目录

编译内存相关 6

C++ 程序编译过程 6

C++ 内存管理 7

栈和堆的区别 8

变量的区别 9

全局变量定义在头文件中有什么问题？ 9

对象创建限制在堆或栈 10

内存对齐 10

类的大小 11

什么是内存泄露 12

怎么防止内存泄漏？内存泄漏检测工具的原理？ 12

智能指针有哪几种？智能指针的实现原理？ 15

一个 unique*ptr 怎么赋值给另一个 unique*ptr 对象？ 18

使用智能指针会出现什么问题？怎么解决？ 18

语言对比 20

C++ 11 新特性 20

C 和 C++ 的区别 23

Java 和 C++ 的区别 23

Python 和 C++ 的区别 24

面向对象 25

什么是面向对象？面向对象的三大特性 25

重载、重写、隐藏的区别 25

如何理解 C++ 是面向对象编程 27

什么是多态？多态如何实现？ 28
关键字库函数 30
sizeof 和 strlen 的区别 30
lambda 表达式（匿名函数）的具体应用和使用场景 31
explicit 的作用（如何避免编译器进行隐式类型转换） 32
C 和 C++ static 的区别 33
static 的作用 33
static 在类中使用的注意事项（定义、初始化和使用） 34
static 全局变量和普通全局变量的异同 36
const 作用及用法 36
define 和 const 的区别 37
define 和 typedef 的区别 38
用宏实现比较大小，以及两个数中的最小值 39
inline 作用及使用方法 39
inline 函数工作原理 40
宏定义（define）和内联函数（inline）的区别 40
new 的作用？ 41
new 和 malloc 如何判断是否申请到内存？ 41
delete 实现原理？delete 和 delete[] 的区别？ 41
new 和 malloc 的区别，delete 和 free 的区别 42
malloc 的原理？malloc 的底层实现？ 42
C 和 C++ struct 的区别？ 42
为什么有了 class 还保留 struct？ 43
struct 和 union 的区别 43
class 和 struct 的异同 44

volatile 的作用？是否具有原子性，对编译器有什么影响？ 45
什么情况下一定要用 volatile， 能否和 const 一起使用？ 46
返回函数中静态变量的地址会发生什么？ 46
extern C 的作用？ 46
sizeof(1==1) 在 C 和 C++ 中分别是什么结果？ 47
memcpy 函数的底层原理？ 47
strcpy 函数有什么缺陷？ 48
auto 类型推导的原理 49
类相关 49
什么是虚函数？什么是纯虚函数？ 49
虚函数和纯虚函数的区别？ 50
虚函数的实现机制 50
单继承和多继承的虚函数表结构 53
如何禁止构造函数的使用？ 61
什么是类的默认构造函数？ 62
构造函数、析构函数是否需要定义成虚函数？为什么？ 63
如何避免拷贝？ 63
如何减少构造函数开销？ 64
多重继承时会出现什么状况？如何解决？ 65
空类占多少字节？C++ 编译器会给一个空类自动生成哪些函数？ 69
为什么拷贝构造函数必须为引用？ 70
C++ 类对象的初始化顺序 72
如何禁止一个类被实例化？ 73
为什么用成员初始化列表会快一些？ 74
实例化一个对象需要哪几个阶段 75

友元函数的作用及使用场景 76
静态绑定和动态绑定是怎么实现的？ 78
深拷贝和浅拷贝的区别 79
编译时多态和运行时多态的区别 81
实现一个类成员函数，要求不允许修改类的成员变量？ 81
如何让类不能被继承？ 82
语言特性相关 83
左值和右值的区别？左值引用和右值引用的区别，如何将左值转换成右值？ 83
std::move() 函数的实现原理 84
什么是指针？指针的大小及用法？ 86
什么是野指针和悬空指针？ 88
C++ 11 nullptr 比 NULL 优势 89
指针和引用的区别？ 90
常量指针和指针常量的区别 90
函数指针和指针函数的区别 92
强制类型转换有哪几种？ 93
如何判断结构体是否相等？能否用 memcmp 函数判断结构体相等？ 96
参数传递时，值传递、引用传递、指针传递的区别？ 97
什么是模板？如何实现？ 98
- 函数模板和类模板的区别？ 99
什么是可变参数模板？ 100
什么是模板特化？为什么特化？ 101
include " " 和 <> 的区别 103
switch 的 case 里为何不能定义变量 103

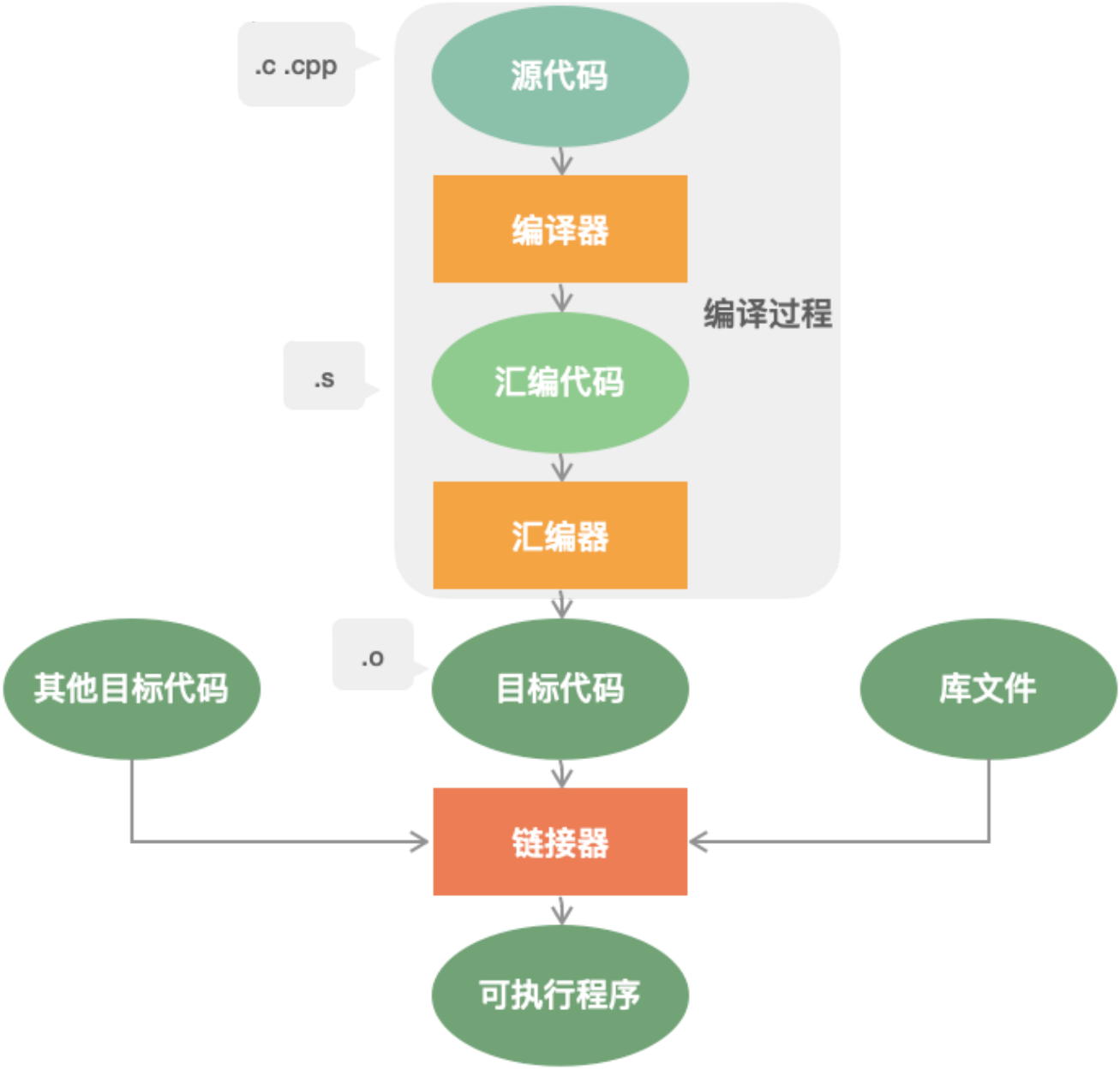
迭代器的作用？ 104
泛型编程如何实现？ 104
什么是类型萃取？ 105
设计模式 105
了解哪些设计模式？ 105
什么是单例模式？如何实现？应用场景？ 106
什么是工厂模式？如何实现？应用场景？ 109
什么是观察者模式？如何实现？应用场景？ 118

编译内存相关

C++ 程序编译过程

编译过程分为四个过程：编译（编译预处理、编译、优化），汇编，链接。

- 编译预处理：处理以 # 开头的指令；
- 编译、优化：将源码 .cpp 文件翻译成 .s 汇编代码；
- 汇编：将汇编代码 .s 翻译成机器指令 .o 文件；
- 链接：汇编程序生成的目标文件，即 .o 文件，并不会立即执行，因为可能会出现：.cpp 文件中的函数引用了另一个 .cpp 文件中定义的符号或者调用了某个库文件中的函数。那链接的目的就是将这些文件对应的目标文件连接成一个整体，从而生成可执行的程序 .exe 文件。



链接分为两种：

- 静态链接：代码从其所在的静态链接库中拷贝到最终的可执行程序中，在该程序被执行时，这些代码会被装入到该进程的虚拟地址空间中。

- 动态链接：代码被放到动态链接库或共享对象的某个目标文件中，链接程序只是在最终的可执行程序中记录了共享对象的名字等一些信息。在程序执行时，动态链接库的全部内容会被映射到运行时相应进行的虚拟地址的空间。

二者的优缺点：

- 静态链接：浪费空间，每个可执行程序都会有目标文件的一个副本，这样如果目标文件进行了更新操作，就需要重新进行编译链接生成可执行程序（更新困难）；优点就是执行的时候运行速度快，因为可执行程序具备了程序运行的所有内容。
- 动态链接：节省内存、更新方便，但是动态链接是在程序运行时，每次执行都需要链接，相比静态链接会有一定的性能损失。

C++ 内存管理

C++ 内存分区：栈、堆、全局/静态存储区、常量存储区、代码区。

- 栈：存放函数的局部变量、函数参数、返回地址等，由编译器自动分配和释放。
- 堆：动态申请的内存空间，就是由 malloc 分配的内存块，由程序员控制它的分配和释放，如果程序执行结束还没有释放，操作系统会自动回收。
- 全局区/静态存储区（.bss 段和 .data 段）：存放全局变量和静态变量，程序运行结束操作系统自动释放，在 C 语言中，未初始化的放在 .bss 段中，初始化的放在 .data 段中，C++ 中不再区分了。
- 常量存储区（.data 段）：存放的是常量，不允许修改，程序运行结束自动释放。
- 代码区（.text 段）：存放代码，不允许修改，但可以执行。编译后的二进制文件存放在这里。

  说明：

从操作系统的本身来讲，以上存储区在内存中的分布是如下形式(从低地址到高地址)：.text 段 --> .data 段 --> .bss 段 --> 堆 --> unused --> 栈 --> env
程序实例：

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

/*
说明：C++ 中不再区分初始化和未初始化的全局变量、静态变量的存储区，如果非要区分下述程序标注在了括号中
*/

int g_var = 0; // g_var 在全局区（.data 段）
char *gp_var;  // gp_var 在全局区（.bss 段）

int main()
{
    int var;                    // var 在栈区
    char *p_var;                // p_var 在栈区
```

```
    char arr[] = "abc";          // arr 为数组变量，存储在栈区；"abc"为字符串常量，存储在常量区
    char *p_var1 = "123456";     // p_var1 在栈区；"123456"为字符串常量，存储在常量区
    static int s_var = 0;        // s_var 为静态变量，存在静态存储区（.data 段）
    p_var = (char *)malloc(10); // 分配得来的 10 个字节的区域在堆区
    free(p_var);
    return 0;
}
```

栈和堆的区别

- 申请方式：栈是系统自动分配，堆是程序员主动申请。
- 申请后系统响应：分配栈空间，如果剩余空间大于申请空间则分配成功，否则分配失败栈溢出；申请堆空间，堆在内存中呈现的方式类似于链表（记录空闲地址空间的链表），在链表上寻找第一个大于申请空间的节点分配给程序，将该节点从链表中删除，大多数系统中该块空间的首地址存放的是本次分配空间的大小，便于释放，将该块空间上的剩余空间再次连接在空闲链表上。
- 栈在内存中是连续的一块空间（向低地址扩展）最大容量是系统预定好的，堆在内存中的空间（向高地址扩展）是不连续的。
- 申请效率：栈是有系统自动分配，申请效率高，但程序员无法控制；堆是由程序员主动申请，效率低，使用起来方便但是容易产生碎片。
- 存放的内容：栈中存放的是局部变量，函数的参数；堆中存放的内容由程序员控制。

变量的区别

全局变量、局部变量、静态全局变量、静态局部变量的区别
C++ 变量根据定义的位置的不同的生命周期，具有不同的作用域，作用域可分为 6 种：全局作用域，局部作用域，语句作用域，类作用域，命名空间作用域和文件作用域。

从作用域看：

- 全局变量：具有全局作用域。全局变量只需在一个源文件中定义，就可以作用于所有的源文件。当然，其他不包含全局变量定义的源文件需要用 extern 关键字再次声明这个全局变量。
- 静态全局变量：具有文件作用域。它与全局变量的区别在于如果程序包含多个文件的话，它作用于定义它的文件里，不能作用到其它文件里，即被 static 关键字修饰过的变量具有文件作用域。这样即使两个不同的源文件都定义了相同名字的静态全局变量，它们也是不同的变量。
- 局部变量：具有局部作用域。它是自动对象（auto），在程序运行期间不是一直存在，而是只在函数执行期间存在，函数的一次调用执行结束后，变量被撤销，其所占用的内存也被收回。
- 静态局部变量：具有局部作用域。它只被初始化一次，自从第一次被初始化直到程序运行结束都一直存在，它和全局变量的区别在于全局变量对所有的函数都是可见的，而静态局部变量只对定义自己的函数体始终可见。

从分配内存空间看：

- 静态存储区：全局变量，静态局部变量，静态全局变量。
- 栈：局部变量。
  说明：
- 静态变量和栈变量（存储在栈中的变量）、堆变量（存储在堆中的变量）的区别：静态变量会被放在程序的静态数据存储区（.data 段）中（静态变量会自动初始化），这样可以在下一次调用的时候还可以保持原来的赋值。而栈变量或堆变量不能保证在下一次调用的时候依然保持原来的值。
- 静态变量和全局变量的区别：静态变量用 static 告知编译器，自己仅仅在变量的作用范围内可见。

全局变量定义在头文件中有什么问题?

如果在头文件中定义全局变量，当该头文件被多个文件 `include` 时，该头文件中的全局变量就会被定义多次，导致重复定义，因此不能再头文件中定义全局变量。

对象创建限制在堆或栈

内存对齐

什么是内存对齐？内存对齐的原则？为什么要进行内存对齐，有什么优点？
内存对齐：编译器将程序中的每个“数据单元”安排在字的整数倍的地址指向的内存之中
内存对齐的原则：

1. 结构体变量的首地址能够被其最宽基本类型成员大小与对齐基数中的较小者所整除；
2. 结构体每个成员相对于结构体首地址的偏移量（offset）都是该成员大小与对齐基数中的较小者的整数倍，如有需要编译器会在成员之间加上填充字节（internal padding）；
3. 结构体的总大小为结构体最宽基本类型成员大小与对齐基数中的较小者的整数倍，如有需要编译器会在最末一个成员之后加上填充字节（trailing padding）。

进行内存对齐的原因：（主要是硬件设备方面的问题）

1. 某些硬件设备只能存取对齐数据，存取非对齐的数据可能会引发异常；
2. 某些硬件设备不能保证在存取非对齐数据的时候的操作是原子操作；
3. 相比于存取对齐的数据，存取非对齐的数据需要花费更多的时间；
4. 某些处理器虽然支持非对齐数据的访问，但会引发对齐陷阱（alignment trap）；
5. 某些硬件设备只支持简单数据指令非对齐存取，不支持复杂数据指令的非对齐存取。

   内存对齐的优点：
6. 便于在不同的平台之间进行移植，因为有些硬件平台不能够支持任意地址的数据访问，只能在某些地址处取某些特定的数据，否则会抛出异常；
7. 提高内存的访问效率，因为 CPU 在读取内存时，是一块一块的读取。

类的大小

说明：类的大小是指类的实例化对象的大小，用 sizeof 对类型名操作时，结果是该类型的对象的大小。
计算原则：

- 遵循结构体的对齐原则。
- 与普通成员变量有关，与成员函数和静态成员无关。即普通成员函数，静态成员函数，静态数据成员，静态常量数据成员均对类的大小无影响。因为静态数据成员被类的对象共享，并不属于哪个具体的对象。
- 虚函数对类的大小有影响，是因为虚函数表指针的影响。
- 虚继承对类的大小有影响，是因为虚基表指针带来的影响。
- 空类的大小是一个特殊情况，**空类的大小为 1**，当用 new 来创建一个空类的对象时，为了保证不同对象的地址不同，空类也占用存储空间。

带有虚函数的情况：（注意：虚函数的个数并不影响所占内存的大小，因为类对象的内存中只保存了指向虚函数表的指针。）

```
/*
说明：程序是在 64 位编译器下测试的
*/
#include <iostream>

using namespace std;

class A
{
private:
    static int s_var; // 不影响类的大小
    const int c_var;  // 4 字节
    int var;          // 8 字节 4 + 4 (int) = 8
    char var1;        // 12 字节 8 + 1 (char) + 3 (填充) = 12
public:
    A(int temp) : c_var(temp) {} // 不影响类的大小
    ~A() {}                      // 不影响类的大小
    virtual void f() { cout << "A::f" << endl; }

    virtual void h() { cout << "A::h" << endl; } // 24 字节 12 + 4 (填充)
+ 8 (指向虚函数的指针) = 24
    //没有虚函数时按照4字节对齐，需要12字节，有虚函数时按照8字节对齐需要24字节了
};

int main()
{
    A ex1(4);
    A *p;
    cout << sizeof(p) << endl;    // 8 字节 注意：指针所占的空间和指针指向的数据
类型无关
    cout << sizeof(ex1) << endl; // 24 字节
    return 0;
```

```
}
```

什么是内存泄露

内存泄漏：由于疏忽或误导致的程序未能释放已经不再使用的内存。
进一步解释：

并非指内存从物理上消失，而是指程序在运行过程中，由于疏忽或错误而失去了对该内存的控制，从而造成了内存的浪费。

常指堆内存泄漏，因为堆是动态分配的，而且是用户来控制的，如果使用不当，会产生内存泄漏。

使用 malloc、calloc、realloc、new 等分配内存时，使用完后要调用相应的 free 或 delete 释放内存，否则这块内存就会造成内存泄漏。

指针重新赋值

```
char *p = (char *)malloc(10);
char *p1 = (char *)malloc(10);
p = np;
```

开始时，指针 p 和 p1 分别指向一块内存空间，但指针 p 被重新赋值，导致 p 初始时指向的那块内存空间无法找到，从而发生了内存泄漏。

怎么防止内存泄漏？内存泄漏检测工具的原理？

防止内存泄漏的方法：

1. 内部封装：将内存的分配和释放封装到类中，在构造的时候申请内存，析构的时候释放内存。

```
#include <iostream>
#include <cstring>

using namespace std;

class A
{
private:
    char *p;
    unsigned int p_size;

public:
    A(unsigned int n = 1) // 构造函数中分配内存空间
    {
        p = new char[n];
        p_size = n;
    };
    ~A() // 析构函数中释放内存空间
```

```
    {
        if (p != NULL)
        {
            delete[] p; // 删除字符数组
            p = NULL;   // 防止出现野指针
        }
    };
    char *GetPointer()
    {
        return p;
    };
};
void fun()
{
    A ex(100);
    char *p = ex.GetPointer();
    strcpy(p, "Test");
    cout << p << endl;
}
int main()
{
    fun();
    return 0;
}
```

说明：但这样做并不是最佳的做法，在类的对象复制时，程序会出现同一块内存空间释放两次的情况，请看如下程序：

```
void fun1()
{
    A ex(100);
    A ex1 = ex;
    char *p = ex.GetPointer();
    strcpy(p, "Test");
    cout << p << endl;
}
```

简单解释：对于 fun1 这个函数中定义的两个类的对象而言，在离开该函数的作用域时，会两次调用析构函数来释放空间，但是这两个对象指向的是同一块内存空间，所以导致同一块内存空间被释放两次，可以通过增加计数机制来避免这种情况，看如下程序：

```
#include <iostream>
#include <cstring>

using namespace std;
class A
{
private:
    char *p;
```

```
    unsigned int p_size;
    int *p_count; // 计数变量
public:
    A(unsigned int n = 1) // 在构造函数中申请内存
    {
        p = new char[n];
        p_size = n;
        p_count = new int;
        *p_count = 1;
        cout << "count is : " << *p_count << endl;
    };
    A(const A &temp)
    {
        p = temp.p;
        p_size = temp.p_size;
        p_count = temp.p_count;
        (*p_count)++; // 复制时，计数变量 +1
        cout << "count is : " << *p_count << endl;
    }
    ~A()
    {
        (*p_count)--; // 析构时，计数变量 -1
        cout << "count is : " << *p_count << endl;

        if (*p_count == 0) // 只有当计数变量为 0 的时候才会释放该块内存空间
        {
            cout << "buf is deleted" << endl;
            if (p != NULL)
            {
                delete[] p; // 删除字符数组
                p = NULL;   // 防止出现野指针
                if (p_count != NULL)
                {
                    delete p_count;
                    p_count = NULL;
                }
            }
        }
    };
    char *GetPointer()
    {
        return p;
    };
};
void fun()
{
    A ex(100);
    char *p = ex.GetPointer();
    strcpy(p, "Test");
```

```
        cout << p << endl;

        A ex1 = ex; // 此时计数变量会 +1
        cout << "ex1.p = " << ex1.GetPointer() << endl;
    }
    int main()
    {
        fun();
        return 0;
    }
```

程序运行结果：

```
count is : 1
Test
count is : 2
ex1.p = Test
count is : 1
count is : 0
buf is deleted
```

解释下：程序运行结果的倒数 2、3 行是调用两次析构函数时进行的操作，在第二次调用析构函数时，进行内存空间的释放，从而会有倒数第 1 行的输出结果。

智能指针
智能指针是 C++ 中已经对内存泄漏封装好了一个工具，可以直接拿来使用，将在下一个问题中对智能指针进行详细的解释。

内存泄漏检测工具的实现原理：

智能指针有哪几种？智能指针的实现原理?

智能指针是为了解决动态内存分配时带来的内存泄漏以及多次释放同一块内存空间而提出的。C++11 中封装在了 头文件中。

C++11 中智能指针包括以下三种：

- 共享指针（shared_ptr）：资源可以被多个指针共享，使用计数机制表明资源被几个指针共享。通过 use_count() 查看资源的所有者的个数，可以通过 unique_ptr、weak_ptr 来构造，调用 release() 释放资源的所有权，计数减一，当计数减为 0 时，会自动释放内存空间，从而避免了内存泄漏。
- 独占指针（unique_ptr）：独享所有权的智能指针，资源只能被一个指针占有，该指针不能拷贝构造和赋值。但可以进行移动构造和移动赋值构造（调用 move() 函数），即一个 unique_ptr 对象赋值给另一个 unique_ptr 对象，可以通过该方法进行赋值。
- 弱指针（weak_ptr）：指向 share_ptr 指向的对象，能够解决由shared_ptr带来的循环引用问题。
  智能指针的实现原理： 计数原理。

```
#include <iostream>
#include <memory>

template <typename T>
class SmartPtr
{
private :
    T *_ptr;
    size_t *_count;

public:
    SmartPtr(T *ptr = nullptr) : _ptr(ptr)
    {
        if (_ptr)
        {
            _count = new size_t(1);
        }
        else
        {
            _count = new size_t(0);
        }
    }

    ~SmartPtr()
    {
        (*this->_count)--;
        if (*this->_count == 0)
        {
            delete this->_ptr;
            delete this->_count;
        }
    }

    SmartPtr(const SmartPtr &ptr) // 拷贝构造：计数 +1
    {
        if (this != &ptr)
        {
            this->_ptr = ptr._ptr;
            this->_count = ptr._count;
            (*this->_count)++;
        }
    }

    SmartPtr &operator=(const SmartPtr &ptr) // 赋值运算符重载
    {
        if (this->_ptr == ptr._ptr)
        {
            return *this;
        }
```

```
        if (this->_ptr) // 将当前的 ptr 指向的原来的空间的计数 -1
        {
            (*this->_count)--;
            if (this->_count == 0)
            {
                delete this->_ptr;
                delete this->_count;
            }
        }
        this->_ptr = ptr._ptr;
        this->_count = ptr._count;
        (*this->_count)++; // 此时 ptr 指向了新赋值的空间，该空间的计数 +1
        return *this;
    }

    T &operator*()
    {
        assert(this->_ptr == nullptr);
        return *(this->_ptr);
    }

    T *operator->()
    {
        assert(this->_ptr == nullptr);
        return this->_ptr;
    }

    size_t use_count()
    {
        return *this->count;
    }
};
```

一个 unique_ptr 怎么赋值给另一个 unique_ptr 对象?

借助 `std::move()` 可以实现将一个 `unique_ptr` 对象赋值给另一个 `unique_ptr` 对象，其目的是实现所有权的转移。

```
// A 作为一个类
std::unique_ptr<A> ptr1(new A());
std::unique_ptr<A> ptr2 = std::move(ptr1);
```

使用智能指针会出现什么问题？怎么解决?

智能指针可能出现的问题：循环引用
在如下例子中定义了两个类 Parent、Child，在两个类中分别定义另一个类的对象的共享指针，由于在程序结束后，两个指针相互指向对方的内存空间，导致内存无法释放。

循环引用的解决方法： weak_ptr

循环引用：该被调用的析构函数没有被调用，从而出现了内存泄漏。

weak_ptr 对被 shared_ptr 管理的对象存在 非拥有性（弱）引用，在访问所引用的对象前必须先转化为 shared_ptr；
weak_ptr 用来打断 shared_ptr 所管理对象的循环引用问题，若这种环被孤立（没有指向环中的外部共享指针），shared_ptr 引用计数无法抵达 0，内存被泄露；令环中的指针之一为弱指针可以避免该情况；
weak_ptr 用来表达临时所有权的概念，当某个对象只有存在时才需要被访问，而且随时可能被他人删除，可以用 weak_ptr 跟踪该对象；需要获得所有权时将其转化为 shared_ptr，此时如果原来的 shared_ptr 被销毁，则该对象的生命期被延长至这个临时的 shared_ptr 同样被销毁。

```
#include <iostream>
#include <memory>

using namespace std;

class Child;
class Parent;

class Parent {
private:
    //shared_ptr<Child> ChildPtr;
    weak_ptr<Child> ChildPtr;
public:
    void setChild(shared_ptr<Child> child) {
        this->ChildPtr = child;
    }

    void doSomething() {
        //new shared_ptr
        if (this->ChildPtr.lock()) {

        }
    }

    ~Parent() {
    }
};

class Child {
private:
    shared_ptr<Parent> ParentPtr;
public:
    void setPartent(shared_ptr<Parent> parent) {
        this->ParentPtr = parent;
    }
    void doSomething() {
        if (this->ParentPtr.use_count()) {
```

```
        }
    }
    ~Child() {
    }
};

int main() {
    weak_ptr<Parent> wpp;
    weak_ptr<Child> wpc;
    {
        shared_ptr<Parent> p(new Parent);
        shared_ptr<Child> c(new Child);
        p->setChild(c);
        c->setPartent(p);
        wpp = p;
        wpc = c;
        cout << p.use_count() << endl; // 2
        cout << c.use_count() << endl; // 1
    }
    cout << wpp.use_count() << endl;  // 0
    cout << wpc.use_count() << endl;  // 0
    return 0;
}
```

语言对比

C++ 11 新特性

说明：C++11 的新特性有很多，从面试的角度来讲，如果面试官问到该问题，常以该问题作为引子，对面试者提到的知识点进行深入展开提问。面试者尽可能的列举常用的并且熟悉的特性，尽可能的掌握相关原理，下文只是对相关知识点进行了简单的阐述，有关细节还需要结合相关知识点的相关问题。
下面对常用的做一下总结：

auto 类型推导

auto 关键字：自动类型推导，编译器会在 编译期间 通过初始值推导出变量的类型，通过 auto 定义的变量必须有初始值。

auto 关键字基本的使用语法如下：

```
auto var = val1 + val2; // 根据 val1 和 val2 相加的结果推断出 var 的类型，
```

注意：编译器推导出来的类型和初始值的类型并不完全一样，编译器会适当地改变结果类型使其更符合初始化规则。

decltype 类型推导

decltype 关键字：decltype 是“declare type”的缩写，译为“声明类型”。和 auto 的功能一样，都用来在编译时期进行自动类型推导。如果希望从表达式中推断出要定义的变量的类型，但是不想用该表达式的值初始化变量，这时就不能再用 auto。decltype 作用是选择并返回操作数的数据类型。

区别：

```
auto var = val1 + val2;
decltype(val1 + val2) var1 = 0;
```

auto 根据 = 右边的初始值 val1 + val2 推导出变量的类型，并将该初始值赋值给变量 var；decltype 根据 val1 + val2 表达式推导出变量的类型，变量的初始值和与表达式的值无关。

auto 要求变量必须初始化，因为它是根据初始化的值推导出变量的类型，而 decltype 不要求，定义变量的时候可初始化也可以不初始化。

lambda 表达式

lambda 表达式，又被称为 lambda 函数或者 lambda 匿名函数。

lambda匿名函数的定义:

```
[capture list] (parameter list) -> return type
{
   function body;
};
```

其中：

capture list：捕获列表，指 lambda 所在函数中定义的局部变量的列表，通常为空。

return type、parameter list、function body：分别表示返回值类型、参数列表、函数体，和普通函数一样。

举例：

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

int main()
{
    int arr[4] = {4, 2, 3, 1};
    //对 a 数组中的元素进行升序排序
    sort(arr, arr+4, [=](int x, int y) -> bool{ return x < y; } );
    for(int n : arr){
        cout << n << " ";
    }
    return 0;
}
```

范围 for 语句

语法格式：

```
for (declaration : expression){
    statement
}
```

参数的含义：

expression：必须是一个序列，例如用花括号括起来的初始值列表、数组、vector，string等，这些类型的共同特点是拥有能返回迭代器的beign、end成员。
declaration：此处定义一个变量，序列中的每一个元素都能转化成该变量的类型，常用auto类型说明符。
实例：

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    char arr[] = "hello world!";
    for (char c : arr) {
        cout << c;
    }
    return 0;
}
```

程序执行结果为：

```
hello world!
```

右值引用
右值引用：绑定到右值的引用，用 && 来获得右值引用，右值引用只能绑定到要销毁的对象。为了和右值引用区分开，常规的引用称为左值引用。
举例：
C++

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    int var = 42;
    int &l_var = var;
    int &&r_var = var; // error: cannot bind rvalue reference of type
//'int&&' to lvalue of type 'int' 错误：不能将右值引用绑定到左值上

int &&r_var2 = var + 40; // 正确：将 r_var2 绑定到求和结果上
return 0;

}
```

标准库 move() 函数

move() 函数：通过该函数可获得绑定到左值上的右值引用，该函数包括在 utility 头文件中。该知识点会在后续的章节中做详细的说明。

智能指针

相关知识已在第一章中进行了详细的说明，这里不再重复。

delete 函数和 default 函数

delete 函数：= delete 表示该函数不能被调用。
default 函数：= default 表示编译器生成默认的函数，例如：生成默认的构造函数。

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

class A
{
public:
    A() = default; // 表示使用默认的构造函数
    ~A() = default; // 表示使用默认的析构函数
    A(const A &) = delete; // 表示类的对象禁止拷贝构造
    A &operator=(const A &) = delete; // 表示类的对象禁止拷贝赋值
};
int main()
{
    A ex1;
    A ex2 = ex1; // error: use of deleted function 'A::A(const A&)'
    A ex3;
    ex3 = ex1; // error: use of deleted function 'A& A::operator=(const
A&)'
    return 0;
}
```

C 和 C++ 的区别

首先说一下面向对象和面向过程：

1. 面向过程的思路：分析解决问题所需的步骤，用函数把这些步骤依次实现。
   面向对象的思路：把构成问题的事务分解为各个对象，建立对象的目的，不是完成一个步骤，而是描述某个事务在解决整个问题步骤中的行为。
   区别和联系：
2. 语言自身：C 语言是面向过程的编程，它最重要的特点是函数，通过 main 函数来调用各个子函数。程序运行的顺序都是程序员事先决定好的。C++ 是面向对象的编程，类是它的主要特点，在程序执行过程中，先由主 main 函数进入，定义一些类，根据需要执行类的成员函数，过程的概念被淡化了（实际上过程还是有的，就是主函数的那些语句。），以类驱动程序运行，类就是对象，所以我们称之为面向对象程序设计。面向对象在分析和解决问题的时候，将涉及到的数据和数据的操作封装在类中，通过类可以创建对象，以事件或消息来驱动对象执行处理。
3. 应用领域：C 语言主要用于嵌入式领域，驱动开发等与硬件直接打交道的领域，C++ 可以用于应用层开发，用户界面开发等与操作系统打交道的领域。

4. C++ 既继承了 C 强大的底层操作特性，又被赋予了面向对象机制。它特性繁多，面向对象语言的多继承，对值传递与引用传递的区分以及 const 关键字，等等。
5. C++ 对 C 的“增强”，表现在以下几个方面：类型检查更为严格。增加了面向对象的机制、泛型编程的机制（Template）、异常处理、运算符重载、标准模板库（STL）、命名空间（避免全局命名冲突）。

Java 和 C++ 的区别

二者在语言特性上有很大的区别：

- 指针：C++ 可以直接操作指针，容易产生内存泄漏以及非法指针引用的问题；Java 并不是没有指针，虚拟机（JVM）内部还是使用了指针，只是编程人员不能直接使用指针，不能通过指针来直接访问内存，并且 Java 增加了内存管理机制。
- 多重继承：C++ 支持多重继承，允许多个父类派生一个类，虽然功能很强大，但是如果使用的不当会造成很多问题，例如：菱形继承；Java 不支持多重继承，但允许一个类可以继承多个接口，可以实现 C++ 多重继承的功能，但又避免了多重继承带来的许多不便。
- 数据类型和类：Java 是完全面向对象的语言，所有函数和变量部必须是类的一部分。除了基本数据类型之外，其余的都作为类对象，包括数组。对象将数据和方法结合起来，把它们封装在类中，这样每个对象都可实现自己的特点和行为。而 C++ 允许将函数和变量定义为全局的。

垃圾回收：

- Java 语言一个显著的特点就是垃圾回收机制，编程人员无需考虑内存管理的问题，可以有效的防止内存泄漏，有效的使用空闲的内存。
- Java 所有的对象都是用 new 操作符建立在内存堆栈上，类似于 C++ 中的 new 操作符，但是当要释放该申请的内存空间时，Java 自动进行内存回收操作，C++ 需要程序员自己释放内存空间，并且 Java 中的内存回收是以线程的方式在后台运行的，利用空闲时间。

应用场景：

- Java 运行在虚拟机上，和开发平台无关，C++ 直接编译成可执行文件，是否跨平台在于用到的编译器的特性是否有多平台的支持。
- C++ 可以直接编译成可执行文件，运行效率比 Java 高。
- Java 主要用来开发 Web 应用。
- C++ 主要用在嵌入式开发、网络、并发编程的方面。

Python 和 C++ 的区别

区别：

- 语言自身：Python 为脚本语言，解释执行，不需要经过编译；C++ 是一种需要编译后才能运行的语言，在特定的机器上编译后运行。
- 运行效率：C++ 运行效率高，安全稳定。原因：Python 代码和 C++ 最终都会变成 CPU 指令来跑，但一般情况下，比如反转和合并两个字符串，Python 最终转换出来的 CPU 指令会比 C++ 多很多。首先，Python 中涉及的内容比 C++ 多，经过了更多层，Python 中甚至连数字都是 object ；其次，Python 是解释执行的，和物理机 CPU 之间多了解释器这层，而 C++ 是编译执行的，直接就是机器码，编译的时候编译器又可以进行一些优化。

- 开发效率：Python 开发效率高。原因：Python 一两句代码就能实现的功能，C++ 往往需要更多的代码才能实现。
- 书写格式和语法不同：Python 的语法格式不同于其 C++ 定义声明才能使用，而且极其灵活，完全面向更上层的开发者。

面向对象

什么是面向对象？面向对象的三大特性

面向对象：对象是指具体的某一个事物，这些事物的抽象就是类，类中包含数据（成员变量）和动作（成员方法）。

面向对象的三大特性：

- 封装：将具体的实现过程和数据封装成一个函数，只能通过接口进行访问，降低耦合性。
- 继承：子类继承父类的特征和行为，子类有父类的非 private 方法或成员变量，子类可以对父类的方法进行重写，增强了类之间的耦合性，但是当父类中的成员变量、成员函数或者类本身被 final 关键字修饰时，修饰的类不能继承，修饰的成员不能重写或修改。
- 多态：多态就是不同继承类的对象，对同一消息做出不同的响应，基类的指针指向或绑定到派生类的对象，使得基类指针呈现不同的表现方式。

重载、重写、隐藏的区别

概念解释：

重载：是指同一可访问区内被声明几个具有不同参数列（参数的类型、个数、顺序）的同名函数，根据参数列表确定调用哪个函数，重载不关心函数返回类型。

```
class A
{
public:
    void fun(int tmp);
    void fun(float tmp);         // 重载 参数类型不同（相对于上一个函数）
    void fun(int tmp, float tmp1); // 重载 参数个数不同（相对于上一个函数）
    void fun(float tmp, int tmp1); // 重载 参数顺序不同（相对于上一个函数）
    int fun(int tmp);             // error: 'int A::fun(int)' cannot be
overloaded 错误：注意重载不关心函数返回类型
};
```

隐藏：是指派生类的函数屏蔽了与其同名的基类函数，主要只要同名函数，不管参数列表是否相同，基类函数都会被隐藏。

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

class Base
{
```

```
public:
    void fun(int tmp, float tmp1) { cout << "Base::fun(int tmp, float
tmp1)" << endl; }
};

class Derive : public Base
{
public:
    void fun(int tmp) { cout << "Derive::fun(int tmp)" << endl; } // 隐藏
基类中的同名函数
};

int main()
{
    Derive ex;
    ex.fun(1);        // Derive::fun(int tmp)
    ex.fun(1, 0.01); // error: candidate expects 1 argument, 2 provided
    return 0;
}
```

说明：上述代码中 ex.fun(1, 0.01); 出现错误，说明派生类中将基类的同名函数隐藏了。若是想调用基类中的同名函数，可以加上类型名指明 ex.Base::fun(1, 0.01);，这样就可以调用基类中的同名函数。

重写(覆盖)：是指派生类中存在重新定义的函数。函数名、参数列表、返回值类型都必须同基类中被重写的函数一致，只有函数体不同。派生类调用时会调用派生类的重写函数，不会调用被重写函数。重写的基类中被重写的函数必须有 virtual 修饰。

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

class Base
{
public:
    virtual void fun(int tmp) { cout << "Base::fun(int tmp) : " << tmp
<< endl; }
};

class Derived : public Base
{
public:
    virtual void fun(int tmp) { cout << "Derived::fun(int tmp) : " <<
tmp << endl; } // 重写基类中的 fun 函数
};
int main()
{
    Base *p = new Derived();
    p->fun(3); // Derived::fun(int) : 3
    return 0;
```

```
}
```

重写和重载的区别：

- 范围区别：对于类中函数的重载或者重写而言，重载发生在同一个类的内部，重写发生在不同的类之间（子类和父类之间）。
- 参数区别：重载的函数需要与原函数有相同的函数名、不同的参数列表，不关注函数的返回值类型；重写的函数的函数名、参数列表和返回值类型都需要和原函数相同，父类中被重写的函数需要有 virtual 修饰。
- virtual 关键字：重写的函数基类中必须有 virtual关键字的修饰，重载的函数可以有 virtual 关键字的修饰也可以没有。

隐藏和重写，重载的区别：

- 范围区别：隐藏与重载范围不同，隐藏发生在不同类中。
- 参数区别：隐藏函数和被隐藏函数参数列表可以相同，也可以不同，但函数名一定相同；当参数不同时，无论基类中的函数是否被 virtual 修饰，基类函数都是被隐藏，而不是重写。

如何理解 C++ 是面向对象编程

说明：该问题最好结合自己的项目经历进行展开解释，或举一些恰当的例子，同时对比下面向过程编程。

- 面向过程编程：一种以执行程序操作的过程或函数为中心编写软件的方法。程序的数据通常存储在变量中，与这些过程是分开的。所以必须将变量传递给需要使用它们的函数。缺点：随着程序变得越来越复杂，程序数据与运行代码的分离可能会导致问题。例如，程序的规范经常会发生变化，从而需要更改数据的格式或数据结构的设计。当数据结构发生变化时，对数据进行操作的代码也必须更改为接受新的格式。查找需要更改的所有代码会为程序员带来额外的工作，并增加了使代码出现错误的机会。
- 面向对象编程（Object-Oriented Programming, OOP）：以创建和使用对象为中心。一个对象（Object）就是一个软件实体，它将数据和程序在一个单元中组合起来。对象的数据项，也称为其属性，存储在成员变量中。对象执行的过程被称为其成员函数。将对象的数据和过程绑定在一起则被称为封装。

面向对象编程进一步说明：
面向对象编程将数据成员和成员函数封装到一个类中，并声明数据成员和成员函数的访问级别（public、private、protected），以便控制类对象对数据成员和函数的访问，对数据成员起到一定的保护作用。而且在类的对象调用成员函数时，只需知道成员函数的名、参数列表以及返回值类型即可，无需了解其函数的实现原理。当类内部的数据成员或者成员函数发生改变时，不影响类外部的代码。

什么是多态？多态如何实现？

多态：多态就是不同继承类的对象，对同一消息做出不同的响应，基类的指针指向或绑定到派生类的对象，使得基类指针呈现不同的表现方式。在基类的函数前加上 virtual 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。
实现方法：多态是通过虚函数实现的，虚函数的地址保存在虚函数表中，虚函数表的地址保存在含有虚函数的类的实例对象的内存空间中。

实现过程：

1. 在类中用 virtual 关键字声明的函数叫做虚函数；
2. 存在虚函数的类都有一个虚函数表，当创建一个该类的对象时，该对象有一个指向虚函数表的虚表指针（虚函数表和类对应的，虚表指针是和对象对应）；
3. 当基类指针指向派生类对象，基类指针调用虚函数时，基类指针指向派生类的虚表指针，由于该虚表指针指向派生类虚函数表，通过遍历虚表，寻找相应的虚函数。

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

class Base
{
public:
    virtual void fun() { cout << "Base::fun()" << endl; }

    virtual void fun1() { cout << "Base::fun1()" << endl; }

    virtual void fun2() { cout << "Base::fun2()" << endl; }
};
class Derive : public Base
{
public:
    void fun() { cout << "Derive::fun()" << endl; }

    virtual void D_fun1() { cout << "Derive::D_fun1()" << endl; }

    virtual void D_fun2() { cout << "Derive::D_fun2()" << endl; }
};
int main()
{
    Base *p = new Derive();
    p->fun(); // Derive::fun() 调用派生类中的虚函数
    return 0;
}
```

基类的虚函数表如下：

派生类的对象虚函数表如下：

简单解释：当基类的指针指向派生类的对象时，通过派生类的对象的虚表指针找到虚函数表（派生类的对象虚函数表），进而找到相应的虚函数 Derive::f() 进行调用。

关键字库函数

sizeof 和 strlen 的区别

1. strlen 是头文件 中的函数，sizeof 是 C++ 中的运算符。
2. strlen 测量的是字符串的实际长度（其源代码如下），以 \0 结束。而 sizeof 测量的是字符数组的分配大小。
   strlen 源代码:

```
size_t strlen(const char *str) {
    size_t length = 0;
    while (*str++)
        ++length;
    return length;
}
```

举例：

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char arr[10] = "hello";
    cout << strlen(arr) << endl; // 5
    cout << sizeof(arr) << endl; // 10
    return 0;
}
```

3. 若字符数组 arr 作为函数的形参，sizeof(arr) 中 arr 被当作字符指针来处理，strlen(arr) 中 arr 依然是字符数组，从下述程序的运行结果中就可以看出。

```
#include <iostream>
#include <cstring>

using namespace std;

void size_of(char arr[])
{
    cout << sizeof(arr) << endl; // warning: 'sizeof' on array
function parameter 'arr' will return size of 'char*' .
    cout << strlen(arr) << endl;
}

int main()
{
    char arr[20] = "hello";
    size_of(arr);
    return 0;
}
```

```
输出结果：
8
5
```

4. strlen 本身是库函数，因此在程序运行过程中，计算长度；而 sizeof 在编译时，计算长度；
5. sizeof 的参数可以是类型，也可以是变量；strlen 的参数必须是 char* 类型的变量。

lambda 表达式（匿名函数）的具体应用和使用场景

lambda 表达式的定义形式如下：

```
[capture list] (parameter list) -> reurn type
{
   function body
}
```

其中：

capture list：捕获列表，指 lambda 表达式所在函数中定义的局部变量的列表，通常为空，但如果函数体中用到了 lambda 表达式所在函数的局部变量，必须捕获该变量，即将此变量写在捕获列表中。捕获方式分为：引用捕获方式 [&]、值捕获方式 [=]。
return type、parameter list、function body：分别表示返回值类型、参数列表、函数体，和普通函数一样。
举例：
lambda 表达式常搭配排序算法使用。

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

int main()
{
    vector<int> arr = {3, 4, 76, 12, 54, 90, 34};
    sort(arr.begin(), arr.end(), [](int a, int b) { return a > b; }); //
降序排序
    for (auto a : arr)
    {
        cout << a << " ";
    }
    return 0;
}
```

explicit 的作用（如何避免编译器进行隐式类型转换）

作用：用来声明类构造函数是显示调用的，而非隐式调用，可以阻止调用构造函数时进行隐式转换。只可用于修饰单参构造函数，因为无参构造函数和多参构造函数本身就是显示调用的，再加上 explicit 关键字也没有什么意义。

隐式转换：

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

class A
{
public:
    int var;
    A(int tmp)
    {
        var = tmp;
    }
};
int main()
{
    A ex = 10; // 发生了隐式转换
    return 0;
}
```

上述代码中，A ex = 10; 在编译时，进行了隐式转换，将 10 转换成 A 类型的对象，然后将该对象赋值给 ex，等同于如下操作：

为了避免隐式转换，可用 explicit 关键字进行声明：

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

class A
{
public:
    int var;
    explicit A(int tmp)
    {
        var = tmp;
        cout << var << endl;
    }
};
int main()
{
    A ex(100);
```

```
    A ex1 = 10; // error: conversion from 'int' to non-scalar type 'A'
requested
    return 0;
}
```

C 和 C++ static 的区别

- 在 C 语言中，使用 static 可以定义局部静态变量、外部静态变量、静态函数
- 在 C++ 中，使用 static 可以定义局部静态变量、外部静态变量、静态函数、静态成员变量和静态成员函数。因为 C++ 中有类的概念，静态成员变量、静态成员函数都是与类有关的概念。

static 的作用

作用：
static 定义静态变量，静态函数。

保持变量内容持久：static 作用于局部变量，改变了局部变量的生存周期，使得该变量存在于定义后直到程序运行结束的这段时间。

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

int fun(){
    static int var = 1; // var 只在第一次进入这个函数的时初始化
    var += 1;
    return var;
}

int main()
{
    for(int i = 0; i < 10; ++i)
        cout << fun() << " "; // 2 3 4 5 6 7 8 9 10 11
    return 0;
}
```

隐藏：static 作用于全局变量和函数，改变了全局变量和函数的作用域，使得全局变量和函数只能在定义它的文件中使用，在源文件中不具有全局可见性。（注：普通全局变量和函数具有全局可见性，即其他的源文件也可以使用。）

static 作用于类的成员变量和类的成员函数，使得类变量或者类成员函数和类有关，也就是说可以不定义类的对象就可以通过类访问这些静态成员。注意：类的静态成员函数中只能访问静态成员变量或者静态成员函数，不能将静态成员函数定义成虚函数。

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

class A
{
```

```
private:
    int var;
    static int s_var; // 静态成员变量
public:
    void show()
    {
        cout << s_var++ << endl;
    }
    static void s_show()
    {
        cout << s_var << endl;
        // cout << var << endl; // error: invalid use of member 'A::a'
in static member function. 静态成员函数不能调用非静态成员变量。无法使用 this.var
        // show();  // error: cannot call member function 'void
A::show()' without object. 静态成员函数不能调用非静态成员函数。无法使用
this.show()
    }
};
int A::s_var = 1;  // 静态成员变量在类外进行初始化赋值，默认初始化为 0

int main()
{



    // cout << A::sa << endl;    // error: 'int A::sa' is private within
this context
    A ex;
    ex.show();
    A::s_show();

}
```

static 在类中使用的注意事项（定义、初始化和使用）

static 静态成员变量：

- 静态成员变量是在类内进行声明，在类外进行定义和初始化，在类外进行定义和初始化的时候不要出现 static 关键字和private、public、protected 访问规则。
- 静态成员变量相当于类域中的全局变量，被类的所有对象所共享，包括派生类的对象。静态成员变量可以作为成员函数的参数，而普通成员变量不可以。

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

class A
{
public:
    static int s_var;
    int var;
```

```
    void fun1(int i = s_var); // 正确，静态成员变量可以作为成员函数的参数
    void fun2(int i = var);   //  error: invalid use of non-static
data member 'A::var'
};
int main()
{
    return 0;
}
```

- 静态数据成员的类型可以是所属类的类型，而普通数据成员的类型只能是该类类型的指针或引用。

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

class A
{
public:
    static A s_var; // 正确，静态数据成员
    A var;          // error: field 'var' has incomplete type 'A'
    A *p;           // 正确，指针
    A &var1;        // 正确，引用
};

int main()
{
    return 0;
}
```

static 静态成员函数：

- 静态成员函数不能调用非静态成员变量或者非静态成员函数，因为静态成员函数没有 this 指针。静态成员函数做为类作用域的全局函数。
- 静态成员函数不能声明成虚函数（virtual）、const 函数和 volatile 函数。

static 全局变量和普通全局变量的异同

相同点：

- 存储方式：普通全局变量和 static 全局变量都是静态存储方式。

不同点：

- 作用域：普通全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，普通全局变量在各个源文件中都是有效的；静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。
- 初始化：静态全局变量只初始化一次，防止在其他文件中使用。

const 作用及用法

作用：

- const 修饰成员变量，定义成 const 常量，相较于宏常量，可进行类型检查，节省内存空间，提高了效率。
- const 修饰函数参数，使得传递过来的函数参数的值不能改变。
- const 修饰成员函数，使得成员函数不能修改任何类型的成员变量（mutable 修饰的变量除外），也不能调用非 const 成员函数，因为非 const 成员函数可能会修改成员变量。

在类中的用法：

const 成员变量：

- const 成员变量只能在类内声明、定义，在构造函数初始化列表中初始化。
- const 成员变量只在某个对象的生存周期内是常量，对于整个类而言却是可变的，因为类可以创建多个对象，不同类的 const 成员变量的值是不同的。因此不能在类的声明中初始化 const 成员变量，类的对象还没有创建，编译器不知道他的值。

const 成员函数：

- 不能修改成员变量的值，除非有 mutable 修饰；只能访问成员变量。
  不能调用非常量成员函数，以防修改成员变量的值。

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

class A
{
public:
    int var;
    A(int tmp) : var(tmp) {}
    void c_fun(int tmp) const // const 成员函数
    {
        var = tmp; // error: assignment of member 'A::var' in read-only
object. 在 const 成员函数中，不能修改任何类成员变量。
        fun(tmp); // error: passing 'const A' as 'this' argument
discards qualifiers. const 成员函数不能调用非 const 成员函数，因为非 const 成员
函数可能会修改成员变量。
    }

    void fun(int tmp)
    {
        var = tmp;
    }

};

int main()
{
```

```
    return 0;
}
```

define 和 const 的区别

区别：

- 编译阶段：define 是在编译预处理阶段进行替换，const 是在编译阶段确定其值。
- 安全性：define 定义的宏常量没有数据类型，只是进行简单的替换，不会进行类型安全的检查；const 定义的常量是有类型的，是要进行判断的，可以避免一些低级的错误。
- 内存占用：define 定义的宏常量，在程序中使用多少次就会进行多少次替换，内存中有多个备份，占用的是代码段的空间；const 定义的常量占用静态存储区的空间，程序运行过程中只有一份。
- 调试：define 定义的宏常量不能调试，因为在预编译阶段就已经进行替换了；const 定义的常量可以进行调试。

const 的优点：

- 有数据类型，在定义式可进行安全性检查。
- 可调试。
- 占用较少的空间。

define 和 typedef 的区别

- 原理：#define 作为预处理指令，在编译预处理时进行替换操作，不作正确性检查，只有在编译已被展开的源程序时才会发现可能的错误并报错。typedef 是关键字，在编译时处理，有类型检查功能，用来给一个已经存在的类型一个别名，但不能在一个函数定义里面使用 typedef 。
- 功能：typedef 用来定义类型的别名，方便使用。#define 不仅可以为类型取别名，还可以定义常量、变量、编译开关等。
- 作用域：#define 没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用，而 typedef 有自己的作用域。
- 指针的操作：typedef 和 #define 在处理指针时不完全一样

```
#include <iostream>
#define INTPTR1 int *
typedef int * INTPTR2;

using namespace std;

int main()
{
    INTPTR1 p1, p2; // p1: int *; p2: int
    INTPTR2 p3, p4; // p3: int *; p4: int *

    int var = 1;
    const INTPTR1 p5 = &var; // 相当于 const int * p5; 常量指针，即不可以通过 p5 去修改 p5 指向的内容，但是 p5 可以指向其他内容。
    const INTPTR2 p6 = &var; // 相当于 int * const p6; 指针常量，不可使 p6 再指向其他内容。
```

```
    return 0;
}
```

用宏实现比较大小，以及两个数中的最小值

```
#define MAX(X, Y) ((X)>(Y)?(X):(Y))
#define MIN(X, Y) ((X)<(Y)?(X):(Y))
```

inline 作用及使用方法

作用：
inline 是一个关键字，可以用于定义内联函数。内联函数，像普通函数一样被调用，但是在调用时并不通过函数调用的机制而是直接在调用点处展开，这样可以大大减少由函数调用带来的开销，从而提高程序的运行效率。

使用方法：

1. 类内定义成员函数默认是内联函数
   在类内定义成员函数，可以不用在函数头部加 inline 关键字，因为编译器会自动将类内定义的函数（构造函数、析构函数、普通成员函数等）声明为内联函数，代码如下：

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

   class A{
   public:
       int var;
       A(int tmp){
         var = tmp;
       }
       void fun(){
           cout << var << endl;
       }
   };

   int main()
   {
       return 0;
   }
   ```

2. 类外定义成员函数，若想定义为内联函数，需用关键字声明
   当在类内声明函数，在类外定义函数时，如果想将该函数定义为内联函数，则可以在类内声明时不加 inline 关键字，而在类外定义函数时加上 inline 关键字。

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

```
class A{
public:
    int var;
    A(int tmp){
      var = tmp;
    }
    void fun();
};

inline void A::fun(){
    cout << var << endl;
}

int main()
{
    return 0;
}
```

另外，可以在声明函数和定义函数的同时加上 inline；也可以只在函数声明时加 inline，而定义函数时不加 inline。只要确保在调用该函数之前把 inline 的信息告知编译器即可。

inline 函数工作原理

- 内联函数不是在调用时发生控制转移关系，而是在编译阶段将函数体嵌入到每一个调用该函数的语句块中，编译器会将程序中出现内联函数的调用表达式用内联函数的函数体来替换。
- 普通函数是将程序执行转移到被调用函数所存放的内存地址，当函数执行完后，返回到执行此函数前的地方。转移操作需要保护现场，被调函数执行完后，再恢复现场，该过程需要较大的资源开销。

宏定义（define）和内联函数（inline）的区别

1. 内联函数是在编译时展开，而宏在编译预处理时展开；在编译的时候，内联函数直接被嵌入到目标代码中去，而宏只是一个简单的文本替换。
2. 内联函数是真正的函数，和普通函数调用的方法一样，在调用点处直接展开，避免了函数的参数压栈操作，减少了调用的开销。而宏定义编写较为复杂，常需要增加一些括号来避免歧义。
3. 宏定义只进行文本替换，不会对参数的类型、语句能否正常编译等进行检查。而内联函数是真正的函数，会对参数的类型、函数体内的语句编写是否正确等进行检查。

使用举例：

```
#include <iostream>

#define MAX(a, b) ((a) > (b) ? (a) : (b))

using namespace std;

inline int fun_max(int a, int b)
{
```

```
    return a > b ? a : b;
}

int main()
{
    int var = 1;
    cout << MAX(var, 5) << endl;
    cout << fun_max(var, 0) << endl;
    return 0;
}
/*
程序运行结果：
5
1

*/
```

new 的作用?

new 是 C++ 中的关键字，用来动态分配内存空间，实现方式如下：

```
int *p = new int[5];
```

new 和 malloc 如何判断是否申请到内存?

- malloc ：成功申请到内存，返回指向该内存的指针；分配失败，返回 NULL 指针。
- new ：内存分配成功，返回该对象类型的指针；分配失败，抛出 bad_alloc 异常。

delete 实现原理？delete 和 delete[] 的区别?

delete 的实现原理：

- 首先执行该对象所属类的析构函数；
- 进而通过调用 operator delete 的标准库函数来释放所占的内存空间。

delete 和 delete [] 的区别：

- delete 用来释放单个对象所占的空间，只会调用一次析构函数；
- delete [] 用来释放数组空间，会对数组中的每个成员都调用一次析构函数。

new 和 malloc 的区别，delete 和 free 的区别

在使用的时候 new、delete 搭配使用，malloc、free 搭配使用。

- malloc、free 是库函数，而new、delete 是关键字。
  -new 申请空间时，无需指定分配空间的大小，编译器会根据类型自行计算；malloc 在申请空间时，需要确定所申请空间的大小。
- new 申请空间时，返回的类型是对象的指针类型，无需强制类型转换，是类型安全的操作符；malloc 申请空间时，返回的是 void* 类型，需要进行强制类型的转换，转换为对象类型的指针。

- new 分配失败时，会抛出 bad_alloc 异常，malloc 分配失败时返回空指针。
- 对于自定义的类型，new 首先调用 operator new() 函数申请空间（底层通过 malloc 实现），然后调用构造函数进行初始化，最后返回自定义类型的指针；delete 首先调用析构函数，然后调用 operator delete() 释放空间（底层通过 free 实现）。malloc、free 无法进行自定义类型的对象的构造和析构。
- new 操作符从自由存储区上为对象动态分配内存，而 malloc 函数从堆上动态分配内存。（自由存储区不等于堆）

malloc 的原理？malloc 的底层实现?

malloc 的原理:

- 当开辟的空间小于 128K 时，调用 brk() 函数，通过移动 _enddata 来实现；
- 当开辟空间大于 128K 时，调用 mmap() 函数，通过在虚拟地址空间中开辟一块内存空间来实现。

malloc 的底层实现：

- brk() 函数实现原理：向高地址的方向移动指向数据段的高地址的指针 _enddata。
- mmap 内存映射原理：
  1.进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域；
  2.调用内核空间的系统调用函数 mmap()，实现文件物理地址和进程虚拟地址的一一映射关系；
  3.进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝。

C 和 C++ struct 的区别?

- 在 C 语言中 struct 是用户自定义数据类型；在 C++ 中 struct 是抽象数据类型，支持成员函数的定义。
- C 语言中 struct 没有访问权限的设置，是一些变量的集合体，不能定义成员函数；C++ 中 struct 可以和类一样，有访问权限，并可以定义成员函数。
- C 语言中 struct 定义的自定义数据类型，在定义该类型的变量时，需要加上 struct 关键字，例如：struct A var;，定义 A 类型的变量；而 C++ 中，不用加该关键字，例如：A var;

为什么有了 class 还保留 struct?

- C++ 是在 C 语言的基础上发展起来的，为了与 C 语言兼容，C++ 中保留了 `struct`。

struct 和 union 的区别

说明：union 是联合体，struct 是结构体。
区别：

- 联合体和结构体都是由若干个数据类型不同的数据成员组成。使用时，联合体只有一个有效的成员；而结构体所有的成员都有效。
- 对联合体的不同成员赋值，将会对覆盖其他成员的值，而对于结构体的对不同成员赋值时，相互不影响。
- 联合体的大小为其内部所有变量的最大值，按照最大类型的倍数进行分配大小；结构体分配内存的大小遵循内存对齐原则。

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

typedef union
{
    char c[10];
    char cc1; // char 1 字节，按该类型的倍数分配大小
} u11;

typedef union
{
    char c[10];
    int i; // int 4 字节，按该类型的倍数分配大小
} u22;

typedef union
{
    char c[10];
    double d; // double 8 字节，按该类型的倍数分配大小
} u33;

typedef struct s1
{
    char c;   // 1 字节
    double d; // 1（char）+ 7（内存对齐）+ 8（double）= 16 字节
} s11;

typedef struct s2
{
    char c;   // 1 字节
    char cc;  // 1（char）+ 1（char）= 2 字节
    double d; // 2 + 6（内存对齐）+ 8（double）= 16 字节
} s22;

typedef struct s3
{
    char c;   // 1 字节
    double d; // 1（char）+ 7（内存对齐）+ 8（double）= 16 字节
    char cc;  // 16 + 1（char）+ 7（内存对齐）= 24 字节
} s33;

int main()
{
    cout << sizeof(u11) << endl; // 10
    cout << sizeof(u22) << endl; // 12
    cout << sizeof(u33) << endl; // 16
    cout << sizeof(s11) << endl; // 16
    cout << sizeof(s22) << endl; // 16
    cout << sizeof(s33) << endl; // 24
```

```
    cout << sizeof(int) << endl;    // 4
    cout << sizeof(double) << endl; // 8
    return 0;
}
```

class 和 struct 的异同

- struct 和 class 都可以自定义数据类型，也支持继承操作。
- struct 中默认的访问级别是 public，默认的继承级别也是 public；class 中默认的访问级别是 private，默认的继承级别也是 private。
- 当 class 继承 struct 或者 struct 继承 class 时，默认的继承级别取决于 class 或 struct 本身， class（private 继承），struct（public 继承），即取决于派生类的默认继承级别。

```
struct A{};
class B : A{}; // private 继承
struct C : B{};  // public 继承
```

举例：

```
#include<iostream>

using namespace std;

class A{
public:
    void funA(){
        cout << "class A" << endl;
    }
};

struct B: A{ // 由于 B 是 struct，A 的默认继承级别为 public
public:
    void funB(){
        cout << "class B" << endl;
    }
};

class C: B{ // 由于 C 是 class，B 的默认继承级别为 private，所以无法访问基类 B 中的 printB 函数

};

int main(){
    A ex1;
    ex1.funA(); // class A

    B ex2;
```

```
    ex2.funA(); // class A
    ex2.funB(); // class B

    C ex3;
    ex3.funB(); // error: 'B' is not an accessible base of 'C'.
    return 0;

}
```

class 可以用于定义模板参数，struct 不能用于定义模板参数。

volatile 的作用？是否具有原子性，对编译器有什么影响？

volatile 的作用：当对象的值可能在程序的控制或检测之外被改变时，应该将该对象声明为 violatile，告知编译器不应对这样的对象进行优化。

volatile不具有原子性。

volatile 对编译器的影响：使用该关键字后，编译器不会对相应的对象进行优化，即不会将变量从内存缓存到寄存器中，防止多个线程有可能使用内存中的变量，有可能使用寄存器中的变量，从而导致程序错误。

什么情况下一定要用 volatile， 能否和 const 一起使用？

使用 volatile 关键字的场景：

- 当多个线程都会用到某一变量，并且该变量的值有可能发生改变时，需要用 volatile 关键字对该变量进行修饰；
- 中断服务程序中访问的变量或并行设备的硬件寄存器的变量，最好用 volatile 关键字修饰。

volatile 关键字和 const 关键字可以同时使用，某种类型可以既是 volatile 又是 const ，同时具有二者的属性。

返回函数中静态变量的地址会发生什么？

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

int * fun(int tmp){
    static int var = 10;
    var *= tmp;
    return &var;
}

int main() {
    cout << *fun(5) << endl;
    return 0;
}
```

```
/*
运行结果：
50
*/
```

说明：上述代码中在函数 fun 中定义了静态局部变量 var，使得离开该函数的作用域后，该变量不会销毁，返回到主函数中，该变量依然存在，从而使程序得到正确的运行结果。但是，该静态局部变量直到程序运行结束后才销毁，浪费内存空间。

extern C 的作用?

当 C++ 程序 需要调用 C 语言编写的函数，C++ 使用链接指示，即 extern "C" 指出任意非 C++ 函数所用的语言。

举例：

```
// 可能出现在 C++ 头文件<cstring>中的链接指示
extern "C"{
    int strcmp(const char*, const char*);
}
```

sizeof(1==1) 在 C 和 C++ 中分别是什么结果?

c语言

```
#include<stdio.h>

void main(){
    printf("%d\n", sizeof(1==1));
}

/*
运行结果：
4
*/
```

C++

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

int main() {
    cout << sizeof(1==1) << endl;
    return 0;
}

/*
1
*/
```

memcpy 函数的底层原理?

```
void *memcpy(void *dst, const void *src, size_t size)
{
    char *psrc;
    char *pdst;

    if (NULL == dst || NULL == src)
    {
        return NULL;
    }

    if ((src < dst) && (char *)src + size > (char *)dst) // 出现地址重叠的
情况，自后向前拷贝
    {
        psrc = (char *)src + size - 1;
        pdst = (char *)dst + size - 1;
        while (size--)
        {
            *pdst-- = *psrc--;
        }
    }
    else
    {
        psrc = (char *)src;
        pdst = (char *)dst;
        while (size--)
        {
            *pdst++ = *psrc++;
        }
    }

    return dst;
}
```

strcpy 函数有什么缺陷?

strcpy 函数的缺陷：strcpy 函数不检查目的缓冲区的大小边界，而是将源字符串逐一的全部赋值给目的字符串地址起始的一块连续的内存空间，同时加上字符串终止符，会导致其他变量被覆盖。

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

int main()
{
    int var = 0x11112222;
    char arr[10];
    cout << "Address : var " << &var << endl;
    cout << "Address : arr " << &arr << endl;
    strcpy(arr, "hello world!");
    cout << "var:" << hex << var << endl; // 将变量 var 以 16 进制输出
    cout << "arr:" << arr << endl;
    return 0;
}

/*
Address : var 0x23fe4c
Address : arr 0x23fe42
var:11002164
arr:hello world!
*/
```

说明：从上述代码中可以看出，变量 var 的后六位被字符串 "hello world!" 的 "d!\0" 这三个字符改变，这三个字符对应的 ascii 码的十六进制为：\0(0x00)，!(0x21)，d(0x64)。
原因：变量 arr 只分配的 10 个内存空间，通过上述程序中的地址可以看出 arr 和 var 在内存中是连续存放的，但是在调用 strcpy 函数进行拷贝时，源字符串 "hello world!" 所占的内存空间为 13，因此在拷贝的过程中会占用 var 的内存空间，导致 var的后六位被覆盖。

auto 类型推导的原理

编译器根据初始值来推算变量的类型，要求用 auto 定义变量时必须有初始值。编译器推断出来的 auto 类型有时和初始值类型并不完全一样，编译器会适当改变结果类型使其更符合初始化规则。

类相关

什么是虚函数？什么是纯虚函数？

虚函数：被 virtual 关键字修饰的成员函数，就是虚函数。

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

class A
{
public:
    virtual void v_fun() // 虚函数
    {
        cout << "A::v_fun()" << endl;
    }
};
class B : public A
{
public:
    void v_fun()
    {
        cout << "B::v_fun()" << endl;
    }
};
int main()
{
    A *p = new B();
    p->v_fun(); // B::v_fun()
    return 0;
}
```

纯虚函数：

- 纯虚函数在类中声明时，加上 =0；
- 含有纯虚函数的类称为抽象类（只要含有纯虚函数这个类就是抽象类），类中只有接口，没有具体的实现方法；
- 继承纯虚函数的派生类，如果没有完全实现基类纯虚函数，依然是抽象类，不能实例化对象。

说明：

- 抽象类对象不能作为函数的参数，不能创建对象，不能作为函数返回类型；
- 可以声明抽象类指针，可以声明抽象类的引用；
- 子类必须继承父类的纯虚函数，并全部实现后，才能创建子类的对象。

虚函数和纯虚函数的区别？

- 虚函数和纯虚函数可以出现在同一个类中，该类称为抽象基类。（含有纯虚函数的类称为抽象基类）
- 使用方式不同：虚函数可以直接使用，纯虚函数必须在派生类中实现后才能使用；

- 定义形式不同：虚函数在定义时在普通函数的基础上加上 virtual 关键字，纯虚函数定义时除了加上virtual 关键字还需要加上 =0;
- 虚函数必须实现，否则编译器会报错；
- 对于实现纯虚函数的派生类，该纯虚函数在派生类中被称为虚函数，虚函数和纯虚函数都可以在派生类中重写；
- 析构函数最好定义为虚函数，特别是对于含有继承关系的类；析构函数可以定义为纯虚函数，此时，其所在的类为抽象基类，不能创建实例化对象。

虚函数的实现机制

实现机制：虚函数通过虚函数表来实现。虚函数的地址保存在虚函数表中，在类的对象所在的内存空间中，保存了指向虚函数表的指针（称为“虚表指针”），通过虚表指针可以找到类对应的虚函数表。虚函数表解决了基类和派生类的继承问题和类中成员函数的覆盖问题，当用基类的指针来操作一个派生类的时候，这张虚函数表就指明了实际应该调用的函数。

虚函数表相关知识点：

- 虚函数表存放的内容：类的虚函数的地址。
- 虚函数表建立的时间：编译阶段，即程序的编译过程中会将虚函数的地址放在虚函数表中。
- 虚表指针保存的位置：虚表指针存放在对象的内存空间中最前面的位置，这是为了保证正确取到虚函数的偏移量。

注：虚函数表和类绑定，虚表指针和对象绑定。即类的不同的对象的虚函数表是一样的，但是每个对象都有自己的虚表指针，来指向类的虚函数表。

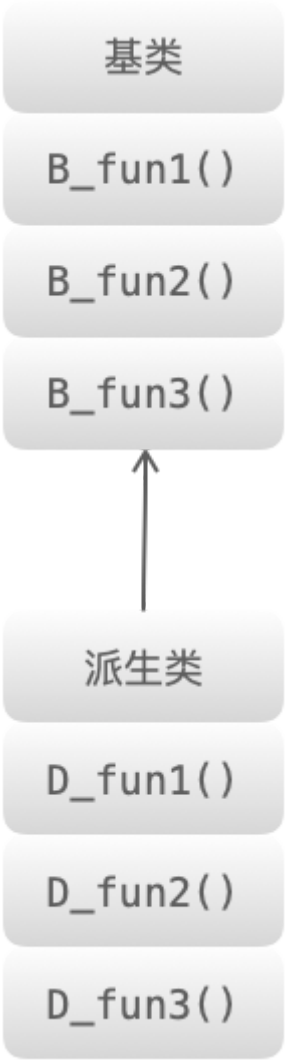
实例：

无虚函数覆盖的情况：

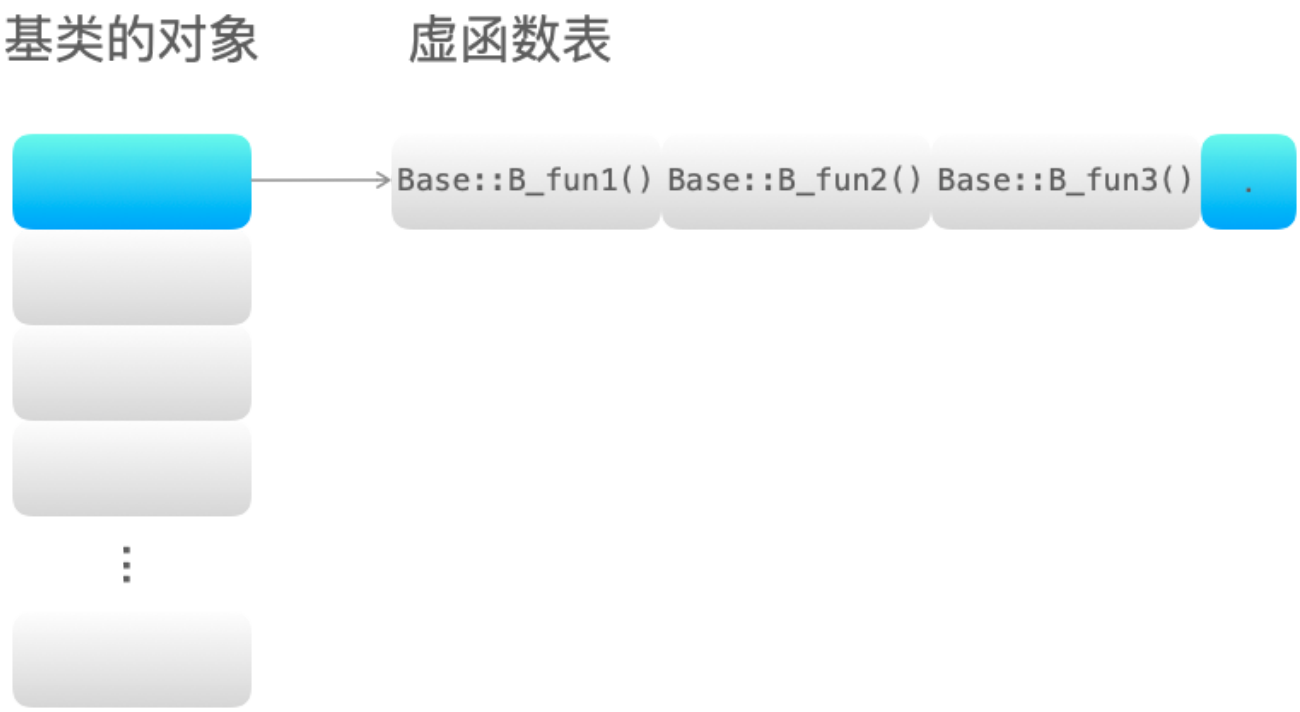
```
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void B_fun1() { cout << "Base::B_fun1()" << endl; }
    virtual void B_fun2() { cout << "Base::B_fun2()" << endl; }
    virtual void B_fun3() { cout << "Base::B_fun3()" << endl; }
};

class Derive : public Base
{
public:
    virtual void D_fun1() { cout << "Derive::D_fun1()" << endl; }
    virtual void D_fun2() { cout << "Derive::D_fun2()" << endl; }
    virtual void D_fun3() { cout << "Derive::D_fun3()" << endl; }
};
int main()
{
```

```
    Base *p = new Derive();
    p->B_fun1(); // Base::B_fun1()
    return 0;
}
```

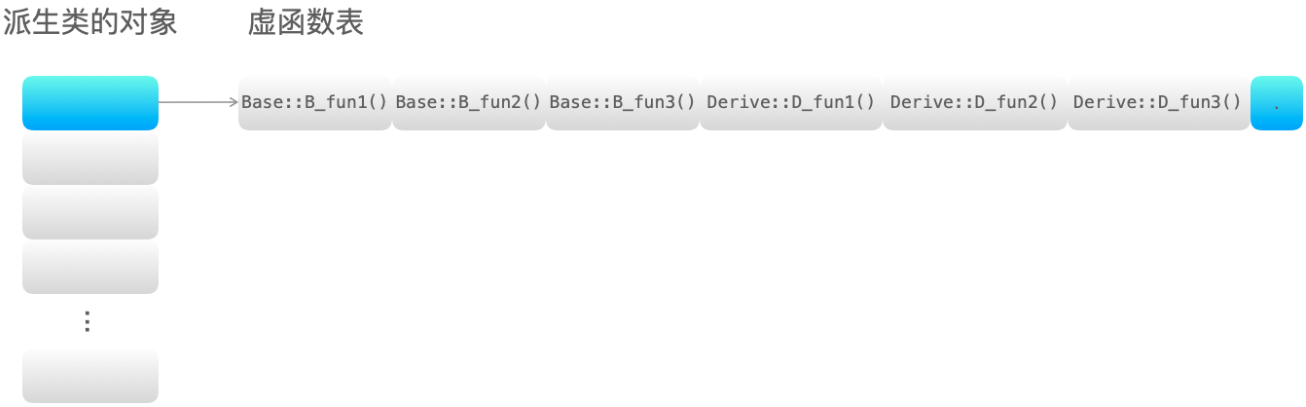
基类和派生类的继承关系：



基类的虚函数表：



派生类的虚函数表：



主函数中基类的指针 `p` 指向了派生类的对象，当调用函数 `B_fun1()` 时，通过派生类的虚函数表找到该函数的地址，从而完成调用。

单继承和多继承的虚函数表结构

编译器处理虚函数表：

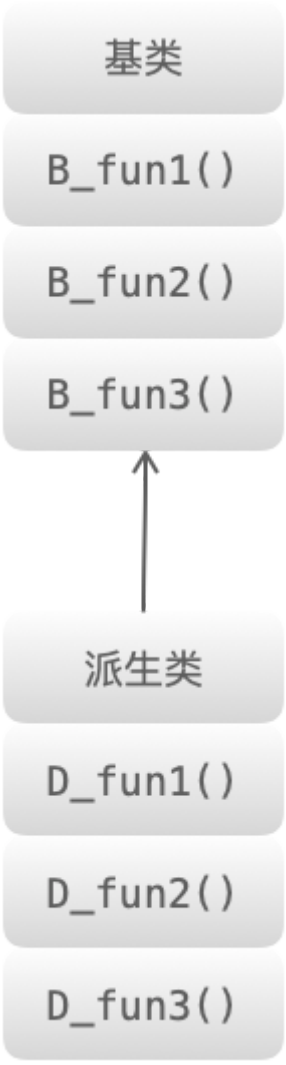
- 编译器将虚函数表的指针放在类的实例对象的内存空间中，该对象调用该类的虚函数时，通过指针找到虚函数表，根据虚函数表中存放的虚函数的地址找到对应的虚函数。
- 如果派生类没有重新定义基类的虚函数 A，则派生类的虚函数表中保存的是基类的虚函数 A 的地址，也就是说基类和派生类的虚函数 A 的地址是一样的。
- 如果派生类重写了基类的某个虚函数 B，则派生的虚函数表中保存的是重写后的虚函数 B 的地址，也就是说虚函数 B 有两个版本，分别存放在基类和派生类的虚函数表中。
- 如果派生类重新定义了新的虚函数 C，派生类的虚函数表保存新的虚函数 C 的地址。

单继承无虚函数覆盖的情况：

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

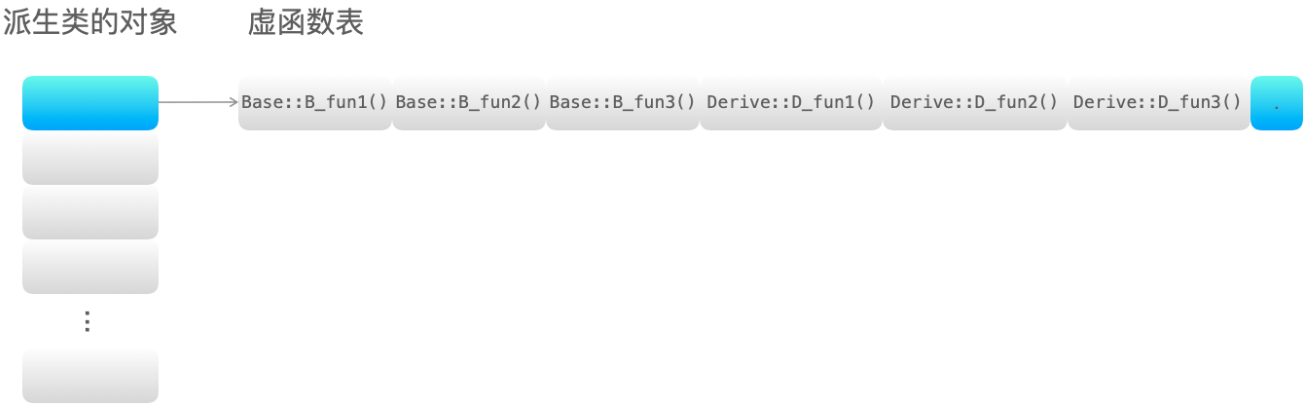
```
class Base
{
public:
    virtual void B_fun1() { cout << "Base::B_fun1()" << endl; }
    virtual void B_fun2() { cout << "Base::B_fun2()" << endl; }
    virtual void B_fun3() { cout << "Base::B_fun3()" << endl; }
};

class Derive : public Base
{
public:
    virtual void D_fun1() { cout << "Derive::D_fun1()" << endl; }
    virtual void D_fun2() { cout << "Derive::D_fun2()" << endl; }
    virtual void D_fun3() { cout << "Derive::D_fun3()" << endl; }
};
int main()
{
    Base *p = new Derive();
    p->B_fun1(); // Base::B_fun1()
    return 0;
}
```

基类和派生类的继承关系：



基类的虚函数表：

派生类的虚函数表：

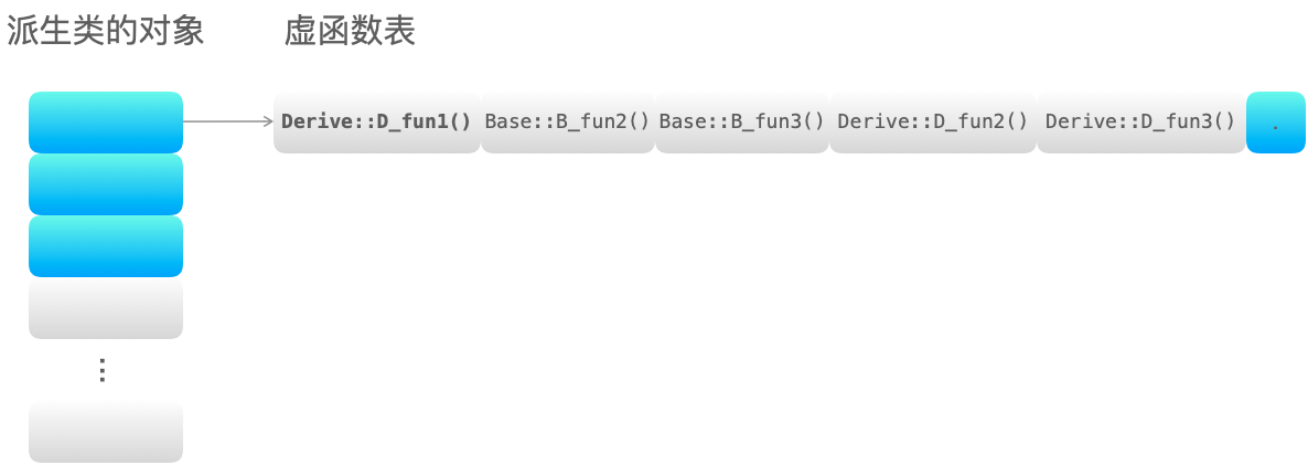


单继承有虚函数覆盖的情况：

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

class Base
{
public:
    virtual void fun1() { cout << "Base::fun1()" << endl; }
    virtual void B_fun2() { cout << "Base::B_fun2()" << endl; }
    virtual void B_fun3() { cout << "Base::B_fun3()" << endl; }
};

class Derive : public Base
{
public:
    virtual void fun1() { cout << "Derive::fun1()" << endl; }
    virtual void D_fun2() { cout << "Derive::D_fun2()" << endl; }
```

```
    virtual void D_fun3() { cout << "Derive::D_fun3()" << endl; }
};
int main()
{
    Base *p = new Derive();
    p->fun1(); // Derive::fun1()
    return 0;
}
```

派生类的虚函数表：

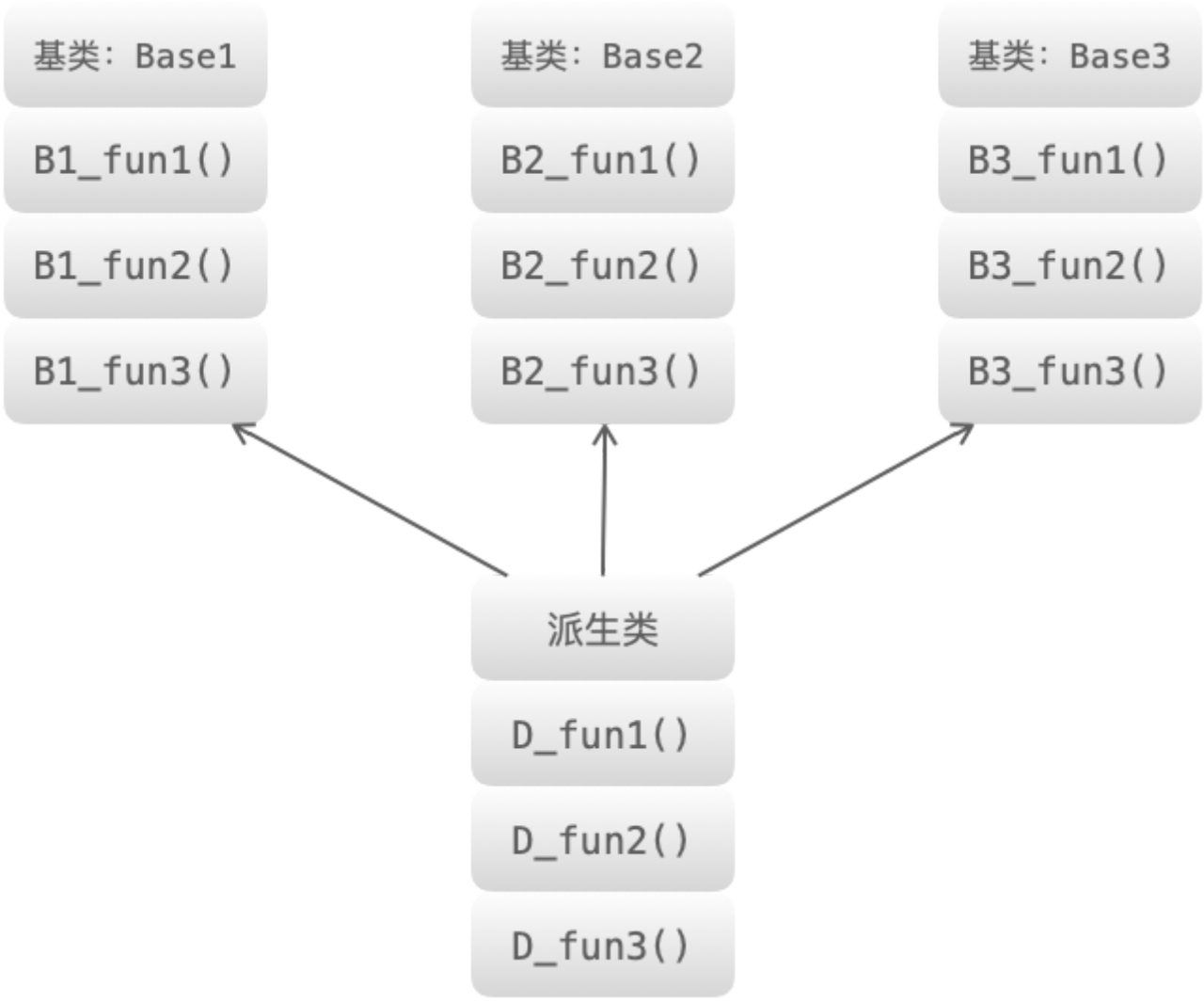


多继承无虚函数覆盖的情况：

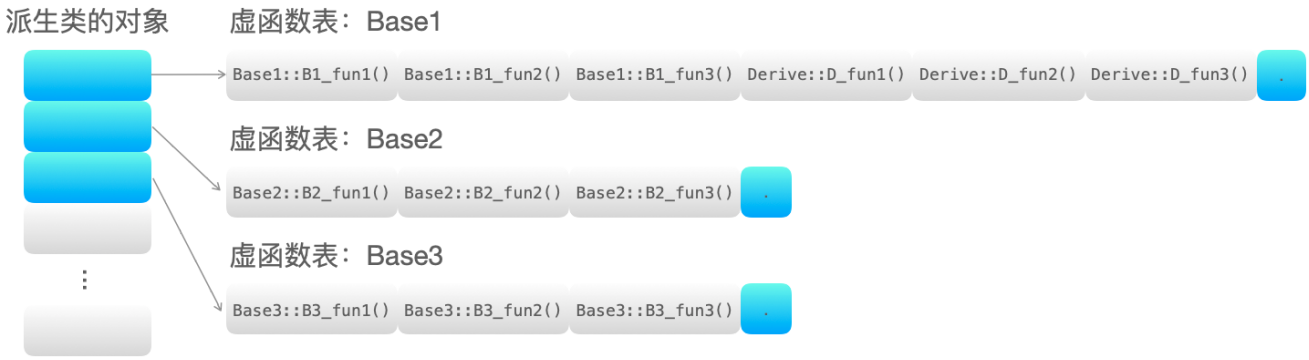
```
#include <iostream>
using namespace std;

class Base1
{
public:
    virtual void B1_fun1() { cout << "Base1::B1_fun1()" << endl; }
    virtual void B1_fun2() { cout << "Base1::B1_fun2()" << endl; }
    virtual void B1_fun3() { cout << "Base1::B1_fun3()" << endl; }
};
class Base2
{
public:
    virtual void B2_fun1() { cout << "Base2::B2_fun1()" << endl; }
    virtual void B2_fun2() { cout << "Base2::B2_fun2()" << endl; }
    virtual void B2_fun3() { cout << "Base2::B2_fun3()" << endl; }
};
class Base3
{
public:
    virtual void B3_fun1() { cout << "Base3::B3_fun1()" << endl; }
    virtual void B3_fun2() { cout << "Base3::B3_fun2()" << endl; }
    virtual void B3_fun3() { cout << "Base3::B3_fun3()" << endl; }
};
```

```
class Derive : public Base1, public Base2, public Base3
{
public:
    virtual void D_fun1() { cout << "Derive::D_fun1()" << endl; }
    virtual void D_fun2() { cout << "Derive::D_fun2()" << endl; }
    virtual void D_fun3() { cout << "Derive::D_fun3()" << endl; }
};

int main(){
    Base1 *p = new Derive();
    p->B1_fun1(); // Base1::B1_fun1()
    return 0;
}
```

基类和派生类的关系：



派生类的虚函数表：（基类的顺序和声明的顺序一致）

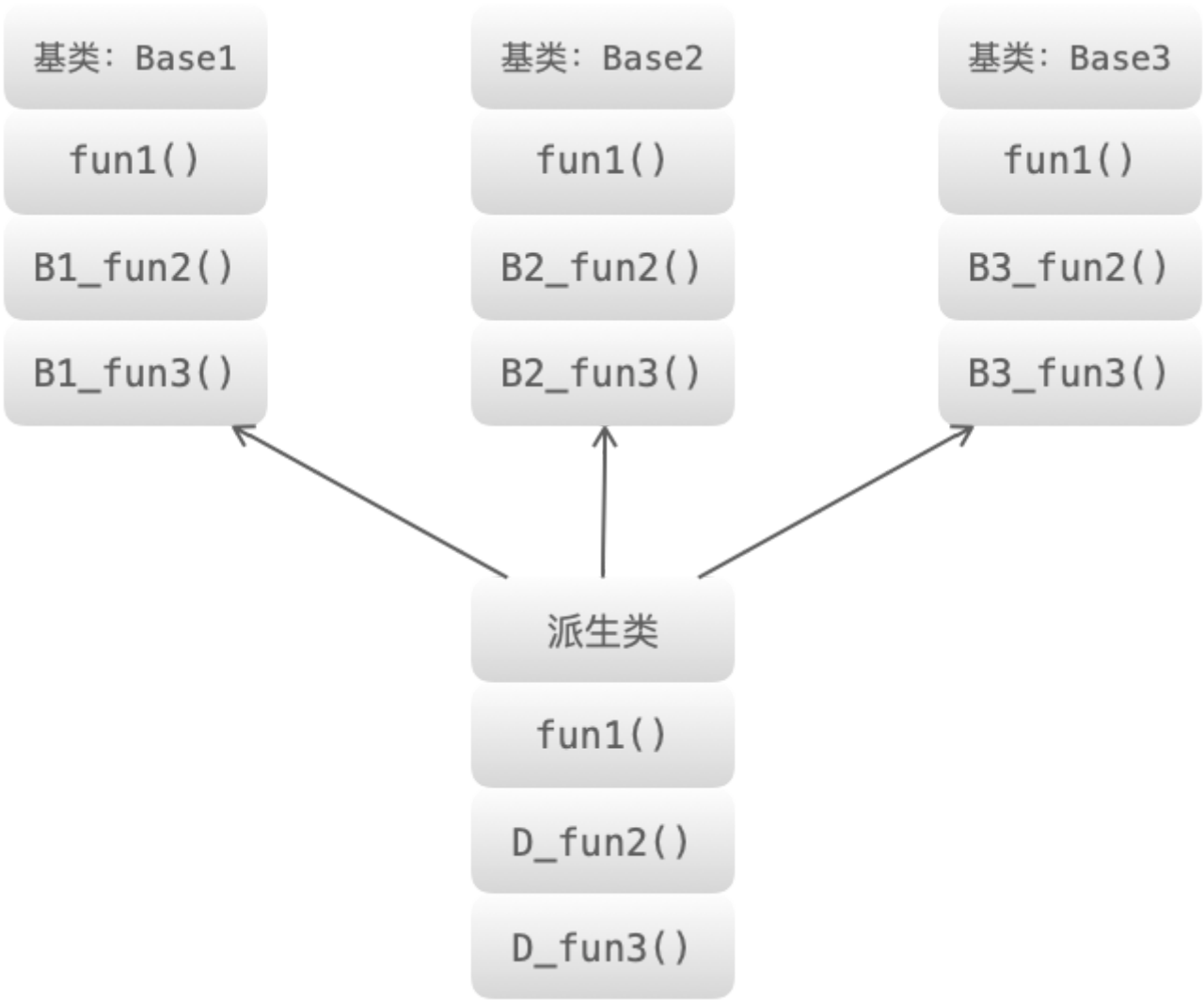


多继承有虚函数覆盖的情况：

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

class Base1
{
public:
    virtual void fun1() { cout << "Base1::fun1()" << endl; }
    virtual void B1_fun2() { cout << "Base1::B1_fun2()" << endl; }
    virtual void B1_fun3() { cout << "Base1::B1_fun3()" << endl; }
};
class Base2
{
public:
    virtual void fun1() { cout << "Base2::fun1()" << endl; }
    virtual void B2_fun2() { cout << "Base2::B2_fun2()" << endl; }
    virtual void B2_fun3() { cout << "Base2::B2_fun3()" << endl; }
};
class Base3
{
public:
    virtual void fun1() { cout << "Base3::fun1()" << endl; }
    virtual void B3_fun2() { cout << "Base3::B3_fun2()" << endl; }
    virtual void B3_fun3() { cout << "Base3::B3_fun3()" << endl; }
};

class Derive : public Base1, public Base2, public Base3
{
public:
    virtual void fun1() { cout << "Derive::fun1()" << endl; }
    virtual void D_fun2() { cout << "Derive::D_fun2()" << endl; }
    virtual void D_fun3() { cout << "Derive::D_fun3()" << endl; }
};

int main(){
    Base1 *p1 = new Derive();
    Base2 *p2 = new Derive();
    Base3 *p3 = new Derive();
    p1->fun1(); // Derive::fun1()
```

```
    p2->fun1(); // Derive::fun1()
    p3->fun1(); // Derive::fun1()
    return 0;
}
```

基类和派生类的关系：



派生类的虚函数表：

如何禁止构造函数的使用?

为类的构造函数增加 = delete 修饰符，可以达到虽然声明了构造函数但禁止使用的目的。

```
#include <iostream>

using namespace std;

class A {
public:
    int var1, var2;
    A(){
        var1 = 10;
        var2 = 20;
    }
    A(int tmp1, int tmp2) = delete;
};

int main()
{
    A ex1;
    A ex2(12,13); // error: use of deleted function 'A::A(int, int)'
    return 0;
}
```

说明：上述代码中，使用了已经删除 delete 的构造函数，程序出现错误

什么是类的默认构造函数?

默认构造函数：未提供任何实参，来控制默认初始化过程的构造函数称为默认构造函数。

```
#include <iostream>

using namespace std;

class A
{
public:
    A(){ // 类的默认构造函数
        var = 10;
        c = 'q';
    }
    int var;
    char c;
};

int main()
{
    A ex;
```

```
    cout << ex.c << endl << ex.var << endl;
    return 0;
}
/*
运行结果：
q
10
*/
```

说明：上述程序中定义变量 ex 时，未提供任何实参，程序运行时会调用默认的构造函数。

构造函数、析构函数是否需要定义成虚函数？为什么？

构造函数一般不定义为虚函数，原因：

- 从存储空间的角度考虑：构造函数是在实例化对象的时候进行调用，如果此时将构造函数定义成虚函数，需要通过访问该对象所在的内存空间才能进行虚函数的调用（因为需要通过指向虚函数表的指针调用虚函数表，虽然虚函数表在编译时就有了，但是没有虚函数的指针，虚函数的指针只有在创建了对象才有），但是此时该对象还未创建，便无法进行虚函数的调用。所以构造函数不能定义成虚函数。
- 从使用的角度考虑：虚函数是基类的指针指向派生类的对象时，通过该指针实现对派生类的虚函数的调用，构造函数是在创建对象时自动调用的。
- 从实现上考虑：虚函数表是在创建对象之后才有的，因此不能定义成虚函数。
- 从类型上考虑：在创建对象时需要明确其类型。

析构函数一般定义成虚函数，原因：
析构函数定义成虚函数是为了防止内存泄漏，因为当基类的指针或者引用指向或绑定到派生类的对象时，如果未将基类的析构函数定义成虚函数，会调用基类的析构函数，那么只能将基类的成员所占的空间释放掉，派生类中特有的就会无法释放内存空间导致内存泄漏。

如何避免拷贝？

最直观的想法是：将类的拷贝构造函数和赋值构造函数声明为私有 private，但对于类的成员函数和友元函数依然可以调用，达不到完全禁止类的对象被拷贝的目的，而且程序会出现错误，因为未对函数进行定义。

解决方法：声明一个基类，具体做法如下。

- 定义一个基类，将其中的拷贝构造函数和赋值构造函数声明为私有 private
- 派生类以私有 private 的方式继承基类

```
class Uncopyable
{
public:
    Uncopyable() {}
    ~Uncopyable() {}

private:
    Uncopyable(const Uncopyable &);             // 拷贝构造函数
    Uncopyable &operator=(const Uncopyable &); // 赋值构造函数
};
class A : private Uncopyable // 注意继承方式
{
};
```

简单解释：

能够保证，在派生类 A 的成员函数和友元函数中无法进行拷贝操作，因为无法调用基类 Uncopyable 的拷贝构造函数或赋值构造函数。同样，在类的外部也无法进行拷贝操作。

如何减少构造函数开销?

在构造函数中使用类初始化列表，会减少调用默认的构造函数产生的开销，具体原因可以参考本章“为什么用成员初始化列表会快些？”这个问题。

```
class A
{
private:
    int val;
public:
    A()
    {
        cout << "A()" << endl;
    }
    A(int tmp)
    {
        val = tmp;
        cout << "A(int " << val << ")" << endl;
    }
};
class Test1
{
private:
    A ex;

public:
    Test1() : ex(1) // 成员列表初始化方式
    {
    }
}
```

多重继承时会出现什么状况？如何解决？

多重继承（多继承）：是指从多个直接基类中产生派生类。

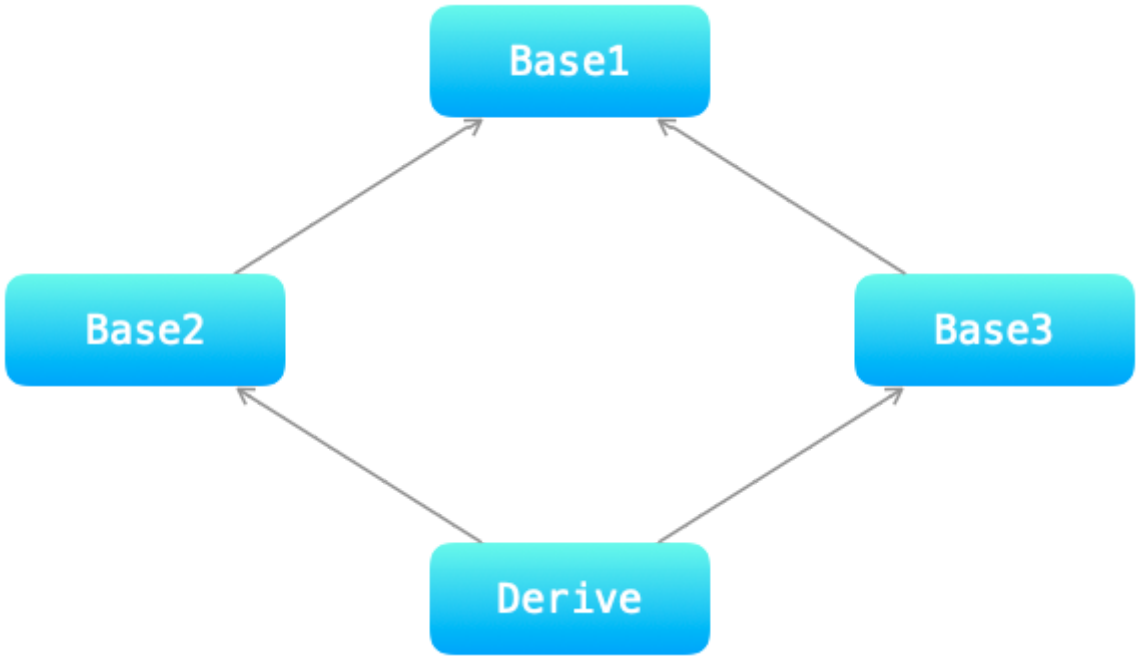
多重继承容易出现的问题：命名冲突和数据冗余问题。

举例:

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

// 间接基类
class Base1
{
public:
    int var1;
};

// 直接基类
class Base2 : public Base1
{
public:
    int var2;
};

// 直接基类
class Base3 : public Base1
{
public:
    int var3;
};

// 派生类
class Derive : public Base2, public Base3
{
public:
    void set_var1(int tmp) { var1 = tmp; } // error: reference to 'var1'
is ambiguous. 命名冲突
    void set_var2(int tmp) { var2 = tmp; }
    void set_var3(int tmp) { var3 = tmp; }
    void set_var4(int tmp) { var4 = tmp; }

private:
    int var4;
};

int main()
{
    Derive d;
    return 0;
```

```
}
```

上述程序的继承关系如下：（菱形继承）



上述代码中存的问题：
对于派生类 Derive 上述代码中存在直接继承关系和间接继承关系。

- 直接继承：Base2 、Base3
- 间接继承：Base1

对于派生类中继承的的成员变量 var1，从继承关系来看，实际上保存了两份，一份是来自基类 Base2，一份来自基类 Base3。因此，出现了命名冲突。

解决方法 1： 声明出现冲突的成员变量来源于哪个类

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

// 间接基类
class Base1
{
public:
    int var1;
};

// 直接基类
class Base2 : public Base1
{
public:
    int var2;
};

// 直接基类
```

```
class Base3 : public Base1
{
public:
    int var3;
};

// 派生类
class Derive : public Base2, public Base3
{
public:
    void set_var1(int tmp) { Base2::var1 = tmp; } // 这里声明成员变量来源于
类 Base2，当然也可以声明来源于类 Base3
    void set_var2(int tmp) { var2 = tmp; }
    void set_var3(int tmp) { var3 = tmp; }
    void set_var4(int tmp) { var4 = tmp; }

private:
    int var4;
};

int main()
{
    Derive d;
    return 0;
}
```

解决方法 2： 虚继承

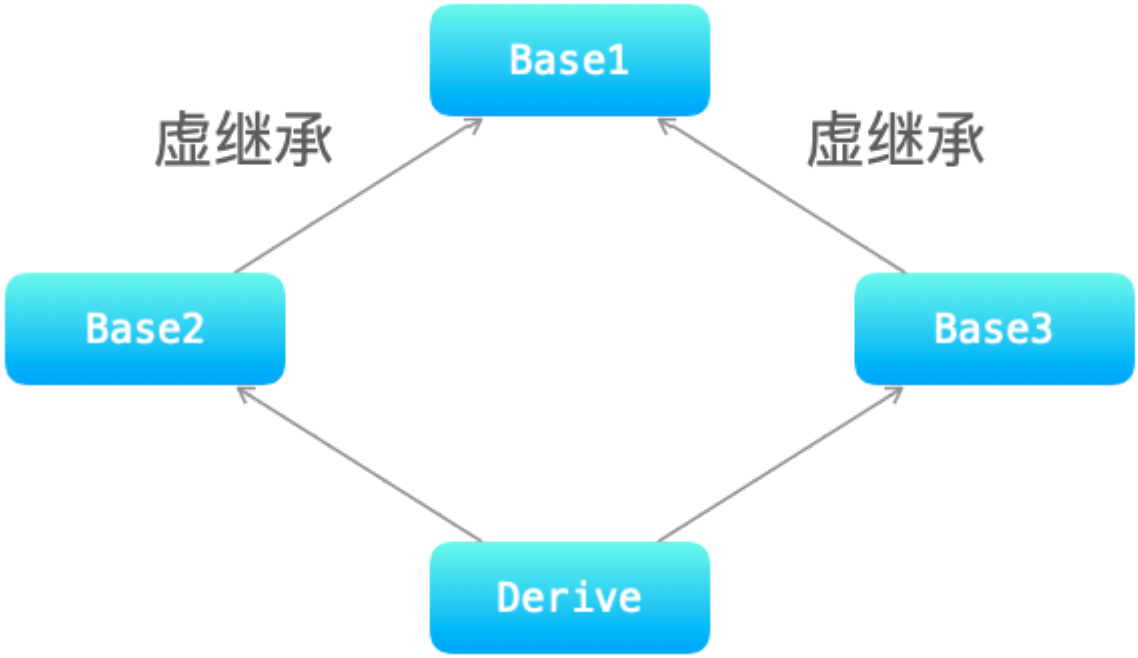
使用虚继承的目的：保证存在命名冲突的成员变量在派生类中只保留一份，即使间接基类中的成员在派生类中只保留一份。在菱形继承关系中，间接基类称为虚基类，直接基类和间接基类之间的继承关系称为虚继承。

实现方式：在继承方式前面加上 virtual 关键字。

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

// 间接基类，即虚基类
class Base1
{
public:
    int var1;
};

// 直接基类
class Base2 : virtual public Base1 // 虚继承
{
public:
```

```
    int var2;
};

// 直接基类
class Base3 : virtual public Base1 // 虚继承
{
public:
    int var3;
};

// 派生类
class Derive : public Base2, public Base3
{
public:
    void set_var1(int tmp) { var1 = tmp; }
    void set_var2(int tmp) { var2 = tmp; }
    void set_var3(int tmp) { var3 = tmp; }
    void set_var4(int tmp) { var4 = tmp; }

private:
    int var4;
};

int main()
{
    Derive d;
    return 0;
}
```

类之间的继承关系：



空类占多少字节？C++ 编译器会给一个空类自动生成哪些函数？

空类声明时编译器不会生成任何成员函数：

对于空类，声明编译器不会生成任何的成员函数，只会生成 1 个字节的占位符。

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

class A
{
};

int main()
{
    cout << "sizeof(A):" << sizeof(A) << endl; // sizeof(A):1
    return 0;
}
```

空类定义时编译器会生成 6 个成员函数：

当空类 A 定义对象时，sizeof(A) 仍是为 1，但编译器会生成 6 个成员函数：缺省的构造函数、拷贝构造函数、析构函数、赋值运算符、两个取址运算符。

```
#include <iostream>
using namespace std;
/*
class A
{}; 该空类的等价写法如下：
*/
class A
{
public:
    A(){};                                          // 缺省构造函数
    A(const A &tmp){};                              // 拷贝构造函数
    ~A(){};                                         // 析构函数
    A &operator=(const A &tmp){};                   // 赋值运算符
    A *operator&() { return this; };                // 取址运算符
    const A *operator&() const { return this; }; // 取址运算符（const 版
本）
};

int main()
{
    A *p = new A();
    cout << "sizeof(A):" << sizeof(A) << endl; // sizeof(A):1
    delete p;
    return 0;
}
```

为什么拷贝构造函数必须为引用?

原因：避免拷贝构造函数无限制的递归，最终导致栈溢出。

举例说明：

C++

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

class A
{
private:
    int val;

public:
    A(int tmp) : val(tmp) // 带参数构造函数
    {
        cout << "A(int tmp)" << endl;
    }

    A(const A &tmp) // 拷贝构造函数
    {
        cout << "A(const A &tmp)" << endl;
        val = tmp.val;
    }

    A &operator=(const A &tmp) // 赋值函数（赋值运算符重载）
    {
        cout << "A &operator=(const A &tmp)" << endl;
        val = tmp.val;
        return *this;
    }

    void fun(A tmp)
    {
    }

};

int main()
{
    A ex1(1);
    A ex2(2);
    A ex3 = ex1;
    ex2 = ex1;
    ex2.fun(ex1);
    return 0;
}
```

```
/*
运行结果：
A(int tmp)
A(int tmp)
A(const A &tmp)
A &operator=(const A &tmp)
A(const A &tmp)
*/
```

- 说明 1：ex2 = ex1; 和 A ex3 = ex1; 为什么调用的函数不一样？
  对象 ex2 已经实例化了，不需要构造，此时只是将 ex1 赋值给 ex2，只会调用赋值函数；但是 ex3 还没有实例化，因此调用的是拷贝构造函数，构造出 ex3，而不是赋值函数，这里涉及到构造函数的隐式调用。
- 说明 2：如果拷贝构造函数中形参不是引用类型，A ex3 = ex1;会出现什么问题？
  构造 ex3，实质上是 ex3.A(ex1);，假如拷贝构造函数参数不是引用类型，那么将使得 ex3.A(ex1); 相当于 ex1 作为函数 A(const A tmp)的形参，在参数传递时相当于 A tmp = ex1，因为 tmp 没有被初始化，所以在 A tmp = ex1 中继续调用拷贝构造函数，接下来的是构造 tmp，也就是 tmp.A(ex1) ，必然又会有 ex1 作为函数 A(const A tmp); 的形参，在参数传递时相当于即 A tmp = ex1，那么又会触发拷贝构造函数，就这下永远的递归下去。
- 说明 3：为什么 ex2.fun(ex1); 会调用拷贝构造函数？
  ex1 作为参数传递给 fun 函数， 即 A tmp = ex1;，这个过程会调用拷贝构造函数进行初始化。

C++ 类对象的初始化顺序

构造函数调用顺序：

- 按照派生类继承基类的顺序，即派生列表中声明的顺序，依次调用基类的构造函数；
- 按照派生类中成员变量的声名顺序，依次调用派生类中成员变量所属类的构造函数；
- 执行派生类自身的构造函数。

综上可以得出，类对象的初始化顺序：基类构造函数–>派生类成员变量的构造函数–>自身构造函数
注：

- 基类构造函数的调用顺序与派生类的派生列表中的顺序有关；
- 成员变量的初始化顺序与声明顺序有关；
- 析构顺序和构造顺序相反。

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

class A
{
public:
    A() { cout << "A()" << endl; }
    ~A() { cout << "~A()" << endl; }
};
```

```
class B
{
public:
    B() { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
};

class Test : public A, public B // 派生列表
{
public:
    Test() { cout << "Test()" << endl; }
    ~Test() { cout << "~Test()" << endl; }

private:
    B ex1;
    A ex2;
};

int main()
{
    Test ex;
    return 0;
}
/*
运行结果：
A()
B()
B()
A()
Test()
~Test()
~A()
~B()
~B()
~A()
*/
```

程序运行结果分析：

- 首先调用基类 A 和 B 的构造函数，按照派生列表 public A, public B 的顺序构造；
- 然后调用派生类 Test 的成员变量 ex1 和 ex2 的构造函数，按照派生类中成员变量声明的顺序构造；
- 最后调用派生类的构造函数；
- 接下来调用析构函数，和构造函数调用的顺序相反。

如何禁止一个类被实例化?

方法一：

在类中定义一个纯虚函数，使该类成为抽象基类，因为不能创建抽象基类的实例化对象；

```
#include <iostream>

using namespace std;


class A {
public:
    int var1, var2;
    A(){
        var1 = 10;
        var2 = 20;
    }
    virtual void fun() = 0; // 纯虚函数
};

int main()
{
    A ex1; // error: cannot declare variable 'ex1' to be of abstract
type 'A'
    return 0;
}
```

方法二：

将类的构造函数声明为私有 private

为什么用成员初始化列表会快一些?

说明：数据类型可分为内置类型和用户自定义类型（类类型），对于用户自定义类型，利用成员初始化列表效率高。

原因：用户自定义类型如果使用类初始化列表，直接调用该成员变量对应的构造函数即完成初始化；如果在构造函数中初始化，因为 C++ 规定，对象的成员变量的初始化动作发生在进入构造函数本体之前，那么在执行构造函数的函数体之前首先调用默认的构造函数为成员变量设初值，在进入函数体之后，调用该成员变量对应的构造函数。因此，使用列表初始化会减少调用默认的构造函数的过程，效率高。

```
#include <iostream>
using namespace std;
class A
{
private:
    int val;
public:
```

```
    A()
    {
        cout << "A()" << endl;
    }
    A(int tmp)
    {
        val = tmp;
        cout << "A(int " << val << ")" << endl;
    }
};

class Test1
{
private:
    A ex;

public:
    Test1() : ex(1) // 成员列表初始化方式
    {
    }
};

class Test2
{
private:
    A ex;

public:
    Test2() // 函数体中赋值的方式
    {
        ex = A(2);
    }
};
int main()
{
    Test1 ex1;
    cout << endl;
    Test2 ex2;
    return 0;
}
/*
运行结果:
A(int 1)

A()
A(int 2)
*/
```

说明：
从程序运行结果可以看出，使用成员列表初始化的方式会省去调用默认的构造函数的过程。

实例化一个对象需要哪几个阶段

- 分配空间
  创建类对象首先要为该对象分配内存空间。不同的对象，为其分配空间的时机未必相同。全局对象、静态对象、分配在栈区域内的对象，在编译阶段进行内存分配；存储在堆空间的对象，是在运行阶段进行内存分配。
- 初始化
  首先明确一点：初始化不同于赋值。初始化发生在赋值之前，初始化随对象的创建而进行，而赋值是在对象创建好后，为其赋上相应的值。这一点可以联想下上一个问题中提到：初始化列表先于构造函数体内的代码执行，初始化列表执行的是数据成员的初始化过程，这个可以从成员对象的构造函数被调用看的出来。
- 赋值
  对象初始化完成后，可以对其进行赋值。对于一个类的对象，其成员变量的赋值过程发生在类的构造函数的函数体中。当执行完该函数体，也就意味着类对象的实例化过程完成了。（总结：构造函数实现了对象的初始化和赋值两个过程，对象的初始化是通过初始化列表来完成，而对象的赋值则才是通过构造函数的函数体来实现。）

注：对于拥有虚函数的类的对象，还需要给虚表指针赋值。

- 没有继承关系的类，分配完内存后，首先给虚表指针赋值，然后再列表初始化以及执行构造函数的函数体，即上述中的初始化和赋值操作。
- 有继承关系的类，分配内存之后，首先进行基类的构造过程，然后给该派生类的虚表指针赋值，最后再列表初始化以及执行构造函数的函数体，即上述中的初始化和赋值操作。

友元函数的作用及使用场景

作用：友元提供了不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。通过友元，一个不同函数或另一个类中的成员函数可以访问类中的私有成员和保护成员。

使用场景：

普通函数定义为友元函数，使普通函数能够访问类的私有成员。

```
#include <iostream>

using namespace std;

class A
{
    friend ostream &operator<<(ostream &_cout, const A &tmp); // 声明为类的友元函数

public:
    A(int tmp) : var(tmp)
    {
```

```
    }

private:
    int var;
};

ostream &operator<<(ostream &_cout, const A &tmp)
{
    _cout << tmp.var;
    return _cout;
}

int main()
{
    A ex(4);
    cout << ex << endl; // 4
    return 0;
}
```

友元类：类之间共享数据。

```
#include <iostream>

using namespace std;

class A
{
    friend class B;

public:
    A() : var(10){}
    A(int tmp) : var(tmp) {}
    void fun()
    {
        cout << "fun():" << var << endl;
    }

private:
    int var;
};

class B
{
public:
    B() {}
    void fun()
    {
        cout << "fun():" << ex.var << endl; // 访问类 A 中的私有成员
    }
```

```
private:
    A ex;
};

int main()
{
    B ex;
    ex.fun(); // fun():10
    return 0;
}
```

静态绑定和动态绑定是怎么实现的?

静态类型和动态类型：

- 静态类型：变量在声明时的类型，是在编译阶段确定的。静态类型不能更改。
- 动态类型：目前所指对象的类型，是在运行阶段确定的。动态类型可以更改。

静态绑定和动态绑定：

- 静态绑定是指程序在 编译阶段 确定对象的类型（静态类型）。
- 动态绑定是指程序在 运行阶段 确定对象的类型（动态类型）。

静态绑定和动态绑定的区别：

- 发生的时期不同：如上。
- 对象的静态类型不能更改，动态类型可以更改。

注：对于类的成员函数，只有虚函数是动态绑定，其他都是静态绑定。

```
#include <iostream>

using namespace std;

class Base
{
public:
    virtual void fun() { cout << "Base::fun()" << endl;
     }
};
class Derive : public Base
{
public:
    void fun() { cout << "Derive::fun()";
    }
};

int main()
{
```

```
    Base *p = new Derive(); // p 的静态类型是 Base*，动态类型是 Derive*
    p->fun(); // fun 是虚函数，运行阶段进行动态绑定
    return 0;
}
/*
运行结果：
Derive::fun()
*/
```

深拷贝和浅拷贝的区别

如果一个类拥有资源，该类的对象进行复制时，如果资源重新分配，就是深拷贝，否则就是浅拷贝。

- 深拷贝：该对象和原对象占用不同的内存空间，既拷贝存储在栈空间中的内容，又拷贝存储在堆空间中的内容。
- 浅拷贝：该对象和原对象占用同一块内存空间，仅拷贝类中位于栈空间中的内容。

当类的成员变量中有指针变量时，最好使用深拷贝。因为当两个对象指向同一块内存空间，如果使用浅拷贝，当其中一个对象的删除后，该块内存空间就会被释放，另外一个对象指向的就是垃圾内存。

浅拷贝实例

```
#include <iostream>

using namespace std;

class Test
{
private:
    int *p;

public:
    Test(int tmp)
    {
        this->p = new int(tmp);
        cout << "Test(int tmp)" << endl;
    }
    ~Test()
    {
        if (p != NULL)
        {
            delete p;
        }
        cout << "~Test()" << endl;
    }
};
```

```
int main()
{
    Test ex1(10);
    Test ex2 = ex1;
    return 0;
}
/*
运行结果：
Test(int tmp)
~Test()
*/
```

说明：上述代码中，类对象 ex1、ex2 实际上是指向同一块内存空间，对象析构时，ex2 先将内存释放了一次，之后 析构对象 ex1 时又将这块已经被释放过的内存再释放一次。对同一块内存空间释放了两次，会导致程序崩溃。

深拷贝实例：

```
#include <iostream>

using namespace std;

class Test
{
private:
    int *p;

public:
    Test(int tmp)
    {
        p = new int(tmp);
        cout << "Test(int tmp)" << endl;
    }
    ~Test()
    {
        if (p != NULL)
        {
            delete p;
        }
        cout << "~Test()" << endl;
    }
    Test(const Test &tmp) // 定义拷贝构造函数
    {
        p = new int(*tmp.p);
        cout << "Test(const Test &tmp)" << endl;
    }

};
```

```
int main()
{
    Test ex1(10);
    Test ex2 = ex1;
    return 0;
}
/*
Test(int tmp)
Test(const Test &tmp)
~Test()
~Test()
*/
```

编译时多态和运行时多态的区别

编译时多态：在程序编译过程中出现，发生在模板和函数重载中（泛型编程）。
运行时多态：在程序运行过程中出现，发生在继承体系中，是指通过基类的指针或引用访问派生类中的虚函数。

编译时多态和运行时多态的区别：

- 时期不同：编译时多态发生在程序编译过程中，运行时多态发生在程序的运行过程中；
- 实现方式不同：编译时多态运用泛型编程来实现，运行时多态借助虚函数来实现。

实现一个类成员函数，要求不允许修改类的成员变量?

如果想达到一个类的成员函数不能修改类的成员变量，只需用 const 关键字来修饰该函数即可。
该问题本质是考察 const 关键字修饰成员函数的作用，只不过以实例的方式来考察，面试者应熟练掌握 const 关键字的作用。

```
#include <iostream>

using namespace std;

class A
{
public:
    int var1, var2;
    A()
    {
        var1 = 10;
        var2 = 20;
    }
    void fun() const // 不能在 const 修饰的成员函数中修改成员变量的值，除非该成员变量用 mutable 修饰
    {
        var1 = 100; // error: assignment of member 'A::var1' in read-only object
```

```
    }
};

int main()
{
    A ex1;
    return 0;
}
```

如何让类不能被继承?

解决方法一：借助 final 关键字，用该关键字修饰的类不能被继承。

```
#include <iostream>

using namespace std;

class Base final
{
};

class Derive: public Base{ // error: cannot derive from 'final' base
'Base' in derived type 'Derive'

};

int main()
{
    Derive ex;
    return 0;
}
```

解决方法二：借助友元、虚继承和私有构造函数来实现

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

template <typename T>
class Base{
    friend T;
private:
    Base(){
        cout << "base" << endl;
    }
    ~Base(){}
};

class B:virtual public Base<B>{    //一定注意 必须是虚继承
public:
```

```
    B(){
        cout << "B" << endl;
    }
};

class C:public B{
public:
    C(){}     // error: 'Base<T>::Base() [with T = B]' is private within
this context
};


int main(){
    B b;
    return 0;
}
```

说明：在上述代码中 B 类是不能被继承的类。
具体原因：

- 虽然 Base 类构造函数和析构函数被声明为私有 private，在 B 类中，由于 B 是 Base 的友元，因此可以访问 Base 类构造函数，从而正常创建 B 类的对象；
- B 类继承 Base 类采用虚继承的方式，创建 C 类的对象时，C 类的构造函数要负责 Base 类的构造，但是 Base 类的构造函数私有化了，C 类没有权限访问。因此，无法创建 C 类的对象， B 类是不能被继承的类。

注意：在继承体系中，友元关系不能被继承，虽然 C 类继承了 B 类，B 类是 Base 类的友元，但是 C 类和 Base 类没有友元关系。

语言特性相关

左值和右值的区别？左值引用和右值引用的区别，如何将左值转换成右值？

左值：指表达式结束后依然存在的持久对象。

右值：表达式结束就不再存在的临时对象。

左值和右值的区别：左值持久，右值短暂

右值引用和左值引用的区别：

- 左值引用不能绑定到要转换的表达式、字面常量或返回右值的表达式。右值引用恰好相反，可以绑定到这类表达式，但不能绑定到一个左值上。
- 右值引用必须绑定到右值的引用，通过 && 获得。右值引用只能绑定到一个将要销毁的对象上，因此可以自由地移动其资源。

std::move 可以将一个左值强制转化为右值，继而可以通过右值引用使用该值，以用于移动语义。

```
#include <iostream>
```

```
using namespace std;

void fun1(int& tmp)
{
  cout << "fun1(int& tmp):" << tmp << endl;
}

void fun2(int&& tmp)
{
  cout << "fun2(int&& tmp)" << tmp << endl;
}

int main()
{
  int var = 11;
  fun1(12); // error: cannot bind non-const lvalue reference of type
'int&' to an rvalue of type 'int'
  fun1(var);
  fun2(1);
}
```

std::move() 函数的实现原理

std::move() 函数原型：

```
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    return static_cast<typename remove_reference<T>::type &&>(t);
}
```

说明：引用折叠原理

- 右值传递给上述函数的形参 T&& 依然是右值，即 T&& && 相当于 T&&。
- 左值传递给上述函数的形参 T&& 依然是左值，即 T&& & 相当于 T&。

小结：通过引用折叠原理可以知道，move() 函数的形参既可以是左值也可以是右值。

remove_reference 具体实现：

```
//原始的，最通用的版本
template <typename T> struct remove_reference{
    typedef T type;  //定义 T 的类型别名为 type
};

//部分版本特例化，将用于左值引用和右值引用
template <class T> struct remove_reference<T&> //左值引用
{ typedef T type; }

template <class T> struct remove_reference<T&&> //右值引用
```

```
{ typedef T type; }

//举例如下,下列定义的a、b、c三个变量都是int类型
int i;
remove_refrence<decltype(42)>::type a;              //使用原版本，
remove_refrence<decltype(i)>::type  b;              //左值引用特例版本
remove_refrence<decltype(std::move(i))>::type  b;   //右值引用特例版本
```

举例：

C++

```
int var = 10;

转化过程：

1. std::move(var) => std::move(int&& &) => 折叠后 std::move(int&)

2. 此时：T 的类型为 int&，typename remove_reference<T>::type 为 int，这里使用 remove_reference 的左值引用的特例化版本

3. 通过 static_cast 将 int& 强制转换为 int&&

整个std::move被实例化如下
string&& move(int& t)
{
    return static_cast<int&&>(t);
}
```

总结：
std::move() 实现原理：

1. 利用引用折叠原理将右值经过 T&& 传递类型保持不变还是右值，而左值经过 T&& 变为普通的左值引用，以保证模板可以传递任意实参，且保持类型不变；
2. 然后通过 remove_refrence 移除引用，得到具体的类型 T；
3. 最后通过 static_cast<> 进行强制类型转换，返回 T&& 右值引用。

什么是指针？指针的大小及用法?

指针： 指向另外一种类型的复合类型。

指针的大小： 在 64 位计算机中，指针占 8 个字节空间。

```
#include<iostream>

using namespace std;

int main(){
    int *p = nullptr;
    cout << sizeof(p) << endl; // 8

char *p1 = nullptr;
cout << sizeof(p1) << endl; // 8
return 0;

}
```

指针的用法：

指向普通对象的指针

```
#include <iostream>

using namespace std;

class A
{
};

int main()
{
    A *p = new A();
    return 0;
}
```

指向常量对象的指针：常量指针

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

int main(void)
{
    const int c_var = 10;
    const int * p = &c_var;
    cout << *p << endl;
    return 0;
}
```

指向函数的指针：函数指针

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

```
int add(int a, int b){
    return a + b;
}

int main(void)
{
    int (*fun_p)(int, int);
    fun_p = add;
    cout << fun_p(1, 6) << endl;
    return 0;
}
```

指向对象成员的指针，包括指向对象成员函数的指针和指向对象成员变量的指针。
特别注意：定义指向成员函数的指针时，要标明指针所属的类。

```
#include <iostream>

using namespace std;

class A
{
public:
    int var1, var2;
    int add(){
        return var1 + var2;
    }
};

int main()
{
    A ex;
    ex.var1 = 3;
    ex.var2 = 4;
    int *p = &ex.var1; // 指向对象成员变量的指针
    cout << *p << endl;

int (A::*fun_p)();
fun_p = A::add; // 指向对象成员函数的指针 fun_p
cout << (ex.*fun_p)() << endl;
return 0;

}
```

this 指针：指向类的当前对象的指针常量。

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

```
class A
{
public:
    void set_name(string tmp)
    {
        this->name = tmp;
    }
    void set_age(int tmp)
    {
        this->age = age;
    }
    void set_sex(int tmp)
    {
        this->sex = tmp;
    }
    void show()
    {
        cout << "Name: " << this->name << endl;
        cout << "Age: " << this->age << endl;
        cout << "Sex: " << this->sex << endl;
    }

private:
    string name;
    int age;
    int sex;
};
```

什么是野指针和悬空指针?

悬空指针：

若指针指向一块内存空间，当这块内存空间被释放后，该指针依然指向这块内存空间，此时，称该指针为“悬空指针”。
举例：

```
void *p = malloc(size);
free(p);
```

// 此时，p 指向的内存空间已释放， p 就是悬空指针。
野指针：

“野指针”是指不确定其指向的指针，未初始化的指针为“野指针”。

```
void *p;
// 此时 p 是“野指针”。
```

C++ 11 nullptr 比 NULL 优势

- NULL：预处理变量，是一个宏，它的值是 0，定义在头文件 中，即 #define NULL 0。
- nullptr：C++ 11 中的关键字，是一种特殊类型的字面值，可以被转换成任意其他类型。

nullptr 的优势：

1. 有类型，类型是 typdef decltype(nullptr) nullptr_t;，使用 nullptr 提高代码的健壮性。
2. 函数重载：因为 NULL 本质上是 0，在函数调用过程中，若出现函数重载并且传递的实参是 NULL，可能会出现，不知和哪一个函数匹配的情况；但是传递实参 nullptr 就不会出现这种情况。

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

void fun(char const *p)
{
    cout << "fun(char const *p)" << endl;
}

void fun(int tmp)
{
    cout << "fun(int tmp)" << endl;
}

int main()
{
    fun(nullptr); // fun(char const *p)
    /*
    fun(NULL); // error: call of overloaded 'fun(NULL)' is ambiguous
    */
    return 0;
}
```

指针和引用的区别?

- 指针所指向的内存空间在程序运行过程中可以改变，而引用所绑定的对象一旦绑定就不能改变。（是否可变）
- 指针本身在内存中占有内存空间，引用相当于变量的别名，在内存中不占内存空间。（是否占内存）
- 指针可以为空，但是引用必须绑定对象。（是否可为空）
- 指针可以有多级，但是引用只能一级。（是否能为多级）

常量指针和指针常量的区别

常量指针：
常量指针本质上是个指针，只不过这个指针指向的对象是常量。
特点：const 的位置在指针声明运算符 * 的左侧。只要 const 位于 * 的左侧，无论它在类型名的左边或右边，都表示指向常量的指针。（可以这样理解，* 左侧表示指针指向的对象，该对象为常量，那么该指针为常量指针。）

```
const int * p;
int const * p;
```

注意 1：指针指向的对象不能通过这个指针来修改，也就是说常量指针可以被赋值为变量的地址，之所以叫做常量指针，是限制了通过这个指针修改变量的值。
例如：

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

int main()
{
    const int c_var = 8;
    const int *p = &c_var;
    *p = 6;            // error: assignment of read-only location '* p'
    return 0;
}
```

注意 2：虽然常量指针指向的对象不能变化，可是因为常量指针本身是一个变量，因此，可以被重新赋值。
例如：

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

int main()
{
    const int c_var1 = 8;
    const int c_var2 = 8;
    const int *p = &c_var1;
    p = &c_var2;
    return 0;
}
```

指针常量：
指针常量的本质上是个常量，只不过这个常量的值是一个指针。
特点：const 位于指针声明操作符右侧，表明该对象本身是一个常量，* 左侧表示该指针指向的类型，即以 * 为分界线，其左侧表示指针指向的类型，右侧表示指针本身的性质。

```
const int var;
int * const c_p = &var;
```

注意 1：指针常量的值是指针，这个值因为是常量，所以指针本身不能改变。

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

int main()
{
    int var, var1;
    int * const c_p = &var;
    c_p = &var1; // error: assignment of read-only variable 'c_p'
    return 0;
}
```

注意 2：指针的内容可以改变。

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

int main()
{
    int var = 3;
    int * const c_p = &var;
    *c_p = 12;
    return 0;
}
```

函数指针和指针函数的区别

指针函数：
指针函数本质是一个函数，只不过该函数的返回值是一个指针。相对于普通函数而言，只是返回值是指针。

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

struct Type
{
  int var1;
  int var2;
};

Type * fun(int tmp1, int tmp2){
    Type * t = new Type();
    t->var1 = tmp1;
    t->var2 = tmp2;
```

```
    return t;
}

int main()
{
    Type *p = fun(5, 6);
    return 0;
}
```

函数指针：
函数指针本质是一个指针变量，只不过这个指针指向一个函数。函数指针即指向函数的指针。

举例：

```
#include <iostream>
using namespace std;
int fun1(int tmp1, int tmp2)
{
  return tmp1 * tmp2;
}
int fun2(int tmp1, int tmp2)
{
  return tmp1 / tmp2;
}

int main()
{
  int (*fun)(int x, int y);
  fun = fun1;
  cout << fun(15, 5) << endl;
  fun = fun2;
  cout << fun(15, 5) << endl;
  return 0;
}
```

```
运行结果：
75
3
*/
```

函数指针和指针函数的区别：

- 本质不同：
  指针函数本质是一个函数，其返回值为指针。
  函数指针本质是一个指针变量，其指向一个函数。

- 定义形式不同：
  指针函数：int* fun(int tmp1, int tmp2); ，这里* 表示函数的返回值类型是指针类型。
  函数指针：int (*fun)(int tmp1, int tmp2);，*这里* 表示变量本身是指针类型。
- 用法不同

强制类型转换有哪几种?

- static_cast：用于数据的强制类型转换，强制将一种数据类型转换为另一种数据类型。

  1.用于基本数据类型的转换。
  2.用于类层次之间的基类和派生类之间 指针或者引用 的转换（不要求必须包含虚函数， 但必须是有相互联系的类），进行上行转换（派生类的指针或引用转换成基类表示）是安全的；进行下行转换（基类的指针或引用转换成派生类表示）由于没有动态类型检查，所以是不安全的，最好用 dynamic_cast 进行下行转换。
  3.可以将空指针转化成目标类型的空指针。
  4.可以将任何类型的表达式转化成 void 类型。
- const_cast：强制去掉常量属性，不能用于去掉变量的常量性，只能用于去除指针或引用的常量性，将常量指针转化为非常量指针或者将常量引用转化为非常量引用（注意：表达式的类型和要转化的类型是相同的）。
- reinterpret_cast：改变指针或引用的类型、将指针或引用转换为一个足够长度的整型、将整型转化为指针或引用类型。
- dynamic_cast：

  1.其他三种都是编译时完成的，动态类型转换是在程序运行时处理的，运行时会进行类型检查。
  2.只能用于带有虚函数的基类或派生类的指针或者引用对象的转换，转换成功返回指向类型的指针或引用，转换失败返回 NULL；不能用于基本数据类型的转换。
  3.在向上进行转换时，即派生类类的指针转换成基类类的指针和 static_cast 效果是一样的，（注意：这里只是改变了指针的类型，指针指向的对象的类型并未发生改变）。

```
#include <iostream>
#include <cstring>

using namespace std;

class Base
{
};

class Derive : public Base
{
};

int main()
{
```

```
    Base *p1 = new Derive();
    Derive *p2 = new Derive();

//向上类型转换
p1 = dynamic_cast<Base *>(p2);
if (p1 == NULL)
{
    cout << "NULL" << endl;
}
else
{
    cout << "NOT NULL" << endl; //输出
}

return 0;

}
```

4.在下行转换时，基类的指针类型转化为派生类类的指针类型，只有当要转换的指针指向的对象类型和转化以后的对象类型相同时，才会转化成功。

```
#include <iostream>
#include <cstring>

using namespace std;

class Base
{
public:
    virtual void fun()
    {
        cout << "Base::fun()" << endl;
    }
};

class Derive : public Base
{
public:
    virtual void fun()
    {
        cout << "Derive::fun()" << endl;
    }
};

int main()
{
    Base *p1 = new Derive();
    Base *p2 = new Base();
    Derive *p3 = new Derive();
```

```
    //转换成功
    p3 = dynamic_cast<Derive *>(p1);
    if (p3 == NULL)
    {
        cout << "NULL" << endl;
    }
    else
    {
        cout << "NOT NULL" << endl; // 输出
    }

    //转换失败
    p3 = dynamic_cast<Derive *>(p2);
    if (p3 == NULL)
    {
        cout << "NULL" << endl; // 输出
    }
    else
    {
        cout << "NOT NULL" << endl;
    }

    return 0;

    }
```

如何判断结构体是否相等？能否用 memcmp 函数判断结构体相等？

需要重载操作符 == 判断两个结构体是否相等，不能用函数 memcmp 来判断两个结构体是否相等，因为 memcmp 函数是逐个字节进行比较的，而结构体存在内存空间中保存时存在字节对齐，字节对齐时补的字节内容是随机的，会产生垃圾值，所以无法比较。

利用运算符重载来实现结构体对象的比较：

```
#include <iostream>

using namespace std;

struct A
{
    char c;
    int val;
    A(char c_tmp, int tmp) : c(c_tmp), val(tmp) {}

    friend bool operator==(const A &tmp1, const A &tmp2); //  友元运算符重载函数

};
```

```
bool operator==(const A &tmp1, const A &tmp2)
{
    return (tmp1.c == tmp2.c && tmp1.val == tmp2.val);
}

int main()
{
    A ex1('a', 90), ex2('b', 80);
    if (ex1 == ex2)
        cout << "ex1 == ex2" << endl;
    else
        cout << "ex1 != ex2" << endl; // 输出
    return 0;
}
```

参数传递时，值传递、引用传递、指针传递的区别?

参数传递的三种方式：

- 值传递：形参是实参的拷贝，函数对形参的所有操作不会影响实参。
- 指针传递：本质上是值传递，只不过拷贝的是指针的值，拷贝之后，实参和形参是不同的指针，通过指针可以间接的访问指针所指向的对象，从而可以修改它所指对象的值。
- 引用传递：当形参是引用类型时，我们说它对应的实参被引用传递。

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

void fun1(int tmp){ // 值传递
    cout << &tmp << endl;
}

void fun2(int * tmp){ // 指针传递
    cout << tmp << endl;
}

void fun3(int &tmp){ // 引用传递
    cout << &tmp << endl;
}

int main()
{
    int var = 5;
    cout << "var 在主函数中的地址：" << &var << endl;

    cout << "var 值传递时的地址：";
    fun1(var);

    cout << "var 指针传递时的地址：";
```

```
    fun2(&var);

    cout << "var 引用传递时的地址：";
    fun3(var);
    return 0;

}

/*
运行结果：
var 在主函数中的地址：0x23fe4c
var 值传递时的地址：0x23fe20
var 指针传递时的地址：0x23fe4c
var 引用传递时的地址：0x23fe4c
*/
```

说明：从上述代码的运行结果可以看出，只有在值传递时，形参和实参的地址不一样，在函数体内操作的不是变量本身。引用传递和指针传递，在函数体内操作的是变量本身。

什么是模板？如何实现？

模板：创建类或者函数的蓝图或者公式，分为函数模板和类模板。
实现方式：模板定义以关键字 template 开始，后跟一个模板参数列表。

- 模板参数列表不能为空；
- 模板类型参数前必须使用关键字 class 或者 typename，在模板参数列表中这两个关键字含义相同，可互换使用。

```
template <typename T, typename U, ...>
```

函数模板：通过定义一个函数模板，可以避免为每一种类型定义一个新函数。

- 对于函数模板而言，模板类型参数可以用来指定返回类型或函数的参数类型，以及在函数体内用于变量声明或类型转换。
- 函数模板实例化：当调用一个模板时，编译器用函数实参来推断模板实参，从而使用实参的类型来确定绑定到模板参数的类型。

```
#include<iostream>

using namespace std;

template <typename T>
T add_fun(const T & tmp1, const T & tmp2){
    return tmp1 + tmp2;
}

int main(){
    int var1, var2;
    cin >> var1 >> var2;
```

```
    cout << add_fun(var1, var2);

    double var3, var4;
    cin >> var3 >> var4;
    cout << add_fun(var3, var4);
    return 0;

}
```

类模板：类似函数模板，类模板以关键字 template 开始，后跟模板参数列表。但是，编译器不能为类模板推断模板参数类型，需要在使用该类模板时，在模板名后面的尖括号中指明类型。

```
#include <iostream>

using namespace std;

template <typename T>
class Complex
{
public:
    //构造函数
    Complex(T a, T b)
    {
        this->a = a;
        this->b = b;
    }

    //运算符重载
    Complex<T> operator+(Complex &c)
    {
        Complex<T> tmp(this->a + c.a, this->b + c.b);
        cout << tmp.a << " " << tmp.b << endl;
        return tmp;
    }

private:
    T a;
    T b;
};

int main()
{
    Complex<int> a(10, 20);
    Complex<int> b(20, 30);
    Complex<int> c = a + b;

    return 0;
```

```
}
```

- **函数模板和类模板的区别?**
- 实例化方式不同：函数模板实例化由编译程序在处理函数调用时自动完成，类模板实例化需要在程序中显示指定。
- 实例化的结果不同：函数模板实例化后是一个函数，类模板实例化后是一个类。
- 默认参数：类模板在模板参数列表中可以有默认参数。
- 特化：函数模板只能全特化；而类模板可以全特化，也可以偏特化。
- 调用方式不同：函数模板可以隐式调用，也可以显示调用；类模板只能显示调用。

函数模板调用方式举例：

```
#include<iostream>

using namespace std;

template <typename T>
T add_fun(const T & tmp1, const T & tmp2){
    return tmp1 + tmp2;
}

int main(){
    int var1, var2;
    cin >> var1 >> var2;
    cout << add_fun<int>(var1, var2); // 显示调用

    double var3, var4;
    cin >> var3 >> var4;
    cout << add_fun(var3, var4); // 隐式调用
    return 0;

}
```

什么是可变参数模板?

可变参数模板：接受可变数目参数的模板函数或模板类。将可变数目的参数被称为参数包，包括模板参数包和函数参数包。

- 模板参数包：表示零个或多个模板参数；
- 函数参数包：表示零个或多个函数参数。

用省略号来指出一个模板参数或函数参数表示一个包，在模板参数列表中，class... 或 typename... 指出接下来的参数表示零个或多个类型的列表；一个类型名后面跟一个省略号表示零个或多个给定类型的非类型参数的列表。当需要知道包中有多少元素时，可以使用 sizeof... 运算符。

```
template <typename T, typename... Args> // Args 是模板参数包
void foo(const T &t, const Args&... rest); // 可变参数模板，rest 是函数参数包
```

实例：

```
#include <iostream>

using namespace std;

template <typename T>
void print_fun(const T &t)
{
    cout << t << endl; // 最后一个元素
}

template <typename T, typename... Args>
void print_fun(const T &t, const Args &...args)
{
    cout << t << " ";
    print_fun(args...);
}

int main()
{
    print_fun("Hello", "wolrd", "!");
    return 0;
}
/*运行结果：
Hello wolrd !

*/
```

说明：可变参数函数通常是递归的，第一个版本的 print_fun 负责终止递归并打印初始调用中的最后一个实参。第二个版本的 print_fun 是可变参数版本，打印绑定到 t 的实参，并用来调用自身来打印函数参数包中的剩余值。

什么是模板特化？为什么特化?

模板特化的原因：模板并非对任何模板实参都合适、都能实例化，某些情况下，通用模板的定义对特定类型不合适，可能会编译失败，或者得不到正确的结果。因此，当不希望使用模板版本时，可以定义类或者函数模板的一个特例化版本。

模板特化：模板参数在某种特定类型下的具体实现。分为函数模板特化和类模板特化

- 函数模板特化：将函数模板中的全部类型进行特例化，称为函数模板特化。
- 类模板特化：将类模板中的部分或全部类型进行特例化，称为类模板特化。

特化分为全特化和偏特化：

- 全特化：模板中的模板参数全部特例化。
- 偏特化：模板中的模板参数只确定了一部分，剩余部分需要在编译器编译时确定。

说明：要区分下函数重载与函数模板特化
定义函数模板的特化版本，本质上是接管了编译器的工作，为原函数模板定义了一个特殊实例，而不是函数重载，函数模板特化并不影响函数匹配。

实例：

```
#include <iostream>
#include <cstring>

using namespace std;
//函数模板
template <class T>
bool compare(T t1, T t2)
{
    cout << "通用版本：";
    return t1 == t2;
}

template <> //函数模板特化
bool compare(char *t1, char *t2)
{
    cout << "特化版本：";
    return strcmp(t1, t2) == 0;
}

int main(int argc, char *argv[])
{
    char arr1[] = "hello";
    char arr2[] = "abc";
    cout << compare(123, 123) << endl;
    cout << compare(arr1, arr2) << endl;

    return 0;

}
/*
运行结果：
通用版本：1
特化版本：0
```

include " " 和 <> 的区别

include<文件名> 和 #include"文件名" 的区别:

- 查找文件的位置：include<文件名> 在标准库头文件所在的目录中查找，如果没有，再到当前源文件所在目录下查找；#include"文件名" 在当前源文件所在目录中进行查找，如果没有；再到系统目录中查找。

- 使用习惯：对于标准库中的头文件常用 include<文件名>，对于自己定义的头文件，常用 #include"文件名"

switch 的 case 里为何不能定义变量

- switch 下面的这个花括号表示一块作用域，而不是每一个 case 表示一块作用域。如果在某一 case 中定义了变量，其作用域在这块花括号内，按理说在另一个 case 内可以使用该变量，但是在实际使用时，每一个 case 之间互不影响，是相对封闭的，参考如下实例。

实例：
下述代码中，在 switch 的 case 中定义的变量，没有实际意义，仅为了解释上述原因。

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

int main()
{
    // 局部变量声明
    char var = 'D';

    switch (var)
    {
    case 'A':
        int cnt = 0; // 定义变量
        cout << "Excellent." << endl
             << cnt;
        break;
    case 'B':
    case 'C':
        ++cnt;
        cout << "Good." << endl
             << cnt;
        break;
    case 'D':
        cout << "Not bad." << endl
             << cnt;
        break;
    case 'F':
        cout << "Bad." << endl
             << cnt;
        break;
    default:
        cout << "Bad." << endl
             << cnt;
    }

    return 0;

}
```

简单解释：上述代码中在符合 A 的条件下定义了变量，当符合 B 或者 C 的条件时，对该变量进行自增操作，但是因为不符合条件 A 未对变量进行定义，该变量无法使用。

迭代器的作用?

迭代器：一种抽象的设计概念，在设计模式中有迭代器模式，即提供一种方法，使之能够依序寻访某个容器所含的各个元素，而无需暴露该容器的内部表述方式。

作用：在无需知道容器底层原理的情况下，遍历容器中的元素。

实例：

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

int main()
{
    vector<int> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    vector<int>::iterator iter = arr.begin(); // 定义迭代器
    for (; iter != arr.end(); ++iter)
    {
        cout << *iter << " ";
    }
    return 0;
}
/*
运行结果：
1 2 3 4 5 6 7 8 9 0
*/
```

泛型编程如何实现?

泛型编程实现的基础：模板。模板是创建类或者函数的蓝图或者说公式，当时用一个 vector 这样的泛型，或者 find 这样的泛型函数时，编译时会转化为特定的类或者函数。

泛型编程涉及到的知识点较广，例如：容器、迭代器、算法等都是泛型编程的实现实例。面试者可选择自己掌握比较扎实的一方面进行展开。

- 容器：涉及到 STL 中的容器，例如：vector、list、map 等，可选其中熟悉底层原理的容器进行展开讲解。
- 迭代器：在无需知道容器底层原理的情况下，遍历容器中的元素。
- 模板：可参考本章节中的模板相关问题。

什么是类型萃取?

类型萃取使用模板技术来萃取类型（包含自定义类型和内置类型）的某些特性，用以判断该类型是否含有某些特性，从而在泛型算法中来对该类型进行特殊的处理用来提高效率或者其他。

C++ 类型萃取一般用于模板中，当我们定义一个模板函数后，需要知道模板类型形参并加以运用时就可以用类型萃取。
比如我们需要在函数中进行拷贝，通常我们可以用内置函数 memcpy 或者自己写一个 for 循环来进行拷贝。

设计模式

了解哪些设计模式?

《大话设计模式》一书中提到 24 种设计模式，这 24 种设计模式没必要面面俱到，但一定要深入了解其中的几种，最好结合自己在实际开发过程中的例子进行深入的了解。

设计模式有 6 大设计原则：

- 单一职责原则：就一个类而言，应该仅有一个引起它变化的原因。
- 开放封闭原则：软件实体可以扩展，但是不可修改。即面对需求，对程序的改动可以通过增加代码来完成，但是不能改动现有的代码。
- 里氏代换原则：一个软件实体如果使用的是一个基类，那么一定适用于其派生类。即在软件中，把基类替换成派生类，程序的行为没有变化。
- 依赖倒转原则：抽象不应该依赖细节，细节应该依赖抽象。即针对接口编程，不要对实现编程。
- 迪米特原则：如果两个类不直接通信，那么这两个类就不应当发生直接的相互作用。如果一个类需要调用另一个类的某个方法的话，可以通过第三个类转发这个调用。
- 接口隔离原则：每个接口中不存在派生类用不到却必须实现的方法，如果不然，就要将接口拆分，使用多个隔离的接口。

设计模式分为三类：

- 创造型模式：单例模式、工厂模式、建造者模式、原型模式
- 结构型模式：适配器模式、桥接模式、外观模式、组合模式、装饰模式、享元模式、代理模式
- 行为型模式：责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式、访问者模式

下面介绍常见的几种设计模式：

- 单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- 工厂模式：包括简单工厂模式、抽象工厂模式、工厂方法模式
    - 简单工厂模式：主要用于创建对象。用一个工厂来根据输入的条件产生不同的类，然后根据不同类的虚函数得到不同的结果。
    - 工厂方法模式：修正了简单工厂模式中不遵守开放封闭原则。把选择判断移到了客户端去实现，如果想添加新功能就不用修改原来的类，直接修改客户端即可。

    - 抽象工厂模式：定义了一个创建一系列相关或相互依赖的接口，而无需指定他们的具体类。
- 观察者模式：定义了一种一对多的关系，让多个观察对象同时监听一个主题对象，主题对象发生变化时，会通知所有的观察者，使他们能够更新自己。
- 装饰模式：动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成派生类更为灵活。

什么是单例模式？如何实现？应用场景？

单例模式：保证类的实例化对象仅有一个，并且提供一个访问他的全局访问点。

应用场景：

- 表示文件系统的类，一个操作系统一定是只有一个文件系统，因此文件系统的类的实例有且仅有一个。
- 打印机打印程序的实例，一台计算机可以连接好几台打印机，但是计算机上的打印程序只有一个，就可以通过单例模式来避免两个打印作业同时输出到打印机。

实现方式：
单例模式可以通过全局或者静态变量的形式实现，这样比较简单，但是这样会影响封装性，难以保证别的代码不会对全局变量造成影响。

- 默认的构造函数、拷贝构造函数、赋值构造函数声明为私有的，这样禁止在类的外部创建该对象；
- 全局访问点也要定义成 静态类型的成员函数，没有参数，返回该类的指针类型。因为使用实例化对象的时候是通过类直接调用该函数，并不是先创建一个该类的对象，通过对象调用。

不安全的实现方式：
原因：考虑当两个线程同时调用 getInstance 方法，并且同时检测到 instance 是 NULL，两个线程会同时实例化对象，不符合单例模式的要求。

```
class Singleton{
private:
    static Singleton * instance;
    Singleton(){}
    Singleton(const Singleton& tmp){}
    Singleton& operator=(const Singleton& tmp){}
public:
    static Singleton* getInstance(){
        if(instance == NULL){
            instance = new Singleton();
        }
        return instance;
    }
};
Singleton* Singleton::instance = NULL;
```

分类：

- 懒汉模式：直到第一次用到类的实例时才去实例化，上面是懒汉实现。
- 饿汉模式：类定义的时候就实例化。

线程安全的懒汉模式实现：
方法：加锁
存在的问题：每次判断实例对象是否为空，都要被锁定，如果是多线程的话，就会造成大量线程阻塞。

```
class Singleton{
private:
    static pthread_mutex_t mutex;
    static Singleton * instance;
    Singleton(){
        pthread_mutex_init(&mutex, NULL);
    }
    Singleton(const Singleton& tmp){}
    Singleton& operator=(const Singleton& tmp){}
public:
    static Singleton* getInstance(){
        pthread_mutex_lock(&mutex);
        if(instance == NULL){
            instance = new Singleton();
        }
        pthread_mutex_unlock(&mutex);
        return instance;
    }
};
Singleton* Singleton::instance = NULL;
pthread_mutex_t Singleton::mutex;
```

方法：内部静态变量，在全局访问点 getInstance 中定义静态实例。

```
class Singleton{
private:
    static pthread_mutex_t mutex;
    Singleton(){
        pthread_mutex_init(&mutex, NULL);
    }
    Singleton(const Singleton& temp){}
    Singleton& operator=(const Singleton& temp){}
public:
    static Singleton* getInstance(){
        static Singleton instance;
        return &instance;
    }
};
pthread_mutex_t Singleton::mutex;
```

饿汉模式的实现：
饿汉模式本身就是线程安全的不用加锁。

```
class Singleton{
private:
    static Singleton* instance;
    Singleton(const Singleton& temp){}
    Singleton& operator=(const Singleton& temp){}
protected:
     Singleton(){}
public:
    static Singleton* getInstance(){
        return instance;
    }
};
Singleton* Singleton::instance = new Singleton();
```

什么是工厂模式？如何实现？应用场景？

工厂模式：包括简单工厂模式、抽象工厂模式、工厂方法模式

- 简单工厂模式：主要用于创建对象。用一个工厂来根据输入的条件产生不同的类，然后根据不同类的虚函数得到不同的结果。
- 工厂方法模式：修正了简单工厂模式中不遵守开放封闭原则。把选择判断移到了客户端去实现，如果想添加新功能就不用修改原来的类，直接修改客户端即可。
- 抽象工厂模式：定义了一个创建一系列相关或相互依赖的接口，而无需指定他们的具体类。

简单工厂模式
主要用于创建对象。用一个工厂来根据输入的条件产生不同的类，然后根据不同类的虚函数得到不同的结果。

应用场景：

- 适用于针对不同情况创建不同类时，只需传入工厂类的参数即可，无需了解具体实现方法。例如：计算器中对于同样的输入，执行不同的操作：加、减、乘、除。

实现方式：

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

// Here is the product class
class Operation
{
public:
    int var1, var2;
    virtual double GetResult()
    {
```

```
        double res = 0;
        return res;
    }
};

class Add_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 + var2;
    }
};

class Sub_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 - var2;
    }
};

class Mul_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 * var2;
    }
};

class Div_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 / var2;
    }
};

// Here is the Factory class
class Factory
{
public:
    static Operation *CreateProduct(char op)
    {
        switch (op)
        {
```

```
        case '+':
            return new Add_Operation();

        case '-':
            return new Sub_Operation();

        case '*':
            return new Mul_Operation();

        case '/':
            return new Div_Operation();

        default:
            return new Add_Operation();
        }
    }

};

int main()
{
    int a, b;
    cin >> a >> b;
    Operation *p = Factory::CreateProduct('+');
    p->var1 = a;
    p->var2 = b;
    cout << p->GetResult() << endl;

    p = Factory::CreateProduct('*');
    p->var1 = a;
    p->var2 = b;
    cout << p->GetResult() << endl;

    return 0;

}
```

工厂方法模式

修正了简单工厂模式中不遵守开放封闭原则。把选择判断移到了客户端去实现，如果想添加新功能就不用修改原来的类，直接修改客户端即可。

应用场景：

- 一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。
- 一个类通过其派生类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其派生类来确定具体要创建的对象，利用面向对象的多

态性和里氏代换原则，在程序运行时，派生类对象将覆盖父类对象，从而使得系统更容易扩展。
- 将创建对象的任务委托给多个工厂派生类中的某一个，客户端在使用时可以无须关心是哪一个工厂派生类创建产品派生类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

实现方式：

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

// Here is the product class
class Operation
{
public:
    int var1, var2;
    virtual double GetResult()
    {
        double res = 0;
        return res;
    }
};

class Add_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 + var2;
    }
};

class Sub_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 - var2;
    }
};

class Mul_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 * var2;
    }
```

```
};

class Div_Operation : public Operation
{
public:
    virtual double GetResult()
    {
        return var1 / var2;
    }
};

class Factory
{
public:
    virtual Operation *CreateProduct() = 0;
};

class Add_Factory : public Factory
{
public:
    Operation *CreateProduct()
    {
        return new Add_Operation();
    }
};

class Sub_Factory : public Factory
{
public:
    Operation *CreateProduct()
    {
        return new Sub_Operation();
    }
};

class Mul_Factory : public Factory
{
public:
    Operation *CreateProduct()
    {
        return new Mul_Operation();
    }
};

class Div_Factory : public Factory
{
public:
    Operation *CreateProduct()
    {
```

```
        return new Div_Operation();
    }
};

int main()
{
    int a, b;
    cin >> a >> b;
    Add_Factory *p_fac = new Add_Factory();
    Operation *p_pro = p_fac->CreateProduct();
    p_pro->var1 = a;
    p_pro->var2 = b;
    cout << p_pro->GetResult() << endl;

    Mul_Factory *p_fac1 = new Mul_Factory();
    Operation *p_pro1 = p_fac1->CreateProduct();
    p_pro1->var1 = a;
    p_pro1->var2 = b;
    cout << p_pro1->GetResult() << endl;

    return 0;

}
```

抽象工厂模式
定义了一个创建一系列相关或相互依赖的接口，而无需指定他们的具体类。

应用场景：

- 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是重要的。
- 系统中有多于一个的产品族，而每次只使用其中某一产品族。
- 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。
- 产品等级结构稳定，设计完成之后，不会向系统中增加新的产品等级结构或者删除已有的
  产品等级结构。

实现方式：

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

// Here is the product class
class Operation_Pos
{
public:
    int var1, var2;
    virtual double GetResult()
    {
```

```
        double res = 0;
        return res;
    }
};

class Add_Operation_Pos : public Operation_Pos
{
public:
    virtual double GetResult()
    {
        return var1 + var2;
    }
};

class Sub_Operation_Pos : public Operation_Pos
{
public:
    virtual double GetResult()
    {
        return var1 - var2;
    }
};

class Mul_Operation_Pos : public Operation_Pos
{
public:
    virtual double GetResult()
    {
        return var1 * var2;
    }
};

class Div_Operation_Pos : public Operation_Pos
{
public:
    virtual double GetResult()
    {
        return var1 / var2;
    }
};
/**********************************************************************
**********/
class Operation_Neg
{
public:
    int var1, var2;
    virtual double GetResult()
    {
        double res = 0;
```

```
        return res;
    }
};

class Add_Operation_Neg : public Operation_Neg
{
public:
    virtual double GetResult()
    {
        return -(var1 + var2);
    }
};

class Sub_Operation_Neg : public Operation_Neg
{
public:
    virtual double GetResult()
    {
        return -(var1 - var2);
    }
};

class Mul_Operation_Neg : public Operation_Neg
{
public:
    virtual double GetResult()
    {
        return -(var1 * var2);
    }
};

class Div_Operation_Neg : public Operation_Neg
{
public:
    virtual double GetResult()
    {
        return -(var1 / var2);
    }
};
/*************************************************************************
*****************************/

// Here is the Factory class
class Factory
{
public:
    virtual Operation_Pos *CreateProduct_Pos() = 0;
    virtual Operation_Neg *CreateProduct_Neg() = 0;
};
```

```
class Add_Factory : public Factory
{
public:
    Operation_Pos *CreateProduct_Pos()
    {
        return new Add_Operation_Pos();
    }
    Operation_Neg *CreateProduct_Neg()
    {
        return new Add_Operation_Neg();
    }
};

class Sub_Factory : public Factory
{
public:
    Operation_Pos *CreateProduct_Pos()
    {
        return new Sub_Operation_Pos();
    }
    Operation_Neg *CreateProduct_Neg()
    {
        return new Sub_Operation_Neg();
    }
};

class Mul_Factory : public Factory
{
public:
    Operation_Pos *CreateProduct_Pos()
    {
        return new Mul_Operation_Pos();
    }
    Operation_Neg *CreateProduct_Neg()
    {
        return new Mul_Operation_Neg();
    }
};

class Div_Factory : public Factory
{
public:
    Operation_Pos *CreateProduct_Pos()
    {
        return new Div_Operation_Pos();
    }
    Operation_Neg *CreateProduct_Neg()
    {
```

```
        return new Div_Operation_Neg();
    }
};

int main()
{
    int a, b;
    cin >> a >> b;
    Add_Factory *p_fac = new Add_Factory();
    Operation_Pos *p_pro = p_fac->CreateProduct_Pos();
    p_pro->var1 = a;
    p_pro->var2 = b;
    cout << p_pro->GetResult() << endl;

    Add_Factory *p_fac1 = new Add_Factory();
    Operation_Neg *p_pro1 = p_fac1->CreateProduct_Neg();
    p_pro1->var1 = a;
    p_pro1->var2 = b;
    cout << p_pro1->GetResult() << endl;

    return 0;

}
```

什么是观察者模式？如何实现？应用场景？

观察者模式：定义一种一（被观察类）对多（观察类）的关系，让多个观察对象同时监听一个被观察对象，被观察对象状态发生变化时，会通知所有的观察对象，使他们能够更新自己的状态。

观察者模式中存在两种角色：

- 观察者：内部包含被观察者对象，当被观察者对象的状态发生变化时，更新自己的状态。（接收通知更新状态）
- 被观察者：内部包含了所有观察者对象，当状态发生变化时通知所有的观察者更新自己的状态。（发送通知）

应用场景：

- 当一个对象的改变需要同时改变其他对象，且不知道具体有多少对象有待改变时，应该考虑使用观察者模式；
- 一个抽象模型有两个方面，其中一方面依赖于另一方面，这时可以用观察者模式将这两者封装在独立的对象中使它们各自独立地改变和复用。

实现方式：

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

```
class Subject;
//观察者 基类 （内部实例化了被观察者的对象sub）
class Observer
{
protected:
    string name;
    Subject *sub;

public:
    Observer(string name, Subject *sub)
    {
        this->name = name;
        this->sub = sub;
    }
    virtual void update() = 0;
};

class StockObserver : public Observer
{
public:
    StockObserver(string name, Subject *sub) : Observer(name, sub)
    {
    }
    void update();
};

class NBAObserver : public Observer
{
public:
    NBAObserver(string name, Subject *sub) : Observer(name, sub)
    {
    }
    void update();
};
//被观察者 基类 （内部存放了所有的观察者对象，以便状态发生变化时，给观察者发通知）
class Subject
{
protected:
    list<Observer *> observers;

public:
    string action; //被观察者对象的状态
    virtual void attach(Observer *) = 0;
    virtual void detach(Observer *) = 0;
    virtual void notify() = 0;
};

class Secretary : public Subject
```

```
{
    void attach(Observer *observer)
    {
        observers.push_back(observer);
    }
    void detach(Observer *observer)
    {
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
        {
            if ((*iter) == observer)
            {
                observers.erase(iter);
                return;
            }
            ++iter;
        }
    }
    void notify()
    {
        list<Observer *>::iterator iter = observers.begin();
        while (iter != observers.end())
        {
            (*iter)->update();
            ++iter;
        }
    }
};

void StockObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭股票，装做很认真工作的样子！" << endl;
    }
}

void NBAObserver::update()
{
    cout << name << " 收到消息：" << sub->action << endl;
    if (sub->action == "梁所长来了!")
    {
        cout << "我马上关闭NBA，装做很认真工作的样子！" << endl;
    }
}

int main()
{
```

```
    Subject *dwq = new Secretary();
    Observer *xs = new NBAObserver("xiaoshuai", dwq);
    Observer *zy = new NBAObserver("zouyue", dwq);
    Observer *lm = new StockObserver("limin", dwq);

    dwq->attach(xs);
    dwq->attach(zy);
    dwq->attach(lm);

    dwq->action = "去吃饭了！";
    dwq->notify();
    cout << endl;
    dwq->action = "梁所长来了!";
    dwq->notify();
    return 0;

}
```