第5章 函数

函数是C++最基本的程序模块。本章介绍如何定义并使用函数。

5.1 定义和调用函数

5.1.1 函数的定义

如果在一个程序中不同的地方,需要经常执行一组相同的语句来完成一种相对独立的功能,则可以把这组语句定义为一个独立的程序模块——函数,需要执行时,只要用函数名对其进行调用即可。

使用函数是为了达到代码重用的目的,避免重复存储完全相同的代码,节省存储器空间;同时也可以减少编程时的重复劳动,提高编程效率;并使程序结构清晰、易读,易于修改,符合结构化程序设计原理。

定义函数的语法形式如下:

```
类型说明符 函数名(形式参数列表)
{
    语句序列;
}
```

例如:

```
int max(int j1, int j2)
{
    int r;
    r = j1 >= j2? j1:j2;
    return r;
}
```

上面的程序语句定义了一个函数max,功能是返回两个整型参数中较大的一个。

以下是对函数定义的几点说明。

(1) 有的函数执行结束时,要向调用它的程序返回一个值,这种函数称为有返回值的函数。函数定义中的类型说明符就是函数返回值的数据类型。如上面定义的max函数的返回值类型为int型。函数的返回值可以是除数组外的其他任何类型,如整型、浮点型、字符型、布尔型、指针型,甚至可以是结构和类的对象。如果一个函数有返回值,则在函数内部必

须使用返回语句(return)把值返回到程序中调用函数的地方。

(2) 如果一个函数没有返回值,则可以用关键字 void 作为类型说明符。这样的函数称为无返回值函数或 void 型函数。C++程序中的 main 函数通常就是一个 void 型函数。

(3) 函数名是一个标识符。一个程序中,除 main 函数和 C++库函数外,其他的函数都是由编程者命名的,如上面定义的函数 max。

(4) 函数名后的小括号中是函数的形式参数列表。函数的参数是函数被调用时,由调用它的程序传递给函数的数据。通常把出现在函数定义中的参数称为**形式参数**(简称形参),如 max 函数的参数 j1 和 j2。形式参数列表可以为空,表示函数没有参数。非空的形参列表的语法形式如下:

```
数据类型说明符 1 参数 1,数据类型说明符 2 参数 2,…,数据类型说明符 n 参数 n
```

数据类型说明符指定了形参的数据类型。形参可以是 C++中的任何数据类型。在函数内部可以像使用变量一样来使用形式参数。

(5) 函数定义中包含函数名的第一行语句称为函数头;函数名后面由花括号括住的语句序列叫作函数体。

5.1.2 函数的调用

函数被定义后,就可以对函数进行调用了。调用函数的语法形式如下:

```
函数名(实际参数列表)
```

函数名后面的小括号中是实际参数列表,把调用函数时向函数传递的参数称为**实际参数**(简称实参)。实参可以是常量、变量或表达式。

函数调用既可以是一条单独的语句,也可以是表达式中的一部分。

在其内部发生函数调用的函数称为主调函数;被调用的函数称为被调函数。

例如:

```
void main()
{
    int i1,i2,lar;
    cout <<"请输入两个正整数: ";
    cin >> i1 >> i2;
    lar = max(i1,i2);
    cout <<"整数"<< i1 <<"和"<< i2 <<"中较大的一个是: "<< lar;
    cout << endl;
}
```

main 函数中的语句 lar=max(i1,i2);调用了前面定义的 max 函数。这里 main 函数是主调函数,max 函数是被调函数。程序执行到语句 lar=max(i1,i2);时发生了函数调用,程序控制转移到 max 函数中,开始执行 max 函数;max 函数执行结束后,程序控制又转回到 main 函数中调用函数的地方,首先把 max 函数的返回值赋值给变量 lar,然后开始顺序执行后面的语句。

C++允许函数进行嵌套调用(一个被调函数内部又发生了函数调用),而且不限制函数

嵌套调用的层次。

5.1.3 函数原型

函数原型用来声明已经定义的函数。在一个程序文件中，如果函数的定义出现在程序中所有函数调用的前面，则可以不使用函数原型；如果函数的定义出现在函数调用语句的后面，或者是在同一个程序的其他源文件中，则在函数调用前，必须使用函数原型对函数进行声明。

声明函数原型的语法形式如下：

```
类型说明符 函数名(数据类型说明符 1 参数 1,数据类型说明符 2 参数 2,…,数据类型说明符 n 参数 n);
```

或

```
类型说明符 函数名(数据类型说明符 1 ,数据类型说明符 2 ,…,数据类型说明符 n );
```

第一种形式类似函数定义中的函数头，唯一的不同是声明函数原型是一条语句，所以后面需要加分号；而函数头后面不能加分号。

和第一种声明形式相比，第二种形式只是在形参列表中去掉了所有形参的名字。例如，max 函数的函数原型为：

```
int max(int j1, int j2);
```

或

```
int max(int , int );
```

函数原型的作用是：告诉 C++编译器函数的返回值类型、参数的个数以及每个参数的数据类型等信息，以便编译器可以检查函数的调用语句形式是否和函数的定义形式相匹配。

例 5.1 创建一个函数比较两个整数的大小，并返回其中较大的一个。从键盘输入两个正整数，比较两个数的大小，并输出其中较大的一个。

```
#include<iostream>
using namespace std;
int max(int,int);
void main()
{
    int i1,i2,lar;
    cout<<"请输入两个正整数:";
    cin>>i1>>i2;
    lar=max(i1,i2);
    cout<<"整数"<<i1<<"和"<<i2<<"中较大的一个是:"<<lar;
    cout<<endl;
}
int max(int j1, int j2)
{
   int r;
   r= j1>=j2? j1:j2;
```

```
    return r;
}
```

例 5.1 程序的运行结果如图 5.1 所示。

```
请输入两个正整数: 100 50
整数100和50中较大的一个是: 100
```

图 5.1 例 5.1 程序的运行结果

例 5.2 编写一个函数求出并返回一个整数各位数字的和,在 main 函数中输入一系列整数,分别输出它们各位数字的和,直到输入 0 时,程序结束。

```
#include<iostream>
using namespace std;
int digsum(int val)
{   int s=0;
    while(val!=0)
    {
        s+=val%10;
        val/=10;
    }
    return s;
}
void main()
{
    int in;
    cout<<"请输入一个正整数: ";
    cin>>in;
    while(in!=0)
    {
        cout<<"整数"<<in<<"的各位数字的和为: "<<digsum(in)<<endl;
        cout<<"请输入一个正整数: ";
        cin>>in;
    }
}
```

例 5.2 程序的运行结果如图 5.2 所示。

```
请输入一个正整数: 100
整数100的各位数字的和为: 1
请输入一个正整数: 745
整数745的各位数字的和为: 16
请输入一个正整数: 999
整数999的各位数字的和为: 27
请输入一个正整数: 56878432
整数56878432的各位数字的和为: 43
请输入一个正整数: 0
```

图 5.2 例 5.2 程序的运行结果

例 5.3 创建一个函数,用辗转取余法求两个整数的最大公约数和最小公倍数。

算法思想为:若 i 和 j 为两个正整数,则令 i1=i,j1=j。

(1) 令 res=i1%j1;

(2) 若 res 等于 0,则 i 和 j 的最大公约数是 j1,最小公倍数是 i×j÷j1。

否则,令 i1=j1; j1=res; 返回第 1 步继续执行。

程序实现如下。

```
#include<iostream>
using namespace std;
void divandmul(int,int);
void main()
{   int i,j;
    cout<<"请输入两个正整数:";
    cin>>i>>j;
    if(i>0 && j>0)
        divandmul(i,j);
    else
        cout<<"输入错误\n";
}
void divandmul(int i,int j)
{
    int i1=i,j1=j,res=i1%j1;
    while(res!=0)
    {
        i1=j1;
        j1=res;
        res=i1%j1;
    }
    cout<<i<<"和"<<j<<"的最大公约数为:"<<j1<<endl;
    cout<<i<<"和"<<j<<"的最小公倍数为:"<<i*j/j1<<endl;
}
```

例 5.3 程序的运行结果如图 5.3 所示。

```
请输入两个正整数: 18 42
18和42的最大公约数为:6
18和42的最小公倍数为:126
```

图 5.3 例 5.3 程序的运行结果

例 5.4 函数的嵌套调用。设计三个函数分别求一个实数的乘方、圆的面积和圆柱体的体积。在 main 函数中输入圆柱体的底面半径和高,调用函数计算并输出圆柱体的体积。

```
#include<iostream>
using namespace std;
const double PI = 3.14159;
double cylinderArea(double, double);
double circleArea(double);
double power(double, int);
void main()
{
    double radius, height, vol;
    cout << "请输入圆柱体的底面半径和高:";
    cin >> radius >> height;
    vol = cylinderArea(radius, height);
    cout << "圆柱体的体积为:" << vol << endl;
}
double cylinderArea(double radius, double height)
```

```
{
    return circleArea(radius) * height;
}
double circleArea(double radius)
{
    return power(radius, 2) * PI;
}
double power(double x, int i)
{
    double prod = 1.0;
    for (int j = 0;j < i;j++)
        prod * = x;
    return prod;
}
```

例 5.4 程序的运行结果如图 5.4 所示。

```
请输入圆柱体的底面半径和高：10.0 5.0
圆柱体的体积为：1570.8
```

图 5.4 例 5.4 程序的运行结果

5.2 传递参数

函数调用时,主调函数通过向被调函数传递参数,把数据传递给被调函数。参数传递的方式有两种：**传值传递**和**引用传递**。

5.2.1 传值传递

定义函数时,形式参数的数据类型只要不是引用类型(4.2 节),则参数的传递方式就是传值传递。参数传值传递的特点是：主调函数中的实际参数和被调函数中的相应的形式参数是两个相互独立的量,即各自具有自己的存储空间。当发生函数调用时,实参的值被复制给相应的形参,而后,它们之间的联系断裂。所以当被调函数中形参的值改变时,主调函数中的实参不会随之改变。

例 5.5 参数的传值传递。

```
#include < iostream >
using namespace std;
void swap(int par1, int par2);
void main()
{
    int i{ 10 }, j{ 20 };
    cout << "调用函数 swap 前: i = " << i << " j = " << j << endl;
    swap(i, j);
    cout << "调用函数 swap 后: i = " << i << " j = " << j << endl;
}
void swap(int par1, int par2)       //传值传递参数
{
```

```
    cout << "swap 函数内,形参值交换前: par1 = " << par1 << " par2 = " << par2 << endl;
    int temp = par1;
    par1 = par2;
    par2 = temp;
    cout << "swap 函数内,形参值交换后: par1 = " << par1 << " par2 = " << par2 << endl;
}
```

函数 swap 的功能是交换两个形式参数 par1 和 par2 的值,并在交换前后分别输出两个参数的值。main 函数中,定义并初始化了两个整型变量 i 和 j,然后以 i 和 j 作为实际参数调用函数 swap,并在调用前后分别输出 i 和 j 的值。

程序的运行结果如图 5.5 所示。

```
调用函数swap前: i=10 j=20
swap函数内, 形参值交换前: par1=10 par2=20
swap函数内, 形参值交换后: par1=20 par2=10
调用函数swap后: i=10 j=20
```

图 5.5 例 5.5 程序的运行结果

从程序的运行结果可以看到,虽然在函数 swap 内部交换了两个形参 par1 和 par2 的值,但 main 函数中的两个实参的值并没有被交换。正如“传值传递”方式的名字,实参只是把值传递给相应的形参,而后它们就互不相干了,形参的改变不会影响到实参。传值传递是信息从实参到形参的单向传递(不包括指针)。

5.2.2 引用传递

在函数的定义中,如果形式参数为引用类型,则参数的传递方式就是引用传递。正如我们所知道的:一个引用类型的变量是另一个变量的别名(4.3.1 节),引用类型的形式参数也是相应的实际参数的别名,它们其实是同一个变量,即在内存中占用同一块存储空间。所以如果在被调函数中形参的值发生了改变,则主调函数中实参的值也会随之改变。

例 5.6 参数的引用传递。

```
#include <iostream>
using namespace std;
void swap(int &par1,int &par2);
void main()
{
    int i = 10,j = 20;
    cout <<"调用函数 swap 前: i = "<< i <<" j = "<< j << endl;
    swap(i,j);
    cout <<"调用函数 swap 后: i = "<< i <<" j = "<< j << endl;

}
void swap(int &par1,int &par2)        //引用传递参数
{
    cout <<"swap 函数内,形参值交换前: par1 = "<< par1 <<" par2 = "<< par2 << endl;
    int temp = par1;
    par1 = par2;
    par2 = temp;
    cout <<"swap 函数内,形参值交换后: par1 = "<< par1 <<" par2 = "<< par2 << endl;
```

```
}
```

本例的程序和例 5.5 的程序基本相同,唯一的区别是:函数 swap 的定义中使用符号 &. 把两个形参 par1 和 par2 定义为引用类型,此时参数的传递方式为引用传递。由于 swap 函数的形参 par1 和 par2 分别是 main 函数中作为实参的变量 i 和 j 的别名,所以交换 par1 和 par2 的值就等价于交换 i 和 j 的值。例 5.6 程序的运行结果如图 5.6 所示。

```
调用函数swap前: i=10 j=20
swap函数内, 形参值交换前: par1=10 par2=20
swap函数内, 形参值交换后: par1=20 par2=10
调用函数swap后: i=20 j=10
```

图 5.6 例 5.6 程序的运行结果

在参数的引用传递方式中,信息是双向传递的。本节讨论的引用就是 C++ 11 中的左值引用。

5.3 局部变量和全局变量

在函数内部或由花括号括住的复合语句内部定义的变量就是局部变量。例如,例 5.6 中 main 函数内定义的整型变量 i,j 是 main 函数的局部变量,swap 函数内定义的变量 temp 是 swap 函数的局部变量。

在程序中变量有效的区域称为变量的作用域。局部变量的作用域就是定义它的函数或复合语句块,所以称它们具有**局部作用域**。例如,例 5.6 中 main 函数内的变量 i 和 j 只能在 main 函数内部使用,而变量 temp 只能在 swap 函数内部使用。在不同的函数和复合语句中可以定义同名的局部变量,它们相互独立互不影响。如果在嵌套的复合语句中定义了同名的局部变量,则内层的局部变量屏蔽了外层的局部变量。

例 5.7 定义和使用局部变量。

```
#include <iostream>
using namespace std;
void main()
{
    float val = (float)123.456;
    {
        int val = 100;
        cout <<"整型变量 val 的值为"<< val << endl;
    }
    cout <<"实型变量 val 的值为"<< val << endl;
}
```

main 函数中定义了 float 类型的局部变量 val,而在内层的由花括号括住的复合语句中定义了同名的 int 型局部变量 val,则在内层花括号中,内层定义的 int 型局部变量 val 屏蔽了外层定义的 float 型同名变量,即在内层花括号内部,外部定义的 float 型变量 val 不可见。例 5.7 程序的执行结果如图 5.7 所示。

在程序的运行过程中,一个变量从被定义时开始,直到它被销毁为止的这段时期叫作该变量的生存期。根据生存期的不同,局部变量又分为自动局部变量和静态局部变量。

```
整型变量val的值为100
实型变量val的值为123.456
```

图 5.7 例 5.7 程序的执行结果

没有使用关键字 static 定义的局部变量都是自动变量。当程序调用一个函数或开始执行一个代码块时，就会创建其中的自动局部变量，为其分配存储空间。随着函数运行结束，自动局部变量也被自动销毁，**所以称自动变量具有动态生存期**。例如，例 5.6 中 swap 函数内定义的变量 temp，函数 swap 开始执行时，变量 temp 被创建，函数 swap 运行结束时，它的生存期也随之结束。函数形参具有和自动局部变量相同的作用域和生存期。

如果使用关键字 static 来定义局部变量，则该变量为静态局部变量。静态变量一经定义，就一直存在，直到整个程序运行结束。程序执行过程中，一个函数第一次被调用执行时，创建其中的静态局部变量，为它们分配存储空间；函数执行结束，其中定义的静态局部变量仍然存在；该函数再次被调用执行时，由于其中的静态变量已经存在，故不需要重新定义。所以称静态局部变量具有**静态生存期**，即从第一次被定义时开始，直到整个程序运行结束。

例 5.8 使用静态局部变量统计函数被调用的次数。

```
#include<iostream>
using namespace std;
int fun();
void main()
{
    int i,j=0;
    cout<<"请输入一个整数:";
    cin>>i;
    while(i!=0)
    {
        j=fun();
        cout<<"请输入一个整数:";
        cin>>i;
    }
    cout<<"函数 fun 被调用了"<<j<<"次\n";
}
int fun()
{
    static int c=0;
    c++;
    return c;
}
```

程序运行时提示用户输入整数，用户输入一个非零整数，函数 fun 就会被调用一次，直到用户输入 0 为止，程序最后输出调用函数 fun 的次数。

函数 fun 第一次运行时，定义了静态整型局部变量 c，并初始化为 0，然后执行自加语句使 c 的值变为 1，最后返回变量 c 的值。函数 fun 运行结束后，静态变量 c 仍然存在而且保持原值不变，以后函数 fun 每次被调用都不再重新定义该变量，也就是说，在程序的执行过程中，静态变量 c 只是在函数 fun 第一次运行时被定义，直到整个程序运行结束才被销毁。并且每次运行函数 fun 时都令变量 c 执行自加 1 操作，故 c 的值就是函数 fun 被调用的次

```
请输入一个整数：5
请输入一个整数：4
请输入一个整数：3
请输入一个整数：2
请输入一个整数：1
请输入一个整数：0
函数fun被调用了5次
```

图 5.8 例 5.8 程序的运行结果

数。变量 c 虽然在程序的执行过程中一直存在，但是由于它是函数 fun 的局部变量，所以只在函数 fun 中可以使用。例 5.8 程序的运行结果如图 5.8 所示。

如果在定义变量 c 的语句中去掉关键字 static，程序将有怎样的运行结果呢？请读者思考。

程序中在任何函数之外定义的变量称为全局变量。全局变量的作用域是定义该变量的整个程序文件。全局变量的生存期就是程序的整个运行期。可以定义和全局变量同名的局部变量，则在该局部，局部变量屏蔽了同名的全局变量。

C++是多文件程序结构，可以使用关键字 extern 声明在其他文件中定义的全局变量，使其在本文件中可用。

程序中定义的全局变量和静态局部变量存放在存储器的静态存储区之中。它们具有相同的生存期和存储方式，但它们的作用域不同。

由于全局变量在定义它的程序文件的所有函数中都可见(除非定义了同名的局部变量)，不利于数据隐藏，降低了程序的安全性，所以应尽可能少地使用全局变量。

5.4 函数调用的实现

为了能更好地理解和掌握函数，本节简要介绍系统实现函数调用的方法。

一个程序执行时，操作系统会在存储器中分配一块空间供程序使用。通过学习 4.2.4 节，得知这块存储空间中应该包含一块称为“堆”的子空间，用来存放程序中采用 new 操作符动态创建的变量或对象。除了“堆”，这块存储空间中还有一块叫作“栈”的子空间专门用来实现函数调用。

“栈”是一种后进先出的顺序存储结构，数据的存取只能从栈顶进行，先存入的数据被逐步压向栈底，最后存入的数据位于栈顶，取数据时，也只能先读取栈顶数据，栈顶数据被弹出栈后，才能顺序读取下边的数据，如图 5.9 所示。

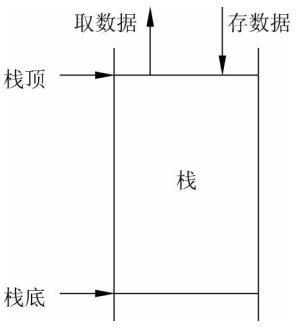

图 5.9 栈的结构

一个函数被调用时，系统首先在“栈”顶为其开辟一块空间，用来存放函数的形式参数和函数中定义的非静态局部变量(自动局部变量)，由于函数执行结束后，要返回到主调函数中并从函数调用的下一条指令开始继续执行，所以在这块“栈”空间中还要保存函数调用时的现场信息(系统寄存器的值)和返回地址；然后控制转到被调函数并开始执行；函数运行结束时，系统根据“栈”中保存的现场信息和返回地址恢复主调函数的执行现场，返回到主调函数中继续执行；分配给被调函数的“栈”空间被自动释放。在程序的执行过程中，“栈”空间的分配和释放是动态进行的。

如果一个程序运行时，在 main 函数中调用了函数 fun1，而在函数 fun1 中又调用了函数 fun2，则该程序运行时，“栈”的动态分配如图 5.10 所示。

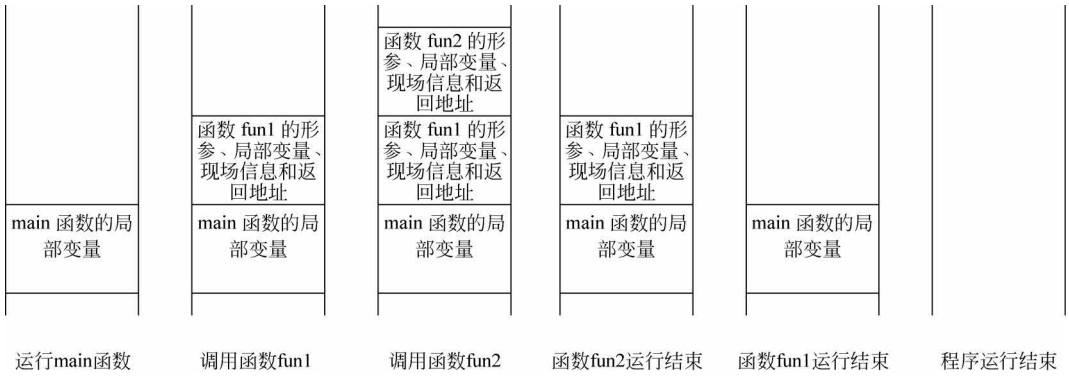

图 5.10 函数调用时“栈”空间的分配和释放

5.5 内联函数

使用函数的根本目的是实现代码重用，其好处是既节省了存储空间，也提高了编程效率。

通过 5.4 节的学习，了解了函数调用的实现过程：函数被调用时，系统首先在“栈”中为函数分配一块空间，并执行入栈指令把函数调用的现场信息、返回地址和参数压入“栈”中；然后还要执行指令(call)使控制转移到函数中，开始执行函数；函数执行结束返回时，还要执行指令传递函数返回值、恢复主调函数的执行现场并将控制转移回主调函数。所有这些指令都降低了程序的执行速度。也就是说，函数调用会带来额外的开销，降低程序的执行速度。

如果一个函数非常短而且在程序中需要频繁调用，那么对空间的节省并不明显，而多次函数调用所付出的代价却十分显著。那么该如何解决上述矛盾呢？方法是使用内联函数。

可以使用关键字 inline 把一些功能简单、规模较小并且需要频繁调用的函数定义为内联函数。语法如下：

```
inline 返回值类型 函数名(形参列表)
{
   函数体;
}
```

例如：

```
inline int max(int num1,int num2)
{  if(num1 > num2)  return num1;
   else  return num2;
}
```

C++编译器处理内联函数调用的方法是：**把内联函数的函数体直接嵌入到发生函数调用的地方，以取代函数调用语句，而不进行常规的函数调用**。

使用内联函数既可以达到代码重用的目的，也可以避免函数调用时的额外开销，提高程序的执行速度。但是需要注意，不要把较长的函数、包含循环语句和 switch 语句的函数定

义为内联函数,这样的函数即使被定义为内联函数,C++编译器也不会把它们作为内联函数来处理。

5.6 递归函数

递归是一种解决问题的数学方法。例如,可以用递归方法求正整数 n 的阶乘 $n!$。$n!$ 的递归定义如下:

$$n!=\begin{cases} n\times(n-1)! & (\text{当 } n>0 \text{ 时}) \quad (5.1)\\ 1 & (\text{当 } n=0 \text{ 时}) \quad (5.2)\end{cases}$$

以上两式给出了 $n!$ 的递归定义。当 n 大于 0 时,$n!=n\times(n-1)!$;同理$(n-1)!=(n-1)\times(n-2)!$;……这是一个**回溯**的过程,目的是把规模较大的问题逐步化简为**相同类型的**规模较小的问题。当 n 等于 0 时,$n!=0$,这时就回到了源头,这是**回溯终止的条件**,也就是说,当问题足够小时,可以容易地得到问题的解,从而终止回溯。根据式(5.1)可以从 0! 递推出 1!=1×0!=1,再从 1! 推出 2!,……,最后求出 $n!$。这是一个**递推**的过程,从小问题的解逐步推出规模较大的问题的解,最后得到原始问题的解。可以看到,用递归的方法解决问题主要包含回溯和递推两个过程。

C++使用递归函数解决递归问题。**如果一个函数直接或间接地调用了函数自己,则这个函数就是递归函数**。例如,可以定义一个求 n! 的递归函数 fac。

```
long fac ( int n)
{
   if(n == 0) return 1;
   else  return n * fac (n - 1);        //递归调用函数本身
}
```

图 5.11 中以 n=4 为例,模拟函数 fac 的调用过程,揭示递归函数的执行原理。

图 5.11 中的矩形表示一次函数调用,矩形间向下的箭头从主调函数指向被调函数,矩形右侧水平方向的箭头表示一次函数调用结束后返回到主调函数。从图中可以看出,每一次递归函数调用执行结束后,总是返回到函数中调用它的地方继续向后执行。

递归函数的逐级调用就是从繁到简的回溯过程;而到达回溯终止条件后,函数开始逐级返回,这是由简到繁的递推过程。

例 5.9 编写一个函数判断一个字符串的子串是否是回文字符串,回文字符串是指从左向右读和从右向左读都一样的不含空格的字符串。函数有三个参数,一个字符数组用来存储字符串,两个整型参数分别表示子串的起始位置和结束位置。

分析:设函数 int pal(char[] ch,int s,int e)是判断回文的函数。每次执行时,首先判断 ch[s]和 ch[e]是否相等,若相等,则以字符数组 ch、整数 s+1 和 e−1 为参数进行递归调用;若不相等,说明子串不是回文字符串,返回 0。递归调用的终止条件是 s≥e,说明子串是回文字符串,返回 1。

源程序如下。

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

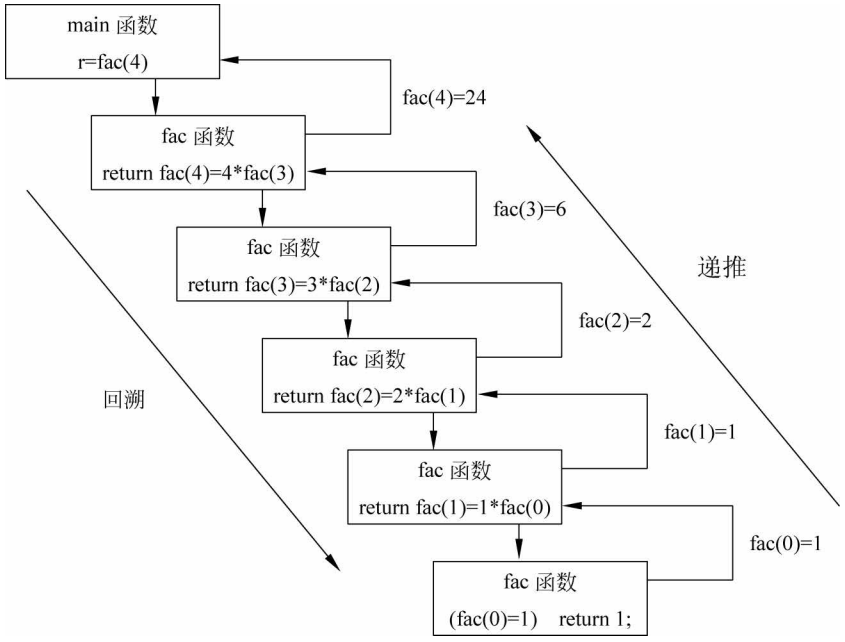

图 5.11 递归函数 fac 的调用过程

```
int pal(char a[], int s,int e);
int main()
{   char ch[100];
    int s,e;
    cout <<"请输入一行字符串: ";
    cin >> ch;
    cout <<"请输入两个不大于字符串长度的整数: ";
    cin >> s >> e;
    if(s > = e)
    {
        cout <<"输入错误\n"<< endl;
        return 0;
    }
    else if(pal(ch,s,e))
        cout <<"从第"<< s <<"个字符到第"<< e <<"个字符的子串是回文字符串\n";
    else
        cout <<"从第"<< s <<"个字符到第"<< e <<"个字符的子串不是回文字符串\n";
    return 1;
}
int pal(char a[], int s,int e)
{   if(s > = e)  return 1;
    else if(a[s] == a[e]) return pal(a,s + 1,e - 1);
    else return 0;
}
```

例 5.9 程序的运行结果如图 5.12 所示。

例 5.10 编写一个递归函数,输出一个单字节正整数 n 的二进制值。

```
请输入一行字符串：qwertyuioppoiuytrasf
请输入两个不大于字符串长度的整数：5 14
从第5个字符到第14个字符的子串是回文字符串
```

图 5.12 例 5.9 程序的运行结果

分析：根据求整数二进制值的方法，可以得出此问题的递归定义：

正整数 n 的二进制值 = 正整数 n/2 的二进制值 × 2 + n % 2

上式中的乘 2 表示将二进制数左移 1 位。根据递归定义可以很快地得出这个问题的递归算法。

源程序如下。

```
#include <iostream>
using namespace std;
void printBinary(unsigned char);
void main()
{
    unsigned char ch;
    int i;
    cout <<"请输入一个大于 0 并且小于 256 的正整数: ";
    cin >> i;
    if(i > 0&&i < 256)
    {
       ch = i;
       cout <<"正整数"<<(int)ch <<"的二进制值为:";
       printBinary(ch);
       cout << endl;
    }
    else
        cout <<"您输入的整数不在规定范围!"<< endl;
}
void printBinary(unsigned char n)      //递归函数,输出正整数 n 的二进制值
{
    if(n!= 0)
    {
        printBinary(n/2);              /* 用 n/2 为参数递归调用本函数,输出 n/2 的二进制值 */
        cout <<(int)n % 2;             //输出 n 的二进制值的最右边一位
    }
}
```

递归函数 printBinary 输出参数 n 的二进制值。递归调用的终止条件是 n＝0，当 n≠0 时，先以 n/2 为参数递归调用本函数输出 n/2 的二进制值，再输出 n 的二进制值的最右一位 n%2。例 5.10 程序的运行结果如图 5.13 所示。

```
请输入一个大于0并且小于256的正整数：66
正整数66的二进制值为:1000010
```

图 5.13 例 5.10 程序的运行结果

请读者思考，能否将递归函数 printBinary 中的语句 printBinary(n/2)；和 cout << (int)n%2；交换位置？如果将它们交换位置，将输出什么？

例 5.11 汉诺塔问题。有 A、B、C 三根柱子，在 A 柱上套着 n 个盘子，盘子的大小互不相同，且大盘在下小盘在上，如图 5.14(a)所示。现在要借助 B 柱子把这 n 个盘子从 A 柱移动到 C 柱，如图 5.14(b)所示。移动的规则是：①每次只能移动一个盘子；②移动中的任何时刻，在三个柱上的盘子必须大盘在下、小盘在上。请找出最佳的移动步骤。

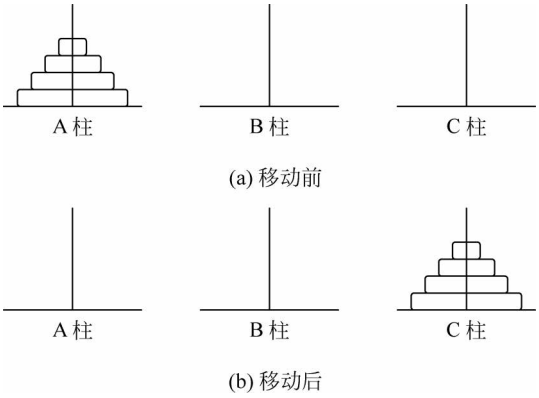

图 5.14 汉诺塔问题

分析：把 n 个盘子从 A 柱借助 B 柱移动到 C 柱可以分为以下三个步骤进行。

(1) 把 A 柱上面的(n－1)个盘子从 A 柱借助 C 柱移动到 B 柱；

(2) 把 A 柱上剩下的一个盘子移动到 C 柱；

(3) 把 B 柱上的(n－1)个盘子借助 A 柱移动到 C 柱。

以上三步采用递归的方法对问题进行了分解。而递归终止条件应该是 n 等于 1，即 1 个盘子可以从 A 柱直接移动到 C 柱。

通过上面的分析，可以容易地写出移动盘子的函数。

```
void hanoi(int n,char a,char b,char c)
{
    if(n == 1) cout << a <<" to "<< c << endl;
    else
    {
        hanoi(n - 1,a,c,b);
        cout << a <<" to "<< c << endl;
        hanoi(n - 1,b,a,c);
    }
}
```

函数的参数 n 存储盘子个数，三个字符型的参数 a、b、c 用来存储代表三个柱子的大写字母'A''B''C'；函数 hanoi 中首先判断是否满足递归调用的终止条件(n＝＝1)，如果满足条件，则把 a 柱上的一个盘子直接移动到 c 柱上，函数运行结束，返回到上次调用处继续执行；如果不满足递归终止条件，则按上面的分析分三步进行，其中包含两次递归调用。以 n＝4 为例，函数递归调用的过程如图 5.15 所示。

图 5.15 中的矩形代表一次函数调用，箭头代表函数调用的方向，箭头末端是函数调用语句，箭头指向调用的函数。图 5.15 中的标号①②…⑦代表程序输出的顺序，即移动盘子的顺序。完整的程序代码如下。

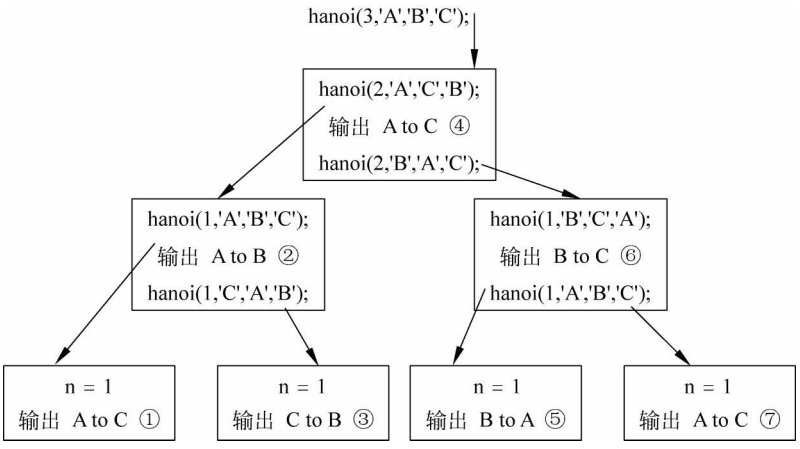

图 5.15 n=3 时函数 hanoi 的递归调用过程

```
#include<iostream>
using namespace std;
void hanoi(int n,char a,char b,char c);
void main()
{
    int n;
    cout<<"请输入盘子的个数:";
    cin>>n;
    cout<<"移动盘子的步骤如下:\n";
    hanoi(n,'A','B','C');
}
void hanoi(int n,char a,char b,char c)
{
    if(n==1) cout<<a<<" to "<<c<<endl;
    else
    {
        hanoi(n-1,a,c,b);
        cout<<a<<" to "<<c<<endl;
        hanoi(n-1,b,a,c);
    }
}
```

```
请输入盘子的个数: 4
移动盘子的步骤如下:
A to B
A to C
B to C
A to B
C to A
C to B
A to B
A to C
B to C
B to A
C to A
B to C
A to B
A to C
B to C
```

例 5.11 程序的运行结果如图 5.16 所示。

图 5.16 例 5.11 程序的运行结果

5.7 参数的默认值

可以在定义或声明函数时,为其参数指定默认值。则在调用函数时,可以传递参数,也可以不传递参数;如果没有传递参数,则在函数中使用参数的默认值。

例 5.12 参数默认值。

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

```
const float PI = (float)3.14159;
float area(float radius = 1.0);
void main()
{
    float radius;
    cout <<"请输入圆的半径: ";
    cin >> radius;
    if(radius == 0.0)
        cout <<"半径为 1 的圆的面积为: "<< area()<< endl;
    else
        cout <<"半径为"<< radius <<"的圆的面积为: "<< area(radius)<< endl;
}
float area(float radius)
{
    return PI * radius * radius;
}
```

程序中声明函数 area 时指定其参数 radius 的默认值是 1.0。main 函数中提示用户输入圆的半径,如果用户的输入值为 0,则调用函数 area 时不传递实参,这时函数 area 使用参数 radius 的默认值。程序运行结果如图 5.17 所示。

```
请输入圆的半径：0
半径为1的圆的面积为：3.14159
```

图 5.17 例 5.12 程序的运行结果

注意:必须按照从右向左的顺序为函数的参数声明默认值,即在具有默认值的参数的右边不能有不带默认值的参数。这是因为 C++编译器在编译函数调用时,总是按从左向右的顺序匹配实参和形参。例如,下边的函数声明语句是错误的。

```
void fun1(int val1 = 1, int val2);
void fun2(int val1 = 1,int val2,int val3 = 3);
```

5.8 指针函数和函数指针

5.8.1 指针函数

如果一个函数的返回值是指针类型,则称为指针函数。定义指针函数的语法为:

```
数据类型 * 函数名(形参列表)
{
  函数体;
}
```

用指针作为函数的返回值的好处是:可以从被调函数向主调函数返回大量的数据。

需要注意:**不要把函数内在“栈”中创建的局部变量的指针作为函数的返回值**。因为随着函数运行结束,在“栈”中创建局部变量都将被自动销毁,存储它们的内存空间被系统回收以作他用。如果将指向这些局部变量的指针返回主调函数,则该指针就指向了一块未知的

内存空间,对该指针的操作将导致严重的后果。

例 5.13 不要把“栈”中局部变量的指针作为函数返回值。

```
#include <iostream>
using namespace std;
int *fun();
void main()
{
    int *ptr;
    ptr = fun();
    cout <<"*ptr = "<<*ptr<<endl;
}
int *fun()
{
    int varinStack;              //在"栈"中创建变量
    int *ptr = &varinStack;
    *ptr = 5;
    cout <<"*ptr = "<<*ptr<<endl;
    return ptr;
}
```

这段程序虽然可以正常地编译、运行,但却存在严重的逻辑错误。因为变量 varinStack 是一个在“栈”中定义的函数局部变量,函数 fun 运行结束后,变量 varinStack 其实已不存在,而它的指针却被返回给 main 函数,在 main 函数中操作该指针容易引发严重的后果。读者可以运行程序并观察运行结果。

虽然不能返回“栈”中的局部变量的指针,但是可以把函数中使用 new 操作符在“堆”中动态创建的变量的指针作为函数的返回值。因为在“堆”中创建的变量只要不使用 delete 操作符释放其存储空间,它就一直存在。

例 5.14 把“堆”中变量的指针作为函数返回值。

```
#include <iostream>
using namespace std;
int *fun();
void main()
{
    int *ptr;
    ptr = fun();
    cout <<"*ptr = "<<*ptr<<endl;
    delete ptr;
}
int *fun()
{
    int *ptr = new int;          //在"堆"中动态创建变量
    *ptr = 5;
    cout <<"*ptr = "<<*ptr<<endl;
    return ptr;
}
```

本例中的程序对例 5.13 中的程序稍做修改,函数 fun 中采用 new 操作符在“堆”中动态

创建了一个整型变量，并将该变量的指针作为函数的返回值。例 5.14 程序的运行结果如图 5.18 所示。

```
*ptr= 5
*ptr= 5
```

图 5.18 例 5.14 程序的运行结果

那么能否把函数的静态局部变量的指针作为指针函数的返回值呢？请读者思考。

5.8.2 函数指针

指针不仅可以指向变量，还可以指向函数，**指向函数的指针称为函数指针**。定义函数指针的语法如下：

```
数据类型(*指针名)(形参列表);
```

其中，数据类型代表指针所指函数的返回值类型，形参列表是指针所指函数的形参列表。例如：

```
int (*fptr)(int,int);
```

上边的语句定义了一个函数指针 fptr，它可以指向带两个整型参数且返回值类型为整型的任意函数。

定义了函数指针后，就可以为它赋值，使它指向某个特定的函数，给函数指针赋值的语法如下：

```
函数指针名 = 函数名;
```

因为函数名代表存储函数的内存地址，所以给函数指针赋值时不需要取地址运算符 &。

函数指针指向某个函数后，就可以像使用函数名一样使用函数指针来调用函数了。

例 5.15 使用函数指针调用函数。

```
#include <iostream>
using namespace std;
float areaofRectangle(float width,float height);
float areaofTriangle(float heml,float height);
void main()
{
    float (*fptr)(float,float);                    //定义函数指针 fptr
    float area,worh,height;
    cout<<"请输入矩形的宽和高: ";
    cin>>worh>>height;
    fptr = areaofRectangle;                        //指向函数 areaofRectangle
    area = fptr(worh,height);                      //使用函数指针调用函数 areaofRectangle
    cout<<"矩形面积为: "<<area<<endl;
    cout<<"请输入三角形的底和高: ";
    cin>>worh>>height;
    fptr = areaofTriangle;                         //指向函数 areaofTriangle
    area = fptr(worh,height);                      //使用函数指针调用函数 areaofTriangle
    cout<<"三角形面积为: "<<area<<endl;
}
float areaofRectangle(float width,float height)
```

```
{
    return width * height;
}
float areaofTriangle(float heml,float height)
{
    return (heml * height)/2;
}
```

例 5.15 程序运行结果如图 5.19 所示。

```
请输入矩形的宽和高：10 5
矩形面积为：50
请输入三角形的底和高：10 5
三角形面积为：25
```

图 5.19 例 5.15 程序的运行结果

5.8.3 用函数指针作为函数的参数

可以把函数指针作为参数传递给另一个函数，这提供了一种把算法作为参数传递的有效途径。

现在假设要为某个企业设计一个计算员工薪水的程序，企业中有 3 类不同的员工：工人、销售人员和研发人员。计算他们薪水的方法各不相同，工人按工时数计算薪水，每个工时的薪水是 50 元；销售人员的薪水由两部分组成，第一部分是基本工资 2000 元，第二部分是销售提成，按他们销售的每件产品提成 10 元来计算；研发人员的薪水也由两部分组成，第一部分是基本工资 3000 元，第二部分是按工时数发的奖金，每工时奖励 15 元。可以设计三个函数分别计算三种员工的薪水，函数原型如下。

```
double forSale(int num);          //计算销售人员的薪水
double forWork(int num);          //计算工人的薪水
double forSc(int num);            //计算研发人员的薪水
```

注意：虽然以上三个函数各自的实现算法不同，但它们的原型是一致的——都有一个 int 型参数，返回值都是 double 型的值。这就为设计算法提供了方便。可以设计一个通用的函数 calculate 来计算员工的薪水，该函数的一个参数是函数指针。在计算不同类型的员工薪水时，可以通过函数指针把不同的计算薪水的函数传递给 calculate，在函数 calculate 内部再通过函数指针去调用计算函数。这样做的好处是：函数 calculate 对于企业中不同类型的员工是通用的，当企业中增加了新的类型的员工时，只需为这种新类型的员工编写计算薪水的函数，而类似函数 calculate 这种通用的代码是完全不需要修改的。函数 calculate 的原型如下：

```
double calculate(int num, double( * ptf)(int));
```

函数 calculate 的第二个参数是一个函数指针，第一个参数是传递给通过该指针所调用的函数的参数。完整的程序如例 5.16 所示。

例 5.16 用函数指针作为函数的参数。

```
#include < iostream >
```

```
using namespace std;
double forSale(int num)
{
    double salary;
    salary = 10.0 * num + 2000;
    return salary;
}
double forWork(int num)
{
    double salary;
    salary = 50.0 * num;
    return salary;
}
double forSc(int num)
{
    double salary;
    salary = num * 15.0 + 3000;
    return salary;
}
double calculate(int num, double( * ptf)(int))
{
    return ptf(num);
}
void main()
{
    double salary;
    int num,choice;
    double( * ptf)(int num);
    while (true)
    {
        cout << "1.销售人员\n";
        cout << "2.工人\n";
        cout << "3.研发人员\n";
        cout << "0.退出程序\n";
        cout << "请输入您的选择: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            cout << "请输入销售员销售的产品个数: ";
            cin >> num;
            ptf = forSale;
            salary = calculate(num, ptf);
            cout << "您的薪水是: " << salary << endl;
            break;
        case 2:
            cout << "请输入工人的工作时数: ";
            cin >> num;
            ptf = forWork;
            salary = calculate(num, ptf);
            cout << "您的薪水是: " << salary << endl;
```

```
            break;
        case 3:
            cout << "请输入研发人员的工作时数: ";
            cin >> num;
            ptf = forSc;
            salary = calculate(num, ptf);
            cout << "您的薪水是: " << salary << endl;
            break;
        case 0:
            cout << "退出程序!\n";
            break;
        }
        if (choice == 0)
            break;
    }
}
```

程序主函数 main 用一个 while 循环输出一个菜单,程序的用户根据自己的类型做出选择。例如,如果用户是工人类型的员工,则输入整数 2;如果是销售人员,则输入 1;如果要退出程序,则输入 0。程序根据用户的输入,使用函数指针 ptf 指向不同的计算工资函数,然后再把该函数指针作为参数传递给函数 calculate。函数 calculate 使用传入的函数指针和 int 型参数 num 调用不同的函数来计算不同类型员工的薪水。例 5.16 程序的运行结果如图 5.20 所示。

```
1.销售人员
2.工人
3.研发人员
0.退出程序
请输入您的选择: 2
请输入工人的工作时数: 160
您的薪水是: 8000
1.销售人员
2.工人
3.研发人员
0.退出程序
请输入您的选择: 1
请输入销售员销售的产品个数: 500
您的薪水是: 7000
1.销售人员
2.工人
3.研发人员
0.退出程序
请输入您的选择: 0
退出程序!
```

图 5.20 例 5.16 程序的运行结果

5.9 函数重载

在例 5.1 中定义了一个函数 max,用于比较两个 int 型数的大小。现在如果要创建一组函数,分别找出两个 short 型整数、两个 long 型整数、两个 float 型实数、两个 double 型实数、三个 int 型数、三个 float 型实数、……、中的最大值。可以看出,它们是一组功能相近的函数。为了区分它们,可以给它们取各不相同的函数名,例如,比较两个 int 型数的函数取名为 imax2、比较三个 int 型数的函数取名为 imax3、比较两个 float 型数的函数取名为 fmax2、……但这样做会给编程带来不便,因为编程者必须记忆大量的函数名。解决问题的方法是**函数重载**。

C++允许在相同的作用域中定义函数名相同但参数形式不同的多个函数,参数形式不同是指:要么参数的类型不同,要么参数的个数不同。这样的编程技术称为函数重载,这组同名的函数称为重载的函数。例如,下面是三个重载的 max 函数,分别找出两个 int 型数、两个 float 型数和三个 int 型数中最大的一个。

```
int max(int num1,int num2)
{
    return (num1 >= num2)? num1:num2;
```

```
}
float max(float num1,float num2)
{
    return (num1 >= num2)? num1:num2;
}
int max(int num1,int num2,int num3)
{
    return (max(num1,num2)>= num3)? max(num1,num2):num3;
}
```

调用重载函数时,C++编译器根据实参的类型和实参的个数找到匹配的重载函数进行调用。例如:

```
float fmax,f1,f2;
…
fm = max(f1,f2);
```

上边的语句 fm=max(f1,f2);调用重载函数 max,由于 f1 和 f2 是两个 float 型变量,所以编译器选择函数 float max(float num1,float num2)进行调用。

使用重载函数时应注意以下几点。

(1) 重载的函数应具有类似的功能,例如,上边定义的一组重载函数 max 都返回两个或三个数中较大的一个。如果两个函数的功能区别很大,则不应定义为重载函数。

(2) 只能以参数的类型来重载函数,而不能以函数返回值的类型来重载函数。例如,若按下面的方式重载 max 函数,则会产生编译错误。

```
int max(int num1,int num2);
float max(int num1,int num2);
```

(3) 不能用形参的名字来重载函数。例如,下边的重载声明也是错误的。

```
int max(int num1,int num2);
int max(int n1,int n2);
```

(4) 如果形参为引用类型或指针类型,则可以使用关键字 const 来重载函数。例如,下边的重载声明是正确的。

```
int max(int &num1,int &num2);
int max(const int &num1,const int &num2);
```

原因请读者思考。

例 5.17 使用重载函数。

```
#include < iostream >
using namespace std;
int max(int num1,int num2);
float max(float num1,float num2);
int max(int num1,int num2,int num3);
void main()
{
    int i1,i2,i3;
```

```
    float f1,f2;
    cout <<"请输入两个整数: ";
    cin >> i1 >> i2;
    cout <<"两个整数中较大的一个是: "<< max(i1,i2)<< endl;
    cout <<"请输入两个实数: ";
    cin >> f1 >> f2;
    cout <<"两个实数中较大的一个是: "<< max(f1,f2)<< endl;
    cout <<"请输入三个整数: ";
    cin >> i1 >> i2 >> i3;
    cout <<"三个整数中最大的一个是: "<< max(i1,i2,i3)<< endl;
}
int max(int num1,int num2)
{
    return (num1 > = num2)? num1:num2;
}
float max(float num1,float num2)
{
    return (num1 > = num2)? num1:num2;
}
int max(int num1,int num2,int num3)
{
    return (max(num1,num2)> = num3)? max(num1,num2):num3;
}
```

例 5.17 程序的运行结果如图 5.21 所示。

```
请输入两个整数: 34 564
两个整数中较大的一个是: 564
请输入两个实数: 100.12 56.34
两个实数中较大的一个是: 100.12
请输入三个整数: 4567 45 569
三个整数中最大的一个是: 4567
```

图 5.21 例 5.17 程序的运行结果

5.10 函数模板

5.9 节中学习了函数重载技术,以下利用重载技术定义几个函数。

```
int max(int num1,int num2)     //返回两个 int 型参数中较大的一个
{
   int themax;
   themax = (num1 > = num2)? num1: num2;
   return themax;
}
float max(float num1,float num2)          //返回两个 float 型参数中较大的一个
{
   float themax;
   themax = (num1 > = num2)? num1: num2;
   return themax;
}
double max(double num1,double num2)       //返回两个 double 型参数中较大的一个
```

```
{
   double themax;
   themax = (num1 >= num2)? num1: num2;
   return themax;
}
```

仔细观察这组重载函数，不难发现，函数中除了函数返回值、形参和局部变量的数据类型不同外，其他的语句内容完全相同。也就是说，这组函数是将相同的算法应用于不同的数据类型。但在编程时，却不得不给出每一个函数的实现，把相同的语句书写了多遍，降低了编程的效率。那么能否用一个函数来取代这一组函数呢？回答是能。C++提供了函数模板技术来解决这类问题。

函数模板就是将具体函数中的数据类型参数化，即用通用的参数取代函数中具体的数据类型，从而形成一个通用模板来代表数据类型不同的一组函数。定义函数模板的语法如下：

```
template < class T1,class T2,…,class Tn >
返回值类型 函数名(用模板参数取代具体类型的形参列表)
{
  用模板参数取代具体数据类型的函数体;
}
```

其中，template 和 class 是定义模板函数时必须使用的 C++关键字，class 也可以用关键字 typename 代替。template 后的尖括号称为模板参数列表；其中的 T1，T2，…，Tn 称为模板参数——参数化的数据类型。函数的返回值类型、形参的数据类型和局部变量的数据类型可以是具体数据类型，也可以是模板参数类型。每一个模板参数在函数定义中应至少使用一次。

现在在程序中，就可以用下面的函数模板取代上面的一组 max 函数。

```
template < class T >
T max(T num1, T num2)
{
   T themax;
   themax = (num1 >= num2)? num1: num2;
   return themax;
}
```

在这个函数模板中，用模板参数 T 取代具体数据类型创建了一个通用的函数模板。

可以看到，一个函数模板可以代表一组函数，这就极大地增强了程序代码的重用性，提高了编程效率。

那么，函数模板是怎样转换成具体函数的呢？结合下面的例子来进行说明。

例 5.18 定义和使用函数模板。

```
#include < iostream >
using namespace std;
template < class T >
T max(T num1, T num2);          //声明函数模板
void main()
```

```
{
    int i1,i2,imax;
    float f1,f2,fmax;
    cout <<"请输入两个整数: ";
    cin >> i1 >> i2;
    imax = max(i1,i2);
    cout <<"两个整数中较大的一个是: "<< imax << endl;
    cout <<"请输入两个实数: ";
    cin >> f1 >> f2;
    fmax = max(f1,f2);
    cout <<"两个实数中较大的一个是: "<< fmax << endl;
}
template < class T >
T max(T num1, T num2)
{
   T themax;
   themax = (num1 > = num2)? num1: num2;
   return themax;
}
```

C++编译器在编译函数调用语句时,用实参的数据类型取代函数模板中的模板参数创建一个具体的函数,并对其进行编译。例如,在上面的程序中,当编译器编译语句 imax=max(i1,i2);时,就用实参 i1 和 i2 的数据类型——int 取代函数模板中的参数 T,创建一个具体的函数用来返回两个整型参数中较大的一个:

```
int max(int num1,int num2)    {
  int themax;
  themax = (num1 > = num2)? num1: num2;
  return themax;
}
```

编译器编译这个函数并实现调用。当编译语句 fmax=max(f1,f2);时,由于实参 f1 和 f2 为 float 型变量,所以编译器用 float 取代函数模板中的模板参数 T,再次创建了一个具体的函数来返回两个 float 型参数中较大的一个:

```
float max(float num1,float num2)  {
  float themax;
  themax = (num1 > = num2)? num1: num2;
  return themax;
}
```

编译器编译这个函数并实现本次调用。

例 5.18 程序的运行结果如图 5.22 所示。

```
请输入两个整数: 100 200
两个整数中较大的一个是: 200
请输入两个实数: 43.76 150.34
两个实数中较大的一个是: 150.34
```

图 5.22 例 5.18 程序的运行结果

5.11 lambda 函数

lambda 函数又称为 lambda 表达式，是 C++ 11/C++ 14 引入的一种新的元素。它本质上是一种匿名函数。使用 lambda 函数的好处是可以用简洁的程序语句完成强大的函数功能。

5.11.1 定义和使用 lambda 函数

定义一个 lambda 函数的语法形式如下：

```
[捕获列表](参数列表)->返回值类型{函数体;};
```

其中：

(1) 捕获列表：是一个可能为空的列表，作用是指明定义 lambda 函数的作用域中的哪些对象(包括变量、常量等)可以在 lambda 函数中使用，以及它们在 lambda 函数中的存在方式是值的拷贝还是变量的引用。

(2) 参数列表：是 lambda 函数的参数列表，形式和普通函数的参数列表相同。如果一个 lambda 函数没有参数，则小括号中为空，也可以省略小括号。

(3) —>返回值类型：lambda 函数定义中的这部分内容表示函数的返回值类型，是可以省略的。因为编译器可以根据函数的内容自动推定函数的返回值类型。

(4) {函数体;}：花括号中是 lambda 函数的函数体。

例如：

```
auto f1 = [](int i,int j)->int{ return i>j? i:j;};
auto f2 = [](int i,int j){ return i>j? i:j;};
```

上面两条语句定义了两个 lambda 函数 f1 和 f2，它们的功能是相同的，都是返回两个整型参数中较大的一个。必须使用关键字 auto 来定义 lambda 函数。

这样定义之后，可以使用下面的语句来调用它们。

```
int max=f1(10,15);
int max=f2(10,15);
```

例 5.19 定义一个 lambda 函数，返回两个整型参数中较大的一个。

```
#include<iostream>
using namespace std;
void main()
{
    int max, one, tow;
    cout << "请输入两个整数: ";
    cin >> one >> tow;
    auto f = [](int i, int j)->int { return i > j ? i : j;};
    max = f(one, tow);
    cout << "两个整数中比较大的是" << max << endl;
}
```

上面程序中先定义了一个 lambda 函数 f,然后用两个整型变量作为参数调用该函数。例 5.19 程序的运行结果如图 5.23 所示。

```
请输入两个整数: 10 20
两个整数中比较大的是20
```

图 5.23 例 5.19 程序的运行结果

5.11.2 lambda 函数的捕获列表

定义 lambda 函数时,方括号[]中的内容称为捕获列表,它的作用是确定 lambda 函数所在的作用域中的哪些对象(包括变量或常量)能在 lambda 函数中使用,以及它们在 lambda 函数中是值的拷贝还是变量的引用。以下是 lambda 函数捕获列表的使用方式。

(1) []: 捕获列表为空,意味着在 lambda 函数中不能使用其外层作用域中的任何对象。

(2) [&]: 表示在 lambda 函数中可以以引用的方式使用其外层作用域中的所有局部对象。

(3) [=]: 表示在 lambda 函数中可以以值的拷贝的方式使用其外层作用域中的所有局部对象。

(4) [a,&b]: 显式地捕获 a 和 b,表示在 lambda 函数中可以以值的拷贝(传值)的方式使用其外层作用域中的对象(变量)a,可以以引用的方式使用其外层作用域中的对象(变量)b。

(5) [this]: 表示在 lambda 函数中可以以值传递的方式使用 this 指针。关于 this 指针的相关内容,将在后续章节中介绍。

(6) [&,a,b]: 表示在 lambda 函数中可以以值传递的方式使用其外层作用域中的变量 a 和 b,以引用的方式使用其外层作用域中其他所有对象。

(7) [=,&a,&b,&c]: 表示在 lambda 函数中可以以引用的方式使用其外层作用域中的变量 a、b 和 c,以传值的方式使用其外层作用域中其他所有对象。

例如,对于例 5.19 中的程序,可按如下几种方式对定义和调用 lambda 函数的两条语句进行修改,修改后程序的功能不变。

(1)
```
auto f = [&]() {return one > tow ? one : tow;};
max = f();
```

(2)
```
auto f = [ = ]() {return one > tow ? one : tow;};
max = f();
```

(3)
```
auto f = [&one, tow]() {return one > tow ? one : tow;};
max = f();
```

第(1)种修改方式的捕获列表为 &,表示在 lambda 函数中可以以引用的方式访问其外层作用域中定义的局部变量 one 和 tow。

第(2)种修改方式的捕获列表为=,表示在 lambda 函数中可以以值的拷贝的方式访问其外层作用域中定义的局部变量 one 和 tow。即在 lambda 函数中存在变量 one 和 tow 的值的拷贝。

第(3)种修改方式的捕获列表为[&one, tow],它显式地指出,在 lambda 函数中可以以引用的方式访问其外层作用域中定义的局部变量 one,且 lambda 函数中存在其外层作用域

中定义的局部变量 tow 的拷贝。

上面这三种修改方式都使得可以不用为 lambda 函数传递参数，故 lambda 函数的参数列表为空。例 5.20 是使用上述第(2)种方式对例 5.19 的程序所做的修改。

例 5.20 使用 lambda 函数的捕获列表。

```
#include <iostream>
using namespace std;
void main()
{
    int max, one, tow;
    cout << "请输入两个整数: ";
    cin >> one >> tow;
    auto f = [ = ]() {return one > tow ? one : tow;};
    max = f();
    cout << "两个整数中比较大的是" << max << endl ;
}
```

5.11.3 lambda 函数作为函数的参数

lambda 函数本身可以作为其他函数的参数，这种功能类似于函数指针(5.8.3 节)。但是用 lambda 函数实现比用函数指针更加简洁，效率更高。

使用 lambda 函数作函数参数的一个关键问题是：被调函数的参数形式。即什么类型的形式参数可以接收一个作为实参的 lambda 函数。常用的解决方法有以下两种。

(1) 把被调函数设计成模板函数，用模板参数作为形式参数接收作为实参的 lambda 函数。这种方法就是让编译器去自行推定参数类型，比较简单。

对于例 5.16 中的计算员工薪水的程序，可做下述修改。

首先把例 5.16 程序中的通用函数 calculate 设计成一个模板函数。在原来的程序中，它的一个形参是函数指针。下面是修改前和修改后的 calculate 函数的代码。

修改前的 calculate 函数代码：

```
double calculate(int num, double( * ptf)(int))
{
    return ptf(num);
}
```

修改后的 calculate 函数代码：

```
template < typename F >
double calculate(int num, F f)
{
    return f(num);
}
```

下一步把原程序中分别计算三种不同类型员工薪水的三个函数 forSale、forWork 和 forSc 设计成三个 lambda 函数，并把它们作为参数直接传递给函数 calculate。

修改后程序的完整代码如例 5.21 所示。

例 5.21 修改例 5.16 中的程序,使用 lambda 作为参数,实现计算员工薪水的功能。

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

template<typename F>
double calculate(int num, F f)        //.....(1)
{
    return f(num);                    //..... (2)
}
void main()
{
    double salary;
    int num, choice;
    double(*ptf)(int num);
    while (true)
    {
        cout << "1.销售人员\n";
        cout << "2.工人\n";
        cout << "3.研发人员\n";
        cout << "0.退出程序\n";
        cout << "请输入您的选择: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            cout << "请输入销售员销售的产品个数: ";
            cin >> num;
            salary = calculate(num, [](int n) ->double { return 10.0*n + 2000; });//....(3)
            cout << "您的薪水是: " << salary << endl;
            break;
        case 2:
            cout << "请输入工人的工作时数: ";
            cin >> num;
            salary = calculate(num, [](int n) ->double { return 50.0*n;}); //.....(4)
            cout << "您的薪水是: " << salary << endl;
            break;
        case 3:
            cout << "请输入研发人员的工作时数: ";
            cin >> num;
            salary = calculate(num, [](int n) { return n * 15.0 + 3000; }); //.....(5)
            cout << "您的薪水是: " << salary << endl;
            break;
        case 0:
            cout << "退出程序!\n";
            break;
        }
        if (choice == 0)
            break;
    }
}
```

上面程序中注释为(3)(4)和(5)的三条语句用 lambda 函数作为实参,把它们传递给注释为(1)的模板函数 calculate。函数 calculate 使用参数 f 接收 lambda 函数,并调用它们计算不同类型员工的工资,再把计算结果作为函数的返回值返回给 main 函数。调用 lambda 函数的语句注释为(2)。程序的功能和例 5.16 中程序的功能完全相同。但和原来的程序相比,修改后的程序更加简洁,执行效率更高。

(2) 第二种在函数中接收作为实参的 lambda 函数的方法是使用 C++ 11 新引进的 function 类。function 类是一个模板类,本书到目前为止还没有涉及有关类的内容,在此读者只需了解怎样使用 function 类的对象引用 lambda 函数和作为函数的形参接收 lambda 函数,以及使用 function 类的对象调用 lambda 函数的方法。有关模板类的相关内容将会在后续章节介绍。

类是由程序员自己定义的一种抽象数据类型,类的对象相当于这种类型的一个变量。类和对象的关系类似于结构体类型和结构体变量之间的关系。

C++ 11 新引入了一个类 function,它是一个在名字空间 std 中定义的模板类。该类的对象称为可调用对象,一个可调用对象可以被绑定到任意一个特定的函数,只要这个函数的声明形式符合可调用对象的声明形式,通过该对象即可调用它绑定到的函数。例如:

```
function<double(int)>  f1;
```

上面的语句定义了一个 function 类的对象 f1,该对象可以被绑定到的函数都必须具有以下的形式:函数的返回值为 double 类型,函数有一个 int 型的形式参数。再如:

```
function<void(double,int)>  f2;
```

上面的语句定义了一个 function 类的对象 f2,该对象可以被绑定到的函数都必须具有以下的形式:函数没有返回值(void 型),函数有两个形式参数,第一个形参的类型为 double,第二个形参的类型为 int。

定义了 function 对象之后,就可以把该对象绑定到某个特定的函数,假设程序中声明了两个函数 fun1 和 fun2,下面是这两个函数的原型声明:

```
double fun1(int);
void fun2(double,int);
```

则可以把前面定义的 function 类的对象 f1 和 f2 绑定到这两个函数:

```
f1 = fun1;
f2 = fun2;
```

接下来,就可以使用对象 f1 和 f2 去调用函数 fun1 和 fun2 了:

```
double re = f1(100);
f2(12.34,100);
```

可以将 function 类的对象绑定到一个 lambda 函数,然后再使用该对象调用该 lambda 函数。例如:

```
f1 = [](int n) -> double{ return 10.5 * n;};
double lr = f1(10);
```

function 类的对象还可以作为函数的形参接收作为实参的 lambda 函数。

例 5.22 修改例 5.21 中的程序,使用 function 类的对象作为函数的形式参数接收作为实参的 lambda 函数。

本程序只需对例 5.21 中的程序稍做修改,在例 5.21 程序中,函数 calculate 被定义为一个模板函数,其第二个参数的数据类型由模板参数指定。在本例程序中,函数 calculate 不是模板函数,且它的第二个参数是一个 function 类的对象,用来接收从主调函数传递来的 lambda 函数,并在函数 calculate 中,使用这个 function 类的对象去调用作为实参的 lambda 函数。完整的程序代码如下。

```
#include<iostream>
#include<functional>
using namespace std;
double calculate(int num, function<double(int)> f)  //.....(1)
{
    return f(num);            //.....(2)
}
void main()
{
    double salary;
    int num, choice;
    while (true)
    {
        cout << "1.销售人员\n";
        cout << "2.工人\n";
        cout << "3.研发人员\n";
        cout << "0.退出程序\n";
        cout << "请输入您的选择: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            cout << "请输入销售员销售的产品个数: ";
            cin >> num;
            salary = calculate(num, [](int n) ->double { return 10.0 * n + 2000; }); //....(3)
            cout << "您的薪水是: " << salary << endl;
            break;
        case 2:
            cout << "请输入工人的工作时数: ";
            cin >> num;
            salary = calculate(num, [](int n) ->double { return 50.0 * n;}); //.....(4)
            cout << "您的薪水是: " << salary << endl;
            break;
        case 3:
            cout << "请输入研发人员的工作时数: ";
            cin >> num;
            salary = calculate(num, [](int n) { return n * 15.0 + 3000; }); //.....(5)
            cout << "您的薪水是: " << salary << endl;
            break;
        case 0:
```

```
            cout << "退出程序!\n";
            break;
        }
        if (choice == 0)
            break;
    }
}
```

和例 5.21 中的程序相比,本例中的程序只做了两点细微的修改。

(1) 把函数 calculate 从模板函数修改为普通函数。

(2) 例 5.21 程序中,函数 calculate 的第二个形式参数的数据类型是模板参数 F,而本例的程序中,函数 calculate 的第二个形式参数是 function 类的一个对象。

例 5.16、例 5.21 和例 5.22 这三个程序使用三种不同的技术,实现完全相同的功能。例 5.16 使用函数指针来传递函数型参数,并实现函数回调;例 5.21 使用模板函数结合 lambda 函数来实现同样的功能;而例 5.22 则使用 lambda 函数结合 function 类来实现。相比于函数指针,后两种方法更加简单明了,逻辑清晰。

5.12 具有可变长参数的函数

C++ 11 引入了定义和调用具有可变长参数函数的方法,这种函数可以接收可变数量的参数,且不同参数的数据类型也可以不同。C++ 11 使用模板函数来定义这种具有可变长参数的函数,为了定义这种函数,C++ 11 还引入了一个新的操作符和两个新的程序元素。下面分别介绍。

(1) 元操作符:元操作符用一个省略号…来表示,表示具有 0 或多个参数的参数包。

(2) 模板参数包:是由 0 个或多个模板参数组成的参数包。

(3) 函数参数包:是由 0 个或多个参数组成的参数包。

C++ 11 用函数参数包来表示可变长的参数,用模板参数包表示这些参数的相应的数据类型。例如:

```
template < typename… Args >
void display(Args… args)
{
  //函数要显示每个参数的值,但是怎么得到这些参数呢?
}
```

上面的程序语句定义了一个模板函数,该函数的所有参数的数据类型都被包含在模板参数包 Args 中,而该函数的所有参数都包含在函数参数包 args 中。该函数就是一个具有可变长参数的模板函数。

定义了具有可变长参数的函数后,面临的一个问题是:在函数内部怎样展开函数参数包,并获取其中的每个参数呢?

函数递归调用是解决这个问题的一个有效方法。可把上面的函数定义形式稍做变形,得到下面的定义形式。

```
template < typename T, typename… Args >
```

```
void display(T t, typename… args)
{
   cout << t << ","
   display(args…);
}
```

上面的语句同样定义了一个具有可变长参数的模板函数,和前面不同的是,这个函数在声明模板参数时把表示第一个参数(最左边的参数)的数据类型的模板参数 T 从模板参数包 Args 中分离出来,单独声明;同时,在函数的参数列表中,也把函数的第一个参数 t 从函数参数包 args 中分离出来,单独声明。这样定义的好处是,在函数内部,可以首先处理函数的第一个参数(最左边的参数),本例中是输出它的值;然后再用剩下的参数作为一个完整的参数包对本函数进行递归调用;下次递归调用又会处理剩下的参数中最左边的一个,依此顺序不断地进行递归调用,直到参数包中的参数个数为 1 个,这时程序要做的是输出这个参数的值,然后结束递归调用。也就是说,还需要为该函数设计一个重载的版本,这个重载函数只包含一个模板参数,当参数包中只有一个参数时,编译器就会自动调用这个重载函数,输出参数的值,然后退出函数,终止递归。程序用它作为函数递归调用的终点。下面是这个重载的模板函数的代码。

```
template < typename T >
void display(T t)
{
    cout << t << endl;
}
```

例 5.23 设计具有可变长参数的函数,输出所有参数的值。

```
# include < iostream >
using namespace std;
template < typename T,typename… Args >
void display(T t, Args… args)
{
    cout << t << ",";
    display(args…);
}
template < typename T >
void display(T t)
{
    cout << t << endl;
}
void main()
{
    display(12.5, 50, 'a', "Hello");
}
```

程序在 main 函数中使用了 4 个不同类型的参数调用了模板函数 display。例 5.23 程序的运行结果如图 5.24 所示。

例 5.24 要求设计一个具有可变长参数的函数,所有的参数都具有相同的数据类型,该函数返回这些参数的累加和。

```
12.5,50,a,Hello
```

图 5.24 例 5.23 程序的运行结果

算法思想：根据本节前面介绍的设计具有可变长参数函数的方法可知：

(1) 本函数应该是一个模板函数。

(2) 由于要使用可变长参数，所以在函数中要使用模板参数包和函数参数包。

(3) 为了能用递归调用的方法展开函数参数包，函数的模板参数列表和函数参数列表都应由两个部分组成：第一部分是调用函数时，传递给函数的最左边的参数；第二部分是代表可变长参数的模板参数包和函数参数包。

(4) 所有参数的累加和＝第一个参数(最左边的参数)＋其他参数的累加和。

(5) 当调用函数的参数个数只剩一个的时候，参数的累加和就是该参数本身。

根据以上 5 步分析，可以设计出如下的两个重载函数。

第一个重载函数：用参数包表示可变长参数的递归函数 sum，代码如下。

```
template < typename T,typename… Args >
T sum(T t, Args… args)
{
    return t + sum(args…); //对自己的递归调用,累加和 = 第一个参数 + 其他参数的累加和
}
```

第二个重载函数：是只有一个模板参数的重载函数 sum，它是第一个函数递归调用的终点。代码如下。

```
T sum(T t)
{
    return t;
}
```

下面是程序的完整代码。

```
#include < iostream >
using namespace std;
template < typename T,typename… Args >
T sum(T t, Args… args)
{
    return t + sum(args…);
}
template < typename T >
T sum(T t)
{
    return t;
}
void main()
{
    int sui;
    sui = sum(1, 2, 3);
    cout <<"三个整数的和为: "<< sui << endl;
    sui = sum(1, 2, 3, 4, 5);
```

```
    cout << "五个整数的和为: "<< sui << endl;
    double sux;
    sux = sum(2.3, 1.4, 5.0);
    cout << "三个浮点数的和为: "<< sux << endl << endl;
}
```

程序在 main 函数中分别用 3 个整数、5 个整数和 3 个浮点数作为参数调用了重载函数 sum。例 5.24 程序的运行结果如图 5.25 所示。

```
三个整数的和为：6
五个整数的和为：15
三个浮点数的和为：8.7
```

图 5.25 例 5.24 程序的运行结果

小结

函数是一组语句的集合,用来完成某种特定功能。使用函数的根本目的是代码重用。5.1 节介绍了定义和调用函数的方法。

调用函数时可以同时向函数传递参数。C++传递参数的方式有两种：传值传递和引用传递。在使用传值传递时,实参把值复制给形参,形参是实参的一个副本；使用引用传递时,形参是实参的别名。

在函数或复合语句中定义的变量称为局部变量。局部变量包括自动局部变量和静态局部变量(定义时使用关键字 static),自动局部变量具有局部作用域和动态生存期；静态局部变量具有局部作用域和静态生存期。程序中在任何函数之外定义的变量叫作全局变量。全局变量具有全局作用域,它的生存期就是程序的整个运行期。

函数被调用时,系统在“栈”中为其开辟一块存储空间,存放函数的形式参数、非静态(自动)局部变量、函数调用时的现场信息和返回地址。函数运行结束时,这块空间被系统销毁。

关键字 inline 用来声明内联函数。编译器处理内联函数调用的方法是：将函数体中的语句直接嵌入到函数调用的地方,而不进行常规的函数调用。使用内联函数既可以达到代码重用的目的,也可以避免函数调用带来的额外开销。但是不能把较长的函数声明为内联函数,因为这样做容易使目标程序的体积急剧增大。

如果一个函数直接或间接地调用了它自己,则称这样的函数为递归函数。设计递归函数时,首先要给出问题的递归定义,然后再推导出相应的递归函数。

可以给函数的形参指定默认值。对于指定了默认值的形参,如果函数调用时没有给该形参传递实参,则使用默认值作为该参数的值。应按照从右向左的顺序指定参数的默认值。

如果一个函数的返回值为指针类型,则称函数为指针函数。

可以定义指向函数的指针,并通过函数指针调用所指向的函数。

函数重载就是在同一个作用域中定义名字相同但参数形式不同的多个函数。参数的形式不同是指参数的类型或个数不同。调用重载函数时,编译器根据实参的类型和个数选择匹配的函数加以调用。

关键字 template 用于创建函数模板。函数模板就是将函数中使用的数据类型参数化,

以建立一个通用的函数。一个函数模板代表一组函数，这组函数是将相同的算法应用于不同的数据类型。调用函数模板时，编译器会根据实参的数据类型创建具体的函数并加以调用。

习题

5.1 程序设计中使用函数的主要目的是什么？

5.2 什么是形式参数？什么是实际参数？它们有什么联系？

5.3 函数的返回值有什么作用？怎样声明没有返回值的函数？

5.4 什么是主调函数？什么是被调函数？它们有什么联系？

5.5 什么是函数原型？为什么要使用函数原型？

5.6 声明一个函数 add，实现两个整数相加，并返回它们的和。

5.7 编写程序，在 main 函数中从键盘输入两个整数，调用上题中编写的 add 函数计算并返回两个数的和。

5.8 编写一个没有返回值的函数，该函数有一个字符型的形式参数，该函数使用这个形参字符绘制一个倒三角形。例如，如果形参字符为'a'，则输出的图形如下：

```
aaaaaaaaa
 aaaaaaa
  aaaaa
   aaa
    a
```

5.9 编写程序，在 main 函数中分别从键盘输入 3 个字符，并以它们为实参分别调用上题中编写的函数，绘制 3 个倒三角形。

5.10 编写一个函数，用来计算并返回矩形的面积。该函数有两个 float 型的形参，分别表示矩形的长和宽。

5.11 编写一个函数 intfrac，该函数的功能是将一个 float 型形参切分成整数部分和小数部分。在 main 函数中从键盘输入一个 float 型变量 var，以 var 为实参调用 intfrac 函数，函数调用结束返回到 main 函数后，再分别输出变量 var 的整数部分和小数部分。

5.12 声明一个存储学生信息的结构体类型 Student，其中包含表示学生姓名、学号、性别、年龄的成员 name、num、sex 和 age。编写一个函数 InFromKeyB，用来从键盘向结构体 Student 型的变量输入一个学生的信息。在 main 函数中定义一个包含 10 个元素的 Student 型数组，并分别以每个数组元素为参数调用函数 InFromKeyB 输入学生的信息。最后在 main 函数中输出学生的信息，检查输入是否正确。

5.13 怎样声明内联函数？内联函数和非内联函数的区别是什么？

5.14 编写一个递归函数，将一个任意位数的整数的各位按逆序输出。例如，输入整数 1234，应输出字符串"4321"。

5.15 编写一个递归函数 Gcd，返回整数 X 和 Y 的最大公约数。Gcd 函数的递归定义如下：

$$Gcd(X,Y)=\begin{cases}X & (\text{如果 } Y=0)\\ Gcd(Y,X\%Y) & (\text{如果 } Y\neq 0)\end{cases}$$

5.16 编写一个函数 strcon。函数的原型如下：

```
void strcon(char * sp1, char * sp2);
```

该函数将字符串 sp2 连接到字符串 sp1 之后，连接后的字符串存放在 sp1 中。要求在 main 函数中输入和输出字符串。

5.17 编写一个函数 strbac。函数原型如下：

```
void strbac(char * sp);
```

该函数将字符串 sp 逆序排列。要求在 main 函数中输入和输出字符串。

5.18 以下声明的函数原型是否正确？为什么？

```
void fun(int v1 = 1,float v2,float v3 = 2.0);
```

5.19 找出下面函数中的错误，并进行修改。

```
int * fun()
{
    int i = 100;
    return &i;
}
```

5.20 编写程序，使用函数指针调用习题 3.6 中声明的 add 函数。

5.21 请设计 3 个重载的 add 函数，分别实现两个整数、两个单精度浮点数和两个双精度浮点数相加的运算。并在 main 函数中分别调用它们。

5.22 设计一个模板函数 add，实现两个相同类型形式参数的求和运算，并返回和数。在 main 函数中分别用两个整数、两个单精度浮点数和两个双精度浮点数作为参数调用该模板函数。