期末复习 含OO十大问题

> 阅读使人充实，会谈使人敏捷，写作与笔记使人精确。 —— 培根

十个OO问题

1、当类中未自主定义构造函数，compiler会提供默认构造函数，Why?

What does Compiler do for an“Empty”class?

合成默认构造函数的作用：

保证对象初始化：对象处于有效的初始状态

简化类的使用，减少代码冗余：如果程序员没有特别的需求，编译器提供的默认构造函数可以避免冗余的初始化代码。

//todo enpty类

2、When构造函数、析构函数被定义为*private*?

友元、static成员的使用，When?

私有的构造函数、析构函数：

强制自主控制对象存储分配

不允许在类外随意创建对象，接管对象创建\销毁，如单例模式、工厂模式

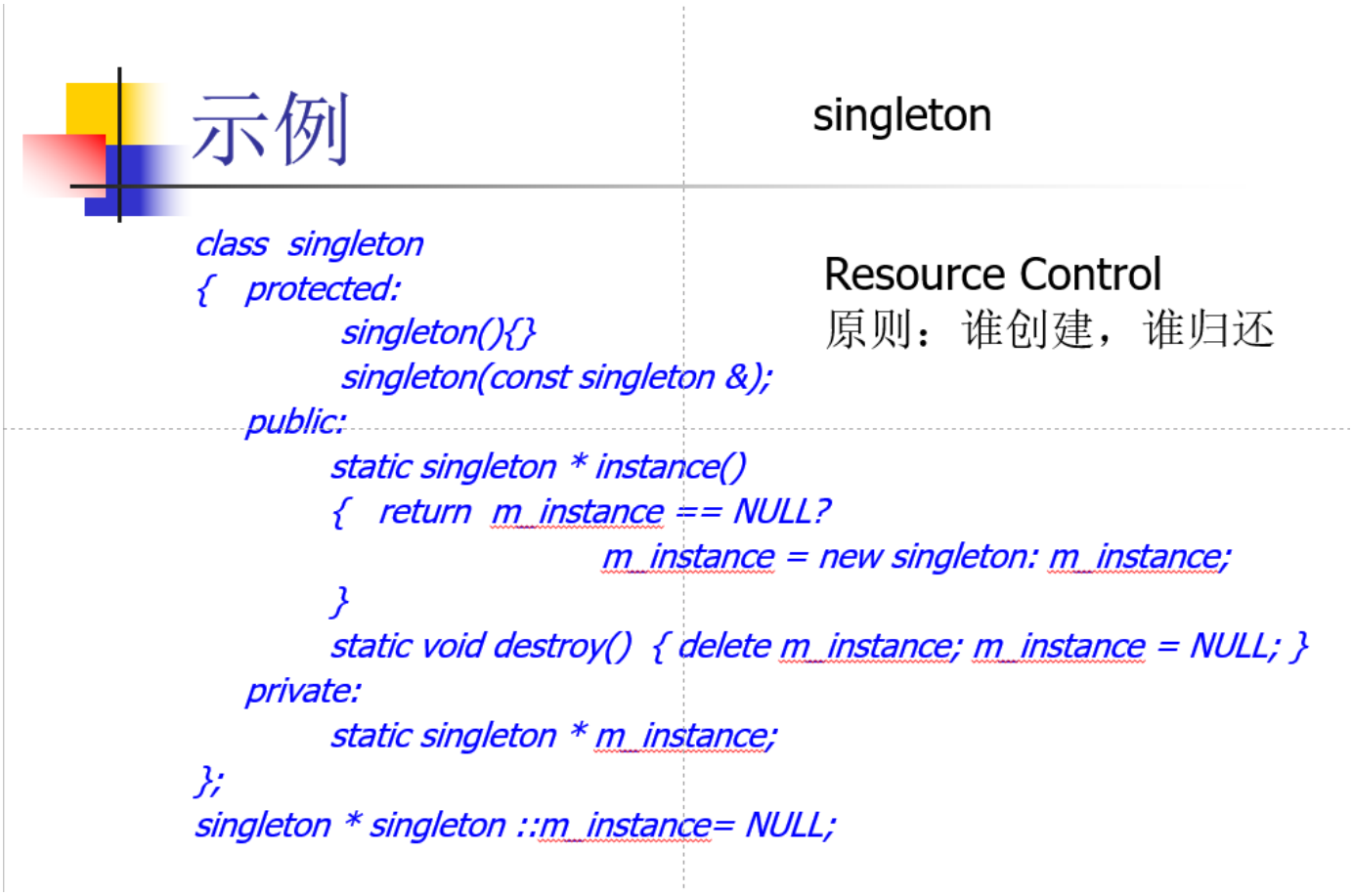
如A* getA() {return new A;}

经典例子-单例模式

（我写的不好，不需要count计数；无差别delete）

```
A* getA() {
    if(ainstance == NULL){
        Return new A;
    }else{
        Return ainstance;
```

```
    }
    count++;
}
A* destroy(){
    count--;
    if(count <= 0){
        Ainstance = nullptr;
        Count = 0;
    }
}
```



友元：

外部函数或外部类，不是类的接口，不是类的成员，但希望向类的成员函数一样访问私有变量；

操作符重载

```
1 void func() ;
2 class B;
3 class C
4 { …….
```

```
    void f();
};
class A
      {     …
   friend void func();     //友元函数
   friend class B;         //友元类
   friend void C::f();     //友元类成员函数
   };
```

- 分类
  - 友元函数
  - 友元类
  - 友元类成员函数

- 作用
  - 提高程序设计灵活性
  - 数据保护和对数据的存取效率之间的一个折中方案

static:

类的所有对象享有共同的静态成员。是类级别，而非对象级别的成员。

类的对象共享数据或行为，如对象计数

常用于定义类的常量

静态属性单独类外定义；

静态函数不能使用非静态成员。

3、Why引入成员初始化表?

Why初始化表执行次序只与类数据成员的定义次序相关?

为什么?

1、提高编译器效率：在对象创建时直接初始化成员变量，而不是先通过**默认构造函数**初始化成员，再通过**构造函数体内的赋值语句重新赋值**。这种方式避免了**两次初始化**，从而提高了效率。

2、有些**变量**必须使用成员初始化表初始化：常量成员变量（`const` 类型）和引用成员变量（`T&` 类型）。

3、**调用基类构造函数**：当一个类有基类时，可以通过成员初始化表显式地调用基类的构造函数。如果**基类没有默认构造函数**或者**需要传递参数给基类构造函数**，则必须使用成员初始化表来显式地调用基

类构造函数。

为什么初始化表执行顺序由成员声明顺序决定：

避免由于成员间依赖关系引发的潜在（未定义）问题

4、Why引入Copy constructor、 move constructor 、=操作符重载?

总回答：正确管理对象的资源，特别是在涉及**动态内存管理**或**资源所有权转移**的情况下。这些构造函数和操作符帮助确保程序在处理对象时不会导致资源泄漏、未定义行为或性能问题。

拷贝构造函数：

创建对象时，用同一类的对象对其初始化

何时自动调用：

A b = a;

值传递的参数传递

以值作为返回值

why?

默认的浅拷贝会造成悬挂指针等问题，而深拷贝可以解决（特别是当对象中包含动态分配的内存、文件句柄或其他外部资源时）

移动构造函数：

移动构造函数会接管资源的所有权。这避免了不必要的内存分配和复制，提高了程序性能。（返回值优化、容器扩容）

=运算符重载：

赋值操作符重载允许**定义对象的赋值行为**

防止资源泄漏

防止自赋值问题：当一个对象赋值给自己时（例如 `a = a;`），默认的赋值操作符可能不会进行自赋值检查，这可能导致删除自己持有的资源，然后再尝试访问已删除的资源。手动重载赋值操作符时，可以添加自赋值检查来避免这种问题。

自定义赋值逻辑：有时，赋值操作不仅仅是简单的成员逐一复制，还可能需要处理更复杂的逻辑，如资源管理、状态复制等。

5. What is Late Binding? How C++implement *vitural*?

前期绑定（Early Binding）

编译时刻

依据对象的静态类型

效率高、灵活性差

动态绑定（Late Binding）

运行时刻

依据对象的实际类型（动态）

灵活性高、效率低

实现virtual:

对象的内存空间中含有指针，指向其虚函数表

*(**((char *)p-4))(p)*

6. When we use *virtual*

三种继承的规则，多态

7. What public继承和non-public继承means?

8. Why *= () [] ->* 不能作为全局函数重载

9. When成员函数能返回*&*

10. When and How to 重载*new*、*delete*?

为什么要引入new delete操作符

支持**动态内存管理**。C++ 是一种支持面向对象编程（OOP）和低级编程的语言，它允许程序员精确控制内存的分配和释放，而这需要使用动态内存分配机制。

确保能够正确调用constructor和destructor

通过 `new` 和 `delete`，程序员可以完全控制内存的分配和释放：

- **控制内存大小**：可以根据运行时的需求决定分配多少内存。
- **控制内存释放时机**：需要手动释放内存，这给了程序员灵活性，也意味着需要特别小心，避免内存泄漏或重复释放。

1. C++ 内存主要分为栈区、堆区、全局区、静态区、常量区、代码区。

char*的介绍

特别好的博客

https://blog.csdn.net/iiiiiiimp/article/details/142110072

2. 继承/多继承构造函数调用顺序

特别好的博客https://blog.csdn.net/IEEEITU/article/details/6785925?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522ed59c6113a59cff392f0913c8e98396b%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=ed59c6113a59cff392f0913c8e98396b&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~baidu_landing_v2~default-1-6785925-null-null.142^v100^pc_search_result_base3&utm_term=c%2B%2B%E4%B8%AD%E5%A4%9A%E7%BB%A7%E6%89%BF%E6%97%B6%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8%E9%A1%BA%E5%BA%8F&spm=1018.2226.3001.4187

```
class B1
{public:
 B1(int i)
 {cout<<"consB1"<<i<<endl;}
};//定义基类B1
class B2
{public:
 B2(int j)
 {cout<<"consB2"<<j<<endl;}
};//定义基类B2
class B3
{
public:
  B3()
  {cout<<"consB3 *"<<endl;}
};//定义基类B3
class C: public B2, public B1, public B3
{public:
  C(int a,int b,int c,int d,int e)
   :B1(a),memberB2(d),memberB1(c),B2(b)
  {m=e; cout<<"consC"<<endl;}
private:
 B1 memberB1;
 B2 memberB2;
 B3 memberB3;
 int m;
};//继承类C
void main()
{ C  obj(1,2,3,4,5);  }//主函数
```

运行结果：consB2 2 consB1 1 consB3 * consB1 3 consB2 4 consB3 * consC

//先按照继承顺序：B2，B1，B3

//第一步：先继承B2,在初始化列表里找到B2(b)，打印"constB22"

//第二步：再继承B1,在初始化列表里找到B1(a)，打印"constB11"

//第三步：又继承B3,在初始化列表里找不到B3(x), 则调用B3里的默认构造函数B3()，打印"constB3 *"

//再按照数据成员定义顺序：memberB1, memberB2, memberB3

//第四步：在初始化列表里找到memberB1(c),初始化一个B1对象，用c为参数，则调用B1的构造函数，打印"constB13"

//第五步：在初始化列表里找到memberB2(d),初始化一个B2对象，用d为参数，则调用B2的构造函数，打印"constB24"

//第六步：在初始化列表里找不到memberB3(x),则调用B3里的默认构造函数B3()，打印"constB3 *"

//最后完成本对象初始化的剩下部分,也就是C自己的构造函数的函数体：{m=e; cout<<"consC" <<endl;}

//第七步：打印"consC"

有虚基类时，虚基类构造函数在最前面调用

虚基类的构造函数由最新派生出的类的构造函数调用

虚基类的构造函数优先非虚基类的构造函数执行

3. 封装代理类&[]运算符重载&实现多维数组

```
class Array2D
{   public:
    class Array1D
    {
      public:
        Array1D(int *p) { this->p = p; }
        int& operator[ ] (int index) { return p[index]; }
        const int operator[ ] (int index) const { return p[index]; }
      private:
        int *p;
    };

    Array2D(int n1, int n2) { p = new int[n1*n2]; num1 = n1; num2 = n2; }
    virtual ~Array2D( ) { delete [ ] p; }
```

```
    Array1D operator[ ] (int index) { return p+index*num2; }
    const Array1D operator[ ] (int index) const { return p+index*num2; }

    private:
    int *p;
    int num1, num2;
};
```

```
class Array3D {
public:
    // 二维数组代理类
    class Array2D {
    public:
        // 一维数组代理类
        class Array1D {
        public:
            Array1D(int *p) : p(p) {}

            int& operator[](int index) { return p[index]; }
            const int operator[](int index) const { return p[index]; }

        private:
            int *p;
        };

        Array2D(int *p, int num2) : p(p), num2(num2) {}

        Array1D operator[](int index) { return Array1D(p + index * num2); }
        const Array1D operator[](int index) const { return Array1D(p + index *
num2); }

    private:
        int *p;
        int num2;
    };

    Array3D(int n1, int n2, int n3) : num1(n1), num2(n2), num3(n3) {
        p = new int[n1 * n2 * n3];
    }

    virtual ~Array3D() {
        delete[] p;
    }

```

```
36      Array2D operator[](int index) { return Array2D(p + index * num2 * num3,
    num2 * num3); }
37      const Array2D operator[](int index) const { return Array2D(p + index *
    num2 * num3, num2 * num3); }
38
39  private:
40      int *p;
41      int num1, num2, num3;
42  };
43
44  int main() {
45      Array3D arr(3, 4, 5); // 三维数组大小为 3x4x5
46
47      // 设置值
48      arr[0][1][2] = 100;
49      arr[1][2][3] = 200;
50      arr[2][0][4] = 300;
51
52      // 获取值
53      std::cout << "arr[0][1][2] = " << arr[0][1][2] << std::endl;  // 输出 100
54      std::cout << "arr[1][2][3] = " << arr[1][2][3] << std::endl;  // 输出 200
55      std::cout << "arr[2][0][4] = " << arr[2][0][4] << std::endl;  // 输出 300
56
57      return 0;
58  }
59
```

4. 定位 `new` 和 定位 `delete`

定位 `new` 和定位 `delete` 是一种特定的内存分配方式，它们允许在特定的内存区域中分配和释放内存。定位 `new` 和 `delete` 通常用于内存池、对象池和其他需要在预先分配的内存区域内管理对象的场景。

2.1 定位 `new`

定位 `new` 是一种特殊形式的 `new`，它允许我们在给定的内存位置分配对象，而不是由系统管理的堆上分配。

定位 `new` 语法：

```
1  void* operator new(size_t size, void* ptr) {
2      return ptr;
3  }
```

这种重载形式允许在指定的内存地址（`ptr`）上“构造”一个对象。这种方法常常用于实现内存池。通过在预先分配的内存区域内构造对象，可以避免重复调用 `malloc` 或 `new` 来分配内存。

示例：

```cpp
#include <iostream>

class MyClass {
public:
    MyClass() { std::cout << "Constructor called\n"; }
    ~MyClass() { std::cout << "Destructor called\n"; }
};

int main() {
    // 创建足够大的内存空间来容纳一个 MyClass 对象
    char buffer[sizeof(MyClass)];

    // 在预分配的内存 buffer 中构造一个 MyClass 对象
    MyClass* obj = new (buffer) MyClass;  // 定位 new

    obj->~MyClass();  // 手动调用析构函数
    return 0;
}
```

在这个示例中，我们首先在栈上创建了一个足够大的字符数组 `buffer`，然后在这个内存区域内通过定位 `new` 构造了一个 `MyClass` 对象。这个对象使用的内存不是由 `new` 操作符分配的，而是直接在 `buffer` 数组内构造的。

2.2 定位 `delete`

类似于 `new`，我们也可以重载 `delete` 来释放在特定内存位置上通过定位 `new` 创建的对象。定位 `delete` 只是调用普通的 `delete` 或者手动调用析构函数来释放对象，但它不会释放传入的内存地址，因为内存是由用户管理的。

定位 `delete` 语法：

```cpp
void operator delete(void* ptr, void* place) {
    // 通常我们不释放内存，仅调用析构函数
    static_cast<MyClass*>(ptr)->~MyClass();
}
```

在定位 `delete` 的实现中，我们不会释放内存，因为它是由用户预先管理的内存空间。如果需要手动释放内存，则通常是在内存池管理的上下文中。

2.3 例子：定位 `delete`

```cpp
#include <iostream>

class MyClass {
public:
    MyClass() { std::cout << "Constructor called\n"; }
    ~MyClass() { std::cout << "Destructor called\n"; }
};

int main() {
    char buffer[sizeof(MyClass)];  // 创建足够大的内存空间

    // 在 buffer 上通过定位 new 创建一个 MyClass 对象
    MyClass* obj = new (buffer) MyClass;

    // 手动调用析构函数
    obj->~MyClass();
    return 0;
}

```

在这个例子中，我们没有调用 `delete` 来释放 `buffer` 内存，因为 `buffer` 是由用户管理的。如果你使用内存池或其他特定的内存管理策略，通常需要手动释放内存。

5. 虚拟化非成员函数

“virtualizing non-member functions”的含义

妙哉

在 C++ 中，`<<` 和 `>>` 操作符的重载通常是 **非成员函数**，这意味着它们通常不属于任何类的成员函数，而是与流对象一起定义的独立函数。将这些非成员函数“虚拟化”（**virtualizing**）指的是使它们能够支持**多态性**，即通过继承和虚函数机制，允许在子类对象上调用合适的重载版本。

例1：ppt

例2：

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

class Shape {
public:
    virtual void print(std::ostream& os) const = 0;  // 定义一个虚拟的接口函数
```

```
    virtual ~Shape() = default;  // 确保派生类可以正确析构
};

// 圆形类
class Circle : public Shape {
public:
    void print(std::ostream& os) const override {
        os << "Circle";
    }
};

// 矩形类
class Rectangle : public Shape {
public:
    void print(std::ostream& os) const override {
        os << "Rectangle";
    }
};

// 重载 << 操作符
std::ostream& operator<<(std::ostream& os, const Shape& shape) {
    shape.print(os);  // 调用虚函数
    return os;
}

int main() {
    Circle c;
    Rectangle r;
    Shape* shapes[2] = {&c, &r};

    for (Shape* shape : shapes) {
        cout << *shape << endl;  // 多态调用
    }

    return 0;
}
```

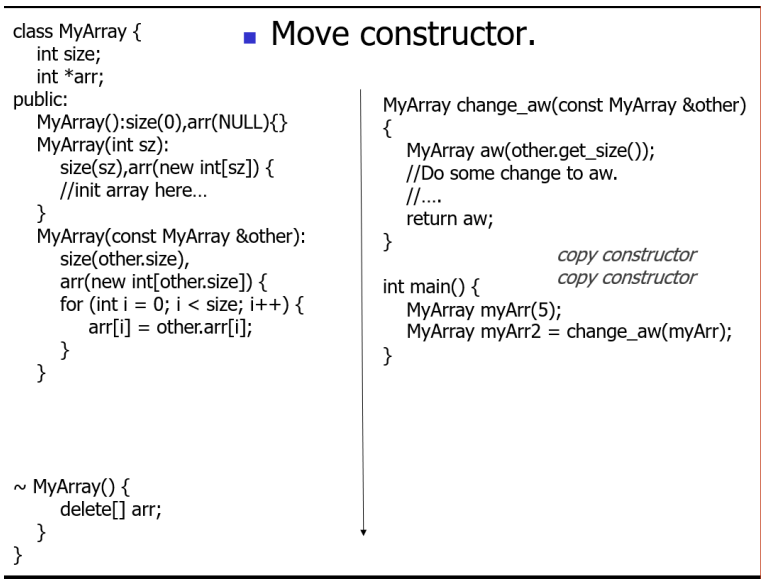
6. 虚拟化构造者

```
virtual NLComponent * clone() const = 0;

virtual TextBlock * clone() const
     {  return new TextBlock(*this); }

```

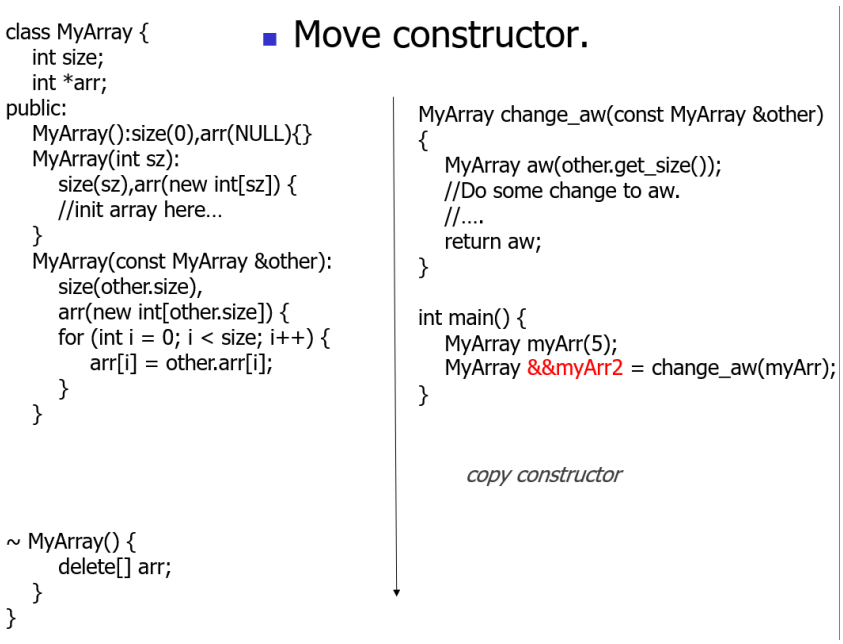
```
virtual Graphic  * clone() const
      {  return new Graphic (*this); }

NewsLetter::NewsLetter( const NewsLetter& rhs)
      {  for ( list<NLComponent *>::iterator it=rhs.component.begin();
   it != rhs.component.end(); ++it )
                component.push_back((*it)->clone());
       }
```

7. 右值&移动构造

（1）拷贝构造 绑定到新对象



（2）拷贝构造 绑定到右值引用



（3）移动构造

- Move constructor.

```
class MyArray {
    int size;
    int *arr;
public:
    MyArray():size(0),arr(NULL){}
    MyArray(int sz):
        size(sz),arr(new int[sz]) {
        //init array here…
    }
    MyArray(const MyArray &other):
        size(other.size),
        arr(new int[other.size]) {
        for (int i = 0; i < size; i++) {
            arr[i] = other.arr[i];
        }
    }
   MyArray (MyArray &&other):
         size(other.size), arr(other.arr) {
         other.arr = NULL;
   }
~ MyArray() {
        delete[] arr;
    }
}
```

```
MyArray change_aw(const MyArray &other)
{
    MyArray aw(other.get_size());
    //Do some change to aw.
    //….
    return aw;
}

int main() {
    MyArray myArr(5);
     MyArray myArr2 = change_aw(myArr);
}
```

copy constructor
move constructor

元编程

Meta programming

```
template <int N>
struct factorial {
    enum { value = N * factorial<N - 1>::value };
};

template <>
struct factorial<0> {
    enum { value = 1 };
};
```

```
std::cout << factorial<5>::value;
```

```cpp
template<int  N>
int fibonacci() {
  return fibonacci<N-1>() + fibonacci<N-2>();
}
template<>
int fibonacci<1>() { return 1; }
template<>
int fibonacci<0>() { return 0; }
```

Meta programming

```cpp
template <int N>
struct factorial {
    enum { value = N * factorial<N - 1>::value };
};

template <>
struct factorial<0> {
    enum { value = 1 };
};
```

std::cout << factorial<5>::value;

----------after C++11

```cpp
template <int N>
struct factorial1 {
    constexpr static int value = N * factorial<N - 1>::value;
};

template <>
struct factorial1<0> {
    constexpr static int value = 1;
};
```

函数回调

function本质上是类模板

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

int add(int a, int b){
    return a + b;
}
struct ADD
{    int operator()(int a, int b){
        return a + b;
    }
};

int main()
{
    map<string, function<int(int, int)>> opmap =
    {  {"fun_pointer", add},
       {"operator()", ADD()},
       {"lambda", [](int a, int b)->int {return a + b; }} ,
       {"bind",NewAdd}
    };

     cout << opmap["fun_pointer"](1, 2) << endl;
     cout << opmap["operator()"](3, 4) << endl;
     cout << opmap["lambda"](5, 6) << endl;
     cout << opmap["bind"](7, 8) << endl;
    return 0;
}
```

```
auto newCallable = bind(callable, arg_list);

int Add3(int a,int b, int c)
{
    cout << a << " " << b << "  " << c << endl;
    return a + b – c;
}
```

```
auto NewAdd = bind(Add3, placeholders::_1, 0, placeholders::_2);
auto NewAdd = bind(Add3, placeholders::_2, 0, placeholders::_1);
```

左值引用和右值引用

左值引用 (`T&`):

- 绑定的是**左值**，即**有名称**且**可以在表达式的左侧出现**的对象。左值通常表示**持久存在**的对象，**可以被修改**，并且其**生命周期持续到作用域结束**。
- 例如：变量、数组元素、解引用指针等。

右值引用 (`T&&`):

- 绑定的是**右值**，即**临时的、无名称**的对象，通常是一个**可以被移动的**对象。右值引用允许修改和转移对象的资源，但它们的生命周期通常较短，**仅存在于当前的表达式中**。
- 例如：字面量常量、临时对象、函数返回的临时值等。

Free malloc new delete

1.1 `new` / `delete` (C++)

- `new`：不仅分配内存，还调用对象的构造函数来初始化对象。
- `delete`：不仅释放内存，还调用对象的析构函数，确保资源的正确释放。

`new` 和 `delete` 专为面向对象的 C++ 设计，可以处理类的构造和析构，同时确保内存的正确管理。

1.2 `malloc` / `free` (C)

- `malloc(size_t size)`：仅分配指定大小的内存块，不会调用构造函数。分配的内存是未初始化的。
- `free(void* ptr)`：仅释放内存块，不会调用析构函数。

`malloc` 和 `free` 是 C 语言中的内存管理函数，它们仅负责内存的分配和释放，而不涉及对象的构造和析构。它们更低级别，仅用于管理原始内存。

顶层/底层const

```
const int* p1;   // 底层 const，p1 指向的内容是 const，不能修改
int const* p2;   // 同上，底层 const
int* const p3;   // 顶层 const，表示 p3 是 const，不能指向其他地址
const int* const p4; // 顶层 const 和底层 const 同时存在
```

用法注意：

cpp 复制代码

```cpp
int x = 10;
int y = 20;
const int* p1 = &x;  // 底层 const，不能通过 p1 修改 x
// *p1 = 30;  // 错误，p1 指向的内容是 const
p1 = &y;    // 合法，p1 本身可以被修改

int* const p2 = &x;  // 顶层 const，表示 p2 本身不能修改
*p2 = 30;   // 合法，可以通过 p2 修改 x
// p2 = &y;  // 错误，p2 是顶层 const
```

函数匹配

6.6.1 实参类型转换

为了确定最佳匹配，编译器将实参类型到形参类型的转换划分成几个等级，具体排序如下所示：

1. 精确匹配，包括以下情况：
   - 实参类型和形参类型相同。
   - 实参从数组类型或函数类型转换成对应的指针类型（参见 6.7 节，第 221 页，将介绍函数指针）。
   - 向实参添加顶层 const 或者从实参中删除顶层 const。
2. 通过 const 转换实现的匹配（参见 4.11.2 节，第 143 页）。
3. 通过类型提升实现的匹配（参见 4.11.1 节，第 142 页）。
4. 通过算术类型转换（参见 4.11.1 节，第 142 页）或指针转换（参见 4.11.2 节，第 143 页）实现的匹配。
5. 通过类类型转换实现的匹配（参见 14.9 节，第 514 页，将详细介绍这种转换）。

Empty

C++编译系统赋予一个空类，如：class Empty{ }，哪些成员函数？（本题8分）

```
class Empty { };
class Empty {
Empty();
Empty(const Empty&);
~Empty();
Empty& operator=(const Empty&);
Empty *operator &();
const Empty* operator &() const;
};
```

为什么操作符重载

思想：自定义类型应和内置类型有看起来一样的操作

可读性、可扩展性

可以利用操作符的结合性、优先级规则等

重载特殊

- . .* :: ?: 不能被重载：

这些操作符的行为与语言的核心语法和语义密切相关，

允许重载会导致类型检查的复杂性，降低编译器的效率。重载这些操作符可能会导致代码的可读性和可维护性极差。

- || && 能重载但尽量不要重载

短路机制，重载后需要两个参数都计算，改变了语法，可能造成混乱

- = () [] 不能作为全局函数重载

因为这几个符号虽然是双目运算符，但不符合交换律，重载函数的第一个参数必须是对象，在全局函数中无法保证这一点

- << >> 必须作为全局函数重载

第一个参数不是自定义参数

- 全局函数重载的方式为什么必须存在
  - 有些操作符重载时第一个参数不是自定义类型参数
  - 灵活性。支持交换律
- 为什么赋值运算符重载函数不能继承
  - 父类和子类的属性方法不同，有可能造成混乱
  - 编译器自动为子类生成拷贝复制运算符重载函数，将父类的赋值运算符重载函数覆盖

多维数组——[]重载 ()重载

proxy class
Surrogate
多维

```
class Array2D
{  public:
      class Array1D
      {    public:
             Array1D(int *p) { this->p = p; }
             int& operator[ ] (int index) { return p[index]; }
             const int operator[ ] (int index) const { return p[index]; }
          private:
             int *p;
      };
      Array2D(int n1, int n2) { p = new int[n1*n2]; num1 = n1; num2 = n2; }
     virtual ~Array2D( ) { delete [ ] p; }
int *
      Array1D operator[ ] (int index) { return p+index*num2; }
      const Array1D operator[ ] (int index) const { return p+index*num2; }
  private:
      int *p;
      int num1, num2;
};
```

```
class Array2D
{    int n1, n2;
     int *p;
public:
      Array2D(int l, int c):n1(l),n2(c)
     {  p = new int[n1*n2];  }
     virtual ~Array2D() { delete[] p; }
     int& operator()(int i, int j)
    {
        return  (p+i*n2)[j];
    }
};
```

智能指针

```
Template <class T>
class auto_ptr
{  public:
        auto_ptr(T *p=0):ptr(p) {}
        ~auto_ptr() { delete ptr; }
        T* operator->()  const { return ptr;}
        T& operator *()  const { return *ptr; }
    private:
        T*  ptr;
};
```

异常处理

```
class MyExceptionBase {
};

class MyExceptionDerived: public MyExceptionBase {
};

void f(MyExceptionBase& e) {
        throw e;
}

int main() {
        MyExceptionDerived e;
        try {
                f(e);
        } catch (MyExceptionDerived& e) {
                cout << "MyExceptionDerived" << endl;
        } catch (MyExceptionBase& e) {
                cout << "MyExceptionBase" << endl;
        }
}
```

//输出 MyExceptionBase

因为返回值的时候进行了值拷贝

New delete重载（我觉得不会考）

为防止在频繁的new、delete的过程中形成过多的内存碎片、影响效率，请你实现一个基于new和delete重载的简易heap空间分配方案

内存池

```
#include <iostream>
#include <cstdlib>
#include <new> // for std::nothrow

class A {
private:
    static const size_t POOL_SIZE = 1024; // 定义池中对象的最大数量
    static char memoryPool[POOL_SIZE * sizeof(A)]; // 内存池
    static bool used[POOL_SIZE]; // 标记每个对象是否被使用

public:
    // 重载 new 操作符
    void* operator new(size_t size) {
        for (size_t i = 0; i < POOL_SIZE; ++i) {
            if (!used[i]) { // 找到一个未被占用的块
                used[i] = true;
                std::cout << "Allocating object from pool at index " << i << std::endl;
                return memoryPool + i * sizeof(A); // 返回内存块的地址
            }
        }
        // 如果池子满了，使用默认的全局 new
        std::cout << "Memory pool exhausted. Using global new." << std::endl;
        return ::operator new(size);
    }

    // 重载 delete 操作符
    void operator delete(void* ptr) {
        size_t offset = static_cast<char*>(ptr) - memoryPool;
        size_t index = offset / sizeof(A);

        if (offset >= 0 && index < POOL_SIZE && used[index]) {
            used[index] = false; // 标记为未使用
            std::cout << "Releasing object back to pool at index " << index << std::endl;
        } else {
            // 如果不属于内存池，调用全局 delete
            std::cout << "Releasing object using global delete." << std::endl;
            ::operator delete(ptr);
```

```
        }
    }

    // 模拟类的成员函数
    void show() {
        std::cout << "Object at address: " << this << std::endl;
    }
};

// 静态成员变量定义
char A::memoryPool[POOL_SIZE * sizeof(A)] = {};
bool A::used[POOL_SIZE] = {};

int main() {
    // 使用重载的 new 分配对象
    A* a1 = new A();
    A* a2 = new A();
    A* a3 = new A();

    a1->show();
    a2->show();
    a3->show();

    // 使用重载的 delete 释放对象
    delete a1;
    delete a2;
    delete a3;

    // 分配超过池大小的对象，测试全局 new
    for (int i = 0; i < 1026; ++i) {
        A* obj = new (std::nothrow) A();
        if (!obj) {
            std::cout << "Allocation failed at iteration " << i << std::endl;
            break;
        }
        delete obj;
    }

    return 0;
}

```

思想

RAII

SFINAE

RITT

SIMPLE

CONST

自定义类型应和内置类型有看起来一样的操作

没看懂

generic2

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <memory.h>

typedef struct {
    void* elems;
    int elemSize;
    int logicalLen;
    int allocLen;
    void (*showFun)(void*);

} Stack;

void StackNew(Stack * s,int elemSize, void (*print)(void*))
{  assert(elemSize > 0);
    s->elemSize = elemSize;
    s->logicalLen = 0;
    s->allocLen = 4;
    s->elems = malloc(s->allocLen * elemSize);
    s->showFun = print;
    assert(s->elems != NULL);
}

void StackPush(Stack* s, void* elemAddr)
{
    if (s->logicalLen == s->allocLen)
    {
        s->allocLen *= 2;
        s->elems = realloc(s->elems, s->allocLen * s->elemSize);
        assert(s->elems != NULL);
    }

    void* target = (char*)s->elems + s->logicalLen*s->elemSize;
    memcpy(target, elemAddr, s->elemSize);
    s->logicalLen++;
}
```

```
void StackPop(Stack* s,void* elemAddr){
    assert(s->logicalLen > 0);

    s->logicalLen--;
    void* source = (char*)s->elems + s->logicalLen * s->elemSize;
    memcpy(elemAddr, source, s->elemSize);
}
```

```
void StackPrint(Stack* s) {
    if (s->logicalLen == 0)
    {    printf(" Empty \n");
         return;
    }

    for (int i = 0; i < s->logicalLen; i++)
        s->showFun((char *)s->elems + i * s->elemSize);
}
```

```
void StackCharPrint(void* p) {
    char** s_p = (char **)p;

    printf("%0x : %s\n", (int*)s_p, *s_p);
}
void StackIntPrint(void *p){
    int* i_p = (int*)p;
    printf("%0x : %d\n",i_p, *i_p);
}
```

```
int i[6] = {1,2,3,4,5,6 };
Stack iStack;
StackNew(&iStack,sizeof(int), StackIntPrint);

StackPush(&iStack, &(i[0]));
……
StackPrint(&iStack);

int x;
StackPop(&iStack, &x);
printf("Pop element: %d\n", x);

StackDispose(&iStack);
```

```
const char* s[] = { "aa","bbb","cc","dd","e","ffffff"};
Stack sStack;
StackNew(&sStack, sizeof(char *), StackCharPrint);

StackPush(&sStack, &(s[0]));
……
StackPrint(&sStack);

char *tmp;
StackPop(&sStack, &tmp);
printf("Pop element: %s\n", tmp);

StackDispose(&sStack);
```

```
void CharsFree(void* p) {
  char** s_p = (char**)p;
  free(*s_p);
}
```

```
void StackDispose(Stack* s)
{  if (s->freeFun != NULL)
        for (int i = 0; i < s->logicalLen; i++)
           s->freeFun((char*)s->elems + i * s->elemSize);
   free(s->elems);
   s->logicalLen = 0;
}
```

CharsFree

```
StackNew(&sStack, sizeof(char*), StackCharPrint  );
char* s;
for (int i = 0; i < 6; i++) {
    s = (char *)malloc(128 * sizeof(char));
    sprintf_s(s,128,"here is  %d  ",i);
    StackPush(&sStack, &s);
}

char *tmp;
StackPop(&sStack, &tmp);
printf("Pop element: %s\n", tmp);
free(tmp);

StackDispose(&sStack);
```

?

```
typedef struct {
    void* elems;
    int elemSize;
    int logicalLen;
    int allocLen;

    void (*showFun)(void*);
     void (*freeFun)(void*);

} Stack;
```

```
void StackNew(Stack * s,int elemSize, void (*print)(void*)=NULL,
                                  void (*dispose)(void *) = NULL)
{  assert(elemSize > 0);
    s->elemSize = elemSize;
    s->logicalLen = 0;   s->allocLen = 4;
    s->elems = malloc(s->allocLen * elemSize);
    s->showFun = print;
    s->freeFun = dispose;
    assert(s->elems != NULL);
}
```