一、 函数

1. 函数介绍

> 在大多数地方，c++ 和 python的函数是一样的，都是用来包裹定义好的语句，避免重复拷贝粘贴。不过还是有些许不一样的地方。
>
> 1. python的函数是以回车换行结尾，c++的函数是以 大括号结尾
> 2. python的函数通常使用缩进方式来表示函数体， ，c++使用大括号区域来表示
> 3. python是动态类型语言，而c++是静态类型语言，所以有时候需要像声明变量一样，声明函数。

python

```python
def add(a, b):
    return a + b


print(add(3, 4))
```

C++

```cpp
int add(int a , int b);

int main(){
    std::cout << add( a: 3, b: 4) << std::endl;
}


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

2. 定义函数

> 函数的定义一般可以包含以下几个部分： `方法名称`、`方法参数`、`返回值`、`方法体` ， 根据可有可无的设置，函数一般会有以下4种方式体现。

- 声明并调用函数

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

void say_hello(){
    count << "hello" << endl;
}

int main(){

    say_hello();
    return 0 ;
}
```

1. 无返回值无参数

```
void say_hello(){
    count << "你好 " << endl;
}

int main(){
    say_hello();
    return 0 ;
}
```

2. 无返回值有参数

```
#include<iostream>

using namespace std;

void say_hello(string name){
    count << "你好 "<< name << endl;
}

int main(){
    say_hello("张三");
    return 0 ;
}
```

3. 有返回值无参数

```
#include<iostream>

using namespace std;

string say_hello(){
   return "你好 张三";
}

int main(){
    cout << say_hello() << endl;
    return 0 ;
}
```

4. 有返回值有参数

```cpp
#include<iostream>

using namespace std;

string say_hello(string name){
   return "你好 "+ name;
}

int main(){
    cout << say_hello("张三") << endl;
    return 0 ;
}
```

3. 函数原型

> 一般来说，c++的函数一般包含声明和定义两个部分。因为c++是静态类型语言，程序属于自上而下编译，所以在使用函数前，必须先表示函数的存在，告诉编译器函数所需要的参数以及函数的返回值是什么。

1. 函数定义在前

> 在调用函数之前，事先先定义好函数。

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

//函数定义 ，函数的真正实现。
int add(int a , int b){
    return a + b ;
}

int main(){
    cout << add(1 ,2)<< endl;
    return 0 ;
}
```

2. 使用函数原型

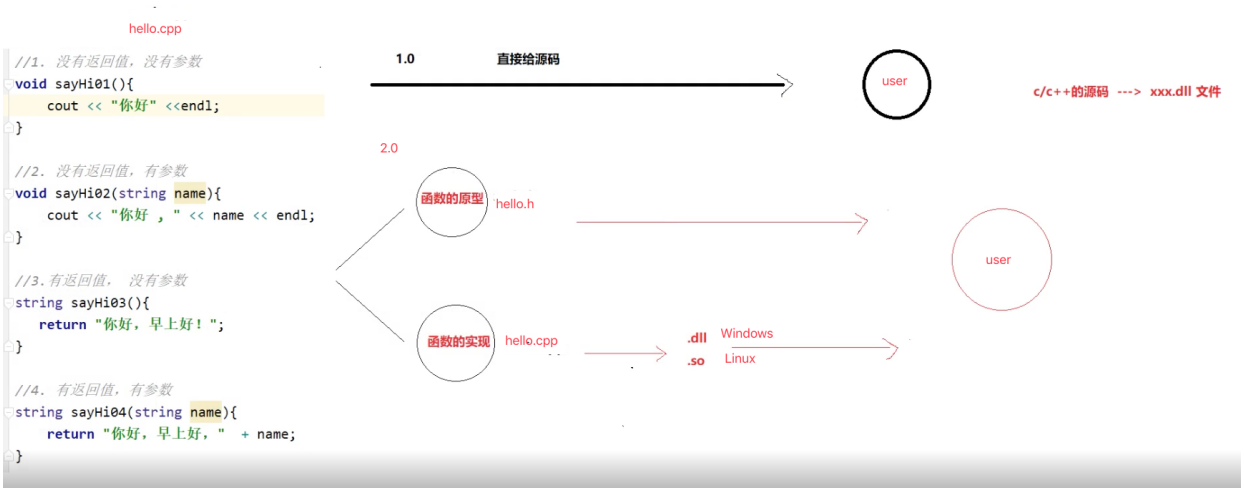
> 把函数分成声明和定义两部分，函数的原型定义在调用的前面，具体实现可以放在后面。

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

//函数声明 ，也叫函数原型 并不知道这个函数具体是如何实现的。只是有一些基本架子而已。
int add (int a , int b);

int main(){
    cout << add(1 ,2)<< endl;
    return 0 ;
}
```

```
//函数定义 ， 函数的真正实现。
int add(int a , int b){
    return a + b ;
}
```

4. 分离式编译

> 一般说来，函数的声明 (函数原型)通常都会放到头文件中，之所以称之为头文件是因为它总是在main函数的前面就引入进来。头文件一般以 .h 或者 .hpp 结尾，通常用于 写类的声明（包括类里面的成员和方法的声明）、函数原型、#define常数等，但一般来说不写出具体的实现



- math.h

> 为了能够让声明和定义能够快速的被关联上，通常它们的名称会被定义成一样的，这已经成为了一种默认的规定

```
//函数声明
int add (int a , int b);
```

- math.cpp

> 在源文件中对前面头文件的函数作出具体实现。

```
#include "math.h"

//函数定义 ， 函数的真正实现。
int add(int a , int b){
    return a + b ;
}
```

- main.cpp

```
#include <iostream>
#include "math.h" //这里使用"" 表示从当前目录查找

int main(){
    add(1 ,2);
    return 0 ;
}
```

5. 函数重载

> 在许多语言中，经常会见到两个或者两个以上的函数名称是一样的，当然他们的 **参数个数** 或者 **参数类型** 或者是 **参数的顺序** 是不一样的。这种现象有一个学名叫做 **重载** overload， 由于python属于动态类型语言，不区分数据类型，参数可以是任意类型，所以它没有重载。
>
> 下面的示例代码即是对加法运行进行了重载，以便能够针对不同的数据类型，不同的参数个数做出匹配

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

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


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

int main(){
    add(3, 3);
    add(3, 3, 3);
    add(2.5 , 2.5);

    return 0 ;
}
```

函数重载：函数名相同，参数列表不同。

```
int Max(int a,int b);
double Max(double a,double b); //实现了Max函数的重载
12
```

注意：函数重载仅以**参数列表**作为重载判断条件，**返回类型不同不构成重载！**
不以返回类型区分的原因：

- 调用时产生二义性
- （对上一点的补充）在调用时各函数均符合其调用规则，此时便无法调用

```
int Max(int a,int b);
double Max(int a,int b); //仅是返回类型不同，非重载函数

int main()
{
    //...
    int a = 1;
    int b = 2;
    Max(a,b); //错误，调用具有二义性，系统无从判断应当调用那个函数
    //...
}
1234567891011
```

另外值得注意的是，即便可以通过参数个数不同实现函数重载，但在某些设置默认值的情况下依然会因为产生二义性而错误，如下面的情况：

```
void fun(int a,int b)

void fun(int a,int b,int c = 0)//设置默认值参数

int main()
{
    fun(12,23);         //错误，产生二义性
    fun(12,23,34);     //正确
    return 0;
}
12345678910
```

函数重载是在编译时期决定的调用哪一个函数——这是C++**静多态**特性的一种表现

C++能进行函数重载的原因——编译时使用了重命名规则—即 *名字粉碎* 技术

6. 函数参数

> 实际上所有的编程语言函数传参都是采用拷贝的方式，把原有的数据拷贝给现在的参数变量，进而能够在函数中让这份数据参与计算。需要注意的是，默认情况下，参数变量得到只是原有数据的一份拷贝而已。所以无权对外部的数据进行修改。

1. 值传递

> C++默认情况下，处理函数参数传递时，多数使用的是值的拷贝，少数部分除外。

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

void  scale_number(int num);

int main(){
```

```
    int number{1000};
    scale_number(number);

    //打印number 1000
    cout << number <endl;
    return 0 ;
}

void scale_number(int num){
    if(num > 100)
        num = 100;
}
```

2. 传递数组

> 函数的参数除了能传递普通简单的数据之外，数组也是可以传递的。但是数组稍微有点特殊，这里多做讲解。
>
> 1. 前面提过，形参实际上就是实参的一份拷贝，就是一个局部变量。
> 2. 数组的数据太大，如果都进行拷贝，那么比较麻烦，也造成了浪费
> 3. 所以实际上传递数组的时候，并不会进行整个数组的拷贝，而只是传递数组的第一个元素内存地址 （指针） 进来。
> 4. 数组的数据还是在内存中，只是把第一个元素（也就是数组的起始）内存地址传进来而已。
> 5. 这就造成了函数的内部根本无法知道这个数组的元素有多少个。

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

using namespace std;
//传递数组长度
void    print_array(int number[] , 5 );

int main(){
    //声明数组
    int array []{1,2,3,4,5};

    //打印数组
    print_array(array , 5);

    return 0 ;

}

//传递数组，打印数组
void print_array(int array[] , int size){
    for (int i {0} ; i < size ; i++){
        count << array[i] << endl;
    }
}
```

3. 传递引用

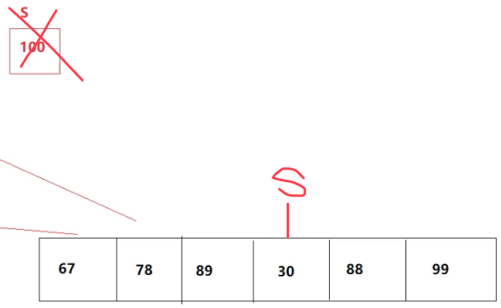
> 目前为止，我们所有函数的参数传递，都是对数据进行了一份拷贝（数组除外）。那么在函数的内部是不能修改值的，因为这仅仅是一份值得拷贝而已（函数外部的值并不会受到影响）。如果真的想在函数内部修改值，那么除了数组之外，还有一种方式就是传递`引用`。
>
> **引用实际上只是原有数据的一种别名称呼而已，使用`&`定义**

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


void  scale_number(int &num);

int main(){
    int number{1000};
    scale_number(number);

    //打印number100
    count << number <endl;
    return 0 ;
}

void scale_number(int &num){
    if(num > 100)
        num = 100;
}
```

4. 练习

> 有一个装有6个学科分数的vector，请把这个vector传给另一个函数changeScore()函数，在该函数内部 请使用基于范围的for循环对vector进行遍历，把vector里面所有低于60分的分数，修改为：100分。

```
void changeScore(vector<int>& scores){
    for(int&s : scores){
        if(s < 60){
            s = 100;
        }
    }
}

int main() {
    vector<int>  scores {67,78,89,30,88,99};
    changeScore(scores);

    for( int s:scores){
        cout <<"s===" << s <<endl;
    }
    return 0;
}
```



7 . 函数是如何被调用工作的

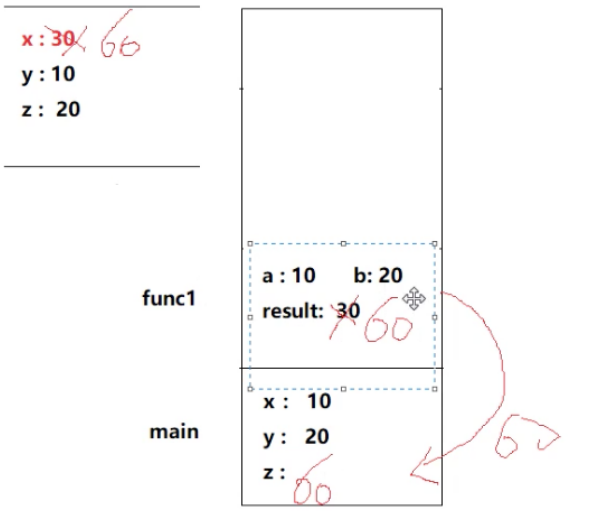
1. 函数是使用函数调用栈来管理函数调用工作的。
    1. 类似盒子的栈
    2. 遵循后进先出
    3. 可以往里面执行压栈和出栈动作（push 和 pop）
2. 栈的结构和激活记录
    1. 函数必须把它的返回值返回给调用它的函数(A ---> B)
    2. 每次函数的调用都需要创建一次激活记录，然后把它压入栈中(push)
    3. 当一个函数被调用完毕的时候，就需要从栈中弹出（pop）
    4. 函数的参数以及内部的局部变量都是存储在栈中。
3. 函数栈有可能抛出栈溢出异常（Stack Overflow）
    1. 一旦函数调用栈中被push进来的函数记录过多，就有可能出现。（例如：无限循环调用 | 递归 ）

```
void func2(int &x , int y , int z){
    x +=y+z;
}

int func1(int a , int b){
    int result{};
    result += a + b;
    func2(result , a , b );
    return result ;
}

int main (){
    int x{10};
    int y{20};
    int z{};

    z = func1(x , y );
```

```
    cout << z << endl;
    return 0 ;
}
```

图讲解过程

```
void func2(int &x , int y , int z){
    x +=y+z;
}

int func1(int a , int b){
    int result{};
    result += a + b;
    func2( &result , a , b );
    return result ;
}

int main (){
    int x{10};
    int y{20};
    int z{};

    z = func1(x , y );

    cout << z << endl;
    return 0 ;
}
```



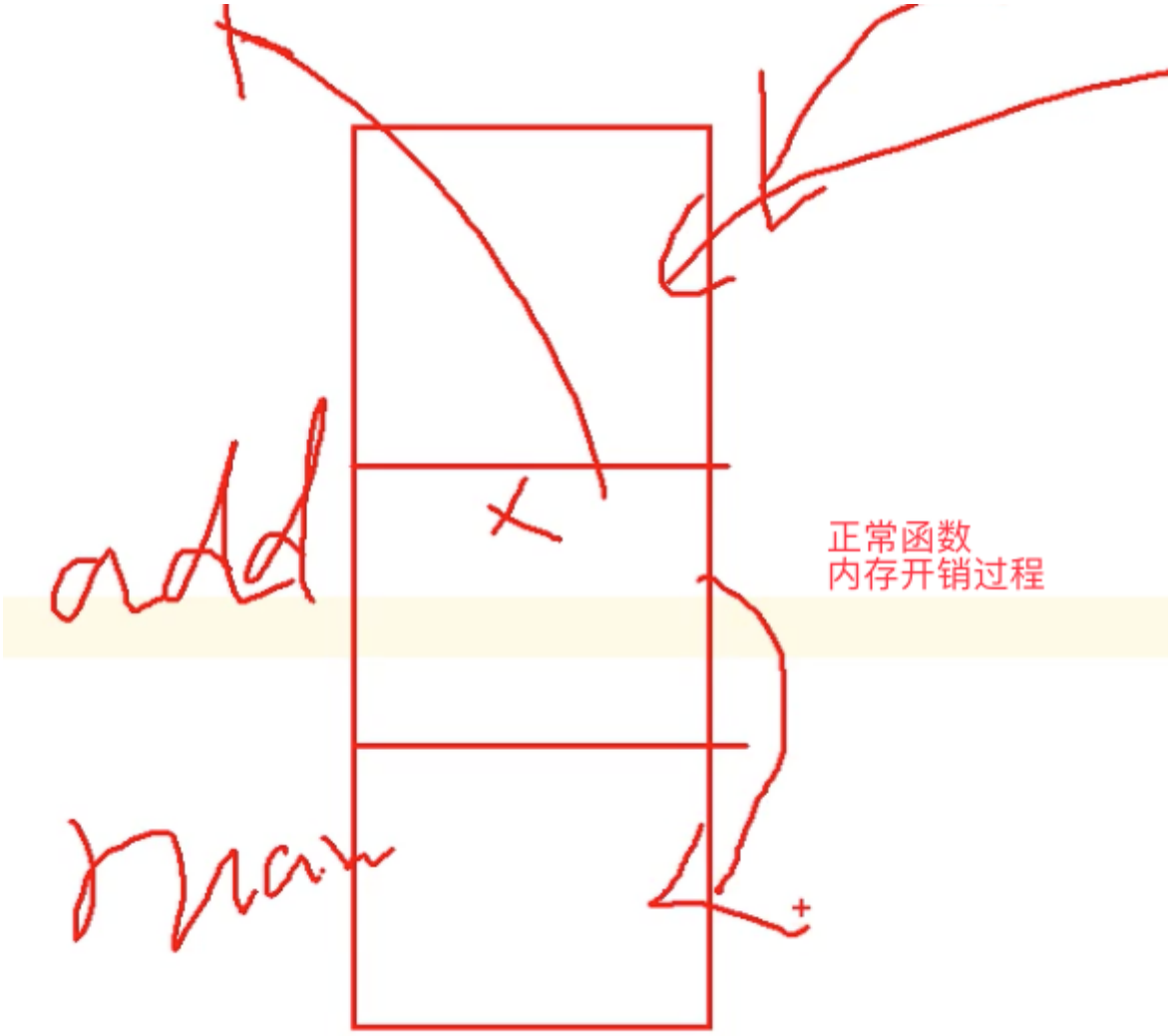
8 . 内联函数

> 函数可以使我们复用代码，但是一个函数的执行，需要开辟空间、形参和实参进行值得拷贝，还要指明函数返回、以及最后回收释放资源的动作，这个过程是要消耗时间的。

- 作为特别注重程序执行效率，适合编写底层系统软件的高级程序设计语言，如果函数中只有简单的几行代码，那么可以使用`inline`关键字来解决了函数调用开销的问题

```
#include<iostream>

inline int calc_Max (int a, int b)
{
    if(a >b)
        return a;
    return b;
}

int main(){

    int max = calc_Max(3, 8);

    std::cout << "max = " << max << std::endl;

    return 0 ;
```

```
}
```



> 增加了 inline 关键字的函数称为“内联函数”。内联函数和普通函数的区别在于：当编译器处理调用内联函数的语句时，不会将该语句编译成函数调用的指令，而是直接将整个函数体的代码插人调用语句处，就像整个函数体在调用处被重写了一遍一样。
>
> 有了内联函数，就能像调用一个函数那样方便地重复使用一段代码，而不需要付出执行函数调用的额外开销。很显然，使用内联函数会使最终可执行程序的体积增加。以时间换取空间，或增加空间消耗来节省时间，这是计算机学科中常用的方法。
>
> 1）内联函数的使用是为了解决 频繁调用小函数而大量消耗栈空间 的问题
>
> 因此可将其视为“空间换时间”的一种办法
>
> > 消耗栈空间：这里主要指现场保护与恢复，开辟栈帧，及栈帧回退的时间
> >
> > - 正常函数在调用时—
> >   1 .传参
> >   2 . call fun //调函数
> >   3 .开辟栈帧

4 .返回值返回
5 .栈帧回退
6.参数清除

2）内联函数调用

```
#include <iostream>
inline int fun(int a,int b)
{
    return c = b + a;
}

int main()
{
    int a = 1 ,b = 2;

    int c = fun(a,b);//普通函数
    int c = a + b;    //内敛函数，编译期在调用点展开

    return 0;
}
1234567891011121314
```

3）inline不总是展开
在debug版本（调试版本） 下与常规函数无差异，不展开；
在release版本（发行版本）下使用，该函数会在调用点展开（编译时期）；

注意
a. 递归函数无法被展开
（编译时无法获取变量值，因而无法实现终止条件）；

b. inline只是对系统**建议**将该函数处理为内联函数，在函数体过大（如行数大于5）或过于复杂（存在循环结构，if语句等）时，编译器会将其视为普通函数处理；少部分编译器甚至会报错

c. inline在debug版本生成的是local符号,只在本地可见；如果处理为内联之后在release版本不生成符号，直接在调用点展开。

函数	展开	调试	类型安全校验	栈帧的开辟	可见性	符号
宏函数	预编译时期在调用点展开	无法调试	无	无	单文件可见	不生成

函数	展开	调试	类型安全校验	栈帧的开辟	可见性	符号
static函数	不展开	可调试	有	有	单文件可见	生成local符号
内联函数	debug版本不展开，release版本（在编译阶段时）于调用点展开	可调试	有	debug版本有栈帧开辟；release版本没有栈帧开辟	单文件可见	debug版本生产local符号，release版本不生成符号
普通函数	不展开	可调试	有	有	多文件可见	生成global符号

> **符号**:
> 所有的数据都会生成符号；指令中只有函数名会生成符号
> 分为——1 .全局符号 **global** 符号 2.局部符号 **local** 符号
> 只有本文件可见

因为内联函数会进行类型检查与安全检查，可将其视为更安全的宏

9. 范围规则

> 在学习过程，我们会定义很多变量或者引用、这些变量由于定义的位置不同，所以它们的作用域范围也不同。一般会划分成几种类型： `代码块` | `局部变量` | `静态变量` | `全局变量`

- 单独代码块

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

int main(){

    int  num{10};
    int  num1{20};

    cout << num << num1 << endl;

    {
```

```
        int num{100};
        cout << "num = "<< num << endl;
        cout << "num1 = "<< num1 << endl;
    }

}
```

- 函数中局部变量

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

int num{300};
void local_example(int x){


    int num{1000};
    cout << "num =" << num << endl;

    num = x ;
    cout << "num =" << num << endl;


}
```

- 静态本地变量

1. 只会初始化一次
2. 重复调用函数也只会初始化一次。

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

void static_local_example(){

    static int num{100};
    cout << "num ="<< num << endl;
    num+=100;
    cout << "num ="<< num << endl;
}

int main(){
    static_local_example();
    static_local_example();
    return 0 ;
}
```

- 全局变量

> 通常声明在所有函数和类的外部 ,若存在局部变量和全局变量同名情况下，可以使用 域操作符 `::` 来访问全局变量

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

int age = 99;
int main(){

    int age =18 ;
    cout << ::age << endl;
    return 0 ;
}
```

10. 打卡作业

1. 有两个字符串 A， B， A 为源字符串，B 为要删除的字符串，判断A是否包含B，如果包含，请把A 里面包含B的字符删除后，输出全新的字符串A，否则直接输出源字符串A。

```
string a = "ab123def78cc09"; //源字符串
string b = "cc";//要删除字符串
字符串的删除函数： a.delete()  | a.remove() | a.erase() 这三个函数，必有一个!
```

2. 定义一个计算器，提供加减乘除功能
3. 定义一个calc.h 作为加减乘除 四个函数的声明
4. 定义一个calc.cpp 作为计算器的具体实现
5. 定义一个main.cpp 作为程序的入口

综合演练题目:

1. 使用二维vector 用于保存张三、李四、王五，三人的6个学科的成绩，

   定义一个二维vector，名叫：score_vector，用来存储三个学生的6个学科成绩
2. 定义一个initScore（vector<vector>）函数负责从键盘录入成绩 获取3个人，6个学科成绩
3. 定义一个函数updateScore（vector<vector>），用于更新分数，把每个人的不及格的成绩全部修改成99分，
4. 定义一个函数 printScore （vector<vector>），遍历打印三个人的每个学科成绩。
5. 要求使用函数原型、分离式写法（头文件和源文件）、函数传递引用，基于范围for循环。
6. 禁止定义全局静态vector，要求2，3，4 步骤的函数要携带参数，把二维vector传递进去。
7. 超纲： 应该要考虑引用，否则更新的操作无法实现。具体使用可看上面的引用。
8. 应该有3个文件 stu.h 、stu.cpp , main.cpp
    1. stu.h :用于声明三个函数
    2. stu.cpp用于实现三个函数
    3. main.cpp 用于程序的执行入口

```
```

```
vector<vector<int>> score_vector;

int main (){

    initScore( score_vector ); //往这个二维vector里面装成绩

    updateScore( score_vector ); // 遍历二维vector，更新成绩  【必须要用引用 | 指针】

    printScore( score_vector ); //打印成绩。

    //超纲：  引用。
    return 0 ;
}
```