对象的构造

> 17 ~ 19

小实验

对象中成员变量的初始值是多少?

`lesson_17-1.h`

```
class Test
{
private:
    int i;
    int j;
public:
    int getI() {return i;}
    int getJ() {return j;}
};
```

`lesson_17-1_main.cpp`

```
#include <stdio.h>
#include "lesson_17-1.h"

// 全局对象
Test gt;

int main(void)
{
    // 局部对象
    Test t;
    // 全局对象输出
    printf("gt.getI() = %d\n", gt.getI());
    printf("gt.getJ() = %d\n", gt.getJ());

    // 局部对象输出
    printf("t.getI() = %d\n", t.getI());
    printf("t.getJ() = %d\n", t.getJ());

    Test* pt = new Test;
    printf("pt->getI() = %d\n", pt->getI());
    printf("pt->getJ() = %d\n", pt->getJ());

    delete pt;

    return 0;
}
```

输出

```
root@ubuntu:~/exp/DT_CPP/part01# ./a.out
gt.getI() = 0
gt.getJ() = 0
t.getI() = -1323098096
t.getJ() = 32766
pt->getI() = 0
pt->getJ() = 0
```

> `Test`只是自定义类型，跟基本数据类型一样，定义的变量可以存储在全局数据区、栈区、堆区。
> 初始值就看定义在哪个区域
> 在堆区中的对象的成员变量也是0只是巧合
> 使用`new`关键字创建的对象最后需要使用`delete`关键字释放

- 从程序设计的角度，**对象只是变量**，因此：
  - 在栈上创建对象时，成员变量初始值为 随机值
  - 在堆上创建对象时，成员变量初始值为 随机值
  - 在静态存储区创建的对象时(**全局对象**和**static修饰的局部对象**)，成员变量初始值为 0

对象的初始化

> 在类中提供一个在`public`中的`initialize`函数
> 对象创建后立即手动调用`initialize`函数进行初始化

`lesson_17-2.h`

```
class Test
{
private:
    int i;
    int j;
public:
    void initialize()
    {
        i = 0;
        j = 0;
    }
    int getI() {return i;}
    int getJ() {return j;}
};
```

`lesson_17-2_main.cpp`

```cpp
#include <stdio.h>
#include "lesson_17-2.h"

// 全局对象
Test gt;

int main(void)
{
    // 局部对象
    Test t;
    // 全局对象输出
    gt.initialize();
    printf("gt.getI() = %d\n", gt.getI());
    printf("gt.getJ() = %d\n", gt.getJ());

    // 局部对象输出
    t.initialize();
    printf("t.getI() = %d\n", t.getI());
    printf("t.getJ() = %d\n", t.getJ());

    Test* pt = new Test;
    pt->initialize();
    printf("pt->getI() = %d\n", pt->getI());
    printf("pt->getJ() = %d\n", pt->getJ());

    delete pt;

    return 0;
}
```

> 以上程序的`initialize`函数需要手动调用，存在问题：
>
> 1. `initialize`函数**只是一个普通函数**，必须显示调用
> 2. 如果未调用`initialize`函数，运行结果不确定

C++可以定义与类名相同的特殊成员函数

- 这种特殊的成员函数叫做 **构造函数**
  - 构造函数没有任何返回类型的声明
  - 构造函数在对象定义时自动被调用

`lesson_17-3.h`

```cpp
class Test
{
private:
    int i;
    int j;
```

```
public:
    /*void initialize()
    {
        i = 0;
        j = 0;
    }*/
    Test()
    {
        i = 0;
        j = 0;
    }
    int getI() {return i;}
    int getJ() {return j;}
};
```

lesson_17-3_main.cpp

```
#include <stdio.h>
#include "lesson_17-3.h"

// 全局对象
Test gt;

int main(void)
{
    // 局部对象
    Test t;
    // 全局对象输出

    printf("gt.getI() = %d\n", gt.getI());
    printf("gt.getJ() = %d\n", gt.getJ());

    // 局部对象输出

    printf("t.getI() = %d\n", t.getI());
    printf("t.getJ() = %d\n", t.getJ());

    Test* pt = new Test;

    printf("pt->getI() = %d\n", pt->getI());
    printf("pt->getJ() = %d\n", pt->getJ());

    delete pt;

    return 0;
}
```

小结

- 每个对象使用前都应该初始化
- 类的构造函数**用于对象的初始化**
- 构造函数**与类名同名**并且没有返回值(**连 void都不用写**)
- 构造函数在对象定义时**自动被调用**

构造函数

- 带有参数的构造函数
  - 构造函数可以根据需要定义参数
  - **一个类中可以存在多个重载的构造函数**
  - **构造函数的重载遵循C++重载的规则**
- 对象定义和对象声明不同
  - 对象定义：申请对象的空间并调用构造函数
  - 对象声明：告诉编译器存在这样一个对象

```
Test t;  // 定义对象并调用构造函数
int main(void)
{
    extern Test t;  // 告诉编译器存在名为 t的Test对象
    return 0;
}
```

构造函数的自动调用

> 根据函数重载规则匹配构造函数

`lesson_18_1_main.cpp`

```
#include <stdio.h>

class Test
{
public:
    Test()
    {
        printf("call Test()\n");
    }
    Test(int v)
    {
        printf("call Test(int v), v = %d\n", v);
    }
};

```

```
int main(void)
{
    Test t;       // 调用Test()
    Test t1(1);   // 调用Test(int v)
    Test t2 = 2;  // 调用 Test(int v)

    int i(100);
    printf("i = %d\n", i);

    return 0;
}
```

输出

```
root@ubuntu:~/exp/DT_CPP/part01# ./a.out
call Test()
call Test(int v), v = 1
call Test(int v), v = 2
i = 100
```

构造函数的手动调用

- 一般情况下，构造函数在对象定义时被自动调用
- 特殊情况，需要手动调用构造函数【创建对象数组】

构造函数的手动调用

`lesson_18_2_main.cpp`

```
#include<stdio.h>
class Test
{
private:
    int m_value;
public:
    Test()
    {
        m_value = 0;
        printf("Test()\n");
    }
    Test(int v)
    {
        m_value = v;
        printf("Test(v), v = %d\n", v);
    }
    int getvalue()
    {
        return m_value;
```

```
    }
};
int main()
{
    Test ta[3] = {Test(), Test(1), Test(2)};  // call
    for (int i = 0; i < 3; i++)
    {
        printf("ta[%d].getvalue() = %d\n", i, ta[i].getvalue());
    }
    Test t = Test(100);  // call
    printf("t.getvalue() = %d\n", t.getvalue());
    return 0;
}
```

输出

```
root@ubuntu:~/exp/DT_CPP/part01# ./a.out
Test()
Test(v), v = 1
Test(v), v = 2
ta[0].getvalue() = 0
ta[1].getvalue() = 1
ta[2].getvalue() = 2
Test(v), v = 100
t.getvalue() = 100
```

自定义实例

> 开发数组类
>
> - 提供函数获取数组长度
> - 提供函数获取数组元素
> - 提供函数设置数组元素

> 原始数组？？？

`lesson_18_3.h`

```
#include <stdio.h>

class Array
{
private:
    int* ptr = NULL;
    int ArrayLen;

```

```
public:
    Array();
    Array(int);
    ~Array();
    int getLength();
    int getElement(int);
    bool setElement(int, int);
};
```

lesson_18-3.cpp

```
#include "lesson_18-3.h"
void ArrayInit(int* ptr, int len)
{
    for (int i = 0; i < len; i++)
        *(ptr + i) = 0;
}
Array::~Array()
{
    if (ptr != NULL)
        delete[] ptr;
}
Array::Array()
{
    ptr = new int[10];
    ArrayLen = 10;
    ::ArrayInit(ptr, 10);
}
Array::Array(int len)
{
    ptr = new int[len];
    ArrayLen = len;
    ::ArrayInit(ptr, len);
}

int Array::getLength()
{
    return ArrayLen;
}

int Array::getElement(int index)
{
    if (index < ArrayLen)
        return ptr[index];

    return -1;
}

bool Array::setElement(int index, int value)
{
    if (index < ArrayLen)
```

```
        ptr[index] = value;
        return true;
    return false;
}
```

lesson_18-3_main.cpp

```
#include <stdio.h>
#include "lesson_18-3.h"

int main(void)
{
    Array* pa = new Array; // 写法1
    Array pa1(30);  // 调用有参构造
    pa->setElement(4, 99);
    printf("pa->getLength() = %d\n", pa->getLength());          // 10
    printf("pa->getElement(4) = %d\n", pa->getElement(4));      // 99
    printf("pa->getElement(10) = %d\n", pa->getElement(10));    // -1

    pa1.setElement(4, 77);
    printf("pa1.getLength() = %d\n", pa1.getLength());        // 30
    printf("pa1.getElement(4) = %d\n", pa1.getElement(4));    // 99
    printf("pa1.getElement(10) = %d\n", pa1.getElement(10)); // -1

    delete pa;

    return 0;
}
```

课程实例

```
// IntArray.h
#ifndef _INTARRAY_H_
#define _INRARRAY_H_
class IntArray
{
private:
    int m_length;
    int* m_pointer;
public:
    IntArray(int len);
    int length();
    bool get(int index, int& value);
    bool set(int index, int value);
    void free();
};
#endif
```

```cpp
// IntArray.cpp
#include"IntArray.h"
IntArray::IntArray(int len)
{
    m_pointer = new int[len];
    for (int i = 0; i < len; i++)
    {
        m_pointer[i] = 0;
    }
    m_length = len;
}
int IntArray::length()
{
    return m_length;
}
bool IntArray::get(int index, int& value)
{
    bool ret = (index >= 0 && index < m_length);
    if (ret)
    {
        value = m_pointer[index];
    }
    return ret;
}
bool IntArray::set(int index, int value)
{
    bool ret = (index >= 0 && index < m_length);
    if (ret)
    {
        m_pointer[index] = value;
    }
    return ret;
}
void IntArray::free()
{
    delete[]m_pointer;
}
```

```cpp
// 18-3.cpp
#include<stdio.h>
#include"IntArray.h"
int main()
{
    IntArray a(5);
    for (int i = 0; i < a.length(); i++)
    {
        a.set(i, i+1);
    }
    for (int i = 0; i < a.length(); i++)
    {
```

```
        int value = 0;
        if (a.get(i, value))
        {
            printf("a[%d] = %d\n", i, value);
        }
    }
    a.free();
    return 0;
}
```

```
root@ubuntu:~/exp/DT_CPP/part01/lesson_18/example_array_class# g++ 18-3.cpp IntArray.cpp
root@ubuntu:~/exp/DT_CPP/part01/lesson_18/example_array_class# ./a.out
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
```

拷贝构造函数

两个特殊的构造函数

- 无参数构造函数
  - 当类中没有**定义任何的构造函数**时，**编译器才默认提供一个无参构造函数**，并且其函数体为空。
- 拷贝构造函数
  - 参数为 `const class_name&` 的构造函数，当类中**没有定义拷贝构造函数**时，**编译器默认提供一个拷贝构造函数，简单的进行成员变量的值复制**

拷贝构造函数

拷贝构造函数也是构造函数，拷贝构造函数实际上就是**有参构造函数**，只要参数类型为 `const class_name&`，那它就是拷贝构造函数，
注意，定义了构造函数，同样存在拷贝构造函数，定义了拷贝构造函数，编译器就没有自动提供无参构造函数

```
class Test
{
private:
    int i;
    int j;
public:
    Test(){}
    Test(const Test& t)
    {
        i = t.i;
        j = t.j;
    }

}
```

所以，当**显式定义**拷贝构造。那么就需要手动添加**无参构造函数**或者**其它有参构造函数**

- 拷贝构造函数的意义
  - 兼容C语言的初始化方式
  - 初始化行为能够符合预期的逻辑
- 浅拷贝
  - 拷贝后对象的物理状态相同
- 深拷贝
  - 拷贝后对象的逻辑状态相同

编译器提供的默认拷贝构造函数只进行**浅拷贝**

浅拷贝

```
#include <stdio.h>

class Test
{
```

```
private:
    int i;
    int j;
    int* p;

public:
    int getI()
    {
        return i;
    }

    int getJ()
    {
        return j;
    }

    int* getP()
    {
        return p;
    }

    Test(int v)
    {
        p = new int;
        *p = v;
    }

    Test(const Test& t)
    {
        i = t.i;
        j = t.j;
        p = t.p;
    }

    void free()
    {
        delete p;
    }
};

int main(void)
{

    Test t1(3);
    Test t2 = t1;  // Test t2(t1);

    printf("t1.getI() = %d, t1.getJ() = %d, t1.getP() = %p\n", t1.getI(), t1.getJ(), t1.getP());
    printf("t2.getI() = %d, t2.getJ() = %d, t2.getP() = %p\n", t2.getI(), t2.getJ(), t2.getP());

    // error
    t1.free();
```

```
    t2.free();

    return 0;
}
```

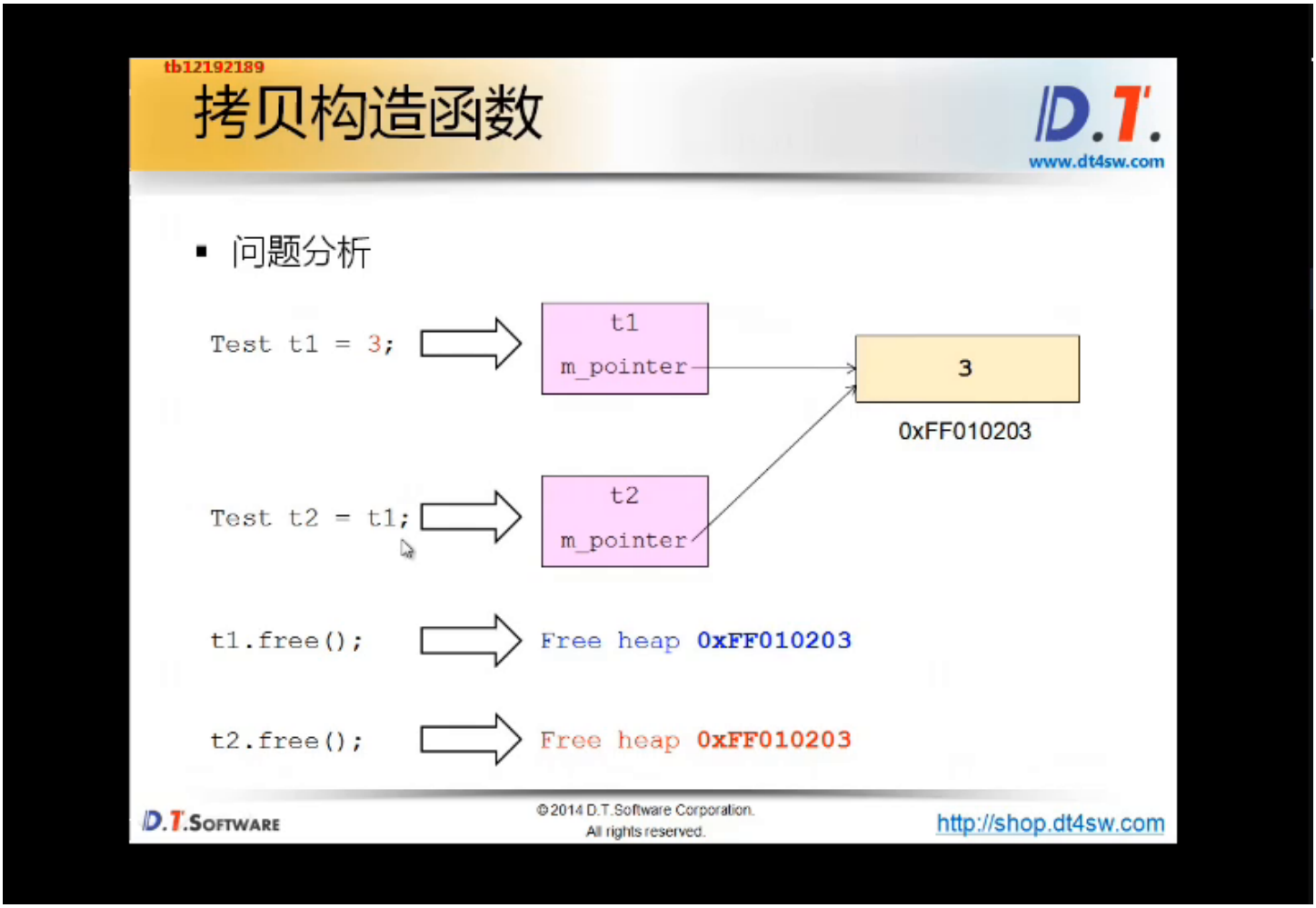
输出

```
root@ubuntu:~/exp/DT_CPP/part01# ./a.out
t1.getI() = 10, t1.getJ() = 0, t1.getP() = 0x5570882e8eb0
t2.getI() = 10, t2.getJ() = 0, t2.getP() = 0x5570882e8eb0
free(): double free detected in tcache 2
Aborted (core dumped)
```

double free，出现两次释放同一块内存

> 对象t1 开辟int大小堆空间，被成员p保存该地址，之后调用默认拷贝构造函数去实例化对象t2，
> 默认拷贝构造函数只是简单将成员变量进行赋值，
> 以致于两个对象t1和t2的成员p指向的同一块内存，又因为是堆空间需要被释放，
> 所以就会对这块空间释放两次，导致出现错误

问题分析



> 产生多次释放同一块内存

深拷贝

```
#include <stdio.h>

class Test
{
private:
    int i;
    int j;
    int* p;

public:
    int getI()
    {
        return i;
    }

    int getJ()
    {
        return j;
    }

    int* getP()
    {
        return p;
    }

    Test(int v)
    {
        p = new int;
        *p = v;
    }

    Test(const Test& t)
    {
        i = t.i;
        j = t.j;
        p = new int;
        *p = *t.p;
    }

    void free()
    {
        delete p;
    }
};

int main(void)
{

    Test t1(3);
    Test t2(t1);
```

```
51
52      printf("t1.getI() = %d, t1.getJ() = %d,t1.getP() = %p, *t1.getP() = %d\n",
    t1.getI(), t1.getJ(), t1.getP(), *t1.getP());
53      printf("t2.getI() = %d, t2.getJ() = %d,t2.getP() = %p, *t2.getP() = %d\n",
    t2.getI(), t2.getJ(), t2.getP(), *t2.getP());
54
55      t1.free();
56      t2.free();
57
58      return 0;
59  }
```

输出

```
1  root@ubuntu:~/exp/DT_CPP/part01# ./a.out
2  t1.getI() = 72704, t1.getJ() = 0,t1.getP() = 0x55dd8a704eb0, *t1.getP() = 3
3  t2.getI() = 72704, t2.getJ() = 0,t2.getP() = 0x55dd8a704ed0, *t2.getP() = 3
```

什么时候需要深拷贝?

- 对象中有成员变量指代了系统中的资源
    i. 成员指向了动态内存空间
    ii. 成员打开了外存中的文件
    iii. 成员使用了系统中的网络端口
    iv. 。。。

一般性原则

自定义拷贝构造函数，必然需要实现深拷贝！！！

编程实验

> 数组类的拷贝构造函数【使用课程中的案例 `lesson_18/example_array_class/*.cpp` 】

```
1  // IntArray.h
2  #ifndef _INTARRAY_H_
3  #define _INRARRAY_H_
4  class IntArray
5  {
6  private:
7      int m_length;
8      int* m_pointer;
9  public:
10     IntArray(int len);
11     IntArray(const IntArray& ref);
```

```
    int length();
    bool get(int index, int& value);
    bool set(int index, int value);
    void free();
};
#endif
```

```
// IntArray.cpp
#include"IntArray.h"
IntArray::IntArray(int len)
{
    m_pointer = new int[len];
    for (int i = 0; i < len; i++)
    {
        m_pointer[i] = 0;
    }
    m_length = len;
}

IntArray::IntArray(const IntArray& ref)
{
    m_length = ref.m_length;
    m_pointer = new int[m_length];
    for (int i = 0; i < m_length; i++)
        *(m_pointer + i) = *(ref.m_pointer + i);
}

int IntArray::length()
{
    return m_length;
}
bool IntArray::get(int index, int& value)
{
    bool ret = (index >= 0 && index < m_length);
    if (ret)
    {
        value = m_pointer[index];
    }
    return ret;
}
bool IntArray::set(int index, int value)
{
    bool ret = (index >= 0 && index < m_length);
    if (ret)
    {
        m_pointer[index] = value;
    }
    return ret;
}
void IntArray::free()
{
```

```
    delete[]m_pointer;
}
```

```
// main.cpp
#include<stdio.h>
#include"IntArray.h"
int main()
{
    IntArray a(5);

    for (int i = 0; i < a.length(); i++)
    {
        a.set(i, i+1);
    }

    IntArray b = a;

    for (int i = 0; i < a.length(); i++)
    {
        int value = 0;
        if (a.get(i, value))
        {
            printf("a[%d] = %d\n", i, value);
        }
    }
    a.free();

    putchar('\n');
    for (int i = 0; i < b.length(); i++)
    {
        int value = 0;
        if (b.get(i, value))
        {
            printf("b[%d] = %d\n", i, value);
        }
    }
    b.free();

    return 0;
}
```

> 以上程序深拷贝，每个数组元素的值是一样的

输出

```
root@ubuntu:~/exp/DT_CPP/part01/lesson_19_for_array_class# ./a.out
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
```

```
a[4] = 5

b[0] = 1
b[1] = 2
b[2] = 3
b[3] = 4
b[4] = 5
```

小结

- C++编译器会默认提供构造函数
- 无参构造函数用于定义对象的默认初始状态
- 拷贝构造函数在创建对象时拷贝对象的状态
- 对象的拷贝有`浅拷贝`和`深拷贝`两种方式
  - 浅拷贝使得对象的物理状态相同
  - 深拷贝使得对象的逻辑状态相同

Other

C++定义一个空类有哪些成员函数

1. 缺省构造函数。
2. 缺省拷贝构造函数。
3. 缺省析构函数。
4. 缺省赋值运算符。
5. 缺省取址运算符。
6. 缺省取址运算符 const。

补充

调用拷贝构造函数主要有以下场景：

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

class Student
{
    const char* name;
    int age;
    double weight;
    public:
```

```
    Student(const char* name, int age, double weight)
    {
        cout << "constructor..." << endl;
        this->name = name;
        this->age = age;
        this->weight = weight;
    }

    Student(const Student &ref)
    {
        cout << "copy constructor..." << endl;
        this->name = ref.name;
        this->age = ref.age;
        this->weight = ref.weight;
    }
};
```

- 对象作为`函数的参数`，**以值传递的方式传给函数。**

```
void func(Student s){
//TODO:
}

int main(void)
{
    Student stu("小明", 16, 90.5);  //普通初始化
    func(stu);  //以拷贝的方式初始化
}
```

- 使用一个**已经存在的对象**给另一个对象初始化，且**有新的对象产生**

```
int main(void)
{
    Student stu1("小明", 16, 90.5);  //普通初始化
    Student stu2 = stu1;  //以拷贝的方式初始化
}
```

- 对象作为`函数的返回值`，**以值的方式从函数返回**

```
Student func(){
    Student s("小明", 16, 90.5);
    return s;
}

int main(void)
{
    Student stu = func();
}
```

现代编译器上，只会调用一次拷贝构造函数，或者一次也不调用.
这是因为，现代编译器都支持返回值优化技术，会**尽量避免拷贝对象**，以提高程序运行效率。