基础知识

数据类型

基本数据类型

`1.char 2. int 3. short 4. long 5. float 6.double 7.struct(自定义数据类型) 8. enum(枚举) 9.union(联合) 10.class(类) 11.template (类模板) 12.reference(引用) 13.void类型 14. 指针类型`

类型（64位系统下）	位	范围
char	1 个字节	-128 到 127 或者 0 到 255
unsigned char	1 个字节	0 到 255
signed char	1 个字节	-128 到 127
int	4 个字节	-2147483648 到 2147483647
unsigned int	4 个字节	0 到 4294967295
signed int	4 个字节	-2147483648 到 2147483647
short int	2 个字节	-32768 到 32767
unsigned short int	2 个字节	0 到 65,535
signed short int	2 个字节	-32768 到 32767
long int	8 个字节	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
signed long int	8 个字节	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807
unsigned long int	8 个字节	0 到 18,446,744,073,709,551,615
float	4 个字节	精度型占4个字节（32位）内存空间，+/- 3.4e +/- 38 (~7 个数字)
double	8 个字节	双精度型占8 个字节（64位）内存空间，+/- 1.7e +/- 308 (~15 个数字)
long double	16 个字节	长双精度型 16 个字节（128位）内存空间，可提供18-19位有效数字。
wchar_t	2 或 4 个字节	1 个宽字符

signed和unsigned可以用来修饰char型和int型（包括long int），不能修饰bool、float、double和long double。

运算符优先级

运算符优先级：！ > 算术运算符 > 关系运算符 > （&& | |） > 条件运算符> 赋值运算符 > 逗号运算符。

优先级	操作符	描述	例子	结合性
1	() [] -> . :: ++ --	调节优先级的括号操作符 数组下标访问操作符 通过指向对象的指针访问成员的操作符 通过对象本身访问成员的操作符 作用域操作符 后置自增操作符 后置自减操作符	(a + b) / 4; array[4] = 2; ptr->age = 34; obj.age = 34; Class::age = 2; for(i = 0; i < 10; i++) ... for(i = 10; i > 0; i--) ...	从左到右
2	! ~ ++ -- - + * & (type) sizeof	逻辑取反操作符 按位取反(按位取补) 前置自增操作符 前置自减操作符 一元取负操作符 一元取正操作符 解引用操作符 取地址操作符 类型转换操作符 返回对象占用的字节数操作符	if(!done) ... flags = ~flags; for(i = 0; i < 10; ++i) ... for(i = 10; i > 0; --i) ... int i = -1; int i = +1; data = *ptr; address = &obj; int i = (int) floatNum; int size = sizeof(floatNum);	**从右到左**
3	->* .*	在指针上通过指向成员的指针访问成员的操作符 在对象上通过指向成员的指针访问成员的操作符	ptr->*var* = *24; obj.*var = 24;	从左到右
4	* / %	乘法操作符 除法操作符 取余数操作符	int i = 2 * 4; float f = 10 / 3; int rem = 4 % 3;	从左到右
5	+ -	加法操作符 减法操作符	int i = 2 + 3; int i = 5 - 1;	从左到右
6	<< >>	按位左移操作符 按位右移操作符	int flags = 33 << 1; int flags = 33 >> 1;	从左到右
7	< <= > >=	小于比较操作符 小于或等于比较操作符 大于比较操作符 大于或等于比较操作符	if(i < 42) ... if(i <= 42) ... if(i > 42) ... if(i >= 42) ...	从左到右
8	== !=	等于比较操作符 不等于比较操作符	if(i == 42) ... if(i != 42) ...	从左到右
9	&	按位与操作符	flags = flags & 42;	从左到右

优先级	操作符	描述	例子	结合性
10	^	按位异或操作符	flags = flags ^ 42;	从左到右
11	|	按位或操作符	flags = flags | 42;	从左到右
12	&&	逻辑与操作符	if(conditionA && conditionB) ...	从左到右
13	||	逻辑或操作符	if(conditionA || conditionB) ...	从左到右
14	? :	三元条件操作符	int i = (a > b) ? a : b;	**从右到左**
15	= += -= *= /= %= &= ^= |= <<= >>=	赋值操作符 复合赋值操作符(加法) 复合赋值操作符(减法) 复合赋值操作符(乘法) 复合赋值操作符(除法) 复合赋值操作符(取余) 复合赋值操作符(按位与) 复合赋值操作符(按位异或) 复合赋值操作符(按位或) 复合赋值操作符(按位左移) 复合赋值操作符(按位右移)	int a = b; a += 3; b -= 4; a *= 5; a /= 2; a %= 3; flags &= new_flags; flags ^= new_flags; flags |= new_flags; flags <<= 2; flags >>= 2;	**从右到左**
16	,	逗号操作符	for(i = 0, j = 0; i < 10; i++, j++) ...	从左到右

例子

```
    int a[] = { 1,2,3 };
    int b[] = { 4,5,6 };
    int c[] = { 7,8,9 };
    int* arr[] = { a,b,c };
    cout << *arr[1] << endl;
    cout << (*arr)[1] << endl;
输出：  4 2
```

转义字符

转义字符	意义	ASCII码值(十进制)
\a	响铃(BEL)	007
\b	退格(BS)	008
\f	换页(FF)	015
\n	换行(LF)	010
\r	回车(CR)	013
\t	水平制表(HT)	009
\v	垂直制表(VT)	011
\\	反斜杠	092
\?	问号字符	063
\'	单引号字符	039
\"	双引号字符	034
\0	空字符(NULL)	000
\ddd	任意字符	三位八进制
\xhh	任意字符	二位十六进制

“\”加数字（一般是8进制数字）来表示。

常规的ASCII码，最大值是为0x7f，后面的从0x80到0xff为扩展ASCII码，不是标准的ASCII码.这些字符是用来表示框线、音标和其它欧洲非英语系的字母。

合法标识符，合法常量，合法转义字符

一， 合法标识符
用户定义的合法标识符需满足以下两个要求：

标识符只能由字母，数字和下划线组成。
标识符不能以数字开头。
二， 合法常量

整型常量：
十进制： 10
八进制： 017（以0开头，不能出现8，9）
十六进制：0xA1（以0x开头）

实型常量： （强调E）
E的前面必须有数字，E的后面必须是整数

字符型： 'n' ,'N' ,'\n'

字符串型："abc","123",""

三，合法转义字符

一般转义字符
\a 响铃
\b 退格
\f 换页
等

八进制转义字符
它是由反斜杠\ 和 随后的1～3个八进制数字构成的字符序列
十六进制转义字符
它是由反斜杠\ 和 字母x(或X)及随后的1～2个十六进制数字构成的字符序列
八进制转义字符和十六进制转义字符，不在前面加0！

ASCII

ASCII中 二进制数加后缀B，八进制数加后缀Q，十进制数加后缀D,十六进制数加后缀H,其中十进制的后缀D可以省略（默认）

隐显转换

- 隐式转换：char->int->long->double 或 float->double（系统根据需要而自动转换）
- 显示转换：强制类型转换
- 尽量用显示转换代替隐式转换

开发一个C++程序的过程

通常包括编辑、编译、链接、运行和调试等步骤。

GOTO

goto语句也称为无条件转移语句，其一般格式如下： goto 语句标号； 其中语句标号是按标识符规定书写的符号， 放在某一语句行的前面，标号后加冒号(：)。语句标号起标识语句的作用，与goto 语句配合使用。

如：

label: i++;

```
int main() {
    int x = 0;
loop:cout << "第" << x << "次到loop" << endl;
    if (x < 7) {
        x++;
        goto loop;
    }
    return 0;
}
输出：
    第0次到loop
    第1次到loop
    第2次到loop
    第3次到loop
```

```
第4次到loop
第5次到loop
第6次到loop
第7次到loop
```

go to语句使用原则：

1、使用goto语句只能goto到同一函数内，而不能从一个函数里goto到另外一个函数里。

2、使用goto语句在同一函数内进行goto时，goto的起点应是函数内一段小功能的结束处，goto的目的label处应是函数内另外一段小功能的开始处。

3、不能从一段复杂的执行状态中的位置goto到另外一个位置，比如，从多重嵌套的循环判断中跳出去就是不允许的。

4、应该避免向两个方向跳转。这样最容易导致"面条代码"。

内联函数

1. 内联函数在编译时将被调用函数代码直接编译到主调函数中，因此不会产生"函数调用"开消。它不是指在一个函数内部定义另一个函数。
2. 在C++中使用inline关键字来定义内联函数。inline关键字放在函数定义中函数类型之前。不过编译器会将在类的说明部分定义的任何函数都认定为内联函数，即使它们没有inline说明。一个内联函数可以有，也可以没有return语句。内联函数在程序执行时并不产生实际函数调用，而是在函数调用处将函数代码展开执行。内联函数是通过编译器来实现的。
3. 在类内部实现的函数都是内联函数，可不用关键字inline定义； 只有函数外部定义的内联函数必须使用关键字inline。

   下面案例中成员函数GetLength()和GetWidth()是内联函数。

   ```
   class Box{
   public:
          double GetLength(){ return length; }//<------
          double GetWidth();
          double GetHeight();
   private:
          double length, width, height;
   };
   inline double Box::GetWidth() { return width; }//<------
   double Box::GetHeight(){ return height; }
   在成员函数GetLenth、GetWidth和GetHeight中，内联函数有___2个__。
   ```

   在一个函数中，要求通过函数来实现一种不太复杂的功能，并且要求加快执行速度，选用 **内联函数。**

Char相关知识

Char数组的声明

静态方式

```
char*p = "Happy";// 其实它存的是H a p p y \0 这样
char a[]="Happy"; 等同于 char a[6] = {'H', 'a', 'p', 'p', 'y', '\0'};
//char a[5] = "Happy";//错误 数组越界 最后应有'\0'
char a[5] = {'H','a','p','p','y'};等同于 char a[] = { 'H','a','p','p','y' };
```

编译器带来的问题

```
//在devc++中可以
char* ch1 = "Hello world";

//在VS中不能用上面的方法 但可以用以下的方式
//1、先用另外的字符数组存储Hello world，再对字符型指针进行初始化
char ch2[] = "Hello world";
char* text = ch2;
//2、将char类型强转为char*
char* text = (char*)"Hello world";
```

动态方式

```
char*p = new char[n];//只能存n-1个字符，C++默认最后一个是字符串结束空字符。

char*target = new char[strlen(传入的字符串)+1];
或 char*target = new char[sizeof(传入的字符串)];

char*p = new char[6];
for(int i=0;i<6;i++)
{
  *(p+i) = 'A' + i;
}
// A B C D E F
```

char数组的长度

```
char str1[] = "Myl";
cout << strlen(str1) << endl;//3
char str2[] = { 'M','y','l' };//等同于char str2[3] = { 'M','y','l' }
cout << strlen(str2) << endl;//随机数 因为不知道\0在哪里
char str3[] = { 'M','y','l' };
cout << strlen(str3) << endl;//随机数 因为不知道\0在哪里
```

strlen在使用时 在`DevC++`里需包含头文件：#include `<string.h>` 或`<cstring>` （考试注意）；而在vs不需要

string char* char[]的相互转换

一、string转char*：

主要有三种方法可以将str转换为char*类型，分别是：data(); c_str(); copy();

data()方法，如：

```
    string str = "hello";
    const char* p = str.data();//加const  或者用char * p=(char*)str.data();的形式
```

同时有一点需要说明，这里在dev c++中编译需要添加const，否则会报错invalid conversion from const char* to char ，*这里可以再前面加上const或者在等号后面给强制转化成char*的类型。

下面解释下该问题，const char*是不能直接赋值到char*的,这样编译都不能通过,理由:假如可以的话,那么 通过char*就可以修改const char指向的内容了,这是不允许的。所以char*要另外开辟新的空间，即上面的 形式。

c_str()方法，如：

```
    string str=“world”; const char *p = str.c_str();//同上，要加const或者等号右边用char*
```

copy()方法，如：

```
    string str="hmmm";  char p[50]; str.copy(p, 5, 0);//这里5代表复制几个字符，0代表复制的位置，    *(p+5)=‘\0’;//注意手动加结束符！！！
```

二、char * 转string:

可以直接赋值。

```
    string s;   char *p = "hello";//直接赋值     s = p;
```

这里有一点要说明，当声明了string类型变量s后，用printf("%s",s);是会出错的，因为“%s”要求后面的对 象的首地址。但是string不是这样的一个类型。所以肯定出错。

三、string转char[]

这个由于我们知道string的长度，可以根据length()函数得到，又可以根据下标直接访问，所以用一个循环就 可以赋值了。

```
    string pp = "dagah";    char p[8];  int i;  for( i=0;i<pp.length();i++)
 p[i] = pp[i];   p[i] = '\0';    printf("%s\n",p);   cout<<p;
```

四、char[]转string

这里可以直接赋值。

sizeof

sizeof语法

sizeof不是函数

1、用于变量名 sizeof(变量名) 或 sizeof 变量名

2、用于数据类型 sizeof(数据类型)

//除变量名的是这样的 eg： sizeof(char[20])

sizeof(void) = 1;

sizeof与strlen的区别

sizeof是运算符，用于计算变量（或数据类型）**占用内存的字节数**；
strlen是函数，用于计算字符串的**实际长度**。

```
char buf[20] = "hellow";
cout << sizeof(buf) << endl;//20
cout << strlen(buf) << endl;//6
```

```
char buf[] = "hellow";
cout << sizeof(buf) << endl;//7 多了个 \0
cout << strlen(buf) << endl;//6
```

在遍历时的区别

```
    char buf[] = "hellow";
    for(int i = 0;i < sizeof(buf);i++){
        cout << "buf["<< i <<"]= " << buf[i]<<endl;
    }
    cout << "----------------------------------"<<endl;
    for(int i = 0;i < strlen(buf);i++){
        cout << "buf["<< i <<"]= " << buf[i]<<endl;
    }
```

输出

```
buf[0]= h
buf[1]= e
buf[2]= l
buf[3]= l
buf[4]= o
buf[5]= w
buf[6]=
----------------------------------
buf[0]= h
buf[1]= e
buf[2]= l
buf[3]= l
buf[4]= o
buf[5]= w
```

可以通过 `sizeof(buf) - 1` 来达到预期效果

用引用传数组参数

在C++中，有时候我们需要将数组作为参数传递到函数里去，常规的做法是：

```
void fun(int* a) {    cout << sizeof(a); //这里会输出4或8(64位)，即a这个指针的大小
//接着对数组进行操作}
```

但是这种做法丢失了一部分信息，有没有更加符合“语法”的写法呢？答案是有的：

```
void fun2(int (&a)[100]) {
    cout << sizeof(a); //这里会输出400，即a数组的大小
    //接着对数组进行操作
}
```

当要想限定传入数组的大小时，可以用上述的写法，其意思为“对数组a的引用”。同时应当注意，这种写法可以安全的使用sizeof()函数。

但只能传入的与 参数数组大小 对应的 数组变量

```
void func3(int (&arr)[10]) {
    cout << "引用传入函数中" << "sizeof(arr) = " << sizeof(arr) << endl;
}

    int arr1[10] = { 0 };
    int arr2[2] = { 0 };
    func3(arr1);
    func3(arr2);//编译出错
```

数组当作函数参数传入时，会退化为指针

```
void func(char buf[]){
    cout << "函数中" << "sizeof(buf)= "<< sizeof(buf) <<endl;
}

void func1(char(&buf)[6]) {//参数这必须怎么写，传进来的参数也要是怎么大
    cout << "引用传入函数中" << "sizeof(buf)= " << sizeof(buf) << endl;
}

void func2(int arr[]){
    cout << "函数中" <<"sizeof(arr) = " << sizeof(arr) <<endl;
}

int main() {
    char buf[] = "abcde";
    cout << "sizeof(buf)= "<< sizeof(buf) <<endl;//sizeof(buf)= 6
    func(buf);//函数中sizeof(buf)= 4或8(64位)
    func1(buf);//引用传入函数中sizeof(buf)= 6

    int arr[] = {1,2,3};
    cout << "sizeof(arr) = " << sizeof(arr) <<endl;//sizeof(arr) = 12
    func2(arr);//函数中sizeof(arr) = 4或8(64位)
    return 0;
}
```

数组

数组初始化

数组未定义

普通数组没有赋初值，默认的数组元素值是随机数，不是0。
如果在定义数组时，数据类型前面加上关键字static，数组变成了静态数组；或者把数组定义在函数的外面，成为全局变量数组，这时数组元素的值自动赋值为0。

初始化方式

```
    int arr1[10];//未初始化为随机值
    for (int i = 0; i < 10; i++)cout << arr1[i] << " ";
    cout << endl;
    cout << "-----------------------------------"<<endl;
    int arr2[10] = {};//用默认值补全后面的值
    for (int i = 0; i < 10; i++)cout << arr2[i] << " ";
    cout << endl;
    cout << "-----------------------------------"<<endl;
    int arr3[10] = { 0 };// 给第一个值 赋值为 0 后面用默认值（0）补全
    for (int i = 0; i < 10; i++)cout << arr3[i] << " ";
    cout << endl;
    cout << "-----------------------------------"<<endl;
    int arr4[10] = { 1 };//只有第一个元素是 1 而不是全部初始化为 1
    for (int i = 0; i < 10; i++)cout << arr4[i] << " ";
    cout << endl;
    cout << "-----------------------------------"<<endl;
    int arr5[10];
    memset(arr5,-1,sizeof(arr5));//在dev中 包含头文件 #include<cstring>
    //参数(数组，初始为值，字节数)
    for (int i = 0; i < 10; i++)cout << arr5[i] << " ";
    cout << endl;
    cout << "-----------------------------------"<<endl;
```

输出：

```
0 11865184 -1842097338 20 11865184 11865204 4663176 6 0 24
-----------------------------------
0 0 0 0 0 0 0 0 0 0
-----------------------------------
0 0 0 0 0 0 0 0 0 0
-----------------------------------
1 0 0 0 0 0 0 0 0 0
-----------------------------------
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-----------------------------------
```

二维数组声明

int (*p)[2]; //是一个列数为2，行数不确定的二维数组，同 int p [] [2];

int *p[2]; //是一个行数为2，列数不确定的int指针数组，它的每一个元素表示一个int指针

数组相关位置

数组偏移

a为二维数组
1、&a+1跨整个变量。
2、a+1跨一维数组。
3、a[0]+1跨一个数据类型

```
a为二维数组//a[10][10]
*(a+0)=a[0][0]          第0行
*(a+1)=a[1][0]          第1行
*(a+2)=a[2][0]          第2行
*(a+n)=a[n][0]          第n行

*(*(a+0) + 0)=a[0][0] 第0行第0列
*(*(a+0) + 1)=a[0][1] 第0行第1列
*(*(a+1) + 0)=a[1][0] 第1行第0列
*(*(a+1) + 1)=a[1][1] 第1行第1列
...
*(*(a+n)+m)=a[n][m] 第n行第m列
```

```
    int a[3][3] = { {1,4,7},{2,5,8},{11,6,9} };
    cout << **a << endl;//1
    cout << *(*a + 2) << endl;//7
    cout << *(*a + 1) << endl;//4
    cout << **(a + 1) << endl;//2
    cout << **(a + 2) << endl;//11
    cout << *(*(a + 1)+1) << endl;//5
```

指针

指针相减

结论：指针相减 = （地址1-地址2）/sizeof(数据类型)

```
    int a[10] = {1,2,3,4,5,6,7,8,9,10};
    int* p = a;
    int* m = p++;
    cout << "p= " << p << endl;
    cout << "m= " << m << endl;
    cout << "m - p = " << m - p <<endl;
    cout << "(char*)m - (char*)p = " << (char*)m - (char*)p <<endl;
输出
p= 0x6bfeb0
m= 0x6bfeb4//地址相差 4个字节（int）
m - p = 1
(char*)m - (char*)p = 4
--------------------------------------------------------------------------------
-------
double g[] = { 1,2 };
double* buf= &g[1];
cout << "buf - g = " << buf - g << endl;
cout << "(int*)buf - (int*)g = " << (int*)buf - (int*)g << endl;
cout << "(double*)buf - (double*)g = " << (double*)buf - (double*)g << endl;
cout << "(char*)buf - (char*)g = " << (char*)buf - (char*)g << endl;
```

```
输出
buf - g = 1
(int*)buf - (int*)g = 2
(double*)buf - (double*)g = 1
(char*)buf - (char*)g = 8
```

for+遍历

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
//1、
for(int i=0;i<10;i++)
    cout << a[i] <<" ";
//2、
for(int* p=a;p<a+10;p++)
    cout << *p <<" ";
cout <<endl;
输出 1 2 3 4 5 6 7 8 9 10
```

深度理解数组和指针的关系

```
//int变量类型的指针
int num = 1;
int* m = &num;

//int数组的指针
int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
int* p = a;//p的值为a的数组首地址
cout << a << endl;
cout << p << endl;
cout << &a << endl;//前3个都是一样的地址
cout << &p << endl;//输出的是 p指针 的地址 （不和前面的一样）

//cout << ++a << endl;数组是常量不可以改变
cout << a + 1 << endl;//但可以这样 不会改变a

cout << ++p << endl;//而指针可以改变 ++指向下一个索引的 地址
cout << &a[1] << endl;

p = a;//初始化
cout << *(p + 5) << endl; //输出5

p = a;//初始化
cout << a[3] << endl;
cout << p[3] << endl;//p可以像数组一样用[]
输出
    3 3

p++//让p偏移
cout << a[3] << endl;
cout << p[3] << endl;
输出
    3 4
```

指针常量与常量指针

1.指针常量与常量指针的概念

指针常量就是指针本身是常量，换句话说，就是指针里面所存储的内容（内存地址）是常量，不能改变。但是，内存地址所对应的内容是可以通过指针改变的。

常量指针就是指向常量的指针，换句话说，就是指针指向的是常量，它指向的内容不能发生改变，不能通过指针来修改它指向的内容。但是，指针自身不是常量，它自身的值可以改变，从而指向另一个常量。

2.指针常量与常量指针的声明

指针常量的声明：数据类型 * const 指针变量。

常量指针的声明： 数据类型 const * 指针变量或者 const 数据类型 * 指针变量。

常量指针常量的声明：数据类型 const * const 指针变量 或者 const 数据类型 * const 指针变量。

函数

函数重载条件

作用：函数名可以相同，提高复用性

函数重载满足条件：

- 同一个作用域下
- 函数名称相同
- 函数参数**类型不同** 或者 **个数不同** 或者 **顺序不同**

注意: 函数的返回值不可以作为函数重载的条件

在对函数进行重载时，不允许为其指定缺省参数

已知程序中已经定义了函数test，其原型是int test(int,int,int);，则下列重载形式中正确的是_____。

```
    char test(int,int,int);//错
    double test(int,int,double);//对
    int test(int,int,int=0);//错
    float test(int,int,float=3.5F);//错
```

函数指针

```
void func(int a){
    cout<<a<<endl;
}
int main(){
    void (*p)(int a);
    p = func;
    p(123);
    //输出123

    //可以定义一个像 类 一样的 函数指针
```

```
    typedef void (*P)(int a);
    P f = func;//初始化f
    f(333);
    P s =func;
    s(32331);
}
```

类

类的六个默认成员函数

构造函数、拷贝构造函数、析构函数、赋值操作符重载、取地址操作符重载、const修饰的取地址操作符重载

构造函数、析构函数

如果**我们不提供构造和析构，编译器会提供编译器提供的构造函数和析构函数是空实现。**

- 构造函数：主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用。
- 析构函数：主要作用在于对象**销毁前**系统自动调用，执行一些清理工作。

  不可以手动调用构造函数，但是可以手动调用析构

  一个类只能定义一个析构函数

注意1：调用无参构造函数不能加括号，如果加了编译器认为这是一个函数声明 `eg:Person p2()`

```
int main() {
    A fun();//声明
    fun();//调用
}

A fun() {//定义
    cout << "返回值为A类的函数" << endl;
    A a;
    return a;
}
```

```
class A{
public:
    A(string m):m_m(m){
        cout<< m_m <<"的构造函数的调用"<<endl;
    }
    A(const A& p) {
        m_m = p.m_m + "的副本";
        cout<< m_m <<"的拷贝构造函数的调用"<<endl;
    }
    ~A(){
        cout<< m_m <<"的析构函数的调用"<<endl;
    }
    string m_m;
};
```

```
int main()
{
    A("a0");
    /*a0的构造函数的调用
    a0的析构函数的调用*/
    ------------------------------------------------------------------------
    A a1("a1");
    A a2 = a1;
    /*a1的构造函数的调用
    a1的副本的拷贝构造函数的调用

    a1的副本的析构函数的调用
    a1的析构函数的调用*/
    ------------------------------------------------------------------------
    A a3("a3");
    A a4(a3);
    /*a3的构造函数的调用
    a3的副本的拷贝构造函数的调用

    a3的副本的析构函数的调用
    a3的析构函数的调用*/
}
```

初始化列表方式初始化

```
 Person(int a, int b, int c) :m_A(a), m_B(b), m_C(c) {}
```

深拷贝与浅拷贝

浅拷贝：简单的赋值拷贝操作

深拷贝：在堆区重新申请空间，进行拷贝操作

```
//拷贝构造函数
    Person(const Person& p) {
        cout << "拷贝构造函数!" << endl;
        //如果不利用深拷贝在堆区创建新内存，会导致浅拷贝带来的重复释放堆区问题
        m_age = p.m_age;
        m_height = new int(*p.m_height);
    }//类属性有 指针时需深拷贝
```

调用拷贝构造的情况

用一个对象去初始化同一类的另一个新对象时

函数的返回值是类的对象，函数执行返回调用时

函数的形参是类的对象，调用函数进行形参和实参结合时

静态成员

静态成员就是在成员变量和成员函数前加上关键字static，称为静态成员

静态成员分为：

- 静态成员变量
  - 所有对象共享同一份数据
  - 在编译阶段分配内存
  - 类内声明，类外初始化
- 静态成员函数
  - 所有对象共享同一个函数
  - 静态成员函数**只能访问静态成员变量**

```
class Person
{
public:
    static int m_A; //静态成员变量
private:
    static int m_B; //静态成员变量也是有访问权限的
};
int Person::m_A = 10;
int Person::m_B = 10;
```

静态成员变量两种访问方式

1、通过对象 p1.m_A

2、通过类名 Person::m_A

//静态成员也是有访问权限的

对象模型和this指针

每个成员函数都有一个指针形参，它的名字是固定的，叫做this指针。this指针是隐式的，并且它是成员函数的第一个参数。要注意的是构造函数、友元函数比较特殊，它没有this指针；

this指针的用途：

- 当形参和成员变量同名时，可用this指针来区分
- 在类的非静态成员函数中返回对象本身，可使用return *this

```
class Person
{
public:
    Person(int age)
    {
        //1、当形参和成员变量同名时，可用this指针来区分
        this->age = age;
    }
    Person& PersonAddPerson(Person p)
    {
        this->age += p.age;
        //返回对象本身
        return *this;
    }
    int age;
```

```
};
```

空指针访问成员函数

C++中空指针也是可以调用成员函数的，但是也要注意有没有用到this指针

如果用到this指针，需要加以判断保证代码的健壮性

```
class Person {
public:
    void ShowClassName() {
        cout << "我是Person类!" << endl;
    }
    void ShowPerson() {
        if (this == NULL) {//判断是否为NULL，可以来解决报错的问题
            return;
        }
        cout << mAge << endl;
    }
public:
    int mAge;
};
void test01()
{
    Person* p = NULL;
    p->ShowClassName(); //空指针，可以调用成员函数
    p->ShowPerson();  //但是如果成员函数中用到了this指针，就会报错 访问权限冲突
}
```

const修饰成员函数

常函数：

- 成员函数后加const后我们称为这个函数为**常函数**
- 常函数内不可以修改成员属性
- 成员属性声明时加关键字mutable后，在常函数中依然可以修改

常对象：

- 声明对象前加const称该对象为常对象
- 常对象只能调用常函数

```
class Person {
public:
    Person() {
        m_A = 0;
        m_B = 0;
    }
    //this指针的本质是一个指针常量，指针的指向不可修改
    //如果想让指针指向的值也不可以修改，需要声明常函数
    void ShowPerson() const {
        //const Type* const pointer;
        //this = NULL; //不能修改指针的指向 Person* const this;
        //this->mA = 100; //但是this指针指向的对象的数据是可以修改的

        //const修饰成员函数，表示指针指向的内存空间的数据不能修改，除了mutable修饰的变量
        this->m_B = 100;
```

```
    }
    void MyFunc() const {
        //mA = 10000;
    }
public:
    int m_A;
    mutable int m_B; //可修改 可变的
};
//const修饰对象  常对象
void test01() {
    const Person person; //常量对象
    cout << person.m_A << endl;
    //person.mA = 100; //常对象不能修改成员变量的值,但是可以访问
    person.m_B = 100; //但是常对象可以修改mutable修饰成员变量

    //常对象访问成员函数
    person.MyFunc(); //常对象只能调用const的函数
}
```

友元

友元的目的就是让一个函数或者类 访问另一个类中私有成员

友元的关键字为 friend

友元的声明可以放在类的任何一个区域(公有、保护、私有)。但是友元的实现不可以在类内部。

友元函数中不能使用this指针来访问类成员变量。

友元的三种实现

- 全局函数做友元
- 类做友元
- 成员函数做友元

```
class Building {
    //告诉编译器 goodGay全局函数 是 Building类的好朋友，可以访问类中的私有内容
    friend void goodGay(Building * building);
}
```

```
class Building
{
    //告诉编译器 goodGay类是Building类的好朋友，可以访问到Building类中私有内容
    friend class goodGay;
}
```

```
class Building
{
    //告诉编译器  goodGay类中的visit成员函数 是Building好朋友，可以访问私有内容
    friend void goodGay::visit();
}
```

运算符重载

运算符重载概念：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型

运算符重载是多态性的一种表现

对于运算符的重载，通常有两种形式。
1、操作结果 operator 运算符(操作数1,[操作数2]) **两个操作数必须至少有一个自定义类**
2、操作结果 类::operator 运算符(操作数)

1.不能改变运算符的优先级。
2.不能改变运算符的结合性。
3.默认参数不能和重载的运算符一起使用，也就是说，在设计运算符重载成员函数时不能使用默认函数。
4.不能改变运算符的操作数的个数。
5.不能创建新的运算符，只有已有运算符可以被重载
6.运算符作用于C++内部提供的数据类型时，原来含义保持不变

限制 ".", ".*", "->*", "::", "?:", "sizeof"等操作符的重载。(带*.的都不行)

必须在作为类成员函数重载的运算符：“=”、“[ ]”、“ () ”、“->”

输入输出运算符不能重载为类的成员函数

```
ostream& operator<<(ostream&,const 类对象引用)
istream& operator>>(istream&,类对象的引用)
```

重载加法运算符

```
//成员函数实现 + 号运算符重载
//类中含有两个成员变量 int m_A;int m_B;
    Person operator+(const Person& p) {
        Person temp;
        temp.m_A = this->m_A + p.m_A;
        temp.m_B = this->m_B + p.m_B;
        return temp;
    }

//全局函数实现 + 号运算符重载
    Person operator+(const Person& p1, const Person& p2) {
        Person temp(0, 0);
        temp.m_A = p1.m_A + p2.m_A;
        temp.m_B = p1.m_B + p2.m_B;
        return temp;
    }

//运算符重载 可以发生函数重载
    Person operator+(const Person& p2, int val)
    {
        Person temp;
        temp.m_A = p2.m_A + val;
        temp.m_B = p2.m_B + val;
        return temp;
    }

//调用
    Person p1(10, 10);
    Person p2(20, 20);

    //成员函数方式
    Person p3 = p2 + p1;  //相当于 p2.operaor+(p1)
```

重载左移运算符

```
class Person {
    friend ostream& operator<<(ostream& out, Person& p);
public:
    Person(int a, int b)
    {
        this->m_A = a;
        this->m_B = b;
    }
    //成员函数 实现不了  p << cout 不是我们想要的效果
    //void operator<<(Person& p){
    //}
private:
    int m_A;
    int m_B;
};
//全局函数实现左移重载
//ostream对象只能有一个
ostream& operator<<(ostream& out, Person& p) {
    out << "a:" << p.m_A << " b:" << p.m_B;
    return out;
}

void test() {
    Person p1(10, 20);
    cout << p1 << "hello world" << endl; //链式编程
}
```

总结：重载左移运算符配合友元可以实现输出自定义数据类型

重载递增运算符

```
class MyInteger {
public:
    MyInteger() {
        m_Num = 0;
    }
    //前置++
    MyInteger& operator++() {
        //先++
        m_Num++;
        //再返回
        return *this;
    }

    //后置++
    MyInteger operator++(int) {
        //先返回
        MyInteger temp = *this; //记录当前本身的值，然后让本身的值加1，但是返回的是以前的值，达到先返回后++；
        m_Num++;
        return temp;
    }
```

```
private:
    int m_Num;
};
```

总结： 前置递增返回引用，后置递增返回值

重载赋值运算符

```
class Person
{
public:
    Person(int age)
    {
        //将年龄数据开辟到堆区
        m_Age = new int(age);
    }
    //重载赋值运算符
    Person& operator=(Person &p)
    {
        if (m_Age != NULL)
        {
            delete m_Age;
            m_Age = NULL;
        }
        //编译器提供的代码是浅拷贝
        //m_Age = p.m_Age;

        //提供深拷贝 解决浅拷贝的问题
        m_Age = new int(*p.m_Age);

        //返回自身
        return *this;
    }
    ~Person()
    {
        if (m_Age != NULL)
        {
            delete m_Age;
            m_Age = NULL;
        }
    }
    //年龄的指针
    int *m_Age;
};
```

赋值运算符只能通过成员函数的方式进行重载

重载关系运算符

```
class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
```

```
    };

    bool operator==(Person & p)
    {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    bool operator!=(Person & p)
    {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
         {
            return false;
        }
        else
        {
            return true;
        }
    }

    string m_Name;
    int m_Age;
};
```

重载函数调用运算符

- 函数调用运算符 () 也可以重载
- 由于重载后使用的方式非常像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

```
class MyPrint
{
public:
    void operator()(string text)
    {
        cout << text << endl;
    }
};
void test01()
{
    //重载的（）操作符 也称为仿函数
    MyPrint myFunc;
    myFunc("hello world");
}


class MyAdd
{
public:
    int operator()(int v1, int v2)
    {
```

```
        return v1 + v2;
    }
};

void test02()
{
    MyAdd add;
    int ret = add(10, 10);
    cout << "ret = " << ret << endl;

    //匿名对象调用
    cout << "MyAdd()(100,100) = " << MyAdd()(100, 100) << endl;
}
```

继承

继承的好处：可以减少重复的代码

class A : public B;

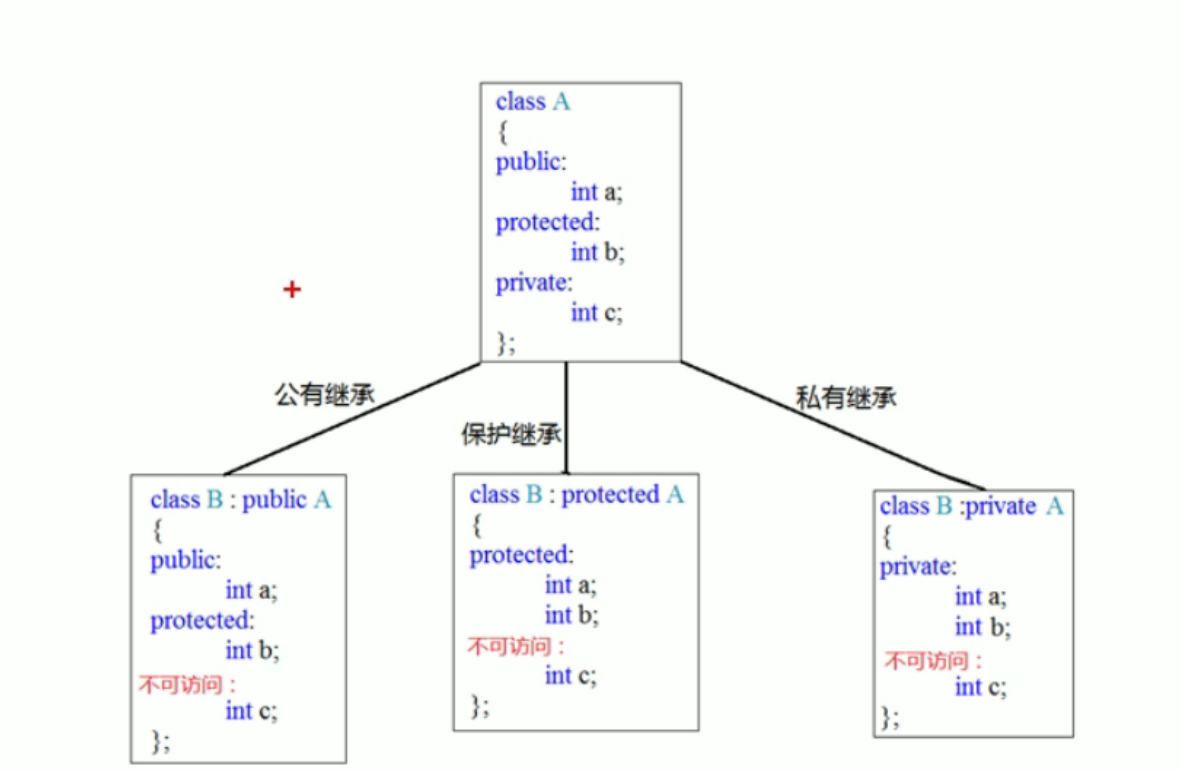
A 类称为子类 或 派生类

B 类称为父类 或 基类

派生类中的成员，包含两大部分：

一类是从基类继承过来的，一类是自己增加的成员。

从基类继承过过来的表现其共性，而新增的成员体现了其个性。



结论： 父类中私有成员也是被子类继承下去了，只是由编译器给隐藏后访问不到

如果不显式地给出继承方式，缺省的继承方式是私有继承。

```
class A {
public:
    int a;
};
class B : A{

};
```

继承中构造和析构顺序

建立派生类对象时，构造函数的执行顺序是，执行**虚拟基类**的构造函数，再是**非虚拟基类**然后执行**派生类成员对象**的构造函数，执行**派生类的构造函数**。最后是**非类对象成员**。

```
class Base
{
public:
    Base()
    {
        cout << "Base构造函数!" << endl;
    }
    ~Base()
    {
        cout << "Base析构函数!" << endl;
    }
};

class Son : public Base
{
public:
    Son()
    {
        cout << "Son构造函数!" << endl;
    }
    ~Son()
    {
        cout << "Son析构函数!" << endl;
    }
};
int main() {
    Son s;
}
输出：
    Base构造函数!
    Son构造函数!
    Son析构函数!
    Base析构函数!
```

继承同名成员处理方式

问题：当子类与父类出现同名的成员，如何通过子类对象，访问到子类或父类中同名的数据呢？

- 访问子类同名成员 直接访问即可
- 访问父类同名成员 需要加作用域

- 同名静态成员处理方式和非静态处理方式一样，只不过有两种访问的方式（通过对象 和 通过类名）

继承中析构函数的奇妙特点

```
class Base {
public:
    Base() {
        cout << "调用Base的构造函数" << endl;
    }
    ~Base()
    {
        cout << "调用Base的析构函数" << endl;
    }
};
class Son :public Base{
public:
    Son() {
        cout << "调用Son的构造函数" << endl;
    }
    ~Son()
    {
        cout << "调用Son的析构函数" << endl;
    }
};
class Grandson :public Son{
public:
    Grandson() {
        cout << "调用Grandson的构造函数" << endl;
    }
    ~Grandson()
    {
        cout << "调用Grandson的析构函数" << endl;
    }
};

int main() {
    Grandson s;
    /*
    调用Base的构造函数
    调用Son的构造函数
    调用Grandson的构造函数
    调用Grandson的析构函数
    调用Son的析构函数
    调用Base的析构函数
    */
    s.~Grandson();
    /*
    调用Grandson的析构函数
    调用Son的析构函数
    调用Base的析构函数
    */
    return 0;
}
```

总结：调用类对象自己析构函数之后会自动调用 父类的 析构函数

继承中遇到的构造函数问题

总结：

子类在构造时，如果没有显式调用父类的构造函数，会先调用父类的默认构造函数（无参数的）

```
#include <iostream>
class Base{
public:
    Base(int a){}
};
class Derive:public Base{
public:
    Derive(int a,int b):Base(a){}//除了 父类有 无参构造函数时，不用这种写法，其他情况都需要有 父类初始化
};

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

虚继承

在采用虚基类时，其成员将仅存唯一的副本，这样就可以解决二义性问题，但不能够实现运行时多态，可通过抽象类或虚拟函数来实现，虚基类的构造函数将会首先被执行。

多态

多态的基本概念

多态是C++面向对象三大特性之一

多态分为两类

- 静态多态: 函数重载 和 运算符重载属于静态多态，复用函数名
- 动态多态: 派生类和虚函数实现运行时多态

静态多态和动态多态区别：

- 静态多态的函数地址早绑定 - 编译阶段确定函数地址
- 动态多态的函数

地址晚绑定 - 运行阶段确定函数地址

虚函数

虚函数是一种单界面多实现版本的实现方法，即函数名、返回类型、函数类型和个数顺序完全相同，但函数体内容可以完全不同。

基类中采用virtual说明一个虚函数后，派生类中定义相同原型函数时可不必加virtual说明

虚函数不允许说明成静态的成员函数

纯虚函数和抽象类

在多态中，通常父类中虚函数的实现是毫无意义的，主要都是调用子类**重写**的内容

因此可以将虚函数改为**纯虚函数**

纯虚函数语法：`virtual 返回值类型 函数名 （参数列表）= 0 ;`

当类中有了纯虚函数，这个类也称为抽象类

抽象类特点：

- 无法实例化对象
- 子类必须重写抽象类中的纯虚函数，否则也属于抽象类

```
class Base
{
public:
    virtual void func() = 0;
};

class Son :public Base
{
public:
    virtual void func()
    {
        cout << "func调用" << endl;
    };
};

void test01()
{
    Base * base = NULL;
    //base = new Base; // 错误，抽象类无法实例化对象
    base = new Son;
    base->func();
    delete base;//记得销毁
}
```

虚析构和纯虚析构

多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码

解决方式：将父类中的析构函数改为**虚析构**或者**纯虚析构**

虚析构和纯虚析构共性：

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

虚析构和纯虚析构区别：

- 如果是纯虚析构，该类属于抽象类，无法实例化对象

虚析构语法：

```
virtual ~类名(){}
```

纯虚析构语法：

```
virtual ~类名() = 0;
类名::~类名(){}
```

```
class Animal {
public:
    Animal()
    {
        cout << "Animal 构造函数调用！" << endl;
    }
    virtual void Speak() = 0;

    //析构函数加上virtual关键字，变成虚析构函数
    //virtual ~Animal()
    //{
    //  cout << "Animal虚析构函数调用！" << endl;
    //}
    virtual ~Animal() = 0;
};
Animal::~Animal()
{
    cout << "Animal 纯虚析构函数调用！" << endl;
}

//和包含普通纯虚函数的类一样，包含了纯虚析构函数的类也是一个抽象类。不能够被实例化。

class Cat : public Animal {
public:
    Cat(string name)
    {
        cout << "Cat构造函数调用！" << endl;
        m_Name = new string(name);
    }
    virtual void Speak()
    {
        cout << *m_Name <<  "小猫在说话!" << endl;
    }
    ~Cat()
    {
        cout << "Cat析构函数调用!" << endl;
        if (this->m_Name != NULL) {
            delete m_Name;
            m_Name = NULL;
        }
    }
public:
    string *m_Name;
};

void test01()
{
    Animal *animal = new Cat("Tom");
    animal->Speak();

    //通过父类指针去释放，会导致子类对象可能清理不干净，造成内存泄漏
```

```
    //怎么解决？给基类增加一个虚析构函数
    //虚析构函数就是用来解决通过父类指针释放子类对象
    delete animal;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

总结：

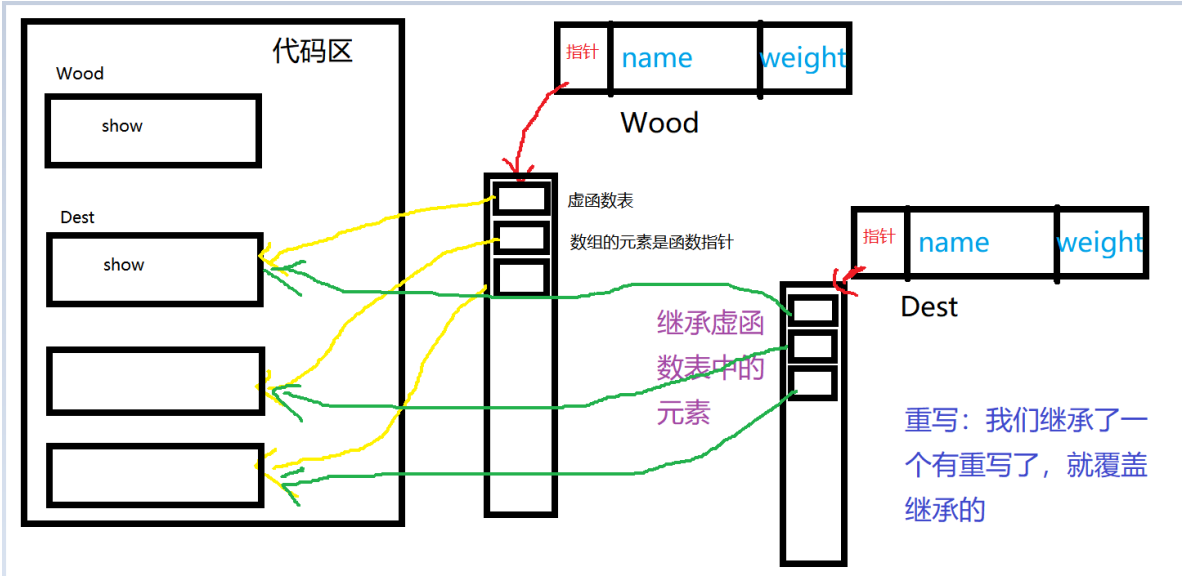
1. 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象
2. 如果子类中没有堆区数据，可以不写为虚析构或纯虚析构
3. 拥有纯虚析构函数的类也属于抽象类

多态案例

```
#include <iostream>
using namespace std;
#include<string>
class Wood {
protected:
    string name;
    int weight;
public:
    Wood(string n ,int w):name(n),weight(w){}
    virtual void Show() {//虚函数
        cout << "这是一块重" << weight << "kg的" << name << endl;
    }
};
//凳子
class Stool :public Wood{
public:
    Stool(string n , int w):Wood(n,w){}
    void Show() {//虚函数
        cout << "这是用一块重" << weight << "kg的" << name << "做的凳子" << endl;
    }
};
//桌子
class Dest :public Wood {
public:
    Dest(string n, int w) :Wood(n, w) {}
    void Show() {//虚函数
        cout << "这是用一块重" << weight << "kg的" << name << "做的桌子" << endl;
    }
};

void test(Wood* p) {
    p->Show();
}

int main() {
```

```
    Wood w1("橡木", 5), w2("楠木", 6);
    Stool s1("橡木", 5), s2("楠木", 6);
    Dest d1("橡木", 5), d2("楠木", 6);

    w1.Show();
    w2.Show();
    cout << "------------------------------" << endl;
    s1.Show();
    s2.Show();
    cout << "------------------------------" << endl;
    d1.Show();
    d2.Show();

    /*这是一块重5kg的橡木
    这是一块重6kg的楠木
    ------------------------------
    这是用一块重5kg的橡木做的凳子
    这是用一块重6kg的楠木做的凳子
    ------------------------------
    这是用一块重5kg的橡木做的桌子
    这是用一块重6kg的楠木做的桌子*/

    cout << "------------------------------" << endl;
    Wood* p;//多肽的体现 同样的指针有不同的结果
    p = &w1;
    test(p);
    cout << "------------------------------" << endl;
    p = &s1;
    test(p);
    cout << "------------------------------" << endl;
    p = &d1;
    test(p);

    /*这是一块重5kg的橡木
    ------------------------------
    这是用一块重5kg的橡木做的凳子
    ------------------------------
    这是用一块重5kg的橡木做的桌子*/

    /*如果没加virtual 则全为“这是一块重5kg的橡木”*/

    //引用也可以
    Dest d1("楠木", 10);
    Wood* W1;
    W1 = &d1;
    W1->Show();

    return 0;
}
```

多肽的实现靠的是虚函数表



虚函数表是链表

重载、重写、重定义

函数重载（overload）
函数重载是指在一个类中声明多个名称相同但参数列表不同的函数，这些的参数可能个数或顺序，类型不同，但是不能靠返回类型来判断。特征是：
（1）相同的范围（在同一个作用域中）；
（2）函数名字相同；
（3）参数不同；
（4）virtual 关键字可有可无（注：函数重载与有无virtual修饰无关）；
（5）返回值可以不同；

函数重写（也称为覆盖 override）
函数重写是指子类重新定义基类的虚函数。特征是：
（1）不在同一个作用域（分别位于派生类与基类）；
（2）函数名字相同；
（3）参数相同；
（4）基类函数必须有 virtual 关键字，不能有 static 。
（5）返回值相同，否则报错；
（6）重写函数的访问修饰符可以不同；

重定义（也称隐藏）
（1）不在同一个作用域（分别位于派生类与基类）；
（2）函数名字相同；
（3）返回值可以不同；
（4）参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载以及覆盖混淆）；
（5）参数相同，但是基类函数没有 virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）；

其他

```
class Name{
   char name[20];
public:
   Name(){
      strcpy(name,"");
   }
   Name(char *fname){
      strcpy(name,fname);
   }
};
```

类中有char*案例

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

class String{
private:
    char* buf;
public:
    String(char* sorce){
        buf = new char[strlen(sorce)+1];
        strcpy(buf,sorce);
        buf[strlen(sorce)+1]='\0';
    }
    bool operator==(String& s){
        cout<<"this:"<< this->buf <<endl;
        cout<<"s:"<< s.buf <<endl;
        cout<<"this的地址: "<< &(this->buf)<<endl;
        cout<<"s的地址: "<< &(s.buf) <<endl;
        cout<<"类里的bool值:"<<(this->buf==s.buf)<<endl;
        if(strcmp(this->buf,s.buf)==0) return true;
        return false;
    }
    ~String(){
        delete []buf;
        buf =NULL;
    }
};
int main(){
    char* a="abc";
    char* b="abc";
    cout<< (a==b) <<endl;
    cout<<"a的地址: "<< &a<<endl;
    cout<<"b的地址: "<< &b<<endl;
    String s1(a),s2(b);
    if(s1==s2)cout<<"true"<<endl;
    else cout<<"false"<<endl;
    return 0;
}
输出:
1
a的地址: 0x6bfeec
b的地址: 0x6bfee8
this:abc
```

```
s:abc
this的地址: 0x6bfee4
s的地址: 0x6bfee0
类里的bool值:0
true
```

C++类所占内存大小计算

`class A {};`

`sizeof( A ) = ?`

`sizeof( A ) = 1`

`明明是空类，为什么编译器说它是1呢？`

1、空类同样可以实例化，每个实例在内存中都有一个独一无二的地址，为了达到这个目的，编译器往往会给一个空类隐含的加一个字节，这样空类在实例化后在内存得到了独一无二的地址．所以sizeof(A)的大小为1．

```
class B
{
public:
 B() {}
 ~B() {}
 void MemberFuncTest( int para ) { }
 static void StaticMemFuncTest( int para ){  }
};
```

`sizeof( B ) = ?`

`sizeof( B ) = 1`

2、类的非虚成员函数是不计算在内的,不管它是否静态。

```
class C
{
 C(){}
 virtual ~C() {}
};
```

`sizeof( C ) = ?`

`sizeof( C ) = 4`

3、类C有一个虚函数，存在虚函数的类都有一个一维的虚函数表叫虚表，虚表里存放的就是虚函数的地址了，因此，虚表是属于类的。这样的类对象的前四个字节是一个指向虚表的指针，类内部必须得保存这个虚表的起始指针。在32位的系统分配给虚表指针的大小为4个字节，所以最后得到类C的大小为4．

```
class D
{
 D(){}
 virtual ~D() {}
 virtual int VirtualMemFuncTest1()=0;
 virtual int VirtualMemFuncTest2()=0;
 virtual int VirtualMemFuncTest3()=0;
};
```

`sizeof( D ) = ?`

`sizeof( D ) = 4`

4、原理同类C，不管类里面有多少个虚函数，类内部只要保存虚表的起始地址即可，虚函数地址都可以通过偏移等算法获得。

```
class E
{
int m_Int;
char m_Char;
};
sizeof( E ) = ?
sizeof( E ) = 8
```

5、32位的操作系统int占4个字节，char占一个字节，加上内存对齐的3字节，为8字节。

```
class F : public E
{
static int s_data ;
};
int F::s_data=100;
sizeof( F ) = ?
sizeof( F ) = 8
```

6、类F为什么跟类E一样大呢？类F的静态数据成员被编译器放在程序的一个global data members中，它是类的一个数据成员,但是它不影响类的大小,不管这个类实际产生了多少实例还是派生了多少新的类，静态成员数据在类中永远只有一个实体存在，而类的非静态数据成员只有被实例化的时候，他们才存在．但是类的静态数据成员一旦被声明，无论类是否被实例化，它都已存在．可以这么说，类的静态数据成员是一种特殊的全局变量.

```
class G : public E
{
virtual int VirtualMemFuncTest1(int para)=0;
int m_Int;
};
class H : public G
{
int m_Int;
};
sizeof( G ) = ?
sizeof( H ) = ?
sizeof( G ) = 16
sizeof( H ) = 20
```

7、可以看出子类的大小是本身成员的大小再加上父类成员的大小.如果父类还有父类，也加上父类的父类，这样一直递归下去。

```
class I : public D
{
virtual int VirtualMemFuncTest1()=0;
virtual int VirtualMemFuncTest2()=0;
};
sizeof( I ) = ?
sizeof( I ) = 4
```

8、父类子类工享一个虚函数指针，虚函数指针保留一个即可。

总结：

空的类也是会占用内存空间的，而且大小是1，原因是C++要求每个实例在内存中都有独一无二的地址。

（一）类内部的成员变量：

普通的变量：是要占用内存的，但是要注意内存对齐（这点和struct类型很相似）。

static修饰的静态变量：不占用内存，原因是编译器将其放在全局变量区。

从父类继承的变量：计算进子类中

（二）类内部的成员函数：

非虚函数(构造函数、静态函数、成员函数等)：不占用内存。

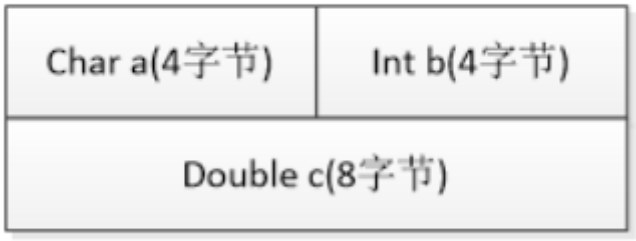
虚函数：要占用4个字节(32位的操作系统)，用来指定虚拟函数表的入口地址。跟虚函数的个数没有关系。父类子类工享一个虚函数指针。

内存对齐进阶

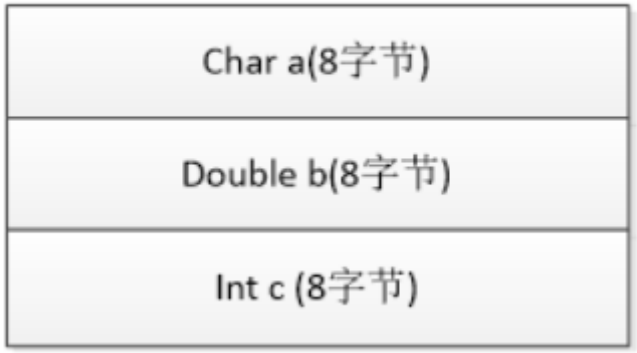
```
class base1
{
private:
    char a;
    int b;
    double c;
};//16

class base2
{
private:
    char a;
    double b;
    int c;
};//24
```

虽然上述两个类成员变量都是一个char，一个int，一个double，但是不同的声明顺序，会导致不同的内存构造模型，对于base1，base2，其成员排列是酱紫的：

base1：



base2：

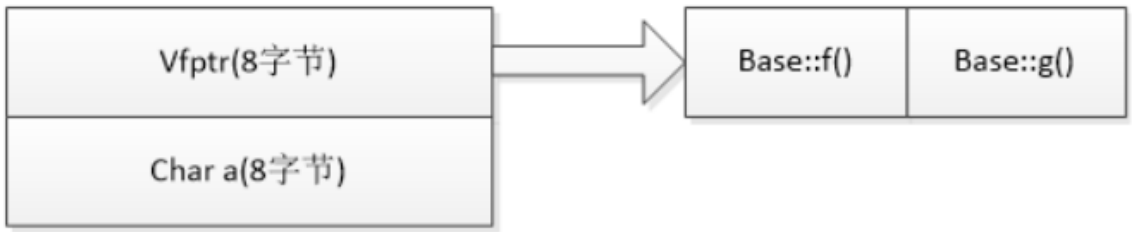


base 1类对象的大小为16字节，而base 2类对象的大小为24字节，因为不同的声明顺序，居然造成了8字节的空间差距，因此，我们将来在自己声明类时，一定要注意到内存对齐问题，优化类的对象空间分布

含虚函数的单一继承（64位）

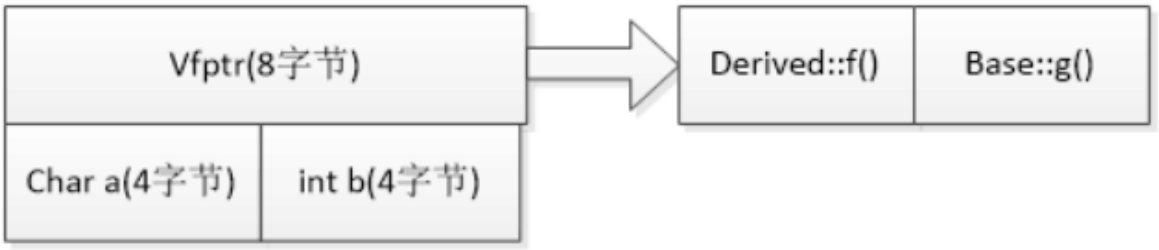
```
class Base{private:    char a;public:    virtual void f();    virtual void
g();};//16class Derived:public Base{private:    int b;public:    void
f();};class Derived1:public Base{private:    double b;public:    void g();
virtual void h();};
```

基类Base中含有一个char型成员变量，以及两个虚函数，此时Base类的内存布局如下：



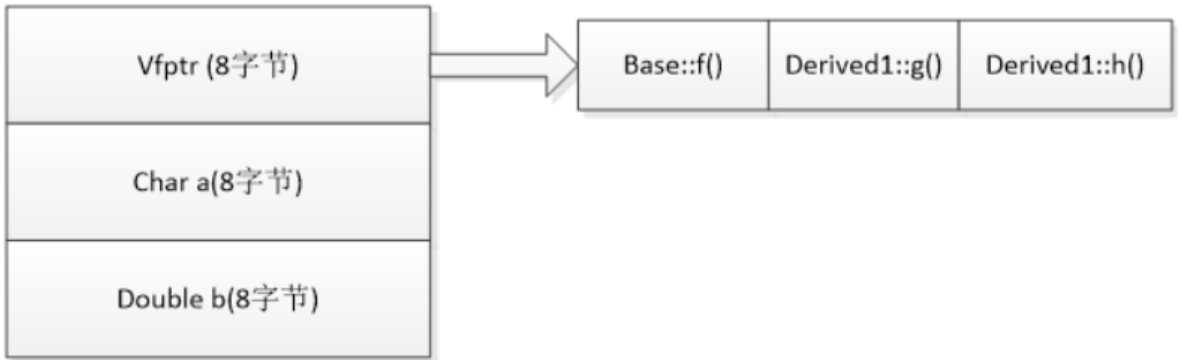
内存布局的最一开始是vfptr（virtual function ptr）即虚函数表指针（只要含虚函数，一定有虚函数表指针，而且该指针一定位于类内存模型最前端），接下来是Base类的成员变量，按照在类里的声明顺序排列，当然啦，还是要像上面一样注意内存对齐原则！

继承类Derived继承了基类，重写了Base中的虚函数f()，还添加了自己的成员变量，即int型的b，这时，Derived的类内存模型如下：



此种情况下，最一开始的还是虚函数表指针，只不过，在Derived类中被重写的虚函数f()在对应的虚函数表项的Base::f()已经被替换为Derived::f()，接下来是基类的成员变量char a，紧接着是继承类的成员变量int b，按照其基类变量声明顺序与继承类变量声明顺序进行排列，并注意内存对齐问题。

继承类Derived1继承了基类，重写了Base中的虚函数g()，还添加了自己的成员变量（即double型的b）与自己的虚函数（virtual h() ），这时，Derived1的类内存模型如下：



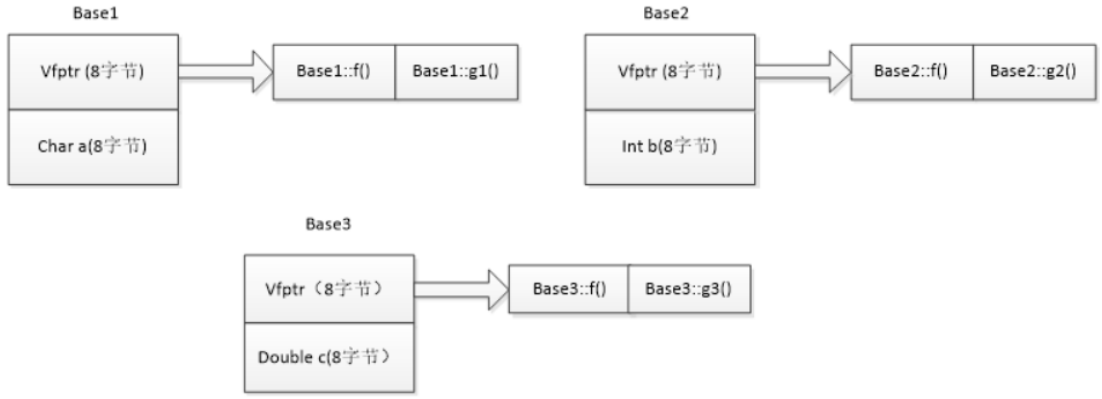
此种情况下，Derived1类一开始仍然是虚函数表指针，只是在Derived1类中被重写的虚函数g()在对应的虚函数表项的Base::g()已经被替换为Derived1::g()，新添加的虚函数virtual h()位于虚函数表项的后面，紧跟着基类中最后声明的虚函数表项后，接下来仍然是基类的成员变量，紧接着是继承类的成员变量。

含虚函数的多重继承 （64位）

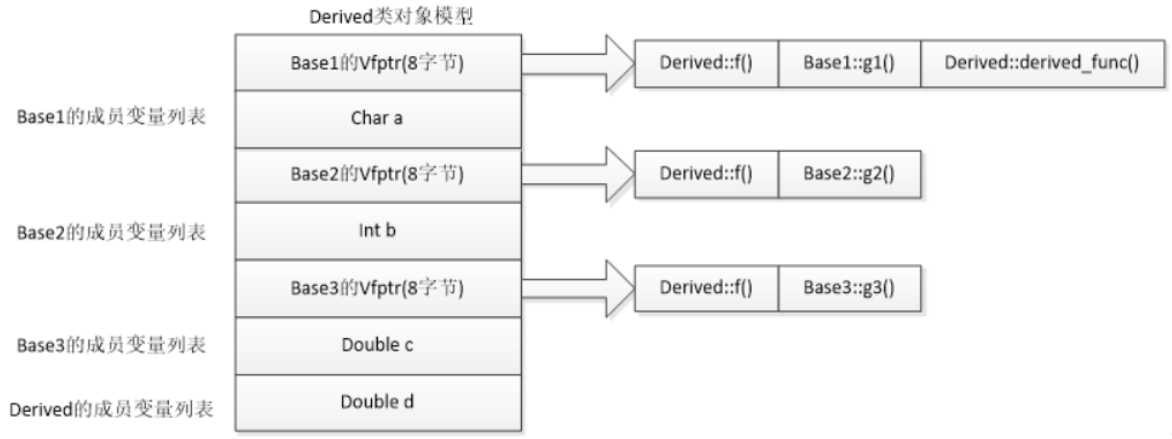
```
class Base1
{
private:
    char a;
public:
    virtual void f();
    virtual void g1();
};
class Base2
{
private:
    int b;
public:
    virtual void f();
    virtual void g2();
};
class Base3
{
private:
    double c;
```

```
public:
    virtual void f();
    virtual void g3();
};
class Derived:public Base1, public Base2, public Base3
{
private:
    double d;
public:
    void f();
    virtual void derived_func();
};//56
```

首先继承类多重继承了三个基类，此外继承类重写了三个基类中都有的虚函数virtual f()，还添加了自己特有的虚函数derived_func()，那么，新的继承类内存布局究竟是什么样子的呢？请看下图！先来看3个基类的内存布局：



紧接着是继承类Derived的内存布局：



首先，Derived类**自己的虚函数表指针与其声明继承顺序的第一个基类Base1的虚函数表指针合并**，此外，若Derived类重写了基类中同名的虚函数，则在三个虚函数表的对应项都应该予以修改，Derived中新添加的虚函数位于第一个虚函数表项后面，Derived中新添加的成员变量位于类的最后面，按其声明顺序与内存对齐原则进行排列。

菱形继承的问题及解决方案：虚拟继承

首先在讲这一节之前，先贴出几个重要的信息（干货）：

（1）不同环境下虚拟继承对类大小的影响

在vs环境下，采用虚拟继承的继承类会有自己的虚函数表指针（假如基类有虚函数，并且继承类添加了自己新的虚函数）

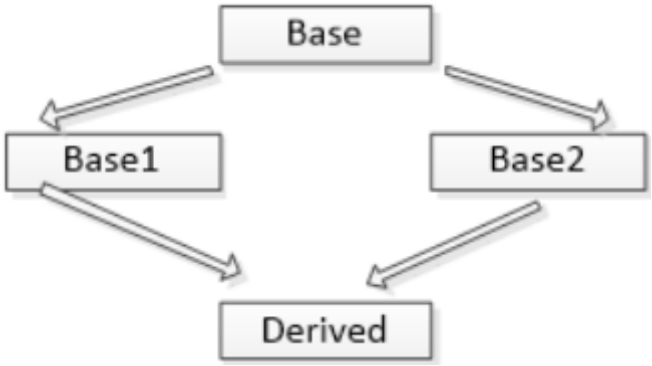
在gcc环境下及mac下使用clion，采用虚拟继承的继承类没有自己的虚函数表指针（假如基类有虚函数，无论添加自己新的虚函数与否），而是共用父类的虚函数表指针

关于以上这一点请详见我的博客：https://blog.csdn.net/longjialin93528/article/details/79874558 ，这里对此进行了超级详细的讲解。

（2）虚拟继承会给继承类添加一个虚基类指针（virtual base ptr 简称vbptr），其位于类虚函数指针后面，成员变量前面，若基类没有虚函数，则vbptr其位于继承类的最前端

关于虚拟继承，首先我们看看为什么需要虚拟继承及虚极继承解决的问题。

虚极继承主要是为了解决菱形继承下公共基类的多份拷贝问题：




```
class Base
{
public:
    int a;
}
class Base1:virtual public Base
{
}
class Base2:virtual public Base
{
}
class Derived:public Base1,public Base2
{
private:
    double b;
public:
}
```

Base1与Base2本身没有任何自身添加的数据成员与虚函数，因此，Base1与Base2都只含有从Base继承来的int a与一个普通的方法，然后Derived又从Base1与Base2继承，这时会导致二义性问题及重复继承下空间浪费的问题：

二义性问题：

1. `Derived de;`
2. `de.a=10;//这里是错误的，因为不知道操作的是哪个a`

重复继承下空间浪费:

`Derived` 重复继承了两次Base中的int a，造成了无端的空间浪费

虚拟继承是怎么解决上述问题的?

虚基继承可以使得上述菱形继承情况下最终的Derived类只含有一个Base类，Base类在虚拟继承后，位于继承类内存布局最后面的位置，继承类通过vbptr寻找基类中的成员及vfptr。

虚拟继承对继承类的内存布局影响可以先看以下示例代码，理解以后，我们在最后列出上述菱形虚拟继承情况下Base1，Base2与Derived代码及内存布局，看到虚拟继承起的作用。

```
class base
{
public:
    int a
    virtual void f();
}
class derived:virtual public base
{
public:
    double d;
    void f();
}
```

Derived类内存布局如下图，由于虚拟继承，Derived只会有一个最初基类的拷贝，该拷贝位于类对象模型的最下面，而想要访问到基类的元素，需要vbptr指明基类的位置（vbptr作用），假如Base中含有虚函数，而继承类中没有增添自己的新的虚函数，那么Derived类统一的布局如下：

如果添加了自己的新的虚函数(代码如下):

```
class base{public:    int a    virtual void f();}class derived:virtual public
base{public:    double d;    void f();    virtual void g();//这是Derived类自己新添
加的虚函数}
```

那么Derived在VC下继承类会**有自己**的虚函数指针，而在Gcc下是**共用**基类的虚函数指针，其分布如下

现在有了上述代码的理解我们可以写出菱形虚拟继承代码及每个类的内存布局:

```
class Base
{
public:
    int a;}
class Base1:public virtual Base
{
}
class Base2:public virtual Base
{
}
class Derived:public Base1,public Base2
{
private:
    double b;
public:
}
```

带实线的框是类确确实实有的，带虚线是针对Base，及Base1，Base2做了扩展后的情况：

Base有虚函数，Base1还添加了自己新的虚函数，Base1也有自己成员变量，Base2添加了自己新的虚函数，Base2也有自己成员变量，则上图全部虚线中的部分都将存在于对象内存布局中。

模板

模板的概念

模板就是建立**通用的模具**，大大**提高复用性**

模板的特点：

- 模板不可以直接使用，它只是一个框架
- 模板的通用并不是万能的

- C++另一种编程思想称为 泛型编程 ， 主要利用的技术就是模板
- C++提供两种模板机制:**函数模板**和**类模板**

函数模板

```
//模板技术 类型参数化 编写代码可以忽略类型
//为了让编译器区分是 普通函数 模板函数
template<classT>//template<typenameT>
void MySwap(T&a,T& b){T temp= a;a = b;b = temp;}
```

模板的定义，在C++中可以在定义类时不指定具体的数据类型，而在编译时进行前期绑定，对于多参数的模板，在参数间用","隔开，如果模板参数为一个类，在模板参数前面必须加上class关键字。

eg: template<class T 1,class T 2>

自动类型推导

```
int a =10;
int b =20;
mySwap(a,b);
double da = 10.20;
double db = 30.40;
mySwap(da,db);
```

显式指定类型

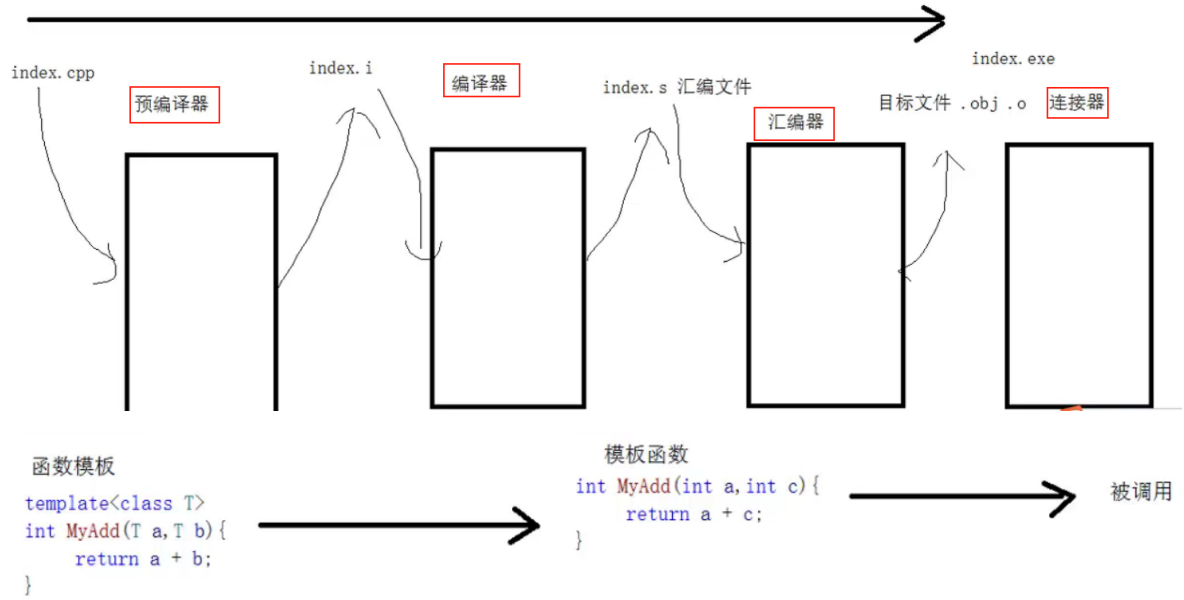
```
mySwap<int>(a,b);
```

调用规则如下：

1. 如果函数模板和普通函数都可以实现，优先调用普通函数
2. 可以通过空模板参数列表来强制调用函数模板
3. 函数模板也可以发生重载
4. 如果函数模板可以产生更好的匹配,优先调用函数模板

普通函数与函数模板区别：

- 普通函数调用时可以发生自动类型转换（隐式类型转换）
- 函数模板调用时，如果利用自动类型推导，不会发生隐式类型转换
- 如果利用显示指定类型的方式，可以发生隐式类型转换



函数模板机制结论：

编译器并不是把函数模板处理成能够处理任何类型的函数

函数模板通过具体类型产生不同的函数

编译器会对函数模板进行两次编译，在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。

类模板

语法：

```
template<typename T>
class A{
public:
    A(T id,T age):Id(id),Age(age){}
    void show(){
        cout<<"id: "<<Id<<" age: "<<Age<<endl;
    }
private:
    T Id;
    T Age;
};
A<int> a(10,20);
```

类模板必须显式指定类型

类模板的成员函数都是模板函数

类模板与函数模板区别

类模板与函数模板区别主要有两点：

1. 类模板没有自动类型推导的使用方式(类模板使用只能用显示指定类型方式)
2. 类模板在模板参数列表中可以有默认参数

类模板中成员函数创建时机

类模板中成员函数和普通类中成员函数创建时机是有区别的：

- 普通类中的成员函数一开始就可以创建
- 类模板中的成员函数在调用时才创建

类模板派生普通类

```cpp
template<class T>
class father{
public:
    father(){age = 0}

    T age;
};
//为什么需要<int> 因为需要分配内存
class son : public father<int>{

};
```

类模板派生类模板

```cpp
template<class T>
class animal{
public:
    void speak(){}
    T age;
};
template<class T>
class cat :public animal<T>{};
```

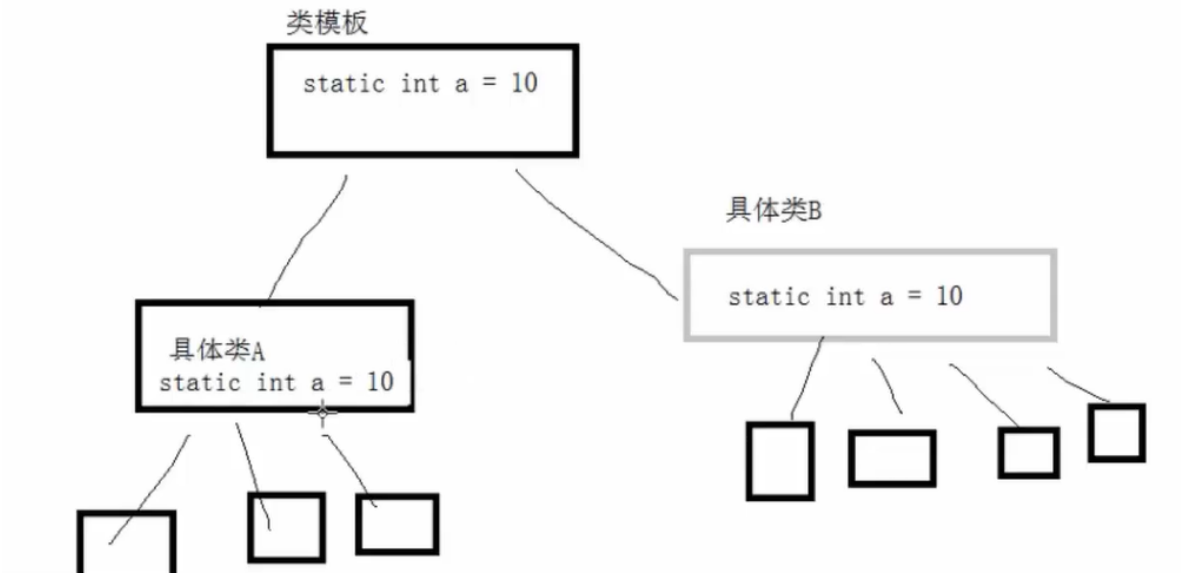
重载运算符

```cpp
template<class T>
class Person{
public:
    //重载左移操作符
    //template<class T>
    friend ostream& operator<<<T>(ostream& os, Person<T>& p);
    Person(T age, T id);
    void Show();
```

.cpp .h分文件书写

```cpp
#include "Person.h"

template<class T>
Person<T>::Person(T age){
    this->age = age;
}

template<class T>
void Person<T>::Show(){
    cout << "Age: " << age << endl;
}
```

类模板的static




```cpp
template<class T>
class Person{
public:
    static int a;
};

//类外初始化
template<class T> int Person<T>::a = 0;
```

```cpp
int main(void)
{

    Person<int> p1, p2, p3;
    Person<char>pp1, pp2, pp3;

    p1.a = 10;
    pp1.a = 100;

    cout << p1.a << " " << p2.a << " " << p3.a << endl;
    cout << pp1.a << " " << pp2.a << " " << pp3.a << endl;

    return 0;
```

其他

省略问题

对于虚拟类型参数所对应的模板实参，如果从模板函数的实参表中获得的信息已经能够判定其中部分或全部虚拟类型参数，而且它们又正好是参数表中最后的若干参数，则模板实参表中的那几个参数可以省略。

对于虚拟类型参数所对应的模板实参，若能够省略可以省略，也可以不省略。

常规参数的信息无法从模板函数的实参表中获得，因此在调用时必须显式的说明。

模板形参理解

模板形参分为两种类型：虚拟类型参数和常规参数。虚拟类型参数须用typename或class定义。

常规参数用具体的类型修饰符（如int、double、char*等）定义。

在定义模板时，关键字typename与class可以互相交换，但在定义类时，只能使用class。typename仅仅用于定义模板，故程序中所有的typename都可以替换成class，但对于用于类定义的class不能使用typename来替换。

问题：用类模板定义对象时，绝对不能省略模板实参 ？

错

在类模板定义对象时，由于没有像函数实参表这样的额外信息渠道，因此无法按函数模板的方式省略模板实参。但是，可以为类模板的参数设置默认值。具体地说，在定义类模板时，可以为模板形参表声明的最后若干个参数设置默认值；而这些有默认值的参数中，最后的若干个对应实参可以在定义对象时省略。