目录

第4章 Object Interactive

构造函数的调用顺序：**基类 → 成员变量 → 派生类**

析构函数的调用顺序：**派生类 → 成员变量 → 基类**

4.1 构造函数

4.1.1 构造函数的作用

(1)函数名与类名相同

(2)通过编译器调用，进行初始化，保证实例化的类一定会初始化

```
class X{
    int i;
public:
    X();
};
void f(){
    X a;//此时，等同于调用a.X()
}
```

4.1.2 构造函数的参数

```
class Tree{
    int i;
public:
    Tree(){};
    Tree(int i){...}
}
void f(){
    Tree t(12);
}
```

(1)构造函数可以有很多个

(2)如果仅定义了一个带参数的构造函数，那么缺省构造函数会丢失，在实例化时，必须添加参数

```
struct Y{
    float f;
    int i;
    Y(int a);
};
Y y2[2]={Y(1)};//会报错，因为只声明了一个数
```

(3)如果没有定义构造函数，系统会默认添加一个缺省构造函数**auto default constructor**

4.2 析构函数

4.2.1 析构函数的定义

(1)函数名为**~类名()**

(2)用于返回内存

(3)当对象超出作用域后，编译器自动调用析构函数

```
class Y{
public:
    ~Y();
};
```

(4)析构函数调用的唯一标志是**右大括号**

对象变量的作用域是一对大括号，出了作用域便会被销毁

```
class V{
    int i;
    MyClass c1;
public:
    ~V(){};
    //成员变量的析构是在本对象变量析构之后调用的
}
int main(){
    {
        V v1;
    }//在这里时，会调用~V()
    return 0;
}
```

4.3 内存分配

(1)在大括号的左括号处，编译器会分配所有的内存

(2)在定义对象的语句处，才会调用对象的构造函数

```
class X {
public:
  X(){};
};

void f(int i) {
    if(i < 10) {
    //! goto jump1; //Error:goto语句可能跳过x1的构造函数
    }
    X x1;  //在这里调用x1的构造函数
jump1:
    switch(i) {
        case 1 :
            X x2;  //在这里调用x2的构造函数
            break;
    //! case 2 : // Error: case bypasses init
            X x3;  //在这里调用x3的构造函数
            break;
    }
}
```

4.4 抽象

(1)抽象：忽略部分细节，以将注意力集中在更高级别问题上

(2)模块化：将整体划分为明确定义的部分的过程，这些部分可以单独构建和检查，并且以明确定义的方式相互作用

4.5 局部变量

(1)局部变量是在函数内部定义的，其作用域仅限于它们所属的函数

(2)如果局部变量的名字与类的成员变量相同，则会使用局部变量而非成员变量

```
int TicketMachine::refundBalance()  {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

4.5.1 字段(Fields)，参数，局部变量

1. 这三种类型的变量都能够存储适合于其定义类型的值。
2. 字段(Fields)
    1. 字段在构造函数和方法之外定义。
    2. 字段用于存储贯穿一个工程生命周期的数据。因此，它们维护对象的当前状态。它们的生命周期和工程的生命周期一样长。
    3. 字段具有类作用域：它们的可访问性扩展到整个类，因此可以在定义它们的类的任何构造函数或方法中使用它们
    4. 一旦字段被定义为私有，它们就不能从定义类之外的任何地方访问。
3. 形式参数和局部变量仅在构造函数或方法执行期间持续存在。它们的生存期仅相当于一次调用，因此在调用之间会丢失它们的值。因此，它们是临时的而不是永久的存储位置。
4. 形参
    1. 形参定义在构造函数或方法的头文件中。它们从外部接收值，由构成构造函数或方法调用一部分的实际参数值初始化。
    2. 形式参数的作用域仅限于其定义的构造函数或方法。
5. 局部变量
    1. 局部变量在构造函数或方法的函数体中定义。它们只能在其定义的构造函数或方法体中进行初始化和使用
    2. 局部变量在表达式中使用之前必须先初始化——它们没有默认值。
    3. 局部变量的作用域仅限于定义它们的块。从那个块以外的任何地方都无法进入。

4.6 初始化

4.6.1 Initializer list

```
class Point{
private:
    const float x,y;
    Point(float xa=0.0, float ya=0.0)
        :y(ya),x(xa){}
};
```

(1)可以初始化任意类型的数据

(a)相当于调用了析构函数

(b)避免了赋值操作

(2)初始化的顺序是写的顺序

(a)例如在例子中，先初始化y，后初始化x

(b)析构时，按照初始化的倒序

4.6.1 Initializer list 与 Assignment的区别

```
Student::Student(string s):name(s){}
//初始化列表
//直接构造了一个内容为s的string
Student::Student(string s){name=s;}
//赋值
//先构造了一个string,然后再将这个string赋值为s
```

4.7 函数的重载

4.7.1 Overload

相同的函数，可以有不同的参数列表

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5

print("Pancakes", 15);
print("Syrup");
print(1999.0, 10);
print(1999, 12);
print(1999L, 15);
```

4.7.2 Overload 和 auto-cast

```
void f(short i); //#1
void f(double d);//#2

//根据c++类型转换的优先级来决定调用哪一种函数
f('a');//自动转换为short,调用#1
f(2);//自动转换为short,调用#1
f(2L);//自动转换为double,调用#2
f(3.2);//自动转换为double,调用#2
```

4.8 缺省参数

```
Stash(int size, int initQuantity = 0);
```

规定：缺省参数必须从右往左写

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j);//不合法
int groucho(int k = 1, int m = 2, int n = 3);
```

4.9 常量对象变量 const objects

4.9.1 定义

```
const Currency the_raise(42, 38);
```

4.9.2 常量成员函数 const member functions

(1)不能改变对象内部的内容

(2)在函数内只能调用常量成员函数

```
int Date::get_day()const;

int Date::set_day(int d){
    day = d;
}
int Date::get_day()const{
    //!day++;//ERROR:更改了对象的内容
    //!set_day(12);//ERROR:调用了non-const成员函数
    return day;
}
```

4.9.3 常量成员变量

```
class A{
   const int i;
   //必须在初始化列表中,初始化i的值
};
```

```
class HasArray{
    const int size;
    int array[size];//ERROR:编译器无法预先分配内存
}
//修改
class HasArray{
    static const int size = 100;
    int array[size];
}
//修改2
class HassArray{
    enum{size = 100};
    int array[size];
}
```

4.10 函数变量中的class

4.10.1 传入参数

```
void f(Student i);
//创建新对象
void f(const Student* i);
//只传递指针,不会传递变量
//在前方加const保证不会修改该对象
void f(Student &i);
//只传递引用,不传递变量
//不使用指针,防止堆内存的任意访问
```

4.10.2 返回值

```
Student f();
//要创建一个临时对象
Student* f();
//要保证指针的有效性,不能返回野指针(指向临时变量)
Student& f();
//与指针类似返回的引用不能是临时变量
```

例:

```
//代码耦合性较强,需要在bar()中释放掉在foo()中分配的内存
char *foo(){
    char *p;
    p = new char[10];
    strcpy(p, "something");
    return  p;
}
void bar(){
    char *p = foo();
    printf("%s", p);
    delete  p;
}
```

修改:**谁分配,谁销毁**

```
void foo(char *p){
    strcpy(p, "something");
    return  p;
}
void bar(){
    char *p = new char[10];
    foo(p);
    printf("%s", p);
    delete  p;
}
```