第 3 章　函　　数

C++语言有两种程序模块：函数（function）和类（class）。任何 C++的应用程序都是由各种标准库提供的模块和程序员定义的模块组装而成的。

函数是功能的抽象。所谓功能抽象，是指这个程序模块定义的操作，适用于指定数据类型的数据集。调用者只关心函数能做什么，而不需要关心它是如何做的。函数有两个重要作用：一是任务划分，即把一个复杂任务划分为若干个简单的小任务，便于分工和处理，也便于验证程序的正确性；二是软件重用，即把一些功能相同或相近的程序段，独立编写成函数，让应用程序随时调用，而不需要编写雷同的代码。

函数是程序设计的重要工具。这一章主要介绍函数的定义和调用、函数参数的传递，以及 C++程序的结构、变量和函数的作用域、条件编译等有关内容。

有关类的知识，将在第 6 章之后讨论。

3.1　函数的定义与调用

函数定义由两部分组成：函数首部和函数操作描述。函数首部是函数的接口，包括函数名、函数的参数和返回值类型。函数操作描述由函数体的语句序列实现。

使用函数称为调用函数。函数调用就是通过表达式或语句激活并执行函数代码的过程。函数调用的形式必须与函数定义的接口对应。

3.1.1　函数定义

从用户使用的角度来看，C++有两种函数：标准库函数和用户自定义的函数。

标准库函数由 C++系统定义并提供给用户使用，可以看作对语言功能的扩充。例如，fabs 函数、get 函数等都是标准库函数。

用户根据特定任务编写的函数称为自定义函数。自定义函数的形式与主函数的形式相似，一般形式为：

```
类型　函数名（[ 形式参数表 ]）
{
    语句序列
}
```

函数定义的第一行（可以分多行写）是函数首部（或称函数头），以大括号相括的语句序列为函数体。

其中，“函数名”是用户自定义标识符。“类型”是函数返回表达式的值的类型，简称为返回类型，可以是各种基本类型、结构类型或类类型。若无返回值，则使用空类型符 void。“形式参数表”是用逗号分隔的参数说明列表。省略形式参数时不能省略圆括号，它是函数的识别符号。“函数体”中的语句序列可以包含各种合法 C++语句。

形式参数表的一般形式为：

类型 参数$_1$，类型 参数$_2$，…，类型 参数$_n$

参数是函数与外部传输数据的纽带。若函数的定义省略参数表，则称为无参函数；否则称为有参函数。

无参函数表示函数不依赖外部数据，执行独立的操作。

【例 3-1】 定义一个无参函数，输出问候语句。

```
void printmessage()
{   cout << "How do you do!" << endl;   }
```

【例 3-2】 定义一个函数，求两个浮点数之中的大值。函数通过参数从外部接收两个浮点型数据，函数体中用 return 语句返回结果值。

```
double max(double x, double y)
{   if (x > y)   return x;
    else     return y;
}
```

如果一个函数没有返回表达式值，通常说这个函数没有返回值，函数返回类型用 void，即函数体内的 return 语句没带表达式，或可以省略 return 语句。函数没有返回值不等于不能接收或修改外部数据，在 3.2 节中将看到，参数是函数与外部传递数据的重要纽带。

3.1.2 函数调用

函数调用要做两件事情：指定函数地址，提供实际参数。函数名是函数的地址，实际参数提供被调用函数执行任务所需要的信息及接收被调用函数返回的信息。

函数调用的一般形式为：

函数名（[实际参数表]）

其中，“实际参数表”中的各参数用逗号分隔，实际参数与被调用函数的形式参数在个数、类型、位置上必须一一对应。

不管函数定义是否有参数或者是否有返回值，都可以用两种形式调用：函数语句或函数表达式。

（1）函数语句

函数调用可以作为一个语句。例如，在以下主函数中，用语句调用例 3-1 定义的函数：

```
int main()
{   printmessage();   }
```

（2）函数表达式

函数可以通过 return 语句返回一个结果值。如果定义了这种具有返回结果值的函数，并且调用时需要使用函数的返回值，可以用表达式形式调用函数。

例如，以下两种形式都可以调用例 3-2 定义的 max 函数：

```
m1 = max(a, b);
cout << max(m1, c) << endl;
```

3.1.3 函数原型

函数原型是 C++的重要特性之一。函数原型是函数的声明，作用是告诉编译器有关函数接口的信息：函数的名字、函数返回值的数据类型、函数的参数个数、参数类型和参数的顺序，编译器根据函数原型检查函数调用的正确性。

例如，例 3-2 定义的 max 函数原型为：

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

表示 max 函数有两个 double 类型参数，返回结果值为 double 类型。函数原型是一个声明语句，由函数首部加上分号组成。由于函数原型没有实现代码，因此不需要参数名。通常添加参数名是为了增加可读性，但编译器将忽略这些名称。例如：

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

是相同的函数原型。

【例 3-3】 定义和调用 max 函数。

```
#include<iostream>
using namespace std;
double max(double, double);                    //声明函数原型
int main()
{   double a, b, c, m1, m2;
    cout << "input a,b,c:\n";
    cin >> a >> b >> c;
    m1 = max(a, b);                            //调用函数
    m2 = max(m1, c);                           //调用函数
    cout << "Maximum = " << m2 << endl;
}
double max(double x, double y)                 //定义函数
{   if (x > y)    return x;
    else     return y;
}
```

如果函数定义出现在程序第一次调用之前，则不需要函数原型声明。这时，函数定义就具有函数原型的作用。例 3-3 程序的不需要函数原型声明的版本为：

```
#include<iostream>
using namespace std;
double max(double x, double y)                 //定义函数
{   if(x>y)     return x;
    else     return y;
}
int main()
{   double a, b, c, m1, m2;
    cout<<"input a,b,c:\n";
    cin>>a>>b>>c;
    m1 = max(a, b);                            //调用函数
    m2 = max(m1, c);                           //调用函数
    cout<<"Maximum = "<<m2<<endl;
}
```

标准库函数的函数原型存放在指定的头文件中，用 include 预处理指令获取（具体请参阅附录 B）。表 3.1 列出了 cmath 头文件中一些常用的数学函数原型。

表 3.1 cmath 中几个常用的数学函数原型

函 数 原 型	说 明
int abs(int n);	*n* 的绝对值
double cos(double x);	*x*（弧度）的余弦
double exp(double x);	指数函数 e^x
double fabs(double x);	*x* 的绝对值
double fmod(double x, double y);	*x*/*y* 的浮点余数
double log(double x);	*x* 的自然对数（以 e 为底）
double log10(double x);	*x* 的对数（以 10 为底）
double pow(double x, double y);	*x* 的 *y* 次方（x^y）

续表

函 数 原 型	说　明
double sin(double x);	*x*（弧度）的正弦
double sqrt(double x);	*x* 的平方根
double tan(double x);	*x*（弧度）的正切

【例 3-4】 求正弦和余弦值。

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{   double PI = 3.1415926535;
    double x, y;
    x = PI/2;
    y = sin(x);          //调用标准函数
    cout<<"sin("<<x<<") = "<<y<<endl;
    y = cos(x);          //调用标准函数
    cout<<"cos("<<x<<") = "<<y<<endl;
}
```

程序运行结果：

```
sin(1.5708) = 1
cos(1.5708) = 4.48966e-011
```

第 2 行输出显示了 0 的近似值。

3.2　函数参数的传递

参数是调用函数与被调用函数之间交换数据的通道。函数定义首部的参数称为形式参数（简称形参），调用函数时使用的参数称为实际参数（实参）。

实际参数必须与形式参数在类型、个数、位置上相对应。函数被调用前，形参没有存储空间。函数被调用时，系统建立与实参对应的形参存储空间，函数通过形参与实参进行通信、完成操作。函数执行完毕，系统收回形参的临时存储空间。这个过程称为参数传递或参数的虚实结合。

C++语言有三种参数传递机制：值传递（值调用）、指针传递（地址调用）和引用传递（引用调用）。实际参数和形式参数按照不同传递机制进行通信。

3.2.1　传值参数

1．值传递机制

在值传递机制中，作为实际参数的表达式的值被复制到由对应的形式参名所标识的对象中，成为形参的初始值。完成参数值传递之后，函数体中的语句对形参的访问、修改都是在这个标识对象上操作的，与实际参数对象无关。

【例 3-5】 传值参数的测试。

```
#include<iostream>
using namespace std;
void count(int x, int y)          //定义函数，x、y 为传值参数，接收实参的值
{   x = x * 2;                    //在形参 x 上操作
    y = y * y;                    //在形参 y 上操作
```

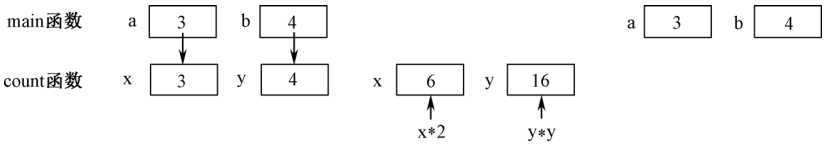
```
    cout << "x = " << x << '\t';
    cout << "y = " << y << endl;
}
int main()
{   int a = 3,   b = 4;
    count(a, b);                    //调用函数，a、b 的值分别传递给 x、y
    cout << "a = " << a << '\t';
    cout << "b = " << b << endl;
}
```

程序运行结果：

```
x = 6       y = 16
a = 3       b = 4
```

main 函数调用 count 函数时，系统建立形式参数对象 x、y，把实参 a、b 值赋给 x、y 作为初始值。count 函数对 x、y 的操作与实参 a、b 无关。返回 main 函数后，形参 x、y 被撤销，实参 a、b 的值没有变。如图 3.1 所示。

如果函数具有返回值，则在函数执行 return 语句时，系统将创建一个匿名对象临时存放函数的返回结果。这个匿名对象在返回调用之后撤销。



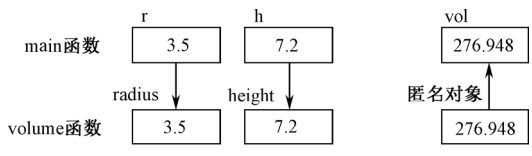
（a）调用函数　　（b）执行函数　　（c）返回

图 3.1　函数的传值参数

【例 3-6】 求圆柱体体积。

```
#include<iostream>
using namespace std;
double volume(double radius, double height);
int main()
{   double vol, r, h;
    cout<<"Input radius and height :\n";
    cin>>r>>h;
    vol = volume(r, h);                          //把 r 和 h 的值传递给形式参数
    cout<<"Volume = "<< vol << endl;
}
double volume(double radius, double height)      //在参数表中定义两个传值参数
{   return 3.14*radius*radius*height; }          //返回表达式的值
```

volume 函数用形参 radius 和 height 计算并返回圆柱体的体积。返回 main 函数时，匿名对象的值赋给变量 vol。参数传递如图 3.2 所示。



（a）调用函数　　（b）返回

图 3.2　具有返回值的函数

因为在传值方式中，实际参数对形式参数进行赋值操作，所以实际参数可以是各种能够对形式参数标识对象赋值的表达式。如果实参值的类型和形参对象类型不相同，将按形参的类型进行强制类型转换，然后赋给形参。例如，有函数定义：

```
int max(int a, int b)
{ return (a > b ? a : b); }
```

如果有调用：

```
m = max(5.2/2, 1.5);
```

则 m 的值等于 2。由于形参 a、b 都是整型的，接收实参值为 int(5.2/2)和 int(1.5)，所以通过 return 返回的值是 2。

【例 3-7】 已知 $s=\dfrac{\max(a,b,c)}{\max(a+b,b,c)*\max(a,b,b+c)}$，其中，max(x, y, z)为求 x、y 和 z 三个数中最大值函数。编写程序，输入 a、b 和 c 的值，求 s 的值。

```
#include<iostream>
using namespace std;
double max(double, double, double);
int main()
{   double a, b, c, s;
    cout << "a, b, c = ";
    cin >> a >> b >> c;
    //三次调用 max 函数，表达式作为实际参数
    s = max(a,b,c)/(max(a+b,b,c)*max(a,b,b+c));
    cout << "s = " << s << endl;
}
double max(double x, double y, double z)
{   double m;
    if (x>=y)   m = x;
       else   m = y;
    if (z>= m)   m = z;
    return m;
}
```

程序首先从 main 函数开始执行。当执行到赋值语句：

```
s = max(a,b,c)/(max(a+b,b,c)*max(a,b,b+c));
```

时，三次调用 max 函数。每次函数调用，都是先计算实际参数的值，把该值传送给相应的形式参数，然后执行函数体。当函数体执行到语句：

```
return m;
```

时，把三个实际参数中的最大值 m 通过匿名对象返回函数调用处。

2．实际参数求值的副作用

C++没有规定在函数调用时实际参数的求值顺序。实际参数求值顺序的不同规定，对一般参数没有什么影响，但若实际参数表达式之间有求值关联，则同一个程序在不同编译器可能产生不同的运行结果。

例如，有一个函数定义为:

```
int add(int a,int b)
{ return a+b; }
```

执行以下语句：

```
x = 4;
y = 6;
cout << add(++x, x+y) << endl;
```

对于自左向右求实际参数的值的编译系统，首先计算++x，表达式的值为 5，变量 x 的值也是 5；然后计算表达式 x+y，表达式的值为 11；分别把 5 和 11 传递给形参 a、b，得到的返回值是 16。

但对于自右向左求实际参数的值的编译系统，首先计算 x+y，表达式的值为 10；然后计算表达式++x，表达式的值为 5；分别把 5 和 10 传递给形参 a、b，得到的返回值是 15。

之所以产生这种语义歧义性，是因为实参中有“++x”这种赋值表达式，而另一个实参表达式又使用了 x 的值。

这种存在赋值依赖关系的传值参数称为有副作用的参数。为了避免这种情况，可以在调用函数之前先执行修改变量的表达式，以消除实参表达式求值的依赖关系，改写程序如下：

```
x = 4;
y = 6;
++x;
cout << add(x, x+y) << endl;
```

3．默认参数

函数传值调用时，实际参数作为右值表达式向形式参数提供初始值。C++允许指定参数的默认值，当函数调用中省略默认参数时，默认值自动传递给被调用函数。

调用带参数默认值的函数时，如果显式指定实际参数值，则不使用函数参数的默认值。

【例 3-8】 定义并调用函数，求两坐标点之间的距离。如果省略一个坐标点，则表示求另一个坐标点到原点的距离；如果省略一个坐标点的一个参数，则表示纵坐标 y 的值等于 0。

```
#include<iostream>
using namespace std;
#include<cmath>
//函数原型指定默认参数值
double dist(double, double, double =0, double =0);
int main()
{   double x1, y1, x2, y2;
    cout << "Enter point (x1, y1) : ";
    cin >> x1 >> y1;
    cout << "Enter point (x2, y2) : ";
    cin >> x2 >> y2;
    cout << "The distance of (" << x1 << ", " <<y1<< ") to (" << x2 << ", " << y2 << ") : "
         << dist(x1, y1, x2, y2) << endl;              //使用指定参数值
    cout << "The distance of (" << x1 << ", " << y1<< ") to (" << 0 << ", " << 0 << ") : "
         << dist(x1, y1) << endl;                      //使用默认参数值，x2、y2 为 0
    cout << "The distance of (" << x1 << ", " << y1<< ") to (" << x2 << ", " << 0 << ") : "
         << dist(x1, y1, x2) << endl;                  //使用默认参数值，y2 为 0
}
double dist(double x1, double y1, double x2, double y2)
{   return sqrt(pow(x1−x2, 2) + pow(y1−y2, 2));   }
```

程序运行结果：

```
Enter point (x1, y1) : 3    4
Enter point (x2, y2) : 5    7
The distance of (3，4) to (5, 7) : 3.60555
```

```
The distance of (3，4) to (0, 0) : 5
The distance of (3，4) to (5, 0) : 4.47214
```

在这个程序中，函数 dist 定义了 4 个形式参数，其中设置了两个默认值。在 main 函数中，三次调用函数 dist，每次调用的实际参数的个数都不一样。第一次调用时，全部实际参数不采用默认值。第二次调用时，实际参数采用两个默认值。第三次调用时，实际参数采用一个默认值。dist 函数至少需要两个实际参数。

有关默认参数的说明如下。

① C++规定，函数的形式参数说明中设置一个或多个实际参数的默认值，默认参数必须是函数参数表中最右边（尾部）的参数。调用具有多个默认参数的函数时，如果省略的参数不是参数表中最右边的参数，则该参数右边的所有参数也应该省略。

② 默认参数应该在函数名第一次出现时指定，通常在函数原型中。若已在函数原型中指定默认参数，则函数定义时不能重复给出。

③ 默认值可以是常量、全局变量或函数调用，但不能是局部量。

④ 默认参数可以用于内联函数（参见 3.5 节）。

3.2.2 指针参数

函数定义中的形式参数被说明为指针类型时，称为指针参数。形参指针对应的实际参数是地址表达式。调用函数时，实际参数把对象的地址值赋给形式参数名标识的指针变量，被调用函数可以在函数体内通过形参指针来间接访问实参地址所指的对象。这种参数传递方式称为指针传递或地址调用。

【例 3-9】 通过函数及其指针参数来实现两个整型变量的值交换。

```
#include<iostream>
using namespace std;
void swap(int *, int *);
int main()
{   int a = 3, b = 8;
    cout << "before swapping…\n";
    cout << "a = " << a << ", b = " << b << endl;
    swap(&a, &b);                          //实际参数是整型变量的地址
    cout << "after swapping…\n";
    cout << "a = " << a << ", b = " << b << endl;
}
void swap(int *x, int *y)                  //形式参数是整型指针
{   int temp = *x;
    *x = *y;
    *y = temp;
}
```

程序运行结果：

```
before swapping…
a = 3, b = 8
after swapping…
a = 8, b = 3
```

main 函数执行函数调用语句：

```
swap(&a, &b);
```

时，把变量 a 和 b 的地址分别传送给形式参数指针变量 x 和 y，令 x 指向 a，y 指向 b。执行函数体时，*x、*y 通过间址访问对变量 a、b 进行操作，交换 a、b 的值。指针传递过程如图 3.3 所示。

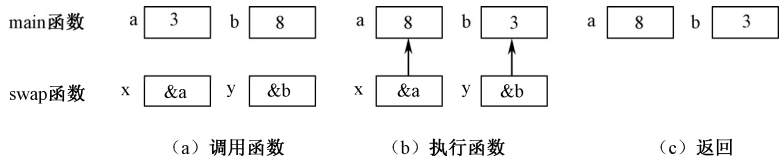


图 3.3　指针传递过程

从上述讨论可知，形参指针可以通过获取对象地址来访问实参地址所指对象。指针参数的本质也是传值参数。对于一般传值参数，实际参数向形式参数传送的是数据表达式。而指针参数对应的实际参数是地址表达式，如果这个表达式是一个实际对象的地址值，则形式参数接收这个地址值后，可以间接访问这个地址所指的对象。

为了避免被调用函数对实参所指对象的修改，可以用关键字 const 约束形参指针的访问特性。

【例 3-10】 使用 const 限定指针，保护实参对象。

```
#include<iostream>
using namespace std;
int func(const int * const p)
{   int a = 10;
    a += *p;
    //*p = a;          //错误，不能修改 const 对象
    //p = &a;          //错误
    return a;
}
int main()
{   int x = 10;
    cout << func(&x) << endl;
}
```

程序运行结果：

```
20
```

实参表达式是变量 x 的地址，形参 p 被定义为指向常量的常指针。调用函数时，用实参地址值初始化后，函数体对 p 和*p 的访问都被约束为只读，从而保护了实参 x 不能通过 p 修改。

可以不约束 p 的访问：

```
int func(const int * p);
```

使在函数体内对 p 的修改变为合法：

```
p = &a;                //合法
```

但这样一来，形参指针 p 就与实参所指对象失去关联了。

当将常对象的地址传递给形参指针时，形参必须用 const 约束。

【例 3-11】 传递常对象地址。

```
#include<iostream>
using namespace std;
int func (const int * p)
{   int a = 10;
    a += *p;
```

```
    return a;
}
int main()
{   const int M = 10;
    cout << func(&M) << endl;
}
```

main 函数中，M 是一个标识常量，其地址作为实参传递给形参指针 p。p 的间址访问被约束，函数不能通过*p 修改 M。在这个程序中，func 确实没有修改*p，但是，不能因此而不对 p 加以约束。例如，func 的函数原型改为：

```
int func (int * p);              //参数 p 没有约束
```

编译器仅从函数原型分析，认为该函数有可能修改常实参 M，从而报告错误。

3.2.3 引用参数

如果 C++函数的形式参数被定义为引用类型，则称为引用参数。引用参数对应的实际参数应该是对象名。函数被调用时，形式参数不需要开辟新的存储空间，形式参数名作为引用（别名）绑定于实际参数标识的对象上。执行函数体时，对形参的操作就是对实参对象操作。直到函数执行结束，撤销引用绑定。

【例 3-12】 通过函数及其引用参数来实现两个整型变量的值交换。

```
#include<iostream>
using namespace std;
void swap(int&, int&);
int main()
{   int a = 3, b = 8;
    cout << "before swapping…\n";
    cout << "a = " << a << ", b = " << b << endl;
    swap(a, b);                          //实际参数是整型变量名
    cout << "after swapping…\n";
    cout << "a = " << a << ", b = " << b << endl;
}
void swap(int &x, int &y)            //形式参数是整型引用
{   int temp = x;
    x = y;
    y = temp;
}
```

调用函数 swap 后，形参 x、y 分别是实参 a、b 的引用，函数体内对 x、y 的操作实际上是对 a、b 的操作。

请注意，若把例 3-9 的 swap 函数改为：

```
void swap(int x, int y)              //传值参数
{   int temp = x;
    x = y;
    y = temp;
}
```

则 main 函数对其调用方式不变，程序运行结果不会交换 a、b 的值。因为形参 x、y 只在调用时接收 a、b 的值作为初值，函数体内的操作与 a、b 无关。调用语句：

```
swap(a, b);
```

对于传值参数，实际参数 a、b 是右值表达式。而当该调用对应引用参数时，a、b 作为标识名将与形式参数名绑定。

引用参数和指针参数都不需要像传值参数那样产生实参对象数据的副本，并且，引用参数不像指针参数那样通过间址访问实参对象，特别适用于大对象参数的高效操作。

和指针参数的情形一样，为了避免被调用函数对实参对象产生不必要的修改，可以使用 const 限定引用。

【例 3-13】 使用 const 引用参数。

```
#include<iostream>
#include<iomanip>
using namespace std;
void display(const int & rk) //定义 const 引用参数
{   cout << rk << " :\n" << "dec : " << rk << endl << "oct : " << oct << rk << endl
          << "hex : " << hex << rk << endl;
}
int main()
{   int m=2618;
    display(m);                    //实际参数是变量
    display(4589);                 //实际参数是常数
}
```

程序运行结果：

```
2618 :
dec : 2618
oct : 5072
hex : a3a
4589 :
dec : 4589
oct : 10755
hex : 11ed
```

注意，在本例 main 函数中第 2 次调用 display 函数时，用常数 4589 作为实际参数。C++规定，函数的 const 引用参数允许对应的实际参数为常数或者表达式。调用函数进行参数传递时将产生一个匿名对象保存实参的值。形参标识名作为这个匿名对象的引用，对匿名对象进行操作。匿名对象在被调用函数运行结束后撤销。这种 const 引用参数的使用效果与传值参数情况类似。

【例 3-14】 const 引用参数的匿名对象测试。

```
#include<iostream>
using namespace std;
void anonym (const int & ref)
{   cout << "The address of ref is : " << &ref << endl;
    return;
}
int main()
{   int val = 10;
    cout << "The address of val is : " << &val << endl;
    anonym(val);
    anonym(val + 5);
}
```

程序运行结果：

```
The address of val is : 0012FF4C
The address of ref is : 0012FF4C
The address of ref is : 0012FE80
```

main 函数第 1 次调用 anonym 函数时，实参是变量名。形参 ref 与实参 val 绑定。在程序输出的第 1 行和第 2 行，实参和形参的地址值相同，说明引用参数与实参对象都是同一个存储单元，引用参数以别名方式在实参对象上进行操作。无论形参是否被约束，情形都一样。

第 2 次调用 anonym 时，实参是表达式。C++为 const 引用建立匿名对象用于存放 val+5 的值。第 3 行输出是匿名对象的地址。只有 const 引用对应的实参可以是常量或表达式，非约束的引用参数对应的实参必须是对象名。

3.2.4 函数的返回类型

C++函数可以通过指针参数或引用参数修改实际参数，从而获取函数的运行结果。return 语句也可以返回表达式的执行结果。return 语句的一般格式为：

return（表达式）；

其中，圆括号可以省略。“表达式”的类型必须与函数原型定义的返回类型相对应，可以为数值型和字符型，也可以为指针和引用。

当函数定义为 void 类型时，return 语句不带返回“表达式”，或者不使用 return 语句。

一个函数体内可以有多个 return 语句，但只会执行其中一个。return 语句的作用是，把“表达式”的值通过匿名对象返回调用点，并中断函数执行。

1．返回基本类型

如果函数定义的返回类型为基本数值类型，则执行 return 语句时，首先计算表达式的值，然后把该值赋给 C++定义的匿名对象。匿名对象的类型是函数定义的返回类型。通过这个匿名对象，把数值带回函数的调用点，继续执行后续代码。

例如，有函数原型：　int function();

函数体若有：　return x;

则执行该语句时，把 x 的值赋给 int 类型的匿名对象，返回到函数调用点。

又如，若有：　return a+b+c;

则首先对表达式求值，然后对 int 类型的匿名对象赋值，返回到函数调用点。

对匿名对象赋值时，如果表达式的值的类型与函数定义的返回类型不相同，将强制转换成函数的返回类型。

2．返回指针类型

函数被调用之后可以返回一个对象的指针值（地址表达式）。返回指针类型值的函数称为指针函数。指针函数的函数原型一般为：

类型 * 函数名（形式参数表）；

函数体中用 return 语句返回对象的指针。

【例 3-15】 定义一个函数，返回较大值变量的指针。

```
#include<iostream>
using namespace std;
int * maxPoint(int * x, int * y)    //函数返回整型指针
{   if (*x > *y)   return x;
    return y;
}
```

```
int main()
{   int a, b;
    cout << "Input a, b : ";
    cin >> a >> b;
    cout << * maxPoint(&a, &b) <<endl;
}
```

调用函数 maxPoint 后，两个形参指针分别指向 main 函数的变量 a 和 b，即 x 的值是实参 a 的地址，y 的值是实参 b 的地址。在 maxPoint 函数中，*x、*y 间址访问 a、b，函数通过匿名对象返回它们之中较大者的指针。匿名对象的类型就是函数的返回类型 int*，接收 return 语句的地址表达式的值。

main 函数在输出语句中调用 maxPoint 函数。函数返回指针（地址值），然后用指针运算符访问所指对象，输出 a、b 之中的大值。

为了约束对实参的访问，函数 maxPoint 还可以写为：

```
const int * maxPoint(const int * x, const int * y)
{   if (*x > *y)   return x;
    return y;
}
```

maxPoint 返回对象的指针，不需要复制产生实际返回对象的值。如果函数需要对大对象进行操作，使用指针函数显然可以节省匿名对象生成的时空消耗。

注意，指针函数不能返回局部量的指针。例如，以下函数定义是错误的：

```
 int * f()
{   int temp;
    //…
    return &temp;
}
```

temp 是在函数 f 运行时建立的临时对象，f 运行结束，系统释放 temp，因此，函数返回局部变量的地址是不合理的。

3．返回引用类型

C++函数返回对象引用时，不产生实际返回对象的副本，返回时的匿名对象是实际返回对象的引用。返回引用比返回指针更直接，可读性更好。

【例 3-16】 定义一个函数，返回较大值变量的引用。

```
#include<iostream>
using namespace std;
int & maxRef(int & x, int & y)            //函数返回整型引用
{   if (x > y)   return x;
    return y;
}
int main()
{   int a, b;
    cout << "Input a, b : ";
    cin >> a >> b;
    cout << maxRef(a, b) <<endl;
}
```

程序运行结果：

```
Input a, b :   3   9
```

9

从函数参数传递的规律可知，一旦调用 maxRef，形参 x、y 分别是实参 a、b 的引用。函数体的 if 语句把 x、y 之中（a、b 之中）大值的对象通过 return 返回。因为 maxRef 函数返回类型是整型引用，所以，C++无须建立返回用的匿名对象，函数调用的返回就是对象的引用。在上述运行示例中，函数返回变量名 y，而 y 是 b 的引用，所以，在 main 函数的调用返回是 b 的引用。变量名在输出流中作为右值表达式，输出 b 的值为 9。

函数返回引用需要依托于一个对象。显然，被依托的返回对象不能是函数体内说明的局部变量。其原因与返回指针的函数一样，被调用函数内定义的局部量是临时对象，函数返回时将被释放。例如，以下函数定义是错误的：

```
int & r()
{   int temp;
    //…
    return temp;
}
```

返回对象可以是非局部对象或静态对象。

函数返回引用，使得函数调用本身是对象的引用，就像返回对象的标识别名。所以，返回引用的函数调用可以作为左值。

【例 3-17】 输入一系列正整数和负整数，以 0 结束，统计其中正整数和负整数的个数。

```
#include<iostream>
using namespace std;
int &count(int);
int a, b;
int main()
{   int x;
    cout << "Input numbers, the 0 is end : \n";
    cin >> x;
    while (x)
    {  count(x) ++;    //根据返回不同变量引用进行++运算
       cin >> x;
    }
    cout << "the number of right: " << a << endl;
    cout << "the number of negative: " << b << endl;
 }
int & count(int n)
{   if (n > 0)   return a;
    return b;
}
```

运行程序，输入数据，显示结果为：

```
Input numbers, the 0 is end :
-2 5 8 -9 20 0
the number of right:3
the number of negative:2
```

函数 count 返回全局变量 a 或 b 的引用。当参数 n 的值大于 0，函数返回变量 a 的引用，main 函数中的调用：

```
count(x)++
```

相当于：a ++

而当参数 n 的值小于 0，函数返回变量 b 的引用， main 函数中的调用：

count(x)++

相当于：b ++

当然，这个程序的可读性并不好，这里仅作为一个简单的示例。事实上，返回对象引用的函数调用可以作为左值，这一点很有意义。我们将在第 7 章中看到，正是因为 C++允许函数返回对象引用，使得在调用运算符重载函数时，可以实现运算符的连续操作。

3.3 函数调用机制

一个 C++程序是由若干个函数构成的。每个函数都是独立定义的模块。函数之间可以互相调用。函数调用关系如图 3.4 所示，图中箭头表示在函数体内出现对另一个函数的调用。

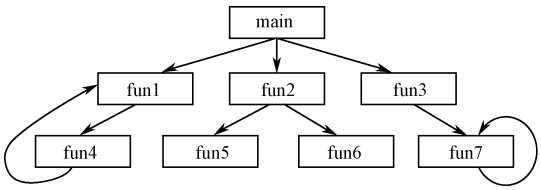


图 3.4 函数调用关系

main 函数可以调用各个自定义函数和库函数，各个自定义函数可以互相调用。每个应用程序只有一个 main 函数，由系统启动。函数之间可以互相调用，可以嵌套调用。例如，在图 3.4 中，main 函数调用函数 fun2，在 fun2 函数体中又调用 fun5，这种方式称为嵌套调用。函数可以自身调用，称为递归调用。图 3.4 中，fun7 在函数体内出现自身调用，称为直接递归。fun4 函数中调用 fun1，而 fun1 中又调用了 fun4，称为间接递归。

函数之所以能够正确地实现调用，是由于系统设置一个先进后出堆栈进行调用信息管理。执行代码出现函数调用时，系统首先把调用现场各种参数、返回后要继续执行的代码地址，压入堆栈；然后传递参数，把控制权交给被调用函数；被调用函数执行结束，堆栈弹出现场参数，控制权交还调用函数继续执行。

3.3.1 嵌套调用

函数嵌套调用的代码结构如图 3.5 所示。

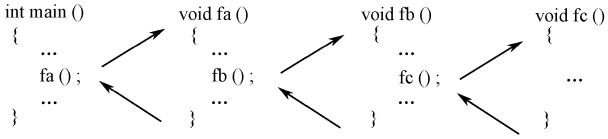


图 3.5 函数嵌套调用的代码结构

图 3.5 表示，在 main 函数中调用 fa 函数，在 fa 函数中调用 fb 函数，在 fb 函数中调用 fc 函数。main 函数由操作系统调用，首先把操作系统的运行状态、返回地址及 main 函数的参数压入堆栈；在 main 函数中，调用 fa，把 main 的运行状态、返回地址及 fa 的参数压入堆栈……一直到 fc 执行结束，堆栈弹出栈顶的第一层信息，接收 fc 的执行结果，恢复 fb 的执行现场，继续执行 fb 的后续代码；当 fb 执行结束后，堆栈又弹出顶层信息，接收 fb 的执行结果，恢复 fa 的执行现场，继续执行 fa 的后续代码；一直到堆栈弹空，程序执行结束。

函数调用信息管理堆栈如图 3.6 所示。

【例 3-18】 已知

$$g(x,y)=\begin{cases}\dfrac{f(x+y)}{f(x)+f(y)} & x\leqslant y\\ \dfrac{f(x-y)}{f(x)+f(y)} & x>y\end{cases}$$

式中，$f(t)=\dfrac{1+\mathrm{e}^{-t}}{1+\mathrm{e}^{t}}$，求 $g(2.5, 3.4)$，$g(1.7, 2.5)$和 $g(3.8, 2.9)$的值。

程序设计按照功能划分和代码重用的原则，首先定义 f 函数，通过 g 函数调用 f 函数，实现函数的完整功能。main 函数向 g 传递实际参数的数据，完成计算。

```
#include<iostream>
#include<cmath>
using namespace std;
double f(double);
double g(double, double);
int main()
{   cout << "g(2.5, 3.4) = " << g(2.5, 3.4) << endl;
    cout << "g(1.7, 2.5) = " << g(1.7, 2.5) << endl;
    cout << "g(3.8, 2.9) = " << g(3.8, 2.9) << endl;
}
double g(double x, double y)
{   if (x <= y)    return f(x+y) / (f(x) + f(y));
    else     return    f(x−y) / (f(x) + f(y));
}
double f(double t)
{   return (1 + exp(−t)) / (1 + exp(t));   }
```

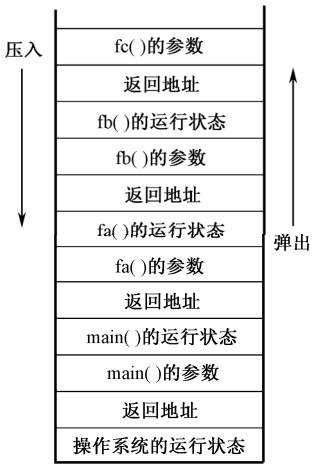


图 3.6　函数调用信息管理堆栈

程序运行结果：

```
g(2.5, 3.4) = 0.0237267
g(1.7, 2.5) = 0.0566366
g(3.8, 2.9) = 5.25325
```

3.3.2　递归调用

递归是推理和问题求解的一种强有力方法，原因在于，许多对象，特别是数学研究对象，具有递归的结构。简单地说，如果通过一个对象自身的结构来描述或部分描述该对象，就称为递归。

递归定义使人们能够用有限的语句描述一个无穷的集合。C++语言允许一个函数体中出现调用自身的语句，称为直接递归调用。也允许被调用的另一个函数又反过来调用原函数，称为间接递归调用。这种功能为递归结构问题提供了求解的实现手段，使程序语言的描述与问题的自然描述完全一致，因而使程序易于理解、易于维护。

下面通过一个简单的例子说明递归函数的构成规律和执行过程。

【例 3-19】 使用递归函数编程序求 $n!$。

根据数学知识，非负整数 n 的阶乘为：

$$n!=n\times(n-1)\times(n-2)\times\cdots\times1$$

其中，$0!=1$。

当 $n\geqslant0$ 时，阶乘可以用循环迭代（非递归）计算：

```
fact = 1;
for (int k = n; k >= 1; k--)
    fact *= k;
```

也可以用另一种递归形式定义阶乘：

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

阶乘的递归定义把问题分解为两部分：一部分用已知的参数 n 作为被乘数，另一部分使用原来的阶乘定义作为乘数。不过，乘数$(n-1)!$的问题规模缩小了。

由定义有，$(n-1)! = (n-1) \times (n-2)!$，问题规模进一步缩小，从而产生越来越小的问题，最后归结到基本情况，0!=1。C++函数调用能够识别并处理这种基本情况，向前一个函数调用返回结果，并回溯一系列中间结果，直到把最终结果返回给调用函数。

以下程序在 main 函数中调用求阶乘的递归函数。

```
#include<iostream>
using namespace std;
long fact(int n)
{   if(n==0)    return 1;                    //递归终止情况
    else    return n * fact(n-1);            //递归调用
}
int main()
{   int n;
    cout << "Enter n (>=0) : ";
    cin >> n;
    cout << n << "! = " << fact(n) << endl;
}
```

递归函数执行由递推和回归两个过程完成。假如执行 main 函数时，输入 n 的值为 3，则函数调用 fact(3)的递推和回归过程如图 3.7 所示。

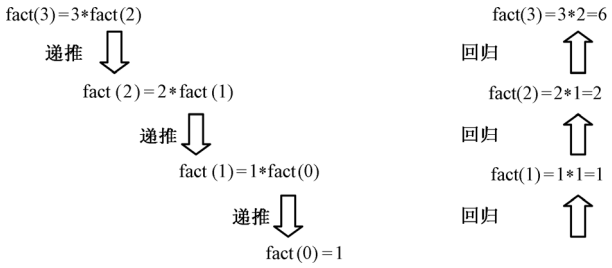


图 3.7　函数调用 fact(3)的递推过程和回归过程

递归调用之所以能够实现，关键是系统使用堆栈来保存函数调用中的传值参数、局部变量和函数调用后的返回地址。函数自身调用进行递推：系统把有关参数和地址压进堆栈，一直递推到满足终止条件，找到问题的最基本模式为止。然后进行回归：系统从堆栈中逐层弹出有关参数和地址，执行地址所指向的代码，一直到栈空为止，得到问题的解。

递归调用与一般函数调用，堆栈管理的操作过程是一致的。不同的是，函数的自身调用就像是产生多个正在运行（等待结束）的相同函数副本。如果这种调用不能终止并产生回归，将是程序十分严重的错误。

构成递归函数有两个基本要素：

- 描述问题规模逐步缩小的递归算法；
- 描述基本情况的递归终止条件。

在 fact 函数中，递归调用的语句为：

```
t = n * fact(n-1);
```

由调用参数 n-1，使问题规模逐步缩小，直至到达递归调用终止条件：

```
n==0
```

返回基本情况值 1。

在程序中，递归算法和递归条件通常用条件语句表达。递归调用可以出现在执行语句中，也可以出现在判断表达式中。

【例 3-20】 求正整数 *a* 和 *b* 的最大公约数。

a 和 *b* 的最大公约数，就是能同时整除 *a* 和 *b* 的最大整数。从数学上可知，求 *a* 和 *b* 的最大公约数等价于求 *b* 与 *a* 除以 *b* 的余数（即 *a* % *b*）的最大公约数。具体的算法可以用递归公式表示为：

$$a \text{ 和 } b \text{ 的最大公约数} = \begin{cases} a & b=0 \\ b \text{ 与（} a \% b \text{）的最大公约数} & b>0 \end{cases}$$

例如，按上述递归公式求 *a*=24 与 *b*=16 的最大公约数的过程为：

因为 *b*=24＞0，

所以求 *a*=24 与 *b*=16 的最大公约数转换为求 *a*=16 与 *b*=(24 % 16)=8 的最大公约数；

因为 *b*=8＞0，

所以求 *a*=16 与 *b*= 8 的最大公约数转换为求 *a*=8 与 *b*=(16 % 8)=0 的最大公约数；

因为 *b*=0，

所以 *a*=8 与 *b*=0 的最大公约数为 8，即 24 和 16 的最大公约数是 8。

按上述递归公式编写程序如下：

```
#include<iostream>
using namespace std;
int gcd(int, int);
int main()
{   int a, b;
    cout << "input a (>=0) : ";
    cin >> a;
    cout << "input b (>=0) : ";
    cin >> b;
    cout << "gcd (" << a << ", " << b << ") = " << gcd(a, b) << endl;
}
int gcd(int a, int b)
{   int g;
    if(b==0)   g = a;
    else   g = gcd(b, a%b);
    return g;
}
```

【例 3-21】 Fibonacci 数列。

Fibonacci 数列的第 1 项为 0，第 2 项为 1，后续的每项是前面两项的和。该数列两项的比例趋于一个常量：1.618…，称为黄金分割。数列形如：

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…
```