第5章 模 块 化

导 学

【主要内容】

在日常生活中要完成一项较复杂的任务时，人们通常会将任务分解成若干个子任务，通过完成这些子任务逐步实现任务的整体目标，这里采用的就是模块化的思想。在利用计算机解决实际问题时，也通常是将原始问题分解成若干个子问题，对每个子问题分别求解后再根据各子问题的解求得原始问题的解。本章重点介绍模块化的思想，以及如何使用C++编写模块化程序。通过本章的学习，读者一方面能够使用模块化的思想对问题进行分解，简化问题求解过程；另一方面能够使用C++编写模块化程序解决实际问题。本章还介绍C++中带默认形参值的函数、函数重载、变量和函数的生存期及作用域、多文件结构以及编译预处理等方面的内容。本章最后给出一个应用程序实例，该实例涉及二分查找方法，启发和激励读者利用计算思维解决更多、更复杂的问题。

【重点】

- 模块化及函数。
- 递归算法及递归函数。
- 变量和函数的生存期和作用域。
- 编译预处理。
- 多文件结构程序。

【难点】

- 正确编写递归函数。
- 合理组织多文件结构程序中各文件的内容。

5.1 模块化及其C++实现

模块化是指解决一个复杂问题时将其自顶向下逐层划分为若干子问题(模块)的过程。

5.1.1　采用模块化思想处理问题

【问题 5-1】　计算组合数 C(n,m)。

问题求解思路：要计算组合数 C(n,m)，即从 n 个不同元素中取出 m(m≤n)个元素的所有组合的个数，其计算公式为 $C(n,m)=\frac{n!}{m!\ (n-m)!}$ 或写为 $n!\ (m!)^{-1}[(n-m)!]^{-1}$。该问题可分解为多个子问题。其中，在 3 次求阶乘的子问题中，只是子问题规模不同，而计算方法完全相同。因此，可以设计如表 5-1 算法来解决该问题。

表 5-1　求解问题 5-1 的算法

步骤	处　　理
1	计算 n 的阶乘 J1
2	计算 m 的阶乘 J2
3	计算 n－m 的阶乘 J3
4	计算组合数 C＝J1/J2/J3

5.1.2　用 C++ 实现结构化程序设计

结构化程序设计方法的核心是将程序模块化，结构化程序设计的程序结构如图 1-8 所示。

在 C++ 中，通过编写和调用函数来实现问题的模块化求解过程。函数就是一个能够完成某个独立功能的程序模块(子程序)。

函数是 C++ 程序的重要组成部分，设计 C++ 程序的过程就是编写函数的过程。前面设计的程序就是编写一个我们已经非常熟悉的主函数——main()。对于一些简单的问题，用一个 main()就可以了。对于复杂的问题，需要按照模块化的思想将一个复杂程序问题分解为多个相对简单的子问题，对每一个子问题使用一个函数实现求解。所以，一个 C++ 程序由一个 main()和若干个函数构成。

一个 C++ 程序至少且仅能包含一个 main()。main()函数是整个程序的入口，通过在 main()中调用其他函数，这些函数还可以相互调用，甚至自己调用自己来实现整个程序的功能。函数和外界的接口体现为参数传递和函数的返回值。

1. 函数的定义

函数定义的一般格式如下：

```
<函数类型> <函数名>([<形参表>])
{
    函数体
}
```

其中，函数的定义分为两部分，函数头和函数体。第一行为函数头，包括＜函数类型＞、＜函数名＞和＜形参表＞。花括号“{}”括起来的部分为函数体。

下面对函数定义中的各部分进行说明。

- ＜函数名＞是一个符合 C++ 语法要求的标识符，其命名规则与变量的命名规则相同。
- ＜形参表＞是函数名后面用一对圆括号“()”括起来的关于函数参数的个数、名称和类型的说明列表。这些参数在定义函数时进行说明，所以被称为形式参数，简称形参。＜形参表＞中参数个数多于 1 个时，参数之间用逗号“,”分开。函数可以没有形参，没有形参的函数称为无参函数，表示调用此函数时不需要给出参数，但无参函数名后面的一对圆括号“()”不能缺省。例如，前面程序中的主函数 main()就是一个无参函数。
- 函数体是用一对花括号括起来的语句。函数就是通过执行函数体中的语句实现特定的功能。
- ＜函数类型＞。函数的类型分为两种：有值函数和无值函数。对于有值函数，在函数体中用转向语句“return ＜表达式＞;”返回函数的值，＜表达式＞的类型要与声明的函数类型相一致。对于无值函数，在定义函数时，函数类型要声明为 void 类型，在函数体内不需要有 return 语句，如果有 return 语句，其后的表达式为空（即“return;”），则表示仅从函数返回。

【例 5-1】 定义 Fac 函数，实现求 n!功能。

```
int Fac(int n)
{
    int J=1, i;
    for(i=2; i<=n; i++)
        J=J*i;
    return J;
}
```

在上面的函数定义中，函数名为 Fac，有一个 int 型的形参 n。函数体是花括号括起来的部分，实现求 n! 的值并将其作为函数值返回的功能。函数类型为 int 型，与函数体中的“return J;”语句中 J 的数据类型一致。

【例 5-2】 定义 RectangleArea 函数，求边长为 x 和 y 的长方形面积。

```
double RectangleArea(double x, double y)
{
    return x*y;
}
```

在上面的函数定义中，函数名为 RectangleArea，有两个 double 型的形参 x 和 y。函数体为花括号括起来的部分，实现将 x×y 的结果作为函数值返回。函数类型为 double 型，与函数体“return x*y;”语句中表达式 x*y 的数据类型一致。

2. 函数的调用

C++ 程序是从主函数 main()开始执行，当执行到函数调用语句时，就会跳转去执行被

调用函数的代码，该函数被执行后又会返回到调用它的函数。被调用的函数也可以调用其他函数。在一个函数里对一个已经定义的函数的调用格式如下：

函数名 ([<实参表>])

上述函数调用格式就是函数调用表达式。其中，函数名就是定义函数时的函数名，实参表是调用函数时实际传递给函数的参数（简称实参）列表，实参的个数、类型、顺序要和形参一一对应。在函数调用时，将实参的值传递给相应的形参。例如，调用 Fac 函数求 5！和 7！如下：

```
Fac(5)
Fac(7)
```

【例 5-3】 根据表 5-1 给出的算法编写模块化程序，实现求组合数 C(n,m)的问题。

程序如下：

```
#include<iostream>
using namespace std;
//定义 Fac()函数
int Fac(int x)
{
    int J=1, i;
    for(i=2; i<=x; i++)
        J *=i;
    return J;
}
//定义主函数
int main()
{
    int n=5, m=2;                    //待处理的数据
    int J1, J2, J3, C;
    J1=Fac(n);                       //调用 Fac 函数求 n 的阶乘,将计算结果保存在 J1 中
    J2=Fac(m);                       //调用 Fac 函数求 m 的阶乘,将计算结果保存在 J2 中
    J3=Fac(n-m);                     //调用 Fac 函数求 n-m 的阶乘,将计算结果保存在 J3 中
    C=J1/J2/J3;                      //计算组合数
    cout<<"组合数为: "<<C<<endl; //输出结果
    return 0;
}
```

3. 函数原型

C++ 中，在调用一个函数之前，必须首先定义这个函数。如果在函数调用之后再进行函数定义的话，就需要在调用之前给出函数原型。函数原型也称为函数声明。编译系统根据函数原型确定函数调用时的函数名、参数个数、类型以及函数返回值类型。

函数原型的一般格式如下：

<函数类型> <函数名> ([<形参说明表>]);

函数原型就是函数头加上分号。在函数原型中形参名可以缺省，例如：

```
int Fac(int x);
```

可以写成

```
int Fac(int);
```

例 5-3 中如果 Fac()函数在后边定义，则必须在函数调用前给出 Fac()的函数原型，否则编译器会报错。此时，例 5-3 的程序代码顺序变为：

```
using namespace std;
int Fac(int);                    //Fac()的函数原型
int main()                       //定义主函数
{…}
int Fac(int n)                   //定义 Fac()函数
{…}
```

5.1.3 函数的调用机制及内联函数

当一个函数调用另一个函数时，系统会将当前函数的运行状态保存起来，然后再去执行被调用的函数；当被调用的函数执行完毕后，系统又会将刚才保存的运行状态恢复，继续执行函数调用后面的语句。

运行环境的保存和恢复、函数跳转都要消耗一定的时间。如果被调用函数实现的功能比较复杂，其计算时间会远远大于函数调用所额外消耗的时间，此时函数调用所带来的额外时间开销就可以忽略不计。但是，如果被调用函数实现的功能非常简单并且会被频繁地调用，在编写程序时就必须考虑因函数调用所造成的额外时间开销。

为了解决上述问题，一种比较好的方案就是使用内联函数。在编译程序时，系统会直接将调用内联函数的地方用内联函数中的函数体做等价替换。这样，在程序运行时就不需要进行函数调用，从而避免运行环境保存和恢复及函数跳转所引起的额外时间开销，提高程序的执行效率。内联函数定义的一般格式如下：

```
inline<函数类型><函数名>([<形参表>])
{
    函数体
}
```

在函数定义的＜函数类型＞前加上 inline 关键字，即为内联函数的定义。例如，对例 5-2，用于求长方形面积的函数的功能非常简单，为了避免函数调用所引起的额外时间开销，就可以将它们定义成内联函数：

```
//定义"求边长为 x 和 y 的长方形的面积"的内联函数
inline double RectangleArea(double x, double y)
{
    return x * y;
}
```

说明：内联函数只适用于功能简单的小函数。对于函数体中包含循环、switch 等复杂结构控制语句的函数以及语句比较多的函数，即便该函数被定义为内联函数，编译器也往往会放弃内联方式，将其作为普通函数处理。如例 5-1 中 Fac()的函数体中包含循环，所以不能定义为内联函数。

必须在内联函数定义处给出 inline 关键字。如果仅在函数声明时给出 inline 关键字、而在函数定义时未给出 inline 关键字，则该函数会被编译器视为普通函数。

5.1.4　调用库函数

C++ 中的函数分为两类：一类是用户根据待求解问题的需要自己定义的函数，如前面定义的 Fac 和 RectangleArea 函数；另一类是系统提供的标准函数，即库函数。系统将一些经常用到的功能定义为一个个的函数，当程序中要使用这些功能时，只需直接调用相应的库函数即可。

【问题 5-2】 求一个数的平方根。

问题求解思路：系统提供了库函数 sqrt()，可以在程序中包含声明了库函数 sqrt()的头文件 cmath，然后直接调用该 sqrt()库函数完成计算(关于文件包含的内容见 5.5 节)。

【例 5-4】 编写程序，调用库函数 sqrt()实现如下计算：

$$S(n) = 1 + 2^{1/2} + 3^{1/2} + \cdots + i^{1/2} + \cdots + n^{1/2}$$

其中，n 为大于 0 的整数。

```
#include<iostream>
#include<cmath>                          //将头文件 cmath 包含在程序中
using namespace std;
int main()
{
    int n=10;                            //待求和的数据项的数量
    double S=1;                          //用于保存处理结果
    int i;
    for(i=2; i<=n; i++)
    {
        S+=sqrt((double)i);
    }
    cout<<"计算结果为："<<S<<endl;       //输出结果
    return 0;
}
```

说明：由于 sqrt()函数的参数要求是 double 类型的，因此例 5-4 中需要将 int 型的 i 强制转换成 double 类型。

5.2　递归算法及其 C++ 实现

5.2.1　递归算法

在某些情况下，分解之后待解决的子问题与原问题有着相同的特性和解法，只是在问题

规模上与原问题相比有所减小，此时就可以设计递归算法进行求解。在递归算法中，一个函数会直接或间接地调用自己来完成某个计算过程。函数直接或间接调用自己的这种方式被称为函数的递归调用，这样的函数称为递归函数。

5.2.2 递归算法实例

【问题 5-3】 求 n!。

问题求解思路： 对于计算 n!的问题，可以将其分解为：n!＝n×(n－1)!。可见，分解后的子问题(n－1)!与原问题 n!的计算方法完全一样，只是规模有所减小。同样，(n－1)!这个子问题又可以进一步分解为(n－1)×(n－2)!，(n－2)!可以进一步分解为(n－2)×(n－3)!……直到要计算 1!或 0!时，直接返回 1。因此，可以设计解决该问题的递归算法。

求 n!的递归算法如下：

```
如果 n=1,或=0
    则 n!的值为 1;
否则,
    n!的值为 n×(n-1)!
```

【例 5-5】 根据求 n!的递归算法编写递归函数，并实现求 5!。

```
#include<iostream>
using namespace std;
int Fac(int n);                                    //函数声明
int main()
{
    int n=5;
    cout<<n<<"的阶乘为"<<Fac(n)<<endl;             //函数调用
    return 0;
}
int Fac(int n)                                     //求 n!的递归函数
{
    if(n==1 || n==0)
        return 1;
    else
        return n * Fac(n-1);                       //函数的递归调用
}
```

【问题 5-4】 求斐波那契数列(1,1,2,3,5,8,13,…,)第 n 项的的值。

问题求解思路： 将问题分解为第 n 项等于第 n－1 项和第 n－2 项之和。分解后的子问题求 n－1 项的值和求 n－2 项的值与原问题计算方法完全一样，只是规模有所减小。问题可以一直分解下去，直到求第 1 项或第 2 项的值，直接返回 1。因此，可以设计解决该问题的递归算法。

求斐波那契数列(1,1,2,3,5,8,13,…,)第 n 项的值的递归算法如下：

```
如果 n=1 或 n=2,
```

斐波那契数列的值为 1；

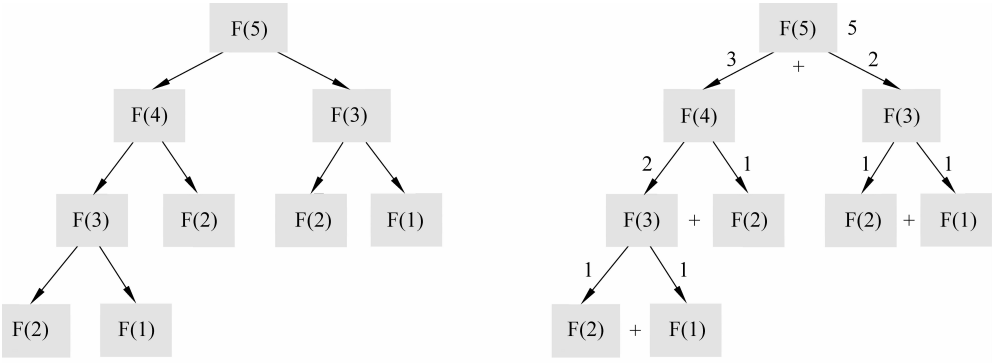
否则，

斐波那契数列的值为第 n-1 项与第 n-2 项之和

【例 5-6】 根据求斐波那契数列(1,1,2,3,5,8,13,…)第 n 项的值的递归算法编写程序，求第 5 项斐波那契数列的值。

```
#include<iostream>
using namespace std;
int F(int n);                                    //函数声明
int main()
{
    int n=5;
    cout<<"第"<<n<<"值为: "<<F(n)<<endl;         //函数调用
    return 0;
}
int F(int n)                                     //函数定义
{
    if(n==1 || n==2)
        return 1;
    else
        return F(n-1)+F(n-2);                    //函数的递归调用
}
```

图 5-1 描述了上面 F 函数的递和归的过程。



(a) 自顶向下分解过程——递　　(b) 自底向上分解过程——归

图 5-1　递归函数 F 的递和归的过程

在编写递归函数时，首先要有递的过程：先假设该函数的功能已经实现并可以直接调用，例如编写 F()函数时，可以通过语句“F(n－1)＋F(n－2);”来计算 F(n)；然后要有归的过程，即递归调用的函数必须有能够结束递归调用的条件语句，否则会一直递归调用下去，使程序处于无响应状态。例如，在使用递归方式计算斐波那契数列第 n 项时，当 n 的值为 1 或 2 时，就不需再执行递归调用了，直接返回 1 就可以。

5.3 默认形参值

在调用函数时,需要针对函数中的每一个形参给出相应的实参。C++ 中也允许在函数定义或函数声明时给出默认的形参值。在调用函数时,对于有默认值的形参,如果没有给出相应的实参,则函数会自动使用默认形参值;如果给出相应的实参,则函数会优先使用传入的实参值。

5.3.1 指定默认形参值的位置

默认形参值可以在两个位置指定:如果有函数声明,则应在函数声明处指定;否则,直接在函数定义中指定。

【例 5-7】 默认形参值函数示例——默认形参值的位置。

```
#include<iostream>
using namespace std;
void f(char * str="abc");        //形参 str 的默认值为"abc",在函数声明处指定
int main()
{
    f();                         //函数调用时没有给出实参,则形参 str 用默认值"abc"
    f("def");                    //函数调用时给出了实参,则形参 str 的值为"def"
    return 0;
}
void f(char * str)
{
    cout<<str<<endl;
}
```

说明:

- 对于有默认值的形参,如果在调用函数时给出了相应的实参,则会优先使用传入的实参值。例如,执行“f("def");”会输出 def。
- 如果有函数声明,则应在函数声明中给出默认形参值,函数定义中不要重复指定。例如:

```
void f(char * str="abc");    //函数声明部分
void f(char * str="abc")     //函数定义部分。错误:默认形参值被重复指定
{
    cout<<str<<endl;
}
```

- 默认形参值可以是全局常量、全局变量,甚至是可以通过函数调用给出,但不能是局部变量。因为默认形参值或其获取方式需在编译时确定,而局部变量在内存中的位置在编译时无法确定。

5.3.2　默认形参值的指定顺序

默认形参值必须严格按照从右至左的顺序进行指定。

【例 5-8】　默认形参值函数示例——指定参数值的顺序。

```
#include<iostream>
using namespace std;
void f(int a, int b=2, int c=3, int d=4)     //在函数定义时指定默认形参值
{
    cout<<a<<" "<<b<<""<<c<<" "<<d<<endl;
}
int main()
{
    f(1);                    //形参 a=1,形参 b、c 和 d 用默认值
    f(5, 10);                //形参 a=5,形参 b=10,形参 c 和 d 用默认值
    f(11, 12, 13);           //形参 a=11,形参 b=12,形参 c=13,形参 d 用默认值
    f(20, 30, 40, 50);       //形参 a、b、c 和 d 均用实参的值。
    return 0;
}
```

说明：当调用函数时，系统按照从左至右的顺序将实参传递给形参，当指定的实参数量不够时，没有相应实参的形参采用其默认值。

如果没有相应实参的形参没有指定默认值，则会出错。如在例 5-8 中

```
f();          //错误：第一个参数 a 没有默认值
```

如果在例 5-8 中有

```
void f(int a=2, int b, int c=3, int d=4);
```

这种指定默认形参值的写法是错误的，这是由于在第二个参数 b 未指定默认形参值的情况下给出了第一个参数 a 的默认形参值，不符合从右至左的指定顺序。

5.4　函 数 重 载

C++ 允许不同的函数具有相同的函数名，这就是函数重载。

对于函数名相同的多个函数，要在调用时能够区分开到底要调用哪个函数，只能根据传递的实参在数量或类型上的不同进行判断。因此，函数名相同的函数形参列表不能完全一样，即参数的个数或参数的类型必须有所区别，否则会因无法区分而报错。

【例 5-9】　函数重载示例——实参类型不同。

```
#include<iostream>
using namespace std;
int f(int x);
```