

# 第三天课堂笔记

## 一、回顾知识点

### 1.1 const和#define的关系

尽量使用const变量替换#define, 因为const变量具有数据类型和作用域。

### 1.2 引用

引用的符号: &, 类似于const 指针。

引用的本质: 取变量的地址, 赋值给 const修饰的指针变量。

```
int a=10;
int &p = a;    // 等价于 int * const p = &a;
```

一个变量的引用, 实际上为变量的空间起个别名, 对引用的操作, 也就是对变量空间的操作。

引用作实参向函数传递是变量空间(地址)。

指针的引用: 数据类型 \*&引用名称 = 指针变量; `*(引用名) = 值`。

【注意】引用在初始化时必须给变量, 不能给常量(字面量)。

常量引用: const 数据类型 &引用名= 常量(字面量)或变量;

常量引用的数据不能被修改的。

数组引用: 数组名本身是一个地址, 定义引用时, 先确定元素的个数。

```
1) typedef 数据类型 新类型名[个数]
   typedef int ArrRef[10];

   ArrRef m = {1, 2, 3};
   int m2[10] = {1, 2, 3};

   ArrRef &mR = m;
   ArrRef &m2R = m2;

2) 数据类型 (&引用名)[数据个数] = 数组名;
   如:
       int (&mR)[10] = m;

3) 数组的引用作为函数的参数
   void sort(ArrRef arr);
   void sort(int (&arr)[10]);

   sort(mR);
   sort(m2);
```

## 1.3 函数参数的默认值和占位

函数的形参可以存在默认值，也可以是没有名称的类型占位（运算符重载）

【注意】形参从第一个存在默认值开始之后，必须都具有默认值。

## 1.4 函数的重载

与返回值类型无关的，与函数名和函数的形参列表有关；函数名相同，参数列表不同，参数列不同的情况有（个数不同、个数相同类型不同）。

## 1.5 extern "C"的作用

c++链接C编写的函数时，可能会按C++编译函数的规则对原函数名进行重命名（c++的函数是重载的），为了防止c++编译对C实现的函数重命名，可以使用 `extern "C" {}` 语法。

用法：在头文件中，对c的函数声明时额外增加条件判断

```
#if __cplusplus
    extern "C" {
#endif
    extern void show();

#if __cplusplus
}
#endif
```

c++编译时：

```
extern "C" {
    extern void show();
}
```

保护函数名不能被c++编译修改。

## 1.6 内联函数

普通函数前加 `inline`关键字。

作用：在调用内联函数时，不会压入栈，会直接向有参宏展开，节省大量调用和调出函数的时间，提高的程序运行效率。

使用内联函数可以完全替代有参宏：内联函数也是函数，包含数据类型检查、返回值类型检查等，而且具有作用域。

## 1.7 类和对象

类的定义：

```
class 类名{
    权限修饰符:
        成员变量;
        成员函数的声明或定义;
};

// 定义或实现类声明的成员函数
返回值类型 类名::声明的成员函数名(形参列表){

}
```

权限修饰符:

public 公开的, 类的内外都可以访问 (对象名.成员);

protected 受保护, 类的内部可以访问, 类的外部不能访问。子类可以访问。

private 私有的, 类的内部可以访问, 类的外部不能访问, 子类也不能访问。

对象: 某一个类的具体的一个实例。

```
类名 对象名;
对象名.成员属性名 = 新数据;

对象名.成员函数(实参列表); // 在调用对象自己的成员函数时, 默认存在this代表当前调用成员函数的对象, this是指针 (类名 *this);
```

## 二、类与对象II

### 2.1 对象的构造与析构

#### 2.1.1 构造函数与析构函数

对象的构造: 创建类的对象时, 默认调用类的某一个函数, 进行内存空间的创建和初始化类的成员变量的值, 这个函数称之为构造函数。

**构造函数:** 无返回值 (不能使用void), 函数名同类名。默认的构造函数是无参数。

对象的存储空间在栈区时, 程序结束时, 自动释放空间, 在释放空间之前, 会调用对象的析构函数, 回收资源。

**析构函数:** 无返回值, 无参数, 函数名同类名, 但名称前加 ~ 标识符。

【注意】构造函数可以存在多个 (具有重载特性), 析构函数只能有一个。构造函数和析构函数的访问权限尽量是public的。

如: 设计动物类 Animal, 具有name、food属性, 及eat()和say()行为。

```
class Animal{
private:
    char *name;
    char *food;

public:
    Animal();
    Animal(const char *name, const char *food);
```

```

    ~Animal(); // 析构函数

public:
    void say(const char *msg);
    void eat();
};

Animal::Animal(){
    cout << "--初始化对象---" << endl;
    name = (char *)malloc(32);
    food = (char *)malloc(32);
    strcpy(name, "小动物");
    strcpy(food, "水");
}

Animal::Animal(const char *name, const char *food){
    this->name = (char *)malloc(32);
    this->food = (char *)malloc(32);
    strcpy(this->name, name);
    strcpy(this->food, food);
}

void Animal::say(const char *msg){
    cout << name << "遇到主人: " << msg << endl;
}

void Animal::eat(){
    cout << name << "吃: " << food << endl;
}

Animal::~~Animal(){
    cout << "--回收资源---" << endl;
    // 回收在构造函数（初始化）中动态创建的空间
    free(name);
    free(food);
}

```

测试类：

```

int main(){
    // 尽量使用堆空间 创建类的对象： 使用类指针和new关键字
    // 类名 *对象名 = new 类的构造函数(参数列表);

    Animal a1; // 默认调用无参的构造函数
    a1.eat();

    return 0;
}

```

完整的代码：

```

#include <iostream>
#include <cstring>
#include <cstdlib>

```

```

using namespace std;
class Animal
{
private:
    char *name;
    char *food;

public:
    Animal();
    Animal(const char *name, const char *food);
    ~Animal(); // 析构函数

public:
    void say(const char *msg);
    void eat();
};

Animal::Animal()
{
    cout << "--初始化对象---" << endl;
    name = (char *)malloc(32);
    food = (char *)malloc(32);
    strcpy(name, "小动物");
    strcpy(food, "水");
}

Animal::Animal(const char *name, const char *food)
{
    this->name = (char *)malloc(32);
    this->food = (char *)malloc(32);
    strcpy(this->name, name);
    strcpy(this->food, food);
}

void Animal::say(const char *msg)
{
    cout << name << "遇到主人: " << msg << endl;
}

void Animal::eat()
{
    cout << name << "吃: " << food << endl;
}

Animal::~~Animal()
{
    cout << "--回收资源---" << endl;
    // 回收在构造函数（初始化）中动态创建的空间
    free(name);
    free(food);
}

int main(int argc, char const *argv[])
{
    // 尽量使用堆空间 创建类的对象: 使用类指针和new关键字
    // 类名 *对象名 = new 类的构造函数(参数列表);

    Animal a1; // 默认调用无参的构造函数

```

```

a1.eat();

Animal a2("小黄", "肉"); // 使用有参的构造函数
a2.eat();
a2.say("旺旺...");
return 0;
}

```

```

disen@qfxa:~/code2/day03$ g++ demo1.cpp
disen@qfxa:~/code2/day03$ ./a.out
--初始化对象---
小动物吃：水
小黄吃：肉
小黄遇到主人：旺旺...
--回收资源---
--回收资源---

```

### 2.1.2 构造函数的分类与调用

按参数类型：分为无参构造函数和有参构造函数

按类型分类：普通构造函数和拷贝构造函数(复制构造函数)

拷贝的构造函数：接收同类的其它对象，将其它对象中的数据复制到当前对象中。

如：

```

class A{
private:
    int x;
public:
    A(){}
    A(A &other){
        x = other.x;
    }
};

int main(){
    A a1; // 调用无参的构造函数进行初始化a1对象。
    return 0;
}

```

【注意】如果没有显式地声明构造函数，则编译器自动添加一个无参的构造函数(隐式)。如果显式地声明构造函数，则编译器不会提供无参的构造函数。

```

disen@qfxa:~/code2/day03$ g++ demo2.cpp
demo2.cpp: In function 'int main()':
demo2.cpp:18:7: error: no matching function for call to 'A::A()'
    A a1; // 调用无参的构造函数进行初始化a1对象。
      ^
demo2.cpp:10:5: note: candidate: A::A(A&)
    A(A &other)
      ^
demo2.cpp:10:5: note:   candidate expects 1 argument, 0 provided

```

如果显式地声明构造函数，也应该提供一个无参的构造函数，可供创建类对象使用（类名 对象名）。

如：优化之后的A类

```

#include <iostream>
using namespace std;

class A
{
public:
    int x; // c++的成员变量，未设置初始值时，默认随机

public:
    A() {}
    A(A &other)
    {
        x = other.x;
    }
};

int main()
{
    A a1; // 调用无参的构造函数进行初始化a1对象。
    a1.x = 100;
    cout << "a1 x = " << a1.x << endl;

    A a2(a1);
    cout << "a2 x = " << a2.x << endl;
    return 0;
}

```

```

disen@qfxa:~/code2/day03$ ./a.out
a1 x = 100
a2 x = 100

```

构造函数的调用方式：

- 1) 无参的调用  
类名 对象名；
- 2) 有参的调用  
类名 对象名(参数列表)
- 3) 匿名（无对象名）调用  
类名(参数列表) // 创建了对象，但是没有名称

类名 对象名 = 类名(参数列表);

4) = 调用, 针对单个参数的构造函数

类名 对象名 = 常量(字面量) // 等价于 类名 对象名= 类名(常量);

类名 对象名 = 同类的其它对象 ; // 等价于 类名 对象名= 类名(同类其它对象)

5) new方式

类名 \*对象名 = new 类名(参数列表);

free(对象名)

如:

```
#include <iostream>
using namespace std;

class B
{
private:
    int n;

public:
    B() { cout << "B()" << endl; }
    B(int n)
    {
        cout << "B(int)" << endl;
        this->n = n;
    }
    B(const B &other)
    {
        cout << "B(B &)" << endl;
        this->n = other.n;
    }

    void showN()
    {
        cout << "n =" << n << endl;
    }
};

int main()
{
    // B b0;
    // B b1 = 20; // 隐式调用 B(20), 只创建一个对象
    B b2 = B(30); // 显式调用 B(30), 只创建一个对象
    B b3 = b2;    // 隐式调用 B(B &) 构造函数
    B &b4 = b3;   // b4 是 b3对象的引用, 不会调用构造函数
    B b5(b4);    // 显式调用 B(B &)
    return 0;
}
```



```
disen@qfxa:~/code2/day03$ ./a.out
B(int)
B(B &)
B(B &)
```

如:

```
#include <iostream>
using namespace std;

class C
{
private:
    int x, y;

public:
    C(int x) { this->x = x; }
    C(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
    C(const C &other)
    {
        this->x = other.x;
        this->y = other.y;
    }
    void show()
    {
        cout << "x=" << x << ", y=" << y << endl;
    }
};

int main()
{
    C c1(1, 2);
    c1.show();
    C c2 = (2, 3);
    c2.show();
    C c3 = C(3, 4);
    c3.show();
    C c4(c3);
    c4.show();
    C &c5 = c4;
    c5.show();

    return 0;
}
```

```
disen@qfxa:~/code2/day03$ ./a.out
x=1, y=2
x=3, y=0
x=3, y=4
x=3, y=4
x=3, y=4
```

如：创建类对象的指针

```
#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;
class A
{
public:
    char *msg;

public:
    A()
    {
        this->msg = (char *)malloc(30);
        strcpy(this->msg, "haha");
    }
    A(const char *msg)
    {
        this->msg = (char *)malloc(30);
        strcpy(this->msg, msg);
    }
    ~A()
    {
        cout << this << "release msg" << this->msg << endl;
        free(this->msg);
    }
};

int main()
{
    A *a1 = new A;
    cout << "a1 msg: " << a1->msg << endl;

    A *a2 = new A("disen 666");
    cout << "a2 msg: " << a2->msg << endl;

    // 【注意】指针释放空间时， 不会执行对象的析构函数
    // free(a1->msg); // 手动释放对象成员的堆空间
    a1->~A(); // 调用析构函数释放成员的空间
    cout << "-----" << endl;
    free(a1);
}
```

```

    free(a2->msg);
    free(a2);
    return 0;
}

```

```

disen@qfxxa:~/code2/day03$ g++ demos.cpp
disen@qfxxa:~/code2/day03$ ./a.out
a1 msg: haha
a2 msg: disen 666

```

```

#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;
class A
{
public:
    char *msg;

public:
    A()
    {
        this->msg = (char *)malloc(30);
        strcpy(this->msg, "haha");
    }
    A(const char *msg)
    {
        this->msg = (char *)malloc(30);
        strcpy(this->msg, msg);
    }
    ~A()
    {
        cout << this << "release msg" << this->msg << endl;
        free(this->msg);
    }
};

int main()
{
    A *a1 = new A;
    cout << "a1 msg: " << a1->msg << endl;

    A *a2 = new A("disen 666");
    cout << "a2 msg: " << a2->msg << endl;

    // 【注意】指针释放空间时，不会执行对象的析构函数
    // free(a1->msg); // 手动释放对象成员的堆空间
    // a1->~A(); // 调用析构函数释放成员的空间
    // cout << "-----" << endl;
    // free(a1);
    // free(a2->msg);
    // free(a2);
    delete a1;
}

```

```

delete a2;

// 【注意】delete只能删除对象的指针不能删除对象，对象在作用域之外，自动释放空间（执行析构函数）
// A a3;
// A &a4 = a3;
// delete a4;
return 0;
}

```

### 【注意事项】

- 1) 创建的类的对象，如果是指针类型，则需要手动释放（free、调用析构函数、delete）
- 2) 对于直接通过类创建的对象，不能手动释放，在对象的作用域之外自地释放（调用析构函数）

## 2.1.3 拷贝构造函数的调用时机

拷贝构造函数的调用时机：

- 1) 类对象作为右值（rvalue）赋值给定义类的对象时，会调用拷贝构造函数
- 2) 函数的局部对象直接返回时，可能会调用构造函数（vs debug会，qt不会）
- 3) 类对象作为函数的参数（以值的方式传递）时，可能会调用构造函数

如：

```

#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;

class A
{
public:
    int x;

public:
    A()
    {
        x = 0;
        cout << this << " A()" << endl;
    }
    A(const A &other)
    {
        x = other.x;
        cout << this << " A(const A &other)" << endl;
    }
    ~A()
    {
        cout << this << " ~A()" << endl;
    }
};

void test1(A a)
{ // ? 调用A(const A &other)
    cout << "test1() a.x=" << a.x << endl;
}

```

```

        // 当函数结束时，a释放
    }

    A test2()
    {
        A a1; // 局部的类的对象
        a1.x = 100;
        return a1; // 释放a1对象
    }

    void test3()
    {
        A a2 = test2(); // 返回A类的对象， ? 拷贝构造函数（g++结果没有执行） 说明 a2是test2()
        函数中创建a1的引用。
        cout << "a2.x = " << a2.x << endl;
        cout << "test3" << endl;
    }

    int main()
    {
        A a0;
        test1(a0);
        test3();
        cout << "over" << endl;
        return 0;
    }

```

```

disen@qfxa:~/code2/day03$ ./a.out
0x7ffe94dc2c70  A()
0x7ffe94dc2c80  A(const A &other)
test1() a.x=0
0x7ffe94dc2c80  ~A()
0x7ffe94dc2c40  A()
a2.x = 100
test3
0x7ffe94dc2c40  ~A()
over
0x7ffe94dc2c70  ~A()

```

当函数返回一个局部对象时，编译器做了优化，在函数外可以接收的（转化为引用），如果手动声明返回类型为对象的引用，反而会导致局部的对象在返回之后自动释放空间，使得外部使用时报 段错误。如下代码：

```

#include <iostream>
#include <cstring>
#include <cstdlib>

```

```

using namespace std;

class A
{
public:
    int x;

public:
    A()
    {
        x = 0;
        cout << this << " A()" << endl;
    }
    A(const A &other)
    {
        x = other.x;
        cout << this << " A(const A &other)" << endl;
    }
    ~A()
    {
        cout << this << " ~A()" << endl;
    }
};

void test1(A &a)
{
    cout << "test1() a.x=" << a.x << endl;
}

// 不要声明为 A &test2(), 返回对象时, 会释放空间, 在外部使用时会发生段错误
// 正确的写法是 去掉 &
A &test2()
{
    A a1; // 局部的类的对象
    a1.x = 100;
    return a1; // 释放a1对象
}

void test3()
{
    // 正确的写法 A a2 = test2();
    A &a2 = test2(); // 返回对象的引用, 错误的写法
    cout << "a2.x = " << a2.x << endl;
    cout << "test3" << endl;
}

int main()
{
    A a0;
    test1(a0);
    test3();
    cout << "over" << endl;
    return 0;
}

```

```

disen@qfxa:~/code2/day03$ g++ demo7.cpp
demo7.cpp: In function 'A& test2()':
demo7.cpp:37:7: warning: reference to local variable 'a1' returned [-Wreturn-local-addr]
    A a1; // 局部的类的对象
      ^
disen@qfxa:~/code2/day03$ ./a.out
0x7ffc18521c70 A()
test1() a.x=0
0x7ffc18521c10 A()
0x7ffc18521c10 ~A()
Segmentation fault (core dumped)

```

## 2.1.4 构造函数调用规则

默认情况下，c++编译器至少为我们写的类增加 3 个函数

1. 默认构造函数(无参，函数体为空)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对类中非静态成员属性简单值拷贝

如果用户定义拷贝构造函数，c++不会再提供任何默认构造函数如果用户定义了普通构造(非拷贝)，c++不在提供默认无参构造，但是会提供默认拷贝构造。

如：

```

#include <iostream>

using namespace std;

class A
{
public:
    int x, y;
    void show()
    {
        cout << this << " x=" << x << ",y=" << y << endl;
    }
};

int main(int argc, char const *argv[])
{
    A a1;
    A a2 = a1; // 自动调用拷贝构造函数 A(const A&obj)
    a1.x = 20;
    a1.y = 30;

    a1.show();
    a2.show();
    return 0;
}

```

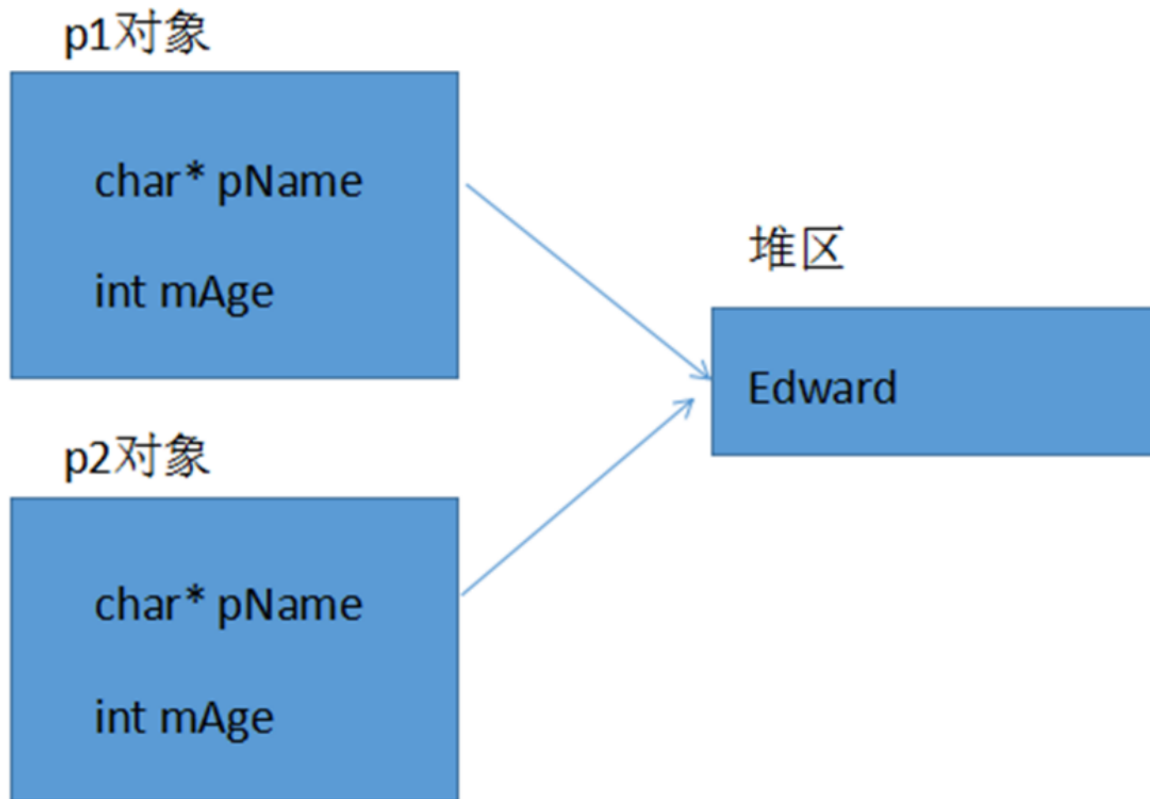
```

disen@qfxa:~/code2/day03$ ./a.out
0x7ffceb9f8160 x=20,y=30
0x7ffceb9f8170 x=4197008,y=0

```

## 2.1.5 深拷贝和浅拷贝

浅拷贝：只复制一层内存空间



如：

```
#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;

class Person
{
private:
    char *name;
    int age;

public:
    Person(const char *name, int age)
    {
        // 为name在堆中申请空间
        this->name = (char *)malloc(32);
        strcpy(this->name, name);
        this->age = age;
    }
    void release_name_pointer()
    {
        // 多个对象共用一个堆空间
        // 当某一个对象执行析构时，就会释放
        // 一个空间不能被多次释放
        if (name != NULL)
        {
            free(name);
        }
    }
}
```



```

    }
}
void setName(const char *name)
{
    strcpy(this->name, name);
}
void show()
{
    cout << name << ", " << age << endl;
}
};

int main()
{
    Person p1("Disen", 20);
    Person p2 = p1;
    Person p3 = Person(p2);
    p3.setName("Jack");

    p1.show();
    p2.show();
    p3.show();

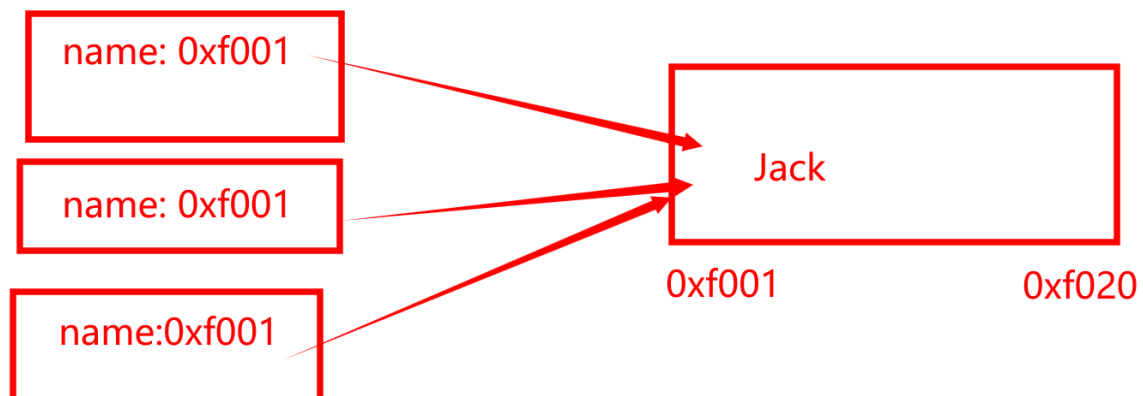
    // 手动释放堆中分配的空间
    p1.release_name_pointer();
    return 0;
}

```

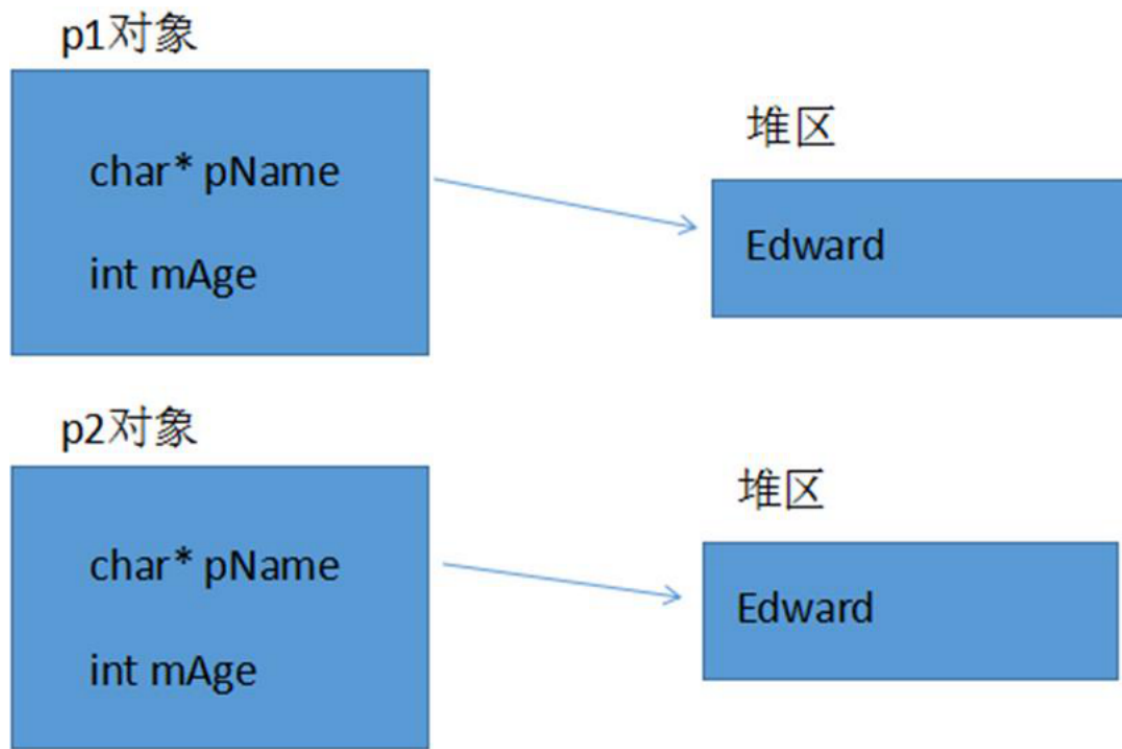
```

disen@qfxa:~/code2/day03$ ./a.out
Jack, 20
Jack, 20
Jack, 20

```



深拷贝：所有层次空间全部复制一份



如：每一层的堆空间的成员变量都要重新分配内存空间，将拷贝对象的数据复制过来。

```
#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;

class Person
{
private:
    char *name;
    int age;

public:
    Person(const char *name, int age)
    {
        // 为name在堆中申请空间
        this->name = (char *)malloc(32);
        strcpy(this->name, name);
        this->age = age;
    }
    Person(const Person &obj)
    {
        // 新创建一个空间
        this->name = (char *)malloc(32);
        strcpy(this->name, obj.name);
        this->age = obj.age;
    }
    ~Person()
    {
        if (name != NULL)
        {
            free(name);
        }
    }
}
```

```

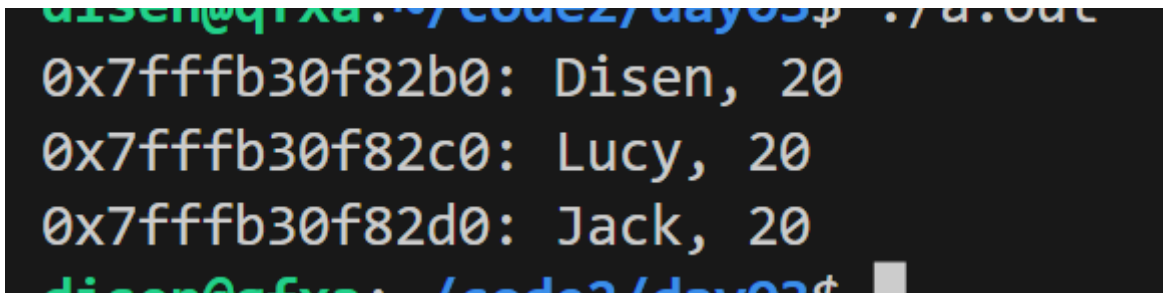
    }
}
void setName(const char *name)
{
    strcpy(this->name, name);
}
void show()
{
    cout << this << ": " << name << ", " << age << endl;
}
};

int main()
{
    Person p1("Disen", 20);
    Person p2 = p1;
    p2.setName("Lucy");
    Person p3 = Person(p2);
    p3.setName("Jack");

    p1.show();
    p2.show();
    p3.show();

    return 0;
}

```



```

0x7ffffb30f82b0: Disen, 20
0x7ffffb30f82c0: Lucy, 20
0x7ffffb30f82d0: Jack, 20

```

## 2.1.6 多个对象构造和析构

### 2.1.6.1 初始化列表

构造函数：主要用于创建类的对象，在定义构造函数时，C++中提供了初始化列表的语法,用于初始化成员变量的值。

类名(参数列表):成员名(参数名),成员名2(参数名2),... { }

如:

```

#include <iostream>

using namespace std;

class A
{
private:
    int x, y, z;
    // char *name;    char * 在c++用string替代了
    string name;

```

```

public:
    A(int x, int y, int z, string name) : x(x), y(y), z(z), name(name) {}
    void show()
    {
        cout << name << ", " << x << ", " << y << ", " << z << endl;
    }
};

int main(int argc, char const *argv[])
{
    A a1(10, 2, 3, "disen");
    a1.show();
    return 0;
}

```

```

disen@qfxa:~/code2/day03$ g++ demo10.cpp
disen@qfxa:~/code2/day03$ ./a.out
disen,10,2,3

```

### 2.1.6.2 类对象作为成员

如：类对象作为成员使用时的构造与析构

```

#include <iostream>

using namespace std;
class A
{
public:
    A()
    {
        cout << "A()" << endl;
    }
    A(int x)
    {
        cout << "A(int)" << endl;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

class B
{
public:
    B() // 构造函数用于初始化成员数据，说明成员变量先创建
    {
        cout << "B()" << endl;
    }
    B(int x)
    {
        cout << "B(int)" << endl;
    }
    ~B() // A的对象是B的空间中，A是B空间释放之后就失效，也会释放空间

```

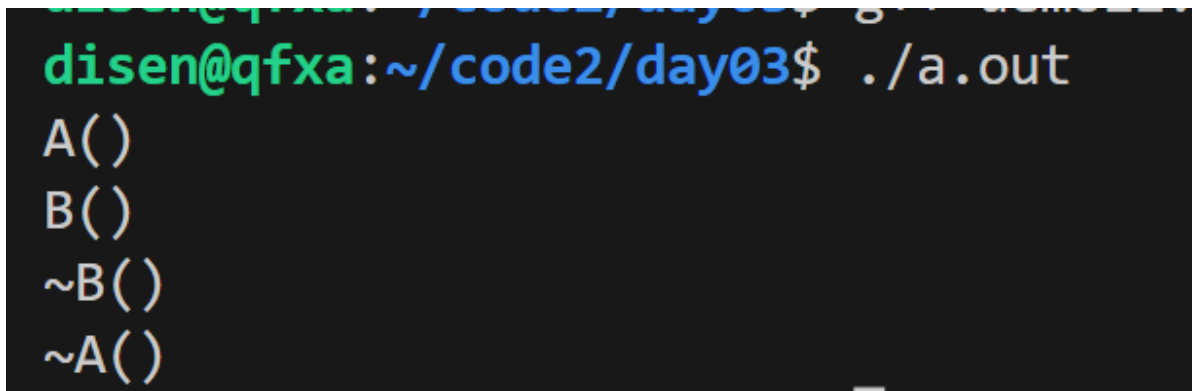
```

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

private:
    A a; // A类的对象作为B类的成员
};

int main(int argc, char const *argv[])
{
    B b1;
    // 输出什么?
    return 0;
}

```



```

disen@qfxa:~/code2/day03$ ./a.out
A()
B()
~B()
~A()

```

说明:

- 1) 创建B空间时，成员变量a的内存空间会自动创建。
- 2) 当成员变量a的空间创建完成后，再调用构造函数，进行数据初始化
- 3) 当有B空间释放时，成员变量a的作用域失效，则也会释放空间

在B类中，通过构造函数的初始化列表，可以指定类成员的构造函数的调用

```

#include <iostream>

using namespace std;
class A
{
public:
    A()
    {
        cout << "A()" << endl;
    }
    A(int x)
    {
        cout << "A(int)" << endl;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

class B
{
public:

```

```

B() // 构造函数用于初始化成员数据，说明成员变量先创建
{
    cout << "B()" << endl;
}
B(int x) : a(x) // 指定a对象的构造函数进行数据初始化
{
    cout << "B(int)" << endl;
}
~B() // A的对象是B的空间中，A是B空间释放之后就失效，也会释放空间
{
    cout << "~B()" << endl;
}

private:
    A a; // A类的对象作为B类的成员
};

int main(int argc, char const *argv[])
{
    B b1(1);
    // 输出什么?
    return 0;
}

```

```

disen@qf+xa:~/code2/day03$ g++ demo11.cpp
disen@qf+xa:~/code2/day03$ ./a.out
A(int)
B(int)
~B()
~A()

```

### 2.1.7 explicit 关键字

c++提供了关键字 explicit，禁止通过构造函数进行的隐式转换（格式：类对象 = 值）。

声明为 explicit 的构造函数不能在隐式转换中使用。构造函数的参数只能存在一个或第一个没有默认值（其它参数都具有默认值）的情况

如：

```

#include <iostream>

using namespace std;
class Person
{
private:
    int age;
    string name;

public:
    Person(const char *name)
    {
        this->name = name;
        age = 18;
    }
}

```

```

    }
    explicit Person(int age) // 禁止 Person p=值;
    {
        this->age = age;
        this->name = "disen";
    }
    void show()
    {
        cout << name << ", " << age << endl;
    }
};

int main(int argc, char const *argv[])
{
    // Person p1 = 20; // error
    Person p1 = Person("jack");
    // Person p1 = "jack"; // OK
    p1.show();
    return 0;
}

```

```

disen@qfxa:~/code2/day03$ g++ demo12.cpp
disen@qfxa:~/code2/day03$ ./a.out
jack, 18
disen@qfxa:~/code2/day03$ █

```

如：构造函数的参数出现默认值

```

#include <iostream>

using namespace std;
class Person
{
private:
    int age;
    string name;

public:
    Person(const char *name, int age = 18)
    {
        cout << "-- Person(const char *, int)---" << endl;
        this->age = age;
        this->name = name;
    }
    explicit Person(int age) // 禁止 Person p=值;
    {
        this->age = age;
        this->name = "disen";
    }
    void show()
    {
        cout << name << ", " << age << endl;
    }
};

int main(int argc, char const *argv[])

```

```

{
    // Person p1 = 20; // error
    // Person p1 = Person("jack");
    Person p1 = "jack"; // OK
    p1.show();
    return 0;
}

```

## 2.1.8 动态创建对象

动态创建对象方式：

- 1) `malloc`, `realloc`, `calloc` 在堆中创建空间， 使用完之后，需要手动释放`free()`  
 当内存空间创建完之后，进行数据的初始化或者调用自定义的初始化函数（默认不会调用构造函数，构造函数不能显示调用）。
- 2) `new` 关键字创建对象空间  
 使用`new` 在堆中创建空间之后，自动调用对象的构造函数进行数据初始化。  
 使用`new`创建的对象，可以通过`delete`关键字来调用对象的析构函数释放内存

如：1)使用`malloc`方式创建

```

#include <iostream>
#include <cstdlib>

using namespace std;

class A
{
private:
    int x;

public:
    A(int x)
    {
        this->x = x;
        cout << "A(int)" << endl;
    }
    void init(int x)
    {
        this->x = x;
        cout << "init(int)" << endl;
    }
    void clean()
    {
        cout << "clean()" << endl;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    // 1) 使用malloc方式创建
    cout << "A size is " << sizeof(A) << endl;
    A *a1 = (A *)malloc(sizeof(A));
}

```



```

// a1->A(20); //error 不让显示调用构造函数
a1->init(20); // 初始化数据
a1->clean(); // 释放数据的空间（成员变量的堆空间）
a1->~A(); // 对象的析构函数可以显式调用，可以省略
free(a1); // 释放对象的空间
return 0;
}

```

```

disen@qfxa:~/code2/day03$ ./a.out
A size is 4
init(int)
clean()
~A()

```

如：2) new 方式创建， delete释放

```

#include <iostream>
#include <cstdlib>

using namespace std;

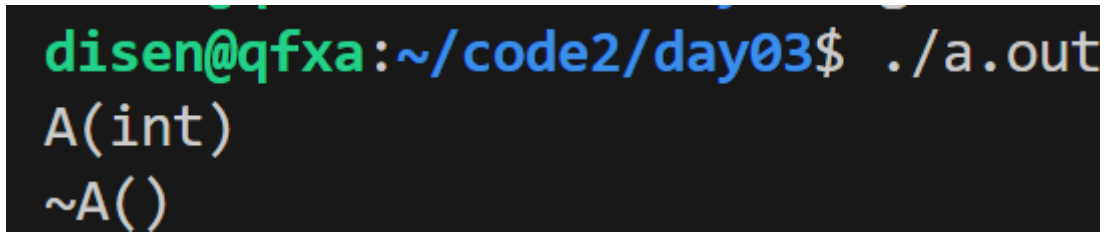
class A
{
private:
    int x;

public:
    A(int x)
    {
        this->x = x;
        cout << "A(int)" << endl;
    }
    void init(int x)
    {
        this->x = x;
        cout << "init(int)" << endl;
    }
    void clean()
    {
        cout << "clean()" << endl;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    // 2) new 方式创建， delete释放
    A *a1 = new A(1);
    // 用完之后，释放空间
    delete a1;
}

```

```
    return 0;
}
```



A terminal window with a black background and green text. The prompt is `disen@qfxa:~/code2/day03$`. The command `./a.out` has been executed. Below the prompt, the following function definitions are visible:  
`A(int)`  
`~A()`

## 2.1.9 扩展new和delete

### 用于数组的 new 和 delete

创建数组： 堆空间中创建  
数据类型 \*指针变量名 = new 数据类型[个数];  
数据类型 \*指针变量名 = new 数据类型[个数]{元素,...};

删除new创建的数组：  
delete[] 指针变量名;

如：

```
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    int *p = new int[6]{1, 2, 3, 4, 5, 6};
    for (int i = 0; i < 6; i++)
        cout << *(p + i) << endl;
    delete[] p;
    return 0;
}
```

动态创建类对象的数组时，如果没有指定类对象的创建方式时，必须提供一个无参的构造函数：

```
类名 *p = 类名[个数];
```

如下方式，初始化类对象数组成员，指定了对象的构造函数

```
类名 *p = 类名[个数]{类名(参数列表), ...};
```

如：

```
#include <iostream>

using namespace std;
class A
{
private:
```

```

    string name;

public:
    A()
    {
        cout << "A()" << endl;
        name = "no name";
    }
    A(const string &name)
    {
        this->name = name;
    }
    ~A()
    {
        cout << name << " ~A()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    A *p = new A[5]; // 必须提供无参的构造函数
    A *p2 = new A[4]{A("a"), A("b"), A("c"), A("d")};
    delete[] p;
    delete[] p2; // 释放数组中成员对象时，从高(地址)到低(地址)依次释放。
    return 0;
}

```

### delete void\*可能会出错

```

#include <iostream>
#include <cstdlib>

using namespace std;
class A
{
private:
    string name;

public:
    A()
    {
        cout << "A()" << endl;
        name = "no name";
    }
    A(const string &name)
    {
        this->name = name;
    }
    ~A()
    {
        cout << name << " ~A()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    A *a = new A("abc");
    free(a); // 只释放指针指向的堆空间，不会调用对象的析构函数
}

```

```
void *a2 = malloc(sizeof(A)); // 指向的类对象大小的空间,未明确是否为类的对象
delete a2; // 可以删除 简单的void *指针, 无法确认类型, 不会调用对应类型的析构

return 0;
}
```