

## 知识点1【new和delete 堆区空间操作】（了解）

### 1、new和delete操作基本类型的空间

new和malloc delete和free 没有区别

区别：

new 申请基本类型的数组

### 2、new和delete操作类的空间

### 3、new申请对象数组

## 知识点2【静态成员】（重要）

### 1、概念的引入

### 2、静态成员数据

案例1：使用静态成员数据 统计对象的个数

### 3、静态成员函数

## 知识点3【单例模式】（重要）

## 知识点4【类的存储结构】（了解）

## 知识点5【this指针】（了解）

### 1、知识点的引入

2、普通成员函数 默认有一个this指针 指向调用该成员函数的对象。

### 3、this来完成链式操作

## 知识点6【const修饰成员函数】（了解）

## 知识点7【友元】（重要）

1、普通全局函数 作为类的友元

2、类的某个成员函数 作为另一个类的友元

3、整个类作为 另一个类的友元

4、友元案例（遥控器的类）

5、设计动态数组类


## 知识点1 【new和delete 堆区空间操作】（了解）

### 1、new和delete操作基本类型的空间

new和malloc delete和free 没有区别

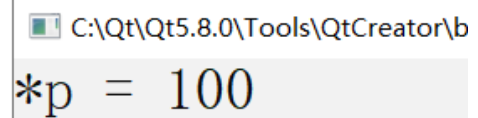
```
int *p = NULL;  
//p = (int *)malloc(sizeof(int));  
p = new int;//堆区空间
```

```
*p = 100;  
cout<<"*p = "<<*p<<endl;  
  
//free(p);  
delete p;
```



C:\Qt\Qt5.8.0\Tools\QtCreator  
\*p = 100

```
int *p = NULL;  
  
//申请int空间 并初始化为100  
p = new int(100);//堆区空间  
cout<<"*p = "<<*p<<endl;  
  
delete p;
```



C:\Qt\Qt5.8.0\Tools\QtCreator\b  
\*p = 100

区别:

new 不用强制类型转换

new在申请空间的时候可以 初始化空间内容

new 申请基本类型的数组

```

void test02()
{
    int *arr = NULL;

    //申请数组 5个int元素
    //arr = new int[5]; //堆区空间
    arr = new int[5]{10,20,30,40,50};
    int i=0;
    for(i=0;i<5;i++)
    {
        cout<<arr[i]<<" ";
    }
    cout<<endl;

    //如果new有[] delete就必须有[]
    delete [] arr;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreat

10 20 30 40 50

## 2、new和delete操作类的空间

malloc不会调用构造函数 free不会调用析构函数

new 会调用构造函数 delete调用析构函数

```

1 void test03()
2 {
3     #if 0 //malloc free
4     A *p = (A *)malloc(sizeof(A));
5
6     p->mA = 100;
7     cout<<p->mA<<endl;
8
9     free(p);
10
11 #endif
12
13 #if 1
14     A *p = new A;
15
16     p->mA = 100;
17     cout<<p->mA<<endl;
18
19     delete p;
20 #endif
21 }

```

new调用有参构造

```
#if 1
    A *p = new A(100);

    //p->mA = 100;
    cout<<p->mA<<endl;

    delete p;
#endif
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtc

有参构造  
100  
析构函数

### 3、new申请对象数组

```
void test04()
{
    //每个元素 默认调用无参构造
    //A *arr = new A[5];

    A *arr = new A[5]{A(10), A(20),A(30),A(40),A(50)};

    delete [] arr;
}

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

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_p

有参构造 mA=10  
有参构造 mA=20  
有参构造 mA=30  
有参构造 mA=40  
有参构造 mA=50  
析构函数 mA=50  
析构函数 mA=40  
析构函数 mA=30  
析构函数 mA=20  
析构函数 mA=10

## 知识点2【静态成员】（重要）

### 1、概念的引入

```
class Data
{
    int a;
    int b;
    int c;
};
```

ob1

int a;  
int b;  
int c;

ob2

int a;  
int b;  
int c;

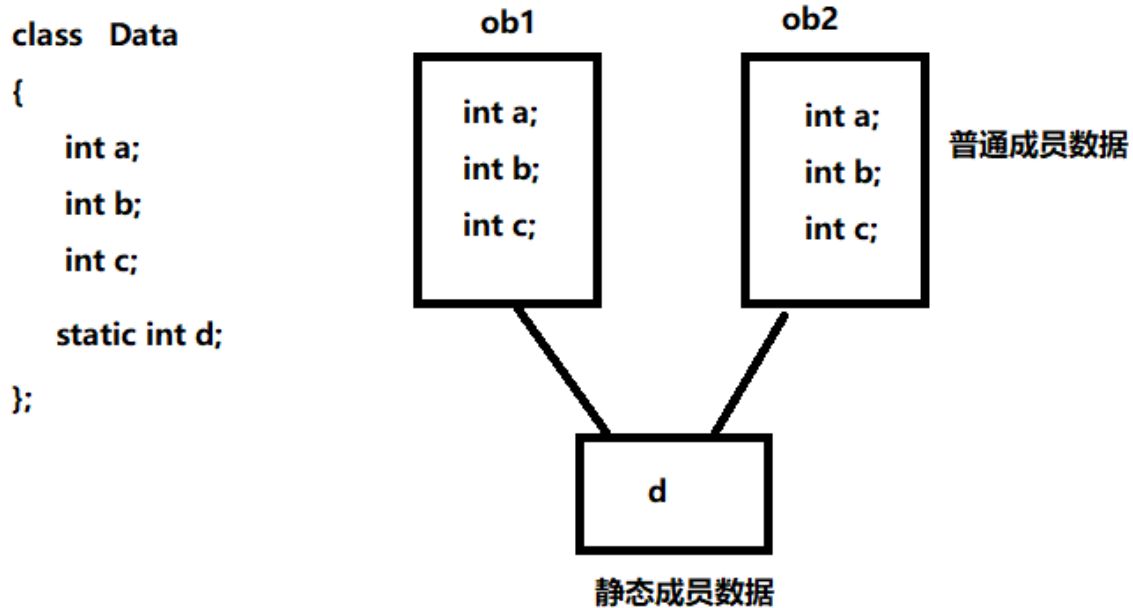
类的对象 拥有独立的 普通成员数据。

static 修饰的成员 叫 静态成员。

```
1 class Data
2 {
3     static int a;//静态成员数据
4     static void func();//静态成员函数
5     {
6
7     }
```

## 2、静态成员数据

static修饰的静态成员 属于类而不是对象。（所有对象 共享 一份 静态成员数据）



static修饰的成员 定义类的时候 必须分配空间。

static修饰的静态成员数据 必须类中定义 类外初始化。

```

1  class Data
2  {
3  public:
4      int a; //普通成员数据
5      //类中定义
6      static int b; //静态成员数据
7  };
8  //类外初始化
9  int Data::b=100; //不用加static
10
11 void test01()
12 {
13     //静态成员数据 通过类名称直接访问（属于类）
14     cout<<Data::b<<endl;
15
16     //静态成员数据 通过对象访问(共享)
17     Data ob1;
18     cout<<ob1.b<<endl;
19
20     ob1.b = 200;
21     Data ob2;

```

```
22  ob2.b = 300;
23  cout<<"Data::b"<<endl;//300
24 }
```

### 案例1：使用静态成员数据 统计对象的个数

```
1  class Data2
2  {
3  public:
4      int mA;
5      static int count;
6  public:
7      Data2()
8      {
9          count++;
10     }
11     Data2(int a)
12     {
13         mA = a;
14         count++;
15     }
16
17     Data2(const Data2 &ob)
18     {
19         count++;
20     }
21     ~Data2()
22     {
23         count--;
24     }
25 };
26 int Data2::count=0;
27 void test02()
28 {
29     Data2 ob1;
30     Data2 ob2(10);
31     Data2 ob3 = ob2;
32     cout<<"对象个数:"<<Data2::count<<endl;//3
33     {
34         Data2 ob4;
35         Data2 ob5;
36         cout<<"对象个数:"<<Data2::count<<endl;//5
```

```

37     }
38     cout<<"对象个数:"<<Data2::count<<endl;//3
39 }

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\  
对象个数:3  
对象个数:5  
对象个数:3

### 3、静态成员函数

静态成员函数 是属于类 而不是对象（所有对象 共享）

```

1 class Data
2 {
3     static void func();//静态成员函数
4     {
5
6     }
7 }

```

静态成员函数 可以直接通过类名称访问

```

class Data3
{
private:
    static int a;
public:
    static int getA()
    {
        return a;
    }
};
int Data3::a = 100;
void test03()
{
    cout<<Data3::getA()<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCre  
100

静态成员函数内 只能操作静态成员数据。

```
class Data3
{
private:
    int data;
    static int a;
public:
    static int getA()
    {
        data = 10; //err 静态成员函数 只能操作 静态成员数据 而data为普通成员数据
        return a;
    }
};
int Data3::a = 100;
```

## 知识点3 【单例模式】（重要）

单例模式的类 只能实例化 一个对象。

重要步骤：将构造函数私有化

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Singleton//单例模式
6  {
7      //构造私有化 防止实例化其他对象
8  private:
9      Singleton(){
10         count=0;
11         cout<<"构造"<<endl;
12     }
13     Singleton(const Singleton &ob){
14         count=0;
15     }
16     ~Singleton()
17     {
18         cout<<"析够"<<endl;
19     }
20 private:
21     //const防止p 在类内部 被修改指向
22     static Singleton * const p;//保存唯一的实例地址
23     int count;//统计任务执行次数
24 public:
25     static Singleton * getSingleton(void)//获取唯一的实例地址
```



```
26 {
27     return p;
28 }
29
30
31 //用户自定义 任务函数
32 void printString(char *str)
33 {
34     count++;
35     cout<<"当前第"<<count<<"次任务打印:"<<str<<endl;
36 }
37
38 };
39 Singleton *const Singleton::p = new Singleton;//创建唯一的实例
40
41 int main(int argc, char *argv[])
42 {
43     //获取单例的地址
44     Singleton *p1 =Singleton::getSingleton();
45     p1->printString("离职证明1");
46     p1->printString("学历证明1");
47     p1->printString("学位证明1");
48     p1->printString("身份证明1");
49
50     Singleton *p2 =Singleton::getSingleton();
51     p2->printString("离职证明2");
52     p2->printString("学历证明2");
53     p2->printString("学位证明2");
54     p2->printString("身份证明2");
55     return 0;
56 }
57
```

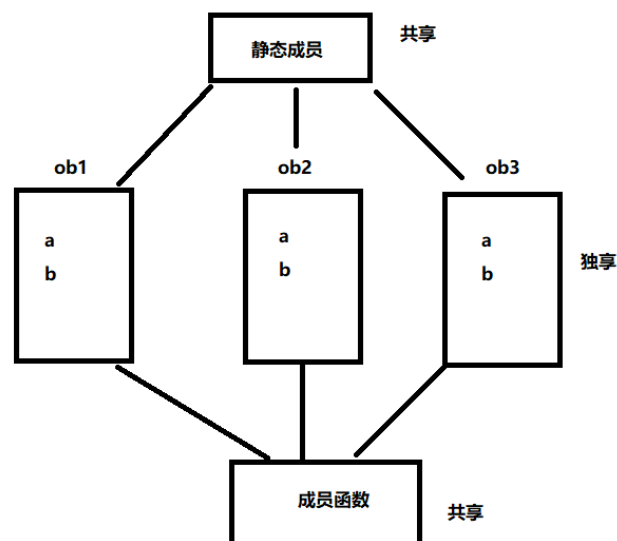
构造

当前第1次任务打印:离职证明1  
当前第2次任务打印:学历证明1  
当前第3次任务打印:学位证明1  
当前第4次任务打印:身份证明1  
当前第5次任务打印:离职证明2  
当前第6次任务打印:学历证明2  
当前第7次任务打印:学位证明2  
当前第8次任务打印:身份证明2

## 知识点4 【类的存储结构】（了解）

成员函数、静态成员 不占类的空间。

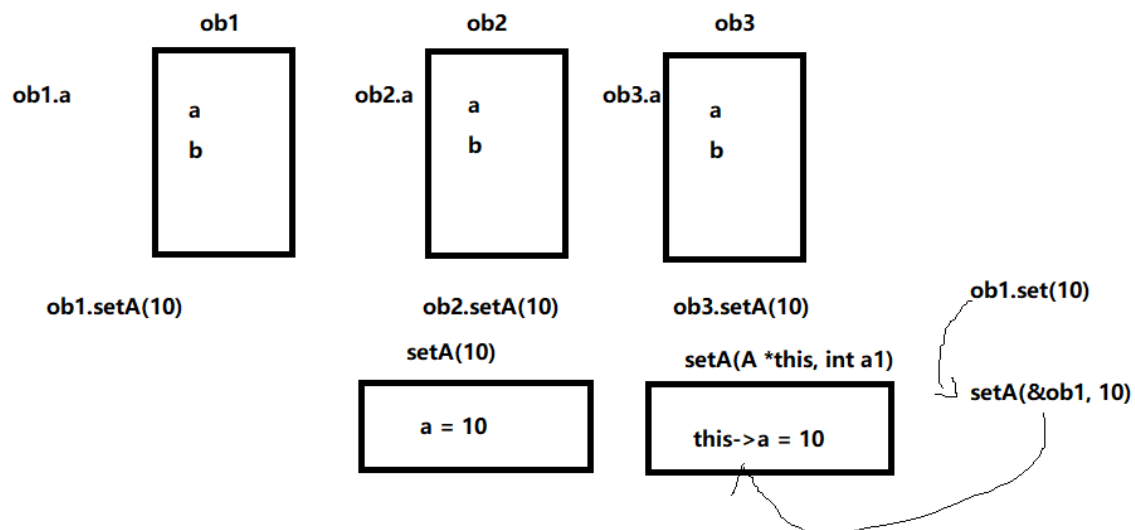
```
class Data1
{
public:
    char a;
    int b;
    static int c;
    void fun(){};
};
```



成员函数、静态成员 是独立存储 是所有对象共享。

## 知识点5 【this指针】（了解）

### 1、知识点的引入



## 2、普通成员函数 默认有一个this指针 指向调用该成员函数的对象。

```
class Data1
{
public:
    int a;
public:
    //函数的形参 和成员同名 可以使用this指针
    Data1(int a)
    {
        this->a = a;
        cout<<this<<endl;
    }
};

void test01()
{
    Data1 ob1(10);
    cout<<ob1.a<<endl;
    cout<<&ob1<<endl;
}
```

```
C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe
0x61fe8c
10
0x61fe8c
█
```

## 3、this来完成链式操作

```

using namespace std;
class Data1
{
public:
    Data1& myPrintf(char *str)
    {
        cout<<str<<" ";
        return *this;//返回调用该成员函数的对象
    }
};
void test01()
{
    Data1().myPrintf("hehe").myPrintf("haha").myPrintf("xixix");
}
int main(int argc, char* argv[])
{
    test01();
    return 0;
}

```

hehe haha xixix

## 知识点6 【const修饰成员函数】（了解）

const 修饰成员函数为只读（该成员函数不允许对 成员数据 赋值） mutable修饰的成员除外

```

1 //const 修饰成员函数为只读（该成员函数不允许对 成员数据 赋值）
2 void showData(void) const
3 {
4     //a = 100;//err
5     cout<<a<<" "<<b<<" "<<c<<endl;
6 }

```

```

1 class Data2
2 {
3 public:
4     int a;
5     int b;
6     mutable int c;
7 public:
8     Data2(int a, int b,int c)
9     {
10         this->a = a;
11         this->b = b;
12         this->c = c;
13     }
14     //const 修饰成员函数为只读（该成员函数不允许对 成员数据 赋值） mutable修饰的
    成员除外
15     void showData(void) const
16     {

```

```

17  //a = 100;//err
18  c=100;
19  cout<<a<<" "<<b<<" "<<c<<endl;
20  }
21  };
22  void test02()
23  {
24  Data2 ob1(10,20,30);
25  ob1.showData();
26  }

```

## 知识点7 【友元】（重要）

类将数据和方法封装在一起 加以权限区分 用户只能通过公共方法 操作私有数据。（封装性）

友元 重要用在运算符重载上。

一个函数或者类 作为了另一个类的友元 那么这个函数或类 就可以直接访问 另一个类的私有数据。

### 1、普通全局函数 作为类的友元

```


1  #include <string>
2  using namespace std;
3  class Room
4  {
5  friend void visiting01(Room &room);
6  private:
7  string bedRoom;//卧室
8  public:
9  string setingRoom;//客厅
10 public:
11 Room(string bedRoom, string setingRoom)
12 {
13 this->bedRoom = bedRoom;
14 this->setingRoom = setingRoom;
15 }
16 };
17
18 //普通全局函数
19 void visiting01(Room &room)
20 {
21 cout<<"访问了"<<room.setingRoom<<endl;

```

```

22  cout<<"访问了"<<room.bedRoom<<endl;
23  }
24
25  int main(int argc, char *argv[])
26  {
27      Room room("刘坤卧室","刘坤客厅");
28      visiting01(room);
29      return 0;
30  }

```

 C:\Qt\Qt5.8.0\Tools\QtCreator\

访问了刘坤客厅  
访问了刘坤卧室

## 2、类的某个成员函数 作为另一个类的友元

```

1  class Room;//向前声明 只能说明类名称
2  class goodGay
3  {
4  public:
5      void visiting01(Room &room);
6      void visiting02(Room &room);
7  };
8
9  class Room
10 {
11     friend void goodGay::visiting02(Room &room);
12 private:
13     string bedRoom;//卧室
14 public:
15     string setingRoom;//客厅
16 public:
17     Room(string bedRoom, string setingRoom)
18     {
19         this->bedRoom = bedRoom;
20         this->setingRoom = setingRoom;
21     }
22 };

```

```

23
24
25
26 int main(int argc, char *argv[])
27 {
28     Room room("吴维的卧室", "吴维的客厅");
29     goodGay ob;
30     ob.visiting01(room);
31     ob.visiting02(room);
32     return 0;
33 }
34
35 void goodGay::visiting01(Room &room)
36 {
37     cout<<"翰文访问了"<<room.setingRoom<<endl;
38     //cout<<"翰文访问了"<<room.bedRoom<<endl;
39 }
40
41 void goodGay::visiting02(Room &room)
42 {
43     cout<<"好基友张三访问了"<<room.setingRoom<<endl;
44     cout<<"好基友张三访问了"<<room.bedRoom<<endl;
45 }

```

C:\Qt\Qt5.8.0\tools\qtcreator\bin\qtcreator\_process\_stub.exe

翰文访问了吴维的客厅  
好基友张三访问了吴维的客厅  
好基友张三访问了吴维的卧室

### 3、整个类作为 另一个类的友元

这个类的所有成员函数 都可以访问另一个类的私有数据

```

1 class Room;//向前声明 只能说明类名称
2 class goodGay
3 {
4 public:
5     void visiting01(Room &room);
6     void visiting02(Room &room);

```

```

7  };
8
9  class Room
10 {
11     friend class goodGay;
12 private:
13     string bedRoom;//卧室
14 public:
15     string setingRoom;//客厅
16 public:
17     Room(string bedRoom, string setingRoom)
18     {
19         this->bedRoom = bedRoom;
20         this->setingRoom = setingRoom;
21     }
22 };
23
24
25
26 int main(int argc, char *argv[])
27 {
28     Room room("吴维的卧室", "吴维的客厅");
29     goodGay ob;
30     ob.visiting01(room);
31     ob.visiting02(room);
32     return 0;
33 }
34
35 void goodGay::visiting01(Room &room)
36 {
37     cout<<"翰文访问了"<<room.setingRoom<<endl;
38     cout<<"翰文访问了"<<room.bedRoom<<endl;
39 }
40
41 void goodGay::visiting02(Room &room)
42 {
43     cout<<"好基友张三访问了"<<room.setingRoom<<endl;
44     cout<<"好基友张三访问了"<<room.bedRoom<<endl;
45 }

```



翰文访问了吴维的客厅  
翰文访问了吴维的卧室  
好基友张三访问了吴维的客厅  
好基友张三访问了吴维的卧室

#### 4、友元的注意

1. 友元关系不能被继承。
2. 友元关系是单向的，类A是类B的朋友，但类B不一定是类A的朋友。
3. 友元关系不具有传递性。类B是类A的朋友，类C是类B的朋友，但类C不一定是类A的朋友

#### 4、友元案例（遥控器的类）

请编写电视机类，电视机有开机和关机状态，有音量，有频道，提供音量操作的方法，频道操作的方法。由于电视机只能逐一调整频道，不能指定频道，增加遥控类，遥控类除了拥有电视机已有的功能，再增加根据输入调台功能。

提示：遥控器可作为电视机类的友元类

```
1 #include <iostream>
2
3 using namespace std;
4 class TV;
5 class Remote
6 {
7 private:
8     TV *p;
9 public:
10     Remote(TV *p);
11     void offOrOn(void);
12     void upVolume(void);
13     void downVolume(void);
14
15     void upChannel(void);
16     void downChannel(void);
17
18     void showTv(void);
19
20     void setChannel(int channel);
21 };
```

```
22 class TV
23 {
24     friend void Remote::setChannel(int channel);
25     enum{OFF, ON};
26     enum{minVol, maxVol=10};
27     enum{minChan, maxChan=25};
28 private:
29     int state;
30     int volume;
31     int channel;
32 public:
33     TV()
34     {
35         state = OFF;
36         volume = minVol;
37         channel = minChan;
38     }
39     void offOrOn(void);
40     void upVolume(void);
41     void downVolume(void);
42
43     void upChannel(void);
44     void downChannel(void);
45
46     void showTv(void);
47
48 };
49
50
51 int main(int argc, char *argv[])
52 {
53     //实例化一个电视机
54     TV tv;
55     Remote re(&tv);
56     re.offOrOn();
57     re.upVolume();
58     re.upVolume();
59     re.upVolume();
60     re.setChannel(20);
61     re.showTv();
```

```
62
63     return 0;
64 }
65
66 void TV::offOrOn()
67 {
68     state = (state==OFF?ON:OFF);
69     return;
70 }
71
72 void TV::upVolume()
73 {
74     if(volume == maxVol)
75     {
76         cout<<"音量已经最大了"<<endl;
77         return;
78     }
79     volume++;
80     return;
81 }
82
83 void TV::downVolume()
84 {
85     if(volume == minVol)
86     {
87         cout<<"音量已经最小了"<<endl;
88         return;
89     }
90     volume--;
91     return;
92 }
93
94 void TV::upChannel()
95 {
96     if(channel == maxChan)
97     {
98         cout<<"频道已经最大了"<<endl;
99         return;
100     }
101     channel++;
```

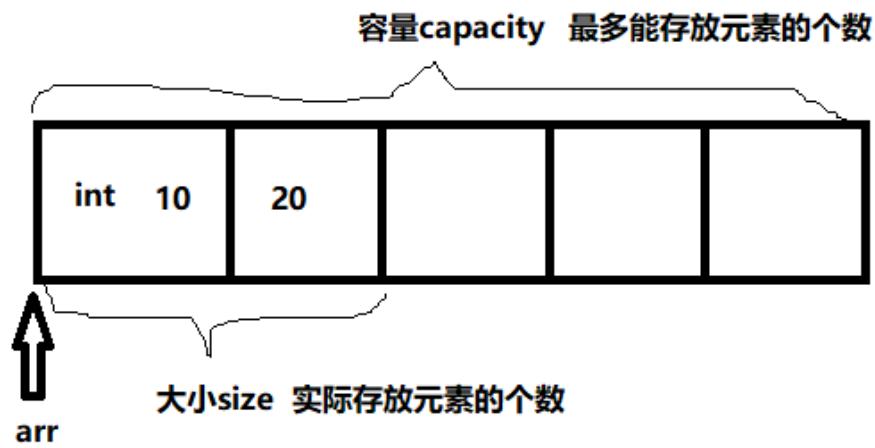
```
102     return;
103 }
104
105 void TV::downChannel()
106 {
107     if(channel == minChan)
108     {
109         cout<<"频道已经最小了"<<endl;
110         return;
111     }
112     channel--;
113     return;
114 }
115
116 void TV::showTv()
117 {
118     cout<<"当前电视机的状态:"<<(state==OFF?"关":"开")<<endl;
119     cout<<"当前电视机的音量:"<<volume<<endl;
120     cout<<"当前电视机的频道:"<<channel<<endl;
121 }
122
123 Remote::Remote(TV *p)
124 {
125     this->p = p;
126 }
127
128 void Remote::offOrOn()
129 {
130     p->offOrOn();
131 }
132
133 void Remote::upVolume()
134 {
135     p->upVolume();
136 }
137
138 void Remote::downVolume()
139 {
140     p->downVolume();
141 }
142
```

```
143 void Remote::upChannel()  
144 {  
145     p->upChannel();  
146 }  
147  
148 void Remote::downChannel()  
149 {  
150     p->downChannel();  
151 }  
152  
153 void Remote::showTv()  
154 {  
155     p->showTv();  
156 }  
157  
158 void Remote::setChannel(int channel)  
159 {  
160     p->channel = channel;  
161 }  
162
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_proc

当前电视机的状态:开  
当前电视机的音量:3  
当前电视机的频道:20

## 5、设计动态数组类



array.h

```
1 #ifndef ARRAY_H
2 #define ARRAY_H
3
4
5 class Array
6 {
7 private:
8     int *arr; //存放首元素地址
9     int capacity; //容量
10    int size; //大小
11 public:
12     Array();
13     Array(int capacity);
14     Array(const Array &ob);
15     ~Array();
16
17     int getCapacity() const;
18     int getSize() const;
19
20     void printArray(void);
21
22     //插入尾部元素
23     void pushBack(int elem);
24     //删除尾部元素
25     void popBack(void);
26     int &at(int pos);
27 };
28
29 #endif // ARRAY_H
```

## array.cpp

```
1  #include "array.h"
2  #include<string.h>
3  #include<iostream>
4
5  using namespace std;
6  int Array::getCapacity() const
7  {
8      return capacity;
9  }
10
11 int Array::getSize() const
12 {
13     return size;
14 }
15
16 void Array::printArray()
17 {
18     int i=0;
19     for(i=0;i<size; i++)
20     {
21         cout<<arr[i]<<" ";
22     }
23     cout<<endl;
24     return;
25 }
26
27 void Array::pushBack(int elem)
28 {
29     //判断容器是否满
30     if(size == capacity)
31     {
32         //申请空间
33         int *tmp = new int[2*capacity];
34         //将就空间的内容 拷贝到新空间
35         memcpy(tmp, arr, capacity*sizeof(int));
36         //释放原有的空间
37         delete [] arr;
38         //更新arr的空间
39         arr = tmp;
```

```
40 //更新容量
41 capacity = 2*capacity;
42 }
43
44 arr[size]=elem;
45 size++;
46 return;
47 }
48
49 void Array::popBack()
50 {
51     if(size == 0)
52     {
53         cout<<"容量为空"<<endl;
54     }
55     else
56     {
57         size--;
58     }
59     return;
60 }
61
62 int& Array::at(int pos)
63 {
64     if(pos<0 || pos >=size)
65     {
66         cout<<"访问违法内存"<<endl;
67         exit(-1);
68     }
69
70     return arr[pos];
71 }
72
73 Array::Array()
74 {
75     capacity = 5;
76     size = 0;
77     arr = new int[capacity];
78     //空间清0
79     memset(arr, 0, sizeof(int)*capacity);
```



```

80 }
81
82 Array::Array(int capacity)
83 {
84     this->capacity = capacity;
85     size = 0;
86     arr = new int[capacity];
87     //空间清0
88     memset(arr, 0, sizeof(int)*capacity);
89 }
90
91 Array::Array(const Array &ob)
92 {
93     capacity = ob.capacity;
94     size = ob.size;
95     //深拷贝
96     arr = new int[capacity];
97     memcpy(arr, ob.arr, sizeof(int)*capacity);
98 }
99
100 Array::~~Array()
101 {
102     if(arr != NULL)
103     {
104         delete [] arr;
105         arr = NULL;
106     }
107 }

```

## main.cpp

```

1  #include <iostream>
2  #include "array.h"
3  using namespace std;
4
5  int main(int argc, char *argv[])
6  {
7      Array ob;
8      cout<<ob.getCapacity()<<" "<<ob.getSize()<<endl;
9
10     ob.pushBack(10);
11     ob.pushBack(20);

```

```

12  ob.pushBack(30);
13  ob.pushBack(40);
14  ob.printArray();
15  cout<<ob.getCapacity()<<" "<<ob.getSize()<<endl;
16  ob.pushBack(50);
17  ob.pushBack(60);
18  ob.printArray();
19  cout<<ob.getCapacity()<<" "<<ob.getSize()<<endl;
20  ob.popBack();
21  ob.popBack();
22  ob.printArray();
23  cout<<ob.getCapacity()<<" "<<ob.getSize()<<endl;
24
25  cout<<"arr[2] = "<<ob.at(2)<<endl;
26  ob.at(2) = 100;
27  ob.printArray();
28  return 0;
29  }

```

C:\Qt\Qt5.8.0\tools\qtcreator\bin\qtcreat

```

5 0
10 20 30 40
5 4
10 20 30 40 50 60
10 6
10 20 30 40
10 4
arr[2] = 30
10 20 100 40

```