第三讲 函数

函数的概念作用

函数（function）代表一个功能。在一个程序文件中，只能有一个主函数（main），主函数作为程序的入口，在程序运行时调用其他函数模块，其他函数模块之间可以相互调用。在实际开发过程中，主函数等同于总调度中心，调动每个函数进行实现所需的功能要求。

函数的分类

用户角度：

- 系统函数
  - 又名库函数，编译系统提供，用户无需定义即可调用
- 自定义函数

  - 用户自己定义的函数，解决用户特定的需求

  函数形式角度：
- 无参函数
  - 调用时无需给出参数的函数
  - `类型标识符 函数名称（[void]）{函数体}`
- 带参函数
  - 需要接受其他数据（参数）的函数
  - `类型标识符 函数名称（形式参数表）{函数体}`

函数的调用和声明

一般形式

`函数名称（[实参列表]）`

若函数为无参函数或带默认参数的函数，则`[实参列表]`可以省略。

实参和形参是相对而言的，主调函数的参数列表称之为实参，被调函数的参数列表称之为形参。

- 调用方的传出参数 --> 实参

- 被调用方传入的参数 --> 形参

  值得注意的是，倘若实参列表存在多个实参，不同编译系统有着不同的求值顺序。假设变量x的值为5，有以下函数调用：

```
#include <iostream>
#include <iomanip>
using namespace std;
void func(int a, int b) {
    cout << a << " " << b << endl;
}
int main(int argc, char const *argv[])
{
    int x = 5;
    func(x, ++x);
    x = 5;
    func(x, x++);
    return 0;
}
// 结果:         6 6
//                      6 5
```

如果按照自左至右的求参顺序，则函数调用相当于func(5,6) ,若按照自右至左顺序进行求参，则等同于func(6,6)。常见的编译系统是按自右至左进行求参。

递归调用

C++允许在调用一个函数的过程中又间接或直接地调用该函数自身，称之为递归调用。例如：

```
#include <iostream>
using namespace std;
// 递归计算 a 的累加
int s(int a) {
    return a == 0 ? 0 : a + s(a - 1);
}
int main(int argc, char const *argv[])
{
    cout << s(3) << endl;
    return 0;
```

```
}
// 结果:        6
```

递归需要占内存去保存调用函数的信息，递归的深度越深所消耗的内存越大。为了考虑效率，常常使用非递归方法求解

函数的嵌套

在C++语言中不允许函数的嵌套定义，换而言之，一个函数内不能存在另一个函数的定义，如下所示。

```
void fun1() //函数fun1首部
{
        ... //函数fun1的函数体
        void fun2() //定义fun2函数，这是非法的
        {
                ... //函数fun2的函数体
        }
}
```

函数的声明

如果调用的是用户自定义函数，如果被调函数处于主调函数之后，则必须要在主调函数中或之前对被调函作出声明。

```
#include <iostream>
using namespace std;
// 声明函数 s(int, int)
int s(int, int);
int main(int argc, char const *argv[])
{
    cout << s(3) << endl;
    return 0;
}
int s(int a)
{
    return a == 0 ? 0 : a + s(a - 1);
}
```

- 整个程序中，函数的定义只能出现一次
- 函数的声明是对编译系统的一个说明，其包括函数类型、函数名称、形参类型。与函数的定义不同，它不包含函数体，形参名称可以省略，并且多次声明。函数的声明必须以分号结束

函数的传递

大多数情况下，函数调用往往是带参调用。主调函数与被调函数之间往往存在数据传递的关系。函数定义首部的括号中的变量名称是形式参数（formal parameter,简称形参）。主调函数调用该函数时，函数名称后的括号里的参数称之为实际参数（actual parameter,简称实参），其中实参也可以是一个表达式。

```
#include <iostream>
using namespace std;
int min(int x, int y) //定义带参函数min, x和y为形参
{
    return x < y ? x : y;
}
int main()
{
    int a = 0, b = 0, c = 0;
    cout << "请输入两个整数: " << endl;
    cin >> a >> b;
    c = min(a, b); //调用min函数，给定实参为a,b，函数返回值赋给c
    cout << "min = " << c;
    return 0;
}
// 结果:          请输入两个整数:
//                        123 456
//                        min = 123
```

带默认值的函数

通常情况下，被调函数的形参的数据是从主调函数的实参中获取的，因此实参的个数与形参的个数应相等。但在特殊情况下，需要多次调用某一函数并使用同一实参，C ++提供了便捷的方法，那就是给定形参一个默认值，从而形参不需要从实参中获取数据，如下所示

```
int area(int r = 3);
```

指定r的默认值为3，如果调用该函数时，确定r的值取3，则无需给出实参，如

`area(); // 相当于 area(3);`

如果需要取其他值，那么可以通过实参给定，如

`area(6);`

形参与实参的结合是由左到右的顺序进行的，因此第一个实参必须与第一个形参相结合，第二个实参与第二个形参相结合……。因此指定默认值的参数当放在函数的参数列表的最右端，否则将出现错误。如下：

- `int fun1(double a, int b = 0, int c ,char d ='y');` ❌错误
- `int fun2(double a, int c, int b = 0 ,char d ='y');` ✔正确

函数的重载

```
#include <iostream>
using namespace std;
double min(double a, double b, double c); //函数声明
long min(long a, long b, long c);         //函数声明
int min(int a, int b, int c);             //函数声明
int main()
{
    double d = 0, d1 = 0, d2 = 0, d3 = 0;
    cin >> d1 >> d2 >> d3; //输入3个双精度数
    d = min(d1, d2, d3);   //求3个双精度数中的最小值
    cout << "3个双精度的最小值为：" << d << endl;
    long l = 0, l1 = 0, l2 = 0, l3 = 0;
    cin >> l1 >> l2 >> l3; //输入3个长整型数
    l = min(l1, l2, l3);   //求3个长整型数中的最小值
    cout << "3个长整型的最小值为：" << l << endl;
    int i = 0, i1 = 0, i2 = 0, i3 = 0, i4 = 0;
    cin >> i1 >> i2 >> i3; //输入3个整型数
    i = min(i1, i2, i3);   //求3个整型数中的最小值
    cout << "3个整型的最小值为：" << i << endl;
    return 0;
}
double min(double a, double b, double c) //定义求3个双精度数中的最小值
的函数
{
    if (b < a)
```

```
        a = b;
    if (c < a)
        a = c;
    return a;
}
long min(long a, long b, long c) //定义求3个长整型中的最小值的函数
{
    if (b < a)
        a = b;
    if (c < a)
        a = c;
    return a;
}
int min(int a, int b, int c) //定义求3个整型中的最小值的函数
{
    if (b < a)
        a = b;
    if (c < a)
        a = c;
    return a;
}
// 结果:         2.1 3.5 2.2
//                      3个双精度的最小值为：2.1
//                      123456789 987654321 4236231467
//                      3个长整型的最小值为：123456789
//                      1 2 3
//                      3个整型的最小值为：1
```

说人话就是同名函数的参数可以有不同个数 or 不同类型

函数的内联

> *为了提高程序的运行速度，C++提供了内联函数的的特性。内联函数与普通函数之间的区别不在于编写方式，而在于编译系统如何将它们组合到程序中。普通函数的来回跳转将消耗计算机一定的性能，内联函数则无需来回跳转，因而相对于普通函数具有更快的运行速度。但是内联函数也有弊端，虽然节省了运行时间但是增加了内存消耗。如果程序在5个不同的地方调用内联函数，那么相当于该程序复制了5份内联函数的代码副本。*
>
> *人话：内联调用的时候，会把代码复制粘贴到目标地点。*

举个栗子：

```
#include <iostream>
using namespace std;
inline int min(int a, int b)
{
    return a > b ? b : a;
}
int main(int argc, char const *argv[])
{
    cout << min(1, 2) << endl;
    return 0;
}
```

在编译的过程中，实际上编译的是：

```
#include <iostream>
using namespace std;
int main(int argc, char const *argv[])
{
    cout << (1 > 2 ? 2 : 1) << endl;
    return 0;
}
```

内联的原始实现

内联函数是C++新增的特征。宏也可以看作内联代码的原始实现，但是二者具有一定的区别。

`#define SQUARE(X) X*X //计算平方的宏`

宏与内联函数相比，宏没有通过参数传递，仅仅是文本替换实现的

所以上述代码会出现❌错误

`a= SQUARE(6.5) //等同于a=6.5*6.5` ✓

`b= SQUARE(6.5+7.5) //等同于b=6.5+7.5* 6.5+7.5` ❌

经过修改，将宏定义为 `#define SQUARE(X) (X)*(X)` 即可修复上述错误

`b= SQUARE(6.5+7.5) //等同于b=(6.5+7.5)*(6.5+7.5)` ✓