类

1.变量的作用域：

2.类的作用域：

类定义的作用域等同于变量

成员函数的{}外定义形式：

*成员函数{}**外定义形式**不要在头文件中定义：

3.内联函数：

4.成员函数里的this指针：

示例：

5.常成员函数：

常成员函数不允许改变对象状态的语句出现。

修改权限不能放大只能缩小：

6.命名细节：

7.函数重载：

定义：

目的：

重载依据：

说明：

举例：

1.**常量指针重载**

2.常引用重载

3.常量和变量不构成重载

4.成员函数重载

5.常成员函数重载

6.相容类型使用重载（尽量避免）

8.函数默认参数：

第一种方式：

第二种方式：

规则：默认从右端开始设置，匹配从左端开始：

对重载的影响：

1.实现类似重载的调用效果

2.对重载造成二义性冲击

9.名空间：

基础定义使用：

{}外定义和合并定义：

名空间的借用：

名空间的嵌套：

10.构造函数与析构函数：

基础定义使用：

构造函数重载：

一般重载：

默认参数引起的构造函数重载：

构造函数执行的顺序：

复用情况：

（1）组合

（2）继承

时空情况：

（1）空间：全局 局部（2）时间：动态 静态

构造函数&析构函数的属性影响

开辟和销毁堆上的对象数组时，构造函数和析构函数工作情况：

组合类的构造函数如何工作:

错误案例：

正确案例：
成员初始化列表的顺序并不影响构造函数执行的顺序，而是取决于成员在类中的声明顺序：
没有自定义构造函数时，系统提供“默认构造函数”的情况：
1.被组合类有构造函数
2.祖先类有构造函数
3.有虚函数
4.虚继承
默认拷贝构造函数：
手写拷贝构造函数：
内存分类
A.代码区（code area)
B.数据区
1.全局数据区(data area)
2.栈区(stack area)
3.堆区(heap area) 自由存储区
malloc和free、new和delete的区别
浅拷贝：
深拷贝：
构造和拷贝的区分：
explicit关键字：
类型转换规则：
11.友元：
友元的作用：开放私有权限给特定成员：
友元函数：
友元类：
友元的特点：
12.运算符重载：
C语言本身为函数机制：
加法重载：
方式一：类外
方式二：类内
重载规则：
不提倡违背本意和改变参数个数：
运算符重载时参数个数不可以超过原来数目
参数顺序很重要（参数和返回类型可以改变，运算顺序和优先级不能改变）：
重载运算符内部的同名运算符维持原来的操作
参数都是基本类型时不能重载：
不允许创建新的运算符
重载运算符函数返回类型不能为空：
重载运算符=函数类型应为引用，值不可以：
运算符位置规定：
与new,delete混用：
前++和后++运算符重载：
前++：
后++：问题：为什么函数返回值没有引用?
类型转换示例：
13.静态成员：
静态成员在main之前构造，生命周期等同于全局对象。
静态成员的空间不包含在对象内：
使用场景：
多个场合修改同一个数据
争夺标记，类似令牌
链首指针：
静态成员函数：
静态成员函数没有this指针
只能有一个对象的类：
实现单向关联：

类

1.变量的作用域：

```
#include <iostream>
using namespace std;
int a = 1;
int main() {
    void f(void);
    cout << a << endl;
    int a = 2;
    cout << a << endl;
    cout << ::a<<endl;//全局对象被屏蔽后的强行访问
    {
        int a = 3;
        cout << a << endl;
    }
    cout << a << endl;
    f();
    cout << a << endl;
}
void f() {
    cout << a << endl;
    int a = 4;
    cout << a << endl;
}
```

result:1 2 1 3 2 1 4 2

2.类的作用域：

类定义的作用域等同于变量

```
#include <iostream>
using namespace std;
class A{};
void f() {
    class B {};
    A s; B t;
    {
        class C {};
        A s; B t; C u;
    }
```

```
    class A{};
    A s1;B t1;C u1;//若A s;错误，同一变量不能在一个作用域被重复定义;C u1,错误
}
int main() {
    A s; B t; C u1;// B t; C u1;错误
    return 0;
}
```

成员函数的{}外定义形式：

好处：类体定义一目了然地增强可读性

```
class Date {
    int year, month, day;
public:
    void set(int y, int m, int d);
    bool LeapYear();
};
void Date::set(int y, int m, int d) {
    year = y;
    month = m;
    day = d;
}
bool Date::LeapYear() {
    return !(year % 400) || (!(year % 4) && (year % 100));
}
```

*成员函数{}外定义形式不要在头文件中定义：

原因：声明可以声明无数次，但定义只能有一次

2.h

```
#include <iostream>
using namespace std;
class student {
public:
    int score;
    void p(void);
private:
    int age;
};
void student::p() {
    age += 1;
}
```

1.cpp

```
#include "2.h"
void f() {
    student a, b;
    a.score = 90;
    b.score = 59;
    a.p();
    b.p();
}
```

3.cpp

```
#include "2.h"
int main() {
    student a;
    a.score = 80;
    cout << a.score << endl;
    a.p();
    return 0;
}
```

void student::p()被重复定义

这就是为什么说成员函数外定义要在每个源文件里单独定义。而内定义方式则没事（实验验证）

3.内联函数：

何为内联函数？inline、调用之前inline显性定义或者声明。

晓得成员函数一般在类内定义，一般默认为内联函数。复杂语句（循环、多分支）不允许（如出现则不视为内敛）。

内联函数的目的是增强系统效率。

4.成员函数里的this指针：

成员函数不属于对象：

成员函数里的代码如何知道调用者是谁？ 内部定义：A* const this=&对象X

this是一个指针常量。

所以this=new Student就是错的。

```
class student {
public:
    void p(void);
    float score;
private:
    string name;
    int age;
    void x() {}
```

```
};
void x() {}
void student::p() {
    int age = 1;
    age = 5;
    this->age = 5;//这里的this是必须的
    student::age = 5;
    x();
    ::x();
}
```

示例：

```
#include <iostream>
using namespace std;
class student {
public:
    student* p(void);
    student& q(void);
    float score;
private:
    string name;
    int age;
};
student* student::p() {
    age += 2; return this;
}
student& student::q() {
    age += 2; return *this;
}
int main() {
    student a;
    (a.p())->score = 80.5;
    (a.q()).score = 93;
    return 0;
}
```

5.常成员函数：

常成员函数不允许改变对象状态的语句出现。

本质是把this指针从指针常量修改成了指向常量的指针常量。const A* const this=&对象X

格式：在参数列表和第一个大括号之间+const

修改权限不能放大只能缩小：

```
class student {
public:
    void p(int a)const { cout << age << a << endl; }
    void r(int k) { age = k; }
private:
    int age;
};
void f(const student& a) {
    a.p(5);//正确
    a.r(5);//错误，因为r有修改权限，权限被放大
}
```

6.命名细节：

```
#include <iostream>
using namespace std;
class student {
    int a;
};
int main() {
    int student = 1;
    class student a;
    return 0;
}
```

可以但不提倡

7.函数重载：

定义：

就是指**在同一作用域内**，可以有一组具有相同函数名，不同参数列表（参数个数、类型、顺序）的函数，该组函数被称为重载函数。

目的：

重载函数通常用来声明一组功能相似的函数，减少了函数名的数量，避免了名字空间的污染，对于程序的可读性有很大的好处。

重载依据:

1.参数类型（包括const引用和const指针）

2.参数个数

3.参数顺序

4.常成员函数

说明:

1.**返回值类型不能够作为重载依据**

2.**要求同一个作用域**

举例:

1.常量指针重载

```
#include <iostream>
using namespace std;
class A{};
class B{};
void p(A const* s) {
    cout << 2 << endl;
}
void p(A* s) {
    cout << 1 << endl;
}//常量指针重载
int main() {
    const A x=...;
    p(&x);
    A y = ...;
    p(&y);
    return 0;
}
//*若无第二个p函数，两者都调用第一个，但若无第一个p函数，只有第二个对象可以调用，第一个调用是错误的，因为权限只能缩小不能扩大。
```

2.常引用重载

```
#include <iostream>
using namespace std;
class A{};
class B{};
void p(A const& s) {
    cout << 2 << endl;
```

```
}
void p(A& s) {
    cout << 1 << endl;
}//常引用重载
int main() {
    const A x=...;
    p(x);
    A y = ...;
    p(y);
    return 0;
}
//*若无第二个p函数，两者都调用第一个，但若无第一个p函数，只有第二个对象可以调用，第一个调用是错误的，因为权限只能缩小不能扩大。
```

3.常量和变量不构成重载

```
void p(A const s) {
    cout << 2 << endl;
}
void p(A s) {
    cout << 1 << endl;
}
//不是函数重载
```

4.成员函数重载

student.h

```
#include <iostream>
using namespace std;
class Student {
public:
    void q(void);
    void q(int a);
protected:
    float score;
    float m;
};
void Student::q() {
    cout << "aa" << endl;
}
void Student::q(int a) {
    cout << "bb" << endl;
}
```

1.cpp

```
#include "student.h"

int main() {
    Student a;
    a.q();
    a.q(2);
    return 0;
}
```

5.常成员函数重载

实际是Class *const this 和 const Class *const this之间的重载

student.h

```
#include <iostream>
using namespace std;
class Student {
public:
    void q(void);
    void q(void)const;
protected:
    float score;
    float m;
};
void Student::q() {
    cout << "aa" << endl;
}
void Student::q() const{
    cout << "bb" << endl;
}
```

1.cpp

```
#include "student.h"

int main() {
    Student a;//需要初始化
    a.q();
    Student const b = a;
    b.q();
    return 0;
}
```

6.相容类型使用重载（尽量避免）

```
#include <iostream>
using namespace std;
void p(double a) {
    cout << "aa" << endl;
}
void p(int a) {
```

```
    cout << "bb" << endl;
}
int main() {
    p(10);
    p(10.9);
    p('a');//相容类型char使用int重载
    return 0;
}
```

规则：首先严格匹配，其次相容类型匹配，最后用户定义类型转换，**尽量避免类型相容二义性**，名称压轧技术实现原理。

8.函数默认参数：

第一种方式：

```
#include <iostream>
using namespace std;
class A{};
//A* x(int i) {
//  return (new A[i]);
//}
A* x(int i = 30) {
    return (new A[i]);
}
int main() {
    A* p = x();
    A* q = x(31);
    return 0;
}
```

第二种方式：

```
#include <iostream>
using namespace std;
class A{};
A* x(int = 30);
A* x(int i) {
    return (new A[i]);
}
int main() {
    A* p = x();
    A* q = x(31);
    return 0;
}
```

规则：默认从右端开始设置，匹配从左端开始：

```
int x(int i=10,int j){//错误
    for(;i>j;i--){
    cout<<"**"<<endl;
    }
}
```

对重载的影响：

注意依靠参数个数进行重载函数中默认参数造成改变参数个数的效果

1.实现类似重载的调用效果

```
#include <iostream>
using namespace std;
int x(int i, int j = 11, int k = 12) {
    cout << i <<' '<< j <<' ' << k;
}
int main() {
    x();//错误
    x(1);//1 11 12
    x(1, 2);//1 2 12
    x(1, 2, 3);//1 2 3
    return 0;
}
```

2.对重载造成二义性冲击

```
#include <iostream>
using namespace std;
void x(int i = 10) {
    cout << i << "11" << endl;
}
void x() {
    cout << "22" << endl;
}
int main() {
    x();//二义性
    return 0;
}
```

9.名空间：

基础定义使用：

```
#include <iostream>
using namespace std;
namespace zhs {
    void a(){
        cout << 1 << endl;
    }
}
int main() {
    zhs::a();
    return 0;
}
```

```
#include <iostream>
using namespace std;
namespace zhs {
    class aa {
    public:
        aa(int a = 1) { id = a; }
        int id;
    private:
        int num;
    };
    int age;
    double score;
}
namespace ls {
    class aa {
    public:
        aa(int x = 1) { age = x; }
        int age;
    };
}
int main() {
    ls::aa a;
    zhs::score = 90.1;
    cout << zhs::score << endl;
    cout << a.age << endl;
    return 0;
}
```

{}外定义和合并定义：

```
#include <iostream>
using namespace std;
namespace zhs {
    class aa;
}
class zhs::aa {//{}外定义
public:
    aa(int a = 1) { id = a; }
    int id;
```

```
private:
    int num;
};
namespace zhs {//合并定义
    int age;
    double score;
}
int main() {
    zhs::aa a;
    zhs::score = 90.5;
    cout << a.id <<'\n'<< zhs::score << endl;
    return 0;
}
```

名空间的借用：

```
#include <iostream>
using namespace std;
namespace zhs {
    int age=1;
    double score;
}
namespace ls {
    using zhs::age;
    void a();
}
void ls::a() { cout << age << endl; }
int main() {
    ls::a();
    return 0;
}
```

```
#include <iostream>
using namespace std;
namespace zhs {
    int age=1;
    double score;
}
namespace ls {
    using namespace zhs;
    void a();
}
void ls::a() { cout << age << endl; }
int age = 2;//放到上面报错
int main() {
    ls::a();
    cout << zhs::age << ::age;
    return 0;
}
```

```
#include <iostream>
using namespace std;
namespace zhs {
    int age=1;
    double score;
}
int main() {
    namespace zhangsan = zhs;
    cout << zhangsan::age;
    return 0;
}
```

名空间的嵌套：

```
#include <iostream>
using namespace std;
namespace zhs {
    int age=1;
    double score;
    namespace zhangsanA {//嵌套
        int age=2;
    }
}
int main() {
    cout << zhs::zhangsanA::age;
    return 0;
}
```

10.构造函数与析构函数：

基础定义使用：

构造函数参数没有限制，因此可以重载；

析构函数没有参数，因此不可以重载；

```
#include <iostream>
using namespace std;
#include <string>
class Student {
public:
    Student();
    ~Student();
protected:
    string name;
    int age;
};
Student::Student() { age = 5; }
```

```
Student::~Student() {
    cout << "对象结束了";
}
int main() {
    Student a;
    //a.Student();错误
    a.~Student();//正确但不提倡
    //result:2*"对象结束了";
}
```

构造函数重载：

一般重载：

```
#include <iostream>
using namespace std;
#include <string>
class Student {
    string name; int credit;
public:
    Student(string pName) {
        cout << "第一个运行了" << endl;
        name = pName;
    }
    Student(string pName,int i) {
        cout << "第二个运行了" << endl;
        name = pName;
        credit = i;
    }
};
int main() {
    Student t("张三", 6);
    Student s("李四");
    Student f;//错误
    //Student f();也错误,编译器认为这是一个函数声明
    return 0;
}
```

默认参数引起的构造函数重载：

```
#include <iostream>
using namespace std;
#include <string>
class Student {
    string name; int credit;
public:
    Student(string pName="张三") {
        name = pName;
    }
    Student(string pName,int i) {
        name = pName;
```

```
        credit = i;
    }
};
int main() {
    Student t("王五", 6);
    Student s("李四");
    Student f;//正确
    return 0;
}
```

构造函数执行的顺序：

复用情况：

（1）组合

对象创建时调用构造函数入栈，对象销毁时调用析构函数出栈。

```
#include <iostream>
using namespace std;
#include <string>
class Student {
    float gpa; int credit;
public:
    Student(){
        cout << "constructing student.\n";
        credit = 4;
        gpa = 3.5;
    }
    ~Student() {
        cout << "~Student\n";
    }
};
class Teacher {
    string name;
public:
    Teacher() {
        cout << "constructing teacher.\n";
    }
    ~Teacher() {
        cout << "~Teaccher\n";
    }
};
//构造函数执行顺序
class TutorPair {
    Student s;
    Teacher t;
    int noMeetings;
public:
    TutorPair() {
        cout << "constructing tutorpair.\n";
        noMeetings = 1;
```

```
    }
    ~TutorPair() {
        cout << "~TutorPair\n";
    }
};
int main() {
    TutorPair p;
    cout << "back in main.\n";
    return 0;
}
```

result:

constructing student. constructing teacher. constructing tutorpair. back in main. ~TutorPair ~Teaccher ~Student

（2）继承

时空情况：

（1）空间：全局 局部（2）时间：动态 静态

全局对象main之前构造，程序结果时析构。

静态对象在首次调用时构造一次，程序结束析构

```
#include <iostream>
using namespace std;
class A {
    int s;
public:
    A(int k) {
        s = k;
        cout << k << "A is constructing\n";
    }
    ~A() {
        cout << s << "~A\n";
    }
};
int main() {
    cout << "main is running\n";
    A s(1);
    void f();
    f();
    f();
    return 0;
}
void f() {
    cout << "f is running\n";
    A s(2);
    static A t(3);
}
A t(4);
```

result:

4A is constructing main is running 1A is constructing f is running 2A is constructing 3A is constructing 2~A f is running 2A is constructing 2~A 1~A 3~A 4~A

局部对象之间以文本定义顺序为顺序（类的数据成员属于此种情况）。

全局对象如果分布在不同文件中，则构造顺序不确定。

A.h

```
#include <iostream>
using namespace std;
class A {
    int s;
public:
    A(int r = 0) {
        s = r;
        cout << 'A';
    }
};
```

B.h

```
#include <iostream>
using namespace std;
class B {
    A x;
public:
    B(A t) {
        x = t;
        cout << "B";
    }
};
```

1.cpp

```
#include "A.h"
A h(6);
void f() {
    //
}
```

2.cpp

```
#include "A.h"
#include "B.h"
extern A h;
B k(h);

int main() {

}
void g() {

}
//全局变量的构造顺序不确定
```

AAB/...

构造函数& 析构函数的属性影响

```
#include <iostream>
using namespace std;
class A {
public:
    int s;
private:
    A() {
        cout << "A is constructing\n";
    }
    ~A() {
        cout << "~A\n";
    }
};
int main() {
    A s;//出错
    A* p = new A;//出错
    return 0;
}
//构造函数&析构函数的属性影响
```

开辟和销毁堆上的对象数组时，构造函数和析构函数工作情况：

```
#include <iostream>
using namespace std;
class Student {
    int value;
public:
    Student() {
        cin >> value;
    }
    ~Student() {
```

```
        cout << value << endl;
    }
};
int main() {
    Student* p = new Student[5];
    cout << "Delete Begin" << endl;
    delete[]p;
    return 0;
}
```

1 2 3 4 5

result:

Delete Begin 5 4 3 2 1

```
#include <iostream>
using namespace std;
class Student {
    int value;
public:
    Student(int i) {
        value = i;
    }
    ~Student() {
        cout << value << endl;
    }
};
int main() {
    Student* p = new Student[5](80);//错误
    cout << "Delete Begin" << endl;
    delete[]p;
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Student {
    int value;
public:
    Student(int i) {
        value = i;
    }
    ~Student() {
        cout << value << endl;
    }
};
int main() {
    Student* p = new Student[5]{ 80,80,80,80,80 };//正确
    cout << "Delete Begin" << endl;
    delete[]p;
    return 0;
}
```

result:

Delete Begin 80 80 80 80 80

组合类的构造函数如何工作:

错误案例:

```
//类成员定义时不允许初始化，因为类仅仅是一个类型，而没有空间分配
#include <iostream>
using namespace std;
#include <string>
class StudentID {
    int value;
public:
    StudentID(int id=0) {
        value = id;
    }
};
class Student{
    string name;
    StudentID id(ssID);//错误
public:
    Student(string pName,int ssID=0) {
        name = pName;
    }
};
int main() {
    Student s("zhangsan", 218);
    return 0;
}
```

```
#include <iostream>
using namespace std;
#include <string>
class StudentID {
    int value;
public:
    StudentID(int id=0) {
        value = id;
    }
};
class Student{
    string name;
    StudentID id;
public:
    Student(string pName,int ssID=0) {
        name = pName;
        //add
        Student id(ssID);//临时对象而已，不能影响类的成员
    }
};
```

```
int main() {
    Student s("zhangsan", 218);
    return 0;
}
```

正确案例：

```
#include <iostream>
using namespace std;
#include <string>
class StudentID {
    int value;
public:
    StudentID(int id=0) {
        value = id;
    }
};
class Student{
    string name;
    StudentID id;
public:
    Student(string pName,int ssID=0):id(ssID) {//引入新语法,构造函数的初始化列表用于初始化类的成员变量，而不是赋值(这点对于引用型成员来说很重要，只能在这里初始化，而不能在函数体里赋值)
        name = pName;
    }
};
int main() {
    Student s("zhangsan", 218);
    return 0;
}
```

成员初始化列表的顺序并不影响构造函数执行的顺序，而是取决于成员在类中的声明顺序：

```
#include <iostream>
using namespace std;
#include <string>
class Student {
    int ID;
public:
    Student(int i) :ID(i) {
        cout << "constructing student.\n";
    }
    ~Student() {
        cout << "~Student\n";
    }
};
class Teacher {
    string name;
public:
```

```
    Teacher(string pName) :name(pName) {
        cout << "constructing teacher.\n";
    }
    ~Teacher() {
        cout << "~Teacher\n";
    }
};
class TutorPair {
    Student s;
    Teacher t;
    int noMeetings;
public:
    TutorPair(int i, int j, string p):noMeetings(i),t(p),s(j) {
        cout << "constructing tutorpair.\n";
    }
    ~TutorPair() {
        cout << "~TutorPair\n";
    }
};
int main() {
    TutorPair tp(4, 218, "John");
    cout << "back in main.\n";
    return 0;
}
```

result：

constructing student. constructing teacher. constructing tutorpair. back in main. ~TutorPair ~Teacher ~Student

因为s比t先声明，就先入栈

没有自定义构造函数时，系统提供“默认构造函数”的情况：

系统为每个类都提供默认构造函数是错的。

1.被组合类有构造函数

"被组合类"是指一个类作为另一个类的成员变量出现。

2.祖先类有构造函数

3.有虚函数

4.虚继承

默认拷贝构造函数：

```
//默认拷贝构造函数
class Student {
    int i;
public:
    Student(int k):i(k){}
    void p() {
        cout << i << endl;
    }
};
int main() {
    Student s(2023);
    s.p();
    Student t(s);
    t.p();
    //因为有默认拷贝
    Student m(2022);
    m.p();
    m = s;
    m.p();
    //因为有默认赋值运算
}
```

result:

2023 2023 2022 2023

手写拷贝构造函数:

```
class Student {
    int i;
public:
    Student(int k):i(k){}
    Student(Student const& s) {
        i = s.i * (-1);
    }
    void p() {
        cout << i << endl;
    }
};
int main() {
    Student s(18);
    s.p();
    Student t(s);
    t.p();
    Student k = s;//等价于Student k(s),所以也输出-18,并不是简单的k和s一样
    k.p();
    Student* q = new Student(s);
    q->p();
    //均调用了拷贝构造
    Student m(88);
    m = s;
    m.p();
```

```
    //调用赋值运算
    //总结：Student k=s和Student m;m=s;不一样
    return 0;
}
```

内存分类

A.代码区（code area)

B.数据区

1.全局数据区(data area)

2.栈区(stack area)

3.堆区(heap area) 自由存储区

malloc和free、new和delete的区别

1.malloc()和对应的free()函数并不属于语言本身，因此不能自动调用构造函数和析构函数，所以引入了new和delete。

2.malloc()和free()是函数，于malloc.h中声明；而new和delete是C++本身的内容。可以理解为关键字.

```
class aa {
    int id;
public:
    aa(int a = 1) {
        id = a;
    }
    void display() {
        cout << id << endl;
    }
    ~aa() {
        cout << "aa is completed." << endl;
    }
};
int main() {
    aa* p = new aa(9);
    p->display();
    aa* q = (aa*)malloc(sizeof(aa));
    q->display();//0xCD表明构造函数未调用
    delete p;
    free(q);//析构函数未调用
    return 0;
}
```

result:

9 -842150451 aa is completed.

注意：堆空间不伴随函数动态释放，程序员要自主管理

```
class aa {
    int id;
public:
    aa(int a = 1) {
        id = a;
    }
    void display() {
        cout << id << endl;
    }
    ~aa() {
        cout << "析构" <<id<< endl;
    }
};
aa& m() {
    aa* p = new aa(9);
    return (*p);
}
int main() {
    aa& s = m();
    s.display();
    return 0;//结果为9，析构未执行
}
```

```
class aa {
    int id;
public:
    aa(int a = 1) {
        id = a;
    }
    void display() {
        cout << id << endl;
    }
    ~aa() {
        cout << "析构" <<id<< endl;
    }
};
aa& m() {
    aa* p = new aa(9);
    //add
    delete p;
    return (*p);
}
int main() {
    aa& s = m();
    s.display();//结果为0xDD，析构已执行，result：析构9，-572662307
    return 0;
}
```

在你的代码中，虽然 `p` 被 `delete` 了，但你仍然试图返回并使用一个已经被释放的对象，这会导致未定义行为。在C++中，当你 `delete` 一个指针后，那个指针指向的内存区域变得无效，再访问它会导致不可预测的行为。

你目前的代码如下：

```
class aa {
    int id;
public:
    aa(int a = 1) {
        id = a;
    }
    void display() {
        cout << id << endl;
    }
    ~aa() {
        cout << "析构" <<id<< endl;
    }
};
aa& m() {
    aa* p = new aa(9);
    //add
    delete p;
    return (*p); // 返回已删除的对象引用，未定义行为
}
int main() {
    aa& s = m();
    s.display(); // 未定义行为
    return 0;
}
```

在这里，`delete p;` 之后，`p` 指向的内存已经被释放，所以 `return (*p);` 返回一个无效的引用，访问这个引用会导致未定义行为。

如何正确管理内存

如果你想手动管理内存并确保正确释放，同时不使用智能指针，你可以通过显式的分配和释放内存来管理生命周期。在这种情况下，不要在 `m` 函数中删除对象，而是由调用者负责删除。

修改后的代码如下：

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

class aa {
    int id;
public:
    aa(int a = 1) {
        id = a;
    }
    void display() {
        cout << id << endl;
    }
    ~aa() {
        cout << "析构" << id << endl;
    }
};

aa* m() {
    aa* p = new aa(9); // 动态分配对象
    return p; // 返回指针
```

```
}

int main() {
    aa* s = m(); // 获取指针
    s->display(); // 正常使用
    delete s; // 手动释放内存，调用析构函数
    return 0;
}
```

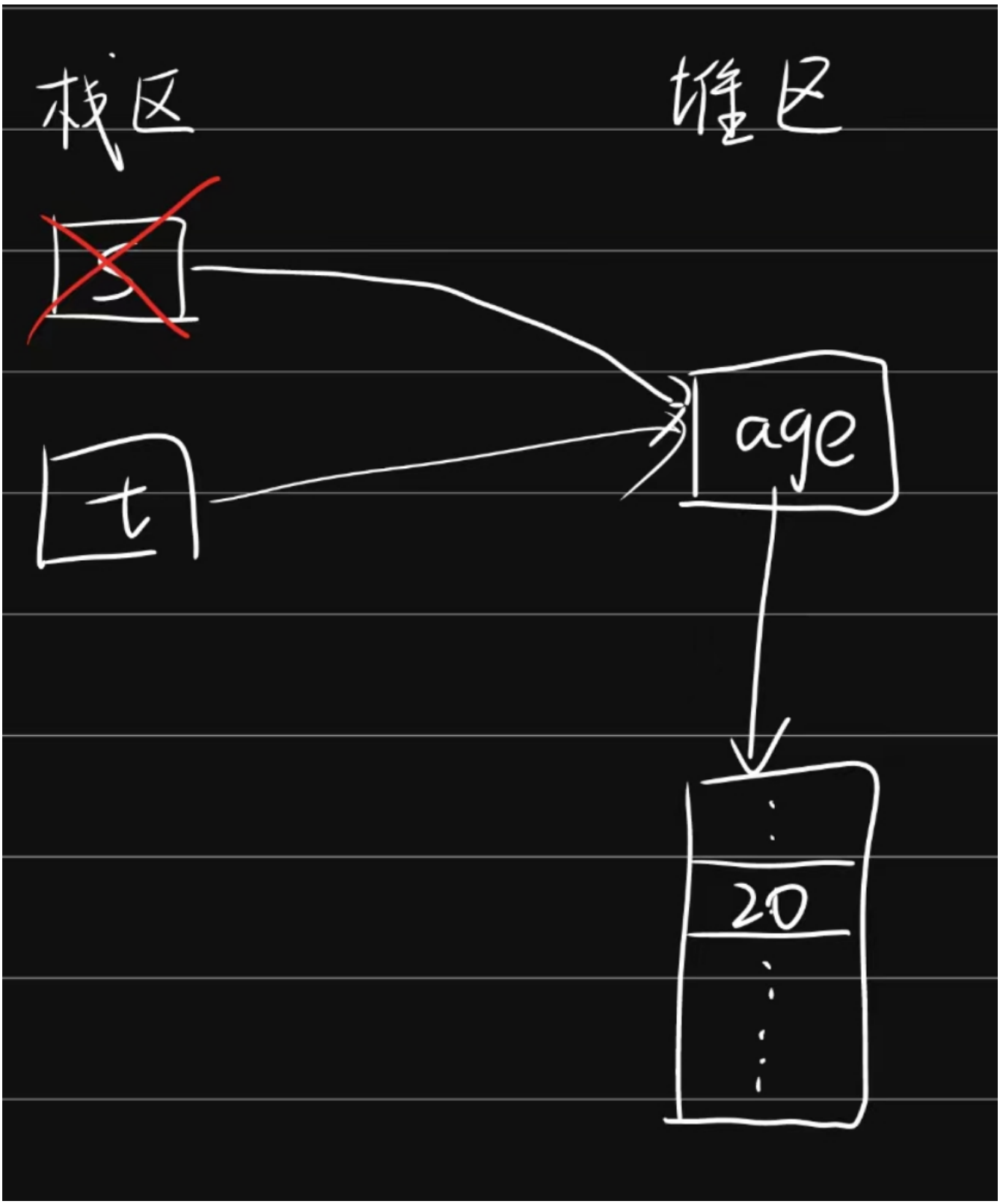
解释

1. `m` **函数**:
   - `aa* p = new aa(9);` 在堆上分配一个 `aa` 对象并返回指针。
   - 不在 `m` 函数中删除对象，而是将指针返回给调用者。
2. `main` **函数**:
   - `aa* s = m();` 获取动态分配对象的指针。
   - `s->display();` 调用 `display` 方法。
   - `delete s;` 在 `main` 函数中显式删除对象，释放内存并调用析构函数。

这种方式确保了对象在使用后被正确释放，避免了内存泄漏和未定义行为。注意，手动管理内存时需要小心处理，确保所有分配的内存都能被正确释放。

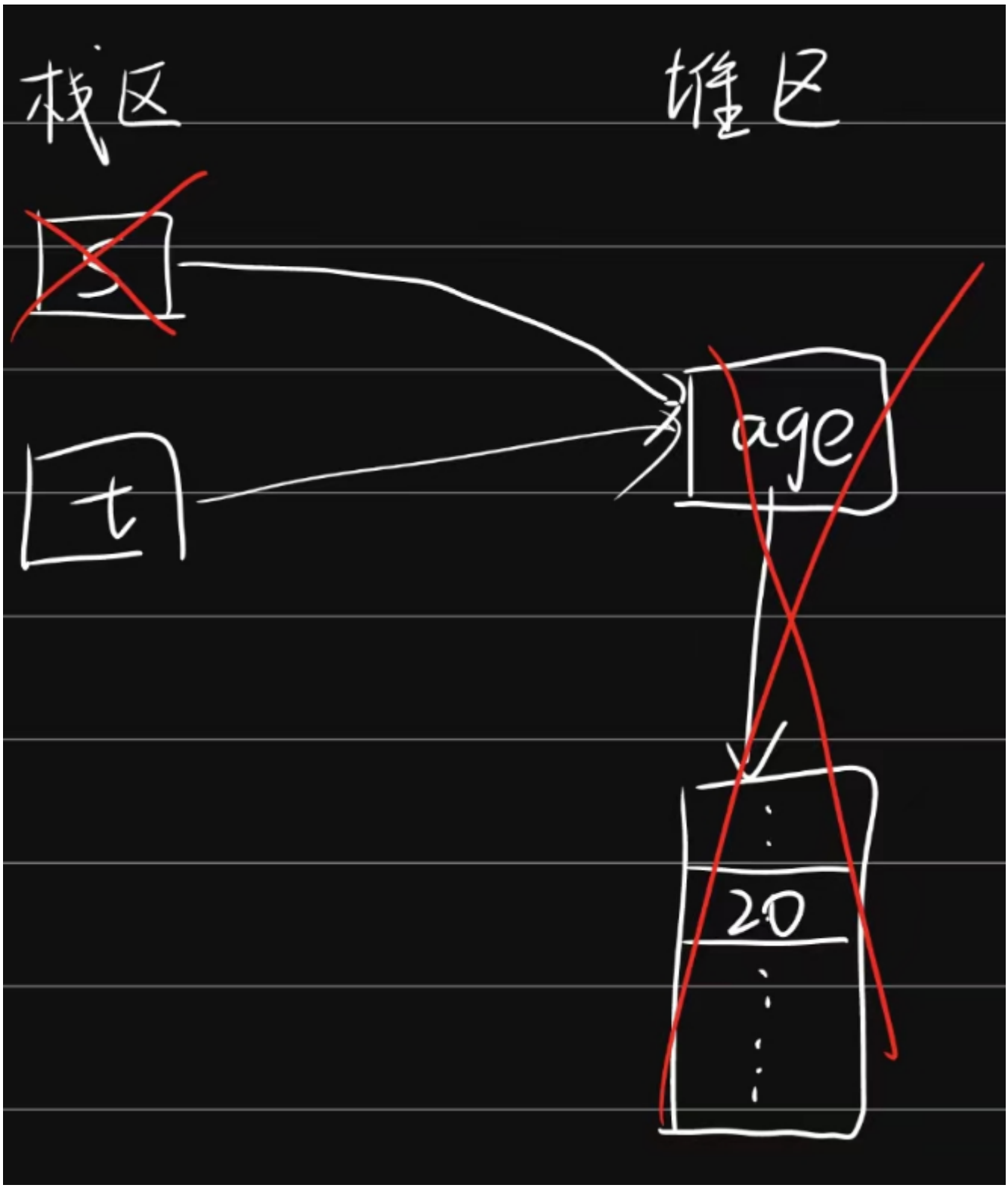
析构函数里delete的作用：

```
#include <iostream>
using namespace std;
class a {
public:
    a(int i = 9) {
        age = new int[i];
    }
    int* age;
};
int main() {
    a* s = new a;
    s->age[0] = 20;
    int* t = s->age;
    delete s;
    cout << t[0] << endl;
    return 0;
    //result:20
}
```




```
class a {
public:
    a(int i = 9) {
        age = new int[i];
    }
    int* age;
    ~a() {
        delete[]age;
```

```
    }
};
int main() {
    a* s = new a;
    s->age[1] = 20;
    int* t = s->age;
    delete s;//会自动调用相应的析构函数
    cout << t[1] << endl;
    return 0;
    //result:-572662307,即0xDD,析构已执行
}
```



浅拷贝：

aa.h

```
class aa {
    string* dept;
public:
    aa():dept(new string){}
    ~aa() { delete dept; }
    void set_dept(string d) {
        *dept = d;
    }
    void display() {
        cout << (*dept).c_str() << endl;
    }
};
```

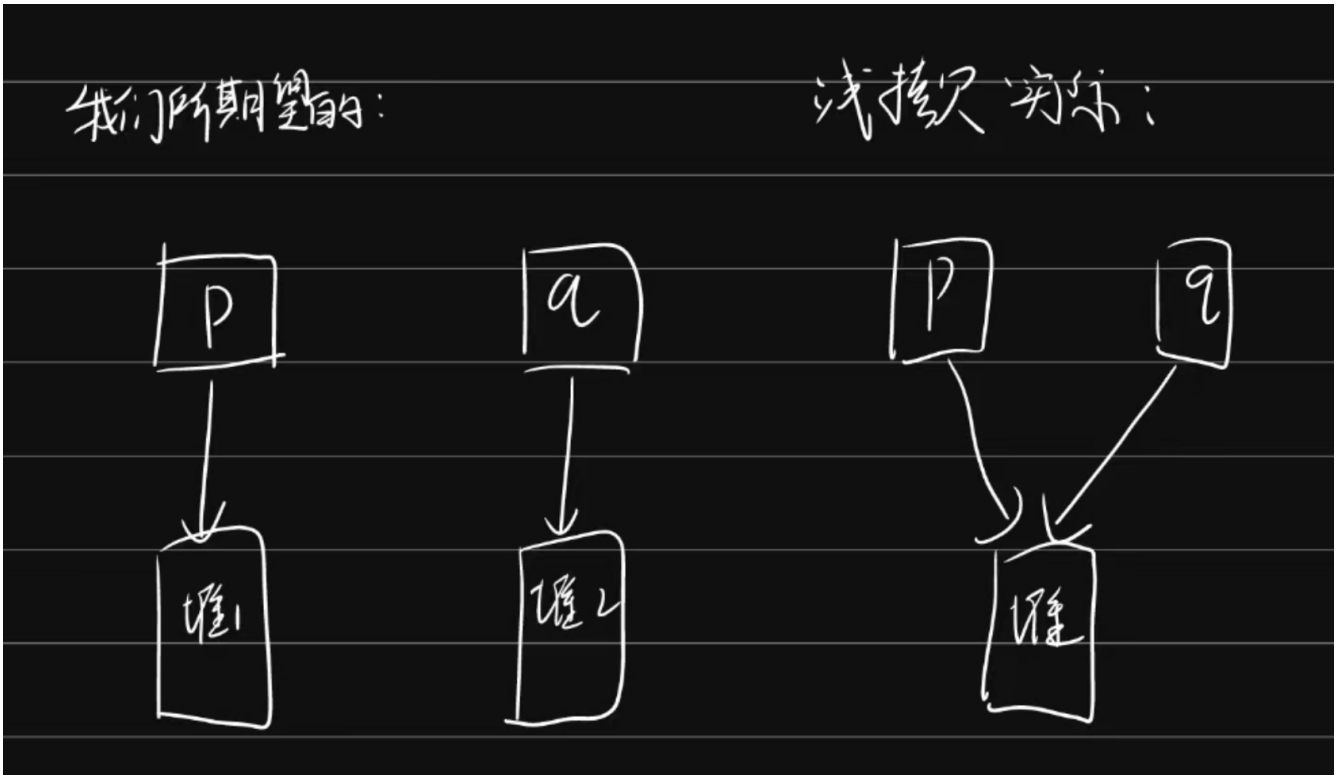
1.cpp

```
#include "aa.h"
int main() {
    aa p;
    p.set_dept("computer");
    p.display();
    aa q(p);//aa q=p;
    q.display();

    q.set_dept("software");
    p.display();
    q.display();
    return 0;
}
```

result:

computer computer software software



深拷贝：

aa.h

```
class aa {
    string* dept;
public:
    aa():dept(new string){}
    aa(aa const& s) {//拷贝参考时不改变参考的对象的状态
        dept = new string(*(s.dept));//仅仅参考s对象的字符串值，而不是整块的内存
    }
    ~aa() { delete dept; }
    void set_dept(string d) {
        *dept = d;
    }
    void display() {
        cout << (*dept).c_str() << endl;
    }
};
```

1.cpp

```
#include "aa.h"
int main() {
    aa p;
    p.set_dept("computer");
    p.display();
    aa q(p);//aa q=p;
    q.display();

    q.set_dept("software");
    p.display();
    q.display();
    return 0;
}
```

result:

computer computer computer software

引用成员也适用深拷贝，但意义不大且稍显复杂： aa.h

```
class aa {
    string& dept;
public:
    aa():dept(*(new string)){}
    aa(aa const& s):dept(*(new string(s.dept))) {//拷贝参考时不改变参考的对象的状态，引用初始化只能在这里而不是函数体内
    }
    ~aa() { delete &dept; }
    void set_dept(string d) {
        dept = d;
    }
    void display() {
        cout << dept.c_str() << endl;
    }
};
```

1.cpp

```
#include "aa.h"
int main() {
    aa p;
    p.set_dept("computer");
    p.display();
    aa q(p);//aa q=p;
    q.display();
    q.set_dept("software");//并没有改变引用本身，而是改变了引用对象的值从computer到software
    p.display();
    q.display();
    return 0;
}
```

result:

computer computer computer software

深拷贝实现了我们期望的效果。

构造和拷贝的区分：

```
class aa {
    int id;
public:
    aa(int a = 1):id(a) {
        cout << "构造" << endl;
    }
    aa(aa const& s) :id(s.id) {
        cout << "拷贝" << endl;
    }
};
int main() {
    aa m;//构造
    aa n(m);//拷贝
    aa o = m;//拷贝 和 aa o(m)等价
    aa s = 9;//构造 和 aa s(9)等价
    aa t;//构造
    t = 9;//*构造 赋值运算
    return 0;
}
```

```
class aa {
    int id;
public:
    aa(int a = 1):id(a) {
        cout << id << endl;
    }
    void display() {
        cout << id << endl;
    }
};
void m(aa a) {
    a.display();
}
int main() {
    aa s;//1
    s = 6;//6
    m(s);//6
    aa t = 'a';//97
    m(t);//97
    m(7);//7     equals to m(aa a=7);  m(aa a(7));
    s = aa(8);//8
    m(s);//8
    return 0;
}
```

explicit关键字:

敬爱的老师:

老师您好!

课上说explicit修饰构造函数只支持显式类型转换，我想问问这种情况下 A a=A（2）这种方式正确还是错误。我查网上结论好像都不太一样，我的编译器可以通过，但是有人说这是拷贝初始化进行隐性转换，我自己的理解是显式调用了构造函数创建了一个A(2)临时对象，然后通过拷贝构造建立了对象a,我觉得这种方式没有隐式类型转换，因此是正确的，不知道我的理解对不对。还有A* c = new A(2);delete c;在这种情况下是否正确呢?

谢谢老师百忙中回答!

----您的一名学生

2024.4.5

实际编译器的结果:

```
#include <iostream>
using namespace std;
class aa {
    int id;
public:
    explicit aa(int a = 1):id(a) {
        cout << id <<  endl;
    }
};
int main() {
    //aa a = 2;//错误，隐式转换
    aa b(2);//正确，显示调用构造函数
    aa* c = new aa(2);//正确
    delete c;//正确
    aa d = aa(3);//正确
    return 0;
}
```

类型转换规则:

转换不能太复杂，不允许多参数，不允许间接转换

```
class aa {
    int id;
public:
    aa(int a = 1) {
        id = a;
    }
    void display() {
        cout << "aa:" << id << endl;
    }
};
class bb {
    int id;
public:
    bb(int a = 2) {
        id = a;
    }
```

```
    void display() {
        cout << "bb:" << id << endl;
    }
};
void m(aa a) {
    a.display();
}
void m(bb a) {
    a.display();
}
int main() {
    m(9);//存在二义性
    //转换不能太复杂，不允许多参数，不允许间接转换
    return 0;
}
```

```
class a {
public:
    int id;
    a(int aa = 1) {
        id = aa;
    }
    void display() {
        cout << "a:" << id << endl;
    }
};
class b {
    int id;
public:
    b(a aa) { id = aa.id; }
    void display() {
        cout << "b:" << id << endl;
    }
};
void m(b aa) {
    aa.display();
}
int main() {
    m(9);//错误
    //转换不能太复杂，不允许多参数，*不允许间接转换
    return 0;
}
```

```
class HW {
    float income;
public:
    HW(float a=2000.0):income(a){}
    float Getincome() {
        return income;
    }
};
class W {
    float income;
```

```
public:
    W(float a=1200.0):income(a){}
    W(HW b) {
        income = 0.75 * b.Getincome();
    }
    float Getincome() {
        return (income);
    }
};
void m(W a) {
    cout << a.Getincome() << endl;
}
int main() {
    W a; m(a);
    HW b; m(b);
}
```

result:

1200

1500

```
class aa {
    int id;
public:
    aa(int a = 1) :id(a) {
        cout << id << "构造" << endl;
    }
    aa(const aa& s):id(s.id) {
        cout << "拷贝" << endl;
    }
    ~aa() {
        cout << "析构" << id<<endl;
    }
};
int main() {
    aa s;//构造
    aa t = s;//拷贝
    aa k = 8;//构造
    s = 9;//*构造 析构   (重点)
    //aa& q = aa(7);//错误，不能指向临时对象      (重点)
    aa m(7);//构造
    aa(6);//构造 析构   匿名临时对象的创建  (重点)
    aa n = aa(5); //?  匿名临时对象的创建+拷贝-->构造(√)    (重点)
    return 0;
}
```

11.友元:

友元的作用：开放私有权限给特定成员：

友元函数：

```
class aa {
    float a;
    float b;
public:
    friend aa sum(aa, aa);
    friend int main();
};
aa sum(aa s, aa t) {
    aa c; c.a = s.a + t.a; c.b = s.b + t.b;
    return c;
}
int main() {
    aa a, b;
    a.a = 1; a.b = 2;
    b.a = 3; b.b = 4;
    aa c = sum(a, b);
    cout << c.a << endl << c.b << endl;//4 6
    for (long i = 0; i < 10; i++) {
        cout << a.a << endl;
    }
    return 0;
}
```

友元类：

```
#include <string>
class Student {
    friend class Teacher;
    int age;
    float score;
};
class Teacher {
    Student sun;
    string name;
public:
    void p(int a) {
        sun.age = a;
    }
    void q(float a) {
        sun.score = a;
    }
};
int main() {
    Teacher a;
    a.p(20);
    a.q(80.9);
}
```

友元的特点：

友元不具有传递性

友元声明位于公有、私有、保护效果一致

扩大了自由函数对类、类对类的访问权限，从而破坏了类的封装性，要慎重使用

12.运算符重载：

C语言本身为函数机制：

```
int operator +(int &a,int &b){
   int c=a;
   for(int i=b;i>0;i--) c++;
   return c;
}
```

……

加法重载：

方式一：类外

```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    friend aa& operator +(const aa&, const aa&);
};
aa& operator +(const aa& a, const aa& b) {
    aa c;
    c.aaa() = a.a + b.a;
    c.bbb() = a.b + b.b;
    return c;
}
int main() {
    aa a, b;
    a.aaa() = 1;
    a.bbb() = 2;
    b.aaa() = 3;
    b.bbb() = 4;
    aa c = a + b;//aa c=operator+(a,b);
    cout << c.aaa() << ' ' << c.bbb() << endl;
}
```

4 6

方式二：类内

```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    aa& operator +(const aa&);
};
aa& aa::operator +(const aa& t) {
    aa c;
    c.a = t.a + a;
    c.b = t.b + b;
    return c;
}
int main() {
    aa a, b;
    a.aaa() = 1;
    a.bbb() = 2;
    b.aaa() = 3;
    b.bbb() = 4;
    aa c = a + b;//aa c=a.operator+(b);
    cout << c.aaa() << ' ' << c.bbb() << endl;
}
```

4 6

重载规则：

不提倡违背本意和改变参数个数：

```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    friend aa& operator+(aa&);
};
aa& operator+(aa& t) {
    aa c;
    c.a = t.a;
    c.b = t.b;
    return c;
}
int main() {
    aa a, b;
    a.aaa() = 1;
    a.bbb() = 2;
    b.aaa() = 3;
```

```
    b.bbb() = 4;
    aa c =+a;//aa c=operator+(a);
    cout << c.aaa() << ' ' << c.bbb() << endl;//1 2
}
```

运算符重载时参数个数不可以超过原来数目

参数顺序很重要（参数和返回类型可以改变，运算顺序和优先级不能改变）：

```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    friend aa& operator+(aa&,int);
};
aa& operator+(aa& t,int a) {
    aa c;
    c.a = t.a+a;
    c.b = t.b+a;
    return c;
}
int main() {
    aa a, b;
    a.aaa() = 1;
    a.bbb() = 2;
    b.aaa() = 3;
    b.bbb() = 4;
    aa c =a+1;//aa c=operator+(a);
    aa d = 1 + a;//错误
    cout << c.aaa() << ' ' << c.bbb() << endl;//1 2
}
```

重载运算符内部的同名运算符维持原来的操作

参数都是基本类型时不能重载：

错误案例：

```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    friend aa& operator+(int,int);
};
aa& operator+(int t,int a) {
    aa c;
    c.a = t+a;
    c.b = t+a;
    return c;
}
int main() {
```

```
    aa c = 1 + 2;
    cout << c.aaa() << ' ' << c.bbb() << endl;
}
```

不允许创建新的运算符

重载运算符函数返回类型不能为空：

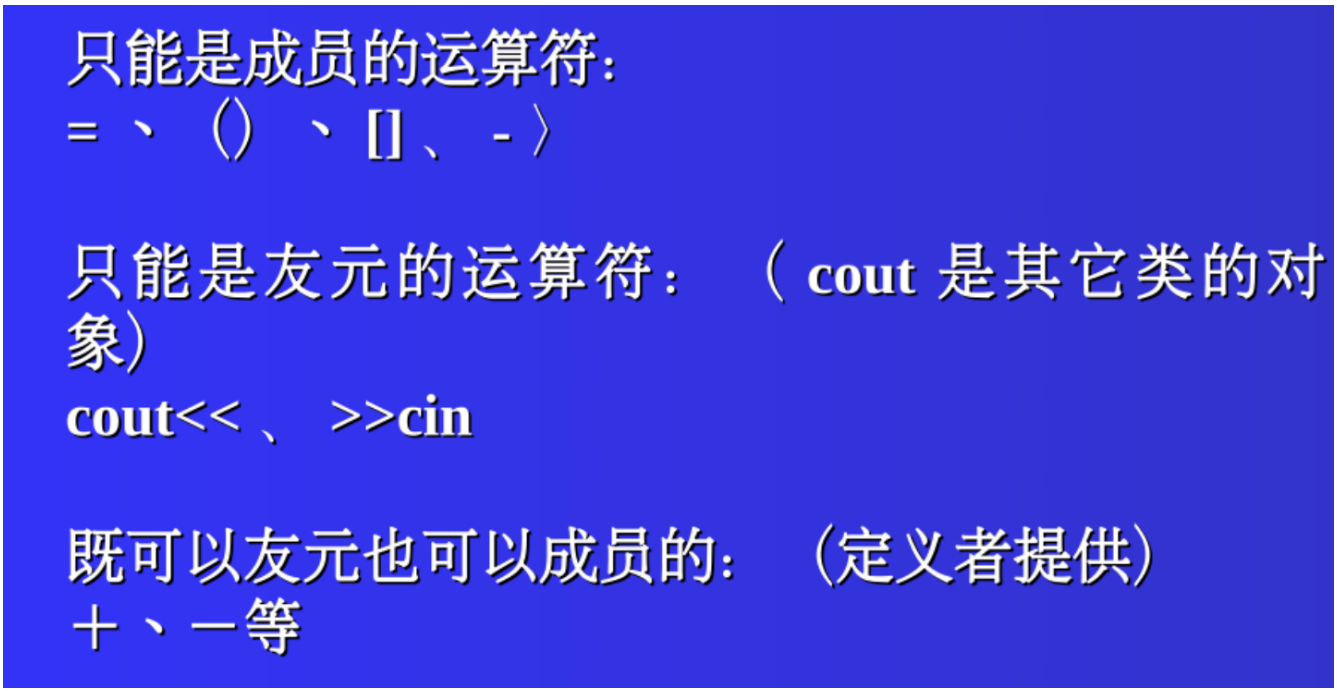
```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    void operator=(aa const&);
};
void aa::operator=(aa const&q) {
    a = q.a;
    b = q.b;
}
int main() {
    aa a, b;
    a.aaa() = 1; a.bbb() = 2;
    b.aaa() = 3; b.bbb() = 4;
    c = a;//c.operator=(a);
    //错误
    cout << c.aaa() << ' ' << c.bbb() << endl;
}
```

重载运算符=函数类型应为引用，值不可以：

```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    aa& operator=(aa const&);
};
aa& aa::operator=(aa const&b) {
    this->a = b.a;
    this->b = b.b;
    return (*this);
}
int main() {
    aa a, b;
    a.aaa() = 1; a.bbb() = 2;
    b.aaa() = 3; b.bbb() = 4;
    aa c;
    c = a;
    c = a = b;
}
```

在C++中，赋值运算符‘operator='通常应该返回一个引用，而不是一个值，这是因为赋值操作通常返回被赋值后的对象，来支持链式赋值，例如c=a=b.否则c=a后返回的是一个异于c的临时变量，何谈让c再为b？

运算符位置规定：



与new,delete混用：

```
class aa {
    float a; float b; int* m;
public:
    aa() :a(1), b(2), m(new int(9)) {}
    float& aaa() { return a; }
    float& bbb() { return b; }
    aa& operator=(aa const&);
    ~aa() {
        delete m;
    }
};
aa& aa::operator=(aa const&q) {
    if (this == &q) return (*this);
    delete m;
    this->a = q.a;
    this->b = q.b;
    this->m = new int(*(q.m));//深拷贝
    return (*this);
}
int main() {
    aa a, b;
    a.aaa() = 1; a.bbb() = 2;
    b.aaa() = 3; b.bbb() = 4;
    aa c;
    c = a;
    c = a = b;
}
```

前++和后++运算符重载：

前++：

```
class aa {
    float a; float b; int* m;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    aa& operator++();
};
aa& aa::operator++() {
    a = a + 1;
    b = b + 1;
    return (*this);
}
int main() {
    aa a;
    a.aaa() = 1; a.bbb() = 2;
    cout << (++a).aaa() << endl;
    cout << a.aaa() << endl;
}
```

2 2

后++：问题：为什么函数返回值没有引用?

原因：m是临时变量，引用的话早被销毁了。

```
class aa {
    float a; float b; int* m;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    aa operator++(int);
};
aa aa::operator++(int) {
    aa m(*this);
    a = a + 1;
    b = b + 1;
    return (m);
}
int main() {
    aa a;
    a.aaa() = 1; a.bbb() = 2;
    cout << a++.aaa() << endl;
    cout << a.aaa() << endl;
}
```

1 2

```
class aa {
    float a; float b;
public:
    float& aaa() { return a; }
    float& bbb() { return b; }
    operator float();//转换运算符
};
aa::operator float() {
    return a;
}
int main() {
    aa a;
    a.aaa() = 1; a.bbb() = 2;
    cout <<float(a) << endl;
    cout << 10+a<< endl;
}
```

1 11

类型转换示例：

正确：

```
#include <iostream>
using namespace std;
class  W;
class HW {
    float income;
public:
    HW(float a = 2000.0) {
        income = a;
    }
    /*operator W() {
        W s; s.Getincome() = 0.75 * income; return s;
    }*/
    float& Getincome() {
        return income;
    }
};
class W {
    float income;
public:
    W(float a = 1200.0) {
        income = a;
    }
    W(HW b) {
        income = 0.75 * b.Getincome();
    }
    float& Getincome() { return income; }
};
void m(W a) { cout << a.Getincome(); }
int main() {
```

```
    W a;
    m(a);
    HW b;
    m(b);
    return 0;
}
```

错误:

```
#include <iostream>
using namespace std;
class W;
class HW {
    float income;
public:
    HW(float a = 2000.0) {
        income = a;
    }
    operator W() {
        W s; s.Getincome() = 0.75 * income; return s;
    }
    float& Getincome() {
        return income;
    }
};
class W {
    float income;
public:
    W(float a = 1200.0) {
        income = a;
    }
    /*W(HW b) {
        income = 0.75 * b.Getincome();
    }*/
    float& Getincome() { return income; }
};
void m(W a) { cout << a.Getincome(); }
int main() {
    W a;
    m(a);
    HW b;
    m(b);
    return 0;
}
```

正确:

```
#include <iostream>
using namespace std;
class W {
    float income;
public:
    W(float a = 1200.0) {
        income = a;
    }
```

```
    /*W(HW b) {
        income = 0.75 * b.Getincome();
    }*/
    float& Getincome() { return income; }
};
class HW {
    float income;
public:
    HW(float a = 2000.0) {
        income = a;
    }
    operator W() {
        W s; s.Getincome() = 0.75 * income; return s;
    }
    float& Getincome() {
        return income;
    }
};
void m(W a) { cout << a.Getincome(); }
int main() {
    W a;
    m(a);
    HW b;
    m(b);
    return 0;
}
```

13.静态成员:

静态成员在main之前构造，生命周期等同于全局对象。

```
class A {
    int s;
public:
    A() :s(9) { cout << "A is Constructing!"<<endl; }
    int R_s() { return s; }
};
class Student {
    int x;
public:
    static A noOfStudents;
};
A Student::noOfStudents;//不允许省略
int main() {
    cout << "Main Fuction is running!" << endl;
    cout << Student::noOfStudents.R_s() << endl;
}
```

A is Constructing! Main Fuction is running! 9

在您提供的代码中，`A Student::noOfStudents;` 是对静态成员变量 `noOfStudents` 的定义和初始化语句。这种定义和初始化静态成员变量的语法形式在 C++ 中是必须的，而不能省略。

静态成员变量的定义和初始化：

静态成员变量在 C++ 中是类的成员，它与类的任何对象实例无关，而是与整个类相关联。静态成员变量的定义和初始化通常需要在类的外部进行，且只能在一个源文件中进行一次定义和初始化。

对于静态成员变量，需要遵循以下规则：

1. **定义静态成员变量** 在类的外部定义静态成员变量，使用类名加作用域解析运算符 :: 来指明所属的类。
2. **初始化静态成员变量** 静态成员变量需要在定义时进行初始化，可以在定义处进行初始化，也可以在类的实现文件中进行初始化。

静态成员的空间不包含在对象内：

```
class Student {
    int x;
public:
    static Student noOfStudents;
};
Student Student::noOfStudents;
int main() {
    Student ss;
    cout << sizeof(ss) << endl;
    return 0;
}
```

4

使用场景：

多个场合修改同一个数据

争夺标记，类似令牌

链首指针：

```
#include <string>
class Student {
public:
    Student(string pName);
    ~Student();
    static Student* pFirst;
    Student* pNext;
    string name;
};
Student* Student::pFirst = 0;
Student::Student(string pName) {
    name = pName;
```

```
    pNext = pFirst;
    pFirst = this;
}
Student::~Student() {
    if (pFirst == this) {
        pFirst = pNext;
        return;
    }
    for (Student* pS = pFirst; pS; pS = pS->pNext) {
        if (pS->pNext == this) {
            pS->pNext = pNext;
            return;
        }
    }
}
void fn() {
    Student s5("S5");
    print();
}
int main() {
    Student s1("S1");
    Student* s2 = new Student("S2");
    Student s3("S3");
    fn();
    delete s2;
}
```

静态成员函数：

```
class Student {
    static int noOfStudents;
    int x;
public:
    int Get() {
        return noOfStudents;
    }
};
int Student::noOfStudents = 0;
int main() {
    //cout << Student::noOfStudents << endl;//错误
    Student ss;
    cout << ss.Get();
    //cout << Student::Get();//希望

    return 0;
}
```

```
class Student {
    static int noOfStudents;
    int x;
public:
```

```
    static int Get() {
        return noOfStudents;
    }
};
int Student::noOfStudents = 0;
int main() {
    cout << Student::Get();
    Student ss;
    cout << ss.Get();//不再必要
    return 0;
}
```

静态成员函数没有this指针

```cpp
class aa {
    int b;
    static int a;
public:
    int GetB() {
        return b;
    }
    void o() { cout << a; cout << this->a; cout << b; }
    static void p() {
        cout << a; cout << this->a;
    }
    static void q() {
        cout << b << GetB();
    }
    static void r(aa& t) {
        t.b = 6; cout << t.GetB();
    }
};
int aa:: a = 0;
int main() {
    aa a;
    aa::r(a);
    return 0;
}
```

调用→	普通数据成员	静态数据成员	普通成员函数	静态成员函数
普通成员函数	√	√	√	√
静态成员函数	X ?	√	X ?	√

只能有一个对象的类：

```
class aa {
    static aa* aH;
    aa() {};
public:
    static aa* get() {
        if (aH == 0) {
            aH = new aa;
        }
        return aH;
    }
};
aa* aa::aH = 0;
int main() {
    aa* s = aa::get();
    aa* t = aa::get();
    aa u;//错误
    return 0;
}
```

实现单向关联：

双向：

```
class Female;
class Male{
    Female*  pWife;
public:
    Female* &getWife( )    {return pWife;}
        //其它方法及实现，略。
};
```

```
class Female {
  Male*   pHusband;
public:
      Male* &getHusband( ) { return pHusband; }
        //其它方法及实现，略。
};

/*类的调用*/
Femal a,c; Male b,d;
a.getHusband()=&b;
b.getWife()=&a;
```

单向：

```
class Female;
class Male{
     Female*  pWife;
public:
      Male( ) {m_buffer[m_sum++] = this;}
      Female* &getWife( ) {return pWife;}
      static  Male* m_buffer[100];  //存放当前产生的所有Male类对象。
      static  int m_sum;
        //其它方法及实现，略。
};
int Male::m_sum = 0;
Male * Male::m_buffer[100];
class Female{
     //无，去掉Male*   pHusband;
public:
    Male * getHusband();
// 其它定义不变
};
Male* Female::getHusband( ) {
  for (int i = 0; i<Male::m_sum;i++)
     if (Male::m_buffer[i]-> getWife( ) == this)
       return Male::m_buffer[i];
  return 0;
}
/*类的调用*/
 Male a,b;  Femal c,d,;
 a.getWife()=&c;
 b.getWife()=&d;
 d.getHusband()=?
```