this 指针指向的地址，所以第二句是将指向寄存器 edx 指向的位置的值，即 this 指针指向的双字，即 VPTR 取出，存在寄存器 eax 中。第三句与我们要讨论的没有关系，它是将堆栈指针保存起来，等该函数调用完返回后，进行检查，这是 VC 编译器的一种保护措施，我们不用管它。第四句就是将 this 指针作为参数传给这个函数。在汇编语言中，给一个函数传递参数可以有两种方法：通过堆栈或寄存器，一般比较小的函数调用或者经过优化后的程序会使用寄存器传参。第五句是去 VTABLE 表中查找虚函数位置，因为一个指针是四个字节，func3 在表中是第三个，所以地址是 eax+8，然后进行调用。这样，编译器就实现了动态联编。

上面我们已经看到：基类 Vehicle 包含有自己的、具体的 message 虚函数的实现。在 C++中，也可以在基类中仅定义虚函数的信号：函数的名字、返回类型和参数，而没有实现，但在派生类中必须有该虚函数实现。

仅定义了函数的信号，而没有函数实现的虚函数称之为纯虚函数。定义纯虚函数的方法是在虚函数参数表右边的括号后加一个"=0"的后缀，例如：

```
class vehicle
{
  …
  virtual void message(void) = 0;
};
```

上面这段代码中，我们便把 vehicle 的 message 成员函数定义为纯虚函数。含有纯虚函数的类，我们称之为抽象类，_____。所以，抽象类也称之为抽象的基类。

下面，我们再看一个抽象类的例子：

```
class CPolygon
{
  protected:
  int width, height;
public:
  void set_values (int a, int b)
    { width=a; height=b; }
  virtual int area (void) =0;
};
```

Cpolygon 类的成员函数 area 被定义为纯虚函数，它返回 int 变量，没有参数。因为 Cpolygon 类包含了纯虚函数 area，所以，它是一个抽象类，下面的定义：

```
CPolygon poly;
```

是错误的。我们虽然不能定义抽象类对象，但可以定义抽象类指针，例如：

```
CPolygon * ppoly1;
CPolygon * ppoly2
```

是完全正确的，这是因为基类的指针可以指向派生类的对象。

下面，我们举一个完整的例子：

例 11-22

```
#include <iostream.h>
class CPolygon {
  protected:
  int width, height;
public:
  void set_values (int a, int b)
    { width=a; height=b; }
  virtual int area (void) =0;
};

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

class CTriangle: public CPolygon
{
```

```
public:
  int area (void)
    { return (width * height / 2); }
};

int main ()
{
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

本程序计算本输出矩形和三角形的面积，运行结果为：

20

10

上面的代码中，我们用基类 CPolygon 指针指向派生类 CRectangle 和 CTriangle 对象，调用派生类的 area 成员函数，而在 main 函数中输出 area 函数的计算结果。我们也可以在 CPolygon 类中定义一个成员函数，用于打印计算结果，而不管是哪一个派生类的 area 函数的计算结果。CPolygon 类修改如下：

```
class CPolygon
{
  …
public:
  …;
  void printarea (void)
    { cout << this->area() <<endl; }
};
```

并把 main 函数作如下修改：

```
int main ()
{
  …
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

有的时候我们的基类只是起到接口的作用，没有必要生成该类的对象，该类本身也不需要对函数进行

实现，这样的基类称之为抽象类。在 C++中，有一种虚函数称之为纯虚函数，含有纯虚函数的类，就是抽象类。C++编译器不允许用抽象类创造对象，它只能被其它类继承。要定义抽象类，就必须定义纯虚函数，它实际上是起到一个接口的作用。

如果试图使用抽象类创造对象，编译器会给出错误信息。另外，由抽象类派生的类，必须对抽象类定义的纯虚函数进行实现。

当一个类声明了纯虚函数后，首先，编译器知道这个函数应使用动态联编，然后它会为这个类建立 VTABLE 表。由于这个函数是纯虚函数，没有实现，所以没有函数指针。编译器遇到这种情况会在 VTABLE 表中为它留下一个间隔，即一个函数指针大小的空间，不放任何东西。只要类中声明了一个纯虚函数，这个类的 VTABLE 表就是不完全的。编译器会禁止使用这个类创建对象。

虽然定义了纯虚函数的类是抽象类，不能用这个类创建对象，但是这不意味这这个类所有的函数都是纯虚函数，在该类里的其它函数就失去了意义。事实上，我们希望把公共的代码放在尽可能靠近基类的地方，这样不仅节省了空间，而且能使修改变得更加容易。

在基类中我们可能希望一段代码对于大部分或者所有的派生类都能使用，而不希望在每个类中重复这样的代码。在这种情况下，我们可以对纯虚函数进行实现。虽然纯虚函数有实现部分，但是，仍然不允许用抽象类创建对象。

下面是一个有纯虚函数实现例子：

例 11-23

```
#include <iostream.h>
class A
{
public:
   int a;
   A():a(0){}
   virtual void func1() =0 {
      cout<<"A::func1"<<endl;
   }
virtual void func2() =0;
};

void A::func2()
{
   cout<<"A::func2"<<endl;
}

class B:public A
{
public:
   int a;
   B():a(0){}
   virtual void func1(){
```

```
    A::func1();
    cout<<"B::func1" <<endl;
  }
  virtual void func2(){
    A::func2();
    cout<<"B::func2"<<endl;
  }
  virtual void func3(){};
};

void main()
{
  A *pa=new B();
  pa->func1();
  pa->func2();
}
```

程序运行结果是：

```
A::func1
B::func1
A::func2
B::func2
```

在 C++中，构造函数不能定义为虚函数，而析构函数可以定义为虚函数。

派生类的析构函数能够自动调用基类的析构函数，但用抽象类指针处理对象时，析构函数如何调用会有问题。例如 11-23，假定抽象类 Shape 表示所有能画出来的不同的对象，而 Picture 是 Shape 派生类，并有指向其它 Shape 派生类对象的指针作为它的数据成员：

例 11-24

```
class Shape // 抽象类
{
public:
  virtual void draw () = 0;
  ~Shape () {} // 空的析构函数
  ....
  // 没有数据成员
};
class Picture : public Shape // 组合对象
{
public:
  void add (Shape *); // 增加一个元素
  int count( ); // 计算元素个数
```

```
    ~Picture (); // 清除所有的元素
    ....
  private:
    Shape *elements[];
    int nNumber;
  };
  Picture::~Picture ()
  {
    int nTotal = count ();
    for (int i = 0; i < nTotal; ++i)
      delete elements[i];
  }
```

下面，创建了一个 Picture 对象，并通过抽象类指针操作。当该对象推迟释放时，我们遇到麻烦：

```
Shape * pShape; // 抽象类指针
pShape = new Picture ();
....
pShape->draw (); // 用 Shape 指针操作
// 现在，我们要销毁对象
delete pShape; // 只调用了 Shape::~Shape ()!
// 错误：Picture 部分保持不变
```

在 delete 操作中，Shape 的析构函数不会释放 Picture 的数据成员，导致内存泄露。为了抽象类指针能够调用正确的析构函数，应将析构函数定义为虚函数。这样，delete 能够根据运行时对象的类型，动态地绑定析构函数，确保正确的析构函数被调用。

```
class Shape
{
  ...
  virtual ~Shape () {}
};
...
class Picture : public Shape
{
  ...
  ~Picture (); // 自动成为虚函数
};
...
Shape * pShape = new Picture ();
...
delete pShape; // 正确: 调用 Picture::~Picture ()
```

虚析构函数也可以定义为纯虚函数，称为纯虚析构函数。不象其它纯虚函数，即使在派生类中，纯虚析构函数被再定义，它也不能被覆盖。

所以，纯虚析构函数是一个例外，如果某一个析构函数定义为纯虚函数，还必须有该纯虚析构函数的

实现，否则，会产生编译错误。例如：

```
class Interface
{
public:
   virtual void Open()=0;
   virtual ~Interface() = 0;
};
```

Interface 类定义了纯虚析构函数，在类外，我们我们必须有该该纯虚析构函数的实现：

```
Interface::~Interface()
{} //纯虚析构函数的实现总是空的
```

在 C++中，构造函数不能定义为虚函数，而析构函数可以定义为虚函数。因为构造函数的功能是构造并初始化这个对象，它的一项功能就是设置 VPTR，所以构造函数不能定义成虚函数，动态联编机制还没有建立起来，是不可能进行虚函数调用的。

构造函数不能是虚函数，但是析构函数能够是虚函数，而且我们常常将析构函数定义为虚函数。

我们再举个例子说明：

例 11-25

```
#include "iostream.h"
class A
{
public:
   int *a;
   A() { a=new(int); }
   virtual void func1(){}
   virtual void func2(){}
   ~A() {delete a; cout<<"delete a"<<endl; }
};

class B:public A
{
public:
   int *b;
   B() {b=new(int); }
   virtual void func1(){}
   virtual void func2(){}
   virtual void func3(){}
   ~B() { delete b; cout<<"delete b"<<endl; }
};

void main()
{
```

```
    A *pb=new B();
    delete pb;
}
```

程序运行结果：

delete a

在 main 函数中，创建了一个 B 类对象。当 B 对象创建时，调用的是 B 类的构造函数。但是，当对象析构时，却调用的是 A 类的析构函数，B 类的析构函数没有调用，发生了内存泄漏，这是我们不希望看到的。造成这种问题的原因是：当 A 类指针指向的内存单元（即 B 类对象的数据）被释放时，编译器看到指针类型是 A 类的，所以调用 A 类的析构函数。其实，这个时候我们需要调用：指针所指向的对象类型的析构函数，即 B 类析构函数，虚函数能够满足这个要求。所以，在这里我们要使用虚析构函数来解决上面遇到的问题：

例 11-26

```
class A
{
public:
    int *a;
    A() { a=new(int); }
    virtual void func1(){}
    virtual void func2(){}
    virtual ~A() {delete a; cout<<"delete a"<<endl; }
};

class B:public A
{
public:
    int *b;
    B() {b=new(int); }
    virtual void func1(){}
    virtual void func2(){}
    virtual void func3(){}
    virtual ~B() { delete b; cout<<"delete b"<<endl; }
};
```

那么程序的运行结果正是我们所希望的：

delete b

delete a

从程序中我们还可以看到：虚析构函数的工作过程与普通虚函数不同，普通虚函数只是调用相应层上的函数，而虚析构函数是先调用相应层上的析构函数，然后逐层向上调用基类的析构函数。

继承是软件复用的一个重要方法，我们在设计基类和派生类时，必须考虑如何复用基类。设计类时，

有一些基本的原则需要遵循：

（1） 类应该定义有：

◇ public 接口，供派生类或其它类使用。

◇ 友元类或友元函数（如果需要地话）。

◇ protected 接口，供派生类使用。

◇ private 部分。

（2） 如果对象创建了受管的资源（例如申请了堆内存），必须提供析构函数释放该资源。

（3） 在继承链中，如果用基类的指针操作对象。要保证调用正确的析构函数，就必须把基类的析构函数应定义为虚函数。

（4） 如果一个类包含指向另一个对象的指针数据成员，应为该类提供复制构造函数，通常还要提供一个重载的赋值运算符，以确保对象能够被正确地复制和赋值。

（5） 被派生类覆盖的成员函数，通常定义为虚函数。

(1)如果对象创建了受管的资源（例如申请了堆内存），必须提供析构函数释放该资源。

例 11-27

```
class A
{
public:
   int *a;
   A() { a=new(int); }
   //构造函数中分配了堆内存
   virtual void func1(){}
   virtual void func2(){}
   virtual ~A() {delete a; cout<<"delete a"<<endl; }
   //在析构函数中释放资源
};
```

(2)特别是类中存在虚函数的情况下，一定要定义虚析构函数，这样才能保证类的安全性，参看节 11.4 的例程。

面向对象的程序设计是把现实世界看成是是由对象组成的，每一个对象都有自己的数据和对数据的操作。解决具体问题时，先要进行抽象：分析有那些对象，这些对象可以分成几类，各类之间有什么关系等，以确定各个类有那些数据成员，有那些成员函数，它们的访问权限如何，然后，编写出实现代码。我们通过公司雇员和算术表达式两个实例，介绍如何进行面向对象的程序设计。

11.6.1 公司雇员实例

一、分析

对于一个公司的雇员来说，无非三种：普通雇员、管理人员和主管。这些雇员有共同的数据：名字、每小时的工资，也有一些共同的操作：数据成员初始化、读雇员的数据成员及计算雇员的工资。但是，他们也有不同。例如，管理人员除有这些共同的特征外，有可能付固定薪水，主管除有管理人员的共同特征外，还有其它物质奖励等。

三种雇员中，管理人员可以看作普通雇员的一种，而主管又可以看作管理人员的一种。我们很容易想到使用类继承来实现这个问题：普通雇员作为基类，管理人员类从普通雇员类派生，而主管人员类又从管理人员类派生。他们之间的关系如下图所示：

Employee

|

Manager

|

Supervior

二、各个类的定义

三、应用

二、 各个类的定义

我们首先定义普通雇员类：

例 11-28

```
//普通雇员类
class Employee
{
public:
   Employee(char *theName, float thePayRate);
   char *getName() const;
   float getPayRate() const;
   float pay(float hoursWorked) const;

protected:
   char *name; //雇员名称
   float payRate;//薪水等级
};
```

这个类的组成包括：

◇ 一个构造函数初始化类的数据成员

◇ 两个成员函数读数据成员

◇ 一个成员函数计算雇员的工资

各个成员函数的定义如下：

例 11-29

```
Employee::Employee(char *theName, float thePayRate)
{
   name = theName;
   payRate=thePayRate;
}

char *Employee::getName() const
```

```
{
    return name;
}

float Employee::getPayRate() const
{
    return payRate;
}
float Employee::pay(float hoursWorked) const
{
    return hoursWorked * payRate;
}
```

管理人员是普通雇员的一种，它的定义如下：

例 11-30

```
//管理人员类
class Manager : public Employee
{
public:
   //isSalaried  //付薪方式：true  付薪固定工资，false 按小时付薪
   Manager(char *theName, float thePayRate, bool isSalaried);
   bool getSalaried()  const;
   float pay(float hoursWorked) const;
   protected:
   bool salaried;
};
```

成员函数的定义如下：

例 11-31

```
Manager::Manager(char  *theName,float thePayRate,bool isSalaried)
: Employee(theName, thePayRate)
{
   salaried = isSalaried;
}

bool Manager::getSalaried()   const
{
   return salaried;
}
```

```
float Manager::pay(float hoursWorked) const
{
   if (salaried)
   return payRate;
   /* else */
   return Employee::pay(hoursWorked);
}
```

主管人员是管理人员的一种，其定义如下（成员函数被定义为内联的）：

例 11-32

```
//主管人员类
class Supervisor : public Employee
{
public:
   Supervisor(char *theName, float thePayRate, float theBouns):
   Employee (theName, thePayRate, true) ,bouns(theBouns) {}
   float getBouns() const { return bouns; }
   float pay(float hoursWorked) const {
   return payRate+bouns;
}
protected:
   float bouns;
}
```

普通雇员类是所有类的基类，描述了雇员的一些基本信息，管理人员类从普通雇员类派生，管理人员的付薪方式与普通雇员可能同，所以该类添加了一个成员变量标识这种差异，并覆盖了基类的 pay()函数。主管类从管理人员类派生，主管人员是管理人员的一种，他们不仅支付固定薪水，而且还有奖金。所以在主管类种添加了 bonus 成员，保存他们的奖金数额，并覆盖了管理人员类的 pay()函数重新计算工资。

三、应用

定义了类之后，便可以定义这些类的对象：

例 11-33

```
#include "iostream.h"
void main()
{
   Employee e("Jack",50.00);
   Manager m("Tom",8000.00,true);
   Supervior s("Tanya",8000.00,8000.00);
   cout<<"Name:"<<e.getName()<<endl;
   cout <<"Pay:"<<e.pay(80)<<endl;//设每月工作 80 小时
```

```
    cout <<"Name:"<<m.getName()<<endl;
    cout <<"Pay:"<<m.pay(40)<<endl;
    cout <<"Name:"<<s.getName()<<endl;
    cout <<"Pay:"<<s.pay(40)<<endl;//参数 40 这里不起作用
}
```

程序中我们建立了三个雇员（一个普通雇员、一个管理人员和一个主管）的档案，并打印出各自的工资表。事实上，这样将三种雇员分开处理会很繁琐，如果能够把他们看作同一种类型，都看成雇员类（他们本来都是雇员），同一处理，但是计算工资时会调用各自的 pay()函数，那事情会简单得多。这就需要利用多态特性，只要将 pay()函数定义成虚函数，便可以实现了。

11.6.2 算术表达式实例

一、分析

x

2x+3

(2(x-2)(x+1)2+3)/x

(2(x-2)(x+1)+3)/sin(x)

从上面的例子可以看到，算术表达式是有数值、变量、各种运算组合而成的。它们可以分解为以下几种简单的形式：

数字：1、2、4 等

变量：x

数学函数：函数参数可以是一个任意表达式,例如 sin(x3)

加法运算：把任意两个表达式加在一起，例如：2x+3

减法运算：将一个表达式减去另外一个表达式，例如：x-2

乘法运算：把任意两个表达式相乘，例如：(x-2)(x+1)

除法运算：把一个表达式除以另一个表达式，例如：(x+1)+3)/x

乘方运算：乘方也可以是一个任意的表达式，例如：(x-3)(x-7)

一个数或一个变量是最简单的算术表达式。通过运算符将多个算术表达式连接，就能组成新的算术表达式，函数可以看作特殊的运算符，它有一个操作数，而+、-、*、/均是二元运算符。

我们可以按照不同的运算符构造类，定义一个 value()成员函数，实现各个表达式的求值。然而，对于参与运算的表达式，我们并不知道它们各自的具体形式，当然也不可能知道它们的值是怎样计算出来的。那我们怎么能仅仅通过调用一个求值函数 value()就得到这个表达式的值呢？这需要利用类多态的特性：将每个运算符类的求值函数 value()定义成虚函数，这样，当需要得到表达式的值的时候，我们只需要调用 value()函数，而不需要知道它是调用了加法类的 value()，还是减法类的 value()。为此，我们要定义一个表达式类作为基类。这个表达式类，只要定义一个

求值的纯虚函数，也不需要数据成员。不同的运算符类从这个基类派生，分别实现这个 value()函数，各个运算符的参数是表达式类的指针，它们被定义为该运算符类的数据成员。

二、各个类的定义
三、应用

二、 各个类的定义

例 11-34a

```
//算术表达式类
class arithexpr
{
public:
   arithexpr(){}
   virtual ~arithexpr(){}
   virtual double value(double x)=0;
};

//数字类
class numexpr: public arithexpr
{
private:
   double val;
public:
   numexpr(double v):val(v) {}
   ~numexpr() {}
   virtual double value(double x) {return val;}
};

//变量类
class varexpr: public arithexpr
{
public:
   varexpr(){}
   ~varexpr() {}
   virtual double value(double v) {return v;}
};

//加法类
class addexpr: public arithexpr
{
private:
   arithexpr* a1;
```

```
    arithexpr* a2;
public:
    addexpr(arithexpr* e1, arithexpr* e2):a1(e1),a2(e2) {}
    ~addexpr() {delete a1;delete a2;}
    virtual double value(double x) {return a1->value(x)+a2->value(x);}
};

//减法类
class subexpr: public arithexpr
{
private:
    arithexpr* a1;
    arithexpr* a2;
public:
    subexpr(arithexpr* e1, arithexpr* e2):a1(e1),a2(e2) {}
    ~subexpr() {delete a1;delete a2;}
    virtual double value(double x) {return a1->value(x)-a2->value(x);}
};
```

例 11-34b

```
//乘法类
class multiexpr: public arithexpr
{
private:
    arithexpr* a1;
    arithexpr* a2;
public:
    multiexpr(arithexpr* e1, arithexpr* e2):a1(e1),a2(e2) {}
    ~multiexpr() {delete a1;delete a2;}
    virtual double value(double x) {return a1->value(x)*a2->value(x);}
};

//除法类
class divexpr: public arithexpr
{
private:
    arithexpr* a1;
    arithexpr* a2;
public:
    divexpr(arithexpr* e1, arithexpr* e2):a1(e1),a2(e2) {}
    ~divexpr() {delete a1;delete a2;}
    virtual double value(double x) {return a1->value(x)/a2->value(x);}
```

```
};

//乘方类
class powexpr: public arithexpr
{
private:
   arithexpr* a1;
   arithexpr* a2;
public:
   powexpr(arithexpr* e1, arithexpr* e2):a1(e1),a2(e2) {}
   ~powexpr() {delete a1;delete a2;}
   virtual double value(double x) {return pow(a1->value(x),a2->value(x));}
};

//算术函数类
typedef double (*func)(double);
class funcexpr: public arithexpr
{
private:
   func f;
   arithexpr* arg;
public:
   funcexpr (func fn,arithexpr* a):arg(a),f(fn){}
   ~ funcexpr () {delete arg;}
   virtual double value(double x) {return f(arg->value(x));}
};
```

三、 应用

例 11-35

```
#include "arithm.h"
#include "iostream.h"
void main()
{
   arithexpr* a;
   //表达式 2x+3
   a=new addexpr(new multiexpr(new numexpr(2),new varexpr()),new numexpr(3));
   cout<<"当 x=3 时，2x+3="<<a->value(3)<<endl;

   arithexpr* a1=new numexpr(3);
   arithexpr* a2=new varexpr();
   arithexpr* a3=new subexpr(a2,a1);
```

```
    arithexpr* a4=new powexpr(a3,a1);
    arithexpr* a5=new funcexpr(sin,a4);
    cout<<"x=5 时，sin((x-3)^3)="<<a5->value(5)<<endl;
    cout<<"x=4 时，sin((x-3)^3)="<<a5->value(4)<<endl;
}
```

程序中我们举了两个算术表达式计算的例子，对于不同的表达式，当变量取不同的值时，我们只需要很简单地调用 value()成员函数，对表达式进行求值。在这里，我们只处理了一个变量的情形，如果是多个变量，情况就会复杂一些。

例 11-36a

```
//arithm.h 文件为：
#ifndef ARITHEXPR_H
#define ARITHEXPR_H
#include <math.h>

class ArithExpr {
public:
   ArithExpr() {}
   virtual ~ArithExpr() {}
   virtual double valueAt(double x)=0;
};

class NumExpr : public ArithExpr {
private:
   double val;
public:
   NumExpr(double v): val(v) {}
   ~NumExpr() {}
   virtual double valueAt(double x) { return val; }
};

class VarExpr  : public ArithExpr {
public:
   VarExpr() {}
   ~VarExpr() {}
   virtual double valueAt(double v) { return v; }
};

class AddExpr : public ArithExpr {
private:
   ArithExpr* e1;
   ArithExpr* e2;
```

```
public:
   AddExpr(ArithExpr* left, ArithExpr* right): e1(left),e2(right) {}
   ~AddExpr() { delete e1; delete e2; }
   virtual double valueAt(double v)
      {return e1->valueAt(v) + e2->valueAt(v); }
};
```

例 11-36b

```
class SubExpr : public ArithExpr {
private:
   ArithExpr* e1;
   ArithExpr* e2;
public:
   SubExpr(ArithExpr* l, ArithExpr* r): e1(l),e2(r) {}
   ~SubExpr() { delete e1; delete e2; }
   virtual double valueAt(double v)
      { return e1->valueAt(v) - e2->valueAt(v); }
};

class MulExpr : public ArithExpr {
private:
   ArithExpr* e1;
   ArithExpr* e2;
public:
   MulExpr(ArithExpr* left, ArithExpr* right): e1(left),e2(right) {}
   ~MulExpr() { delete e1; delete e2; }
   virtual double valueAt(double v)
      { return e1->valueAt(v) * e2->valueAt(v); }
};

class DivExpr : public ArithExpr {
private:
   ArithExpr* e1;
   ArithExpr* e2;
public:
   DivExpr(ArithExpr* left, ArithExpr* right): e1(left),e2(right) {}
   ~DivExpr() { delete e1; delete e2; }
   virtual double valueAt(double v)
      { return e1->valueAt(v) / e2->valueAt(v); }
};
```

例 11-36c

```
class PowExpr : public ArithExpr {
private:
   ArithExpr* e1;
   ArithExpr* e2;
public:
   PowExpr(ArithExpr* base, ArithExpr* exp): e1(base), e2(exp) {}
   ~PowExpr() { delete e1; delete e2; }
   virtual double valueAt(double v)
      { return pow(e1->valueAt(v), e2->valueAt(v)); }
};

typedef double (*UnaryFunction)(double);
   // Pointer to function that takes one arg of type double
   // and returns a value of type double

class FunExpr: public ArithExpr {
private:
   double (*f)(double);
   ArithExpr* a;
public:
   FunExpr(UnaryFunction fn, ArithExpr* arg): f(fn),a(arg) {}
   ~FunExpr() { delete a; }
   virtual double valueAt(double v) { return f(a->valueAt(v)); }
};
#endif
```

输入 Menu:A(dd),D(elete),S(ort),Q(uit)

然后由用户输入 A,D,S,Q

然后再相应的执行程序

公司雇员实例

由于 Employee 和 Manager 类的成员函数均比较简单，所以大多数的成员函数定义为内联函数，并将 pay 成员函数定义为虚函数：

例 11-37

```
#include "employee.h"
//普通雇员类
class Employee
{
public:
   //构造函数
   Employee(string theName, float thePayRate):
   name(theName),payRate(thePayRate) {}
```

```
    //取雇员姓名
    string getName()  const {return name; }
    //取雇员薪水等级
    float getPayRate() const { return payRate; }
    //计算雇员薪水
    virtual float pay(float hoursWorked) const { return hoursWorked*payRate; }

protected:
    string name; //雇员名称
    float payRate;//薪水等级
};

//管理人员类
//继承普通雇员类
class Manager : public Employee
{
public:
    //构造函数
    //isSalaried 标识管理人员类的付薪方式
    //true  按阶段付薪（固定工资）
    //false  按小时付薪
    Manager(string theName, float thePayRate, bool isSalaried):
    Employee(theName,thePayRate),salaried(isSalaried) {}
    //取付薪方式
    bool getSalaried()  const { return salaried; }
    //计算薪水
    virtual float pay(float hoursWorked) const ;
protected:
    bool salaried;
};
float Manager ::pay(float hoursWorked) const
{
    if (salaried) //固定付薪方式
        return payRate;
    else //按小时付薪
        return hoursWorked * payRate;
}

//主管人员类
class Supervisor : public Manager
{
public:
```

```
    //构造函数
    Supervisor（string theName, float thePayRate, float theBouns）:
    Manager(theName, thePayRate, true), bouns(theBouns) {}
    //取奖金数额
    float getBouns() const { return bouns; }
    //计算薪水
virtual float pay(float hoursWorked) const
{
    return payRate+bouns;
}
protected:
    float bouns;
}
```

我们把上面的源程序进行了修改，将 pay 函数变成虚函数。这样就可以象下面这样方便地使用了：

例 11-38

```
#include "employee.h"
#include "iostream.h"
void main()
{
    Employee *ep[3];
    ep[0]=new Employee("Jack","50.00");
    ep[1]=new Manager("Tom","8000.00",true);
    ep[2]=new Supervior("Tanya","8000.00","8000.00");
    for (int i=0;i<3;i++) {
        Cout<<"Name:"<<ep[i]->getName()<<endl;
    Cout<<"Pay:"<<ep[i]->pay(80)<<endl;//设每月工作 80 小时
    }
}
```

11.6.3 算术表达式实例

这个程序有个问题，就是没有内存释放语句，仔细观察你会发现，对于指针 a，我们可以使用"delete a;"来释放它的内存单元，但是第二个表达式就没有那么简单了。当我们释放指针 a5 指向的对象时，它先调用析构函数，释放 a4 指向的对象，而 a4 的析构函数将释放 a3 和 a1 指向的对象，下面问题就来了：a3 的析构函数会再次释放 a1 的内存单元，释放一个空指针，这样会使系统崩溃。

对于这个问题，我们可以这样简单地解决：在析构函数中用 delete 销毁对象时，先判断指针是否为空，不为空再释放空间，并在释放后立即把指针的值置为 NULL。事实上，这也是一个很好的习惯。

例 11-39

```
#include "arithm.h"
#include "iostream.h"
```

```
void main()
{
   arithexpr* a;
   //表达式 2x+3
   a=new addexpr(new multiexpr(new numexpr(2),new varexpr()),new numexpr(3));
   cout<<"当 x=3 时，2x+3="<<a->value(3)<<endl;
   //delete a;
   arithexpr* a1=new numexpr(3);
   arithexpr* a2=new varexpr();
   arithexpr* a3=new subexpr(a2,a1);
   arithexpr* a4=new powexpr(a3,a1);
   arithexpr* a5=new funcexpr(sin,a4);
   //delete a5;
   cout<<"x=5 时，sin((x-3)^3)="<<a5->value(5)<<endl;
   cout<<"x=4 时，sin((x-3)^3)="<<a5->value(4)<<endl;
}
```

在这个例子中，从多边形基类 CPolygon 派生出了矩形类 CRectangle 和三角形类 CTriangle。

CPolygon 类中定义了两个保护成员 width、height，分别表示多边形的宽和高。另外，定义了三个公有成员函数 set_values、area、printarea。其中 set_values 用于设置数据成员的值；area 用于计算多边形的面积，它是一个纯虚函数，没有实现，要在派生类实现；printarea 用于打印多边形的面积。由于 CPolygon 类有一个纯虚函数 area，所以，它是一个抽象类，不能用它定义对象，且它只能作为基类。

在两个派生类 CRectangle 和 CTriangle 中，仅仅重定义了 area 函数，以覆盖基类 CPolygon 的 area 函数。它们的数据成员、对数据成员的设置及对面积的输出的成员函数均从基类继承得到。

在 main 函数中，分别定义了一个一个 CRectangle 对象 rect 和一个 CTriangle 对象 trgl，定义了两个 CPolygon 指针 ppoly1 和 ppoly2，并让它们分别指向对象 rect 和 trgl。语句：

```
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
```

是分别设置对象 rect 和 trgl 的数据成员，由于 set_values 是非虚函数，ppoly1 和 ppoly2 都是 CPolygon 指针，所以，这是调用 CRectangle 的 set_values 函数。而语句：

```
ppoly1->printarea();
ppoly2->printarea();
```

则存在多态调用。同样，这是调用 CRectangle 的 printarea 函数，但是，在 printarea 内调用 area 虚函数，而当前的 this 指针是分别指向对象 rect 和 trgl，且这两个对象中分别覆盖了 CPolygon 类的 area 虚函数，所以，上面两个语句在 printarea 函数中，分别调用对象 rect 和 trgl 中的 area 虚函数。运行该程序，得到的运行结果如下：

```
20
10
```

例 11-40a

```
#ifndef ARITHEXPR_H
#define ARITHEXPR_H
```

```
#include <math.h>

class ArithExpr {
public:
   ArithExpr() {}
   virtual ~ArithExpr() {}
   virtual double valueAt(double x)=0;
};

class NumExpr : public ArithExpr {
private:
   double val;
public:
   NumExpr(double v): val(v) {}
   ~NumExpr() {}
   virtual double valueAt(double x) { return val; }
};

class VarExpr : public ArithExpr {
public:
   VarExpr() {}
   ~VarExpr() {}
   virtual double valueAt(double v) { return v; }
};

class AddExpr : public ArithExpr {
private:
   ArithExpr* e1;
   ArithExpr* e2;
public:
   AddExpr(ArithExpr* left, ArithExpr* right): e1(left),e2(right) {}
   ~AddExpr() { delete e1; delete e2; }
   virtual double valueAt(double v)
      { return e1->valueAt(v) + e2->valueAt(v); }
};
```

例 11-40b

```
class SubExpr : public ArithExpr {
private:
   ArithExpr* e1;
   ArithExpr* e2;
public:
```

```
    SubExpr(ArithExpr* l, ArithExpr* r): e1(l),e2(r) {}
    ~SubExpr() { delete e1; delete e2; }
    virtual double valueAt(double v)
        { return e1->valueAt(v) - e2->valueAt(v); }
};

class MulExpr : public ArithExpr {
private:
    ArithExpr* e1;
    ArithExpr* e2;
public:
    MulExpr(ArithExpr* left, ArithExpr* right): e1(left),e2(right) {}
    ~MulExpr() { delete e1; delete e2; }
    virtual double valueAt(double v)
        { return e1->valueAt(v) * e2->valueAt(v); }
};

class DivExpr : public ArithExpr {
private:
    ArithExpr* e1;
    ArithExpr* e2;
public:
    DivExpr(ArithExpr* left, ArithExpr* right): e1(left),e2(right) {}
    ~DivExpr() { delete e1; delete e2; }
    virtual double valueAt(double v)
        { return e1->valueAt(v) / e2->valueAt(v); }
};
```

例 11-40c

```
class PowExpr : public ArithExpr {
private:
    ArithExpr* e1;
    ArithExpr* e2;
public:
    PowExpr(ArithExpr* base, ArithExpr* exp): e1(base), e2(exp) {}
    ~PowExpr() { delete e1; delete e2; }
    virtual double valueAt(double v)
        { return pow(e1->valueAt(v), e2->valueAt(v)); }
};

typedef double (*UnaryFunction)(double);
// Pointer to function that takes one arg of type double
```

```
// and returns a value of type double

class FunExpr: public ArithExpr {
private:
   double (*f)(double);
   ArithExpr* a;
public:
   FunExpr(UnaryFunction fn, ArithExpr* arg): f(fn),a(arg) {}
   ~FunExpr() { delete a; }
   virtual double valueAt(double v) { return f(a->valueAt(v)); }
};
#endif
```

【本章小结】

面向对象程序设计的主要特征是抽象、封装、继承和多态。面向对象的程序设计思想是现代软件工程很重要的思想，我们要通过本章的学习，理解它的精髓。

用进行面向对象程序设计方法解决实际问题，先要进行抽象，确定需要哪些类，各类有哪些数据和操作，各类的成员如何访问和继承等。

封装是通过类实现的，类的成员包括数据和函数，都可以被声明为公有、保护或私有。定义类之后，就可定义该类的对象；利用点运算符（.）可以访问对象的公有成员。

继承的重要性是支持程序代码复用，它能够从已存在的类中派生出新类，继承基类的成员，而且可以通过覆盖基类成员函数，产生新的行为。派生可以从一个基类派生，称为单继承；也可以从多个基类派生，称为多继承；派生类也可以再派生出新类，称为多级继承。

多态性是面向对象程序设计又一重要特征，多态性是通过虚函数实现的。多态是一个强大的功能，通过虚函数来实现，它能够用同一个指针访问不同对象的、具有相同信号的成员函数。没有掌握多态，就不能说掌握了面向对象的程序设计技术。

如果我们能够很好地利用面向对象的技术编写应用程序，会使应用程序代码更简洁、可读性更好，且更易于扩展和维护。

学习目标】

◇ 了解 I/O 流类的层次结构

◇ 掌握 C++标准输入输出流的用法

◇ 能够使用操纵算子格式化输入输出

◇ 掌握文件流的用法

◇ 了解字节流的用法

◇ 能够确定输入输出流的状态，并进行流错误处理

【重点和难点】

◇ C++标准输入输出流的使用（重点）

◇ 操纵算子的使用（重点，难点）

◇ 文件流的用法(重点)

◇ 流错误处理（难点）

【学习方法指导】

本章涉及到较多的面向对象的概念，初学时，关键是要掌握输入输出流的用法，而对输入输出流库中的许多细节可不必深究。深层次的内容，可在以后的应用中，逐渐地领会和掌握。

【知识点】

输入输出流类层次；标准输出；标准输入；操纵算子；文件流；字节流类；输入流；输出流

C++语言同 C 语言一样，也不具有内部输入输出能力，这样做的目的是为了最大限度地保证语言与平台的无关性。计算机语言的输入输出功能都是与操作系统相关的，如果 C++为某种操作系统实现内部输入输出功能，那它也就被限制在这个操作系统上了，这是我们所不希望的。

如果一个应用程序没有输入和输出，那它也就没有应用价值。在 C++中，输入输出功能，是通过调用该操作系统的 I/O 库来实现的。

scanf、printf 都是 C 语言标准输入输出库函数，不能否认，C 语言的标准输入输出库函数也是安全、高效的，为什么说 C++的输入输出更安全高效呢？关键在于 C++的输入输出与 C 的输入输出实现方法不同。

C++的输入输出是如何实现的？是不是依据面向对象的思想，把 C 的标准 I/O 库封装成类，然后进行处理呢？比如操作一个文件，我们想确保文件能够安全地打开，及时地关闭，而不完全依赖用户调用 open()、close()函数，可以构造一个文件类，定义一个成员变量，作为文件指针，分别在构造函数和析构函数中打开、关闭文件。再进一步，可以在类中实现 C 的标准输入输出库中所有的 I/O 函数，并把文件指针置为私有变量。从外面看来，这已经是一个封装得很好的文件类。在某种意义上讲，这种方法是相当有效的，我们也可以为标准 I/O 和存储块构造类似的类。

我们已经知道，C++是使用 iostream 流库，并没有使用 C 的输入输出库，是不是 iostream 流库更好呢？答案是肯定的。C 的 I/O 库函数主要用来处理基本数据类型（字符、整型、浮点数等），使用参数表进行数据传输，使用格式字符串指定数据类型和输入输出格式。它们在运行时对格式字符串进行语法分析，并据此对变量进行解释。例如：

```
printf("a=%d\tb=%d\ta+b=%d\n",a,b,a+b);
```

C 语言 I/O 库的缺点是：

1. 即使只使用了解释程序的一个功能，也要全部装载。如上面的例子，我们要装载整个包，包括解释浮点数和字符串那部分程序段，无法减少程序的长度。

2. 虽然 printf 族函数已经优化得很好，但是，它是在运行期间进行解释，如果能在编译期间分析格式字符串里的变量，根据不同的类型调用各自的函数处理，运行会快得多，而且 C++编译期间的类型检查会有助于我们发现错误。

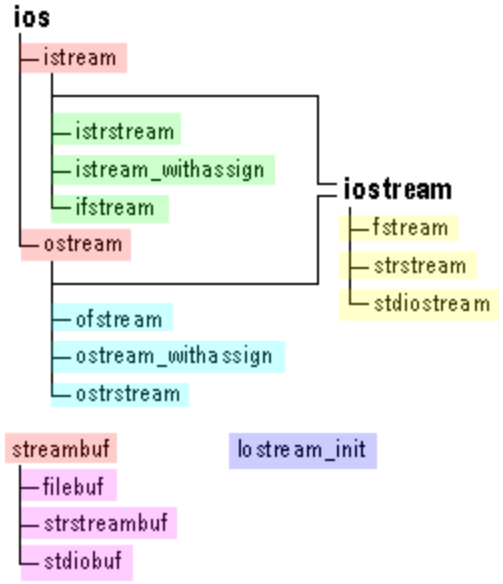
3. 对于 C++来说，printf 不能被扩展是它最大的缺点。我们不能通过重载函数对它进行扩展，因为重载函数要有不同类型的参数，而 printf 族函数把类型信息隐藏在可变参数表和格式字符串中。

iostream 是通过类的继承，类成员函数的重载来实现的，利用类的可继承性和多态性，使 iostream 类库使用统一的函数接口操作标准 I/O、文件、存储块等输入输出设备。通过函数重载，为每种内部数据类型定义了流输入输出函数，使得用户可以用相同的格式对各种数据类型进行操作，编译程序根据数据的类型自动选择相应的输入输出函数，不必将所有函数一并加载。同时，iostream 拥有了很好的扩展性，用户通过重载还可以对自定义对象进行流的操作。因此，与标准 C 输入输出库的各种各样的函数相比，输入输出流更容易、更安全、更有效。

12.1.1 输入输出流类层次

iostream 是一组 C++类，用于实现面向对象模型的输入输出，可以提供无缓冲的（低级）和缓冲的 I/O 操作。在某些情况下，如果 C++编译器提供的 iostream 库中没有合适的输入输出函数可用，我们还可以利用类的继承和多态特性来改进它们，左图给出了 I/O 流类之间的继承关系。

图 12-1 I/O 流类层次图



抽象流基类

ios	流基类
输入流类	
istream	普通输入流类和用于其它输入流的基类
ifstream	输入文件流类
istream_withassign	用于 cin 的输入流类
istrstream	输入串流类
输出流类	
ostream	普通输出流类和用于其它输出流类的基类
ofstream	输出文件流类
ostream_withassign	用于 cout、cerr 和 clog 的流类
ostrstream	输出串流类
输入输出流类	
iostream	普通输入输出流类和用于其它输入输出流的基类
fstream	输入输出文件流类
strstream	输入输出串流类
stdiostream	用于标准输入输出文件的输入输出类
缓冲流类	
streambuf	抽象缓冲流基类
filebuf	用于磁盘文件的缓冲流类
strstreambuf	用于串的缓冲流类
stdiobuf	用于标准输入输出文件的缓冲流类
预定义流初始化类	
iostream_init	预定义流初始化的类

其中，ios、istream、ostream 和 streambuf 类构成了 C++中 iostream 输入输出功能的基础。

在图中，streambuf 类和 ios 类没有继承关系，它们是组合使用的，streambuf 对象作为 ios 类的成员出

现。事实上，ios 类提供编程界面和格式特征，而 streambuf 做实质性的工作。

iostream 类库中，streambuf、ios、istream、ostream、iostream、istream_withassign 和 ostream_withassign 这些基本 I/O 流类和预定义的 cin、cout、cerr 和 clog 在 iostream.h 文件中说明。

filebuf、ifstream、ofstream 和 fstream 在 fstream.h 中说明。

strstream、istrstream、ostrstream 和 strstream 在 strstrea.h 中说明。

需要注意的是：fstream.h 和 strstrea.h 中都包含了 iostream.h，所以如果使用标准输入输出（控制台 I/O），只要包含 iostream.h 头文件即可，如果使用 fstream 或者 strstream，只要包含相应的 fstream.h 和 strstrea.h 即可。

12.1.2 标准输入和输出

输出流类在 iostream.h 中预定义了四个全局的流对象：cout、cerr、clog 和 cin，用于标准输出和输入，cout 和 cin 在程序设计中会经常用到。

cout 流对象控制向控制台（显示器）的标准输出，cin 控制从控制台（键盘）输入。cerr 与标准错误设备连在一起，是非缓冲输出，也就是说插入到 cerr 的数据会被立刻显示出来，非缓冲输出可以迅速把出错信息告知用户。clog 也是与标准错误设备连在一起的，但它是缓冲输出。

一、标准输出

程序 12-1 中出现了<<运算符，它是 C++中的逐位左移运算符。不过，此运算符在这里被重载，用于流输出操作，称之为插入运算符。它不仅完成逐位左移，而且还能够将对象插入到数据流中，而数据流可以连接到任何与计算机相连的输出设备。cout 对象是与控制台相连的，当对象插入到 cout 中时，它们被发送到控制台（监视器）。例如程序 12-1 完成的是向控制台输出"pi=3.14159"。同样，其它内部数据类型也可以插入到数据流中。因为<<运算符返回的是 ostream 对象的引用，所以程序 12-1 可以简化成：

```
cout<<"pi="<<pi;
```

这样可以使程序更加简洁易懂。

引入了插入运算符<<，使 ostream 类具有了通用性和设备无关性，这是通过多态来实现的。用户可以通过重载插入运算符，将自定义的数据类型进行翻译转换，插入到数据流中，这在后面要详细介绍。

应用<<运算符，执行格式化插入到 ostream 类中的对象并不象它们原来那样发送到输出，而是首先被转换成简单的数据类型（字节流）。但是在许多情况下，我们希望将数据直接插入到流中而不进行转换。为此 ostream 类提供了两个函数：put()和 write()。前者是将一个字符、无符号字符或者有符号字符插入到字节流中；后者用于插入一个字符数组到字节流中，但不进行任何转换，用于将非格式化的二进制组写到文件和输出设备中。

程序 12-2 将字符'A'和一组代表非格式 float 值的字节 1.4（4 个字节）直接显示到控制台上，显示到控制台的结果实际上是一堆乱码。

程序 12-1

```
#include
<iostream.h>
void main()
{
   float
pi=3.14159;
```

```
    cout<<"pi=";
    cout<<pi;
    cout<<endl;
}
```

程序 12-1 的输出结果为：

pi=3.14159

程序 12-2

```
#include <iostream.h>
void main()
{
   float pi=1.4;
   cout.put（'A'）;
   cout.write((unsigned char*)&pi,sizeof(float));
}
```

二、标准输入

程序 12-5 出现了>>运算符，它是 C++中的逐位右移运算符。不过，此运算符在这里也被重载，用于流输入操作，称之为抽取运算符。它不仅完成逐位右移，而且还能够从数据流中抽取数据对象。而数据流可以连接到任何与计算机相连的输入设备，cin 是与控制台输入相连。当控制台输入数据时，cin 中生成数据并保存到相应变量中，如程序 12-5 所示。由于>>操作符返回的是 istream 对象的引用，所以程序 12-5 可以简化成：

```
cin>>n>>ch>>pi>>str;
```

这里有一个问题。在程序 12-5 中，当输入"5 c 3.14159 hello world!"时，运行结果并不是我们所期望的：

n=5

ch=c

pi=3.14159

string=hello world!

而是：

i=5

ch=c

pi=3.14159

string=hello

str 只得到第一个字"hello"，而不是"hello world!"，这是由于输入机制是通过寻找空格来分隔输入的，而空格在"hello"的后面，所以"world!"被截掉了。要解决这个问题，可以使用非格式化抽取的方法。

程序 12-5

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

```
    cin>>n;
    char ch;
    cin>>ch;
    float pi;
    cin>>"pi=";
    char str[20];
    cin>>str;
    cout<<"n="<<n<<endl;
    cout<<"ch="<<ch<<endl;
    cout<<"pi="<<pi<<endl;
    cout<<"string="<<str<<endl;
}
```

当输入是：5 c 3.14159 hello 时，程序运行结果是：

n=5

ch=c

pi=3.14159

string=hello

istream 有三个从流中进行非格式化抽取的成员函数：get()、getline()和 read()。

流输入的抽取过程由于类型不匹配等原因，有可能会出现错误，当一步抽取出现错误，以后各步的抽取过程也是错误的。所以在抽取数据期间测试错误也是很重要的，我们将在后面单独进行介绍。

表 12-1 成员函数 get()的用法

int get()	从流中抽取单个字符并返回。
istream& get(char*,int,char)	从流中抽取字符直到终止符（缺省为'\n'）或者抽取字符达到第二个参数给定的数量或者已到文件尾，将其存储在第一个参数指定的字符数组里。
istream& get(char &)	从流中抽取单个字符并存入引用变量中
istream& get(streambuf &,char)	从流中取得字符存入 streambuf 对象，直到终止符或文件尾

get()在上述任何情况下，都不从流中提取终止符。

表 12-2 成员函数 getline()的用法

istream& getline(char*,int,char)	从流中抽取字符直到终止符（缺省为'\n'），或者抽取字符达到参数给定的数量-1，或者已到文件尾，将其存储在第一个参数指定的字符数组里。如果发现终止符，它从流中提取终止符，但只是抛弃掉，并不把它存在结果缓冲区里。

表 12-3 成员函数 read()的用法

istream& read(char*,int)	从流中抽取字节直到参数给定的数量或者到文件结束，将其存储在第一个参数指定的字符数组里。该函数用于二进制输入流。

程序 12-5 出现了>>运算符，是 C++中的逐位右移运算符。该运算符在这里也被重载，用于流输入操作，称之为抽取运算符。

下面是 iostream.h 中 istream 类的部分定义。

程序 12-6

```
/*
* Formatted extraction operations
*/
istream _FAR & _Cdecl operator>>(istream _FAR & (_Cdecl *_f)(istream _FAR &));
istream _FAR & _Cdecl operator>>(ios _FAR & (_Cdecl *_f)(ios _FAR &) );
istream _FAR & _Cdecl operator>>( signed char _FAR *);
istream _FAR & _Cdecl operator>>(unsigned char _FAR *);
istream _FAR & _Cdecl operator>>(unsigned char _FAR &);
istream _FAR & _Cdecl operator>>( signed char _FAR &);
istream _FAR & _Cdecl operator>>(short _FAR &);
istream _FAR & _Cdecl operator>>(int _FAR &);
istream _FAR & _Cdecl operator>>(long _FAR &);
istream _FAR & _Cdecl operator>>(unsigned short _FAR &);
istream _FAR & _Cdecl operator>>(unsigned int _FAR &);
istream _FAR & _Cdecl operator>>(unsigned long _FAR &);
istream _FAR & _Cdecl operator>>(float _FAR &);
istream _FAR & _Cdecl operator>>(double _FAR &);
istream _FAR & _Cdecl operator>>(long double _FAR &);

// extract from this istream, insert into streambuf
istream _FAR & _Cdecl operator>>(streambuf _FAR *);
```

从上面运算符重载的定义可以看到：>>运算符返回的数据类型是 istream，这样我们就可以很好的解释类似 cin>>a>>b>>c>>d 写法。cin>>a>>b>>c>>d 可以写成（（（cin>>a）>>b）>>c）>>d，程序先执行 cin>>a，返回的是 istream 参数，传给第二个>>运算符，依此类推，实际上是前一个函数的返回值作为后一个函数的参数，类似 f1(a,f2(b,f3(c,f4(d))))。插入运算符<<也可以有与抽取运算符>>类似的用法，例如：cout<<a<<b<<c<<d;原因就不再赘述了。

12.1.3 操纵算子

操纵算子是插入到流中或从流中抽取出来、影响流的格式状态的函数或对象。流的格式状态由 ios 类定义，其中包括指定数据对象的基数，如十进制、八进制、十六进制等，还有输出宽度、精度、填充字符等等。事实上，ios 类有自己的成员函数可以设置、清除这些格式变量。操纵算子与这些成员函数的功能是

重叠的，但是引入操纵算子为我们提供了很大的方便和表达能力，它们有助于提高程序的可读性。

表 12-4 列出了 I/O 流类预定义的操纵算子，其中的无参数操纵算子在 iostream.h 中定义，在使用时应包含该头文件。另外，在使用带参数操纵算子时，应包含 iomanip.h 头文件，这个文件中包含了解决建立带参数操纵算子遇到的一般问题的代码。

表 12-4 操纵算子清单

操纵算子	作用
endl	输出换行符并刷新流
ends	输出空字符
flush	刷新流
dec	将数值转化为十进制
hex	将数值转化为十六进制
oct	将数值转化为八进制
ws	跳过空白字符（用于输入）
setiosflags(fmtflags n)	设置由 n 指定的格式标志；设置一直起作用直到下一个变化为止，作用同 ios::setf()。格式标志在 ios 类中定义，参见表 12-5。
setbase(base n)	把基数改成 n，这里 n 取 10、8 或 16（任何别的值结果为 0）。如果 n 是 0，输出基数为 10，但输入使用 c 语言的约定： 10 是 10，010 是 8 而 0xf 是 15。推荐使用 dec、oct 和 hex
setfill(char n)	把填充字符改成 n，作用同 ios::fill()
setprecision(int n)	把精度改成 n，作用同 ios::precision()
setw(int n)	把域宽改成 n，作用同 ios::width()，只影响它后面的要输出的数据，当这个数据打印完毕后，域宽恢复到缺省状态

表 12-5 格式标志清单

showbase	在打印一整数值时,标明数字基数(十进制,八进制和十六进制);所用的格式能被 c＋＋编译器读出
noshowbase	
showpos	显示正值符号加(+)
noshowpos	
uppercase	显示代表十六进制值的大写字母 a - f 以及科学记数法中的 e
nouppercase	
showpoint	表明浮点数值的小数点和后面的零
skipws	跳过输入中的空白字符
noskipws	
left	左对齐，右填充
right	右对齐，左填充

internal	在引导符或基数指示符和值之间填充
scientific	使用科学记数法
fixed	setprecision()或 ios::precision()设置小数点后面的位数

程序 12-7

```
#include "iostream.h"
#include "iomanip.h"
#include "strstrea.h"

void main()
{
   int n=15;
   double f=3.14159;
   cout<<"n="<<n<<endl;
   cout<<"f="<<f<<endl;

   cout<<"n="<<setw(10)<<n<<endl;
   cout<<"f="<<setw(10)<<f<<endl;

   cout<<setfill('*')<<setiosflags(ios::left);
   cout<<"n="<<setw(10)<<n<<endl;
   cout<<"f="<<setw(10)<<f<<endl;

   cout<<setfill(' ');
   cout<<"n="<<setw(10)<<setiosflags(ios::showbase)<<hex<<n<<endl;
   cout<<"f="<<setw(10)<<setiosflags(ios::scientific)<<f<<endl;}
```

程序 12-7 的输出结果为：

```
n=15
f=3.14159
n= 15
f= 3.14159
n=15********
f=3.14159***
n=0xf
f=3.141590e+000
```

下表中列出了 ios 中控制流格式的成员函数：

表 12-6

Flags	设置或读取流的格式标志
Setf	操纵流的格式标志
Unsetf	清除流的格式标志

Fill	设置或读取流的填充字符
Precision	设置或读取流的浮点数显示精度
Width	设置或读取流的输出宽度

ios 类中定义的格式标志：

程序 12-8

```
// formatting flags
enum {
   skipws = 0x0001, // skip whitespace on input
   left = 0x0002, // left-adjust output
   right = 0x0004, // right-adjust output
   internal = 0x0008, // padding after sign or base indicator
   dec = 0x0010, // decimal conversion
   oct = 0x0020, // octal conversion
   hex = 0x0040, // hexadecimal conversion
   showbase = 0x0080, // use base indicator on output
   showpoint = 0x0100, // force decimal point (floating output)
   uppercase = 0x0200, // upper-case hex output
   showpos = 0x0400, // add '+' to positive integers
   scientific= 0x0800, // use 1.2345E2 floating notation
   fixed = 0x1000, // use 123.45 floating notation
   unitbuf = 0x2000, // flush all streams after insertion
   stdio = 0x4000 // flush stdout, stderr after insertion
};
```

iostream 中预定义的操纵算子：

程序 12-9

```
/*
* Manipulators
*/
ostream _FAR & _Cdecl endl(ostream _FAR &);
      //insert newline and flush
ostream _FAR & _Cdecl ends(ostream _FAR &);
      //insert null to terminate string
ostream _FAR & _Cdecl flush(ostream _FAR &);
      //flush the ostream
ios _FAR & _Cdecl dec(ios _FAR &);
      //set conversion base to decimal
ios _FAR & _Cdecl hex(ios _FAR &);
      //set conversion base to hexadecimal
```

```
ios _FAR & _Cdecl oct(ios _FAR &);
    //set conversion base to octal
istream _FAR & _Cdecl ws(istream _FAR &);
    //extract whitespace characters
```

表 12-4 所列操纵算子有很多适用于输入流，但实际只有少数几个影响输入流对象，其中最重要的就是 dec、hex 和 oct，它们决定输入流进制的转换方式。

程序 12-10

```
#include "iostream.h"
void main()
{
  int n;

  while (cin){
    cin>>hex>>n;
    cout<<"n="<<n<<endl;
  }
}
```

当输入 1 10 f F 0xf 0XF 时，程序的运行结果为：

```
n=1
n=16
n=15
n=15
n=15
n=15
```

又例如：

```
cin>>hex>>n;
```

这条语句的功能是将输入流中的数据按十六进制的格式取出，并保存到变量 n 中。如果将 hex 改为 oct，那么该语句的作用变为将输入流按八进制的格式取出，并保存到变量 n 中。当输入 1 10 010 时，程序的运行结果为：

```
n=1
n=8
n=8
```

12.1.4 用户自定义对象的插入与抽取

iostream 之所以具有可扩展性，是因为它不仅可以对内部数据类型进行插入和抽取，而且用户可以根据需要，重载插入和抽取运算符，实现对自定义数据类型的插入和抽取。

下面的实例是一个复数类的说明：

程序 12-11

```
class complex
```

```
{
private:
   double a;
   double b;
public:
   complex();
   complex(double i, double j);
   complex operator+(complex& x);
   complex operator-(complex& x);
   complex operator*( complex& x);
   complex operator/( complex& x);
   void setreal(double x);
   void setimag(double y);
   double getreal();
   double getimag();
   friend ostream& operator<<(ostream& Out,complex& x);
   friend istream& operator>>(istream& In, complex& x);
};
```

在这里，重载的插入和抽取运算符被定义成复数类的友元，通过定义友元函数，可以访问该类的私有成员。下面是它们的实现：

程序 12-12

```
ostream& operator<<(ostream& Out, complex& x)
{
   Out<<x.a<<"+j"<<x.b;
   return Out;
}

istream& operator>>(istream& In, complex& x)
{
   In>>x.a>>x.b;
   return In;
}
```

复数类的其它成员函数的实现略去，下面是应用复数类的例子：

程序 12-13

```
void main()
{
   complex a,b,c;
   cin>>a>>b;
```

```
    c=a+b;
    cout<<c;
}
```

输入 2 3 4 5，程序运行结果为：

6+j8

利用文件流操作打开一个文件，只需要建立一个对象，它的构造函数负责打开文件，当该对象生存期结束时，它会调用析构函数关闭文件。当然，我们也可以调用成员函数 open()和 close()进行文件的打开和关闭，下面这个例子说明了如何用文件流进行文件操作。

文件流类其实是输入输出流类的一部分，由于在实际中，会经常用到文件操作，所以我们把文件流类再单独介绍。

程序 12-14

```
#include "fstream.h"
//使用文件流要包含 fstream.h，该文件内部包含了 iostream.h

#include "assert.h"
//包含这个头文件是为了调用 assert()

void main()
{
  int line;
  char buf[80];
  ifstream in("in.txt");
  assert(in);
  //断言函数，确保文件被成功打开。
  ofstream out("out.txt");
  assert(out);
  //在每一行前面加上行号
  while (in.getline(buf,80)){
    cout<<line++<<":"<<buf<<endl;
    out<<buf<<endl;
  }
}
```

该程序实现从文件"in.txt"将数据读入，在屏幕上显示，并输出到"out.txt"文件中

一、打开文件

用文件流打开文件可以利用无参的构造函数，然后调用 open():

```
ofstream outfile;
outfile.open("outfile", iosmode);
```

也可调用带参数的构造函数，指定文件名和打开方式：

ofstream outfile("outfile", iosmode);

文件打开方式由下面几种：

表 12-7

ios::in	打开一个输入文件，用这个标志作为 ifstream 的打开方式，以防止截断一个现成的文件
ios::out	打开一个输出文件，当用于一个没有 ios::app、ios::ate 或 ios::in 的 ofstream 时，ios::trunc 被隐含
ios::app	以追加的方式打开一个输出文件
ios::ate	打开一现成文件（不论是输入还是输出）并寻找末尾
ios::nocreate	仅打开一个存在的文件（否则失败）
ios::noreplace	仅打开一个不存在的文件（否则失败）
ios::trunc	如果一个文件存在，打开它并删除旧的文件
ios::binary	打开一个二进制文件，缺省的是文本文件

这些标志可以通过逐位或（|）运算来连接。

ios 类中定义的文件打开方式：

程序 12-15

```
// stream operation mode
enum open_mode {
   in = 0x01, // open for reading
   out = 0x02, // open for writing
   ate = 0x04, // seek to eof upon original open
   app = 0x08, // append mode: all additions at eof
   trunc = 0x10, // truncate file if already exists
   nocreate = 0x20, // open fails if file doesn't exist
   noreplace= 0x40, // open fails if file already exists
   binary = 0x80 // binary (not text) file
};
```

它们可以用逐位或（|）运算符来连接。例如，下面的语句用来打开一个文件用于输出，并确保它以前是不存在的：

ofstream outfile("outfile", ios::out|ios::noreplace);

当文件打开失败时，文件流对象的 bad 标志会置位。可以使用 ios 类成员函数 clear 清除标志位，然后重新操作。

二、文件操作

由于 iostream 的设备无关性，构造了文件流以后，就可以象前面标准输入输出流的方法一样使用了。

三、关闭文件

在文件操作结束时，可以用 close()成员函数关闭该文件。

Outfile.close();

不过，在该文件流对象生存期结束时，对象也会自动调用析构函数来关闭文件。最好在文件操作结束时，关闭文件，这样会使程序的可读性更好。

程序 12-16

```
outfile<<"hello world"<<name<<endl;
infile>>pi>>a>>complex;
outfile.write(buf,80);
infile.getline(buf,80);
```

需要注意的是：文件操作不能超过该文件打开方式设定的权限。

如果查看 fstream.h，可以会发现：类 ifstream 和 ofstream 仅仅定义了一些必要的构造函数、open()和 close()。这说明它们继承了 istream 和 ostream 负责具体操作的成员函数。也就是说，文件操作和前面讲的标准流操作是完全一样的。但是，如果文件操作超过该文件打开方式设定的权限，比如试图向一个输出文件流中输入数据，那么该文件流的 bad 标志会置位。

字节流可直接与内存而不是与文件或标准输出一起工作。我们可以用与标准输出同样的格式，操作内存里的数据（字节）。如果我们想把数据放入字节流，可以建立一个 ostrstream 对象；如果想从字节流中提取数据，就建立一个 istrstream 对象。

12.3.1 输入流

istrstream 类支持一个字符数组作为源的输入流。在构造 istrstream 对象前，必须存在一个字符数组，而且这个数组中已经填充了我们想要提取的字符。下面是两个构造函数的原型：

istrstream::istrstream(char* buf);

istrstream::istrstream(char* buf, int size);

第一个构造函数取一个指向以"\0"作为结尾符的字符数组的指针,我们可以提取字节直至遇到"\0"为止。第二个构造函数还需要这个数组的大小，但不需要数组包含字符串的结尾符"\0"，我们可以一直提取字节到 buf[size-1]，而不管是否遇到"\0"。

程序 12-17

```
#include "strstrea.h"
//使用 strstream 要包含头文件 strstrea.h
//这个文件名很奇怪，这是因为 DOS 文件命名规则的限制。

void main()
{
   int n;
   float f;
   char buf[80];
   istrstream arg("3.14159  hello");
   arg>>n>>f>>buf;
   cout<<"n="<<n<<endl;
```

```
    cout<<"f="<<f<<endl;
    cout<<"buf="<<buf<<endl;
}
```

程序的运行结果是：

n=3

f=0.14159

buf=hello

在程序中变量 n 提取到的是 3，变量 f 提取到的是 0.14159，这说明抽取不仅依赖于空白的分隔符，还依赖于提取的数据类型。同时，程序中也会出现前面提到的问题。例如：

istrstream arg("3.14159 hello world!");

的运行结果并不是想象中那样，把"hello world!"被抽取到 buf 中，仅仅是 hello 被抽取。

如果该输入流定义为：

istrstream arg("3.14159 hello",10);

强制限定字符数组元素个数为 10，则程序运行结果为：

n=3

f=0.14159

buf=he

12.3.2 输出流

ostrstream 类支持一个字符数组作为数据传输目的地的输出流，它可以使用我们为它申请的存储空间，这时字节在内存中被格式化；也可以使用自动分配的存储空间。

我们为 ostrstream 申请存储空间的方法是通过 ostrstream 有参的构造函数：

ostrstream(char*, int, int=ios::out);

第一个参数是缓冲区的指针，第二个参数是缓冲区的大小，第三个参数是打开模式。如果是缺省的模式，则从缓冲区头部开始添加新的字符；如果打开模式是 ios::ate 或 ios::app，则从缓冲区中的字符串的结尾符处开始添加新的字符 (结尾符不后移，只是被简单地覆盖，下面程序中 os<<ends 的作用就是在 buf 后面加上结尾符)。

下面是一个使用 istrstream 的例子：

程序 12-18

```
#include "strstrea.h"
void main()
{
   int n;
   float f;
   char buf[80];
   ostrstream os(buf,80);
   cin>>n>>f;
   os<<"n="<<n<<endl;
   os<<"f="<<f<<endl;
```

```
    os<<ends;
    cout<<buf;
}
```

程序的运行结果是：

n=3

f=0.14159

如果使用自动分配存储空间的方法，则使用无参的构造函数：

ostrstream os;

这时对象 os 在堆中分配一块存储空间，并自己维护。当用户输入不断增加，原存储块不够用时，它可以再分配更多的存储空间。用户可以通过调用成员函数 rdbuf()返回缓冲区指针，以访问缓冲区的内容。

ostrstream 类支持一个字符数组作为数据传输目的地的输出流，它可以使用我们为它申请的存储空间，这时字节在内存中被格式化；也可以使用自动分配的存储空间。

我们为 ostrstream 申请存储空间的方法是通过 ostrstream 有参的构造函数：

ostrstream(char*, int, int=ios::out);

当插入操作超过申请的存储空间的小时，流的 bad 标志会置位。

如果使用自动分配存储空间的方法，则使用无参的构造函数：

ostrstream os;

当不知道数据需要多少空间时，这是一种很好的方法。这时对象 os 在堆中分配一块存储空间，并自己维护。当用户输入不断增加，原存储块不够用时，如果有必要，它将移动存储块以分配更多的存储空间。用户可以通过调用成员函数 rdbuf()返回缓冲区指针，以访问缓冲区的内容。但是有一个问题：当空间不够，流对象自己移动存储块以分配更多空间时，原来的缓冲区指针已经失效，但是用户是不知道的，因为流对象自己负责维护存储空间。ostrstream 处理这个问题的方法是"冻结"自己。用户通过 str()函数返回指向输出流内容的字符数组的指针，并将自己"冻结"。用户不能再它添加字符，ostrstream 对象也不再负责存储空间的自动释放，必须由用户自己清理存储器：

delete os.str();

如果想添加更多的字符，一般的做法是新建一个 ostrstream，将旧的流灌入新流中，并向新流添加字符。

12.4.1 流状态测试

在 iostream 中，每一个流对象都有一个表示操作是否成功的状态位。

每一步操作后流的状态有下面五种：

表 12-8

good	流状态正常。
end-of-file	表明输入操作到达输入序列尾部。
fail	表明出现了格式化问题或者不影响缓冲区的其他问题，如果 fail 位被清除，流还是可用的。
bad	表明缓冲区出现错误，数据丢失。
hardfail	出现不可恢复性错误。

iostream 提供了一些成员函数，如 good()、fail()等来查询当前流的状态，下表列出了所有用于状态查询的成员函数。

表 12-9

int rdstate()	返回当前状态。
int good()	若未设置错误状态时返回非零值。
int eof()	当 end-of-file 位设置时，返回非零值。通常在执行过程中遇到文件的末尾时设置。不能从缓冲区读入更多的字符，也不能向缓冲区输出更多的字符。
int fail()	当 fail、bad 或 hardfail 位设置时，返回非零值。
int bad()	当 bad 或 hardfail 位设置时，返回非零值。
int operator!()	当流状态非正常时，返回非零值。

在对流进行操作时，我们应该先对流的状态进行检测，以确保流的状态正常，通常的做法是：

```
if (!(cout << "Hello World !"))
handle_error();
```

使用插入或者抽取运算符的优点就是用户可以把插入或者抽取操作成组进行。例如，

```
int n=5;
cout<<"n="<<n;
if (!cout) handle_error();
```

这样可以使程序简洁明了，但是随之会产生一个问题：用户不可能在每次流操作结束后检测流的状态。C++的例外可以解决这个问题，因此标准 iostream 允许使用例外处理流错误。

在 iostream 中，每一个流对象都有一个表示操作是否成功的状态变量。对于每一步流操作，成员函数根据当前程序执行的状况为状态变量赋值，并设定该步操作后流的状态。

ios 类中定义的流状态位：

程序 12-19

```
// stream status bits
enum io_state {
   goodbit = 0x00, // no bit set: all is ok
   eofbit = 0x01, // at end of file
   failbit = 0x02, // last I/O operation failed
   badbit = 0x04, // invalid operation attempted
   hardfail = 0x80 // unrecoverable error
};
```

下面是用于查询的成员函数定义：

```
inline int _Cdecl ios::rdstate() { return state; }
inline int _Cdecl ios::eof() { return state & eofbit; }
```

inline int _Cdecl ios::fail() { return state & (failbit | badbit | hardfail); }

inline int _Cdecl ios::bad() { return state & (badbit | hardfail); }

inline int _Cdecl ios::good() { return state == 0; }

如果我们查询到流的状态出现了不正常的情况，就可以使用 iostream 提供的成员函数来清除当前的错误状态，使流成为可用的。这种做法是有意义的，当 fail（）函数返回 1，但是 bad（）函数返回的是 0。也就是说，上一步流操作可能出现了格式化问题，如果把错误标志清除，流还有可能是可用的。

在 iostream 类库中，ios 类提供了一个成员函数：

void ios::clear(int nState = 0);

如果参数为 0，那么该函数的作用是清除所有错误标志，否则参数可以设置为 goodbit、eofbit、failbit、badbit 中的一种或者它们的组合，这时该函数的作用是将所有的标志清除，然后将参数指定的标志置位。

我们举一个例子：

程序 12-20

```
#include "iostream.h"
void main()
{
   int i;
   cout<<"输入整数："<<endl;
   cin>>i;
   if (!cin){
      cout<<"eof:"<<cin.eof()<<endl;
      cout<<"fail:"<<cin.fail()<<endl;
      cout<<"bad:"<<cin.bad()<<endl;
   }else{
      cout<<"correct."<<endl;
   }
}
```

如果输入的是整数，那么将输出：correct。否则，程序输出：

eof:0

fail:2

bad:0

如果流状态为 fail，表示出现格式化错误，流变为不可用。我们可以用 clear( )函数，清除错误标志，使流变为可用：

程序 12-21

```
#include "iostream.h"
void main()
{
   ofstream out("out.txt",ios::nocreate);
```

```
    if (!out){
        cout<<"eof:"<<out.eof()<<endl;
        cout<<"fail:"<<out.fail()<<endl;
        cout<<"bad:"<<out.bad()<<endl;
    }else{
        cout<<"correct."<<endl;
    }
}
```

该程序用来打开一个已存在的文件 out.txt，如果 out.txt 确实在当前目录下已存在，则打开成功，程序输出：correct。否则，程序输出：

eof:0

fail:2

bad:0

表明文件打开失败，可以调用 clear( )函数清除错误标志，然后重新调用正确的文件：

```
out.clear();
out.open("out1.txt",ios::nocreate);
```

12.4.2 例外处理流错误

I/O 流库中允许使用例外对流错误进行处理。在 ios 类中添加了两个成员函数：

```
void exceptions(iostate except_mask);
iostate exceptions();
```

第一个函数用来设置标志位，是流抛出相应的例外。这些标志位可以是 eof、bad、fail 或者它们的组合，第二个函数用来返回当前的标志。

下面的例子为我们演示了如何使用例外处理流错误：

程序 12-22

```
#include "iostream.h"
void main()
{
    int a=5;

    try
    {
        cout.exceptions(ios::failbit);
        cout<<"a="<<a<<endl;
    }
    catch(ios_base::failure&  excep)
    {
        cerr<<excep.what()<<endl;
    }
}
```

在 try 模块里 cout 调用 exceptions()抛出一个例外，使 catch 模块里的程序能够执行，这一步仅仅是为了测试之用。

例外处理是 C＋＋语言的一个新特征，它提出了出错处理更完美的方法，并使出错处理程序和主代码分离，从而简化了出错处理程序的编写。例外处理的一般格式：

```
try {
   //可能会生成例外的代码段
} catch (typeid1) {
//例外处理程序
}
```

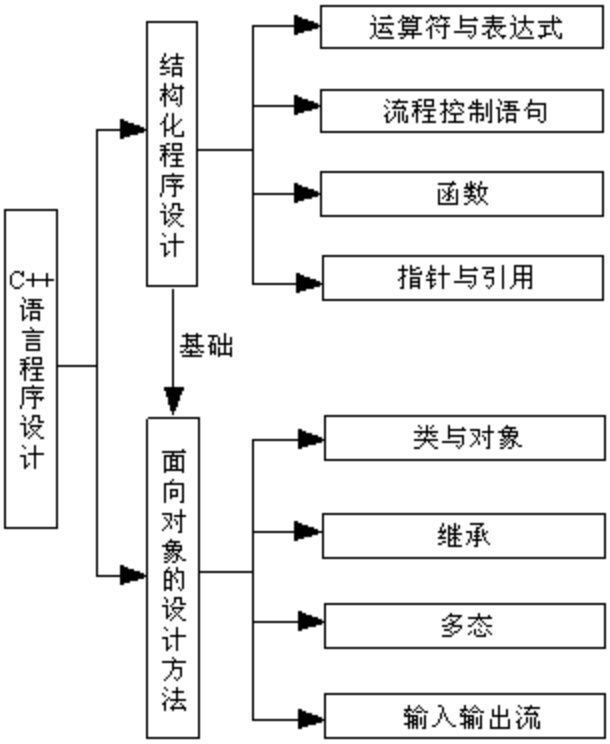
关键字 try 引导的程序段为测试程序段，在测试程序段中生成的例外由 catch 引导的例外处理器捕获并进行处理，catch 带的参数是要捕获的例外类型。

当例外抛出时，如果被捕获，主程序段立即中止执行，转到例外处理程序中执行例外处理，处理结束并不返回到异常抛出的地方，通常的作法是报告错误，进行一些保护性工作，如关闭文件，然后退出函数段。

例外处理的内容很多，详细了解请参考其它书籍。

【本章小结】

这一章中，我们对输入输出流类库进行了一个简略的介绍。从功能上讲，整个输入输出流类库可以分为输入流和输出流两部分。从操作的设备来看，可以分为标准输入输出流（面向控制台）、文件流（面向文件）和字节流（面向存储器）。我们可以通过操纵算子来对流进行格式化，将输入、输出流变成我们需要的格式。输入输出流类库也提供了错误检测机制，使用该库提供的状态测试函数，我们可以在每一步操作结束后，查询当前流的状态。I/O 流库中，使用 try、catch 和 throw 语句对流错误进行处理，它使程序中错误的检测简单化，并提高程序处理错误的能力。


C++语言程序设计
结构化程序设计
面向对象的设计方法
基础
运算符与表达式
流程控制语句
函数
指针与引用
类与对象
继承
多态
输入输出流