第 3 章　C++程序的结构

教学要求

- 掌握：局部变量、全局变量和外部变量在多文件程序中的联络作用，以及实现数据共享的方法和共享数据的保护。
- 理解：作用域、可见性与生存期的概念。
- 了解：编译预处理。

3.1　作用域和可见性

3.1.1　作用域

作用域是标识符在程序中有效的范围，标识符的引入与声明有关，作用域开始于标识符的声明处。C++的作用域范围分为：块作用域（局部作用域）、函数原型作用域、函数作用域、类作用域和文件作用域。其中类作用域将在第 4 章介绍，下面介绍其他几种。

1．块作用域

当标识符的声明出现在由一对大括号括起来的一段程序（块）内时，该标识符的作用域从声明点开始，到块结束的大括号处为止，该作用域的范围具有局部性。

例如，下面的代码描述了块作用域：

```
void fun(int a)          // a 的作用域起始处
{
    int b=1;             // b 的作用域起始处
    if(a>b)
    {
        int c;           // c 的作用域起始处
        c=a-b;
        cout<<c;
        ...
    }                    // c 的作用域终止处
}                        // a 和 b 的作用域终止处
```

此段代码中，声明的标识符 a、b 和 c 处在不同的块中。其中标识符 a 和 b 是在整个函数的函数体的块中，而 c 是在 if 语句块中声明的，故它的作用域是从声明处开始到 if 语句结束处终止。

2．函数原型作用域

函数原型作用域指的是声明函数原型所指定的参数标识符的作用范围。这个作用范围是在函数原型声明中的左、右括号之间。正因为如此，在函数原型中声明的标识符可以与函数定义中说明的标识符名称不同。由于所声明的标识符与该函数的定义及调用无关，所以可以在函数声明中只做参数的类型声明而省略参数名。例如：

```
double  max(double x, double y);
```

或

```
double  max(double, double);
```

是等价的。不过，考虑到程序的可读性，还是要在声明函数原型时给形参声明有意义的标识符，并且和函数定义时的参数名相同。

3. 函数作用域

在 C++语言中，只有 goto 语句中的标号标识符具有函数作用域。但 goto 语句的滥用会导致程序流程无规则、可读性差。因此现代程序设计方法不推荐使用 goto 语句。

4. 文件作用域

文件作用域是在所有函数定义之外声明的，其作用域从声明之处开始，直到文件结束。

3.1.2　可见性

可见性是从对标识符引用的角度而谈的概念。如果标识符在某处可见，则就可以在该处引用此标识符。程序运行到某一点能够引用到的标识符就是该处可见的标识符。作用域指的是标识符有效的范围，而可见性是分析在某一位置标识符的有效性。图 3-1 描述了作用域的一般关系。可见性表示从内层作用域向外层作用域“看”时能看到什么。

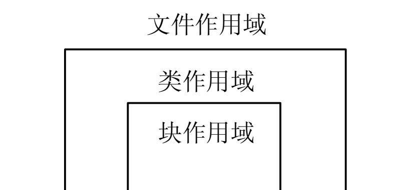

图 3-1　作用域关系图

作用域可见性的一般规则是：

（1）标识符应先声明，后引用。

（2）在同一作用域中，不能声明同名的标识符。

（3）在没有互相包含关系的不同的作用域中声明的同名标识符互不影响。

（4）对于两个嵌套的作用域，如果在内层作用域中声明了与外层作用域中同名的标识符，则外层作用域的标识符在内层不可见。

【例 3-1】作用域与可见性实例

```
//Ch03_01.cpp
#include <iostream>
using namespace std;
int i;                          //变量 i 具有文件作用域
void main()
{
    i=5;                        //文件作用域的 i 赋初值
    {                           //子块 1
        int i;                  //变量 i 具有块作用域
        i=7;
        cout<<"i="<<i<<endl;    //输出 7
    }
    cout<<"i="<<i<<endl;        //输出 5
}
```

这是一个文件作用域与块作用域相互包含的实例。在这个例子中，主函数之前声明的变量 i 具有文件作用域，它的有效作用范围是整个源代码文件。在主函数开始处给这个具有文件作用域的变量 i 赋初值 5，接下来在子块 1 中又声明同名变量并赋初值 7。在主函数内子块 1 之前，可以引用到具有文件作用域的变量 i，这时它是可见的。当程序允许进入子块 1，就只能引用到具有块作用域的同名变量，具有文件作用域的同名变量不可见，这就是所谓的内层屏蔽，有时也称为同名覆盖，即内层的变量覆盖了外层的同名变量。所以第一次输出的结果是 7，这是因为具有块作用域的变量把具有文件作用域的变量屏蔽掉了，也就是具有文件作用域的变量变得不可见。当程序运行到子块 1 结束后，进行第 2 次输出时，输出的就是具有文件作用域的变量的值 5。

3.2 生存期

变量从诞生到结束的这段时间就是它的生存期。在生存期内，变量将保持它的值不变，直到它们被更新为止。对象的生存期可以分为静态生存期和动态生存期。

3.2.1 静态生存期

静态生存期与程序的运行期相同，只要程序一开始运行，这种生命期的变量就存在，当程序结束时，其生命期就结束。具有静态生命期的变量在定义时就分配固定的存储单元，并一直保持不变，直至整个程序结束。在文件作用域中声明的变量都是具有静态生存期的。如果要在函数内部的块作用域中声明具有静态生存期的变量，则要使用关键字 static。例如下列语句声明的变量 i 便是具有静态生存期的变量，也称为静态变量。

```
static  int  i;
```

静态生命期的变量，若无显式初始化，则自动初始化为 0。

3.2.2 动态生存期

在块作用域中声明的变量具有动态生存期，也可称为局部生存期变量。这种变量的生存期开始于程序执行经过其声明点时，结束于其作用域结束处。具有动态生存期的变量是在程序执行过程中，使用时才分配存储单元，使用完毕就立即释放。

【例 3-2】变量的生存期与可见性

```
//Ch03_02.cpp
#include <iostream>
using namespace std;
int i=1;                // i 具有文件作用域和静态生存期
void main(void)
{
   static int a;        // a 具有块作用域和静态生存期，局部可见
   int b=-10;           // b, c 具有块作用域和动态生存期
   int c=0;
   void other(void);
   cout<<"---MAIN---\n";
   cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
   c=c+8;
   other();
```

```
    cout<<"---MAIN---\n";
    cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
    i=i+10;
    other();
}
void other(void)
{
    static int a=2;
    static int b;
    // a,b 具有块作用域和静态生存期，局部可见
    // 只在第一次进入函数时被初始化
    int c=10;             // c 具有块作用域和动态生存期
                          // 每次进入函数时都初始化
    a=a+2; i=i+32; c=c+5;
    cout<<"---OTHER---\n";
    cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
    b=a;
}
```

程序运行结果为：

```
---MAIN---
i: 1  a: 0  b: -10  c: 0
---OTHER---
i: 33  a: 4  b: 0  c: 15
---MAIN---
i: 33  a: 0  b: -10  c: 8
---OTHER---
i: 75  a: 6  b: 4  c: 15
```

3.3　变量的存储类型

存储类型是针对变量而言的，它规定了变量的生存期。编译系统往往根据其存储方式定义、分配和释放相应的内存空间。变量的存储类型反映了变量占用内存空间的期限。

在 C++中，变量有 4 种存储类型：自动类型（auto）、静态类型（static）、外部类型（extern）和寄存器类型（register）。

自动类型变量和寄存器类型变量属于动态存储，静态类型变量和外部类型变量属于静态存储。在知道变量生存期的性质后，变量的说明就可以完整地表达为：

<存储类型说明符><数据类型说明符><变量名表>;

例如：

```
static  int  x, y;       // 定义 x，y 为静态整型变量
auto  char  c1, c2;      // 定义 c1，c2 为自动字符变量
extern  float  a, b;     // 定义 a，b 为外部实型变量
```

变量的存储类型不同，其作用域也不同。C++中的变量按其作用范围可以分为局部变量和全局变量。

3.3.1　局部变量

具有块作用域的变量称为局部变量。如果把数据存储在局部变量中，在不同函数体内的

局部变量是互相不可见的，因此函数在不同的块之间只能通过参数传递共享数据。

一般说来，用自动存储类型（auto）声明的变量都是限制在某个程序范围内使用，即为局部变量。从系统角度来说，自动存储类型变量是采用堆栈方式分配内存空间。因此，当程序执行到超出该变量的作用域时，就释放它所占用的内存空间，其值也随之消失。若自动存储类型的变量是在函数内或语句块中声明的，则可省略关键字 auto。

使用关键字 register 声明寄存器类型的变量的目的是将所声明的变量放入寄存器内，从而加快程序的运行速度。寄存器变量和自动变量具有相同的性质，都属于动态存储方式。只有局部自动变量和形式参数可以声明为寄存器变量，需要采用静态存储方式的变量不能定义为寄存器变量。

静态类型（static）变量也是一种局部变量。它和自动存储类型变量的最大不同之处在于：静态存储类型变量在内存中是以固定地址存放的，而不是以堆栈方式存放的。因此，只要程序还在继续执行，静态存储类型变量的值就一直有效，不会随它所在的函数或语句块的结束而消失。

在 C++语言中，声明局部变量的时候加上关键字 static 就构成了静态局部变量。

【例 3-3】使用静态存储类型的局部变量

```
//Ch03_03.cpp
#include <iostream>
using namespace std;
int fun(int x)
{
    static int a=3;     // 静态类型的局部变量
    a+=x;
    return a;
}
void main()
{
    int i=2,j=1,n;
    n=fun(i);
    n=fun(j);
    cout<<n<<endl;
}
```

程序运行结果为：

```
6
```

在这个程序中，函数 fun 中定义了静态局部变量 a，因为静态变量的函数调用结束后，存储单元不释放，值具有继承性，也就是说下次调用该函数时，该静态变量 a 的初值就是上一次调用结束时变量的值。程序中第一次调用 fun 函数时，a 的值为 5，第二次调用时，a 继承了上一次的结果 5，然后加上 1 得到 6，所以最后的输出为 6。

3.3.2 全局变量

具有文件作用域的变量称为全局变量。在整个程序中，除了在定义有同名局部变量的块之外，都可以直接访问。将数据存放在全局变量中，不同的函数在不同的地方对同一全局变量进行访问，就实现了这些函数之间的数据共享。

在程序中声明的全局变量总是静态存储类型，若在全局变量前加上 static，使该变量只在这个源程序文件内使用，称为全局静态存储变量或静态存储全局变量。若一个程序由一个文件组成，在声明全局变量时，有无 static 并没有区别，但若多个文件组成一个程序时，加与不加

static，其作用完全不同。例如，有一个名为 StaticScope 的项目，包含两个源文件，一个是 StaticScope_1.cpp，另一个是 StaticScope_2.cpp。

【例 3-4】使用静态存储类型的全局变量

```
// StaticScope_1.cpp
#include <iostream>
using namespace std;
int n;
void add();
void main()
{
    n=20;
    cout<<"StaticScope_1.cpp : n= "<<n<<endl;
    add();
}
// StaticScope_2.cpp
#include <iostream>
using namespace std;
static int n;
void add()
{
    n++;
    cout<<"StaticScope_2.cpp : n= "<<n<<endl;
}
```

执行项目 StaticScope，运行结果如下：

```
StaticScope_1.cpp : n= 20
StaticScope_2.cpp : n= 1
```

由此可见，文件 StaticScope_2.cpp 中函数 add 输出 1 而不是 21，表示两个变量互不相干。因此，静态全局变量对组成该程序的其他源文件是无效的，它能很好地解决在程序多文件组织结构中全局变量的重名问题（关于多文件组织结构会在 3.4.2 中介绍）。

同静态全局变量相类似，静态函数也是在某个函数声明前加上 static，它的目的也是使该函数只在声明的源文件中使用，对于其他源文件则无效。

3.3.3　外部变量

使用关键字 extern 声明的变量称为外部变量。一般是指定义在本程序外部的变量。外部变量在程序中只能被定义一次，但在不同的地方可以被多次声明为外部变量。当某个变量被声明成外部变量时，不必再次为它分配内存就可以在本程序中引用这个变量。例如：

```
extern  int  a;
```

说明整型变量 a 在其他源程序文件中已经定义为全局变量，在本程序文件中被声明是外部存储类型的，因而本程序文件可以引用。

extern 定义符的作用是将全局变量的作用域延伸到其他源程序文件。在 C++中，只有在两种情况下要使用外部变量。第一种情况是在同一个源程序文件中，若定义的变量使用在前，声明在后，这时在使用前要声明为外部变量。

【例 3-5】同一个源文件中的外部变量使用

```
//Ch03_05.cpp
#include <iostream>
using namespace std;
```

```
extern int a;        // 声明外部变量 a，若没有此语句，函数 count 中的语句将出错
void count()
{
    a++;
    cout<<a<<endl;
}
int a=10;            // 外部变量 a 的实际定义处
void main()
{
    count();
    cout<<a<<endl;
}
```

程序运行结果为：

```
11
11
```

第二种情况是当由多个源文件组成一个完整的程序时，在一个源程序文件中定义的变量要被其他若干个源文件引用时，引用的文件中要使用 extern 声明外部变量。例如，有一个名为 ExternFile 的项目由两个源文件 ExternFile_1.cpp 和 ExternFile_2.cpp 构成，文件 ExternFile_1.cpp 定义主函数和一个全局变量 n，文件 ExternFile_2.cpp 定义函数 fact( )，并声明变量 n 为外部变量。

【例 3-6】不在同一个源文件中的外部变量使用

```
//ExternFile_1.cpp
#include <iostream>
using namespace std;
int n;
void fact();
void main()
{
    n=20;
    cout<<"n= "<<n<<endl;
    fact();
}
//ExternFile_2.cpp
#include <iostream>
using namespace std;
extern int n;        // 声明外部变量 a
void fact()
{
    n++;
    cout<<"n= "<<n<<endl;
}
```

执行该项目，运行结果如下：

```
n= 20
n= 21
```

由于默认的函数声明或定义总是 extern 的，因此上述代码中的函数 fact()的声明和定义可以在不同的源文件中进行。

需要说明的是，虽然外部变量在不同源文件中或函数之间的数据传递有用，但是也应该看到，这种能被许多函数共享的外部变量，其数据值的任何一次改变，都将影响到所有引用此

变量的函数的执行结果，其危险性是显然的。

最后我们将变量的存储类型从作用域和生存期的角度归纳为表 3-1（注：表中“ √ ”表示“是”，“×”表示“否”）。

表 3-1　不同变量的作用域和生存期

变量存储类型		定义语句块内		定义语句块外		文件外	
		作用域	生存期	作用域	生存期	作用域	生存期
局部变量	auto	√	√	×	×	×	×
	register	√	√	×	×	×	×
	static	√	√	×	√	×	√
全局变量	static	√	√	√	√	×	×
	extern	√	√	√	√	√	√

3.4　编译预处理和多文件结构

3.4.1　编译预处理命令

预处理程序又称预处理器，它包含在编译器中。编译器在对源程序进行编译之前，首先要由预处理程序对程序文本进行预处理，把源代码转化成包含机器语言指令的目标文件。预处理程序提供了一组编译处理指令，这些指令称为预处理指令。预处理指令实际上不是 C++语言的一部分，它是用来扩充 C++程序设计的环境。所有的预处理指令在程序中都是以“#”来引导，每一条预处理指令单独占用一行，不要用分号结束。预处理指令可以根据需要出现在程序中的任何位置。

1．*宏定义指令*#define *和*#undef

在程序中用#define 定义一个符号常量，例如：

```
#define  PI  3.1415926
```

其中，#define 是宏定义指令，其作用是将 3.1415926 由 PI 代替，PI 称为宏名。在 C++中更常用的方法是在类型说明语句中用 const 进行修饰。

#define 还可以定义带参数的宏，以实现简单的函数计算，提高程序的运行效率。例如：

```
#difine  MIN(x,y)  ((x)<(y)?(x):(y))
```

则语句：c=MIN(3+8,7+6);

将被替换为语句：c= ((3+8)<(7+6)?(3+8):(7+6));

但是，在 C++中这一功能已经被内联函数取代。

#undef 的作用是删除由#define 定义的宏，使其不再起作用。

2．*文件包含指令*#include

所谓“文件包含”是指将另一个源文件嵌入到当前源文件中该点处。所包含的文件一般是以.h 为扩展名，因而称它为“头文件”。通常使用#include 指令嵌入头文件。文件包含指令有两种格式：

① #include <文件名>

这种格式用于嵌入 C++提供的头文件。这些头文件一般位于 C++系统目录的 include 子目

录下。C++预处理器遇到该命令后，就到 include 子目录下搜索对应的文件，并将其嵌入到当前文件中。这种方式是标准方式。

例如，利用 include 语句将定义 iostream 的 iostream.h 头文件载入：

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

上述的文件包含指令包含一个旧的 iostream 库头文件，在编译时将自动链接一个旧的 iostream 库。而在前面的例题中我们采用的是新的头文件载入方式，其中引入了 namespace 的概念，在 ANSI/ISO 的 C++标准里定义了一个名为 std 的 namespace，并将许多类定义在这个 namespace 里。以往的头文件定义方式虽然仍然可以使用，但是当要使用 ANSI/ISO C++标准中定义的类库时（std 类库），就不得不使用新的头文件载入方式。例如，以下文件包含指令包含一个标准 C++库头文件，在编译时将自动链接一个标准 C++库：

```
#include <iostream>
using namespace std;
```

② #include "文件名"

预处理器遇到这种格式的包含指令后，首先在当前文件所在目录中进行搜索，如果找不到，再按标准方式搜索。这种方式适合于指定用户自定义的头文件。

include 文件可以嵌套，即在头文件中还可以有包含指令。

3．条件编译指令

一般情况下，源程序中所有的语句都参加编译，但有时也希望根据一定的条件去编译源文件的不同部分，这就是“条件编译”。条件编译使得同一源程序在不同的编译条件下得到不同的目标代码。

C++提供的条件编译语句有下列几种常用的形式：

① # if 形式

```
#if  <表达式>
        <程序段 1>          // 当表达式值为真（非零）时编译程序段 1
[#else
        <程序段 2>]         // 当表达式值为假（零）时编译程序段 2
#endif
```

② #ifdef 形式或#ifndef 形式

```
#ifdef（或#ifndef）<标识符>
        <程序段 1>
 [#else
        <程序段 2>]
#endif
```

预处理程序扫描到#ifdef（或#ifndef）时，判别其后面的<标识符>是否被定义过（一般用#define 命令定义），从而选择对哪个程序段进行编译。对#ifdef 格式而言，若<标识符>在编译命令行中已被定义，则条件为真，编译<程序段 1>；否则，条件为假，编译<程序段 2>。而#ifndef 的检测条件与#ifdef 恰好相反，若<标识符>没有被定义，则条件为真，编译<程序段 1>；否则，条件为假，编译<程序段 2>。#else 部分可以省略，若被省略，且<标识符>在编译命令行中没有被定义时（针对#ifdef 形式），就没有语句被编译。

③ #if-#elif 形式

```
#if <表达式 1>
```

```
        <程序段 1>          // 当表达式 1 值为真（非零）时编译程序段 1
    [#elif <表达式 2>
        <程序段 2>          // 当表达式 1 值为假、表达式 2 值为真时编译程序段 2
            …
    [#elif <表达式 n>
        <程序段 n>]
    // 当表达式 1、…表达式 n-1 值均为假，表达式 n 值为真时编译程序段 n
    [#else
        <程序段 n+1>]       // 其他情况下编译程序段 n-1
    #endif
```

【例 3-7】利用条件编译命令完成大、小写字母互换

```
//Ch03_07.cpp
#include <iostream>
using namespace std;
#define TYPE 1
void main()
{
    char  s[20];
    int  i=0;
    cout<<"Enter String: ";
    cin>>s;
    while(s[i]!='\0')
    {
        #ifdef  TYPE
           if(s[i]>='a'&&s[i]<='z')
             s[i]=s[i]-32;
        #else
           if(s[i]>='A'&&s[i]<='Z')
              s[i]=s[i]+32;
        #endif
        cout<<s[i];
        i++;
    }
}
```

程序运行结果为：

```
Enter String: aBcD
ABCD
```

从输出结果可以看到，程序中定义了#define TYPE 1（或#define TYPE），由于条件编译#ifdef TYPE……#else……#endif 的作用，使输入字符串中的小写字母变为大写字母；如果去掉#define TYPE 1 命令行，输出结果为小写字母 abcd，即输入字符串中的大写字母变为小写字母，原因是由于条件编译#ifdef TYPE……#else……#endif 中#else 的作用。

3.4.2 多文件组织结构

至此，已经学习了很多完整的 C++源程序实例，由于我们所举的例子都比较小，所以将所有的程序代码都写在同一个源程序文件中。在实际的程序设计中，一个程序通常由多个头文件和源文件组成，每个源文件是一个可编译的程序单位，头文件起着源文件之间接口的作用。

我们可以从下面的程序开发示意图来看看多文件的组织结构，如图 3-2 所示。

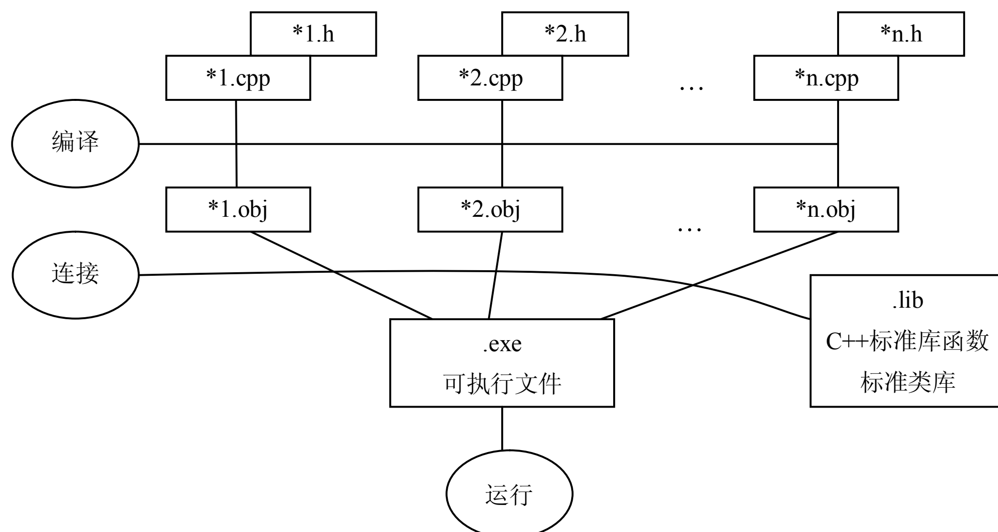

图 3-2　C++程序开发示意图

图中，源文件中含有包含头文件的预编译语句，经过预编译后，产生翻译单元，然后编译，进行语法检查，产生目标文件（.obj）。若干个目标文件经过连接，产生可执行文件（.exe）。连接包括 C++库函数的连接和标准类库的连接。

采用这样的文件组织结构，可以将程序按逻辑功能划分，分解成各个源文件，把相关函数放到特定源文件中，使程序更加容易管理。由于编译器总是以文件为单位工作的，可以对不同的文件进行单独编写、编译，最后再连接。如果修改了源文件中的任何部分，只需再次启动编译器对其中被修改的文件重新编译一次，这样的程序便于调试和维护。

例如，下面的程序由三个源文件组成：

```
mycircle.cpp
myrect.cpp
myarea.cpp
```

上述三个源文件中，都包含了自己定义的头文件 myarea.h。三个源文件分别为：调用不同功能计算面积，定义计算圆面积和定义计算矩形面积的函数。

【例 3-8】多文件组织结构。

```
// 头文件myarea.h: 声明计算圆面积和矩形面积的函数
double circle(double radius);
double rect(double width,double length);

// 源文件mycircle.cpp: 定义计算圆面积的函数
#include "myarea.h"
const double pi=3.14;
double circle(double radius)
{
    return pi*radius*radius;
}

// 源文件myrect.cpp: 定义计算矩形面积的函数
#include "myarea.h"
```

```
double rect(double width,double length)
{
    return width*length;
}

// 源程序 myarea.cpp: 调用不同功能计算面积的主程序
#include "myarea.h"
#include <iostream>
using namespace std;
void main()
{
    double width,length;
    cout<<"please input the width and length of a rectangle: ";
    cin>>width>>length;
    cout<<"area of rectangle is "<<rect(width,length)<<endl;

    double radius;
    cout<<"please input the radius of a circle: ";
    cin>>radius;
    cout<<"area of circle is "<<circle(radius)<<endl;
}
```

程序运行结果为：

```
please input the width and length of a rectangle: 6.5  3.5 ↙
area of rectangle is 22.75
please input the radius of a circle: 5 ↙
area of circle is 78.5
```

程序中的三个源程序都含有 myarea.h 的头文件，这样可以使信息共用，保证程序的一致性。要运行该程序，首先分别编译头文件 myarea.h 和其他 C++源文件，再进行编译和连接，产生一个名为 myarea.exe 的可执行文件，运行就会得到应有的结果。

项目设计 2　预处理及多文件结构

1．设计题目

预处理及多文件结构。

2．设计概要

要求掌握多文件 C++程序的编码方法和预处理的使用。

3．系统分析

每个 C++程序通常分为两个文件。一个文件用于保存程序的声明，称为头文件。另一个文件用于保存程序的实现，称为定义文件。C++程序的头文件以“.h”为后缀，C++程序的定义文件通常以“.cpp”为后缀。头文件通常由三部分内容组成：头文件开头处的版权和版本声明、预处理块和函数与类结构声明等。

书写头文件时应该遵循的几点规则：

① 为了防止头文件被重复引用，应当用 ifndef/define/endif 结构产生预处理块。

② 采用#include <filename.h>格式引用标准库的头文件（编译器将从标准库目录开始搜索）。

③ 采用#include"filename.h"格式引用非标准库的头文件（编译器将从用户的工作目录开始搜索）。

④ 头文件中只存放“声明”而不存放“定义”。在 C++语法中，类的成员函数可以在声明的同时被定义，并且自动成为内联函数。这虽然会带来书写上的方便，但却造成了风格不一致，弊大于利。建议将函数的定义与声明分开，不论该函数体有多么小。

⑤ 不提倡使用全局变量，尽量不要在头文件中出现 extern int value 这类声明。

定义文件主要也是由三部分组成：定义文件开头处的版权和版本声明、对一些头文件的引用和程序体的实现。

在校园信息管理系统中，为了更好的组织源代码，我们都将程序以头文件加定义文件的方式编写。

4．功能模块设计

本章项目设计的主要工作，就是将前面写好的代码，按照上面所讲的多文件结构重新组织。

在校园信息管理系统中，所有的功能模块都会分别封装起来，然后由用户操作界面部分程序，根据用户的选择，调用执行不同的功能。

用户操作界面部分程序代码，将存放在 main.cpp 中，所有程序中将使用到的头文件将统一放在 main.h 文件中。

5．完整程序代码

```
//main.h
/*
* Copyright (c) 2005,公司名称
* All rights reserved.
*
* 文件名称：main.h
* 文件标识：...
* 摘　　要：主程序头文件
*
* 当前版本：1.0
* 作　　者：输入作者（或修改者）名字
* 完成日期：...
*
* 取代版本：...
* 原作者　：输入原作者（或修改者）名字
* 完成日期：...
*/
#ifndef MAIN_H
#define MAIN_H
#include <iostream>
#include <iomanip>
#endif

//main.cpp
/*
```

```
* Copyright (c) 2005,公司名称
* All rights reserved.
*
* 文件名称：main.cpp
* 文件标识：...
* 摘　　要：主程序定义文件
*
* 当前版本：1.0
* 作　　者：输入作者（或修改者）名字
* 完成日期：...
*
* 取代版本：...
* 原作者　：输入原作者（或修改者）名字
* 完成日期：...
*/
#include "main.h"
using namespace std;
void helpCMD()
{
    cout<<setiosflags(ios::left)<<setw(20)<<"CDM"<<"DESCRIBE"<<endl<<endl;
    cout<<setiosflags(ios::left)<<setw(20)<<"input"<<"input data"<<endl;
    cout<<setiosflags(ios::left)<<setw(20)<<"output"<<"output data"<<endl;
    cout<<setiosflags(ios::left)<<setw(20)<<"analyze"<<"analyze
data"<<endl;
    cout<<setiosflags(ios::left)<<setw(20)<<"save"<<"save data"<<endl;
    cout<<setiosflags(ios::left)<<setw(20)<<"load"<<"load data"<<endl;
    cout<<setiosflags(ios::left)<<setw(20)<<"exit"<<"exit system"<<endl;
}
void inputCMD()
{
    cout<<"this is input cmd."<<endl;
}
void outputCMD()
{
    cout<<"this is output cmd."<<endl;
}
void analyzeCMD()
{
    cout<<"this is analyze cmd."<<endl;
}
void saveCMD()
{
    cout<<"this is save cmd."<<endl;
}
void loadCMD()
{
    cout<<"this is load cmd."<<endl;
}
bool exitCMD()
{
    char bExit;
    cout<<"are you sure to exit?(Y/N)"<<endl;
    cin>>bExit;
    cin.sync();
    if(bExit=='Y'||bExit=='y')
        return true;
```

```
    else
        return false;
}
void main()
{
    char cmd[64];
    do
    {
        cout<<">";
        cin>>cmd;
        cin.sync();
        if(strcmp(cmd,"exit")==0)
        {
            if(exitCMD())
                break;
        }
        else if(strcmp(cmd,"help")==0||strcmp(cmd,"?")==0)
        {
            helpCMD();
        }
        else if(strcmp(cmd,"input")==0)
        {
            inputCMD();
        }
        else if(strcmp(cmd,"output")==0)
        {
            outputCMD();
        }
        else if(strcmp(cmd,"analyze")==0)
        {
            analyzeCMD();
        }
        else if(strcmp(cmd,"save")==0)
        {
            saveCMD();
        }
        else if(strcmp(cmd,"load")==0)
        {
            loadCMD();
        }
        else
        {
            cout<<"ERROR:no this cmd!"<<endl;
        }
    }while(true);
}
```

6．总结

本章项目设计主要从预处理及多文件结构入手，论述 C++的编程风格。难度不高，但是细节比较多。别小看这些内容，提高程序代码质量就是要从这些点点滴滴做起。世上不存在最好的编程风格，一切因需求而定。如果读者觉得本书的编程风格比较适合你的工作，那么就采用它，不要只看不做。人在小时候说话发音不准，写字潦草，如果不改正，总有后悔的时候。编程也是同样道理。

本章小结

所谓变量的作用域，就是变量的作用范围，也可以说是变量的有效性范围。C++程序中的作用域可以分为函数原型作用域、块作用域、类作用域和文件作用域。变量按作用域范围可分为局部变量和全局变量。一个变量只有在可见的情况下才能访问。

变量的生存期是指变量占用内存空间的时间，也可以称为变量的存储方式。变量的生存期可以分为静态生存期和动态生存期两种，静态生存期与程序的运行期相同，动态生存期则只是在程序运行过程中的某一段时期。

在 C++中，对变量的存储类型说明有四种：自动类型（auto）、静态类型（static）、外部类型（extern）和寄存器类型（register）。

自动变量和寄存器变量属于动态存储，外部变量和静态变量属于静态存储。

最后，我们讨论了 C++程序的文件结构和编译预处理命令。为使在不同源文件中保持声明的一致性，采用了头文件。一个 C++程序由多个头文件和源文件组成，这些文件相互之间用包含指令（include）联系在一起，统一管理，从而提高了程序编写、编译和连接的效率。

习题

3-1　写出下列程序的运行结果。

```
#include <iostream>
using namespace std;
int a=3,b=5;
void sub(int x)
{
    int a;
    a=x;
}
void main()
{
    int b=8;
    sub(b);
    cout<<"a="<<a<<", b="<<b<<endl;
}
```

3-2　写出下列程序的运行结果。

```
#include <iostream>
using namespace std;
int max(int a,int b)
{
  int c;
  if(a>b)  c=a;
  else c=b;
  return(c);
}
void main()
{
  extern x,y;
  int z;
  z=max(x,y);
  cout<<"max="<<z<<endl;
}
```

```
int x=5,y=8;
```

3-3　写出下列程序的运行结果。

```
#include <iostream>
using namespace std;
void fun();
void main()
{
  int i=1;
  if(i==1)
  {
    int i=2;
    cout<<i++;
  }
  { extern int i;
    cout<<++i;
  }
  fun();
  fun();
}
void fun()
{
  static int i;
  cout<<i++;
}
int i;
```

3-4　写出下列程序的运行结果。

```
// file1.cpp
int g(int v);
static int i=20;
int x;
void f(int v)
{
    x=g(v);
}
static int g(int p)
{
    return i+p;
}

// file2.cpp
#include <iostream>
using namespace std;
extern int x;
void f(int);
void main()
{
    int i=5;
    f(i);
    cout<<x;
}
```