

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

```
1 class Test
2 {
3 private:
4     int i;
5     int j;
6 public:
7     int getI() {return i;}
8     int getJ() {return j;}
9 };
```

lesson_17-1_main.cpp

```
1 #include <stdio.h>
2 #include "lesson_17-1.h"
3
4 // 全局对象
5 Test gt;
6
7 int main(void)
8 {
9     // 局部对象
10    Test t;
11    // 全局对象输出
12    printf("gt.getI() = %d\n", gt.getI());
13    printf("gt.getJ() = %d\n", gt.getJ());
14
15    // 局部对象输出
16    printf("t.getI() = %d\n", t.getI());
17    printf("t.getJ() = %d\n", t.getJ());
18
19    Test* pt = new Test;
20    printf("pt->getI() = %d\n", pt->getI());
21    printf("pt->getJ() = %d\n", pt->getJ());
22
23    delete pt;
24
25    return 0;
26 }
```

输出

```
1 root@ubuntu:~/exp/DT_CPP/part01# ./a.out
2 gt.getI() = 0
3 gt.getJ() = 0
4 t.getI() = -1323098096
5 t.getJ() = 32766
6 pt->getI() = 0
7 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

```
1 class Test
2 {
3 private:
4     int i;
5     int j;
6 public:
7     void initialize()
8     {
9         i = 0;
10        j = 0;
11    }
12    int getI() {return i;}
13    int getJ() {return j;}
14};
```

lesson_17-2_main.cpp

```

1  #include <stdio.h>
2  #include "lesson_17-2.h"
3
4  // 全局对象
5  Test gt;
6
7  int main(void)
8  {
9      // 局部对象
10     Test t;
11     // 全局对象输出
12     gt.initialize();
13     printf("gt.getI() = %d\n", gt.getI());
14     printf("gt.getJ() = %d\n", gt.getJ());
15
16     // 局部对象输出
17     t.initialize();
18     printf("t.getI() = %d\n", t.getI());
19     printf("t.getJ() = %d\n", t.getJ());
20
21     Test* pt = new Test;
22     pt->initialize();
23     printf("pt->getI() = %d\n", pt->getI());
24     printf("pt->getJ() = %d\n", pt->getJ());
25
26     delete pt;
27
28     return 0;
29 }

```

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

```

1  class Test
2  {
3  private:
4      int i;
5      int j;

```

```

6 public:
7     /*void initialize()
8     {
9         i = 0;
10        j = 0;
11    }*/
12    Test()
13    {
14        i = 0;
15        j = 0;
16    }
17    int getI() {return i;}
18    int getJ() {return j;}
19 };

```

lesson_17-3_main.cpp

```

1  #include <stdio.h>
2  #include "lesson_17-3.h"
3
4  // 全局对象
5  Test gt;
6
7  int main(void)
8  {
9      // 局部对象
10     Test t;
11     // 全局对象输出
12
13     printf("gt.getI() = %d\n", gt.getI());
14     printf("gt.getJ() = %d\n", gt.getJ());
15
16     // 局部对象输出
17
18     printf("t.getI() = %d\n", t.getI());
19     printf("t.getJ() = %d\n", t.getJ());
20
21     Test* pt = new Test;
22
23     printf("pt->getI() = %d\n", pt->getI());
24     printf("pt->getJ() = %d\n", pt->getJ());
25
26     delete pt;
27
28     return 0;
29 }

```

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

```
1 Test t; // 定义对象并调用构造函数
2 int main(void)
3 {
4     extern Test t; // 告诉编译器存在名为 t的Test对象
5     return 0;
6 }
```

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

```
1 #include <stdio.h>
2
3 class Test
4 {
5 public:
6     Test()
7     {
8         printf("call Test()\n");
9     }
10    Test(int v)
11    {
12        printf("call Test(int v), v = %d\n", v);
13    }
14 };
15
```

```

16 int main(void)
17 {
18     Test t;        // 调用Test()
19     Test t1(1);   // 调用Test(int v)
20     Test t2 = 2;  // 调用 Test(int v)
21
22     int i(100);
23     printf("i = %d\n", i);
24
25     return 0;
26 }

```

输出

```

1 root@ubuntu:~/exp/DT_CPP/part01# ./a.out
2 call Test()
3 call Test(int v), v = 1
4 call Test(int v), v = 2
5 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

```

1 #include<stdio.h>
2 class Test
3 {
4 private:
5     int m_value;
6 public:
7     Test()
8     {
9         m_value = 0;
10        printf("Test()\n");
11    }
12    Test(int v)
13    {
14        m_value = v;
15        printf("Test(v), v = %d\n", v);
16    }
17    int getvalue()
18    {
19        return m_value;

```

```

20     }
21 };
22 int main()
23 {
24     Test ta[3] = {Test(), Test(1), Test(2)}; // call
25     for (int i = 0; i < 3; i++)
26     {
27         printf("ta[%d].getvalue() = %d\n", i, ta[i].getvalue());
28     }
29     Test t = Test(100); // call
30     printf("t.getvalue() = %d\n", t.getvalue());
31     return 0;
32 }

```

输出

```

1 root@ubuntu:~/exp/DT_CPP/part01# ./a.out
2 Test()
3 Test(v), v = 1
4 Test(v), v = 2
5 ta[0].getvalue() = 0
6 ta[1].getvalue() = 1
7 ta[2].getvalue() = 2
8 Test(v), v = 100
9 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

```

1 #include <stdio.h>
2
3 class Array
4 {
5 private:
6     int* ptr = NULL;
7     int ArrayLen;
8

```

```

9 public:
10     Array();
11     Array(int);
12     ~Array();
13     int getLength();
14     int getElement(int);
15     bool setElement(int, int);
16 };

```

lesson_18-3.cpp

```

1 #include "lesson_18-3.h"
2 void ArrayInit(int* ptr, int len)
3 {
4     for (int i = 0; i < len; i++)
5         *(ptr + i) = 0;
6 }
7 Array::~~Array()
8 {
9     if (ptr != NULL)
10        delete[] ptr;
11 }
12 Array::Array()
13 {
14     ptr = new int[10];
15     ArrayLen = 10;
16     ::ArrayInit(ptr, 10);
17 }
18 Array::Array(int len)
19 {
20     ptr = new int[len];
21     ArrayLen = len;
22     ::ArrayInit(ptr, len);
23 }
24
25 int Array::getLength()
26 {
27     return ArrayLen;
28 }
29
30 int Array::getElement(int index)
31 {
32     if (index < ArrayLen)
33         return ptr[index];
34
35     return -1;
36 }
37
38 bool Array::setElement(int index, int value)
39 {
40     if (index < ArrayLen)

```



```

41     ptr[index] = value;
42     return true;
43     return false;
44 }

```

lesson_18-3_main.cpp

```

1  #include <stdio.h>
2  #include "lesson_18-3.h"
3
4  int main(void)
5  {
6      Array* pa = new Array; // 写法1
7      Array pa1(30); // 调用有参构造
8      pa->setElement(4, 99);
9      printf("pa->getLength() = %d\n", pa->getLength()); // 10
10     printf("pa->getElement(4) = %d\n", pa->getElement(4)); // 99
11     printf("pa->getElement(10) = %d\n", pa->getElement(10)); // -1
12
13     pa1.setElement(4, 77);
14     printf("pa1.getLength() = %d\n", pa1.getLength()); // 30
15     printf("pa1.getElement(4) = %d\n", pa1.getElement(4)); // 99
16     printf("pa1.getElement(10) = %d\n", pa1.getElement(10)); // -1
17
18     delete pa;
19
20     return 0;
21 }

```

课程实例

```

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     int length();
12     bool get(int index, int& value);
13     bool set(int index, int value);
14     void free();
15 };
16 #endif

```

```

1 // IntArray.cpp
2 #include "IntArray.h"
3 IntArray::IntArray(int len)
4 {
5     m_pointer = new int[len];
6     for (int i = 0; i < len; i++)
7     {
8         m_pointer[i] = 0;
9     }
10    m_length = len;
11 }
12 int IntArray::length()
13 {
14     return m_length;
15 }
16 bool IntArray::get(int index, int& value)
17 {
18     bool ret = (index >= 0 && index < m_length);
19     if (ret)
20     {
21         value = m_pointer[index];
22     }
23     return ret;
24 }
25 bool IntArray::set(int index, int value)
26 {
27     bool ret = (index >= 0 && index < m_length);
28     if (ret)
29     {
30         m_pointer[index] = value;
31     }
32     return ret;
33 }
34 void IntArray::free()
35 {
36     delete[] m_pointer;
37 }

```

```

1 // 18-3.cpp
2 #include <stdio.h>
3 #include "IntArray.h"
4 int main()
5 {
6     IntArray a(5);
7     for (int i = 0; i < a.length(); i++)
8     {
9         a.set(i, i+1);
10    }
11    for (int i = 0; i < a.length(); i++)
12    {

```

```
13     int value = 0;
14     if (a.get(i, value))
15     {
16         printf("a[%d] = %d\n", i, value);
17     }
18 }
19 a.free();
20 return 0;
21 }
```

```
1 root@ubuntu:~/exp/DT_CPP/part01/lesson_18/example_array_class# g++ 18-3.cpp
  IntArray.cpp
2 root@ubuntu:~/exp/DT_CPP/part01/lesson_18/example_array_class# ./a.out
3 a[0] = 1
4 a[1] = 2
5 a[2] = 3
6 a[3] = 4
7 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

```
1  class Test
2  {
3  private:
4      int i;
5      int j;
6  public:
7      Test(){}
8      Test(const Test& t)
9      {
10         i = t.i;
11         j = t.j;
12     }
13
14 }
```

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

```
1  #include <stdio.h>
2
3  class Test
4  {
```

```

5 private:
6     int i;
7     int j;
8     int* p;
9
10 public:
11     int getI()
12     {
13         return i;
14     }
15
16     int getJ()
17     {
18         return j;
19     }
20
21     int* getP()
22     {
23         return p;
24     }
25
26     Test(int v)
27     {
28         p = new int;
29         *p = v;
30     }
31
32     Test(const Test& t)
33     {
34         i = t.i;
35         j = t.j;
36         p = t.p;
37     }
38
39     void free()
40     {
41         delete p;
42     }
43 };
44
45 int main(void)
46 {
47
48     Test t1(3);
49     Test t2 = t1; // Test t2(t1);
50
51     printf("t1.getI() = %d, t1.getJ() = %d, t1.getP() = %p\n", t1.getI(), t1.getJ(),
t1.getP());
52     printf("t2.getI() = %d, t2.getJ() = %d, t2.getP() = %p\n", t2.getI(), t2.getJ(),
t2.getP());
53
54     // error
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() = 10, t1.getJ() = 0, t1.getP() = 0x5570882e8eb0
3 t2.getI() = 10, t2.getJ() = 0, t2.getP() = 0x5570882e8eb0
4 free(): double free detected in tcache 2
5 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

tb12192189

拷贝构造函数

D.T. www.dt4sw.com

- 问题分析

```
Test t1 = 3; → [t1 m_pointer] → [3 0xFF010203]
```

```
Test t2 = t1; → [t2 m_pointer] → [3 0xFF010203]
```

```
t1.free(); → Free heap 0xFF010203
```

```
t2.free(); → Free heap 0xFF010203
```

D.T. SOFTWARE © 2014 D.T. Software Corporation. All rights reserved. <http://shop.dt4sw.com>

产生多次释放同一块内存

深拷贝

```
1  #include <stdio.h>
2
3  class Test
4  {
5  private:
6      int i;
7      int j;
8      int* p;
9
10 public:
11     int getI()
12     {
13         return i;
14     }
15
16     int getJ()
17     {
18         return j;
19     }
20
21     int* getP()
22     {
23         return p;
24     }
25
26     Test(int v)
27     {
28         p = new int;
29         *p = v;
30     }
31
32     Test(const Test& t)
33     {
34         i = t.i;
35         j = t.j;
36         p = new int;
37         *p = *t.p;
38     }
39
40     void free()
41     {
42         delete p;
43     }
44 };
45
46 int main(void)
47 {
48
49     Test t1(3);
50     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);

```



```
12     int length();
13     bool get(int index, int& value);
14     bool set(int index, int value);
15     void free();
16 };
17 #endif
```

```
1 // IntArray.cpp
2 #include "IntArray.h"
3 IntArray::IntArray(int len)
4 {
5     m_pointer = new int[len];
6     for (int i = 0; i < len; i++)
7     {
8         m_pointer[i] = 0;
9     }
10    m_length = len;
11 }
12
13 IntArray::IntArray(const IntArray& ref)
14 {
15     m_length = ref.m_length;
16     m_pointer = new int[m_length];
17     for (int i = 0; i < m_length; i++)
18         *(m_pointer + i) = *(ref.m_pointer + i);
19 }
20
21 int IntArray::length()
22 {
23     return m_length;
24 }
25 bool IntArray::get(int index, int& value)
26 {
27     bool ret = (index >= 0 && index < m_length);
28     if (ret)
29     {
30         value = m_pointer[index];
31     }
32     return ret;
33 }
34 bool IntArray::set(int index, int value)
35 {
36     bool ret = (index >= 0 && index < m_length);
37     if (ret)
38     {
39         m_pointer[index] = value;
40     }
41     return ret;
42 }
43 void IntArray::free()
44 {
```

```
45     delete[] m_pointer;
46 }
```

```
1 // main.cpp
2 #include<stdio.h>
3 #include"IntArray.h"
4 int main()
5 {
6     IntArray a(5);
7
8     for (int i = 0; i < a.length(); i++)
9     {
10         a.set(i, i+1);
11     }
12
13     IntArray b = a;
14
15     for (int i = 0; i < a.length(); i++)
16     {
17         int value = 0;
18         if (a.get(i, value))
19         {
20             printf("a[%d] = %d\n", i, value);
21         }
22     }
23     a.free();
24
25     putchar('\n');
26     for (int i = 0; i < b.length(); i++)
27     {
28         int value = 0;
29         if (b.get(i, value))
30         {
31             printf("b[%d] = %d\n", i, value);
32         }
33     }
34     b.free();
35
36     return 0;
37 }
```

以上程序深拷贝，每个数组元素的值是一样的

输出

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

```
6 | a[4] = 5
7 |
8 | b[0] = 1
9 | b[1] = 2
10 | b[2] = 3
11 | b[3] = 4
12 | 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：

```
1 | #include <iostream>
2 | using namespace std;
3 |
4 | class Student
5 | {
6 |     const char* name;
7 |     int age;
8 |     double weight;
9 |     public:
```

```

10 Student(const char* name, int age, double weight)
11 {
12     cout << "constructor..." << endl;
13     this->name = name;
14     this->age = age;
15     this->weight = weight;
16 }
17
18 Student(const Student &ref)
19 {
20     cout << "copy constructor..." << endl;
21     this->name = ref.name;
22     this->age = ref.age;
23     this->weight = ref.weight;
24 }
25 };

```

- 对象作为 函数的参数 ， 以值传递的方式传给函数。

```

1 void func(Student s){
2 //TODO:
3 }
4
5 int main(void)
6 {
7     Student stu("小明", 16, 90.5); //普通初始化
8     func(stu); //以拷贝的方式初始化
9 }

```

- 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

```

1 int main(void)
2 {
3     Student stu1("小明", 16, 90.5); //普通初始化
4     Student stu2 = stu1; //以拷贝的方式初始化
5 }

```

- 对象作为 函数的返回值 ， 以值的方式从函数返回

```

1 Student func(){
2     Student s("小明", 16, 90.5);
3     return s;
4 }
5
6 int main(void)
7 {
8     Student stu = func();
9 }

```

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。