创建与销毁

1. 构造函数: 对象的生

构造函数是类的特殊的成员函数，它用来确保类的每个对象都能正确地初始化。

- 构造函数没有返回值类型，函数名与类名相同
- 类的构造函数可以**重载**，即可以使用不同的函数参数进行对象初始化

例如:

```
class Student {
    long ID;
public:
    Student(long id) : ID(id) { } //可以用**初始化列表**来初始化
    Student(int year, int order) {
            ID = year * 10000 + order;
     }
    ...
};
```

在构造函数的初始化列表中，还可以调用其他构造函数，被称为"委派构造函数":

```
class Info {
public:
    Info() { Init(); }
    Info(int i) : Info() { id = i; }
    Info(char c) : Info() { gender = c; }
private:
    void Init() { .... } // 其他初始化
    int id {2016};
    char gender {'M'};
    ...
};
```

就地初始化:

•在C++11之前，对于类中的**非静态**成员变量，不能在类定义的时候进行初始化，它们的初始化操作只能通过构造函数来进行。而c++11中支持如下初始化操作。

```
class A{
private:
    int a = 1;
    double b {2.0};
};
```

2. 析构函数：对象的“死”

对象的“死”（清除和释放资源）是由编译器在对象**作用域**结束处自动生成调用析造函数代码来完成的。

- 当执行到“包含对象定义范围结束处”时，编译器自动调用对象的析构函数。
- 动态分配的内存是一种典型的需要释放的资源。清除对象占用的资源是无条件的，不需要任何选项。因此，析构函数没有参数，且只有一个（即清除方式唯一）。

```
class ClassRoom {
    int num;
    long* ID_list;
public:
    ClassRoom() : num(0), ID_list(0); { }
    ...
    ~ClassRoom() {  // 析构函数
        if (ID_list) delete[] ID_list; // 释放内存
    }
};
```

总结：创建和销毁的"How"由程序员决定，"When"由编译器决定。

3. 类中的静态成员(static)

使用**static**修饰的数据成员，称为类的**静态数据成员**，被该类的所有对象**共享**（即所有对象中的这个数据域处在同一内存位置）。

在返回值前面添加static修饰的成员函数，称为类的**静态成员函数**。

类的静态成员（数据、函数）既可以通过对象来访问，也可以通过**类名**来访问。

注意，静态成员函数不能访问非静态成员。这是因为静态成员函数属于整个类，在类实例化对象之前已经分配了内存空间；而类的非静态成员必须在类实例化对象后才分配内存空间。如果使用静态成员函数访问非静态成员，相当于没有声明一个变量却要使用它。

4. 类中的常量成员(const)

使用const修饰的数据成员，称为类的**常量数据成员**，在对象的整个生命周期里**不可更改**。

常量数据成员可以在：

- 构造函数的**初始化列表**中被初始化
- 就地初始化
- 不允许在构造函数的函数体中通过赋值来设置 ❌

成员**函数**也能用const来修饰，称为常量成员函数。例如：`int func() const {...}`该成员函数的实现语句**不能修改类的数据成员** —— 即改变对象状态（内容）

若**对象**被定义为常量，则它只能调用以const修饰的成员函数。

5. 函数静态对象

函数静态对象是指在**函数内部**定义的static对象。在程序执行到该局部静态对象的代码时被初始化，但是离开作用域不析构；第二次执行到该对象代码时，不再初始化，直接使用上一次的对象。在main()函数结束后被析构。

例如：

```
void fun(int i, int n) {
    if (i >= n)
        static A static_obj("static");
}
```

6. 参数对象的构造与析构

（1）如果传递的是形参：

```
void fun(A b) {
    cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
```

在函数被调用时，b被**构造**，调用**拷贝构造函数**（以后内容）进行初始化。默认情况下，对象b的属性值和a一致。

在函数结束时，调用析构函数，**b被析构**。在离开main()的时候，a被析构。所以，构造一次、析构两次。在类中包含指针的时候，要格外小心，因为构造一次、析构两次会导致指针指向位置的内存被重复释放，产生问题。这也就是为什么我们不喜欢用拷贝构造函数，而希望用移动构造函数的原因。（之后内容）

```
class A {…//A定义见前几页ppt}
void fun(A b) {
    cout << "In fun: b.s=" << b.s << endl;
}

int main() {
    A a("a");
    fun(a);
    return 0;
}
```

运行结果：
a A constructing
In fun: b.s=a
a A destructing //离开fun()
a A destructing //离开main()

构造一次，
析构两次！

（2）如果参数是类对象的**引用**

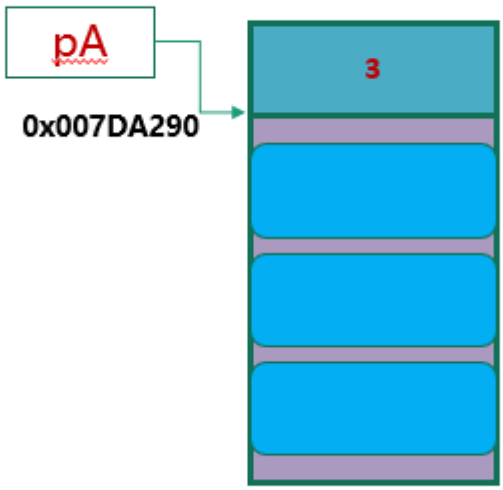
```
void fun(A &b) {
    cout << "In fun: b.s=" << b.s << endl;
}
fun(a);
```

在函数被调用时，b不需要初始化，因为b就是a的**引用**。在函数结束时，也不需要调用析构函数，因为b只是一个引用，而不是A的对象。

对象的new和delete

如果想生成一个类对象的数组，则需要调用new[]来分配足够大的原始未类型化的内存。注意要多出4个字节来存放数组的大小。例如：

```
A *pA = new A[3];
```



对于new[], 一定要配以delete**[]**, 来删除该对象数组及数组中的**每个元素**（释放内存资源），还要删除多的那4字节。

如果使用了new[], 但是只使用delete来删除，会出大问题！例如：

```
A *pA = new A[3];
delete pA;
```

该delete命令做了两件事：

- 调用一次 pA 指向的对象的**析构函数。**
- 释放pA地址的内存。

后果如下：

- 只调用一次析构函数。如果类对象中有大量申请内存的操作，那么因为没有调用析构函数，这些内存无法被释放，造成内存泄漏。

- 直接释放pA指向的内存空间，这个会造成严重的段错误，程序必然会崩溃。因为分配空间的起始地址，是pA-4byte。