目录

* + - [构造与析构](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "构造与析构" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [构造函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "构造函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [析构函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "析构函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [复制构造函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "复制构造函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [深复制和浅复制](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "深复制和浅复制" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [组合类的构造函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "组合类的构造函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [const与static](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "const与static" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [常对象](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "常对象" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [常成员函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "常成员函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [常数据成员](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "常数据成员" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [类中静态变量和常量的初始化](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "类中静态变量和常量的初始化" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [常引用](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "常引用" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [函数声明](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "函数声明" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [函数默认值](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "函数默认值" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [枚举](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "枚举" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [异常](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "异常" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [异常接口声明](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "异常接口声明" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [异常处理中的构造与析构](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "异常处理中的构造与析构" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [抛出string类型的异常的注意事项](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "抛出string类型的异常的注意事项" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [友元](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "友元" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [类的继承与派生](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "类的继承与派生" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [继承方式](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "继承方式" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [虚基类](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "虚基类" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [派生类的构造函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "派生类的构造函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [派生类的复制构造函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "派生类的复制构造函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [cingets和getline](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "cingets和getline" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [内联函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "内联函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [类的内联成员函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "类的内联成员函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [动态创建的数据与对象](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "动态创建的数据与对象" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [动态创建基本类型的变量](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "动态创建基本类型的变量" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [动态创建类的对象](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "动态创建类的对象" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [动态创建数组类型的对象](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "动态创建数组类型的对象" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [删除动态申请的内存](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "删除动态申请的内存" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [运算符重载](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "运算符重载" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [运算符重载规则](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "运算符重载规则" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [运算符重载的两种形式](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "运算符重载的两种形式" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [对于和--的重载](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "对于和-的重载" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [需要重载为非类成员函数的情况](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "需要重载为非类成员函数的情况" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [引用 指针和数组](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "引用-指针和数组" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [引用](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "引用" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [指针](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "指针" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [函数指针](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "函数指针" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [this指针](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "this指针" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [指向类的非静态成员的指针](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "指向类的非静态成员的指针" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [指向类的非静态成员的指针](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "指向类的非静态成员的指针-1" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [数组初始化](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "数组初始化" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [字符数组](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "字符数组" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [结构体和联合体](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "结构体和联合体" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [结构体](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "结构体" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [联合体](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "联合体" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [结构体成员初始化](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "结构体成员初始化" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [模板](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "模板" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [函数模板](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "函数模板" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [类模板](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "类模板" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
    - [虚函数与运行时多态](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "虚函数与运行时多态" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [虚函数](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "虚函数" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [运行时多态的条件](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "运行时多态的条件" \t "https://blog.csdn.net/qq_21397217/article/details/_self)
      * [纯虚函数和抽象类](https://blog.csdn.net/qq_21397217/article/details/51656754" \l "纯虚函数和抽象类" \t "https://blog.csdn.net/qq_21397217/article/details/_self)

### **构造与析构**

#### **构造函数**

* 构造函数是类的成员函数

#### **析构函数**

* 析构函数在对象生存期结束前****自动调用****
* 析构函数不接受任何参数
* 析构函数可以是虚函数

#### **复制构造函数**

* 类的复制构造函数的形参是此类对象的引用

类的复制构造函数在以下情况被调用：

* 使用类的对象去初始化此类的另一个对象时
* 函数的参数是类的对象，在调用函数进行实参和形参的结合时
* 函数的返回值是类的对象，在函数执行完毕返回值时

#### **深复制和浅复制**

* 默认的复制构造函数实现的是浅复制
* 为类中的每个内嵌对象都实现复制构造函数才能实现深复制
* 一般将复制构造函数的参数设为const类型

声明和实现复制构造函数的一般方法：

class Point {public:

Point(Point &p);private:

int x, y;

};

Point::Point(const Point &p){

x = p.x;

y = p.y;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

#### **组合类的构造函数**

创建组合类的对象时的构造函数调用顺序：

* ****首先****调用内嵌对象的构造函数，初始化内嵌对象
* 内嵌对象的构造函数的调用顺序和该对象在类中的声明顺序****相同****
* 内嵌对象的构造函数的调用顺序和其在初始化列表中的顺序****无关****
* 若对象没有出现在初始化列表中，则调用该对象的默认构造函数
* ****最后****调用本类构造函数
* 析构函数的调用顺序与构造函数的调用顺序相反
* 内嵌对象的析构函数的调用顺序和其在类中的声明顺序****相反****
* 若没有编写复制构造函数，则会自动生成隐含的复制构造函数，该函数自动调用内嵌对象的复制构造函数

****必须****在初始化列表中初始化的数据成员：

* 没有默认的无参构造函数的内嵌对象——此类对象初始化时必须提供参数
* 引用类型的数据成员——引用类型变量必须在声明的同时进行初始化

组合类构造函数定义的一般形式：

类名 :: 构造函数名(形参表) : 内嵌对象1(形参表), ... {

/\*构造函数体\*/

}

* 1
* 2
* 3

形参表中的形参可以是此类对象的引用（将调用复制构造函数）。   
其中内嵌对象1(形参表), 内嵌对象2(形参表), ...称为初始化列表，其作用是对内嵌对象进行初始化。

class Inner {public:

Inner(Iparam1, Iparam2, ...);

};

...

class Outer {public:

Outer(Oparam1, Oparam2, ...);private:

Inner1 i1;

Inner2 i2;

...

};

Outer :: Outer(Oparam1, Oparam2, ...) : i1(Iparam1, Iparam2, ...), i2(...), ...{

//构造函数主体

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

### **const与static**

#### **常对象**

* 常对象的数据成员值在对象整个生存期间内不能改变
* 常对象必须在声明的同时初始化，而且不能更新
* 通过常对象只能调用其常成员函数

#### **常成员函数**

* 常成员函数在定义和声明定义的时****都要使用****const关键字
* const可以用作区分重载
* 在仅以const作为重载的区分时，普通对象将默认调用普通成员函数
* 可以通过常函数访问普通数据成员
* 常函数不能更新目的对象的数据成员(下一条为原因)
* 在调用常成员函数期间，即使目的对象是普通对象，也按常对象处理

#### **常数据成员**

* 任何函数中都不能对常数据成员进行赋值
* 类中的常数据成员只能通过其构造函数在初始化列表中进行初始化

#### **类中静态变量和常量的初始化**

* 类中的静态变量和常量都应该在类外加以定义
* 上一条的****例外****：若类的静态常量如果具有整形或者枚举类型则可以直接在类中为其指定常量值

class A {public:

A(int i);private:

const int a;

static int b;

};//在初始化列表中初始化常量

A::A(int i):a(i){

//构造函数的其他内容

}//在类外初始化静态成员int A::a = 1;//必须要加数据类型关键字

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

#### **常引用**

* 常引用所引用的对象不能被更新
* 非常引用只能绑定到普通对象，不能绑定到常对象
* 通过常引用访问普通对象时，将该对象按常对象处理

### **函数**

#### **函数声明**

* 可以在函数内部声明函数，不可以在函数内部定义(实现)函数
* 在函数内部声明的函数只在此函数体内有效

#### **函数默认值**

* 带有默认值的参数必须在参数表的最后边
* 相同作用域内不可以对同一个参数的默认值重新定义，即使值相同也不行
* 如果函数声明中没有为函数参数设置默认值，则可以在函数实现时为参数设置默认值
* 类成员函数的默认值必须写在类定义中，不能写在类实现中

//下面的做法是错的，声明和实现不能重复设置默认值void fun(int a = 1, int b = 2);void fun(int a = 1, int b = 2){ }

* 1
* 2
* 3

### **枚举**

* 枚举元素按整型常量处理，不能赋值，所以不能进行++、–等运算
* 枚举元素默认值从0开始，可以在定义枚举时赋初值，元素值自动递增
* 使用枚举元素时直接使用元素名，不可以加枚举名，在元素前加MyEnum.或MyEnum::是错的
* 将整数赋值给枚举类型变量时，需要强制类型转换myEnum = MyEnum(number);
* MyEnum myEnum; myEnum = MyEnum(1);则e的值为1，无论枚举中是否有1这个值

### **异常**

* throw可以抛出普通数据类型，也可以抛出类的对象
* catch的参数可以是普通数据类型，也可以是类对象
* catch按顺序检查是否可以捕获所抛出的异常
* 如果异常被前面的catch捕获，则后面的catch不会执行
* 如果异常类型声明是一个省略号catch(...)，则该catch子句可以捕获****所有类型****的异常
* 能够捕获所有类型的异常的catch块****必须****放在****最后****

catch后的异常类型可以不声明形参，但这样无法访问所捕获到的异常对象。   
使用不带操作数的throw可以将当前捕获的异常再次抛出，但是只能在catch块或catch块中调用的函数中使用。   
若异常抛出点本身不在任何try-catch块内，则结束当前函数的执行，回到当前函数的调用点，把调用点作为异常的抛出点，然后重复这一过程。

/\*throw表达式语法\*/throw 表达式 ;

/\*try-catch块语法\*/try {

//可能发生异常的内容

} catch (异常类型1 形参名) {

//捕获到异常后的处理方法

} catch (异常类型2 形参名) {

//将当前捕获到的异常再次抛出，将抛出源异常对象，而不是其副本

throw;

} catch (...){

//使用...捕获所有类型的异常

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

#### **异常接口声明**

* 声明了异常接口的函数****能且只能****抛出声明了的异常
* 若函数没有异常接口声明，则此函数可以抛出任何异常
* 若写成throw ()的形式，则此函数不抛出任何异常
* 若要使用异常接口声明，则在函数定义和实现时都要声明异常接口

如果函数抛出了异常接口声明中所没有的异常，unexpected函数会被调用，该函数默认会调用terminate函数中止程序。用户可以自定义unexpected函数的行为。

/\*在函数声明中说明函数可以抛出哪些异常\*/

返回类型 函数名(参数表) throw (异常类型1, 异常类型2, ...);

/\*不抛出任何异常的函数\*/

返回类型 函数名(参数表) throw ();

* 1
* 2
* 3
* 4
* 5

#### **异常处理中的构造与析构**

发生异常时，从进入try块（捕获到异常的catch所对应的那个）直到异常抛出前，这期间栈上构造的并且没被析构的所有对象都会被****自动析构****，这一过程被称为****栈的解旋****。

#### **抛出string类型的异常的注意事项**

当抛出的异常为string类型时，要注意****不能****直接抛出匿名字符串，而要先声明一个string类型的变量或常量，然后再将该变量或常量抛出。

void fun() throw (string) {

...

const string str = "msg";

throw msg;

//throw "msg";//这样写的话捕获不到异常

}

* 1
* 2
* 3
* 4
* 5
* 6

### **友元**

* 友元关系不能传递
* 友元关系不能继承
* 声明友元类不需要前向引用声明，因为不涉及友元类的对象
* 声明友元类时，即使友元类还不存在，也不会出错
* 一个类的友元类中的函数全部是该类的友元函数

class A {

...

friend class B;//声明友元类B

friend void fun();//声明友元函数fun()

...

}

* 1
* 2
* 3
* 4
* 5
* 6

### **类的继承与派生**

* 派生类将接受基类的****除构造函数和析构函数以外****的全部成员，包括static和const成员
* 如果派生类中声明了和基类成员函数同名的新函数，即使函数参数表不同，也会将从基类继承来的同名函数的****全部****重载形式隐藏
* 如果要访问被隐藏的成员，要使用作用域分辨符或基类名来限定

//使用基类名限定

obj.Base1::var;

obj.Base2::fun();

//使用作用域标识符限定class Derived:public Base1, public Base2{

...

using Base1::var;

using Base2::fun;//不加括号

...

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

#### **继承方式**

公有继承 public   
当类的继承方式为公有继承时，基类的public成员和protected成员的访问属性在派生类中不变，而基类的private成员不可直接访问。

保护继承 protected   
当类的继承方式为保护继承时，基类的public成员和protected成员的访问属性在派生类中变为protected，而基类的private成员不可直接访问。

私有继承 private   
当类的继承方式为私有继承时，基类的public成员和protected成员的访问属性在派生类中变为private，而基类的private成员不可直接访问。****默认的****继承方式为private。

#### **虚基类**

若派生的的多个直接基类还有共同的基类，则直接基类中从共同基类继承来的成员有相同的名称。在派生类对象中这些同名数据成员在内存中同时拥有多个副本，同名函数会有多个映射。   
可以使用作用域标识符来区分它们，也可以将共同基类设为****虚基类****，这时从不同路径继承来的同名数据成员在内存中就只有一个副本，同名函数也只有一个映射。

//class 派生类名:virtual 继承方式 基类名

class Base0{};

class Base1:virtual public Base0{};

class Base2:virtual public Base0{};

class Drived:public Base1, public Base2{};//Base0不是Drived的直接基类，不加virtual

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

#### **派生类的构造函数**

派生类构造函数的语法形式：

派生类名::构造函数名(参数表):基类名(参数表), ..., 派生类初始化列表{

//派生类函数体

}

* 1
* 2
* 3

如果虚基类有含参构造函数，并且没有声明无参构造函数，则在整个继承过程中，直接或者间接继承虚基类的所有派生类，都****必须****在构造函数的初始化列表中显式对虚基类初始化。

虚基类的构造函数不会被多次调用，只有****距离虚基类最远****的派生类的构造函数才会调用虚基类的构造函数，而其他派生类对虚基类的构造函数的调用会被****自动忽略****。

class Base0{public:

Base0(param);//虚基类含参构造函数

};

class Base1:virtual public Base0{public:

Base1(param):Base0(param);//初始化列表传参

};

class Base2:virtual public Base0{public:

Base2(param):Base0(param);//初始化列表传参

};

class Drived:public Base1, public Base2{public:

Drived(param):Base0(param);//初始化列表传参

};

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

派生类对象的构造顺序：

* 按照****声明派生类时的继承顺序****调用基类构造函数来初始化基类的数据成员
* 初始化派生类新增的成员对象
* 执行派生类构造函数的函数体

#### **派生类的复制构造函数**

如果为派生类编写复制构造函数，一般要为其基类的复制构造函数传递参数。   
应该将派生类对象作为其基类复制构造函数的参数。

Derived::Derived(const Derived &param):Base(param), ...{

//派生类复制构造函数体

}

* 1
* 2
* 3

在定义类之前使用该类，需要使用前向引用声明。   
在提供类的完整定义之前，不能定义该类的对象，也不能在****内联****成员函数中使用该类的对象。

class B;//前向引用声明

class A {

...

B b;//错误！类A不完整，不能定义它的对象

B &rb;//正确

B \*pb;//正确

};

class B {

...

};

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

### **cin、gets()和getline()**

使用cin读取数据时，遇到空格会停止读入。   
使用gets(char\*)和getline(cin, string, char)读入一整行数据。   
如果在gets()或getline()函数前使用了cin，注意要清除cin留下的换行符，参见下面的示例。

getline()默认使用换行\n作为读取结束的标志，也可以自定义结束标志。   
在getline()函数的第三个参数处设置结束标志，传入的字符将会最为结束的标志（\n仍然有效）。

char ch[100];string str;

gets(ch);

getline(cin, str);

getline(cin, str, ',');//将半角逗号`,`设为读取结束标志

/\*

\* 如果gets()或者getline()函数的前一行使用cin读取数据，

\* 那么应该在gets()或者getline()函数之前使用getchar(),

\* 否则gets()或者getline()会把cin结束的换行符读入而产生错误

\*/

cin>>n;

getchar();//使用getchar()防止下一行读入cin的换行符

gets(ch);

getline(cin, str);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

### **内联函数**

* 使用inline关键字声明内联函数
* 内联函数不在调用时发生跳转，而是在编译时将函数体嵌入到每一个调用函数的地方
* 内联函数应该是比较简单的函数

#### **类的内联成员函数**

* 隐式声明：将函数体直接放在类定义中
* 显示声明：在类外实现类函数时，在函数返回值类型前加inline关键字

### **动态创建的数据与对象**

#### **动态创建基本类型的变量**

type \* ptr = new type(val);   
或   
type \* ptr;   
ptr = new type(val);

* val将成为所申请的空间中所存储的默认值
* 如果()中不写任何值，则将初值设为0
* 如果不希望设定初值，可以将()省略

#### **动态创建类的对象**

创建方法同上，将val换成初始化列表

* 若类存在自定义的无参构造函数，则new ClassName等效于new ClassName()
* 若类无自定义的无参构造函数，则new ClassName调用隐含的默认构造函数，new ClassName()调用隐含的默认构造函数，还会将类中基本数据成员和指针成员赋初值0，并且该约束****递归****作用于类的内嵌对象

#### **动态创建数组类型的对象**

type \* ptr = new type[len];//末尾加()可以全部初始化为0

* 1

#### **删除动态申请的内存**

delete ptr;delete[] ptr;

* 1
* 2

### **运算符重载**

#### **运算符重载规则**

* C++中.、\*、::和三目运算符?:不可以重载，其他运算符都可以重载
* =、[]、()和->****只能****重载为类的成员函数
* 派生类中的=运算符函数总是会隐藏基类中的=运算符函数
* 只能重载C++中已有的运算符
* 运算符重载后优先级和结合性不变

#### **运算符重载的两种形式**

（使用op代指被重载的运算符）

* 重载为类的非静态成员函数，obj1 op obj2相当于obj1.operator op(obj2)
* 重载为非类成员函数，obj1 op obj2相当于operator op(obj1, obj2)
* 上述****4种****写法都能调用相应的运算符重载函数
* 运算符重载函数的参数通常是参与运算的对象的引用

返回类型 operator 运算符 (形参表){

//运算方法体

}

* 1
* 2
* 3

class A{public:

A(int n){this->n = n;}

//重载为类的非静态成员函数

A operator + (const A &a){

return A(n + a.n);

}

int n;

} ;

//重载为非类成员函数

A operator - (const A &a1, const A &a2){

return A(a1.n - a2.n);

}

int main(){

A a1(10);

A a2(20);

cout<< a1.n <<" "<< a2.n <<endl;//10 20

//两种调用方式

A a31 = a1 + a2;

A a32 = a1.operator + (a2);

cout<< a31.n <<" "<< a32.n <<endl;//30 30

//两种调用方式

A a41 = a1 - a2;

A a42 = operator - (a1, a2);//-10 -10

cout<< a41.n <<" "<< a42.n <<endl;

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29

当以非类成员函数的方式重载运算符时，有时需要访问类的私有成员，可以将运算符重载函数设为类的****友元函数****。可以不使用友元函数时，不要使用。   
当运算符重载为类的成员函数时，函数的参数要比运算符原来的操作数少一个（后置++和--除外）；   
当运算符重载为非类成员函数时，函数的参数和运算符原来的操作数相同。

#### **对于++和--的重载**

* 当++和--重载为前置运算符时，运算符重载函数****不需要****任何参数
* 当++和--重载为后置运算符时，运算符重载函数****需要一个**int**型形参****，该参数只用作区分前置运算和后置运算，没有其他作用，声明和实现函数时，都****可以不给出形参名****

#### **需要重载为非类成员函数的情况**

* 要重载的操作符的第一个参数为不可更改的类型
* 以非类成员函数的形式重载，可以支持更灵活的类型转换

### **引用、 指针和数组**

#### **引用**

* 声明引用的同时必须对其进行初始化
* 引用被初始化之后，不能再更改其指向的对象
* 使用&声明引用

type var;//声明变量

type &ref = var;//声明变量var的引用

* 1
* 2

#### **指针**

* 将指针赋值为0或NULL表示空指针，它不指向任何地址
* 通过指针访问内含的成员时要使用->

对于指针和数组：   
\*(ptr + val) 等价于 ptr[val]

指向常量的指针   
const type \* ptr = &const\_val;   
指向常量的指针本身可以被改变，再指向其他的对象。

指针类型的常量   
type \* const ptr = &val;   
常指针不能再指向其他对象，若其所指对象不是常量，则所指对象可以被修改

void类型的指针可以指向任何类型的对象

#### **函数指针**

声明一个函数指针，初始化

type (\* ptrName)(params);

type fun(params){...}

ptrName = fun;

* 1
* 2
* 3

不加\*和&，指针所指函数必须和指针具有相同的返回类型和形参表。

#### **this指针**

* this指向调用当前方法的那个对象自身

class A {public:

A(int a);private:

int a;

};

A::A(int a){

//通过this消除内部变量对外部变量的屏蔽

this->a = a;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

#### **指向类的非静态成员的指针**

指向数据成员的指针   
type ClassName::\*ptrName;   
ptrName = &ClassName::varName;

指向函数成员的指针   
type (ClassName::\*ptrName)(params);

以上要注意访问权限。

* 访问数据成员   
  objName.\*ptrName 或者 objPtrName->\*ptrName

#### **指向类的非静态成员的指针**

访问类的非静态成员不依赖对象，所以可以用普通指针访问类的非静态成员。

type \*ptrName = &ClassName::varName;

* 1

#### **数组初始化**

* 指定的初值的个数小于数组大小时，剩下的数组元素自动初始化为0
* 静态生存期的数组元素会被默认初始化为0
* 动态生存期的数组元素默认的值是不确定的

int arr[len] = {1, 2, 3, ..., len};int arr[] = {1, 2, 3, ..., len};//数组长度为lenint arr[len] = {1, 2, 3, ..., i};//i<len，后续元素自动初始化为0

int arr[row][col] = {1, 2, 3, ..., row \* col};int arr[row][col] = {

{1, 2, 3, ..., col},

{2, 3, 4, ...},

...,

{col, ...}

};

int arr[][col] = {1, 2, 3, ...};//可以省略行数，但不能省略列数

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

#### **字符数组**

char str[5] = {'a', 'b', 'c', 'd', '\0'};//最后一位要放'\0'char str[5] = "abcd";//最多存放5-1个char str[] = "abcdef";

* 1
* 2
* 3

### **结构体和联合体**

#### **结构体**

* 结构体使用struct定义
* 结构体成员的默认访问权限为public
* 结构体可以有数据成员、成员函数、构造函数和析构函数
* 结构体支持访问权限控制、继承和多态

#### **联合体**

* 联合体使用union定义
* 联合体的全部数据成员共享同一组内存单元
* 联合体的数据成员同一时刻最多有一个是有效的
* 联合体可以有数据成员、成员函数、构造函数、析构函数和访问权限控制
* 联合体不支持继承，因此不支持多态
* 联合体中的对象成员不能有自定义的构造函数、析构函数和复制赋值运算符
* 联合体中的对象中的对象也要满足上一条限制

#### **结构体成员初始化**

如果结构体的全部数据成员都是public类型的，并且没有自定义的构造函数、基类和虚函数，则可以使用如下方式直接初始化：

struct A {

int i;

char ch;

...

};

int main(){

A a = {100, 'c', ...}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

满足上述条件的类对象也可以使用如上方式进行初始化。

### **模板**

#### **函数模板**

* 所有模板的定义都是用关键字template标识的
* 模板参数表中的多个参数用逗号分隔
* 模板参数表可以是class或typename关键字加参数名

/\*函数模板的定义形式\*/template<模板参数表>

返回类型 函数名 (形参表){

//函数体

}

* 1
* 2
* 3
* 4
* 5
* 函数模板本身在编译时****不会****产生任何目标代码，****只有****函数模板生成的实例才会生成目标代码
* 被多个源文件引用的函数模板，应当****连同函数体一起放在头文件中****，而不能只将声明放在头文件中
* 函数指针****只能****指向函数模板的实例，而****不能****指向模板本身

#### **类模板**

* 模板类声明自身不是类，它说明了类的一个家族
* 只有在被其他代码引用时，模板才根据引用的需要生成具体的类

/\*类模板的定义形式\*/template<模板参数表>class 类名 {

//类成员

};

/\*在类模板意外定义其成员函数\*/template<模板参数表>

返回类型 类名<模板参数标识符列表>::函数名(参数表){

//函数体

}

/\*使用类模板建立对象\*/

模板类名<模板参数表> 对象名;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

### **虚函数与运行时多态**

#### **虚函数**

* 虚函数是动态绑定的基础，只有虚函数才能实现多态
* 只有类成员才能是虚函数
* 虚函数****必须****是****非静态****的成员函数
* virtual关键字****只能****出现在函数原型声明中，而****不能****出现在函数定义中
* ****只有****通过基类的指针会引用调用虚函数时，才会发生动态绑定
* 派生类在覆写基类成员函数时，使不使用virtual关键字****都一样****
* 虚函数在其类的所有直接和间接派生类中任然是虚函数
* 虚函数声明为内联函数后****仍可以****动态绑定
* 虚函数一般不声明为内联函数，因为对内联函数的处理是静态的
* 虚函数的参数默认值是静态绑定的，它****只能****来自基类的定义
* 构造函数****不能****是虚函数（构造函数不能被继承，声明为虚函数没有意义，会报错）
* 析构函数应该是虚函数，除非不作为基类（避免内存泄露）

如果没有将基类的析构函数设为虚函数，在通过基类指针删除派生类对象时调用的是基类的析构函数，派生类的析构函数没有执行，因此派生类对象中动态分配的内存空间没有被释放，造成了内存泄露。

#### **运行时多态的条件**

* 类之间满足兼容规则
* 使用虚函数
* 通过指针或引用来访问虚函数（直接通过对象名访问虚函数不能做到动态绑定）

//声明虚函数virtual 返回类型 函数名(形参表);

* 1
* 2

#include<iostream>using namespace std;

class Base0 {public:

virtual void fun();

};

void Base0::fun(){

cout<<"Base0"<<endl;

}

class Base1:public Base0{public:

void fun();

};

void Base1::fun(){

cout<<"Base1"<<endl;

}

class Drived:public Base1{public:

void fun();

};

void Drived::fun(){

cout<<"Drived"<<endl;

}

void ref(Base0 & r){

r.fun();

}

void ptr(Base0 \* p){

p->fun();

}

void normal(Base0 b){

b.fun();

}

int main(){

Base0 b0; Base1 b1; Drived d;

//使用基类引用可以做到动态绑定

ref(b0); ref(b1); ref(d);

/\*\*输出

\* Base0

\* Base1

\* Drived

\*/

//使用基类指针访问虚函数可以做到动态绑定

ptr(&b0); ptr(&b1); ptr(&d);

/\*\*输出

\* Base0

\* Base1

\* Drived

\*/

//使用对象名访问虚函数不能做到动态绑定

normal(b0); normal(b1); normal(d);

/\*\*输出

\* Base0

\* Base0

\* Base0

\*/

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68

#### **纯虚函数和抽象类**

* 抽象类是带有纯虚函数的类
* 抽象类含有没有实现的函数，是不完整的类，所以****不能实例化****
* 声明为纯虚函数后，基类中不需要给出函数的实现部分
* 基类可以给出纯虚函数的实现，但是****仍然不能****实例化
* 基类可以给出纯虚函数的实现，但是派生类中****仍然必须****实现该函数后才能实例化
* 如果将析构函数声明为纯虚函数，必须给出其实现
* 如果要访问在基类中给出的纯虚函数的实现，需要使用基类名::函数名(参数表)

//声明纯虚函数virtual 返回类型 函数名(形参表) = 0;

* 1
* 2

//如果要访问在基类中给出的纯虚函数的实现，需要使用`基类名::函数名(参数表)`#include<iostream>using namespace std;

class Base {public:

virtual void vfun() = 0;

virtual void fun1(){

vfun();//访问到的是派生类中的实现

}

virtual void fun2(){

Base::vfun();//访问到的是基类中的实现

}

};

void Base::vfun(){

cout<<"Base"<<endl;

}

class Drived:public Base{public:

void vfun(){

cout<<"Drived"<<endl;

}

};

int main(){

Drived d;

d.vfun();//Drived

d.fun1();//Drived

d.fun2();//Base

return 0;

}