struct 默认是 public

class 默认是 private

宏函数：

C语言宏定义和宏定义函数

宏定义可以帮助我们防止出错，提高代码的可移植性和可读性等。

　　在软件开发过程中，经常有一些常用或者通用的功能或者代码段，这些功能既可以写成函数，也可以封装成为宏定义。那么究竟是用函数好，还是宏定义好？这就要求我们对二者进行合理的取舍。

　　我们来看一个例子，比较两个数或者表达式大小，首先我们把它写成宏定义：

　　#define MAX( a, b) ( (a) > (b) (a) : (b) )

　　其次，把它用函数来实现：

　　int max( int a, int b)

　　{

　　return (a > b a : b)

　　}

　　很显然，我们不会选择用函数来完成这个任务，原因有两个：首先，函数调用会带来额外的开销，它需要开辟一片栈空间，记录返回地址，将形参压栈，从函数返回还要释放堆栈。这种开销不仅会降低代码效率，而且代码量也会大大增加，而使用宏定义则在代码规模和速度方面都比函数更胜一筹；其次，函数的参数必须被声明为一种特定的类型，所以它只能在类型合适的表达式上使用，我们如果要比较两个浮点型的大小，就不得不再写一个专门针对浮点型的比较函数。反之，上面的那个宏定义可以用于整形、长整形、单浮点型、双浮点型以及其他任何可以用“>”操作符比较值大小的类型，也就是说，宏是与类型无关的。

　　和使用函数相比，使用宏的不利之处在于每次使用宏时，一份宏定义代码的拷贝都会插入到程序中。除非宏非常短，否则使用宏会大幅度增加程序的长度。

　　还有一些任务根本无法用函数实现，但是用宏定义却很好实现。比如参数类型没法作为参数传递给函数，但是可以把参数类型传递给带参的宏。

　　看下面的例子：

　　#define MALLOC（n, type） \

　　（ (type \*) malloc（（n）\* sizeof（type）））

　　利用这个宏，我们就可以为任何类型分配一段我们指定的空间大小，并返回指向这段空间的指针。我们可以观察一下这个宏确切的工作过程：

　　int \*ptr;

　　ptr = MALLOC ( 5, int );

　　将这宏展开以后的结果：

　　ptr = (int \*) malloc ( (5) \* sizeof(int) );

　　这个例子是宏定义的经典应用之一，完成了函数不能完成的功能，但是宏定义也不能滥用，通常，如果相同的代码需要出现在程序的几个地方，更好的方法是把它实现为一个函数。

　　下面总结和宏和函数的不同之处，以供大家写代码时使用，这段总结摘自《C和指针》一书。

Map:是一种容器，存储若干元素，这些元素都是由关键值的配对组成的：在一个map中，Key值通常是用来排序或特指元素，映射值用来存储与该Key值绑定的内容。

函数前，类型后面有&，说明返回的是该对象的一个引用，该引用不会被复制。

例如：

Complex &operator=(Complex A1)  
{........}

Decltype：

与auto一样，用于进行编译时类型推导，不过还是有一些区别。Decltype的类型推导并不是像auto一样是从变量声明的初始化表达式获得变量的类型，而是总是以一个普通表达式作为参数，返回该表达式的类型，而且decltype并不会对表达式进行求值。

例如：int i=4;decltype(i) a;//推导结果为int

（可以用来重用匿名类型）

**泛型编程**：结合auto，用于追踪函数的返回值类型。

例如：（**返回类型后置语法**）

Template<typename \_Tx,typename\_Ty>

Auto multiply(\_Tx x, \_Ty y)->decltype(\_Tx\*\_Ty)

{

Return x\*y;

}

注意，是双下划线，而不是单下划线  
[\_\_FILE\_\_](https://www.baidu.com/s?wd=__FILE__&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao) 包含当前程序文件名的字符串  
\_\_LINE\_\_ 表示当前行号的整数  
\_\_DATE\_\_ 包含当前日期的字符串  
\_\_STDC\_\_ 如果编译器遵循[ANSI](https://www.baidu.com/s?wd=ANSI&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank) [C标准](https://www.baidu.com/s?wd=C%E6%A0%87%E5%87%86&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，它就是个非零值  
\_\_TIME\_\_ 包含当前时间的字符串  
例如：  
输出当前程序文件名的程序：  
//---------------------------------------------------------------------------  
#include <stdio.h>  
int main(void)  
{  
printf("%s",[\_\_FILE\_\_](https://www.baidu.com/s?wd=__FILE__&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao));  
return 0;  
}  
//---------------------------------------------------------------------------

虚函数：

虚函数是指一个类中你希望重载的成员函数，当你用一个基类指针或引用指向一个继承类对象的时候，你调用一个虚函数，实际调用的是继承类的版本。

在成员函数的形参后面写上=0，则成员函数为纯虚函数。

（1）纯虚函数没有函数体；   
（2）最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是虚函数”；   
（3）这是一个声明语句，最后有分号。

纯虚函数只有函数的名字而不具备函数的功能，不能被调用。

纯虚函数的作用是**在基类中为其派生类保留一个函数的名字，以便派生类根据需要对他进行定义**。如果在基类中没有保留函数名字，则无法实现多态性。

如果在一个类中声明了纯虚函数，在其派生类中没有对其函数进行定义，则该虚函数在派生类中仍然为纯虚函数。

我们把包含纯虚函数的类称之为**抽象类**。比如刚刚举的Shape类当中就含有一个计算周长的纯虚函数，那么，我们就说这个Shape类是一个抽象类。大家可以想一想，如果我们使用Shape这个类去实例化一个对象，那么这个对象实例化之后，如果想要去调用纯虚函数（比如要去调用这个计算周长的纯虚函数），那怎么去调用呢？？？我们说，显然是无法调用的。所以，我们得出一个结论：**对于抽象类来说，C++是不允许它去实例化对象的**。也就是说，抽象类无法实例化对象。

**对于抽象类来说，它无法实例化对象，而对于抽象类的子类来说，只有把抽象类中的纯虚函数全部实现之后，那么这个子类才可以实例化对象**。

关于构造函数和new

class CA  
{  
public:  
CA(int a) { }  
};

CA \*cap = new CA(1); 指针  
CA ca(1); 对象

**重载：**

在一个类内，如果存在若干个同名函数，而且这些函数之间可以用形参个数或者形参类型区分开来的时候（注意不能靠函数返回类型区分），这几个函数就互为重载函数。这时，当你通过类对象调用这几个函数时，编译器就可以通过你传递的实参个数或者类型，去匹配相应的函数，而不会发生歧义。这也就是重载函数的作用所在（让你可以使用若干个同名函数）。需要注意的是：

1.重载绝对不会发生在基类和派生类之间，如上例所示。当基类和派生类中存在同名函数时，无论同名函数的形参个数或者类型是否相同，派生类中的同名函数都会将基类中的同名函数隐藏掉，因此它们是隐藏关系，而不是重载关系。关于隐藏，后边会提到。如此以来，上例的14行在编译时就会报错，提示类B中没有test（int）类型的函数。

2.在同一个类中，重载函数之间必须依靠形参个数或者形参类型来进行区分，不能依靠返回类型。也就是说，如果同一个类中的两个同名函数形参个数和类型完全相同，但是返回值类型不同，这时候编译就会报错，因为当你通过类对象调用该同名函数时，编译器会出现二义性，不知道该选择哪个函数。记着，重载必须靠形参来区分。

3.在同一个类中，虚函数和虚函数，虚函数和普通函数之间也可以重载，规则完全同上。虚函数下边会提到。

**隐藏：**

隐藏只能出现在基类和派生类之间，而不能发生在同一个类内（比如上述2中，只会引起编译器出现二义性）。当基类和派生类中存在同名函数时，无论同名函数的形参个数或者类型是否相同，派生类中的同名函数都会将基类中的同名函数隐藏掉，而不会是重载关系。这时，当你通过派生类对象调用该同名函数时，只能访问派生类的该函数，如果硬要访问基类的该函数，则需要在函数名前加上类作用域，如上边代码所示。

**覆盖：**

覆盖也只能出现在基类和派生类之间，当派生类和基类中的存在同名函数，且参数个数和参数类型完全相同，并且基类中的该函数有virtual修饰（派生类中的该函数可有可无），则派生类的该函数覆盖掉基类的该函数。该性质用来实现多态。

**虚函数：**

在一个类中，用virtual关键字声明的函数都是虚函数。虚函数存在的唯一目的，就是为了实现多态（动态绑定/运行时绑定）。关于多态，后面会提到。虚函数只有在基类和派生类之间才能发挥虚特性（也就是说才能发挥虚函数的真正的目的）。在同一个类中，所有虚函数就和普通函数是一样，使用同样的重载规则（重载的第3点中提到过）。因此在同一个类中可以把虚函数看作普通函数来使用（因为其虚特性发挥不出来），使用方法和注意事项与普通函数一模一样。

**多态：**

多态是面向对象思想的精髓所在。说白了，就是通过基类指针或引用调用一个成员函数时，直到运行阶段在才能决定该成员函数是哪个派生类中定义的成员函数。

父类指针指向子类：

当基类的指针（P）指向派生类的时候，只能操作派生类中从基类中继承过来的数据。  
指向派生类的指针，因为内存空间比基类长，会导致严重了后果，所以不允许派生类的指针指向基类。而基类的指针可以指向派生类。

赋值兼容规则永远不变，父类指针既可以指向父类对象，也可以指向子类对象；当父类指针指向父类对象时，访问父类的成员；当父类指针指向子类对象时，那么**只能访问子类中从父类继承下来的那部分成员**；**不能访问子类独有的成员**，如果访问，编译阶段会报错；

函数隐藏（将父类的成员隐藏，**通过子类对象或者在子类内部只能访问子类的同名成员；但是父类的成员仍然存在**），是指当子类中有和父类同名的成员变量时，此时子类中有两个名字相同的成员变量；那么如果此时我们用父类指针指向子类对象，来访问父子类中名字相同的成员变量时，由于赋值兼容规则，此时绝对访问的是继承来的父类的同名成员变量，但是，如果我们在子类内部或者用子类的对象在子类外部访问此同名成员，此时编译器会将父类的成员隐藏，子类对象访问到的是自己类中重新定义的成员；对于成员函数来说，只要构成了隐藏机制，访问效果也一样；

多态是指；当子类中重写了父类的虚函数，那么通过父类的指针或者引用指向父类对象或着子类对象；用该父类指针或者引用调用父子中被重写的函数；**就会有父类指针指向父类对象，调用的是父类中的虚函数；父类指针指向的是子类对象，调用的是子类虚函数**；此时，也符合赋值兼容规则；不贵父类的指针或者引用指向父类对象或者子类对象；这个父类指针或者引用都只能访问父类的部分，

当指向父类的对象时，由于本身是父类指针；且父类对象模型中的虚函数表指针所指向的虚函数表中只存放父类的虚函数地址；所以父类的指针调用的重写虚函数时父类的虚函数；

但是当父类指针指向子类对象时，此时子类对象中的父类的虚函数表指针所指的虚函数表中的父类重写虚函数的地址被改写为子类的重写虚函数的地址，所以此时父类指针访问子类中父类的虚函数表时，找到的要调用的同名虚函数是子类的同名虚函数地址；所以父类指针调用子类的重写的同名虚函数；

**虚函数重写（覆盖）的实质就是重写父类虚函数表中的父类虚函数地址；**

当定义一个指向子类实例的父类指针的时候，内存中实例化了子类，由于子类继承了父类,因此内存中的子类里包含父类的所有成员。但由于生命的时父类指针，因此该指针不能够访问子类的成员，而只能访问父类的成员。然而在父类里可以声明纯虚函数和定义虚函数，使用父类指针访问虚函数或纯虚函数的时候，访问到的是子类里重写的函数。当然，**对于虚函数，如果子类里没有对其重写的话，仍然访问到父类里定义的虚函数。**可见虚函数和纯虚函数的却别仅仅在于：纯虚函数没有定义,只有声明。

关于C++中，类的常成员函数

声明样式为：   返回类型 <类标识符::>函数名称(参数表) const

一些说明：

1、const是函数声明的一部分，在函数的实现部分也需要加上const

2、const关键字可以重载函数名相同但是未加const关键字的函数

3、常成员函数**不能用来更新类的成员变量**，也**不能调用类中未用const修饰的成员函数**，只能调用常成员函数。即常成员函数不能更改类中的成员状态，这与const语义相符。

析构函数在下边3种情况时被调用：

1. 对象生命周期结束，被销毁时；
2. 主动调用delete ；
3. 对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用。

#include <iostream>

using namespace std;

class A

{

public:

A()

{

cout << "constructing A" << endl;

}

~A()

{

cout << "destructing A" << endl;

}

private:

int a;

};

class C

{

public:

C()

{

cout << "constructing C" << endl;

}

~C()

{

cout << "destructing C" << endl;

}

private:

int c;

};

class B : public A

{

public:

B()

{

cout << "constructing B" << endl;

}

~B()

{

cout << "destructing B" << endl;

}

private:

int b;

C c;

};

void main()

{

B b;

}

运行结果：

constructing A

constructing C

constructing B

destructing B

destructing C

destructing A

B的析构函数调用之后，又调用了B的成员c的析构函数 。

若将上边的代码中的main()函数内容改成

A\* a = new B;

delete a;

我们知道，这将不会调用class B的析构函数不会被调用，所以class C的析构函数也不会被调用。

运行结果：

constructing A

constructing C

constructing B

destructing A

**若将class A中的析构函数声明为虚函数 ，这时class B的析构函数也会被调用，例如：**

#include <iostream>

using namespace std;

class A

{

public:

A()

{

cout << "constructing A" << endl;

}

virtual ~A()

{

cout << "destructing A" << endl;

}

private:

int a;

};

class C

{

public:

C()

{

cout << "constructing C" << endl;

}

~C()

{

cout << "destructing C" << endl;

}

private:

int c;

};

class B : public A

{

public:

B()

{

cout << "constructing B" << endl;

}

~B()

{

cout << "destructing B" << endl;

}

private:

int b;

C c;

};

void main()

{

A\* a = new B;

delete a;

}

运行结果：

constructing A

constructing C

constructing B

destructing B

destructing C

destructing A

迭代器使用（iterator）：

#include <vector>

#include <iostream>

using namespace std;

int main() {

vector<int> ivec;

ivec.push\_back(1);

ivec.push\_back(2);

ivec.push\_back(3);

ivec.push\_back(4);

for(vector<int>::iterator iter = ivec.begin();1. iter != ivec.end(); ++iter)

cout << \*iter << endl;

}

assert宏的原型定义在<assert.h>中，其作用是如果它的条件返回错误，则终止程序执行。

#include <assert.h>  
void assert( int expression );

**assert的作用就是计算表达式 expression ，如果其值为假（即为0），那么它先向stderr打印一条出错信息，然后通过调用 abort 来终止程序运行。**

RTTI（Run-Time Type Identification)，通过运行时类型识别能够使用[基类](https://baike.baidu.com/item/%E5%9F%BA%E7%B1%BB/9589663" \t "_blank)的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304)或引用来检查这些指针或引用所指的对象的实际[派生类](https://baike.baidu.com/item/%E6%B4%BE%E7%94%9F%E7%B1%BB)型。

RTTI提供了以下两个非常有用的操作符：

（1）typeid操作符，返回[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "_blank)和引用所指的实际类型。

（2）[dynamic\_cast](https://baike.baidu.com/item/dynamic_cast)操作符，将[基类](https://baike.baidu.com/item/%E5%9F%BA%E7%B1%BB)类型的指针或引用安全地转换为[派生类](https://baike.baidu.com/item/%E6%B4%BE%E7%94%9F%E7%B1%BB)型的指针或引用。

make clean  
清除上次的make命令所产生的object文件（后缀为“.o”的文件）及可执行文件。

typedef std::vector<T>::size\_type size\_type;//why not?

实际上，模板类型在实例化之前，**编译器并不知道vector<T>::size\_type是什么东西**，事实上一共有三种可能：

**静态数据成员**

**静态成员函数**

**嵌套类型**

那么此时typename的作用就在此时体现出来了——**定义就不再模棱两可**。

总结

所以根据上述两条分析，

typedef typename std::vector<T>::size\_type size\_type;

语句的真是面目是：

typedef创建了存在类型的别名，**而typename告诉编译器std::vector<T>::size\_type是一个类型而不是一个成员。**

https://www.boost.org/doc/libs/1\_31\_0/libs/mpl/doc/ref/Reference/

Push\_front：

typedef push\_front<Sequence,T>::type s;

A model of [Extensible Sequence](https://www.boost.org/doc/libs/1_31_0/libs/mpl/doc/ref/Extensible_Sequence.html)

Equivalent to typedef insert< Sequence,begin<Sequence>::type,T >::type s;

List\_c:

**Description**

Similary to [vector\_c](https://www.boost.org/doc/libs/1_31_0/libs/mpl/doc/ref/Reference/vector_c.html), list\_c is a shorcut interface whose whole purpose is to make the creation of a list of [Integral Constant](https://www.boost.org/doc/libs/1_31_0/libs/mpl/doc/ref/Integral_Constant.html)s less verbose:

typedef list\_c<unsigned long,-1,0,1,1,-1,0,0,1,-1> data;

If list\_c didn't exist, instead of the above line you would have to write this:

typedef list<

integral\_c<unsigned long,-1>

, integral\_c<unsigned long,0>

, integral\_c<unsigned long,1>

, integral\_c<unsigned long,1>

, integral\_c<unsigned long,-1>

, integral\_c<unsigned long,0>

, integral\_c<unsigned long,0>

, integral\_c<unsigned long,1>

, integral\_c<unsigned long,-1>

> data;

**Explcit**（阻止“以赋值语法进行带有转型操作的初始化”）

例如：A a=3;//编译器能通过，因为A中有构造函数只有一个参数（单参数），编译通过隐式转换，调用了构造类型。可以通过epplcit来阻止这个行为。

**Noexpect**（声明函数不能抛出任何异常，若定义了noexpect的函数抛出异常，则程序终止）

占位符：

#include <functional>

using namespace std::placeholders;

1.可以提前声明函数类型:function< 返回值 ( 参数列表) >  函数名;

              function< int (int, double, string) > fn

  也可以不用声明类型直接bind前面：auto fn = bind(  ....)

   函数参数的对应：以实际函数的参数对应声明的位置：

            int test(double, string, int)

            fn = bind(test, \_2, \_3, \_1)

-->test参数的第一个类型double，对应于function的\_2位置，也可设置常量参数，fn = bind(test, 3.5, "hello", 6);

**2.bind类的成员函数时：**

**bind(&A::print, &a, ...)**