overwrite,overload, override

- Overloading occurs when two or more methods **in one class** have the same method name but different parameters.（不同的参数）
- Overriding or overwrite means having two methods with the **same method name and parameters**（存在于子类和父类之间）
- override: 多态用virtual，父类和子类的某个函数同名，参数表相同，子类不一定要用virtual
- overwrite: 不构成多态关系的时候，子类继承父类，函数名相等就会被覆盖

resolveer

- "::"叫resolver，用来限定作用范围

```
1.<class name>:: <function name>

2.void A::f(){
::f();          //只写四个点，表示全局的函数，不加则默认为当前类的f函数，为递归
::a;
}
```

默认参数

- 必须写在最右边
- 缺省参数只能写在声明（一般在.h）中
- 缺省参数不能重复赋值（只能出现一次）

内存分配

heap: 动态分配的内存

stack: 本地变量

global data: 全局变量， static 变量

- The compiler allocates all the storage for a scope at the opening brace of that scope. （只要一进入函数，本地变量的空间就已经分配好了）
- The constructor call doesn’t happen until the sequence point where the object is defined

格式控制

- 设置一次格式后面就会一直保存，要用unsetf()或resetiosflags()取消格式

- 注意是ios
- 使用控制符或流成员函数： `cout<<oct;` ， `cout.width(3)`
- `#include<iomanip>`
- float和double输出的默认精度为6位

控制符	作用
dec	设置数值的基数为10
hex	设置数值的基数为16
oct	设置数值的基数为8
setfill(c)	设置填充字符c，c可以是字符常量或字符变量
setprecision(n)	设置浮点数的精度为n位。在以一般十进制小数形式输出时，n代表有效数字。在以fixed(固定小数位数)形式和 scientific(指数)形式输出时，n为小数位数
setw(n)	设置字段宽度为n位
setiosflags(ios::fixed)	设置浮点数以固定的小数位数显示
setiosftags(ios::scientific)	设置浮点数以科学记数法(即指数形式)显示
setiosflags(ios::left)	输出数据左对齐
setiosflags(ios::right)	输出数据右对齐
setiosflags(ios::skipws)	忽略前导的空格
setiosflags(ios::uppercase)	数据以十六进制形式输出时字母以大写表示
setiosflags(ios::lowercase)	数据以十六进制形式输出时字母以小写表示
setiosflags (ios::showpos)	输出正数时给出“+”号

```
![[格式控制2.jpg]]
```

```
int main(){
int b=10;

double a=16;

cout<<setfill('0')<<setw(5)<<b<<endl;

cout<<setprecision(1)<<a<<endl;
```

```
//固定小数位数，首先设置固定小数位数显示，然后设置小数位数，此时setprecision是设置小数位数

cout<<dec<<setiosflags(ios::fixed)<<setprecision(1)<<a<<endl;

//设置回默认精度

cout<<setprecision(6)<<resetiosflags(ios::fixed)<<a<<endl;
}
```

输出：

```
00010
2e+001
16.0
16


cout.unsetf(ios::hex);
```

oop

三大特性： 多态，封装，继承

初始化与赋值

```
class A{

public:

    int a;

    A(int aa):a(aa){}

};

class B{

public:

    int b;

    B(int bb):b(bb){};

    B(const A& obj){b=obj.a; cout<<"construct"<<endl;}

    B(const B& obj){this->b=obj.b; cout<<"copy constructor"<<endl;}
```

```
    B operator=(const A&obj){
        this->b=obj.a;
        cout<<"operator="<<endl;
        return *this;
    }
};

int main(){
    A a(3);
    B b(4);
    B c=a;//初始化：直接找有没有对应的构造函数
    b=a;  //赋值，调用重载过的运算符函数，返回一个B对象，再用拷贝构造函数用B(*this)
}

打印结果:
construct

operator=
copy constructor
```

New和delete

- 相比于malloc，除了分配内存还会调用构造函数来做初始化
- 会记录分配的内存大小和地址
- 如果delete不加方框号，只会delete一次，调用一次析构函数
- 按照构造顺序的**逆序**进行析构

内联函数（inline)

- 正常函数在运行时要压栈，有额外的开销（overhead），内联函数则在编译时将函数嵌入main中，使其没有额外开销，用空间换时间（不是真正的调用）
- 可以类比宏(macro)

- inline函数的定义实际上是声明，不存在函数体的定义，**只能定义在.h文件中**，不可以写在cpp中
- 因为只是声明，可以写很多次，被include很多次
    - 如果在使用时，inline函数体在main后面，编译器会在把inline的函数体变成static，这并不是正常的操作
    - inline函数在被使用时，**前面**应该已经要出现函数体
- **直接在类中定义**的成员函数都是inline
- 也可以在类中声明，在类后面进行inline定义
- 函数过大时，编译器会可能会拒绝inline
- 会导致代码膨胀，以及不简洁(clutter)
- inline
    - small functions
    - frequently called funtions
- not inline
    - very large functions
    - recursive function

```
inline f(int i){
    return i*2;
}
main(){
    int a=4;
    int b=f(a);
}

编译--->
main(){
    int a=4;
    int b=a+a;
}
```

NameSpace

- 可以解决不同cpp文件里同名函数冲突
- 就像用类一样
- 如果声明的两个namespace里有同名，同名的函数要显式写出所属的namespace
- 可以在不同的头文件里写一样的namespace,不会有重名冲突

```
Math.h
//定义namespace
namespace Math{
```

```
    void foo();
    class Cat{
        public:
        ...
    }
}


Math.cpp
#include"Math.h"
void Math::foo(){..}

//main.cpp
#include"math.h"
int main(){
    using namespace Math;
        foo();
    //也可以:
    using Math::foo;
    foo();
}

/*可以定义简单的namespace名字*/
namespace short=Math;
short::foo()

 /*可以组合*/
namespace mine{
    using namespace first;
    using namespace second;
    using orig::cat;//use cat class from orig
    void f();
}
```

Const

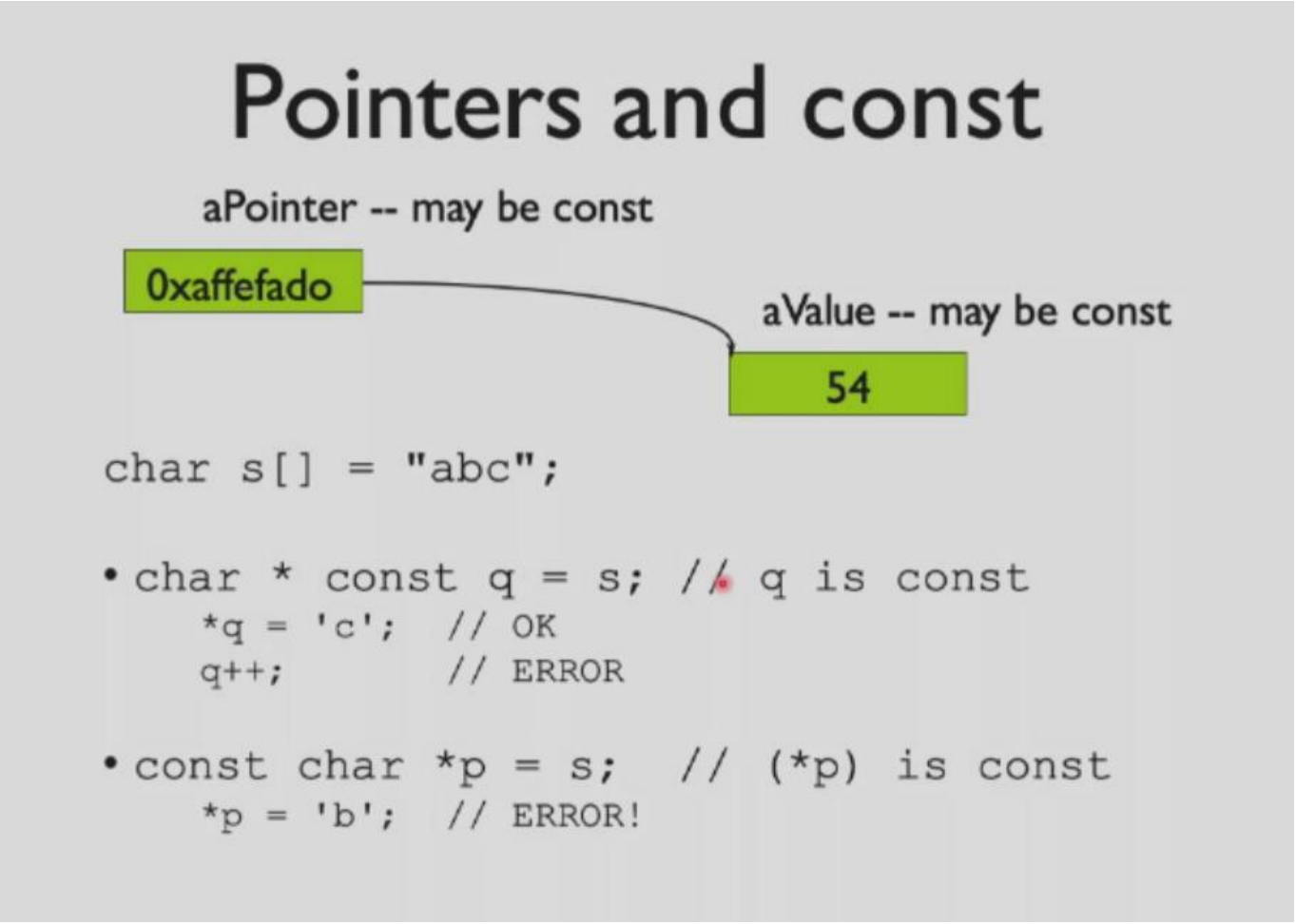
- const对象必须初始化
- const对象被设定为仅在单个文件内有效
    - **编译器**会避免为const变量分配内存
    - const变量的值被存在symbol table里，所以**可以被放在头文件里**
    - 在定义和声明处都加extern就会让它强制分配内存，其他文件也可以用

Run-time constants

- **const value can be exploited**

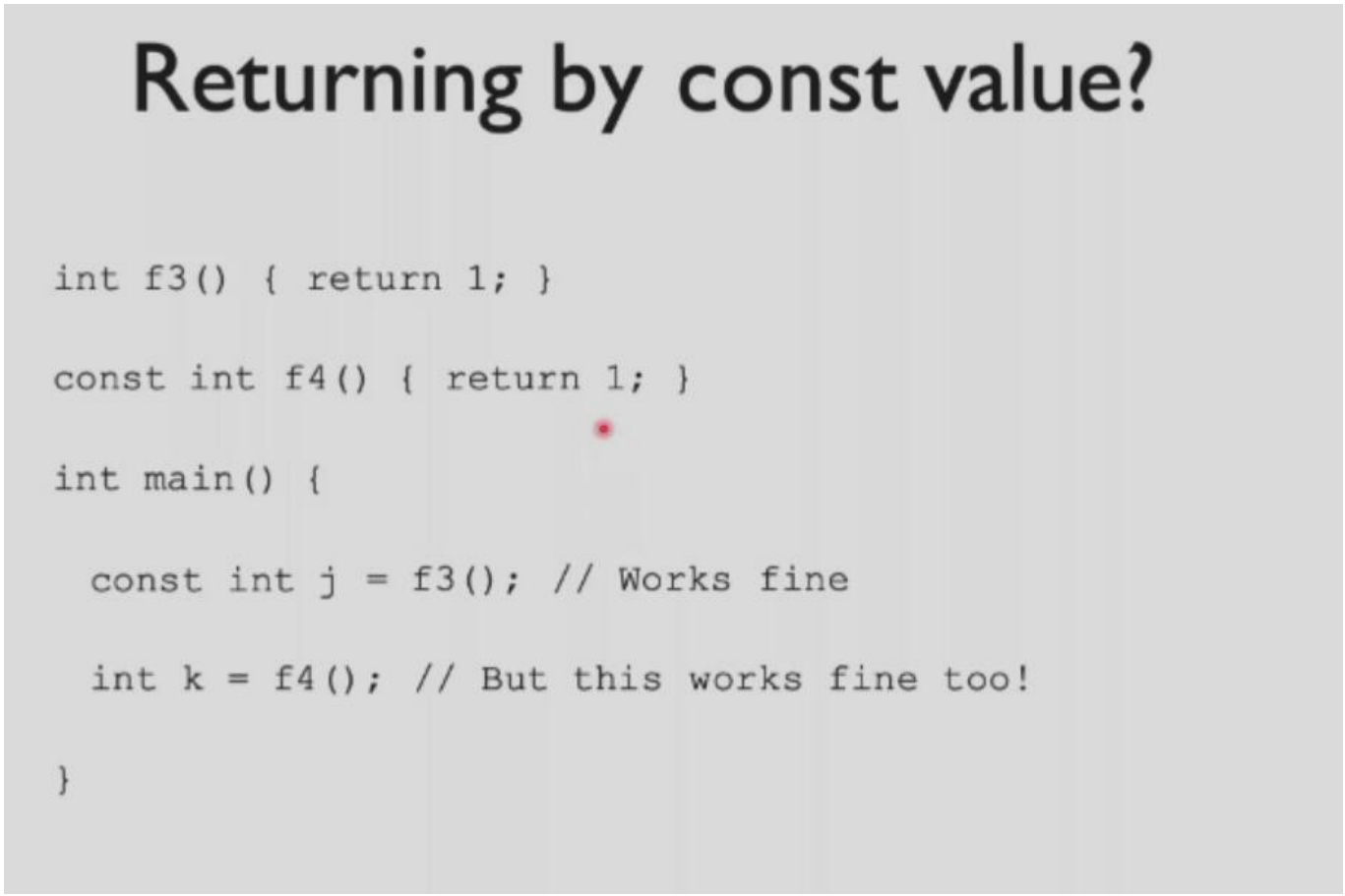
```
const int class_size = 12;
int finalGrade[class_size]; // ok

int x;
cin >> x;
const int size = x;
double classAverage[size]; // error!
```

- 上图下面例子错误的原因：编译器要知道数组长度才可以



- 要看const修饰的是什么
- 不可以用变量指针指向const，否则就意味着const可以被修改

```
const int ci=3;
int * ip=&ci;//error
```



- 这里返回的是值，不是引用，所以可行

```
int f(int &a);
int fc(const int &a);
int i=3;
f(3*i);  //wrong
fc(3*i);//ok  传入的参数会被转化成一个const参数
=const int tmp=i*3;
fc(tmp);
```

- const对象：
  - 要在成员函数声明中重复const关键字
  - **const对象只能使用声明为const的成员函数**
  - 非const对象可以使用声明为const的成员函数
  - 一般如果不用更改成员变量，都可以声明为const
  - 不可以在const成员函数里调用非const成员函数

```
class A{
    int day;
```

```
    int get_day() const{ return day;}
    /*上面这行等于int get_day (const A*){return day;}
    也就是把this指针指向的内容看作const
    */
    int get_day() {return day;}
}

int main(){
    const A a;
    A b;
    a.get_day();//如果没有定义const类型的成员函数会报错
    b.get_day();//只定义了const类型的成员函数也可以

}
```

Static

- 全局变量+static / 函数+static: 只能在该文件内使用(C++内不再使用)
- 对于全局，加static影响了其被访问的**范围**
- 对于局部，加static影响了其存在的**时间**
- 静态成员变量和静态本地变量一样，static在被执行到的那句语句时才做初始化分配内存，且后续不会重新改变值（实际上是全局的）

C++中的static：

- 静态本地变量 （persisitent storage)
- 静态成员变量 (shared by all instances)
- 静态成员函数：只能访问静态成员变量（因为被整个类共享）
- 静态对象
    - 被销毁时遵守LIFO

静态成员变量

- 实例成员的存储空间属于具体的实例，不同实例（对象）的同名成员拥有不同的存储空间；静态成员的存储空间是**固定**的，**不与具体的实例（对象）绑定**，被该类的所有实例**共享**

- 只初始化一次，一般在类外初始化
- 静态成员在类内只是**声明**，**必须要在类外定义一次全局变量**

静态成员函数

- 只能访问静态成员变量
- 静态成员函数不包括this指针
- 使用静态成员函数可以**不实例化对象**，使用classname::func来访问
- 不可以**dynamically overridden**

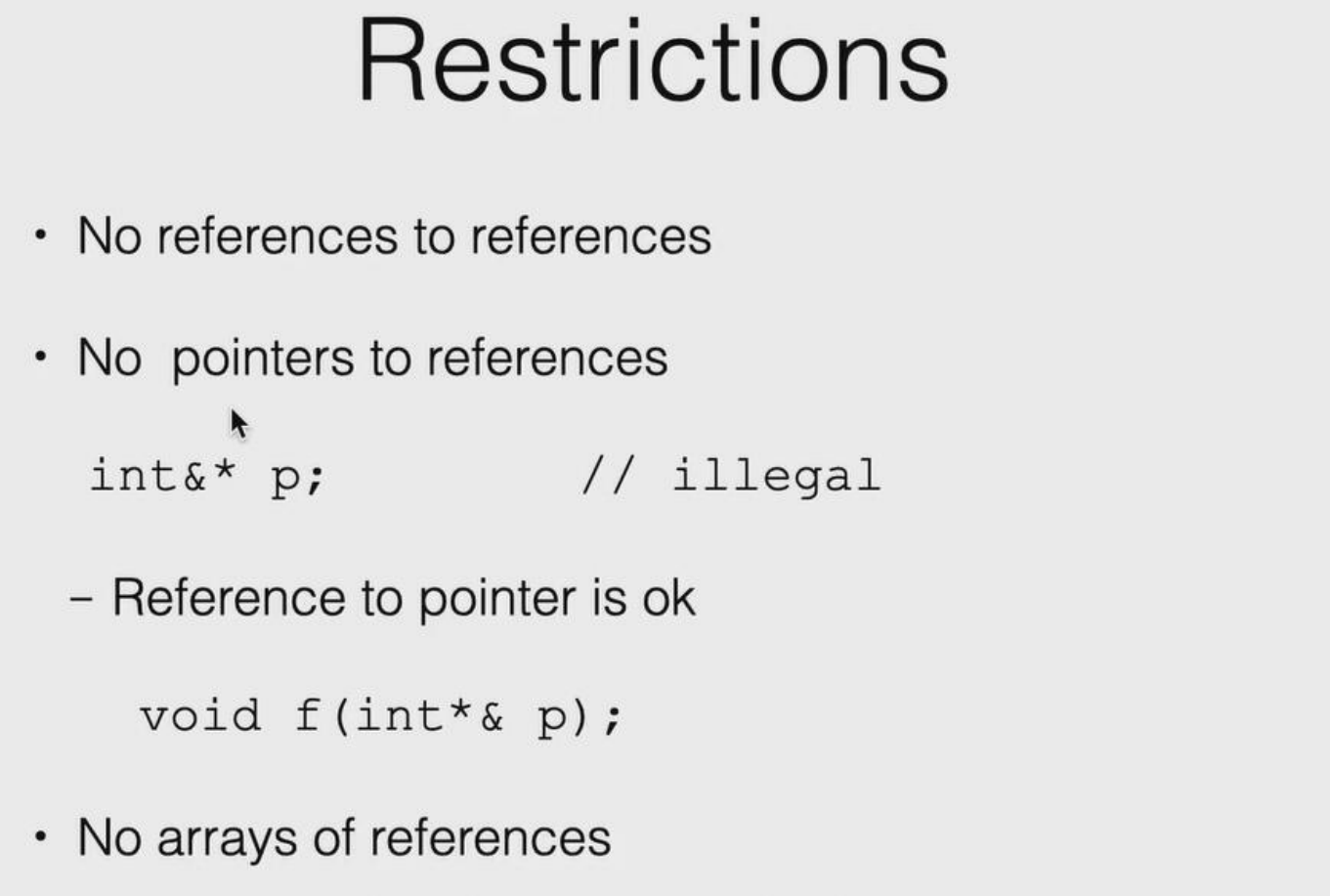
```
class B;//声明

class A{
public:
    static void rate(double newrate);
private:
    double amount=100;
    static int m;

   /*注意不能return amount，因为amount和实例捆绑，但静态成员函数不与实例捆绑*/
    static int initRate(){return period;};
    static int interestRate;
    static constexpr int period=30;
    double str[period]; //可用static做函数等的默认参数
    static B ptr;//静态成员可以是不完全类型，因为所需要的内存在外面申请
};
//static只在类内写，外部定义不写
void A::rate(double newrate){
    interestRate=newrate;
}
//初始化
int A::interestRate=initRate();
int m;//不写static
```

引用

- 指向同一个对象，对a操作即对b指向绑定对象进行操作，就像一个别名，指向的地址就是绑定对象的地址，用sizeof也是绑定对象的大小
- 相比于指针，必须初始化则更方便，且在使用时代码更简洁
- 在定义的时候就必须和**对象**绑定
- 不能重复绑定（绑定的一直是同一个值，后续的所有操作都是对绑定值的赋值等操作），可以把引用就当做一开始绑定的那个变量
- 不可以为NULL

- 类型 & 引用名字=对象

```
int i=10;
int &r=i;
int &k=r;  //ok

int &k=10;//false
```



- 赋值：即改变a指向的对象
- 指针：对象的地址
- 用"::"解析符告知编译器name来自哪里，如果不加，默认是函数内部

```
void student::f(){
        ::f();//would be recursive otherwise
        ::a++;//select the global a
        a--;//The a at class scope
}
```

- 用const type&在函数中传递参数，可以避免传入整个对象的值影响效率，并避免使用指针改变对象内部的值
- 默认传入为const

```
void f(const int &a){
    cout<<i<<endl;
}
int main(){
    int i=3;
    f(i*3);             //打印9
}
/*不会报错，i*3会临时生成一个const int类型的临时变量并传入函数，但如果改为f(int &a)，就会报错*/
```

- 类中的引用类型成员变量
    - 要用初始化列表进行初始化

```
class a{
public:
    A(int &i):data(i){}
private:
    int &data;
};
```

- 引用作为函数参数时，**不能传入表达式**，否则用const修饰

异常

```
//限定了异常类型
void func()throw(){
}
//可能抛出任何异常
void func(){
        if(..)
                throw tmp;
}
```

链式检测

- 异常发生：
  - 如果不在try里：跳出到上一级函数
  - 在try里：
    - 有对应catch捕捉，执行
    - 无对应catch捕捉：跳出函数，再次回到循环检测的开始

throw

```
throw exp; //抛出值

throw;
//只能放在catch里，代表重新抛出之前接受到的异常
//reraises the exception being handled
//valid only within a handler
```

Try

- 不是必要的

```
try{

}catch(Type v){

}catch(...){

}
```

Catch

- 接收一个参数，或者只用...代替
- 可以re-raise exceptions
- 参数可以是：`Type v`或`...`，后者代表任意类型的异常
- 找第一个匹配的catch，一旦匹配到就不会在看后面的catch

异常的派生

- 抛出子类的异常，参数为父类的catch也可以捕捉到

标准异常

- new异常会抛出一个bad_alloc（标准库异常的一个组成）

```
catch (bad_alloc&c)
```

```
class Person {
public:
        Person(int age) {
                if (age < 0 || age >100) {
                        throw out_of_range("年龄越界.");
                }
                this->m_age = age;
        }
        // 类成员没有指针类型 我就没写拷贝构造和等号运算符
private:
        int m_age;
};

int main(){

        try {
                Person(101);
        }
        catch(exception &e){        // 一般都用父类接  不会直接用out_of_range 且最好用引用接收匿名对象out_of_range
                cout << e.what() << endl;
        }

        return 0;
}
```

细节

- 如果函数抛出了非声明异常类型的异常，编译器会插入一段代码，内部会再抛出异常把程序终止
- 但编译器并不管是否有程序对抛出的异常进行处理
- 析构函数应该从不抛出异常。如果析构函数中需要执行可能会抛出异常的代码，那么就应该在析构函数内部将这个异常进行处理，而不是将异常抛出去。
    - 原因：在为某个异常进行栈展开时，析构函数如果又抛出自己的未经处理的另一个异常，将会导致调用标准库 terminate 函数。而默认的terminate 函数将调用 abort 函数，强制从整个程序非正常退出。
- 构造函数中可以抛出异常。但是要注意到：如果构造函数因为异常而退出，那么该类的析构函数就得不到执行。所以要手动销毁在异常抛出前已经构造的部分。

- catch的参数如果是对象，会调用拷贝构造函数;对于指针和引用，不太安全，一般实现的机制是，throw后栈空间的内容不会改变，所以可以继续用，但指针要记得delete

类

构造与析构

- 创建对象时，先分配内存，再调用构造函数进行初始化
- 销毁对象时，默认的析构函数只会删除栈上的空间，也就是对象自己的空间，而不会回收堆上（new出来的）空间
    - 要在析构函数里写**delete**动态分配内存的句子

构造：

- **顺序**：初始化顺序是成员变量声明的顺序，destroy相反
- 默认构造函数：
    - 不带参数，或者**为所有的形参提供默认实参**
    - 写在初始化列表里的成员变量必须有默认构造函数
- 不能有返回值
- 对象创建时自动调用，不能被主动调用
- 两个阶段：初始化阶段，计算阶段
- 拷贝构造
    - 若类中有指针，可能会共享数据
    - 是成员变量间的拷贝，若有类，则会递归下去

```
class CPU{
public:
    //含参构造函数
    CPU(CPU_Rank rank1,int fre1,double vol1){
        rank=rank1;
        frequency=fre1;
        voltage=vol1;
    }
    //默认构造函数
    CPU(){
        rank=P1;
        frequency=2;
        voltage=100;
```

```
    }
   //拷贝构造函数
    CPU(const CPU &p){
        rank=p.rank;
        frequency=p.frequency;
        voltage=p.voltage;

    }
    //析构函数
    ~CPU(){
        cout<<"destruct a CPU!"<<endl;
    }
}

int main(){
    CPU a(P6,3,300);
    CPU b;
    CPU c(a);
}
```

拷贝构造

- 拷贝构造函数要传递引用而不是值，如果传入值，值在创造临时的参数时又会调用拷贝构造函数
- 在没有写默认构造函数和拷贝构造函数时，编译器会自动生成一个（浅拷贝）
- 拷贝构造发生的情况：
  - 用对象来初始化的时候
  - 函数参数表是对象的时候
  - 函数返回的时候，`A a=f()`如果f()返回的是一个对象，那么此处调用两次拷贝构造；但这和编译器高度相关，可能会被优化

浅拷贝与深拷贝

- 浅拷贝：单纯的**按字节复制**的拷贝（值拷贝），对于动态分配内存的变量会产生问题
- 深拷贝：创建内存空间
- 浅拷贝例子：默认拷贝构造函数，**会不停地调用各成员变量的拷贝构造函数**

例子：

```
/*因为没有定义拷贝构造函数，此处使用了默认的拷贝构造函数，在堆区的内容不会被拷贝到新的对象里。

新对象的name和原对象的name拥有相同的值（字符串地址），因此更改旧对象的name，新对象也会改
```

变
但name的地址是不同的，因为不同对象拥有不同的内存空间，即：

对象1：
0x61fe10:0xef1730 //name
对象2：
0x61fe00:0xef1730 //name

使用string不会有这种情况，因为会调用string的构造函数
*/

```
class A{
public:
    char *name;
    int in;
    A(const char* s,const int inn){name=new char[strlen(s)+1];
        strcpy(name,s);
        in=inn;
    }
    A(){cout<<"a's defalut constructor"<<endl;}
    void print(){cout<<name<<" "<<in<<endl;}

};

int main()
{

    A obj("123",1);
    obj.print();
    A obj2=obj;
    obj2.print();

    cout<<(void*)obj.name<<endl;
    cout<<(void*)obj2.name<<endl;


    char s[]="12";
    strcpy(obj.name,s);
    obj.print();
    obj2.print();

}
```

输出：
123 1
123 1
0x61fe10
0x61fe00

```
0xef1730
0xef1730
12 1
12 1



A &f(A& r){
    return r;

}

A f(A&r){
    return r;
}
```

访问限制

public/private/protected

- 结构体 vs class：结构体包含的函数默认为public，而class中默认是private
- public和private的区分在编译过程中进行
- public
- private:（数据）
    - 只有成员函数可以访问；
    - private是对**类**而言的，不是对对象。同一个类的不同对象可以访问对方的私有成员变量

```
struct X{
private:
    int i;
public:
    friend void g(X*,int);
    friend void struct z;
}

struct z{
private:
    int j;
}
```

```
/*这些函数可以用X中的私有成员*/
void g(X* x,int i)
    x->i=i;
}
```

- protected（留给子类的接口）
    - 只有自己和子类可以访问
    - **注意子类不可以访问父类的private成员**，必须用protected

friend

被声明的函数或类可以访问对应的私有成员

- 对函数：`friend void func()`
- 对类：`friend class class_name;`
- 只能一开始在类定义里就已经声明

初始化列表

- 区分初始化和赋值，初始化列表在函数体外面进行了**内存的分配和初始化**，如果在函数内进行，则为**赋值**；**由于一些操作必须在定义时就初始化，因此全部成员变量都使用初始化列表是安全的做法**
- 初始化顺序：按**类中成员定义**的顺序进行
- 类中成员必须使用初始化列表的情况：
    - 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
    - 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
    - 没有**默认构造函数**的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化

```
class A{
    int i;
    int *p;
public:
    A():p(0){cout<<"A::A()"<<endl;}
};

class P{
private:
    const float x,y;
    P(float xa=0,float ya=0):y(ya),x(xa){}
};
```

组合

- 一个对象中包含另一个对象(embeded objects)
- 在构建对象的时候，如果没有使用初始化列表，内置对象的默认构造函数会被调用

继承

```
class A{
public:

};

class B: public A{

};

int main(){
    B b; //可以调用A的构造函数
        b.set(10); //可以调用A里的函数
}
```

- 子类**拥有**父类的所有东西，但不能**直接接触**父类私有的东西
- 子类不能访问父类private,要用**protected**
- 创建子类对象时，必须要将基类**初始化**
- 构造顺序：基类->子类 析构顺序：子类->基类
    - **基类一定会先被构造**
- 使用基类函数记得加作用域
- **重写(over write)：派生类存在重新定义的函数**
- 当子类函数和父类函数重名，父类函数被隐藏，只能使用**子类函数**，**除非加作用域**或者写`using 类名::函数名`，后面接函数
    - 其他编程语言中子类同名函数和父类构成overwrite关系（替代），C++里没有任何关系，因此编译器必须要将父类的隐藏(name hiding)，否则就会混乱

向上造型(upcasting)

- 把子类(derived)的引用或指针转换成父类(base class)的引用或指针
- 此时子类的东西被丢失，使用的是基类的函数和成员变量（但是地址还是那个地址）
- 这里更像是：把子类当做基类“看”，而不是让子类拥有特性（多态）
- 使用virtual后，基类指针、基类引用都可以是动态绑定，使用的是子类的函数和成员变量

```
class A{
        int i;
}
class B:public A{
        int j;
}
B b;
int *p=(int*)&b;
//则p指向b.i,++P指向b.j

//upcast
Manager pete("pete","16871","baa");
Employee *ep=&pete;
Employee &er=pete;
```

多态(polymorphism)

- 一个类派生出不同的形态
- 动态绑定只能通过指针或引用来实现
- 纯虚函数?
- 静态与动态绑定
    - 静态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。
    - 动态绑定：绑定的是对象的动态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。

重点!

- 对于一个`void f(A*p)`函数，有两种形态：静态和动态
    - 访问成员变量时是静态，也就是只会访问到A类型的成员变量
    - 访问函数时，检测是不是virtual，如果不是就用A的，如果是就用虚函数表里的
- vtable 存储了所有的**虚函数地址**
    - 继承的类里没有被覆盖的虚函数
    - 本类重写了的父类的虚函数
    - 自己新加的虚函数

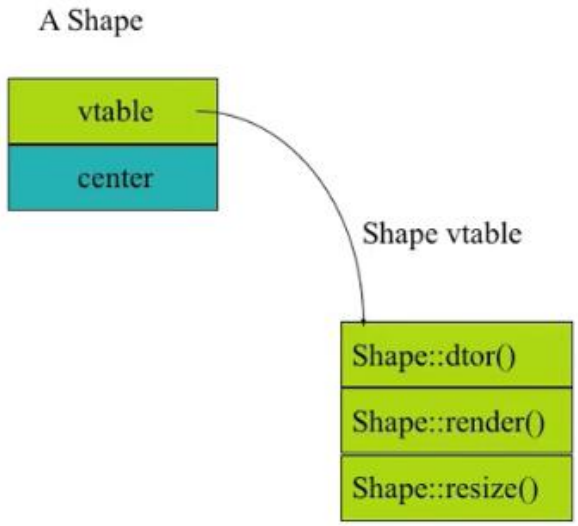
实现

- `virtual void f()`：虚函数
- 基于upcast和动态绑定
- 虚函数在类中相比于其他函数所占空间更大

- 每一个类的内存空间分为：**虚函数地址**，函数，成员变量，虚函数地址指向虚函数表(虚函数地址为8B)
- 虚函数表是**类**的
- virtual实现动态绑定：因为首地址存的是vtablle

How virtuals work in C++

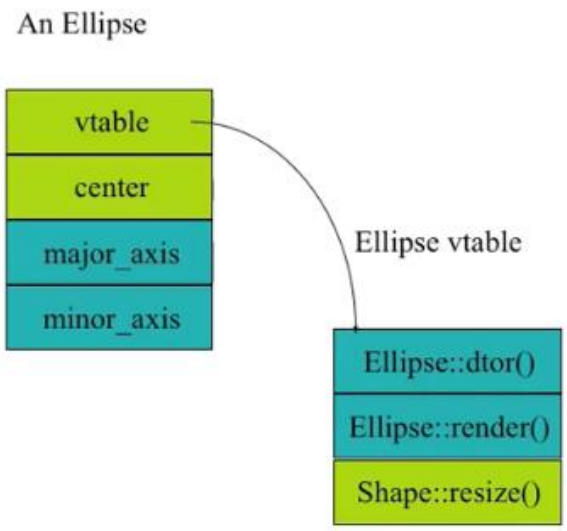
```
class Shape {
public:
   Shape();
   virtual ~Shape();
   virtual void render();
   void move(const
   XYPos&);
   virtual void resize();
protected:
   XYPos center;
};
```



see: virtual.cpp

Ellipse

```
class Ellipse :
              public Shape
   {
public:
   Ellipse(float majr,
           float minr);
   virtual void render();
protected:
   float major_axis;
   float minor_axis;
};
```




```
class A{
        virtual f(){this->g();}
        virtual g(){}
};


class B{
        virtual g(){}
};

B b;
A *p=&b;
p->f();
```

里面会调用B的g，因为现在地址指向的是B的vtable
可以把所有有virtual的都当做动态绑定算了

构造

- 构造函数中，virtual不起作用
- 因为构造的时候，填入的vtable是自己的，在继承的时候先初始化父类，父类构造的时候填的是自己的vtable

```
class A{
        A(){f();}
        void f();
}

class B:public A{
        f()

}
```

析构

- 析构要virtual，如果不是virtual，析构的时候只会析构静态绑定的那个值，如此处只会用A的析构，而不会用B的析构

```
class A{

public:

     virtual ~A(){cout<<"a"<<endl;}

};

class B:public A{

public:

    virtual ~B(){cout<<"b"<<endl;}
};

int main(){

    A *b=new B;

    delete b;
}
```

赋值

- 子类赋值给父类`a=b`，父类a只会得到b中属于a的成员变量值，虚函数表不会被改变（因为vtable在构造时创建）
- 用指针、引用则情况不同，此处的指针将指向circle的vtable

```
Ellipse *elly=new Ellipse(20F,40F); //base
Circle* circ=new Circle(60F); //derived
elly=circ;
elly->render();//Circle::render()

void func(Ellipse& elly){
        elly.render();
}
Circle circ(60F);
func(circ);//使用的是circle的render
```

return types relaxation

- **virtual函数要求返回的类型相同，参数表相同，名称相同**
- 不相同的不会override
- virtual函数子类可以返回子类的指针、引用，不能返回类型（多态只建立在相同返回类型的函数，此处做了个松弛，指针和引用也可以，但对象不行）

```
class A{
public:
        virtual A* f();
        virtual A& f2();
        virtual A f3();

};

class B: public A{
public:
        virtual B* f();//OK
        virtual B& f2();//OK
        virtual B f3();//错误
}
```

Abstract class

- 只要父类的一个virtual函数等于0（纯虚函数pure virtual），那么这个类就是抽象类
- 抽象类不可以制造对象，但可以调用函数
- protocal/interface classes

- all **non static member functions** are pure virtual except destructor（所有非静态成员函数和非析构函数都是纯虚函数）
- virtual destructor with empty body
- no non-static member variables(may contain static members)

MI：多继承

- 为了解决没有单根结构
- 模板是更好的方案

类的其他属性

可变数据成员

- 非静态成员函数后面加const（加到非成员函数或静态成员后面会产生编译错误），表示成员函数隐含传入的this指针为const指针，决定了在该成员函数中，任意修改它所在的类的成员的操作都是不允许的（因为隐含了对this指针的const引用）
- 唯一的例外是对于**mutable**修饰的成员
- 加了const的成员函数可以被非const对象和const对象调用，但不加const的成员函数只能被非const对象调用

```
class S{
public:
    void some()const;
private:
    mutable size_t access_ctr;
}
void S::some() const{
    ++access_ctr;
}
```

返回*this的成员函数

- 返回值为**引用**，则返回原有对象，否则返回的是**副本**
- **const成员函数**以引用形式返回*this，那么它的返回类型将是常量引用
- 当成员函数重载，const对象会调用const版本函数，非常量对象调用非常量版本函数

类的声明

- **不完全类型**：只声明，未定义

运算符重载(overload)

运算符

- 可以被overloaded的运算符：[], (), ->, ","等
- 不能重载的运算符：. .* :: ?: 等

注意

- 当运算符重载同时定义为成员函数和全局函数时，会产生冲突，所以必须要选择一种

Restrictions

- 只能重载已有运算符
- 运算顺序不保证（短路不适用
- 必须保持原有的操作数数目和优先级
- 只能在class或enumeration上进行重载（不能重载int等）

类型

- 必须是非成员函数：**输入输出运算符**（一般声明为友元）
- 应该为非成员函数：**有对称性的操作如算术、相等性的运算符**（可以对两边进行转换）
- 应该为成员函数：单目运算符
- 必须是成员函数：= [] () ->
- 一般不建议重载**逗号，取地址，逻辑与和逻辑或**

过程

- 当为双目运算符，会传入两个参数，若为成员函数，传入的第一个参数默认为this`const Integer operator+(const Integer &a,const Integer&b)`

作为成员函数

- 第一个参数默认是this，因此会隐藏一个参数
- receiver: 运算符左边的对象，在运算时不会被改变类型
- 使用的是receiver的重载运算符 `x+y==>x.operator+(y)`
- 当运算符左右类型不同，将右边的对象变为左边对象的类型，如果无法改变，就会出错
- 必须是const，否则返回的对象可作左值(会出现x+y=6的情况)

作为全局函数

- 运算符左右两边参数都进行类型转变，调用构造函数
- 因为全局函数一般访问不了对象的私有成员变量，通常在类里面声明这个函数为**友元**

```
//[const] 类名 operator 运算符 (const 类名 & n) const{}
class In{
public:
    In(int n=0):i(n){}
    const In operator+(const Integer& n) const{
        return Integer(i+n.i);
    }
private:
    int i;
};

In a(1),b(2),c(3); //或a=1,b=2...
c=a+b;
```

- 不改变算子的运算符要加const
- 返回值：返回新对象应加const
- ++和--

```
/*表示++a，返回的是原有的自增了的对象*/
注意这里返回的值不可以做左值，所以加const
返回*this，节省空间可以传引用
const Integer& Integer::operator++(){
    *this+=1;
    return *this;
}
返回的是局部变量，不能返回引用
/*a++，用int来表示后缀*/
const Integer Integer::operator++(int){
        Integer old(*this);//拷贝构造
        ++（*this);//原对象自增
        return old;//返回没有自增对象
}
```

模板+运算符重载

- 如果成员都是基本类型，不用写重载，直接用默认重载

- 在用"="时，默认是以成员单位的复制，即分别调用成员的构造函数
- 对于有指针的类，必须写重载

```
template<class T>

class A{

public:
    T *s;
    T a;
    A(T aa):a(aa){s=new T[2];}

    A& operator=(const A& rhs);//内部不要写A::operator

    void print(){cout<<a<<" "<<s[0]<<" "<<s[1]<<endl;}

};



template<class T>

A<T> & A<T>::operator=(const A<T>& rhs){

    if(&rhs!=this){//先判断是不是在同一块内存！因为要先delete

        delete []this->s;

        this->a=rhs.a;

        this->s=new T[2];

        memcpy(this->s,rhs.s,sizeof(T)*2);

    }

    return *this;
}
```

重载类型转换符

```
class R{

    int numerator;
```

```
    int denomelator;
public:
    R(int n,int d):numerator(n),denomelator(d){}
    operator double() const;
};
R::operator double()const{
    return (double)numerator/(double)denomelator;
}

int main(){
    R r(1,3);
    double d=1.0*r;
}
```

Stream inserter and extractor

- `cout <<` <<即为inserter（把东西输入到流）， >>即为extractor（把东西从流里解析出来）

模板

- Function Template
- Class Template
    - template member function

https://blog.csdn.net/hsxyfzh/article/details/95797029

- `template <class T1,class T2...>` 模板声明, 下面的内容都是模板

函数模板

- 在编译的时候，发生模板的实例化，创建成对应类型的函数
- 参数类型要完全匹配，类型无法进行隐式转换

- 可以显式写出类型

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

类模板

- 注意有些地方需要用到运算符重载
- `Name<T1,T2..>` 此时模板类的名字变为类名+参数类型，因此在写成员函数时，要写成：

```
template <class T> //声明
void Vec<T>::push_back(const T &x);  //类型 模板类名::函数名(参数表)
```

non-type parameters

- template arguments can be constant expressions
- non-type parameters-can have a default argument
- 此处bounds在编译时会变成字面量
- can make code faster
- make use more complicated
- can lead to code bloat（代码膨胀）

```
template <class T, int bounds=100>
class A{
        T elements[bounds];
}

template<class T, int bounds>
T& A<T,bounds>::operator[]()..

A<int,50>v1;
A<int> v2;
```

类模板的静态成员

- 需要在类外初始化
- 应该不能在类内定义类型为T的成员

```
template <class T>
```

```
class A{

    static int num;

public:

    void f(){cout<<num;}

};

template <class T>

int A<T>::num=10;
```

模板和继承

- 模板子类可以继承非模板父类
- 模板子类可以继承模板父类
- 非模板子类可以继承模板父类，但要写明父类的类型

注意点

- 模板所有东西都要放头文件，因为模板的所有东西是**声明**，放在cpp无法被识别、生成对应的模板函数
- 运行顺序：普通函数 > 函数（全）特化模板 > 函数模板
    - 注意只写f(1.0,2.0)，默认类型是double不是float
    - 如果和函数模板的类型匹配不上，使用普通函数进行隐式转换
    - 如果还是不行，使用重载函数