

对C++还有爱吗？理应如此！

知识点

1. buzzword

1. responsibility-driven 责任负责制，一个函数只干自己负责的那件事
2. iterator 迭代器
3. override 覆盖重写（有些人错写为overwrite，OOP里没有这个术语），在派生类重写基类的虚函数
4. overload 重载（不是过载），允许存在同名函数（参数不同）
5. overhead 额外开销，如调用函数的开销
6. couple 形容代码之间的耦合程度，越松越好
7. cohesion 形容代码内部的凝聚程度，越紧越好
8. template 模板
9. 语法糖：指计算机语言中添加的某种语法，这种语法对语言的功能没有影响，但是更方便程序员使用
如C++是C的语法糖

2. OOP三大特性

- 继承 inheritance
- 封装 encapsulation
- 多态 polymorphic

3. new&delete

对象被 `new` 的时候会调用**构造函数**，被 `delete` 的时候会调用**析构函数**

`new` 之后一定要 `delete`，c++是最后一个没有垃圾回收的语言

```
A *p1 = new stu[10];
delete[] p1; // 回收内存并对所有对象执行一遍析构函数
delete p1; // 是ub，一般也可以正确回收内存，但只有p1[0]执行了析构函数，后面的没管，可能会出错
```

4. class

1. 成员类型

- `public` (公有成员): 类的内部和外部都可以访问
- `private` (私有成员): 类的内部可以访问 (边界是类而不是对象), 外部不可以访问, 因此只能通过成员函数或友元函数进行访问
- `protected` (保护成员): 与私有成员类似, 但在派生类中可以访问

2. 构造函数

- 与类同名, 没有返回类型
- 在创建类时自动调用, **不可显式调用**
- 可以重载, 有参无参均可
- 类一般都要有一个**默认构造函数** (没有参数的构造函数), 除非这个类不应该被默认构造
- 存在缺省的默认构造函数, 自己写了任意一个构造函数后失效

3. 拷贝构造函数

- 与类同名, 没有返回类型
- 写法: `A(const A& other)`
- **浅拷贝vs深拷贝**: 拷贝时, 如果内部指针指向同一个地址, 为浅拷贝; 如果开辟了新的地址, 为深拷贝

存在缺省的拷贝构造函数, 为浅拷贝, 一般够用

但如果类内存在指针需要动态分配内存, 就需要自己写**深拷贝**的拷贝构造函数
- 使用时机: (1)用类的一个对象去**初始化**另一个对象时 (注意跟赋值区分)
(2)当函数的参数或返回值是类的对象时 (值传递时, 引用的时候不会调用)

这里也可以看出, 声明拷贝构造函数时参数必须为**引用**否则拷贝构造函数传参时需要调用自己 (而它自己还没被定义), 就出问题了
- 如果拷贝构造函数是 `private` 的, 那么这个类将**不可拷贝**, 但更好的写法为加上 `= delete` 把函数删除掉

4. 析构函数

- 与类同名, 前面加上~
- 没有返回值
- 只存在一个, 无参
- 一般要写成**虚函数**
- 先构造的后析构 (可理解为栈)
- 存在缺省的析构函数, 一般够用, 为了支持虚函数可以写成 `virtual ~classname() = default`

5. 大三律/大五律(Rule of Five)

如果声明了**拷贝构造函数**、**拷贝赋值函数**、**析构函数**三个其中的一个, 就需要把三个全部声明

因为一般情况下，这些函数并不需要自己写，编译器实现的函数已经够用，如果自己写了（八成是因为内存要动态分配），就意味着编译器实现的有问题

Rule of five就是加上了移动构造函数、移动赋值函数

6. • this

- 不能显式声明，因为 `this` 是关键字
- 在使用 `class` 的成员函数时，自动传递了一个 `this` 指针
- 写代码时可以先打 `this.`，IDE会自动选择成员（不用记成员名字）

```
class Point {  
private:  
    int a, b;  
public:  
    void print() {cout << a << ' ' << b << endl;}  
};
```

此时 `print()` 函数相当于

```
void print(Point *this) { // 自动会有一个this指针参数传递  
    cout << this->a << ' ' << this->b << endl;  
}  
// 使用x.print()时，相当于是Point::print(&x)
```

7. class和object的区别

- 类和对象的关系是一种数据类型与变量的关系
- 成员函数被存放在代码区，因此对象里只有成员变量而没有成员函数

5. 初始化列表

- 只能在构造函数中使用
- 用于初始化非静态成员变量或基类
- 初始化列表中的初始化顺序与变量的声明顺序一致，与初始化列表中书写的顺序无关
- 发生在函数body执行之前
- 本质上是函数的body，应写在定义中而非声明

```

class A {
private:
    int x;
public:
    A(int t) { x = t; }
};
class B {
private:
    string s;
public:
    B(string t) { s = t; }
};
class C : public A { // C继承了A
private:
    double f;
    B b; // C的成员里有其他类的对象
public:
    C(int x, string s, double t) : A(x), b(s) { f = t; }
    //: A(x), b(s)这一部分就是初始化列表
    // 需要调用基类的构造函数、成员变量的构造函数
};

```

6. 静态成员

声明静态成员时需要使用 `static` 关键字进行修饰，如

```

class MyClass {
public:
    static int count; // 声明一个静态成员变量，之后还需要定义和初始化，否则无法使用
    void incrementCount() { count++; } // 非静态成员函数
    static int Count() { return count; } // 静态成员函数
} t;
// 建议使用MyClass::count而不是t::count去访问
static const int count = 0; // 不要忘记

```

- **属于整个类**，只会有一份拷贝，被所有该类的对象所共享
- 最好**使用类名来访问**，所有对象访问（不推荐）到的静态成员都是一样的
- 不要用构造函数去初始化，否则每创建一个对象就会把静态成员赋值一遍
- 若静态变量为 `const` 类型，则可以在类内直接初始化，如 `static const int count = 0;`，但之后不能再改变
- 若不是 `const` 类型，则需要在类外进行静态成员的定义和初始化（此时需要指定静态成员变量的类型），如 `int MyClass::count = 0;`
- 静态变量的**生命周期**和全局变量相同，为整个程序

- 静态成员可以被继承，但静态成员函数**不能实现多态**（因为不能是虚函数）

7. friend 友元

借助声明友元，可以使得类外的函数或其他类，可以访问某个类的私有成员和保护成员。

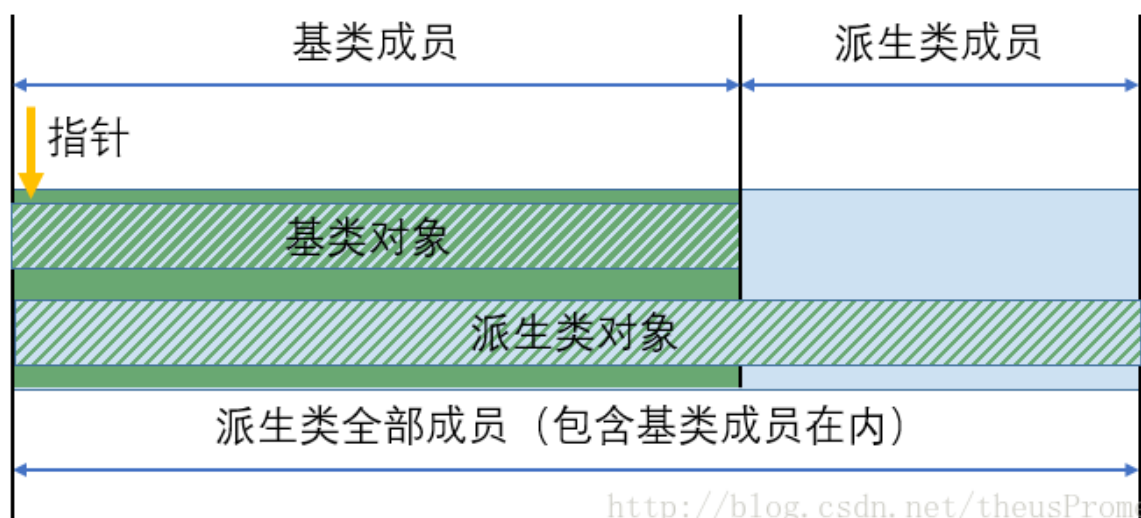
- 友元**不属于该类的成员**，内部不能使用 `this` 指针来表示该类的对象
- 在类中的声明位置没有要求
- 友元是**单向的**，没有对称性和传递性
 - B是A的友元，不代表A是B的友元
 - B是A的友元，C是B的友元，不代表C是A的友元
- 友元函数**不能被继承**

友元方便了对类内私有和保护成员的访问，同时也破坏了类的封装性；但在宏观上看，维护了整个程序的封装性

8. 派生类

1. 基本概念

- 派生类拥有基类所有的成员，还可以定义自己的成员
- 友元不是类的成员，因此不能被继承



```
class Derived : public Base1, private Base2{  
    // 派生类的成员列表  
}
```

2. 继承方式 在继承时，`public/protected/private` 为访问说明符，表明继承方式

- 在派生类**内部**（成员函数或友元函数）使用基类成员时，不受继承方式的影响，只看该成员在基类中的属性（因此私有成员不能被使用）
- 但在派生类**外部**使用基类成员时，继承方式会影响基类成员的访问属性

- `public` : 所有基类成员在派生类中保持原有的访问级别
- `protected` : `public` -> `protected` , 其余不变
- `private` : 所有基类成员在派生类中变为 `private` 成员

只有 `public` 继承是符合OOP的, derived class is-a base class

其余两个应该直接使用组合 (直接在类内定义别的类的对象) 来实现

3. • 构造函数

调用顺序为: 先调用所有**基类**的构造函数, 再调用派生类中**成员**的构造函数(如果成员里有别的类的对象), 最后调用**派生类**自己的构造函数

前两部分一般发生在初始化列表中, 最后一部分发生在构造函数的body里

处于同一层次各基类构造函数的调用顺序, 取决于派生类中的**声明顺序**而不是实际的调用顺序

```
class A {
private:
    int x;
public:
    A(int t) { x = t; }
};
class B {
private:
    string s;
public:
    B(string t) { s = t; }
};
class C : public A {
private:
    double f;
    B b;
public:
    C(int x, string s, double t) : A(x), b(s) { f = t; }
    // 基类构造函数->成员构造函数->成员
};
```

4. • 多继承

YBB, 坑超多, 慎用

原本是用于解决容器问题的, 但现在这个问题被 `template` 解决了, 所以不要再使用多继承

9. 虚函数

1. **目的**: 用基类指针指向派生类的时候, 能正确调用派生类中的实现函数, 从而实现多态

2. **定义** 在成员函数**声明**时，在前面加 `virtual`，即可使其变为虚函数（定义时可以不加）

- 拥有虚函数的类会自动生成一个**虚函数表** `vtbl`（属于类而不是对象），是一个指针数组，里面的元素是虚函数的**函数指针**
- 创建对象时，对象内部会自动生成一个**虚表指针** `*vptr`（通常会在对象内存的最起始位置），指向类的虚表 `vtbl`
- 在调用虚函数时，会**经由** `vptr` **找到** `vtbl`，再通过 `vtbl` 中的函数指针找到对应虚函数的代码并进行调用
- ps1: 普通函数、虚函数、虚函数表都是同一个类的所有对象公有的，只有成员变量和虚表指针是每个对象私有的
- ps2: 自然，非虚函数可以直接调用，不用经过虚表，符合零成本抽象原则

3. • **重写** `override`

在继承时，派生类会继承基类的虚表，虚函数继承之后仍是虚函数（不用加 `virtual`）

同时也可以**重写**（`override`）虚函数（此时在派生类的虚表中，派生类的函数地址会覆盖基类的函数地址）

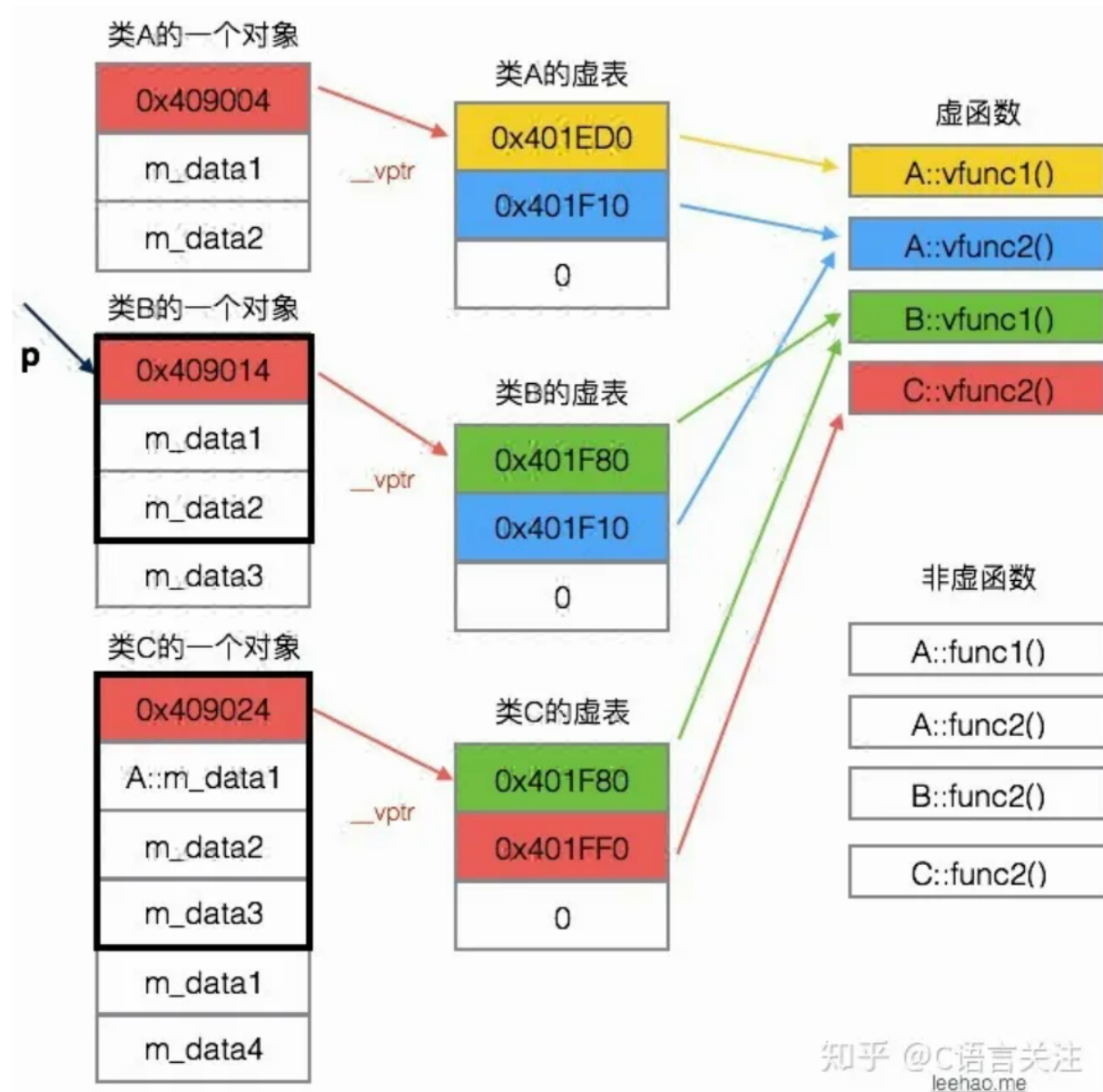
```
class A {
public:
    virtual void vfunc1();
    virtual void vfunc2();
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};

class B : public A {
public:
    void vfunc1() override; // 最好用override表达这个函数是继承而来的
    void func1();
private:
    int m_data3;
};

class C: public B {
public:
    void vfunc2() override;
    void func2();
private:
    int m_data1, m_data4;
};

B bobject; // 类B的一个对象
A* p = &bobject; // 通过基类指针*p指向派生类B的对象
```

这些类的虚表如图所示



可以发现，通过基类指针 `p`，可以访问基类中的所有非虚成员，无法访问派生类独有的非虚成员（其内存位置在基类成员之后），却可以通过虚表来访问派生类重写的虚函数

- ps1: 若想访问派生类独有的非虚成员，可以通过强制类型转换，将 `p` 转换为派生类的指针，如 `dynamic_cast<B*>(p)->func2()`
- ps2: 不要重写(overwrite)继承而来的非虚函数（区别于override），因为会破坏多态性和is-a原则，并且会发生name hiding，隐藏掉基类中的同名函数

4. 静态绑定vs动态绑定

- 静态绑定 (static/early binding)，在编译阶段决定函数是哪个类的函数（此时对象还未创建）适用于普通成员函数，根据指针自身的类型来决定
- 动态绑定 (dynamic/late binding)，在运行阶段决定函数是哪个类的函数 适用于虚函数，根据指针指向的对象的实际类型来决定（必须在指针指向的对象创建出来后才能决定，因此只能在运行阶段判断）

(对于很多其它语言，默认为动态绑定)


```

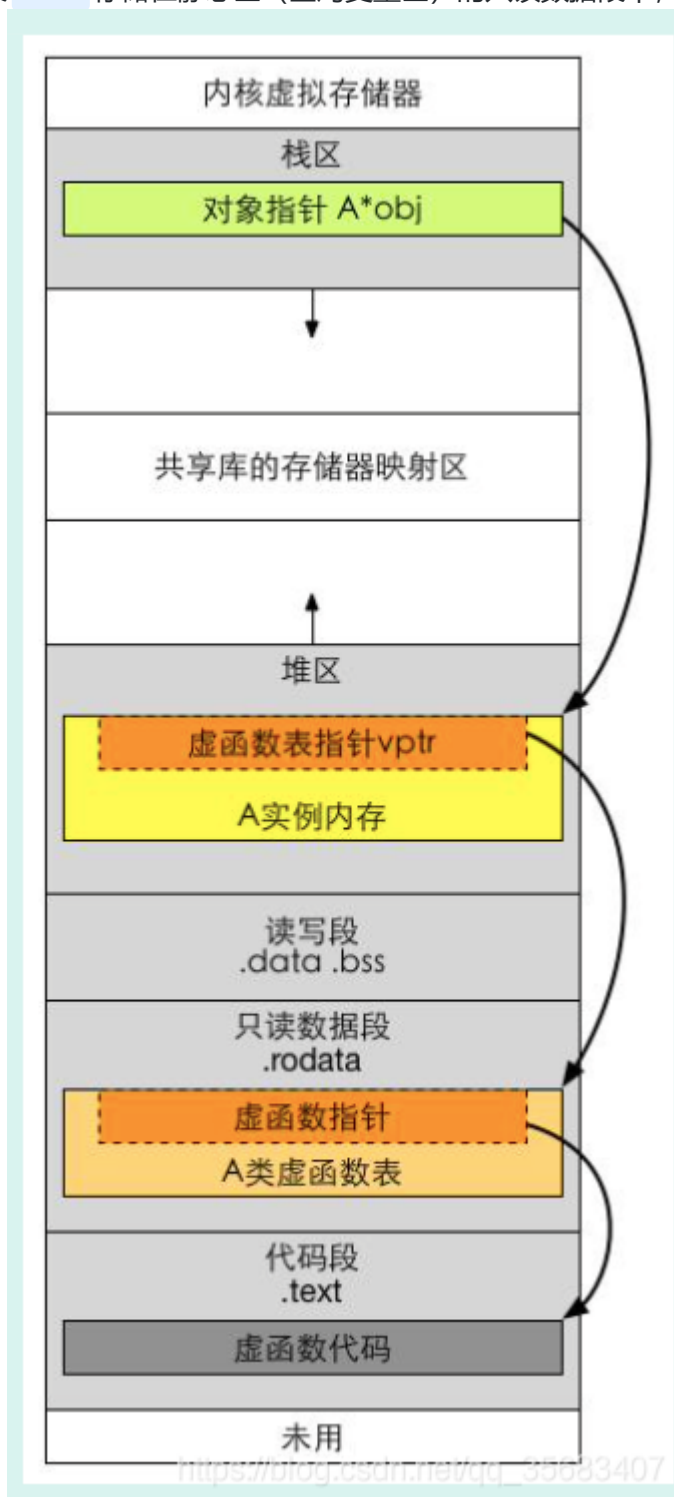
class animal {
public:
    void print() { cout << "I'm an animal." << endl; }
    virtual void vprint() { cout << "I'm an animal." << endl; }
    virtual ~animal() = default;
};

class dog : public animal {
public:
    void print() { cout << "I'm a dog." << endl; }           // 这是overwrite, 不好
    void vprint() override{ cout << "I'm a dog." << endl; } // override
    virtual ~dog() = default;
};

int main() {
    dog D;
    animal* p = &D; // 基类指针指向派生类对象
    p->print(); // early binding, 指针p为animal*类型, 直接采用基类animal中的print(),
    // 跟派生类dog无关, 无法做到多态
    // 被翻译为animal::print(p);
    p->vprint(); // late binding, 指针p指向的对象为dog类型, 故采用派生类dog中的
    vprint()而不是基类animal中的vprint()
    // 被翻译为( *(p->vptr)[0] )(p), 即( p->vtbl[0] )(p)
    // 原理: 先通过指针p找到对象D, 再通过对象D中的虚指针找到类dog的虚表, 再到虚表里找到
    vprint()的函数指针
    return 0;
}

```

5. 存储位置 虚表 `vtbl` 存储在静态区（全局变量区）的只读数据段中，虚表指针 `*vptr` 存储在对象的



内存中，如图

6. 注意点：

- 静态成员函数不能定义为虚函数，因为static成员函数不属于任何对象
- **构造函数不能为虚函数**（虚函数一般需要在对象创建后才能调用，而构造函数正是用于创建对象的）
- **析构函数最好写成虚函数**（不然 `delete` 基类指针时，无法调用派生类的析构函数），除非它没有子类
- 不要在构造函数和析构函数里面调用虚函数，如果调用了则是静态联编 因为在构造函数和析构函数中，对象是不完整的，因此无法确定该调用哪个子类的虚函数，会发生UB
- 虚函数也可以是内联函数，但是当虚函数表现多态性的时候内联失效（因为要通过函数指针调用）

- （不建议这么做）如果在派生类中overload基类的虚函数，会发生name hiding，破坏了多态，并且派生类中定义的新版本只能通过派生类对象调用，而不能通过基类指针或引用调用

7. final

- 用于修饰一个虚函数，使其不能被派生类override
- 一般用在继承关系的中途，终止派生类的override

8. override

- 用于修饰一个函数，检查其是否override了基类的虚函数
- 应广泛使用，防止函数名写错/函数参数类型不匹配/重载(overload)非虚函数

10. 纯虚函数

1. **目的**：定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的派生类必须实现这个函数

使用纯虚函数，一般发生在基类无法提供合理的缺省实现的时候（否则用虚函数就行了）

2. **声明**：在虚函数声明时，在后面加 `=0`，即可使其变为纯虚函数

3. 注意点：

- 纯虚函数**没有body，只有声明**，因此纯虚函数**不能被直接调用**
- 纯虚函数必须被override，若派生类没有给出实现，则其仍然为纯虚函数

4. 抽象类

- 只要类中有一个纯虚函数，那么就是抽象类，**不能创建该类的对象**，但可以创建指针
- 只能当作基类，是一个概念化的东西

5. 协议类

类里没有非静态成员变量，而且所有成员函数都是纯虚函数，则称为协议类，可以安全地实现多继承

11. inline

- 以前的作用：告诉编译器这个函数可以被**内联**（相当于在调用处展开代码，省去调用函数的开销），但编译器不一定听你的，因此这个功能可以忽略
- 现在的作用：被 `inline` 修饰的函数或变量在**多个编译单元**（多个cpp文件）出现的时候不会违反单次定义原则(ODR)

多的不说了，只需要记住 `inline` 修饰的函数要写在头文件里，是**声明**而不是定义

12. namespace命名空间

- 用于区分位于不同地方同名的东西
- 后面也可以随意往一个namespace里加东西

- `using namespace std;` 使用该命名空间，之后就不需要写 `std::`，但更好的方法是使用 `using std::cin;`

13. `const`

- `const` 表示常量 (constant)，“不变的变量”，但实际不一定在静态存储区里，要看具体情况

```
const int x = 123; // x为编译时常量 (Compile-time constant)，123为字面量 (literal)
// 在这种简单的情况下，编译器会直接把x优化为汇编里的立即数，存储在静态存储区
cin >> size;
const int SIZE = size; // SIZE为运行时常量 (Runtime constant)，只有运行的时候才能知道常量的值
// 因此编译器只能把它设置成变量，然后保证它不会被修改，这样就会存储在栈或堆里，跟普通变量一样
// 这里的const纯粹是编译器帮你检查的工具，对二进制来说完全是透明的
// 所以实际上更好的叫法是只读readonly
```

- `const` 默认作用于其左边的东西，否则作用于其右边的东西
- 传参时加 `const` 可以保证函数不修改这个变量，只要能加就都加上

• `const` 修饰成员函数

被 `const` 修饰的对象，只能使用被 `const` 修饰的成员函数

```
class A {
private:
    int x, y;
public:
    A(int a, int b) { x = a, y = b; }
    void set_x(int a) { x = a; } // true
    void set_y(int a) const { y = a; } // false
    // 此时相当于set_y(const A* this, int a)，也就是不能通过this来修改这个对象
    int get_x() const { return x; } // true
    int get_y() { return y; } // true
};

const A a(1,2);
// 定义了一个const的对象，不能修改
a.get_x(); // true
a.get_y(); // false, 因为该函数没有保证不修改对象（没有加const修饰），所以不让调用
```

• `const` 指针

```

int a = 3;
const int b = 4;
int* p0 = &b; // false, 因为有可能通过(*p0)修改const的b

// 用法1: 定义指向const的指针 (不能通过指针去修改指向的东西)
const int* p1 = &b; // true
int const* p1 = &a; // true
// 用法2: 定义const指针(指针本身的值不能改变), 必须要初始化
int* const p2 = &a;
// 用法3: 指针本身和它指向的内容都不能被改变
const int* const p3 = &a;
int const* const p3 = &a; // 两种写法等价
// const默认作用于其左边的东西, 否则作用于其右边的东西

```

• • `const_cast<T*>`

对于一个被 `const` 修饰的变量, 实际上还是可以修改的, 即:

```

const string a = "12";
string& b = *const_cast<string*>(&a); // 强行去掉const修饰
b = "23";

```

然而, 对于基本类型, 这样的操作会直接被编译器优化掉, 从而保证 `a` 不被修改, 只有程序足够复杂 (如序列化) 时才能骗过编译器

要谨慎使用 `const_cast<T*>`, 因为它破坏了常量性约束, 在实践中往往是有害的

9. 引用Reference

- 是已经存在的变量的**别名**, 在定义时**必须初始化**
- **不能改变**绑定的变量, 在底层上是个 `T *const` 指针
- 引用和被引用的对象类型必须**完全一致**
- 不过引用的时候可以自行添加 `const`, 以避免通过这个引用修改原来变量的值

```
int a = 1;
int& b = a;      // T
const int& c = a; // T
int& d = c;      // F, c是const int类型
const int& e = c; // T
```

- 没有引用数组（数组的每个元素都是别名，没有存在的必要）
- **重要**：作为函数参数时，只会传递一个地址，而不会传递变量本身，且可以在函数里直接改变该变量

```
void swap(int& x, int& y) { // 可以直接在swap中改变变量的值
    int t = x;
    x = y;
    y = t;
}
int a = 1, b = 2;
swap(a, b); // 此时a=2,b=1
```

10. 默认函数参数值

在写函数参数的时候，可以给其赋默认值

```
void f(int x, int y = 1) { // 此时y的默认值为1
    cout << x << ' ' << y << endl;
}
```

这时候实际上产生了两个重载函数，`void f(int x, int y) {...}` 和 `void f(int x) {int y = 1; ...}`

- 默认值只能写在函数**声明**里，而不能写在定义里（如果二者分开的话）
- 默认参数值不在函数的代码里，而是在调用的时候，编译器自动给你加上
- 重载函数就可以解决这个问题，**慎用默认函数参数值**

11. 重载运算符

- 不能重载不存在的运算符
- 运算符重载后**优先级不变**，单目双目不变（因为编译器只认识原本的运算符）

- 重载后的含义可以跟原本的含义**完全无关**
- 实际上是函数调用的另一种形式，比如 `a.operator+(b)`

• 写法

在类内：可以少写一个参数（因为有 `this` 充当左边的那个），而且可以直接访问私有变量，单目运算符这样写更好 `bool operator <(const T& other) const{} , const T operator +(const T& other) const{}`

在全局：不能直接访问私有变量（可以通过友元解决），源码不可见的时候只能这样写，**二元运算符**这样写更好（左右都可以自动类型转换） `bool operator <(const T& x, const T& y){} , const T operator +(const T& x, const T& y){}`

示例：

```
// 不要重载逻辑运算符和逗号(&&||,)
// + - * /
const T operator+(const T& x, const T& y) // 返回一个右值
// <
bool operator<(const T& x, const T& y) // 返回一个bool
// [],一般T是某个容器，E为其存储的类型
E& T::operator[](int index) {return buf[index];} //返回一个左值
```

赋值运算符重载=：

```
T& operator=(const T& other) { // 自己写拷贝赋值函数的时候，往往需要深拷贝
    if (this != &other) { // 要判断是不是自己
        delete buf[]; // 要记得是释放原来分配过的内存，以免内存泄漏
        // do something
    }
    return *this; // 应该返回自己的引用，以满足连等的需求
}
```

前缀后缀++：

```

const T& operator ++() { // 前缀++
    *this += 1; // 需要重载+=
    return *this; // 返回一个左值
}
const T operator ++(int) { // 后缀++, 这个int用于区分, 没有实际意义
    T old(*this); // 要先拷贝一个原本的对象
    ++(*this); // 需要调用前缀++
    return old; // 返回一个右值
}

```

输入输出流重载:

```

istream& operator <<(istream& is, T& obj) {
    // 把一些东西写入obj
    return is;
}
ostream& operator <<(ostream& os, const T& obj) {
    // 输出obj的一些东西
    return os;
}

```

• 类型转换符函数

- 是一种特殊的成员函数，用于将一个类的对象转换为另一个类型
- 是隐式类型转换
- 以关键字 `operator` 开头，后面跟着要转换的目标类型，**没有返回类型**（其实要转换的类型就是返回类型）
- 类型安全很重要，所以要谨慎使用，最好用显式的转换 `double toDouble() const;`

```

class A {
public:
    A(int x) {value = x;}
    operator int() const {return value;}
private:
    int value;
};
A a(1);
int x = a; // 会隐式调用operator int()a

```


• 初始化vs赋值

`operator=` 是赋值运算符，而不是初始化 使用 `T b = a` 或 `T b = {a}` 时（其实和 `T b(a)` 等价），这里的 `=` 并不是赋值运算符，而是**初始化操作符**，因此会调用**拷贝构造函数**，而不是拷贝赋值函数 使用 `T b; b = a;` 时，就是赋值操作，使用拷贝赋值函数

12. template

函数模板 `template<typename T>` 或者类模板 `template<class T>`，还可以加常数（**无类型参数**）
`template<class T, int size = 100>`

- 根据实际数据类型，编译器把**类模板**实例化为**模板类**
- 实例化的时机在**运行时**（函数调用时）
- 需要**精准匹配**，不会有自动类型转换
- 优先级比常规函数低（即如果原本就有匹配的函数，就不会用模板）
- 是**声明**而不是定义（跟 `inline` 类似），要写在头文件里，运行时会自动实例化

```
template <typename T>
class A {
public:
    T a;
    void print();
    A(T x) : a(x) {}
};
template <typename T>
void A<T>::print() { cout << a << endl; } // 注意这里的template <typename T>和A<T>
A<int> a(2);
a.print();
```

12. 异常

代码只需要分两种，不会发生异常和**可能发生异常**

- 好处：不用写一堆 `if-else`，业务逻辑和异常处理是分离的，逻辑更清晰
- 坏处：异常会把栈开解，会造成效率的下降；实际中做到异常安全非常难

• throw

- `throw` 用于抛出异常（可以抛出任意东西，一般是一个异常类），然后离开这个大括号（有点像 `return`），直到被匹配的 `catch` 捕捉到
- `throw` 之后的语句都不会执行
- 在栈中的本地变量都会被正确析构，但 `throw` 出来的东西直到 `catch` 之后才被析构

- try

- try 后面可以跟任意数量的 catch，按顺序来匹配（只会匹配一次）

- catch

- 匹配时不会有隐式类型转换，但基类可以捕获派生类
- catch 里要写一个变量（类似于函数或 lamada 表达式，应采用引用传递），用于捕获 throw 出来的东西，
- catch(...) 意味着可以捕获任意类型
- 在 catch 中还可以再次重抛异常 throw;（后面不用跟任何东西），用于告诉上级代码这里出问题了

- 异常声明

- 函数后面可以跟异常声明 :throw(int, double)，告诉用户可能会抛出什么异常类型
- 如果在函数中抛出了其它的异常，会再抛出一个特殊的异常，直接终止程序
- :throw() 或 noexcept 表示函数不会抛出任何异常，编译器不用产生栈开解相关的代码，从而进行优化
- 事实上，c++11后应当使用 noexcept 代替 throw 的异常声明

- 标准 exception 类

```
class exception{
public:
    exception () throw(); //构造函数
    exception (const exception&) throw(); //拷贝构造函数
    exception& operator= (const exception&) throw(); //运算符重载
    virtual ~exception() throw(); //虚析构函数
    virtual const char* what() const throw();
    //虚函数，用于描述错误的具体情况
    //继承的时候要override这个what()函数
}
```

13. 输入输出流stream


- 以前：输入输出被统一成了文件，那么只需要 open/close, read/write, seek/tell
- 现在：不再用 seek/tell（以前需要数据文件，但现在有了数据库），改用流直接从头读到尾，符合现在文件的需求

流的特点：一维、单向

- 与 scanf、printf 相比

- + 更安全, 更方便
- + 可拓展 (重载`<<`extractors、`>>`inserters)
- + 更面向对象
- + 更啰嗦、更慢
- + 混用有问题 (如果关闭了流同步)

• cin&cout

- + `cin` 是 C++ 标准输入流对象, 即 `**istream` 类的对象**`
- + `cout` 是标准输出流对象, 即 `ostream` 类的对象`
- + `cerr` 也是 `ostream` 类的对象, 用于向标准错误流输出错误和诊断信息, 跟标准输出流有所不同`
- + `istream` 家族的继承关系如图所示 !` (./图/istream.png)

• manipulators

用于操纵格式, 也可以自己定义

manipulator	effect	type
dec, hex, oct	set numeric conversion	I, O
endl	insert newline and flush	O
flush	flush stream	O
setw(int)	set field width	I, O
setfill(ch)	change fill character	I, O
setbase(int)	set number base	O
ws	skip whitespace	I
setprecision(int)	set floating point precision	O
setiosflags(long)	turn on specified flags	I, O
resetiosflags(long)	turn off specified flags	I, O

14. 程序填空常考

- 赋值运算符重载: 要判断是不是this, 要释放原来分配的内存
- 类型转换运算符: `operator int() const {}`
- 模板: 在外部定义时记得写 `template <typename T> void A<T>::print()`, 类模板->模板类

- 异常：try-catch，怎么捕捉？捕捉结果？
- 静态成员变量要**在外部定义**
- 要写在声明里的：`inline`，`template`，函数默认参数值，`virtual`，纯虚函数

2. Alan Kay

最后以面向对象之父总结的OOP特性作为结尾：

1. 一切都是对象
2. 程序是一堆对象互相请求其它对象做什么（而不是怎么做，怎么做由对象自己决定）
3. 每个对象有自己的内存，由其它小对象组成（最终分到最基础的类型）
4. 所有对象都有类型
5. 属于同一个特定类型的对象都可以做相同的动作（可以做相同动作的对象都属于同一个类型）