# C++第三讲

## 一、c++中类的结构体的区别

### 1.1 区别内容

唯一区别: 默认权限不同 类的默认权限: private 结构体默认权限: public

### 1.2 使用场景

当涉及到数据结构中比如节点等这些需要使用结构体,因为成员必须要在外部 访问,所有应该设为公有

## 二、this指针

### 2.1 this指针的内涵

指代该对象本身, 是类中隐藏的指针, 哪个对象使用我, 我就指代哪个对象

### 2.2 this指针的格式

类名 \* const this;

### 2.3 必须使用this指针的场景

- 1> 当成员函数的形参名和成员变量名同名时,可以用this指针区分,可以用初始化列表解决
  - 2> 在拷贝赋值函数中,要返回自身引用时,要用this指针(后期会讲)

```
#include <iostream>

using namespace std;

class Stu
{
 private:
    int age;
    string name;
    int score;
```

```
11
   public:
12
       void init(int age, string name, int score )
13
14
15
            this->age = age;
16
            this->name = name;
17
           this->score = score;
18 //
             age = age;
19 //
             name = name;
20 //
             score = score;
21
      }
22
23
      void show()
24
       {
25
            cout<<age<<endl;</pre>
            cout<<name<<end1;</pre>
26
27
            cout<<score<<endl;</pre>
28
        }
29
30
   };
31
32
   int main()
33
34 Stu s1;
       s1.init(18,"zhangpp", 99);
35
       s1.show();
36
       return 0;
37
38
   }
39
```

# 三、类中特殊的成员函数 (重中之重)

在类中有四个常用的特殊的成员函数:

- 1> 如果用户不显性定义,系统会提供一个默认版本,但是如果用户显性定义了,默认提供的版本就没有了。
- 2> 这些特殊的成员函数,无论是系统提供的还是用户自己定义的,都无需手动调用,系统在对应位置会自动调用

### 3.1 构造函数

### 3.1.1 功能

用类去实例化对象时,为对象申请资源和初始化用的

### 3.1.2 格式

```
      1
      1、没有返回值

      2
      2、函数名与类同名

      3
      3、访问权限一般为public

      4
      格式: 类名(参数列表) {}
```

### 3.1.3 调用时机

用类实例化对象的过程中,系统自动调用,无需手动调用

1> 栈区实例化对象的时候自动调用 类名 对象名 (实参列表);

2> 堆区: 只有申请空间时调用, 定义指针时不调用

类名\*指针名; //此时不调用构造函数

指针名 = new 类名 (实参); //此时调用构造函数

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Stu
6 {
7
  private:
8
      string name;
       int age;
9
      double score;
10
11
12
   public:
       Stu()
                //定义无参构造
13
      {
14
           name = "";
15
16
           age = 0;
          score = 0;
17
          cout<<"无参构造"<<end1;
18
19
       }
20
21
      //定义有参构造
       Stu(string n, int a, double s = 100)
22
```

```
23
          this->name = n;
24
          this->age = a;
25
          this->score = s;
26
27
          cout<<"有参构造"<<end1;
      }
28
29
30
     //展示函数
31
     void show()
32
          cout<<"name = "<<name<<" age = "<<age<<" score =</pre>
33
   "<<score<<endl;
34
      }
  };
35
36
37
  int main()
38
39
40
   Stu s1;
                       //调用无参构造
    s1.show();
41
42
43
    Stu s2("zhangpp",18); //调用自定义的有参构造
44
     s2.show();
45
     /***************
46
               //此时不会调用构造函数
47
     Stu *p1;
     p1 = new Stu;
                    //调用无参构造
48
49
     p1->show();
50
     Stu *p2 = new Stu("张三", 20, 88.5); //调用有参构造
51
52
     p2->show();
53
54
     return 0;
55 }
56
```

### 3.1.4 构造函数支持重载

- 1> 当类中没有显性定义构造函数(包括有参和无参),系统会默认提供一个无参构造,来完成对象资源的构造
- 2> 当类中显性定义了构造函数,系统就不再提供默认的构造函数了,如果还想使用无参构造函数,需要手动定义一个无参构造
  - 3> 一个类中可以定义多个构造函数,这些函数构成重载关系

```
3 using namespace std;
 4
 5 class Stu
 6
 7 private:
      string name;
 8
 9
      int age;
      double score;
10
11
12 public:
13 // Stu()
                //定义无参构造
14 //
       {
15 // name = "";
16 // age = 0:
17 // score = 0;
18 //
          cout<<"无参构造"<<end1;
19 // }
20
   //定义有参构造
21
      Stu(string n, int a, double s = 100)
22
23
      {
24
          this->name = n;
25
          this->age = a;
26
          this->score = s;
          cout<<"有参构造"<<end1;
27
    }
28
29
    //展示函数
30
      void show()
31
32
          cout<<"name = "<<name<<" age = "<<age<<" score =</pre>
33
   "<<score<<endl;
      }
34
35
  };
36
37
38 int main()
39 | {
                       //调用无参构造 因为显性定义了有参构造,默认的
40
      Stu s1;
   无参构造就不再提供了, 所以报错
41
      s1.show();
42
43 return 0;
  }
44
```

### 3.1.5 构造函数初始化列表

在类中,可以使用构造函数的初始化列表来完成对成员变量的初始化,此时,对成员的初始化,可以放到初始化列表中,函数体内可以执行相关逻辑代码格式:类名(形参1,形参2):成员变量1(形参1),成员变量2(形参2){}

注意: 不要办把成员变量和形参位置写反了

### 3.1.5 必须使用初始化列表的情况

1> 当构造函数的形参名和成员变量名同名时,可以用初始化列表来完成初始 化

```
1 #include <iostream>
 2
 3
   using namespace std;
 4
 5
   class Stu
 6
    {
7
   private:
 8
       string name;
9
       int age;
10
       double score;
11
12
    public:
13
        Stu()
                         //定义无参构造
        {
14
15
            name = "";
16
            age = 0;
            score = 0;
17
18
            cout<<"无参构造"<<end1;
19
        }
20
        //定义有参构造
21
22
        Stu(string name, int age, double
    score):name(name),age(age),score(score)
23
        {
            cout<<"有参构造"<<end1;
24
25
        }
26
27
       //展示函数
       void show()
28
29
```

```
cout<<"name = "<<name<<" age = "<<age<<" score =</pre>
30
    "<<score<<end1;</pre>
        }
31
   };
32
33
34
   int main()
35
36
   {
37
        Stu s1("zhangpp", 18, 99);
38
        s1.show();
39
40
       return 0;
41
   }
42
```

2> 当类的成员变量是引用成员时,对该成员必须使用初始化列表来完成初始化

```
#include <iostream>
 2
 3
   using namespace std;
 4
 5
   class Stu
 6
   {
 7
   private:
8
      string name;
9
      int age;
       double score;
10
      int #
                      //引用成员
11
12
   public:
13
14 // Stu()
                   //定义无参构造
15
   //
        {
16 //
        name = "";
17 //
           age = 0;
18 //
            score = 0;
            cout<<"无参构造"<<end1;
19 //
20 //
        }
21
22
       //定义有参构造
       Stu(string name, int age, double score, int &n):num(n)
23
24
25
           this->name = name;
26
           this->age = age;
           this->score = score;
27
28
           //this->num = n;
```

```
cout<<"有参构造"<<end1;
29
        }
30
31
        //展示函数
32
33
        void show()
34
             cout<<"name = "<<name<<" age = "<<age<<" score =</pre>
35
    "<<score<<endl;</pre>
            cout<<"num = "<<num<<end1;</pre>
36
37
        }
    };
38
39
40
    int main()
41
42
   {
43
        int num = 1001;
        Stu s1("zhangpp", 18, 99, num);
44
45
        s1.show();
46
47
        return 0;
48 }
```

3> 当类中的成员是其他类的子对象时,对该成员的初始化,也要用初始化列表完成

```
1 #include <iostream>
 2
 3
   using namespace std;
 4
 5
   class Stu
 6
    {
7
    private:
8
        string name;
 9
        int score;
10
11
    public:
        Stu() {cout<<"Stu::无参构造"<<end1;}
12
        Stu(string n, int s):name(n), score(s)
13
        {cout<<"Stu::有参构造"<<end1;}
14
15
16
        void show()
        {
17
            cout<<"stu::name = "<<name<<endl;</pre>
18
            cout<<"stu::score = "<<score<<endl;</pre>
19
        }
20
```

```
21 | };
22
23
   class Teacher
24 {
25
       string name;
       string subj;
26
27
       Stu student;
                         //其他类的子对象
28
   public:
       Teacher(string n1, string s1, string n2, int s2):name(n1),
29
    subj(s1),student(n2, s2)
       {
30
           //this->name = n1;
31
32
           //this->subj = s1;
           //this->student = Stu(n2, s2); //调用Stu类的构造函数
33
34
       }
35
       void show()
36
37
       {
            cout<<"teacher::name = "<<name<<endl;</pre>
38
            cout<<"teacher::subj = "<<subj<<endl;</pre>
39
           //cout<<"teacher::student::name = "</pre>
40
   <<student.name<<endl;
41
           this->student.show(); //调用成员对象中的函数
       }
42
   };
43
44
45
46 int main()
47 {
       Teacher t1("zhangpp", "C++", "zhangsan", 99);
48
       t1.show();
49
       return 0;
50
51
   }
52
```

## 3.2 析构函数

### 3.2.1 功能

在对象消亡时,用来回收空间和资源用的

#### 3.2.2 格式

```
      1
      1、没有返回值

      2
      2、函数名: ~类名

      3
      3、权限: 一般为public

      4
      4、没有参数

      5
      5、格式: ~类名() {}
```

### 3.2.3 调用时机

在对象消亡时, 系统自动调用

栈区: 随着程序结束, 自动调用析构函数

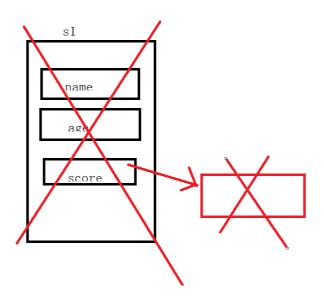
堆区: 在使用delete关键字时, 系统自动调用析构

```
1 #include <iostream>
2
 3 using namespace std;
 4
 5 class Stu
 6 {
7
  private:
       string name;
8
9
      int age;
10
       double score;
11
12
   public:
       Stu()
13
14
       {
15
           cout<<"无参构造"<<end1;
16
       }
       Stu(string n, int a, double s):name(n), age(a), score(s)
17
18
      {
           cout<<"有参构造"<<end1;
19
20
       }
      ~Stu()
21
22
      {
           cout<<"析构函数"<<" "<<this<<endl;
23
24
25 };
26
27
28
29 int main()
30 {
```

```
31
       Stu s1; //调用无参构造
32
       cout<<&s1<<endl;</pre>
33
34
            Stu *p = new Stu;
35
            cout<<p<<end1;</pre>
                                 //释放空间时,自动调用析构函数
36
            delete p;
37
38
       Stu s2("zhangpp", 18, 99);
39
       cout<<&s2<<end1;</pre>
40
41
      return 0;
42 }
43
```

### 3.2.4 默认析构函数

- 1> 如果类中没有显性定义析构函数,系统会默认提供一个析构函数,完成对类对象空间的销毁
  - 2> 如果显性定义了析构函数,系统就不再提供默认的析构函数了



### 3.2.5 构造函数和析构函数调用顺序

栈区: 先构造的后析构, 后构造的先析构

堆区:使用new时调用构造函数,使用delete时调用析构函数

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

class Stu
{
private:
```

```
string name;
9
       int age;
10
       double *score;
11
12 public:
13
       Stu()
14
       {
15
           cout<<"无参构造"<<end1;
16
       }
17
       Stu(string n, int a, double s):name(n), age(a)
18
       {
19
           score = new double(s);
20
21
           cout<<"有参构造"<<end1;
22
       }
23
       ~Stu()
24
      {
25
           delete score;
           cout<<"析构函数"<<" "<<this<<endl;
26
27
       }
28 };
29
30
31
32 int main()
33 {
                  //调用无参构造
34
      Stu s1;
35
      cout << \&s1 << end1;
36
           Stu *p = new Stu;
37
38
           cout<<p<<endl;</pre>
                                //释放空间时,自动调用析构函数
39
           delete p;
40
       Stu s2("zhangpp", 18, 99);
41
       cout<<&s2<<end1;</pre>
42
43
44
      return 0;
45 }
46
```

### 3.2.6 构造函数和析构函数个数

一个类中可以有多个构造函数,但是,只能有一个析构函数

### 3.3 拷贝构造函数

### 3.3.1 功能

是一种特殊的构造函数,用来完成用一个类对象给另一个类对象初始化用的

### 3.3.2 格式

```
      1
      1、没有返回值

      2
      2、函数名与类同名

      3
      3、参数: 同类的其他类对象

      4
      4、public的访问权限

      5
      5、格式

      6
      类名(const 类名 &)
```

### 3.3.3 调用时机

1、用一个对象给另一个对象初始化时自动调用

```
      string s1("hello");
      //调用有参构造

      string s2(s1);
      //调用拷贝构造

      string s3 = s1;
      //拷贝构造
```

```
1 #include <iostream>
 2
 3 using namespace std;
 4
 5 class Stu
 6 {
 7
   private:
       string name;
8
9
       int age;
10
   public:
11
       Stu()
12
13
           cout<<"无参构造"<<end1;
14
15
       Stu(string n, int a):name(n), age(a)
16
17
       {
            cout<<"有参构造"<<end1;
18
19
       }
```

```
~Stu()
20
21
       {
22
           cout<<"析构函数"<<end1;
23
       }
24
25
       //定义拷贝构造函数
       Stu(const Stu &other):name(other.name), age(other.age)
26
27
28
           cout<<"拷贝构造"<<end1;
29
       }
30
      void show()
31
32
      {
           cout<<"name = "<<name<<" age = "<<age<<endl;</pre>
33
34
       }
35
36
  };
37
38
39
  int main()
40
41 {
42
      Stu s1("zhangs", 18); //有参构造
      s1.show();
43
44
    Stu s2(s1);
45
                             //拷贝构造
    s2.show();
46
47
                      //拷贝构造
48
      Stu s3 = s1;
      s3.show();
49
50
51
52
53
      return 0;
54
   }
55
```

- 2> 函数值传递时,实参取代形参过程中会调用拷贝构造函数
- 3> 函数返回值返回类对象类型时,会调用拷贝构造函数

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

class Stu
```

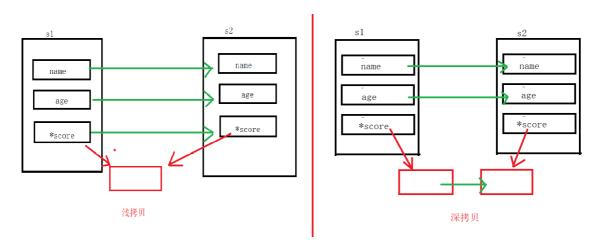
```
6 {
7
   private:
       string name;
8
9
       int age;
10
   public:
11
12
       void set_age()
13
       {
14
           age++;
15
       }
16
17
       Stu()
18
       {
19
           cout<<"无参构造"<<end1;
20
       }
21
       Stu(string n, int a):name(n), age(a)
22
       {
23
           cout<<"有参构造"<<end1;
24
       }
25
       ~Stu()
26
       {
27
           cout<<"析构函数"<<end1;
28
       }
29
30
       //定义拷贝构造函数
       Stu(const Stu &other):name(other.name), age(other.age)
31
32
33
           cout<<"拷贝构造"<<end1;
34
       }
35
       void show()
36
37
           cout<<"name = "<<name<<" age = "<<age<<endl;</pre>
38
39
       }
40
41 };
42
   //定义函数实现,传过来一个对象,将该对象中的年龄增加一岁后返回
43
   Stu fun(Stu s)
44
   {
45
46
47
       s.set_age();
48
49
       return s;
50
   }
```

```
51
   int main()
52
53
    {
        Stu s1("zhangs", 18); //有参构造
54
55
        s1.show();
56
57
        fun(s1).show();
58
        s1.show();
59
60
        return 0;
   }
61
62
```

### 3.3.4 默认拷贝构造函数

如果没有显性定义拷贝构造函数,系统会默认提供一个拷贝构造函数,但是 如果显性定义了拷贝构造函数,系统就不再提供默认的拷贝构造函数了

### 3.3.4 深拷贝和浅拷贝



```
1 #include <iostream>
 2
   using namespace std;
 3
 4
 5
   class Stu
 6
7
   private:
       string name;
 8
9
       int age;
10
       int *score;
11
   public:
12
13
       void set_age()
14
       {
15
            age++;
        }
16
17
18
       Stu()
19
        {
20
            cout<<"无参构造"<<end1;
21
        }
22
       Stu(string n, int a, int s):name(n), age(a)
        {
23
            score = new int(s);
24
            cout<<"有参构造"<<end1;
25
26
        }
27
       ~Stu()
28
       {
29
            delete score;
30
            cout<<"析构函数"<<end1;
        }
31
32
33
       //定义拷贝构造函数
34
        Stu(const Stu &other):name(other.name),
   age(other.age),score(other.score)
        {
35
            cout<<"拷贝构造"<<end1;
36
37
        }
38
39
       void show()
40
            cout<<"name = "<<name<<" age = "<<age<<endl;</pre>
41
42
   };
43
44
```

```
45
46 int main()
47 {
       Stu s1("zhangs", 18, 99); //有参构造
48
49
       s1.show();
50
51
52
      Stu s2(s1);
      s2.show();
53
54
55
  return 0;
56 }
57
```

#### 深拷贝

```
1
   #include <iostream>
2
 3
4 using namespace std;
 5
6 class Stu
7
8
   private:
9
       string name;
10
       int age;
       int *score;
11
12
13
   public:
14
       void set_age()
15
       {
16
           age++;
17
       }
18
        Stu()
19
20
        {
21
            cout<<"无参构造"<<end1;
22
        }
23
       Stu(string n, int a, int s):name(n), age(a)
24
        {
25
            score = new int(s);
           cout<<"有参构造"<<end1;
26
27
        }
       ~Stu()
28
29
        {
```

```
30
            delete score:
            cout<<"析构函数"<<end1;
31
        }
32
33
34
35
       //定义拷贝构造函数
        Stu(const Stu &other):name(other.name),
36
    age(other.age),score( new int( *(other.score) ) )
37
        {
            cout<<"拷贝构造"<<end1;
38
39
        }
40
       void show()
41
42
            cout<<"name = "<<name<<" age = "<<age<<" *score =</pre>
43
   "<< *score<<endl;
            cout<<score<<endl;</pre>
44
45
        }
46
47
   };
48
49
   int main()
50
       Stu s1("zhangs", 18, 99); //有参构造
51
52
       s1.show();
53
54
       Stu s2(s1);
55
       s2.show();
56
57
      return 0;
58 }
```

#### 深浅拷贝问题(笔试面试题)

如果一个类中,没有显性给定拷贝构造函数,系统会默认提供一个拷贝构造函数,来完成类对象之间的简单赋值,这是一个浅拷贝,当类中没有指针成员时,使用浅拷贝是没有问题的。

但是,如果类中有指针成员时,就会出现double free的段错误,原因是,两个类对象的指针成员,同时指向同一片内存空间,当把其中一个对象进行析构时,该指针成员指向的空间就被释放掉了,在析构另一个对象时,同一片空间就会释放两次,造成段错误。

此时,需要进行深拷贝:需要显性定义拷贝构造函数,在拷贝构造函数的初始 化列表中,对指针成员重新申请空间,只需要将原指针空间中的值赋值给新申请的空间,即可完成深拷贝。

### 3.4 拷贝赋值函数

### 3.4.1 功能

这是一个运算符重载函数,也被称为等号运算符重载,是完成用一个类对象给 另一个类对象**赋值**用的

### 3.4.2 格式

```
      1
      1、返回值: 自身的引用

      2
      2、函数名: operator=

      3
      3、参数: 同类的其他对象

      4
      4、格式:

      5
      成员函数版: 类名 & operator=(const 类名& other){}

      6
      全局函数版: 类名 & operator=(类名 & left, const 类名& right);
```

### 3.4.3 调用时机

```
用一个类对象给另一个类对象进行赋值时,系统自动调用例如: string s1("hello"); //有参构造 string s2; //无参构造 s2 = s1; //拷贝赋值函数
```

### 3.4.4 也涉及深浅拷贝问题

```
1
   #include <iostream>
 3
 4 using namespace std;
 5
 6 class Stu
7
   {
8
   private:
9
        string name;
        int age;
10
       int *score;
11
12
   public:
13
        void set_age()
14
15
       {
16
            age++;
17
        }
18
19
        Stu()
```

```
20
            cout<<"无参构造"<<end1;
21
22
        }
23
       Stu(string n, int a, int s):name(n), age(a)
24
25
            score = new int(s);
            cout<<"有参构造"<<end1;
26
27
       }
       ~Stu()
28
29
       {
            delete score;
30
            cout<<"析构函数"<<end1;
31
       }
32
33
34
       //定义拷贝构造函数
       Stu(const Stu &other):name(other.name), age(other.age),
35
    score(new int(*(other.score)))
36
        {
            cout<<"拷贝构造"<<end1;
37
38
        }
39
40
       //定义拷贝赋值函数
41
       Stu &operator=(const Stu&other)
42
       {
           if(&other != this)
43
            {
44
45
                this->name = other.name;
46
                this->age = other.age;
47
                //this->score = other.score; //浅拷贝
48
                *(this->score) = *(other.score); //深拷贝
49
50
                cout<<"拷贝赋值函数"<<end1;
51
52
            }
53
54
            return *this;
        }
55
56
57
58
       void show()
59
        {
60
61
            cout<<"name = "<<name<<" age = "<<age<<end1;</pre>
62
        }
63
   };
```

```
64
65
66 int main()
   {
67
       Stu s1("zhangs", 18, 99); //有参构造
68
       Stu s2("li", 20, 88);
69
       Stu s3("wangwu", 30, 95);
70
71
72
       s2 = s1;
73
       s1.show();
74
75
       s2.show();
76
       s3.show();
77
78
      return 0;
79 }
80
```

如果一个类中,没有显性给定拷贝赋值函数,系统会默认提供一个拷贝赋值函数,来完成类对象之间的简单赋值,这是一个浅拷贝,当类中没有指针成员时,使用浅拷贝是没有问题的。

但是,如果类中有指针成员时,就会出现double free的段错误,原因是,两个类对象的指针成员,同时指向同一片内存空间,当把其中一个对象进行析构时,该指针成员指向的空间就被释放掉了,在析构另一个对象时,同一片空间就会释放两次,造成段错误。

此时,需要进行深拷贝:需要显性定义拷贝赋值函数,在拷贝赋值函数的函数体中,对指针成员指向的空间里,赋值为新空间中的数据值,即可完成深拷贝。

### 3.4.5 空类中会默认提供哪些函数

```
class Test
 2
   {
 3
       //构造函数
 4
       Test(){}
 5
      //析构函数
 6
       ~Test(){}
 7
       //拷贝构造
       Test(const Test &other){}
8
9
       //拷贝赋值
       Test &operator=(const Test &other){}
10
11 | };
12
```

# 四、匿名对象

```
1 #include <iostream>
2
 3
   using namespace std;
 4
 5
   class A
 6 {
7
   private:
8
       int a;
9
10
   public:
       A() {cout<<"无参构造"<<end1;}
11
       A(int a):a(a){cout<<"有参构造"<<end1;}
12
13
      void show()
14
       {
15
           cout << "a = " << a << end ];
       }
16
  };
17
18
19 void fun(A a)
20 {
       a.show();
21
   }
22
23
24
25 int main()
26 {
       A n = A(15); //A(15)就是匿名对象
27
28
      n.show();
29
       A arr[3] = {A(15), A(20), A(10)}; //用匿名对象给对象数据
30
   进行初始化
       for(int i=0; i<3; i++)
31
32
       {
           arr[i].show();
33
34
       }
35
       //使用匿名对象作为函数参数
36
37
       fun(A(30));
38
39
       return 0;
40
   }
41
```

### 5.1 explicit

当构造函数只有一个参数时,实例化对象时,可以直接用等于号完成隐式 (implicit) 调用,此时会造成代码阅读障碍,此时可以在构造函数前面explicit,那么 该函数,不能再进行隐式调用了。

### 5.2 default

当想要使用系统提供的默认函数时,可以在函数名后写=default,此时,该函数就是用的系统的

### 5.3 delete

- 1> 释放new关键字申请的空间
- 2> 删除系统提供的某些函数

```
1 #include <iostream>
 2
 3
   using namespace std;
 4
 5 class A
 6
   {
7
   private:
 8
       int a;
        int b;
9
10
    public:
11
12
        A() = default;
13
14
        explicit A(int a):a(a)
15
16
        {cout<<"有参构造"<<end1;}
17
        void show()
18
19
        {
            cout << "a = " << a << end ];
20
21
        }
22
        A(const A\&) = delete;
23
24
25
    };
26
```

```
27
28 int main()
29 {
30     A k;
31
32     return 0;
33 }
```

# 作业

参考string类完成my\_string类

```
class my_string
2
   {
 3
       private:
           char *str;
 4
 5
           int len;
 6
       public:
 7
           //无参构造
 8
           my_string(){
 9
               len = 15;
10
                str = new char[len];
11
           }
12
           //有参构造
           my_string(char *p){}
13
           //拷贝构造
14
           my_string(const my_string &other){}
15
           //拷贝赋值
16
           my_string &operator=(const my_string &other){}
17
           //析构函数
18
           ~my_string(){}
19
20
           //判空
21
           bool empty(){}
22
23
           //求总长度
           int size(){}
24
25
           //at()
           char &at(int pos){}
26
           //转c风格字符串函数
27
           char *c_str(){};
28
29
   };
```