1.5函数

函数

- 函数是组成程序的基本功能单元，它将一个复杂的任务分解为若干个相对独立且功能单一的子程序。
- 函数的一个重要目的是复用。
- 掌握函数的关键是通信方式：形参列表、返回值

函数定义：

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

<函数类型>表示函数返回值的类型；<形参表>表示函数被问

的入口参数列表，可为空或void。

C++不允许嵌套定义函数（将一个函数定义在另一个函数内）。

```
int sum(int x, int y)
{
    return x + y
}
```

函数声明（函数原型）

- C++允许函数调用在前，函数定义在后。
  - 此时要求在调用函数前必须先进行函数声明，以便告诉编译程序被调用函数的类型和参数类型。
  - 函数原型是主调函数与被调函数间的协议。
- 函数声明的一般形式为：

```
[<存储类型>]<函数类型><函数名>(<形参表>);
```

- <存储类型>包括内部函数（static）和外部函数（extern，默认值），内部函数只能被同一个源文件中的函数调用，而外部函数可以被其它源文件中的函数调用。
- 调用外部函数前必须先进行外部函数声明。
- 函数声明的作用：
  - 把函数名、函数类型、形参告诉编译系统，以便调用时进行语法检查。
- 一般用法（推荐）：
  - 函数声明放*.h，函数定义放*.cpp。
  - 需要调用时，#include头文件。
- 个人习惯：
  - 声明和定义分开

- 声明集中放到程序的前面部分

```
//函数声明: eg2_19_26.hpp
void eg2_19();
void eg2_20();
void eg2_21();
void eg2_22();
void eg2_23();
void eg2_23_1();
void eg2_24();
void eg2_25();
void eg2_26();
```

```
//函数定义: eg2_19_26.cpp
#include "eg2_19_26.hpp"

void eg2_19() {
    // Some code ...
}
```

```
//函数调用: main函数
#include "eg2_19_26.hpp"

int main(int argc, const char * argv[]) {
    eg2_19();
    return 0;
}
```

- 一般用法

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

int sum1(int x=0, int y=0);
void fun1();

int main(int argc, const char * argv[]) {
    // insert code here...
    int x, y;
    cout << "Please input x, y:";
    cin >> x >> y;
    cout << "sum = " << sum1(x, y) << endl;
}

int sum1(int x, int y) {
    return x + y;
}

void fun1() {
    // insert code here...
}
```

函数调用

- 函数定义后便可以反复调用（cal），每次调用通过赋予形参实际的参数值（实参），从而完成对实际数据的处理。函数调用的一般形式为：

```
<函数名>(实参1，实参2，…，实参n)
```

- 函数调用过程：
  - 中断当前函数（主调函数）的执行，将程序的执行流程转移到被调用函数，并将实参传递给形参。调用结束后返回（return）到主调函数。

- 实参是一个实际的参数值，它可以是常量、变量或表达式。
- 传递方式：值传递，按地址传递、引用传递。

例子：函数调用、值传递

```
void swap(int x, int y);

void eg2_21()
{
    int a = 20, b = 40;
    cout << "data: a = " << a << ", b = " << b << endl;
    swap(a, b);
    cout << "data: a = " << a << ", b = " << b << endl;
}
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

```
data: a = 20,  b = 40
swap: a = 20,  b = 40
```

因为是值传递，所以并不能交换

例子：函数调用、地址、引用

解决方案

```
//调用：swap(&a, &b);
void swap(int* x, int* y){
    int temp = *x;
    *x = *y;
    *y = temp;
}
//调用：swap(a, b);
void swap(int& x, int& y){
    int temp = x;
    x = y;
    y = temp;
}
```

```
void swap1 (int* x, int* y);
void swap2 (int& x, int& y);
void eg2_21() {
    int a = 20, b = 40;
    cout << "data: a = " << a << ", " << " b = " << b << endl;
    swap1(&a, &b);
    swap2(a, b);
    cout << "swap: a = " << a << ", " << " b = " << b << endl;
}
```

```
void swap1(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

```
void swap2(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```

例子：数组为函数参数

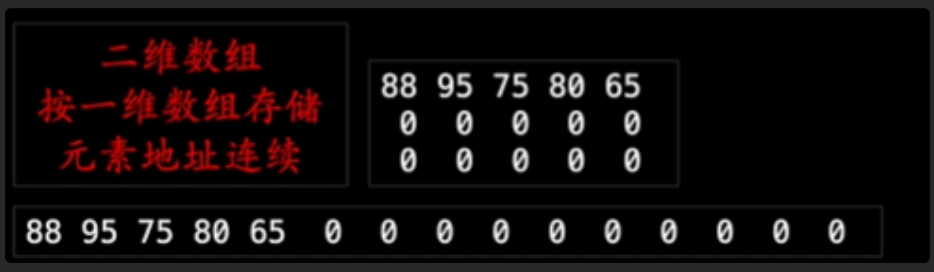
```
#include<iostream>
using namespace std;

float average1(const float array[], int n);
float average2(const float* array, int n);
void eg2_23()
{
    float aray1[] = {88, 95, 75 80, 65};
    const int n = sizeof(aray1) / sizeof(float);
    cout << "average1 = " << average1(aray1, n) << endl;
    cout << "average2 = " << average2(aray1, n) << endl;//传参方法1，数组名
    cout << "average2 = " << average2(&aray1[0], n) << endl;//传参方法2，数组首元素地址
}
float average1(const float array[], int n)
{
    float ave = 0;
    for (int i = 0; i < n; i++)
        ave += array[i];
    return ave / n;
}
float average2(const float* array, int n)
{
    float ave = 0;
    for (int i = 0; i < n; i++)
        ave += array[i];
    return ave / n;
}
```

例子：数组为函数参数

- 二维数组按一维数组存储，元素地址连续
- 图中第三行方式可拓展性强，但对代码的易读性有一定影响

```
void show1(const float array[][5], int row, int col);
void show2(const float (*array)[5], int row, int col);
void show3(const float *array, int row, int col);
```

```
4
5   void eg2_23_1()
6   {
7       const int row = 3;
8       const int col = 5;
9       float array1[row][col] = {88, 95, 75, 80, 65};
10      show1(array1, row, col);
11      show2(array1, row, col);
12      show3(&array1[0][0], row, col);
13  }
```




```
1   void show1(const float array[][5], int row, int col)
2   {
3       for (int i = 0; i < row; i++)
4       {
5           for (int j = 0; j < col; j++)
6               cout << array[i][j] << " ";
7           cout << endl;
8       }
9   }
10  void show2(const float (*array)[5], int row, int col)
11  {
12      for (int i = 0; i < row; i++)
13      {
14          for (int j = 0; j < col; j++)
15              cout << array[i][j] << " ";
16          cout << endl;
17      }
18  }
19  void show3(const float* array, int row, int col)
20  {
21      for (int i = 0; i < row; i++)
22      {
23          for (int j = 0; j < col; j++)
24              cout << *(array + i * col + j) << " ";
25          cout << endl;
26      }
27  }
```

特殊调用：递归

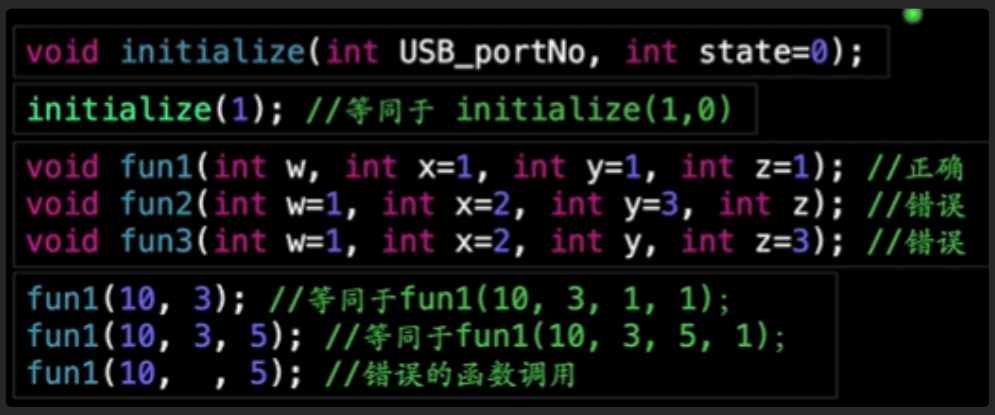
自己调用自己，一定要注意出口

```
1   void eg_20()
2   {
3       int n;
4       cout << "Enter a number: ";
5       cin >> n;
6       cout << n << "! = " << factorial(n) << endl;
```

```
    cout << n << "! = " << factorial_recursion(n) << endl;
}

int factorial(int n)
{
    int a = 1;
    for (int i = 1; i <= n; i++)
    {
        a *= i;
    }
    return a;
}

int factorial_recursion(int n)
{
    if (n <= 1) return 1;
    return n * factorial_recursion(n - 1);
}
```

函数的默认参数值

- 在函数声明或定义时，可为形参指定默认值。
  - 调用时，若省略实参，则默认值传递给形参。
  - 有多个默认参数时，应放在参数表的右部。
  - 调用函数时，若省略某实参，则该实参右边的所有实参都必须省略。



引用

引用（reference）是一种特殊类型的变量，它是另一个变量的别名。

- 声明一个引用一般采用如下格式：

```
<数据类型>&<引用名>=<变量名>;
```

- 声明引用时，必须同时对其进行初始化。
- 编译器一般将引用实现为const指针，即指向位置不可变的指针（相当于常指针），所以引用实际上与一般指针同样占用内存。
- 对引用的操作相当于对被引用变量的操作，它们代表同一个变量并且占用相同的内存单元。

```
int i = 10;
int& r = i;
r++;
cout << "i = " << i << ", r = " << r << endl;
i = 88;
cout << "i = " << i << ", r = " << r << endl;
```

```
i = 11, r = 11
i = 88, r = 88
```

- 引用作为一般变量使用几乎没有意义，主要用于将其作为函数参数。
- 采用引用传递方式，只需在函数定义时使用引用作为形参，在函数调用时直接使用一般变量作为实参
- 当引用被用作函数参数时，被调函数任何对引用的修改都将影响主调函数中的实参，被调函数对引用的操作即是通过实参的别名对实参进行操作。
- 下面的引用没有意义

```
1  void swap(const int& x);
```

特殊函数：内联函数

- 调用函数时，系统要进行现场处理工作，需要占用附加的现场处理时间。
- 解决方案：
  1. 把函数体直接嵌入函数调用处，则可消除附加的现场处理的时间开销，提高程序的运行效率。
  2. 调用内联（inline）画数时不发生控制转移，而是在编译时就将函数体嵌入到调用处。
- 内联函数的定义：
  - 在函数头前加入关键字inline
  - 当编译程序遇到内联函数调用语句时，会将该内联函数的函数体替换调用语句。
- 优缺点：
  - 加快代码调用速度。
  - 增加代码占用内存的空间开销。
- 使用说明
  - 适用于代码较短的函数。
  - 函数体内不能有循环语句和switch语句。
  - 递归函数不能作为内联函数。
  - 内联函数必须定义在主调函数之前。
  - inline修饰的函数不一定被编译为内联函数；没有被
  - inline修饰的函数也可能被编译为内联函数。

建议：不用。