数组类的封装：

|  |
| --- |
| class MyArray  {  public:  MyArray(); //默认构造 默认100容量  MyArray(int capacity);  MyArray(const MyArray& array);  ~MyArray();  //尾插法  void push\_Back(int val);  //根据索引获取值  int getData(int index);  //根据索引设置值  void setData(int index, int val);  //获取数组大小  int getSize();  //获取数组容量  int getCapacity();  //[]运算符重载  int& operator[](int index );  private:  int \* pAddress; //指向真正存储数据的指针  int m\_Size; //数组大小  int m\_Capacity; //数组容量  }; |

|  |
| --- |
| #include "MyArray.h"  //默认构造  MyArray::MyArray()  {  this->m\_Capacity = 100;  this->m\_Size = 0;  this->pAddress = new int[this->m\_Capacity];  }  //有参构造 参数 数组容量  MyArray::MyArray(int capacity)  {  //cout << "有参函数调用" << endl;  this->m\_Capacity = capacity;  this->m\_Size = 0;  this->pAddress = new int[this->m\_Capacity];  }  //拷贝构造  MyArray::MyArray(const MyArray& array)  {  cout << "拷贝构造调用" << endl;  this->pAddress = new int[array.m\_Capacity];  this->m\_Size = array.m\_Size;  this->m\_Capacity = array.m\_Capacity;  for (int i = 0; i < array.m\_Size;i++)  {  this->pAddress[i] = array.pAddress[i];  }  }  //析构  MyArray::~MyArray()  {  if (this->pAddress != NULL)  {  //cout << "析构调用" << endl;  delete[] this->pAddress;  this->pAddress = NULL;  }  }  void MyArray::push\_Back(int val)  {  //判断越界？ 用户自己处理  this->pAddress[this->m\_Size] = val;  this->m\_Size++;  }  int MyArray::getData(int index)  {  return this->pAddress[index];  }  void MyArray::setData(int index, int val)  {  this->pAddress[index] = val;  }  int MyArray::getSize()  {  return this->m\_Size;  }  int MyArray::getCapacity()  {  return this->m\_Capacity;  }  //[]重载实现  int& MyArray::operator[](int index)  {  return this->pAddress[index];  } |

|  |
| --- |
| void test01()  {  //堆区创建数组  MyArray \* array = new MyArray(30);  MyArray \* array2 = new MyArray(\*array); //new方式指定调用拷贝构造  MyArray array3 = \*array; //构造函数返回的本体  //MyArray \* array4 = array; //这个是声明一个指针 和array执行的地址相同，所以不会调用拷贝构造  delete array;  //尾插法测试  for (int i = 0; i < 10;i++)  {  array2->push\_Back(i);  }  //获取数据测试  for (int i = 0; i < 10;i++)  {  cout << array2->getData(i) << endl;  }    //设置值测试  array2->setData(0, 1000);  cout << array2->getData(0) << endl;;  //获取数组大小  cout << "array2 的数组大小为： " << array2->getSize() << endl;  //获取数组容量  cout << "array2 的数组容量为： " << array2->getCapacity() << endl;  //获取 设置 数组内容 如何用[]进行设置和访问  array3.push\_Back(100000);  cout << array3.getData(0) << endl;  cout << array3[0] << endl;  array3[0] = 100; // 100000 = 100  cout << array3[0] << endl;  } |

1. 加号运算符重载
   1. 如果想让自定义数据类型 进行+运算，那么就需要重载 + 运算符
   2. 在成员函数 或者 全局函数里 重写一个+运算符的函数
   3. 函数名 operator+ () {}
   4. 运算符重载 也可以提供多个版本

|  |
| --- |
| 成员函数中的+重载  class Person  {  public:  Person(){};  Person(int a, int b) :m\_A(a), m\_B(b)  {}  //+号运算符重载 成员函数 二元  /\*Person operator+ ( Person & p)  {  Person tmp;  tmp.m\_A = this->m\_A + p.m\_A;  tmp.m\_B = this->m\_B + p.m\_B;  return tmp;  }\*/  int m\_A;  int m\_B;  }; |

|  |
| --- |
| 全局函数+重载  class Person  {  public:  Person(){};  Person(int a, int b) :m\_A(a), m\_B(b)  {}  int m\_A;  int m\_B;  };  //利用全局函数 进行+号运算符的重载  Person operator+ ( Person &p1,Person& p2) //二元 p1 + p2  {  Person tmp;  tmp.m\_A = p1.m\_A + p2.m\_A;  tmp.m\_B = p1.m\_B + p2.m\_B;  return tmp;  }  Person operator+ (Person &p1, int a) //二元  {  Person tmp;  tmp.m\_A = p1.m\_A + a;  tmp.m\_B = p1.m\_B + a;  return tmp;  }  void test01()  {  Person p1(10, 10);  Person p2(10, 10);  Person p3 = p1 + p2; // p1 + p2 从什么表达式转变的？ p1.operator+(p2) operator+(p1,p2)  Person p4 = p1 + 10; //重载的版本  cout << "p3 的 m\_A: " << p3.m\_A << " m\_B: " << p3.m\_B << endl;  //operator+(p1, p2);    } |

1. 左移运算符重载
   1. 不要随意乱用符号重载
   2. 重载左移运算符不可以卸载成员函数中
   3. 内置数据类型 的运算符不可以重载
   4. cout << 直接对Person自定义数据类型 进行输出
   5. 写到全局函数中 ostream& operator<< ( ostream & cout, Person & p1 ) {}
   6. 如果重载时候想访问 p1的私有成员，那么全局函数要做Person的友元函数

|  |
| --- |
| 1. class Person 2. { 3. friend ostream& operator<<(ostream &cout, Person & p1); 4. public: 5. Person(){} 6. Person(int a, int b) 7. { 8. this->m\_A = a; 9. this->m\_B = b; 10. } 11. /\*void operator<<() 重载左移运算符不可以写到成员函数中 12. { 13. }\*/ 14. private: 15. int m\_A; 16. int m\_B; 17. }; 18. ostream& operator<<(ostream &cout , Person & p1 ) //第一个参数 cout 第二个参数 p1 19. { 20. cout << "m\_A = " << p1.m\_A << " m\_B = " << p1.m\_B; 21. return cout; 22. } 23. void test01() 24. { 25. Person p1(10, 10); 26. cout << p1 << "helloworld" <<endl; 28. } |

1. 前置 后置 ++ 运算符重载
   1. 自己实现int类型 MyInteger
   2. 内部维护以int数据
   3. MyInteger myInt
   4. myInt ++ 后置 ++myInt 前置
   5. 重载++运算符 operator++() 前置 operator++(int) 后置
   6. 前置理念 先++ 后返回自身 后置理念 先保存住原有值 内部++ 返回临时数据

|  |
| --- |
| class MyInteger  {  friend ostream& operator<<(ostream& cout, MyInteger & myInt);  public:  MyInteger()  {  m\_Num = 0;  };  //前置++重载  MyInteger& operator++()  {  this->m\_Num++;  return \*this;  }  //后置++ 重载  MyInteger operator++(int)  {  //先保存目前数据  MyInteger tmp = \*this;  m\_Num++;  return tmp;  }  int m\_Num;  };  ostream& operator<<( ostream& cout ,MyInteger & myInt)  {  cout << myInt.m\_Num;  return cout;  }  void test01()  {  MyInteger myInt;  // 前置++    cout << ++(++myInt) << endl;  cout << myInt << endl;  //cout << myInt++ << endl; // 后置++  //cout << myInt << endl;  }  void test01()  {  MyInteger myInt;  // 前置++    cout << ++(++myInt) << endl;  cout << myInt << endl;  //cout << myInt++ << endl; // 后置++  //cout << myInt << endl;  } |

* 1. 练习 自己实现递减运算符重载 --

1. 智能指针实现
   1. Person类有showAge 成员函数
   2. 如果new出来的Person对象，就要让程序员自觉的去释放 delete
   3. 有了智能指针，让智能指针托管这个Person对象，对象的释放就不用操心了，让智能指针管理
   4. 为了让智能指针想普通的Person\*指针一样使用 就要重载 -> 和\*

|  |
| --- |
| class Person  {  public:  Person(int age)  {  this->m\_Age = age;  }  void showAge()  {  cout << "年龄为：" << this->m\_Age << endl;  }  ~Person()  {  cout << "Person的析构调用" << endl;  }  int m\_Age;  };  //智能指针  //用来托管自定义类型的对象，让对象进行自动的释放  class smartPointer  {  public:  smartPointer(Person \* person)  {  this->person = person;  }  //重载->让智能指针对象 像Person \*p一样去使用  Person \* operator->()  {  return this->person;  }  //重载 \*  Person& operator\*()  {    return \*this->person;  }  ~smartPointer()  {  cout << "智能指针析构了" << endl;  if (this->person !=NULL)  {  delete this->person;  this->person = NULL;  }  }  private:  Person \* person;  };  void test01()  {  //Person p1(10); //自动析构  //Person \* p1 = new Person(10);  //p1->showAge();  // delete p1;  smartPointer sp(new Person(10)); //sp开辟到了栈上，自动释放  sp->showAge(); // sp->->showAge(); 编译器优化了 写法  (\*sp).showAge();  } |

1. 赋值运算符重载
   1. 系统默认给类提供 赋值运算符写法 是简单值拷贝
   2. 导致如果类中有指向堆区的指针，就可能出现深浅拷贝的问题
   3. 所以要重载 = 运算符
   4. 如果想链式编程 return\*this

|  |
| --- |
| //一个类默认创建 默认构造、析构、拷贝构造 operator=赋值运算符 进行简单的值传递  class Person  {  public:  Person(int a)  {  this->m\_A = a;  }  int m\_A;  };  void test01()  {  Person p1(10);  Person p2(0);  p2 = p1; //赋值  cout << "p2 的m\_A" << p2.m\_A <<endl;  }  class Person2  {  public:  Person2(char \* name)  {  this->pName = new char[strlen(name) + 1];  strcpy(this->pName, name);  }  //重载 = 赋值运算符  Person2& operator= ( const Person2 & p)  {  //判断如果原来已经堆区有内容，先释放  if (this->pName != NULL)  {  delete[] this->pName;  this->pName = NULL;  }  this->pName = new char[strlen(p.pName) + 1];  strcpy(this->pName, p.pName);  return \*this;  }  ~Person2()  {  if (this->pName != NULL)  {  delete[] this->pName;  this->pName = NULL;  }  }  char \* pName;  };  void test02()  {  Person2 p1("狗蛋");  Person2 p2("狗剩");  Person2 p3("");  p3 = p2 = p1;  cout << p2.pName << endl;  cout << p3.pName << endl;  //int a = 10;  //int b = 20;  //int c;  //c = a = b; //都是20  //cout << a << " " << b << " " << c << endl;  } |

1. []运算符重载
   1. 返回数组索引的引用
   2. int & operator[](int index)
   3. return this->pAddress[index]