一: 冒号作用域

:: 运算符是一个作用域,如果::前面什么都没有加 代表是全局作用域

```
#include <iostream>
using namespace std;
int a = 100;
```

```cpp
void test01()
{
 int a = 10;
 cout << a << endl;//输出局部变量a
 cout << ::a << endl;//输出全局变量a
}
int main()
{
 test01();
 return 0;
}
```

二: 名字控制

1 命令空间

namespace

本质是作用域,可以更好的控制标识符的作用域

命名空间 就可以存放 变量 函数 类 结构体 ...

2 命令空间的使用

命令空间的定义 必须定义在全局范围

命名空间下可以存放 变量 函数 结构体 类

命名空间可以重名 重名的命名空间相当于做合并操作

命名空间可以嵌套命名空间

命名空间可以取别名, namespace newname = oldname;

命名空间可以没有名字 ,没有名字相当于给命名空间 内的所有成员加上了static修饰

命名空间中的函数可以先声明,在外部定义,定义时需要加上命名空间作用域

```cpp
#include <iostream>
using namespace std;
// 命令空间的定义 必须定义在全局范围
// 命名空间下可以存放 变量 函数 结构体 类
// 命名空间可以重名 重名的命名空间相当于做合并操作
// 命名空间可以嵌套命令空间
namespace A
{
 int a = 1;
 void fun()
```

```
 {
 cout << "hello namespace" << endl;
 }
 void foo(int agr);
 struct std
 {};
 class obj
 {};
}
void A::foo(int arg)
{
 cout << arg << endl;
}

//命名空间是可以取别名
// namespace newname = oldname
namespace newA = A;

namespace B
{
 int a = 10;
 int b = 20;
}
namespace B
{
 int c = 30;
}
namespace C
{
 int a = 10;
 int b = 20;
 namespace D
 {
 int a = 100;
 }
}

//注意: 如果命名空间没有名字 name这个命名空间内的所有成员都被编译器加上了static修饰
//只能被当前文件调用 这个属于内部链接属性
```

```
namespace {

 int a = 10;
 void func() { cout << "hello namespace" << endl; }
}

void test03()
{
 A::foo(222);
 newA::foo(333);

}
void test02()
{
 cout << C::a << endl;
 cout << C::D::a << endl;

}
void test01()
{
 cout <<A::a << endl;
 cout << B::a << endl;
 cout << B::b << endl;
 A::fun();
 cout << B::c << endl;

}
int main()
{
 test03();
 return 0;
}
```

3 using的声明

usinng 的声明可以使得指定标识符可用

注意: 当using声明的标识符和其他同名标识符有作用域的冲突时,会产生二义性

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

namespace nameA
```

```
{
 int a = 10;
 void foo()
 {
 cout << "hello using" << endl;
  }
}
void test01()
{
 //注意: 当using声明的标识符和其他同名标识符有作用域的冲突时,会产生二义性
 int a = 100;
 using nameA::a;
 using nameA::foo;
 cout << a << endl;
 cout << a << endl;
 cout << a << endl;

 foo();

}
int main()
{

 test01();
 return 0;
}
```

4 using的编译指令

using编译指令使整个命名空间标识符可用.

并且命名空间标识符如果和局部变量的标识符同名,不会有冲突,优先使用局部变量

```
namespace nameA
{
 int a = 10;
 void foo()
 {
 cout << "hello using" << endl;
 }
}
void test02()
```

```
{
 int a = 1000;
 //using编译指令使整个命名空间标识符可用.
 using namespace nameA;
 cout << a << endl;
 foo();

}
```

三:全局变量的检测增强

c++的编译器对于全局变量的声明和定义有严格的区分,检测会增强

c语言下的全局变量的声明和定义

```
//全局变量
int a; // 定义
int a;//声明
int a;//声明
```

c++语言下的全局变量的声明和定义

```
//全局变量
int a;
extern int a;
extern int a;
```

c++语言下的全局变量的声明和定义,如果写成以下形式,编译器编译时通不过

```
//全局变量
int a;
int a;
int a;
```

四:C++中形参必须有类型,返回值和实参个数做检测

c语言中的函数的形参类型可以不写,没有返回值可以返回,实参的个数不做检测

```
void foo(x,y)
{
```

```
 return 100;
}
void test01()
{
 foo(1);
 foo(1, 2);
 foo(1,2,3);

}
```

c++语言中的函数的形参类型必须写,没有返回值不可以返回,实参的个数做检测

```
void foo(x, y) // 编译器报错 形参没有类型
{
 return 100; //编译器报错 没有返回值但是返回了
}
void test01()
{
 foo(1);//实参的个数和形参的个数不一致
 foo(1, 2);
 foo(1, 2, 3);//实参的个数和形参的个数不一致

}
```

五:更严格的类型转换

c++中对类型转换有严格的要求,需要的类型和给的类型不一致时,可能会编译保存

例如.c语言中这段代码可以编译通过:

```
void test02()
{
 char * p = malloc(100);

}
```

但是在c++中这段代码编译不通过,需要做类型转换

```
void test02()
{
 char * p = (char*)malloc(100);

```

```
}
```

六:struct类型增强

在c++中使用结构体类型时,可以不写struct关键字

例如c语言中:

```
struct stu
{
 int a;
 int b;
};

void test03()
{
 struct stu obj;
}
```

在c++中

```
struct stu
{
 int a;
 int b;
};

void test03()
{
 stu obj;
}
```

七:新增bool类型关键字

c++中可以直接使用bool类型

在c语言中,一下代码中的bool类型,需要包含stdbool.h头文件,但是在c++可以直接使用

```
void test04()
{
 // bool类型的变量只有两个值 true false
 //true 和false 可以直接当成常量来用
 bool flag = true;
```

```
6
7 }
```

八:三目运算符功能增强

c++中的三目运算符表达式返回的可以是一个变量,但是c语言中返回的是一个常量

c语言中:

```
1 //三目运算符
2 void test05()
3 {
4  int a = 10;
5  int b = 20;
6  printf("%d\n", a < b ? a : b);
7  //在c语言中三目运算符返回的是表达式的值,是一个常量
8  //(a < b ? a : b) = 100; 编译报错
9  *(a < b ?&a :&b) = 100;
10
11 }
```

c++中:

```
1 //三目运算符
2 void test05()
3 {
4  int a = 10;
5  int b = 20;
6  printf("%d\n", a < b ? a : b);
7  //在c++语言中三目运算符返回的是变量
8  (a < b ? a : b) = 100;//编译可通过
9
10 }
```

九:c/c++中的const

1 const概述

const 修饰的对象为一个常量,不能被改变

2 c/c++中const的区别

2.1c语言中的const

1 const修饰的局部变量,存在栈区,虽然不能通过const修饰的变量去修改栈区内容,但是可以通过地址去修改

2 const修饰的全局变量是保存在常量区,不能通过变量名去修改.也不能通过地址去修改

3 const修饰的全局变量,如果其他文件想使用,直接extern声明外部可用即可

main.c

```c
#include <stdio.h>
const int b = 10;//const修饰的全局变量保存在常量区

void test03()
{
 extern const int num;//声明num是外部可用的
 printf("num=%d\n",num);

}
 //const修饰的全局变量
 void test02()
 {
  //b = 100;
  int *p = &b;
  *p = 100;//错误的 不能修改常量区的内容
  printf("b=%d\n",b);
 }
 //const修饰的局部变量
 void test01()
 {
  //在c语言中const修饰的变量保存在栈区
  const int a = 10;
  //a = 100;
  int *p = &a;
  *p = 100;
  printf("a=%d\n",a);
 }
 int main()
 {
  test03();
  return 0;
 }
```

test.c

```
const int num = 1;
```

2.2 c++中的const

1 const修饰的局部变量赋值常量时,局部变量保存在符号表中,修改不了,是一个常量

2 const修饰的全局变量保存在常量区,不能被修改

3 const修饰的全局变量默认是内部链接属性,加上extern修饰变成外部链接属性

main.c

```
#include <iostream>
using namespace std;
const int b = 1;

void test03()
{
 extern const int num;
 cout << num << endl;

}
void test02()
{
 //const修饰的全局变量存在常量区
 int *p = (int *)&b;
 *p = 100;//错误的
 cout << b<< endl;

}


//c++中const修饰的局部变量
void test01()
{
 //c++中const修饰的局部变量存在符号表中
 const int a = 10;
 //a = 100;
 //对const修饰的局部变量取地址  编译器会产生一个临时变量来保存a的地址
 // int tmp = a; int *p = &tmp
 int *p = (int *)&a;
 *p = 100;
 cout << a << endl;
```

```
}
int main()
{

 test03();
 return 0;
}
```

test.cpp

```
extern const int num = 1;// const修饰的全局变量默认是内部链接属性
```

2.3 c/c++中的const异同

相同的点:

- c和c++中的const修饰的全局变量都是保存在常量区,不能被修改

不同的点:

- c语言中const修饰的局部变量赋值为常量时,,局部变量保存在栈区,可以被指针修改
- c++中,const修饰的局部变量赋值为常量时,局部变量保存符号表中,不能被修改
- c语言中const修饰的全局变量默认是外部链接属性
- c++语言中const修饰的全局变量默认是内部链接属性

c++中const修饰的变量,分配内存情况

- const修饰的全局变量在常量区分配了内存
- 对const修饰的局部变量赋值为常量时,对其取地址,会在栈区分配临时的内存空间
- const修饰的局部变量赋值为变量时,局部变量保存在栈区
- const修饰的局部变量时一个自定义变量,也是在栈区分配内存

只有一种情况,const'修饰的局部变量被赋值为常量时,这个局部变量保存在符号表中

```
#include <iostream>
using namespace std;
const int a = 1;//const修饰的全局变量在常量区分配了内存
void test01()
{
 const int a = 10;//const修饰的局部变量赋值为常量没有分配内存,存在符号化表中
 int *p = (int *)&a;//对const修饰的局部变量赋值为常量的变量取地址 会分配一个临时的空间 int tmp =a *p =&tmp
}
```

```
void test02()
{
 int b = 2;
 //const修饰的局部变量赋值变量时 局部变量存在栈区
 const int a = b;
 int *p = (int *)&a;
 *p = 100;
 cout << a << endl;

}
struct stu
{
 int a;
 int b;
};
void test03()
{
 //const修饰的变量为自定义变量时,保存在栈区
 const struct stu obj = {1,2};
 struct stu *p = (struct stu *)&obj;
 p->a = 3;
 p->b = 4;
 cout << obj.a << " " << obj.b << endl;

}
int main()
{
 test03();

 return 0;
}
```

3 尽量以const替换define

有两点原因:

1. const修饰的全局变量或const修饰的局部变量赋值为常量,是有类型的,而define的宏没有 类型
2. const修饰的全局变量或const修饰的局部变量赋值为常量有作用域的,而define的宏没有作用域

```cpp
#include <iostream>
using namespace std;
namespace A
{
 const int max = 1024;
 const short max1 = 1024;
 #define MAX 1024
}
// 宏没有作用域 宏没有类型(int)
 void fun(int a)
 {

 }
 void fun(short a)
 {


 }

 void test01()
 {
  cout << A::max << endl;
  cout << MAX << endl;
  fun(MAX);//void fun(int a)
  fun(A::max);//void fun(int a)

  fun(A::max1);//void fun(short a)
 }
 int main()
 {

  return 0;
 }
```

十:引用

1.引用的基本用法

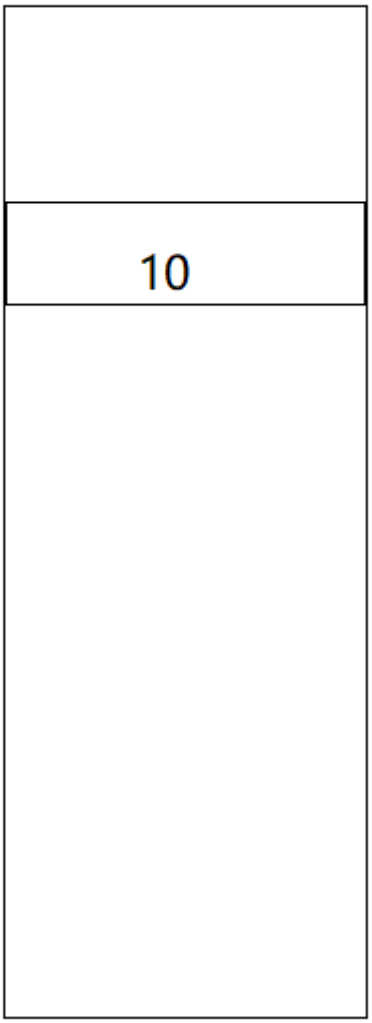
1.1 引用的实质

原类型 &别名 = 旧名

```
int a =10;
int  *p = &a;
*p
```

a

b



引用的本质是取别名:

原类型名 &别名 = 旧名

```
int   &b =  a;
b  =100;
```

1.2 引用的用法

注意事项:

- 引用一旦初始化,不能更改引用的指向
- 引用定义时必须初始化
- 不能引用NULL
- 引用可以引用任意类型包括数组
- &在等号的左边是引用,在等号的右边是取地址

```
#include <iostream>
using namespace std;
void test01()
{
 int a = 10;
 //引用一旦初始化之后不能改变引用的标识
```

```
 int &b = a;
 b = 100;
 cout << a << endl;
  int c = 1;
  //b = c; 代表把c的值赋值给b 不是给c取别名为b
  //int &d; 引用定义时必须初始化
}
void test02()
{
  int a[5] = { 1,2,3,4,5 };
  //int(&arr)[5] = a;
  typedef int ARR[5];
  //type & 别名 = 旧名
  ARR & arr = a;
  for (int i = 0; i < 5; i++)
  {
  cout << arr[i] << " ";

  }
  cout << endl;

}
int main()
{
  test02();

  return 0;
}
```

2.函数中的引用

- 引用可以作为函数的形参
- 不能返回局部变量的引用

```
#include <iostream>
#include <stdlib.h>
using namespace std;

//形参是引用
void swap(int *x, int *y)
```

```
{
 int tmp = *x;
 *x = *y;
  *y = tmp;
}
void test01()
{
 int a = 10;
 int b = 20;
 swap(&a,&b);
 cout << a << " " << b << endl;

}
void swap_ref(int &x, int &y)// int &x =a, int &y =b
{
 int tmp = x;
 x = y;
 y = tmp;

}
void test01_ref()
{
 int a = 10;
 int b = 20;
 swap_ref(a, b);
}
//形参是指针 引用
void get_mem(int **q)
{
 *q = (int *)malloc(5 * sizeof(int));
}
void get_mem_ref(int * &q)//int * (&q) = p
{
 q = (int *)malloc(5 * sizeof(int));
}
void test03()
{
 int *p = NULL;
 get_mem(&p);
 get_mem_ref(p);
```

```
}
//int &t =a
int & test04()
{
 //能不能返回一个变量的引用 看这个变量的空间是否被释放了
 static int b = 100;
 int a = 10;
 //return a;//err 不能返回局部变量的引用
 return b;//可以返回静态的变量的引用

}
int main()
{
 //test01();
 //test01_ref();
 test04() = 1000;
 return 0;
}
```

3.引用的本质

引用的本质是一个指针常量

type &b = a; 编译器底层这么实现的:

type *const b = &a;

```
#include <iostream>
using namespace std;
void test01()
{
 int a = 10;
 int &b = a;//编译器优化 int* const b = &a
 //指针常量 不能改变指针变量的指向
 // b =0x100;

 b = 1000;// *b =1000

}
void fun(int *&q)//int *&q = p ==> 编译器 int * const q =&p
{
```

```
15
16 }
17 void test02()
18 {
19  int *p = NULL;
20  fun(p);
21
22 }
23 int main()
24 {
25
26
27
28  return 0;
29 }
```

4.指针的引用

套用引用公式: type &q = p

假设:

type为指针类型

```
1 void fun (int* &q) // int* &q = p
2 {
3
4 }
5 void test()
6 {
7  int *p=NULL;
8  fun(p);
9 }
```

5 常量引用

const type &p = q;

常量引用代表不能通过引用去修改引用标识的那块空间

```
1 #include <iostream>
2 using namespace std;
3 void test01()
```

```
{
 int a = 10;
 // const修饰的是引用& 不能通过引用去修改引用的这块空间的内容
 const int &b = a;
 //b = 1000;//err
}

void test02()
{
 //int &b = 100;//不能引用常量
 const int &b = 1;//int tmp =1 ,const int &b= tmp

}
int main()
{

 return 0;
}
```

十一:内联函数

1 为什么要有内联函数

第一个在c中也会出现，宏看起来像一个函数调用，但是会有隐藏一些难以发现的错误。

第二个问题是c++特有的，预处理器不允许访问类的成员，也就是说预处理器宏不能用作类类的成员函数。

内联函数就是继承了宏函数的高效,并且不会出错,还可以当成类的成员函数用

2 宏函数和内联函数的区别

宏函数的替换是发生在预处理阶段

内联函数的替换是发生在编译阶段

宏函数容易出错,内联函数不会

内联函数和宏函数一样,都省去了调用函数的开销

```
#include <iostream>
using namespace std;
#define MYADD(a,b) a+b

inline int myadd(int a, int b)
{

```

```
 return a + b;
}

void test01()
{
 int a = 10;
 int b = 20;
 //int c = MYADD(a, b)*5;// a+b*5 替换发生在预处理阶段
 int c = myadd(a, b) * 5;// (a + b)*5 替换发生在编译阶段 也和宏函数一样不会有函数调用的开销
 cout << c << endl;
}
int main()
{
 test01();
 return 0;
}
```

```
#define MYCOMPARE(a,b) (a)<(b)?(a):(b)

inline int mycpmpare(int a, int b)
{
 return a < b ? a : b;
}
void test02()
{
 int a = 1;
 int b = 5;
 //int c = MYCOMPARE(++a, b);// ++a<b?++a:b
 int c = mycpmpare(++a, b);
 cout << c << endl;

}
```

3. 内联函数

3.1 内联函数的基本概念

在c++中，预定义宏的概念是用内联函数来实现的，而内联函数本身也是一个真正的函数。内联函数具有普通函数的所有行为。唯一不同之处在于内联函数会在适当的地方像预定义宏一样展开，所以不需要函数调用的开销。因此应该不使用宏，使用内联函数。

在普通函数(非成员函数)函数前面加上inline关键字使之成为内联函数。但是必须注意必须函数体和声明结合在一起，否则编译器将它作为普通函数来对待。

inline void func(int a);

以上写法没有任何效果，仅仅是声明函数，应该如下方式来做:

inline int func(int a){return ++;}

注意: 编译器将会检查函数参数列表使用是否正确，并返回值(进行必要的转换)。这些事预处理器无法完成的。

内联函数的确占用空间，但是内联函数相对于普通函数的优势只是省去了函数调用时候的压栈，跳转，返回的开销。我们可以理解为内联函数是以空间换时间。

3.2 类的成员函数默认编译器会将它做成内联函数

任何在类内部定义的函数自动成为内联函数。

```
class Person{
public:
Person(){ cout << "构造函数!" << endl; }
void PrintPerson(){ cout << "输出Person!" << endl; }
}
```

3.3 内联函数和编译器

内联函数并不是何时何地都有效，为了理解内联函数何时有效，应该要知道编译器碰到内联函数会怎么处理？

对于任何类型的函数，编译器会将函数类型(包括函数名字，参数类型，返回值类型)放入到符号表中。同样，当编译器看到内联函数，并且对内联函数体进行分析没有发现错误时，也会将内联函数放入符号表。

当调用一个内联函数的时候，编译器首先确保传入参数类型是正确匹配的，或者如果类型不正完全匹配，但是可以将其转换为正确类型，并且返回值在目标表达式里匹配正确类型，或者可以转换为目标类型，内联函数就会直接替换函数调用，这就消除了函数调用的开销。假如内联函数是成员函数，对象this指针也会被放入合适位置。

类型检查和类型转换、包括在合适位置放入对象this指针这些都是预处理器不能完成的。

但是c++内联编译会有一些限制，以下情况编译器可能考虑不会将函数进行内联编译：

不能存在任何形式的循环语句

不能存在过多的条件判断语句

函数体不能过于庞大

不能对函数进行取址操作

内联仅仅只是给编译器一个建议，编译器不一定会接受这种建议，如果你没有将函数声明为内联函数，那么编译器也可能将此函数做内联编译。一个好的编译器将会内联小的、简单的函数。

十二:函数的默认参数

c++中的函数,形参可以设置默认参数,设置时需要注意以下几点:

- 设置默认参数时,某个参数设置了默认参数,从这个参数开始,后面的每一个都要设置
- 函数的声明和定义处设置默认参数只能一处设置
- 有实参传入则使用实参,实参没有传入使用默认参数

```
#include <iostream>
using namespace std;
//void fun01(int a = 1, int b = 2);
//默认参数 在设置时 声明和定义只能一处设置默认参数
//设置默认参数时 有一个参数设置了默认参数,从这个参数往后的每一个参数都要设置默认参数
void fun01(int a=1,int b=2)
{
 cout << a << " "<<b << endl;
}
 void test01()
 {
  fun01(1,2);
  fun01(1, 3);
  fun01(1, 4);
  fun01(1);
  fun01();
 }
int main()
{
 test01();
 return 0;
}
```

十三: 占位参数

占位参数给函数形参设置的,调用时需要传参,也可以设置占位参数为默认参数;

占位参数在符号重载++时会用到

```
#include <iostream>
using namespace std;
void fun(int a, int=4)
{

}
void test01()
{
 fun(1,2);
  fun(2);
 }
 int main()
 {


  return 0;
 }
```

十四: 函数的重载

1 函数的重载:

在c++中函数的名字是刻印重名的,也就是可以有多个相同函数名的函数存在,

重载: 名字相同,意义不一样

2函数重载的条件

实现函数重载的条件:

同一个作用域

参数个数不同

参数类型不同

参数顺序不同

函数的返回值不能成为函数重载的条件

默认参数作为函数重载的条件可能发生二义性

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

// 函数名一样 意义 不一样
//函数的重载发送必须在同一个作用域
//参数的个数不一样可以发送函数的重载
//参数 的类型不一样可以发生函数的重载
```

```
//参数的顺序不一样可以发生函数的重载
//返回值不能作为函数重载的条件
//默认参数作为函数重载的条件需要注意二义性

void fun(double a, int b)
{
 cout << " fun(double a, int b)" << endl;
}
void fun( int a, double b)
{
 cout << " void fun( int a, double b)" << endl;
}
void fun(double a)
{
 cout << " void fun(double a)" << endl;
}

void fun(int a)
{
 cout << "void fun(int a) " << endl;
}
//int fun(int a)
//{
// cout << "void fun(int a) " << endl;
//}
void fun(int a,int b)
{
 cout << "void fun(int a,int b) " << endl;
}
//void fun(int a, int b=2)
//{
// cout << "void fun(int a,int b) " << endl;
//}

void test01()
{
 fun(1);
 fun(2,3);
 fun(3.14);
 fun(1, 3.14);
```

```
 fun(3.14, 1);
}
int main()
{
 test01();
 return 0;
}
```

3.函数重载的本质

编译器为了实现函数重载，也是默认为我们做了一些幕后的工作，编译器用不同的参数类型来修饰不同的函数名，比如void func(); 编译器可能会将函数名修饰成*func，当编译器碰到void func(int x),编译器可能将函数名修饰为*func*int,当编译器碰到void func(int x,char c),编译器可能会将函数名修饰为*func*int*char我这里使用”可能”这个字眼是因为编译器如何修饰重载的函数名称并没有一个统一的标准，所以不同的编译器可能会产生不同的内部名。

void func(){}

void func(int x){}

void func(int x,char y){}

以上三个函数在linux下生成的编译之后的函数名为:

_Z4funcv //v 代表void,无参数

_Z4funci //i 代表参数为int类型

_Z4funcic //i 代表第一个参数为int类型，第二个参数为char类型

4 extern "C"浅析

c++编辑器编译.c的函数时,需要声明为 extern "C"

main.cpp

```
#include <iostream>
#include "test.h"
using namespace std;

int main()
{
 cout << myadd(2, 3) << endl;
 return 0;
}

```

test.c

```c
int myadd(int a, int b)
{

 return a + b;
}
```

test.h

```c
#pragma once

#if __cplusplus
extern "C" {
#endif

 int myadd(int a, int b);


#if __cplusplus
}
#endif

```