

# 编程语言C/C++

---

## 一、语言的区别

---

### 1.C++和Python的区别

包括但不限于：

1. Python是一种脚本语言，是解释执行的，而C++是编译语言，是需要编译后在特定平台运行的。python可以很方便的跨平台，但是效率没有C++高。
2. Python使用缩进来区分不同的代码块，C++使用花括号来区分
3. C++中需要事先定义变量的类型，而Python不需要，Python的基本数据类型只有数字，布尔值，字符串，列表，元组等等
4. Python的库函数比C++的多，调用起来很方便

### 2.C++和C语言的区别

1. C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。
2. 标准C++中的字符串类取代了标准C函数库头文件中的字符数组处理函数（C中没有字符串类型）。
3. C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。
4. C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。
5. C++函数可以重载，C语言不允许。
6. C++中，允许变量定义语句在程序中的任何地方，只要是在使用它之前就可以；而C语言中，必须要在函数开头部分。
7. 在++中，除了值和指针之外，新增了引用。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。
8. C++相对与C增加了一些关键字，如：bool、using、dynamic\_cast、namespace等等

### 3.C++与Java的区别

语言特性：

1. Java语言给开发人员提供了更为简洁的语法；完全面向对象，由于JVM可以安装到任何的操作系统上，所以说它的可移植性强
2. Java语言中没有指针的概念，引入了真正的数组。不同于C++中利用指针实现的“伪数组”，Java引入了真正的数组，同时将容易造成麻烦的指针从语言中去掉，这将有利于防止在C++程序中常见的因为数组操作越界等指针操作而对系统数据进行非法读写带来的不安全问题
3. C++也可以在其他系统运行，但是需要不同的编码（这一点不如Java，只编写一次代码，到处运行），例如对一个数字，在windows下是大端存储，在unix中则为小端存储。Java程序一般都是生成字节码，在JVM里面运行得到结果
4. Java用接口(Interface)技术取代C++程序中的抽象类。接口与抽象类有同样的功能，但是省却了在实现和维护上的复杂性

垃圾回收：

2. Java用析构函数回收垃圾，而C和C++程序中一定要注意内存的申请和释放，内存的分配和回收都是手动进行的，程序员无须考虑内存碎片的问题

## 应用场景

1. Java在桌面程序上不如C++实用，C++可以直接编译成exe文件，指针是c++的优势，可以直接对内存的操作，但同时具有危险性。（操作内存的确是一项非常危险的事情，一旦指针指向的位置发生错误，或者误删除了内存中某个地址单元存放的重要数据，后果是可想而知的）
2. Java在Web 应用上具有C++ 无可比拟的优势，具有丰富多样的框架
3. 对于底层程序的编程以及控制方面的编程，C++很灵活，因为有句柄的存在

## 4.为什么C++没有垃圾回收机制？这点跟Java不太一样

1. 首先，实现一个垃圾回收器会带来额外的空间和时间开销。你需要开辟一定的空间保存指针的引用计数和对他们进行标记mark。然后需要单独开辟一个线程在空闲的时候进行free操作
2. 垃圾回收会使得C++不适合进行很多底层的操作

## 二、关键字

### 1.new/delete/malloc/free/realloc/calloc

#### 1.1new和delete是如何实现的？

1. new的实现过程是：首先调用名为**operator new**的标准库函数，分配足够大的内存，以保存指定类型的一个对象；接下来运行该类型的一个构造函数，用指定初始化构造对象；最后返回指向新分配并构造后的对象的指针
2. delete的实现过程：对指针指向的对象运行适当的析构函数；然后通过调用名为**operator delete**的标准库函数释放该对象所用内存
3. new只会调用一次构造函数，new[]会调用多次，同理delete只会析构一次，delete[]会析构多次，在调用new[]时，需要多分配4个字节的空间，专门用来存储数组的大小，delete[]时就可以根据数组的大小来确定调用析构函数的次数

#### 1.2malloc和new的区别？

1. malloc和free是标准库函数，支持覆盖；new和delete是运算符，支持重载。
2. malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险；new和delete除了分配回收功能外，还会调用构造函数和析构函数。
3. malloc和free返回的是void类型指针（必须进行类型转换），new和delete返回的是具体类型指针。

#### 1.3new / delete 与 malloc / free的异同

相同点：

1. 都可用于内存的动态申请和释放

不同点：

2. 前者是C++运算符，后者是C/C++语言标准库函数

3. new自动计算要分配的空间大小，malloc需要手工计算
4. new是类型安全的，malloc不是，malloc返回的void类型指针，不会区分类型
5. new调用名为**operator new**的标准库函数分配足够空间并调用相关对象的构造函数，delete对指针所指对象运行适当的析构函数；然后通过调用名为**operator delete**的标准库函数释放该对象所用内存。后者均没有相关调用
6. 后者需要库文件支持，前者不用
7. new是封装了malloc，直接free不会报错，但是这只是释放内存，而不会析构对象

## 1.4有了malloc/free，没什么还要有new/delete

1. malloc/free和new/delete都是用来申请内存和回收内存的
2. 在对非基本数据类型的对象使用的时候，对象创建的时候还需要执行构造函数，销毁的时候要执行析构函数。而malloc/free是库函数，是已经编译的代码，所以无法实现调用构造函数和析构函数的功能，所以new/delete是必不可少的
3. malloc /free主要为了兼容C，new和delete 完全可以取代malloc /free

## 1.5被free回收的内存是立即返还给操作系统吗？

不是的，被free回收的内存会首先被内存分配器**ptmalloc**使用双链表保存起来，当用户下一次申请内存的时候，会尝试从这些内存中寻找合适的返回。这样就避免了频繁的系统调用，占用过多的系统资源。同时ptmalloc也会尝试对小块内存进行合并，避免过多的内存碎片。

## 1.6malloc、realloc、calloc的区别

```
void* malloc(unsigned int num_size);
int *p = malloc(20*sizeof(int)); // 申请20个int类型的空间；

// 省去了人为空间计算；malloc申请的空间的值是随机初始化的，calloc申请的空间的值是初始化为0的；
void* calloc(size_t n, size_t size);
int *p = calloc(20, sizeof(int));

// 给动态分配的空间分配额外的空间，用于扩充容量。
void realloc(void *p, size_t new_size);
```

## 1.7C++中有几种类型的新

在C++中，new有三种典型的使用方法：plain new，nothrow new和placement new

### (1)plain new

plain new就是我们通常使用的new，它会先分配空间，然后调用类的构造函数，如果空间分配失败，会抛出**bad\_alloc**异常，而不是返回nullptr指针

### (2)nothrow new

nothrow new在空间分配失败的情况下不会抛出异常，而是返回nullptr

```
char *p = new(nothrow) char[10]; // 用法
```

### (3)placement new

placement new允许在一块已经分配成功的内存上重新构造对象或对象数组。placement new不用担心内存分配失败，因为它根本不分配内存，它做的唯一一件事情就是调用对象的构造函数

注意：

1. placement new的主要用途就是反复使用一块较大的动态分配的内存来构造不同类型的对象或者他们的数组
2. placement new构造起来的对象数组，要显式的调用他们的析构函数来销毁（析构函数并不释放对象的内存），千万不要使用delete，这是因为placement new构造起来的对象或数组大小并不一定等于原来分配的内存大小，使用delete会造成内存泄漏或者之后释放内存时出现运行时错误。

```
#include <iostream>
#include <string>
using namespace std;
class ADT{
    int i;
    int j;
public:
    ADT(){
        i = 10;
        j = 100;
        cout << "ADT construct i=" << i << "j=" << j << endl;
    }
    ~ADT(){
        cout << "ADT destruct" << endl;
    }
};
int main()
{
    char *p = new(nothrow) char[sizeof ADT + 1];
    if (p == NULL) {
        cout << "alloc failed" << endl;
    }
    // placement new:不必担心失败，只要p所指对象的的空间足够ADT创建即可
    // 用法，new(p),p为已经分配内存的指针
    ADT *q = new(p) ADT;

    //delete q; // 错误!不能在此处调用delete q;
    q->ADT::~~ADT(); //显示调用析构函数

    // 最后通过p来delete
    delete[] p;
    return 0;
}
//输出结果：
//ADT construct i=10j=100
//ADT destruct
```

## 2.define/typedef

### 2.1宏定义和函数有何区别？

1. 宏在预处理阶段完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
2. 宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。
3. 宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
4. 宏定义不要在最后加分号。

### 2.2内联函数和宏定义的区别

1. 在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且具有返回值。
2. 内联函数在编译时直接将函数代码嵌入到目标代码中，省去函数调用的开销来提高执行效率，并且进行参数类型检查，具有返回值，可以实现重载。
3. 宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义
4. 内联函数有类型检测、语法判断等功能，而宏没有

#### 内联函数适用场景：

1. 使用宏定义的地方都可以使用 inline 函数。
2. 作为类成员接口函数来读写类的私有成员或者保护成员，会提高效率。

#### 内联函数不适用场景：

内联函数以代码复杂为代价，以省去函数调用的开销来提高执行效率，因此如果内联函数体内代码执行时间相比函数调用开销较大，则没有太大的意义；内联函数的调用都要复制代码，消耗更多的内存空间。不是使用内联函数的场景：

1. 函数体内的代码比较长，将导致内存消耗代价
2. 函数体内有循环，函数执行时间要比函数调用开销大

### 2.3宏定义和typedef区别？

1. 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
2. 宏替换发生在预处理阶段，属于文本插入替换；typedef发生在编译阶段
3. 宏不检查类型；typedef会检查数据类型。
4. 宏不是语句，不在最后加分号；typedef是语句，要加分号标识结束。
5. 注意对指针的操作，typedef char \* p\_char和#define p\_char char \*区别巨大。

```
typedef char* PCHAR;  
PCHAR pa,pb;    // pa,pb都是字符指针类型  
  
#define PCHAR char*  
PCHAR pa,pb;    // PCHAR pa,pb;会被翻译成char *pa,pb, 这导致pa是字符指针类型，而pb是字符  
类型
```

## 2.4define宏定义和const的区别

### 作用阶段

1. define是在编译的**预处理**阶段起作用，而const是在编译、运行的时候起作用

### 安全性

2. define只做替换，不做类型检查和计算，也不求解，容易产生错误，一般最好加上一个大括号包住全部的内容，要不然很容易出错
3. 宏不检查类型；const会检查数据类型，const更加安全

### 内存占用

4. define只是将宏名称进行替换，在内存中会产生多份相同的备份，占用代码段空间。const在程序运行中只有一份备份，占用数据段空间
5. 宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配内存空间。

### 其他

6. const不能重定义，而define可以通过#undef取消某个符号的定义，进行重定义；
7. define可以用来防止文件重复引用

## 3.ifdef/ifndef/endif

### 3.1说一下你理解的 ifdef endif代表着什么？

1. 用于条件编译，一般情况下，源程序的所有代码都会参与编译，但有时希望一部分代码只在满足一定条件才进行编译，这时可以用ifdef /ifndef/endif，进行条件编译
2. 用法

```
// 当标识符已经被定义过(一般是用#define命令定义)，则对程序段1进行编译，否则编译程序段2
#ifdef 标识符
    程序段1
#else
    程序段2
#endif

// #else部分也可以没有
#ifdef 标识符
    程序段1
#endif
```

3. 在一个大的软件工程里面，可能会有多个文件同时包含一个头文件，当这些文件编译链接成一个可执行文件上时，就会出现大量“重定义”错误。在头文件中使用#define、#ifndef、#ifdef、#endif能避免头文件重定义。

## 4.const

### 4.1C++的顶层const和底层const

- **顶层const**: 指的是const修饰的变量**本身**是一个常量, 无法修改, 指的是指针, 就是 \* 号的右边
- **底层const**: 指的是const修饰的变量**所指向的对象**是一个常量, 指的是指针所指变量, 就是 \* 号的左边

**总结: 顶层const修饰变量本身, 底层const修饰指向**

```
int a = 10;
int* const b1 = &a;           // 顶层const, b1本身是一个常量
const int* b2 = &a;           // 底层const, b2本身可变, 所指的对象是常量
const int b3 = 20;            // 顶层const, b3是常量不可变
const int* const b4 = &a;      // 前一个const为底层, 后一个为顶层, b4不可变
const int& b5 = a;            // 用于声明引用变量, 都是底层const
```

### 4.2const关键字的作用有哪些?

#### (1) 类外:

1. const修饰变量(引用)和数组, 表示常量, 通知编译器该变量是不可修改的
2. const修饰指针: 指针常量、常量指针
3. const修饰函数:
  1. const修饰函数参数: 表示该参数是不允许修改的, 一般修饰指针(常量指针)和引用参数
  2. const修饰函数返回值: 值传递没有必要, 一位传递的只是一个临时变量; 返回值为指针, 要用const变量来接收(即对const变量进行初始化), 否则报错

#### (2) 类内:

1. 修饰类成员函数(常函数), 表示该函数不能修改类成员变量(mutable修饰的变量除外)
  1. const对象只能访问const成员函数, 非const对象都可以访问
  2. const可以作为函数重载的参考, 函数名、参数、返回值都相同的const成员函数和非const成员函数是可以构成重载(const对象调用const成员函数, 非const对象默认调用非const的成员函数)
  3. const成员函数只能访问const成员函数, 不能访问非const成员函数(因为非const成员函数可能会修改成员变量)
2. 修饰类成员变量, 修饰普通类成员变量和修饰类外变量类似, 但static const 初始化时机与只使用static有所不同:
  1. static const 支持定义时初始化(C++11之后, 普通成员变量也可以)
  2. **static只能类内声明, 类外初始化**
  3. const成员变量只能在初始化列表中初始化(C++03), C++11也允许定义时初始化

## 5.static

### 5.1static的用法和作用？

C语言的static主要有三种作用：

1. 全局静态变量具有文件隔离的作用，当同时编译多个文件时，没有用static修饰的全局变量和函数具有全局可见性，用static修饰的全局变量和函数只在本文件中可见
2. 局部静态变量具有保持内容的持久性的作用，
  1. 静态变量和全局变量都存储在静态存储区
  2. 局部静态变量只会初始化一次（C语言编译阶段分配内存，进行初始化；C++首次执行到相关代码进行初始化），默认初始化为0
  3. 局部静态变量的作用域是函数体内，但声明周期长于函数调用，下次调用还是原来的值
3. 静态函数与全局静态变量类似，具有文件隔离的作用

C++的static的两种作用：修饰类成员函数和类成员变量

1. 修饰类成员函数，
  1. 静态成员函数属于整个类所有
  2. 不接收this指针，只能访问类的静态成员变量和静态成员函数（this指针是指向本对象的指针，没有this指针不能访问非静态成员变量），非静态成员函数可以访问静态/非静态成员变量/函数
  3. 静态成员函数不能是虚函数。static成员不属于任何对象或实例，所以加上virtual没有任何实际意义；静态成员函数没有this指针，虚函数的实现是为每一个对象分配一个vptr指针，而vptr是通过this指针调用的，所以不能为virtual；虚函数的调用关系，this->vptr->ctable->virtual function
2. 修饰类成员变量，
  1. 静态成员变量属于整个类所有，类内声明，类外初始化（先于实例存在）

## 6.sizeof

### 6.1strlen和sizeof区别？

1. sizeof是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen是字符处理的库函数。
2. sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化）；strlen的参数只能是字符指针且结尾是'\0'的字符串。

```
const char* str = "name";

sizeof(str); // 取的是指针str的长度
strlen(str); // 取的是这个字符串的长度，不包含结尾的\0，大小是4
```

补充：一个指针占多少字节？

一个指针占内存的大小跟编译环境有关，而与机器的位数无关。32位编译环境大小为4个字节，64位编译环境大小为8个字节



## 7.class/struct

### 7.1C++中struct和class的区别

#### 相同点

1. 两者都拥有成员函数、公有和私有部分
2. 任何可以使用class完成的工作，同样可以使用struct完成

#### 不同点

1. 两者中如果不对成员不指定公私有，struct默认是公有的，class则默认是私有的
2. class默认是private继承，而struct默认是public继承

### 7.2C++和C的struct区别

1. C语言中：struct是用户自定义数据类型（UDT）；C++中struct是抽象数据类型（ADT），支持成员函数的定义，（C++中的struct能继承，能实现多态）
2. C中struct是没有权限的设置的，且struct中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员**不可以是函数**
3. C++中，struct增加了访问权限，且可以和类一样有成员函数，成员默认访问说明符为public（为了与C兼容）
4. struct作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在C中必须在结构标记前加上struct，才能做结构类型名；C++中结构体标记（结构体名）可以直接作为结构体类型名使用，此外结构体struct在C++中被当作类的一种特例

## 8.final/override

### 8.1final和override关键字

#### override

修饰类成员函数，表示这个函数是重写的父类的虚函数

#### final

可以修饰类和虚函数，修饰类表示该类不能被继承，修饰虚函数表示该虚函数不能被重写，否则编译器会报错

## 9.extern "C"

### 9.1extern"C"的用法

**extern "C"**的作用是告诉C++编译器用C规则编译指定的代码（除函数重载外，**extern "C"**不影响C++其他特性）。

**extern "C"**是C++特有的指令（C无法使用该指令），目的在于支持C++与C混合编程。

**C和C++的编译规则不一样，主要区别体现在编译期间生成函数符号的规则不一致。**

哪些情况下使用extern "C"：

(1) C++调用C语言代码;

我有要调用一段C语言的代码，编译的时候给我按照C语言的规则编译

```
//xx.h
extern int add(...)
//xx.c
int add(){

}

// C++调用C语言代码，在包含头文件时用extern "C"
//xx.cpp
extern "C" {
    #include "xx.h"
}
```

(2) C语言调用C++代码

我有一段代码会被C语言调用，为了它能够正常调用，你给我用C语言的规则来编译

```
//xx.h
extern "C"{
    int add();
}
//xx.cpp
int add(){
}

//xx.c
extern int add();// 使用前要声明
```

## 10.volatile/mutable/explicit

### 10.1volatile、mutable和explicit关键字的用法

#### (1)volatile

volatile 关键字是一种类型修饰符，**用它声明的类型变量表示可以被某些编译器未知的因素更改**，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

当要求使用 volatile 声明的变量的值的时候，**系统总是重新从它所在的内存读取数据**，即使它前面的指令刚刚从该处读取过数据。

**volatile定义变量的值是易变的，每次用到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。多线程中被几个任务共享的变量需要定义为volatile类型。**

总结：

1. volatile 关键字是一种类型修饰符，编译器不会对用它修饰的变量的访问代码进行优化。每次访问 volatile修饰的变量，都要求从它所在的内存读取数据，而不是读寄存器中的备份
2. volatile修饰指针与const类似，存在与常量指针和指针常量相应的概念

- (a) volatile在前, 说明指针指向的对象是volatile的
- (b) \*在前, 说明指针是volatile的
- 3. 在多线程编程中, 当两个线程都要用到每一个变量, 且该变量的值会被改变时, 该变量应该用volatile声明, 防止优化编译器把变量从内存装入CPU寄存器中, 造成两个线程一个读取的是内存中的变量, 另一个读取的是寄存器中的变量的错误

## (2)mutable

mutable关键字是为了突破const的限制而设置的, 被mutable修饰的变量, 将永远处于可变的状态。

- 1. 在类的const函数中, 被mutable修饰的成员变量可以修改
- 2. 被声明为const的类对象, 可以修改其中的mutable成员变量

```
class person
{
    int m_A;
    mutable int m_B; //特殊变量 在常函数里值也可以被修改
public:
    void add() const //在函数里不可修改this指针指向的值 常量指针
    {
        m_A=10; //错误 不可修改值, this已经被修饰为常量指针
        m_B=20; //正确
    }
}

class person
{
    int m_A;
    mutable int m_B; //特殊变量 在常函数里值也可以被修改
}

int main()
{
    const person p; //修饰常对象 不可修改类成员的值
    p.m_A=10;        //错误, 被修饰了指针常量
    p.m_B=200;       //正确, 特殊变量, 修饰了mutable
}
```

## (3)explicit

explicit关键字用来修饰类的构造函数, 被修饰的构造函数的类, 不能发生相应的隐式类型转换, 只能以显示的方式进行类型转换

- 1. explicit 关键字只能用于类内部的构造函数声明上
- 2. explicit 关键字作用于单个参数的构造函数, 或者多个参数, 但除第一个参数外其他参数都有默认值, 否则不会发生隐式转换, 所以explicit关键字就失效了

### 补充隐式转换:

- 1. 隐式转换是指, 不需要用户干预, 编译器进行的数据类型转换行为
- 2. 基本数据类型之间的隐式转换, 以取值范围的作为转换基础 (保证精度不丢失, 从小到大进行转换); 自定义对象, 子类可以隐式转换为父类
- 3. 类的隐式转换是指从构造函数的形参类型到该类类型的编译器的自动转换。隐式类型转换发生在可以用单个形参来调用的构造函数上。可以用单个形参调用并不是指构造函数只有一个形参, 而是它可能有多个形参, 但除第一个外都有默认值

#### 4. 在构造函数声明的时候加上explicit关键字，能够禁止隐式转换

```
// 隐式类型转换是指从构造函数的形参类型到该类类型的编译器的自动转换
// 隐式类型转换发生在可以用单个形参来调用的构造函数上。可以用单个形参调用并不是指构造函数只有一个形参，而是它可能有多个形参，但除第一个外都有默认值
class Mystr
{
public:
    char *m_str;
    int m_size;

    // Mystr(int size)          // 不使用explicit
    explicit Mystr(int size)    // 使用explicit
    {
        m_size = size;
        m_str=malloc(size + 1);
    }

    ~Mystr(){
        free(m_str);
    }
};

Mystr str = 10;                // 在不使用explicit关键字修饰构造函数时，会发生隐式类型转换，这样是ok的
                                // 使用explicit关键字会取消隐式类型转换，会报错
```

## 11.try/throw/catch

### 11.1C++的异常处理的方法

#### C++中的异常情况：

1. 语法错误（编译错误）：比如变量未定义、括号不匹配、关键字拼写错误等等编译器在编译时能发现的错误，这类错误可以及时被编译器发现，而且可以及时知道出错的位置及原因，方便改正。
2. 运行时错误：比如数组下标越界、系统内存不足等等。这类错误不易被程序员发现，它能够通过编译且能进入运行，但运行时会出错，导致程序崩溃。为了有效处理程序运行时错误，C++中引入异常处理机制来解决此问题，**即异常处理机制是为了解决运行时错误的**

在程序执行过程中，由于程序员的疏忽或是系统资源紧张等因素都有可能导致异常，任何程序都无法保证绝对的稳定，常见的异常有：

- 数组下标越界
- 除法计算时除数为0
- 动态分配空间时空间不足
- ...

如果不及对这些异常进行处理，程序多数情况下都会崩溃。

#### (1) try、throw和catch关键字

C++中的异常处理机制主要使用try、throw和catch三个关键字，其在程序中的用法如下：

```
try{// 尝试执行
    if(发生异常的条件1){// 符合条件抛出异常
```

```

        throw 某类型的数据1, 如int型数据, 1, 也可以自定义异常class
    }else if(发生异常的条件2){
        throw 某类型的数据2, 如字符串类型数据, "除0"
    }else{
        程序正常运行的代码
    }
}
// 根据throw抛出的数据类型进行精确捕获 (不会出现类型转换)
// 如果匹配不到就直接报错, 可以使用catch(...)的方式捕获任何异常 (不推荐)
catch(捕获异常类型1, 如, int& a){
    异常处理代码1
    throw; // 如果当前层无法处理, 可以向调用的上层重新抛出异常
}
catch(捕获异常类型2, 如, string& s){
    异常处理代码2
}
// 如果找不到匹配该异常对象的catch语句, 则递归回退到调用栈的上一层的try...catch...块来处理该异常。
// 如果一直退到主函数 main() 都不能处理该异常, 则调用系统函数 terminate() 终止程序。

#include <iostream>
using namespace std;
int main()
{
    double m = 1, n = 0;
    try {
        cout << "before dividing." << endl;
        if (n == 0)
            throw - 1; //抛出int型异常
        else if (m == 0)
            throw - 1.0; //抛出 double 型异常
        else
            cout << m / n << endl;
        cout << "after dividing." << endl;
    }
    catch (double d) {
        cout << "catch (double)" << d << endl;
    }
    catch (...) {
        cout << "catch (...)" << endl;
    }
    cout << "finished" << endl;
    return 0;
}
//运行结果
//before dividing.
//catch (...)
//finished

```

## (2) 函数的异常声明列表

有时候, 程序员在定义函数的时候知道函数可能发生的异常, 可以在函数声明和定义时, 指出所能抛出异常的列表, 写法如下:

```
int fun() throw(int,double,A,B,C){...};
```

这种写法表明函数可能会抛出int,double型或者A、B、C三种类型的异常，如果throw中为空，表明不会抛出任何异常，如果没有throw则可能抛出任何异常

### (3) C++标准异常类 exception

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来的:

1. bad\_typeid: 使用typeid运算符，如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会抛出此异常

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

class A{
public:
    virtual ~A();
};

using namespace std;
int main() {
    A* a = NULL;
    try {
        cout << typeid(*a).name() << endl; // Error condition
    }
    catch (bad_typeid){
        cout << "Object is NULL" << endl;
    }
    return 0;
}
//运行结果: bject is NULL

// 补充: typeid运算符
// typeid用来获取一个表达式的类型信息
// typeid会把获取到的类型信息保存到一个type_info类型的对象里面，并返回该对象的常引用，其中的成员函数name()返回类型的名称
// 如果其操作数是一个多态类的指针，则根据运行时指针所指向的实际类型去计算，指针为空，则抛出bad_typeid异常
```

1. bad\_cast: 在用 dynamic\_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会抛出此异常
2. bad\_alloc: 在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常
3. out\_of\_range: 用 vector 或 string的at 成员函数根据下标访问元素时，如果下标越界，则会抛出此异常

## 12.reinterpret\_cast/const\_cast/static\_cast /dynamic\_cast

### 12.1C++的四种强制转换reinterpret\_cast/const\_cast/static\_cast /dynamic\_cast

数据类型的本质：对确定数据的解释方式。例如int a，表明a这份数据是整数，不能理解为浮点数；

数据类型转换的本质：数据类型转换，就是对数据所占用的二进制位做出重新解释；

### 1. reinterpret\_cast:

**reinterpret\_cast**<type-id> (expression)

仅仅是对二进制位的重新解释，不会借助已有的转换规则对数据进行调整，有着和C风格的强制转换同样的能力，非常简单粗暴，所以非常危险。

### 2. const\_cast

**const\_cast** <type-id> (expression)

去除expression的const/volatile属性，指针/引用指向对象不变

### 3. static\_cast

**static\_cast** <type-id> (expression)

用于父子类之间指针/引用和的转换

    向上转换：安全，向上转换是无条件的

    向下转换：不安全，没有动态类型检查

基本数据类型的相互转换

把空指针转换成目标类型的空指针

把任何类型的表达式转换成void类型

### 4. dynamic\_cast

**dynamic\_cast** <type-id> (expression)

上述三种转换方式都是在编译时完成的，dynamic\_cast是在运行时完成的，运行时需要进行类型检查

用于父子类之间指针/引用和的转换

    向上转换：安全，同static

    向下转换：安全，如果父类指针指向的就是转换类这个子类，则返回转型后的指针，否则返回nullptr

使用dynamic\_cast进行转换的，基类中一定要有虚函数，否则编译不通过（只有存在虚函数才有转换的意义）

补充：

    向上转换：子类指针/引用转成父类指针/引用

    向下转换：父类指针/引用转成子类指针/引用，这种转换是不安全的，因为父类的指针或者引用的内存中可能不包含子类的成员的内存。

static\_cast与dynamic\_cast之间的区别

    向上转换总是安全的，二者一样，没有区别

    向下转换存在风险，dynamic\_cast会进行类型检查，确定安全才会转换成功，否则返回nullptr

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

class Base
{
public:
    Base() :b(1) {}
    virtual void fun() {};
    int b;
};

class Son : public Base
{
public:
    Son() :d(2) {}
    int d;
};

int main()
```

```

{
    int n = 97;
    //1. reinterpret_cast
    int* p = &n;
    //以下两者效果相同
    char* c = reinterpret_cast<char*> (p);
    char* c2 = (char*)(p);
    cout << "reinterpret_cast输出: " << *c2 << endl;

    //2. const_cast
    const int* p2 = &n;
    int* p3 = const_cast<int*>(p2);
    *p3 = 100;
    cout << "const_cast输出: " << *p3 << endl;

    Base* b1 = new Son; // 父类指针指向子类对象
    Base* b2 = new Base;

    //3. static_cast
    Son* s1 = static_cast<Son*>(b1); //同类型转换
    Son* s2 = static_cast<Son*>(b2); //下行转换, 不安全
    cout << "static_cast输出: " << endl;
    cout << s1->d << endl;
    cout << s2->d << endl; //下行转换, 原先父对象没有d成员, 输出垃圾值(不安全的点在这)

    //dynamic_cast
    Son* s3 = dynamic_cast<Son*>(b1); //同类型转换
    Son* s4 = dynamic_cast<Son*>(b2); //下行转换, 安全
    cout << "dynamic_cast输出: " << endl;
    cout << s3->d << endl;
    if (s4 == nullptr) // 转换失败, 输出空指针
        cout << "s4指针为nullptr" << endl;
    else
        cout << s4->d << endl;

    return 0;
}

```

输出结果:

reinterpret\_cast输出: a

const\_cast输出: 100

static\_cast输出:

2

-33686019

dynamic\_cast输出:

2

s4指针为nullptr



## 12.2.static\_cast比C语言中的转换强在哪里？

1. 更加安全；
2. 更直接明显，能够一眼看出是什么类型转换为什么类型，容易找出程序中的错误；可清楚地辨别代码中每个显式的强制转；可读性更好，能体现程序员的意图

## 三、指针和引用

### 1.指针和引用的区别

1. 指针是一个变量，存储的是一个地址，在32位编译环境中占用4个字节的内存空间（sizeof）；引用是给原变量起的别名，跟原变量本质上是同一个东西，sizeof的结果与原变量相同
2. 指针可以有多级（链表），没有所谓的二级引用，但引用也可以有引用，本质都是最开始的变量
3. 引用在声明时必须初始化，且一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针声明和初始化可以分开，可以先只声明指针变量而不初始化，等到使用时再指向具体变量，并且指针变量可以改变指向
4. 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。
5. 当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量
6. 引用本质是一个指针，同样会占4字节内存，但编译器对它做了处理，使得程序认为它步单独占用内存空间

### 2.在传递函数参数时，什么时候该使用指针，什么时候该使用引用呢？

1. 需要返回函数内局部变量的内存的时候用指针。使用指针传参需要开辟内存，用完要记得释放指针，不然会内存泄漏。
2. 对栈空间大小比较敏感（比如递归）的时候使用引用。使用引用传递不需要创建临时变量，开销要更小
3. 类对象作为参数传递的时候使用引用，这是C++类对象传递的标准方式
4. 能用引用的地方尽量用引用

### 3.区别以下指针类型？

```
int *p[10]           // 指针数组，强调数组概念，是一个数组变量，数组大小为10，
                    // 数组内每个元素都是指向int类型的指针变量

int (*p)[10]         // 数组指针，强调是指针，只有一个变量，是指针类型，
                    // 不过指向的是一个int类型的数组，这个数组大小是10

int *p(int)          // 函数声明，函数名是p，参数是int类型的，返回值是int *类型的

int (*p)(int)        // 函数指针，强调是指针，该指针指向的函数具有int类型参数，并且返回值是int类型的
```

## 4.常量指针和指针常量区别？

1. 常量指针是一个指针，指向一个只读变量，可以写作`int const *p`或`const int *p`。
2. 指针常量是一个不能给改变指向的指针，指针是个常量，必须初始化，一旦初始化完成，它的值（也就是存放在指针中的地址）就不能在改变了，即不能中途改变指向，如`int *const p`。

## 5.a和&a有什么区别？

假设数组`int a[10]; int (*p)[10] = &a`;其中：

1. `a`是数组名，是数组首元素地址，`+1`表示地址值加上一个`int`类型的大小，如果`a`的值是`0x00000001`，加1操作后变为`0x00000005`。`*(a + 1) = a[1]`。
2. `&a`是数组的指针，其类型为`int (*)[10]`（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个`int`型变量），值为数组`a`尾元素后一个元素的地址（非法）。但是`a`和`&a`存的地址值是相同的，都是数组首元素的地址
3. 若`(int *)p`，此时输出`*p`时，其值为`a[0]`的值，因为被强转为`int *`类型，解引用时按照`int`类型大小来读取。

## 6.数组名和指针（这里为指向数组首元素的指针）区别？

1. 数组在内存中是连续存放的，开辟一块连续的内存空间；数组所占存储空间（字节数）：`sizeof(数组名)`；数组大小：`sizeof(数组名)/sizeof(数组元素数据类型)`；`sizeof(指针)`，得到的是指针变量的字节数，而不是指针指向内存的大小
2. 二者均可通过增减偏移量来访问数组中的元素
3. 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作
4. 当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但`sizeof`运算符不能再得到原数组的大小了。

## 7.野指针和悬空指针

野指针和悬空指针都是指向无效内存区域的指针，访问行为将会导致未定义行为。

1. 野指针：没有被初始化的指针 --> 定义指针变量及时初始化，要么置空
2. 悬空指针：最初指向的内存已经被释放的指针 --> 释放操作后立即置空

c++引入了智能指针，C++智能指针的本质就是避免悬空指针的产生

```
int* p;      // 未初始化，野指针
```

```
int* p1 = nullptr;
int* p2 = new int;
p1 = p2;
delete p2;
// p1和p2都是悬空指针
```

## 8.值传递、指针传递、引用传递的区别和效率

### (1) 值传递:

值传递过程中，被调函数的形参作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主调函数传递进来的实参值，从而形成了实参的一个副本（有一个数据拷贝的过程）。

值传递的特点是，被调函数对形参的任何操作都是作为局部变量进行的，不会影响主调函数的实参变量的值

### (2) 指针传递:

指针传递本质上是值传递，它所传递的是一个地址值。

### (3) 引用传递:

引用传递过程中，被调函数的形参也作为局部变量在栈中开辟了内存空间，但是存放的是由主调函数放进来的实参变量的地址（也就是这时形参和实参使用的是同一片内存空间）。

被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量（根据别名找到主调函数中的本体）。因此，被调函数对形参的任何操作都会影响主调函数中的实参变量。

### 指针传递和引用传递的区别:

虽然引用传递和指针传递都是在被调函数栈空间上的一个局部变量，但他们是不同的:

1. 对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。
2. 对于指针传递，如果改变被调函数中的指针变量的值（其中存储的地址），它将找不到主调函数的相关变量。
3. 从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。

指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（即形参和实参使用的是同一片内存空间）。

符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

**效率上讲**，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、清晰（可读性好）。引用传递可以在函数内部对实参进行修改

```
// 数据对象是数组，则使用指针，也可以使用引用
#include <iostream>
using namespace std;

void Print(int(&b)[5])
{
    for (int i = 0; i < 5; ++i)
    {
        cout << b[i] << " ";
    }
    cout << endl;
}

int main()
{
    int arr[5] = {1, 2, 3, 4, 5};
```

```

    Print(arr);
    return 0;
}

// 模板
#include <iostream>
using namespace std;

template<typename T, int len>    //len 是非类型常量
void Print(T (&b)[len])
{
    for (int i = 0; i < len; ++i)
    {
        cout << b[i] << " ";
    }
    cout << endl;
}

int main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    char crr[6] = {'a', 'b', 'c', 'd', 'e', 'f'};
    Print(arr);
    Print(crr);
    return 0;
}

```

## 9.指针加减计算要注意什么？

指针加减本质是对其所指地址的移动，移动的步长跟指针的类型相关，指针每移动一位，它实际跨越的内存间隔是指针类型的长度

## 10.继承机制中对象之间如何转换？指针和引用之间如何转换？

### 向上类型转换：

派生类指针或引用转换为基类指针或引用称为向上类型转换，向上类型转换会自动进行，而且向上类型转换是安全的。

### 向下类型转换：

将基类指针或引用转换为派生类指针或引用被称为向下类型转换，向下类型转换不会自动进行，因为一个基类对应几个派生类，所以向下类型转换时不知道对应哪个派生类，所以在向下类型转换时必须加动态类型识别技术，用dynamic\_cast进行向下类型转换。

## 11.函数指针

### (1) 什么是函数指针？

1. 函数指针本质是一个指针，该指针存储的地址指向一个函数；
2. 函数的定义存储与代码段，每个函数在代码段中有一个入口地址，函数指针就是指向代码段中函数入口地址的指针

## (2) 函数指针的定义和初始化

```
//1. 函数指针的定义
int (*p)(int, int);
int (*p)(int a, int b);

//2. 函数指针的初始化
指针名 = 函数名
指针名 = &函数名

//3. 函数指针的使用
p(a,b)
(*p)(a,b)
```

## (3) 为什么要使用函数指针？

在一个函数中，我们希望根据相同的形参，根据传递的实参不同，而产生不同的函数调用效果。

回调函数使用的就是函数指针

函数的类型是由其返回的数据类型和其参数列表共同决定的，而函数的名称则不是其类型的一部分

## (4) 区别于指针函数

指针函数是一个函数，返回值是一个指针。

注意：在指针函数中不要返回局部变量的指针，可以返回静态局部变量的指针

# 12.你知道回调函数吗？它的作用？

## (1) 什么是回调函数？

一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，就说这是回调函数。

## (2) 为什么要使用回调函数？

回调函数的作用：解耦！

1. 将函数指针传入库函数，只需要改变传入库函数的参数（传入的函数指针）就可以实现不同的功能，且不需要修改库函数的实现，这就是解耦
2. 主函数和回调函数在同一层，而库函数在另一层，库函数对我们不可见，不能改变库函数的实现，想要完成不同的功能只能通过这种形式

面向对象的程序设计思想，设计模式中要求的模块独立性，高内聚低耦合等特性

## 四、内存

### 1.内存对齐问题？

#### 什么是内存对齐？

计算机系统对基本类型数据在内存中存放的位置有所限制，要求这些数据的首地址的值是某个数k（通常为4或8）的倍数，这就是所谓的内存对齐。

#### 为什么要进行内存对齐？

1. 提高cpu的访问速度：cpu不是按字节块来存取内存，而是字节的整数倍（2、4、8、16、32bytes），存取单位称为内存存取粒度。对于未对齐的内存，处理器可能需要访问两次内存才能将数据完全读出，而对于对齐的内存，处理器可能只需要一次即可（内存从0开始存取，首地址为1的int，需要两次完全存取；而内存对齐之后，只需要一次）
2. 便于移植：不是所有的硬件平台都支持访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

#### 结构体中的数据是如何存储的？

1. 结构体内成员按照声明顺序存储，第一个成员地址和整个结构体地址相同。
2. 按结构体中size最大的成员对齐（对齐单位）（若有double成员，按8字节对齐），可以通过 `#pragma pack(n)` 改变系统的改变对齐单位，取二者最小值

#### 结构体中的内存对齐规则？

1. 结构体第一个成员的偏移量（offset）为0，以后每个成员相对于结构体首地址的offset都是**该成员大小与对齐单位中较小那个的整数倍**。（两变量之间的内存，编译器会添加填充字节）
2. 结构体的总大小为有对齐单位的整数倍（多出来的内存空间，编译器会在最末一个成员之后加上填充字节）
3. c++11以后引入两个关键字 `alignas` 与 `alignof`。其中 `alignof` 可以计算出类型的对齐方式，`alignas` 可以指定结构体的对齐单位，与上述规则得出的对齐单位求最小。

```
#include <iostream>
#include <stddef.h>

// #pragma pack(4) // 1.改变系统的对齐单位，具体结构体需要取最小

using namespace std;
struct S
// struct alignas(4) S // 2.alignas可以指定结构体的对齐单位
{
    int x;
    char y;
    int z;
    double a;
};

int main()
{
    // 3. alignof求的内存对齐单位
    cout << alignof(S) << endl; // 8
    // 4. offsetof宏求结构成员相对于结构开头的字节偏移量
    cout << offsetof(S, x) << endl; // 0
    cout << offsetof(S, y) << endl; // 4
}
```

```

cout << offsetof(S, z) << endl; // 8
cout << offsetof(S, a) << endl; // 16

// 5. sizeof结构体总大小
cout << sizeof(S) << endl; // 24
return 0;
}

```

## 2.如何用代码判断大小端存储?

在内存中地址都是从低地址开始的，所以大端序和小端序讨论的是，数据的高字节还是低字节在低地址

大端存储：字数据的高字节存储在低地址中

小端存储：字数据的低字节存储在低地址中

例如：32bit的数字0x12345678

补充：

1byte=8bit

16进制的一位占4个bit ----> 16进制的两位占1个byte

所以在Socket编程中，往往需要将操作系统所用的小端存储的IP地址转换为大端存储，这样才能进行网络传输，即网络是大端序

小端模式中的存储方式为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

大端模式中的存储方式为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

怎么判断主机是大端序还是小端序？

1. 使用强制类型转换
2. 巧用union联合体

```

#include <iostream>
using namespace std;

//union联合体的重叠式存储，endian联合体占用内存的空间为每个成员字节长度的最大值
union endian
{
    int a;
    char ch;
};
int main()
{
    // 2.使用联合体
    endian value;
}

```

```

value.a = 0x1234;
//a和ch共用4字节的内存空间
if (value.ch == 0x12)
    cout << "big endian"<<endl;
else if (value.ch == 0x34)
    cout << "little endian"<<endl;

// 2.强制类型转换
int a = 0x1234;
//由于int和char的长度不同，借助int型转换成char型，只会留下低地址的部分
char c = (char)(a);
if (c == 0x12)
    cout << "big endian" << endl;
else if(c == 0x34)
    cout << "little endian" << endl;
}

```

### 3.什么是内存泄露，如何检测与避免

#### 内存泄露

一般我们常说的内存泄露是指**堆内存的泄漏**。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)内存块，使用完后必须显式释放的内存。一般使用malloc、realloc、new等函数从堆中分配内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了

#### 避免内存泄露的几种方式

1. 计数法：使用new或者malloc时，让该数+1，delete或free时，该数-1，程序执行完打印这个计数，如果不为0则表示存在内存泄露（new/delete、malloc/free成对出现）
2. 一定要将基类的析构函数声明为**虚函数**
3. 对象数组的释放一定要用**delete []**
4. 使用智能指针

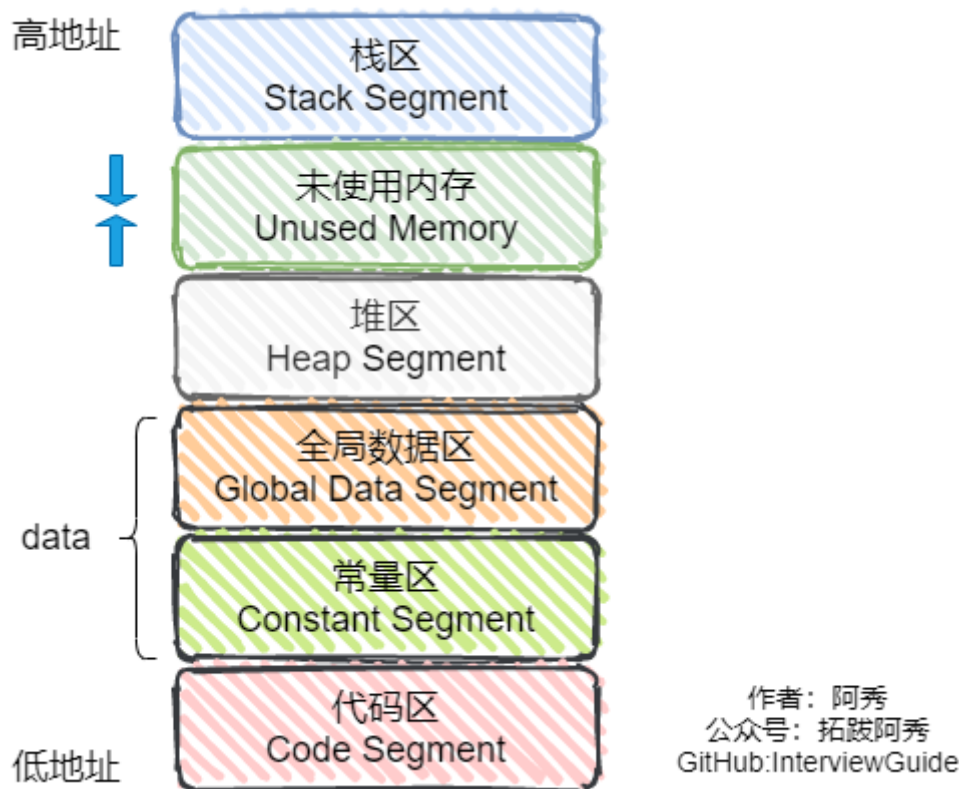
#### 检测工具

1. Linux下可以使用**Valgrind工具**
2. Windows下可以使用**CRT库**

### 4.简要说明C++的内存分区（管理）

C++中的内存分区，从高地址到低地址分别是栈、堆、全局/静态存储区、常量存储区和代码区，此外还有自由存储区，自由存储区是对new/delete动态分配和释放空间的抽象概念，自由存储区和堆比较像，但不等价。





**栈：**在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限

**堆：**由应用程序控制分配和释放的内存块。由new分配，delete释放，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收

**全局/静态存储区：**全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量和静态变量又分为初始化的和未初始化的，在C++里面没有这个区分了，它们共同占用同一块内存区，在该区定义的变量若没有初始化，则会被自动初始化，例如int型变量自动初始为0

**常量存储区：**这是一块比较特殊的存储区，这里面存放的是常量，不允许修改

**代码区：**存放函数体的二进制代码，包括普通函数和类成员函数

## 5.什么是内存池，如何实现

内存池（Memory Pool）是一种**内存分配**方式。通常我们习惯直接使用new、malloc等申请内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性能。内存池则是在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块，若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升。

## 五、类

### 1.C++中的重载、重写（覆盖）和隐藏的区别

#### (1) 重载 (overload)

重载是指在同一范围定义中的同名成员函数才存在重载关系。

主要特点是函数名相同，参数类型或数目有所不同，返回值类型不做要求。

重载和函数成员是否是虚函数无关。

#### (2) 重写（覆盖） (override)

重写指的是在派生类中覆盖基类中的同名函数，**重写就是重写函数体，要求基类函数必须是虚函数且重写函数与基类虚函数参数个数和参数类型都相同，返回值类型也要相同**

补充：

基类指针指向派生类对象时，基类指针可以直接调用到派生类的覆盖函数，也可以通过 :: 调用到基类被覆盖的虚函数；而基类指针只能调用基类的被隐藏函数，无法识别派生类中的隐藏函数。

**重载与重写的区别：**

1. 重写是父类和子类之间的垂直关系，重载是不同函数之间的水平关系
2. 重写要求参数列表和返回值类型相同，重载则要求参数列表不同，返回值不要求
3. 重写关系中，调用方法根据对象类型决定，重载根据调用时实参与形参表的对应关系来选择函数体

#### (3) 隐藏 (hide)

隐藏指的是某些情况下，派生类中的函数屏蔽了基类中的同名函数，包括以下情况：

1. 两个函数参数相同，但是基类函数不是虚函数。**和重写的区别在于基类函数是否是虚函数。**

```
//父类
class A{
public:
    void fun(int a){
        cout << "A中的fun函数" << endl;
    }
};

//子类
class B : public A{
public:
    //隐藏父类的fun函数
    void fun(int a){
        cout << "B中的fun函数" << endl;
    }
};

int main(){
    B b;
    b.fun(2);           //调用的是B中的fun函数
    b.A::fun(2);        //调用A中fun函数
    return 0;
}
```

2. 两个函数参数不同，无论基类函数是不是虚函数，都会被隐藏。和重载的区别在于两个函数不在同一个类中

```
//父类
class A{
public:
    virtual void fun(int a){
        cout << "A中的fun函数" << endl;
    }
};

//子类
class B : public A{
public:
    //隐藏父类的fun函数
    virtual void fun(char* a){
        cout << "A中的fun函数" << endl;
    }
};

int main(){
    B b;
    b.fun(2);          //报错，调用的是B中的fun函数，参数类型不对
    b.A::fun(2);       //调用A中fun函数
    return 0;
}
```

## 2.浅拷贝和深拷贝的区别

### 浅拷贝

浅拷贝只是拷贝一个指针，并没有新开辟一个地址，拷贝的指针和原来的指针指向同一块地址，如果原来的指针所指向的资源释放了，那么再释放浅拷贝的指针的资源就会出现错误。

### 深拷贝

深拷贝不仅拷贝值，还开辟出一块新的空间用来存放新的值，即使原先的对象被析构掉，释放内存了也不会影响到深拷贝得到的值。在自己实现拷贝赋值的时候，如果有指针变量的话是需要自己实现深拷贝的。

## 3.public，protected和private访问和继承权限/public/protected/private的区别？

访问权限：public > protected > private

1. public的变量和函数在类的内部外部都可以访问。
2. protected的变量和函数只能在类的内部和其派生类中访问。
3. private修饰的元素只能在类内访问。

访问权限	外部	派生类	内部
public	✓	✓	✓
protected	✗	✓	✓
private	✗	✗	✓

继承权限：

- 1. 派生类继承自基类的成员权限有四种状态：public、protected、private、不可见
- 2. 派生类对基类成员的访问权限取决于两点：一、继承方式；二、基类成员在基类中的访问权限
- 3. 派生类对基类成员的访问权限是取以上两点中的更小的访问范围（除了 private 的继承方式遇到 private 成员是不可见外，因为基类的private权限的成员在除类内部外全部不可见）。

基类成员	private	protected	public	private	protected	public	private	protected	public
派生方式	private			protected			public		
派生类中	不可见	private	private	不可见	protected	protected	不可见	protected	public
外部	不可见	不可见	不可见	不可见	不可见	不可见	不可见	不可见	可见

4.C++有哪几种的构造函数

- 1. 默认构造函数：没有参数
- 2. 初始化构造函数：有参数和参数列表
- 3. 拷贝构造函数：复制本类的对象
- 4. 转换构造函数：将其他类型的变量（只有一个），隐式转换为本类对象
- 5. 移动构造函数：解决拷贝构造函数时间开销大的问题

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

class Student{
public:
    Student(){// 1.默认构造函数，
        this->age = 20;
        this->num = 1000;
    };

    Student(int a, int n):age(a), num(n){}; // 2.初始化构造函数

    Student(const Student& s){// 3.拷贝构造函数
        this->age = s.age;
        this->num = s.num;
    };

    Student(int r){    // 4.转换构造函数,形参是其他类型变量，且只有一个形参
        this->age = r;
        this->num = 1002;
    };
    ~Student(){ }
public:
    int age;
    int num;
};

class demo{
public:
    demo(demo &&de) {    // 5.移动构造函数
        num = de.num;
```

```
de.num = NULL;
cout<<"move constructor"<<endl;
}
private:
    int *num;
};
```

## 5.什么情况下会调用拷贝构造函数

1. 用类的一个实例化对象去初始化另一个对象的时候
2. 函数的参数是类的对象时（非引用传递），根据传入的实参产生临时对象，再用拷贝构造去初始化这个临时对象，在函数中与形参对应，函数调用结束后析构临时对象
3. 函数的返回值是函数体内局部对象的类的对象时，此时虽然发生（Named return Value优化）NRV优化，但是由于返回方式是值传递，所以会在返回值的地方调用拷贝构造函数（用一个变量接收函数返回值，会根据函数内部的临时变量去调用拷贝构造去初始化这个变量）

注意：在linux g++环境下，进行nrv优化，管值返回方式还是引用方式返回的方式都不会发生拷贝构造函数；而Windows + VS2019在值返回的情况下发生拷贝构造函数，引用返回方式则不发生拷贝构造函数

补充：

在一个类返回值的函数中，函数返回之前会调用拷贝构造将函数中的局部类对象进行拷贝到一个临时对象中，然后返回临时对象（实际上是给函数增加一个引用参数，用局部类对象调用拷贝构造函数来初始化这个引用参数）。在函数外部，用一个变量来接收这个临时对象，会再调用一次拷贝构造

NRV（Named **return value**）优化是用增加的那个引用来从当函数内局部变量的角色，从而减少一次拷贝构造

## 6.如何禁止程序自动生成拷贝构造函数和重载赋值运算符？

1. 声明一个私有拷贝构造函数，而不真的定义它，类成员函数和友元函数仍能调用拷贝构造函数，此时出现链接错误（链接阶段）
2. 设计一个基类，在基类中声明私有拷贝构造函数，派生类中编译器将不会自动生成拷贝构造函数，由于base类中该函数是私有的，因此，派生类将阻止编译器执行相关的操作
3. 重载赋值运算符与拷贝构造函数一样

## 7.什么是类的继承？

### (1) 类与类之间的关系

1. has-A：包含关系，一个类的成员变量是另一个已经定义好的类对象；
2. use-A，使用关系，通过类成员函数传参的方式来实现；
3. is-A，继承关系，关系具有传递性；

### (2) 继承的概念

所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称为子类或者派生类，被继承的类称为父类或者基类

### (3) 继承的特点

子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，父类指针可以指向子类对象（引子类对象可以当作父类对象的引用）；

### (4) 继承方式

public、protected、private

## 8.知道C++中的组合吗？它与继承相比有什么优缺点吗？

**继承：**

继承是is a 的关系，比如说Student继承Person,则说明Student is a Person。

优点：子类可以重写父类的方法来方便地实现对父类的扩展。

缺点：

1. 父类的内部细节对子类是可见的
2. 子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为（虚函数除外）
3. 继承是一种高耦合关系，违背了面向对象的思想

**组合：**

组合是has a的关系，一个类的成员变量是另一个已经定义好的类对象

优点：

1. 当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的（内部细节不可见）
2. 当前对象与包含的对象是低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码
3. 当前对象可以在运行时动态的绑定所包含的对象（包含的可以是一个父类指针，运行时确定是哪个具体的对象）

缺点：

1. 容易产生过多的对象
2. 为了能组合多个对象，必须仔细对接口进行定义

## 9.类成员初始化方式？构造函数的执行顺序？为什么用成员初始化列表会快一些？

**(1) 类成员初始化方式：**

1. 赋值初始化，通过在函数体内进行赋值初始化
2. 列表初始化，在类的构造函数中，不在函数体内对成员变量赋值，而是在构造函数的花括号前面使用冒号和初始化列表赋值

赋值初始化在所有的数据成员被分配内存空间后才进行的；列表初始化是给数据成员分配内存空间时就进行初始化；先进行列表初始化，后进行赋值初始化。

**(2) 一个派生类构造函数的执行顺序（也是构造函数中的扩展过程）如下：**

1. 虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）
2. 基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）
3. 类类型的成员对象的构造函数（按照声明的顺序）
4. 派生类自己的构造函数

**析构顺序相反**

补充:

在执行完上述的基类的构造函数之后, 按顺序执行:

1. 对象的vp<sub>tr</sub>被初始化;
  2. 如果有成员初始化列表, 将在构造函数体内扩展开来, 这必须在vp<sub>tr</sub>被设定之后才做;
  3. 执行程序员所提供的代码;
- // 这些算在派生类自己的构造函数中

### (3) 初始化列表的方式效率更高:

对于类成员变量是另一个类的实例化对象的情况下 (类B的成员对象是类A的实例化对象), 赋值初始化之前会先调用A类的默认构造函数进行初始化, 然后在初始化构造函数体内会再调用一次A类初始化构造函数; 而初始化列表方式, 只调用一次A类的初始化构造函数。

总结:

1. 初始化列表只调用一次构造函数 (初始化构造函数), 赋值初始化调用两次构造函数 (默认构造 + 初始化构造函数)
2. 初始化列表先于函数体执行

```
#include <iostream>
using namespace std;
class A
{
public:
    A()
    {
        cout << "默认构造函数A()" << endl;
    }
    A(int a)
    {
        value = a;
        cout << "初始化构造函数A(int "<<value<<")" << endl;
    }
    A(const A& a)// 不会被调用
    {
        value = a.value;
        cout << "拷贝构造函数A(A& a): " <<value << endl;
    }
    int value;
};

class B
{
public:
    B() : a(1)
    {
        b = A(2);
    }
    A a;
    A b;
};

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

```
// 执行结果：

// 构造a
// 初始化构造函数A(int 1)

// 构造b
// 默认构造函数A()
// 初始化构造函数A(int 2)
```

## 10.有哪些情况必须用到列表初始化？作用是什么？

四种情况必须使用列表初始化：

1. 初始化引用成员
2. 初始化常量成员（const，也可以声明时初始化）
3. 调用基类的构造函数，而它拥有一组参数
4. 调用成员类的构造函数，而它拥有一组参数

列表初始化的作用：

1. 编译器会——操作初始化列表，以适当的顺序在构造函数之内安插初始化操作，并且在任何显示用户代码之前；
2. list中的项目顺序是由类中的成员声明顺序决定的，不是由初始化列表的顺序决定的；

## 11.如果想将某个类用作基类，为什么该类必须定义而非声明？

派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类必须知道他们是什么，所以必须定义而非声明。

## 12.说说移动构造函数

1. 移动构造的初衷是用一个之后不再使用的对象a来初始化当前对象b，直接使用a的空间（指针成员指向的空间），从而避免新的空间分配，大大降低构造的成本
2. 拷贝构造中，对于指针成员变量，需要采用深拷贝的方式进行构造，采用深拷贝的原因是防止两个指针指向同一片内存空间（其中一个指针将其释放，另一指针将变成悬空指针，如果在进行使用/delete将报错）。而移动构造采用的是浅拷贝，避免上述问题的方法是将a对象中的指针置空（反正a之后不再使用，且析构的时候有判空操作）
3. 拷贝构造的参数是一个左值引用；移动构造函数的参数是一个右值引用（右值/将亡值），move可以将一个左值变成一个右值

## 13.类如何实现只能静态分配和只能动态分配

建立类的对象有两种方式：

1. 静态建立，静态建立一个类对象，就是由编译器为对象在栈空间中分配内存；
2. 动态建立，`A *p = new A()`；动态建立一个类对象，就是使用new运算符为对象在堆空间中分配内存。这个过程分为两步，第一步执行operator new()函数，在堆中分配一块内存；第二步调用类构造函数构造对象；

实现只能静态分配：



只能静态分配，也就是禁止动态分配，因此只要限制new运算符就可以实现，可以将new运算符重载位private属性（delete也是）

#### 实现只能动态分配：

如果对外提供构造函数的接口，那么一定可以使用静态分配，为了禁用静态分配，可以不对外提供构造函数的接口，但是动态分配也需要构造函数。所以可以将构造、析构函数设为protected属性，再用子类来动态创建（类外创建对象不能控制，子类可以控制）

## 14.何阻止一个类被实例化？有哪些方法？

将类定义为抽象基类或者将构造函数声明为private，不允许类外部创建类对象，只能在类内部创建对象

## 15.结构体变量比较是否相等

重载了“==”操作符

```
struct foo {
    int a;
    int b;

    bool operator==(const foo& rhs) // 操作运算符重载
    {
        return( a == rhs.a) && (b == rhs.b);
    }
};
```

## 16.你知道重载运算符吗？

91

## 17.如果有一个空类，它会默认添加哪些函数？

```
1) Empty(); // 默认构造函数
2) Empty( const Empty& ); // 拷贝构造函数
3) ~Empty(); // 析构函数
4) Empty& operator=( const Empty& ); // 赋值运算符重载
```

## 18.设计一个能够计算类的对象个数的类？

1. 为类设计一个static静态变量count作为计数器，类定义结束后初始化count=0;
2. 构造函数中让count+1，析构函数中让count-1

## 19.友元

友元提供了不同类的成员函数之间、类的成员函数和一般函数之间进行数据共享的机制。

通过友元，非成员函数或另一个类的成员函数可以访问类中的所有成员，包括 public、protected、private 属性的。

友元的正确使用能提高程序的运行效率，但同时也破坏了类的封装性和数据的隐藏性，导致程序可维护性变差。

## (1) 友元函数

// 1.友元函数是普通函数

```
class A {
public:
    friend void set_show(int x, A& a); // 友元函数声明

private:
    int data;
};

void set_show(int x, A& a) { // 友元函数定义
    a.data = x;
    cout << a.data << endl;
}
```

// 2.友元函数是类成员函数

// 1.B中遇到A了，需要提前声明

// 提前声明，告诉编译器有A这个类，但是想使用类中的具体成员，需正式声明之后

```
class A;
```

// 2.此处编译会报错

// 类只有在正式声明之后才能创建对象，因为创建对象时要为对象分配内存，在正式声明类之前，编译器无法确定应该为对象分配多大的内存

// 但提前声明之后可以使用该类去定义指针或引用，因为指针变量和引用变量本身的大小是固定的，与它所指向的数据的大小无关

```
A a;
```

```
class B {
public:
    // 3.声明和实现分开了，并且将A的声明放在二者之间
    // 因为编译器是从上到下编译代码，set_show中用到了A的成员变量data
    // 如果不知道A声明的具体内容，就不能确定A中是否有data（类的声明中指明了类中有哪些成员）
    void set_show(int x, A& a);
};
```

// 4.正式声明

// 告诉编译器A这类中具体的成员有哪些

```
class A {
public:
    friend void B::set_show(int x, A& a);

private:
    int data;
};

void B::set_show(int x, A& a) {
    a.data = x;
    cout << a.data << endl;
}
```

// 一个函数可以是多个类的友元函数，但是每个类中都要声明这个函数

## (2) 友元类

```
// 1. 友元类的所有成员函数都是另一个类的友元函数
// 2. 需要在另一个类中进行声明
// 3. 友元关系不能被继承
// 4. 友元关系是单向的，A是B的友元类，B不一定是A的友元类
// 5. 友元关系不具有传递性，类B是类A的友元，类C是B的友元，类C不一定是类A的友元
// 6. 除非有必要，一般不建议把整个类声明为友元类，而只将某些成员函数声明为友元函数，这样更安全一些
class A {
public:
    friend class B; // 友元类声明

private:
    int data;
};

class B { // 友元类定义
public:
    void set_show(int x, A& a) {
        a.data = x;
        cout << a.data << endl;
    }
};
```

## 20. 类的实例化对象存储空间大小？

1. 空类的对象占用1个字节的存储空间，而不是0个字节（C++中要求对于类的每个实例都必须有独一无二的地址）。当该空白类作为基类时，该类的大小就优化为0了，子类的大小就是子类本身的大小。这就是所谓的空白基类最优化。
2. 类的静态成员变量不占用类的存储空间（类里的静态成员变量在全局数据区中分配空间，不占用类的存储空间，全局只有一份，不会随着类的实例化存储在每个对象里）
3. 类的成员函数（非虚函数）不占用类的存储空间（成员函数（包括构造和析构函数、静态函数）编译后存放在代码区，不占用类的存储空间）
4. 类中的虚函数占用类的存储空间，但所占的空间不会随着虚函数的个数增长（有虚函数的类实例化的时候，会有一个虚指针vpPtr指向虚函数表vtbl，而虚函数表里就存储着类中定义的所有虚函数）

总结：类对象所占用的空间由3部分组成：

1. 类的非静态成员变量大小（包括继承来的成员变量）
2. 内存对齐另外分配的空间大小，类内的数据也是需要进行内存对齐操作的；
3. 虚函数带来的虚指针内存开销

## 21. 关于this指针你知道什么？全说出来

1. this是C++中的一个关键字，也是一个const 指针，指向当前对象，通过它可以访问当前对象的所有成员，this指针并不是对象本身的一部分
2. this指针实际上是类成员函数的一个形参（隐式形参），在调用成员函数时将对象的地址作为实参传递给this。静态成员函数中没有this
3. this作为隐式形参，本质上是成员函数的局部变量，对象在调用成员函数时才给this赋值。只能在成员函数内部使用
4. this是成员函数和成员变量关联的桥梁

## this指针的使用

1. 在类的非静态成员函数中返回类对象本身，return \*this;
2. 当形参数与成员变量名相同时用于区分，如this->n = n （不能写成n = n）

## 在成员函数中调用delete this会出现什么问题？对象还可以使用吗？

1. 在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码内容，类的成员函数单独放在代码段中。
2. 在调用成员函数时，隐含传递一个this指针，让成员函数知道当前是哪个对象在调用它。
3. 当调用delete this（前提是当前对象是new出来的）时，类对象的内存空间（存储的只有成员变量和虚函数表指针）被释放（之前会调用析构函数）。
4. 在delete this之后进行的其他任何函数调用，只要不涉及到this指针的内容，都能够正常运行。一旦涉及到this指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。（等同于delete p之后，用p调用函数没有问题，但是访问成员变量将会出现不可预期的问题）

为什么是不可预期的问题？

delete this之后不是释放了类对象的内存空间了么，那么这段内存应该已经还给系统，不再属于这个进程。

照这个逻辑来看，应该发生指针错误，无访问权限之类的令系统崩溃的问题才对啊？

这个问题牵涉到操作系统的内存管理策略。delete this释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是可以访问的，你可以加上100，加上200，但是其中的值却是不确定的。当你获取数据成员，可能得到的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

## 如果在类的析构函数中调用delete this，会发生什么？

会导致堆栈溢出。

delete会先调用析构函数，然后释放内存。delete this会去调用本对象的析构函数，而析构函数中又调用delete this，形成无限递归，造成堆栈溢出，系统崩溃。

## 22.为什么基类的析构函数一般写成虚函数

由于类的多态性，父类指针可以指向子类对象，如果删除该基类指针，就会调用派生类的析构函数，然后派生类的析构函数又会调用基类的析构函数，这样整个派生类的对象完全被释放。

如果析构函数不被声明为虚函数，编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数，而不调用派生类的析构函数，这样就造成派生类对象析构不完全，造成内存泄漏。

析构函数可以写成纯虚函数，含有纯虚函数的类是抽象类，不能被实例化，需要在派生类中重写基类的纯虚函数

```
// 不把析构函数写成虚函数的情况
#include <iostream>
using namespace std;

class Parent{
public:
    Parent(){
        cout << "Parent construct function" << endl;
    };
    ~Parent(){
        cout << "Parent destructor function" << endl;
    }
};
```

```
};

class Son : public Parent{
public:
    Son(){
        cout << "Son construct function" << endl;
    };
    ~Son(){
        cout << "Son destructor function" << endl;
    }
};

int main()
{
    Parent* p = new Son();
    delete p;
    p = NULL;
    return 0;
}
//运行结果:
//Parent construct function
//Son construct function
//Parent destructor function
```

```
// 把析构函数写成虚函数的情况
#include <iostream>
using namespace std;

class Parent{
public:
    Parent(){
        cout << "Parent construct function" << endl;
    };
    virtual ~Parent(){
        cout << "Parent destructor function" << endl;
    }
};

class Son : public Parent{
public:
    Son(){
        cout << "Son construct function" << endl;
    };
    ~Son(){
        cout << "Son destructor function" << endl;
    }
};

int main()
{
    Parent* p = new Son();
    delete p;
    p = NULL;
    return 0;
}
```

```
//运行结果：
//Parent construct function
//Son construct function
//Son destructor function
//Parent destructor function
```

构造时，先父后子  
析构时，先子后父

## 23.构造函数能否声明为虚函数？

**构造函数不能被声明为虚函数：**

1. 从内存空间角度：每一个含有虚函数的类都存在一个虚函数表，用于存储虚函数的函数指针。每一个类对象都含有一个指向该类虚表的指针，存储在对象的内存空间。如果构造函数为虚函数，就需要使用虚表进行调用，但是类对象还没有实例化，没有内存空间分配，自然也就无法调用虚构造函数
2. 从使用角度：虚函数主要用于实现运行时多态，即通过父类指针或引用指向子类对象，进而用父类指针调用子类函数。但是构造函数是通过创建对象时自动调用的，不可能通过父类的指针或者引用去调用，所以构造函数没有必要是虚函数

## 24.基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间

**虚函数表的特征：**

1. 虚函数表全局共享，只有一个，编译时完成构造
2. 虚函数表类似数组，存储的是虚函数指针，类对象中的vptr指针指向虚函数表，不是函数，不是代码，不可能存在代码段
3. 类中虚函数的个数在编译期即可确定，即虚函数表的大小在编译期确定，不用动态分配，不会再堆区

C++中虚函数表位于只读数据段（.rodata），也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。虚函数表指针在构造函数中初始化，存放在内存中的最前面

## 25.构造函数、析构函数、虚函数可否声明为内联函数

都可以声明为内联函数，语法上没有问题，因为inline只是对编译器的一个建议，编译器并不一定真的内联

1. 构造函数和析构函数声明为内联函数是没有意义的，编译器并不真正对其进行内联操作。因为编译器在构造和析构函数中还会添加其他额外操作，导致构造和析构函数没有看上去那么精简。另外，class中的函数是默认内联的，所以再将构造函数和析构函数声明为内联函数是没有什么意义的。
2. 虚函数是否内联，取决于调用该虚函数的指针是否具有多态性。父类指针指向派生类对象时，因为是在运行期确定调用哪个虚函数的，而inline是在编译器决定的，所以不会内联展开；当不具有多态性时，会内联展开，前提时函数不复杂

## 26.构造/析构函数的作用，如何起作用？

1. 构造函数起到初始化成员变量的作用；编译器自动调用，可以重载
2. 析构函数与构造函数的作用相反，用于释放对象分配的内存空间；析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数；当析构一个对象时，编译器自动调用析构函数

### 什么时候调用析构函数？

1. 对象生命周期结束，被销毁时；
2. delete指向对象的指针时，或delete指向对象的基类类型指针，而其基类虚函数是虚函数时；
3. 对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用

## 27.构造函数和析构函数可以调用虚函数吗，为什么

1. 可以调用，但不提倡这样做，达不到想要的效果
2. 派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。同样，进入基类析构函数时，对象也是基类类型。因此在构造函数或析构函数中调用虚函数，则运行的是构造函数或析构函数自身类型定义的版本；

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

class Base
{
public:
    Base()
    {
        Function1();
    }

    virtual void Function1()
    {
        cout << "Base::构造" << endl;
    }
    virtual void Function2()
    {
        cout << "Base::析构" << endl;
    }
    virtual ~Base()
    {
        Function2();
    }
};

class A : public Base
{
public:
    A()
    {
        Function1();
    }

    virtual void Function1()
    {
        cout << "A::构造" << endl;
    }
    virtual void Function2()
    {
        cout << "A::析构" << endl;
    }
}
```

```

    ~A()
    {
        Function2();
    }
};

int main()
{
    Base* a = new Base;
    delete a;
    cout << "-----" << endl;
    Base* b = new A; //语句1
    delete b;
}

//输出结果
//Base::构造
//Base::析构
//-----
//Base::构造
//A::构造
//A::析构
//Base::析构
// 都是调用的自己类中的虚函数，没有达到多态的效果

```

## 28.构造函数的几种关键字

### (1) default

default关键字可以显式要求编译器生成默认构造函数（一般情况下，提供了有参构造，编译器就不会再生成默认构造函数了），防止在调用相关构造函数时函数没有定义而报错

```

#include <iostream>
using namespace std;

class CString
{
public:
    CString() = default; //语句1
    //构造函数
    CString(const char* pstr) : _str(pstr) {}

    //析构函数
    ~CString() {}
public:
    string _str;
};

int main()
{
    auto a = new CString(); // 如果没有语句1，这里会报错

    system("pause");
    return 0;
}

```



## (2) delete

delete关键字可以删除构造函数、赋值运算符等默认提供的方法，这样在使用的时候会得到友善的提示

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

class CString
{
public:
    void* operator new(size_t size) = delete; //这样不允许使用new关键字

    //析构函数
    ~CString() {}

private:
    void* operator new(size_t size);
};

int main()
{
    auto a = new CString(); //语句1

    system("pause");
    return 0;
}
```

## (3) 0

将虚函数声明成纯虚函数：

```
class CString
{
public:
    virtual void function()=0;

};
```

## 29.构造函数、拷贝构造函数和赋值操作符的区别

### 构造函数

对象不存在，没用别的对象初始化，在创建一个新的对象时调用构造函数

### 拷贝构造函数

对象不存在，使用别的已经存在的对象来进行初始化

### 赋值运算符

对象存在，用别的对象给它赋值，这属于重载“=”号运算符的范畴，“=”号两侧的对象都是已存在的

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

```

class A
{
public:
    A()
    {
        cout << "我是构造函数" << endl;
    }
    A(const A& a)
    {
        cout << "我是拷贝构造函数" << endl;
    }
    A& operator = (A& a)
    {
        cout << "我是赋值操作符" << endl;
        return *this;
    }
    ~A() {};
};

int main()
{
    A a1;          // 调用构造函数
    A a2 = a1;     // 调用拷贝构造函数
    a2 = a1;       // 调用赋值操作符
    return 0;
}
//输出结果
//我是构造函数
//我是拷贝构造函数
//我是赋值操作符

```

### 30.拷贝构造函数和赋值运算符重载的区别？

1. 调用拷贝构造函数会产生新的对象，调用赋值运算符重载不会产生新的对象
2. 形参传递是调用拷贝构造函数，但出现=的地方不一定调用赋值运算符重载
3. 类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符

```

Student s;
Student s1 = s;    // 调用拷贝构造函数
Student s2;
s2 = s;           // 赋值运算符操作

```

### 31.什么情况会自动生成默认构造函数（前提是没有定义其他的构造函数）

#### (1) 常见误区：

1. 任何没有定义default constructor，都会被合成default constructor。这句话是错的，只有在需要的时候才会被合成，这里的默认构造函数不包括trivial constructor（啥都不干）
2. 合成的默认构造函数会初始化类的数据成员为默认值，这句话也是错的。合成的默认构造函数中，只有基类子对象和成员类对象会被初始化。所有其他的非静态数据成员都不会被初始化。

## (2) 下面四种情况会合成non-trivial constructor:

1. 含有类对象数据成员，该类对象类型有默认构造函数

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

class B
{
public:
    A a;    //含有A类的成员
    int num;
};

void Test()
{
    B b;    //需要调用A类的默认构造函数
           //此时并没有给出显示定义的构造函数，但是编辑器会自动生成一个默认（无参）的构造函数
    cout << b.num << endl;
}

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

// 编辑器在给B类生成无参的默认构造函数的初始化列表的位置调用A类的构造函数，完成对a对象的初始化
```

2. 基类带有默认构造函数的派生类

编译器合成的默认构造函数将根据基类声明顺序调用上层的基类默认构造函数

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

class Derived :public Base
{
public:
    int d;
};

void Test()
{
    Derived d;
}
```

```

}

int main()
{
    Test();
    return 0;
}
// 因为派生类被合成时需要显式调用基类的默认构造函数

```

### 3. 带有虚函数的类

类带有虚函数分两种情况：类本身定义了虚函数 和 类继承的的类中存在虚函数。这种情况下也会合成默认构造函数，原因是含有虚函数的类对象都含有一个虚表指针vptr，编译器需要对vptr设置初值以满足虚函数机制的正确运行，编译器会把这个设置初值的操作放在默认构造函数中。

### 4. 虚继承下

虚继承也会在子类对象中合成一个指向虚基类的指针，因此也要被初始化，所以必须要构造函数，虚基类或者虚继承保证子类对象中只有一份虚基类的对象

自动合成拷贝构造函数的情况同上

## 32.为什么拷贝构造函数必须传引用不能传值？

拷贝构造函数的参数必须使用引用传递。如果拷贝构造函数中的参数不是一个引用，而是一个类对象，值传递的时候会调用该类的拷贝构造函数，从而造成无穷递归地调用拷贝构造函数。

## 33.什么是纯虚函数，与虚函数的区别

1. 虚函数是实现多态的基础。在基类函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据所指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
2. 纯虚函数相当于只是一个接口名，含有纯虚函数的类不能够实例化。纯虚函数首先是虚函数，其次它没有函数体，取而代之的是用“=0”。
3. 既然是虚函数，它的函数指针会被存在虚函数表中，由于纯虚函数并没有具体的函数体，因此它在虚函数表中的值就为0，而具有函数体的虚函数则是函数的具体地址。
4. 一个类中如果有纯虚函数的话，称其为抽象类。抽象类不能用于实例化对象，否则会报错。抽象类一般用于定义一些公有的方法。子类继承抽象类也必须实现其中的纯虚函数才能实例化对象。

```

#include <iostream>
using namespace std;

class Base
{
public:
    virtual void fun1()
    {
        cout << "普通虚函数" << endl;
    }
    virtual void fun2() = 0;
    virtual ~Base() {}
};

```

```

class Son : public Base
{
public:
    virtual void fun2()
    {
        cout << "子类实现的纯虚函数" << endl;
    }
};

int main()
{
    Base* b = new Son;
    b->fun1(); //普通虚函数
    b->fun2(); //子类实现的纯虚函数
    return 0;
}

```

## 34.虚函数的代价是什么？

1. 带有虚函数的类，每一个类会产生一个虚函数表（存储在常量区），用来存储指向虚成员函数的指针，占用内存空间
2. 带有虚函数的类的每一个对象，都会有有一个指向虚表的指针，会增加对象的空间大小；
3. 当虚函数表现多态性的时候不能内联。

内联是在编译期建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。

`inline virtual` 唯一可以内联的时候是：编译器知道所调用的对象是哪个类，这只有在编译器具有实际对象而不是对象的指针或引用时才会发生。

```

#include <iostream>
using namespace std;
class Base
{
public:
    inline virtual void who()
    {
        cout << "I am Base\n";
    }
    virtual ~Base() {}
};

class Derived : public Base
{
public:
    inline void who() // 不写inline时隐式内联
    {
        cout << "I am Derived\n";
    }
};

int main()
{

```

```
// 此处的虚函数who(), 是通过类 (Base) 的具体对象 (b) 来调用的, 编译期间就能确定了, 所以它可以是内联的, 但最终是否内联取决于编译器。
```

```
Base b;  
b.who();
```

```
// 此处的虚函数是通过指针调用的, 呈现多态性, 需要在运行时期才能确定, 所以不能为内联。
```

```
Base *ptr = new Derived();  
ptr->who();
```

```
// 因为Base有虚析构函数 (virtual ~Base() {}), 所以 delete 时, 会先调用派生类 (Derived) 析构函数, 再调用基类 (Base) 析构函数, 防止内存泄漏。
```

```
delete ptr;  
ptr = nullptr;
```

```
system("pause");  
return 0;
```

```
}
```

## 35.静态函数能定义为虚函数吗？常函数呢？说说你的理解

### 静态函数不能是虚函数

静态与非静态成员函数之间有一个主要的区别, 那就是静态成员函数没有this指针。

虚函数依靠vptr和vtable来处理。vptr是一个指针, 在类的构造函数中创建生成, 并且只能用this指针来访问它, 因为它是类的一个成员。vptr指向保存虚函数地址的vtable

对于静态成员函数, 它没有this指针, 所以无法访问vptr。

**虚函数的调用关系: this -> vptr -> vtable -> virtual function**

**常函数可以为虚函数**

## 36.哪些函数不能是虚函数？把你知道的都说一说


**五类函数不能声明为虚函数:**

1. **普通函数 (非成员函数)**: 普通函数不属于类的成员函数, 不具有继承特性, 因此普通函数没有虚函数
2. **构造函数**: 虚表指针的初始化时在构造函数进行的, 而虚函数需要放到虚表中。在调用虚函数前, 必须首先知道虚表指针, 此时矛盾就出来了。
3. **内联函数**: 内联函数属于静态联编, 即内联函数是在编译期间直接展开, 可以减少函数调用的开销, 即是在编译阶段就确定调用哪个函数了。但是虚函数是属于动态联编, 即是在运行时才确定调用哪一个函数。显然这两个是冲突的。(内联函数也可以是虚函数, 见14)
4. **静态函数**: 从技术层面上说, 静态函数的调用不需要传递this指针。但是虚函数的调用需要this指针, 来找到虚函数表, 相互矛盾; 从存在的意义上说, 静态函数的存在时为了让所有类共享, 可以在对象产生之前执行一些操作。与虚函数的作用不是一路的。
5. **友元函数**: 友元函数不属于类的成员函数, 不能被继承。对于没有继承特性的函数没有虚函数的说法

## 37.什么是虚继承

### (1) 多继承的问题

多继承容易产生命名冲突，即使将所有类的成员变量和成员函数都命名为不同的名字，仍然可能出现命名冲突，菱形继承：

 20210104152509818

### (2) 虚继承

为了解决多继承时的**命名冲突和冗余数据**问题，C++ 提出了虚继承，使得在派生类中只保留一份间接基类的成员。

虚继承的目的是让某个类做出声明，承诺愿意共享它的基类。这个被共享的基类称为**虚基类**（Virtual Base Class）。

继承关系中，越靠近派生类的基类优先级越高，即当A和B中有同名成员，那么D中访问的是B中的成员

### (3) 虚继承时的构造函数

1. 普通继承中。派生类构造函数只调用直接基类的构造函数，不能调用间接基类的构造函数；而在虚继承中，派生类除了要调用直接基类的构造函数，还要调用虚基类的构造函数（虚基类时间接基类）

普通继承是由直接基类调用间接基类的构造函数，但是在菱形继承时有两个直接基类调用同一个间接基类的构造函数，这样会使得编译器不知道按照哪个调用初始化，因此将直接基类的构造函数调用视为无效的，转而由派生类调用间接基类的构造函数

2. 普通继承的构造函数调用是按照初始化列表中构造函数的出现顺序依次进行的；而虚继承必须先调用虚基类的构造函数，而不管初始化列表中的顺序

```
#include <iostream>
using namespace std;
//虚基类A
class A{
public:
    A(int a): m_a(a){ }
protected:
    int m_a;
};

//直接派生类B
class B: virtual public A{
public:
    B(int a, int b): A(a), m_b(b){ }
public:
    void display(){
        cout<<"m_a="<<m_a<<"", m_b="<<m_b<<endl;
    }
protected:
    int m_b;
};

//直接派生类C
class C: virtual public A{
public:
```

```

    C(int a, int c): A(a), m_c(c){ }
public:
    void display(){
        cout<<"m_a="<<m_a<<"", m_c="<<m_c<<endl;
    }
protected:
    int m_c;
};

// 派生类D
class D: public B, public C{
public:
    D(int a, int b, int c, int d): A(a), B(90, b), C(100, c), m_d(d){ }
public:
    void display(){
        cout<<"m_a="<<m_a<<"", m_b="<<m_b<<"", m_c="<<m_c<<"", m_d="<<m_d<<endl;
    }
private:
    int m_d;
};

int main(){
    B b(10, 20);
    b.display();

    C c(30, 40);
    c.display();
    D d(50, 60, 70, 80);
    d.display();
    return 0;
}

```

#### (4) 虚继承的原理

1. 继承自虚基类A的类B和类C分别会多出一个成员变量vbptr (virtual base table pointer, 虚基类表指针), 该指针指向虚基类表, 可以被继承
2. 虚基类表有两项, 第一项为vbptr到本类的偏移地址 (就是B、C类中的虚基类表指针与B、C类的偏移地址); 第二项为vbptr到虚基类的偏移地址 (就是B、C类的虚基类表指针与虚基类A的偏移地址)

// 查看C++对象内存布局  
 工具 --> 命令行 --> 开发者命令提示  
 --> 键入cl /d1 reportSingleClassLayout[类名] [cpp文件夹名]

```

// 类A
class A size(4):
    +---
0      | m_a
    +---

// 类B
class B size(12):
    +---
0      | {vbptr}
4      | m_b

```



```

    +---
    +--- (virtual base A)
8      | m_a
    +---

B::$vtable@:
0      | 0
1      | 8 (Bd(B+0)A)
vbi:    class offset o.vbptr o.vbte fvtorDisp
          A      8      0      4 0

// 类C
class C size(12):
    +---
0      | {vbptr}
4      | m_c
    +---
    +--- (virtual base A)
8      | m_a
    +---

C::$vtable@:
0      | 0
1      | 8 (Cd(C+0)A)
vbi:    class offset o.vbptr o.vbte fvtorDisp
          A      8      0      4 0

// 类D
// 继承自类B的虚基类表指针与类B的偏移为0，与类A的偏移为20
// 继承自类C的虚基类表指针与类C的偏移为0，与类A的偏移为12
// 而A类中的成员变量在类C中的地址为20，那么继承自类B的虚基类表指针加上偏移20即可找到
// 继承自类C的虚基类表指针加上偏移12即可找到
class D size(24):
    +---
0      | +--- (base class B)
0      | | {vbptr}
4      | | m_b
    | +---
8      | +--- (base class C)
8      | | {vbptr} // 地址为8，加上到虚基类A的地址偏移12，找到m_a
12     | | m_c
    | +---
16     | m_d
    +---
    +--- (virtual base A)
20     | m_a
    +---

D::$vtable@B@:
0      | 0
1      | 20 (Dd(B+0)A)

D::$vtable@C@:
0      | 0
1      | 12 (Dd(C+0)A)
vbi:    class offset o.vbptr o.vbte fvtorDisp

```

## 38.多继承的优缺点，作为一个开发者怎么看待多继承

### (1) 多继承

一个派生类可以有两个或多个基类。

构造函数的调用顺序按照继承时的顺序，与初始化列表顺序无关

```
#include <iostream>
using namespace std;
//基类
class BaseA {
public:
    BaseA(int a, int b) : m_a(a), m_b(b) {
        cout << "BaseA constructor" << endl;
    }
    ~BaseA() {
        cout << "BaseA destructor" << endl;
    }
protected:
    int m_a;
    int m_b;
};

//基类
class BaseB {
public:
    BaseB(int c, int d) : m_c(c), m_d(d) {
        cout << "BaseB constructor" << endl;
    }
    ~BaseB() {
        cout << "BaseB destructor" << endl;
    }
protected:
    int m_c;
    int m_d;
};

//派生类
class Derived : public BaseA, public BaseB {
public:
    Derived(int a, int b, int c, int d, int e) : BaseA(a, b), BaseB(c, d), m_e(e)
    {
        cout << "Derived constructor" << endl;
    }
    ~Derived() {
        cout << "Derived destructor" << endl;
    }
public:
    void show() {
        cout << m_a << ", " << m_b << ", " << m_c << ", " << m_d << ", " << m_e
        << endl;
    }
}
```

```
private:
    int m_e;
};

int main() {
    Derived obj(1, 2, 3, 4, 5);
    obj.show();
    return 0;
}

// 输出结果
BaseA constructor
BaseB constructor
Derived constructor
1, 2, 3, 4, 5
Derived destructor
BaseB destructor
BaseA destructor
```

## (2) 优点

派生类对象可以调用多个基类中的接口

## (3) 缺点：容易出现二义性

当多个基类具有同名的成员（包括多个基类又同时继承自相同基类）时，会出现二义性。消除二义性：

1. 可以在名字前面加上类名和域解析符::，以显示地指明到底使用哪个类的成员，消除二义性；
2. 为了解决多继承时的命名冲突和冗余数据问题，C++ 提出了虚继承，使得在派生类中只保留一份间接基类的成员。

```
// 假如上面的例子中，两个基类都有一个名为m_age的成员变量，派生类对象为d
d.m_age;           // 产生二义性
d.BaseA::m_age;    // 调用BaseA类的m_age
```

# 六、面向对象

## 1.介绍面向对象的三大特性，并且举例说明

三大特性：封装、继承、多态

### 封装：

就是把客观事物封装成抽象的类，对类中的属性和行为加以权限控制，从而达到保护数据安全，隐藏方法实现细节，简化编程的目的；

### 继承：

继承是类与类之间的一种关系，即子类继承父类的特征和行为，使得子类具有和父类相同的属性和行为。继承的好处在于子类继承了父类的属性和方法从而实现了代码的复用。继承也是实现多态的前提。

### 多态：

指一个类对象的相同方法在不同情形下有不同的表现形式，多态使得具有不同内部结构的对象可以共享相同的外部接口，即一个接口，多种实现。C++中，多态有两种形式，一种是静态多态（编译时多态），另外一种动态多态（运行时多态）。静态多态通过重载函数实现，在同一个类中完成；动态多态通过虚函数完成重写完成（父类指针指向子类对象），发生在不同的类（父类与派生类之间）。

## 2.静态类型和动态类型，静态绑定和动态绑定的介绍

1. 静态类型：对象在声明时采用的类型，在编译期既已确定；
2. 动态类型：通常是指一个指针或引用目前所指对象的类型，是在运行期决定的；（父类指针可以指向子类对象）
3. 静态绑定：绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期；
4. 动态绑定：绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期；

从上面的定义也可以看出，非虚函数一般都是静态绑定，而虚函数都是动态绑定（如此才可实现多态性）。

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

class A
{
public:
    /*virtual*/ void func() { std::cout << "A::func()\n"; }
};
class B : public A
{
public:
    void func() { std::cout << "B::func()\n"; }
};
class C : public A
{
public:
    void func() { std::cout << "C::func()\n"; }
};

int main()
{
    C* pc = new C(); // pc的静态类型是它声明的类型C*，动态类型也是C*；
    B* pb = new B(); // pb的静态类型和动态类型也都是B*；
    A* pa = pc;      // pa的静态类型是它声明的类型A*，动态类型是pa所指向的对象pc的类型C*；
    pa = pb;         // pa的动态类型可以更改，现在它的动态类型是B*，但其静态类型仍是声明时候的A*；
    C* pnull = NULL; // pnull的静态类型是它声明的类型C*，没有动态类型，因为它指向了NULL；

    pa->func();       // A::func() pa的静态类型永远都是A*，不管其指向的是哪个子类，都是直接调用A::func();
    pc->func();       // C::func() pc的动、静态类型都是C*，因此调用C::func();
    pnull->func();     // C::func() 不用奇怪为什么空指针也可以调用函数，因为这在编译期就确定了，和指针空不空没关系；

    return 0;
}
```

```
}
```

如果将A类中的virtual注释去掉，则运行结果是：

```
pa->func();          // ::func() 因为有了virtual虚函数特性，pa的动态类型指向B*，因此先在B中查找，找到后直接调用；
pc->func();          // ::func() pc的动、静态类型都是C*，因此也是先在C中查找；
pnul1->func();       // 指针异常，因为func是virtual函数，因此对func的调用只能等到运行期才能确定，然后才发现pnul1是空指针；
```

### 总结动态绑定和静态绑定的区别：

1. 静态绑定发生在编译期，动态绑定发生在运行期；
2. 对象的动态类型可以更改，但是静态类型无法更改；
3. 要想实现多态，必须使用动态绑定；
4. 在继承体系中只有虚函数使用的是动态绑定，其他的全部是静态绑定；（只有虚函数的调用使用的是动态类型，其他函数使用的都是静态类型）

### 建议：

1. 不要重新定义继承而来的非虚函数，会发生隐藏，最后调用的结果取决于对象声明的静态类型，不具有多态性（比如父类指针指向子类对象，我的本意是调用子类对象中的函数a，但由于函数a不是虚函数，导致我实际调用的是父类中的函数a）
2. 在虚函数中，要注意默认参数的使用，当父子类的虚函数中都存在默认参数时，会使用父类的默认参数，而调用子类的虚函数

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

class E
{
public:
    virtual void func(int i = 0)
    {
        std::cout << "E::func()\t" << i << "\n";
    }
};

class F : public E
{
public:
    virtual void func(int i = 1)
    {
        std::cout << "F::func()\t" << i << "\n";
    }
};

int main()
{
    F* pf = new F();
    E* pe = pf;
    pf->func(); //F::func() 1 正常，就该如此；
    pe->func(); //E::func() 0 哇哦，这是什么情况，调用了子类的函数，却使用了基类中参数的默认值！
}
```

```
    return 0;
}
```

### 引用也可以实现动态绑定

引用在创建的时候必须初始化，在访问虚函数时，编译器会根据其所绑定的对象类型决定要调用哪个函数。注意只能调用虚函数。对于非虚函数会根据引用的静态类型进行调用

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

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

class Son : public Base
{
public:
    virtual void fun()
    {
        cout << "son :: fun()" << endl;
    }
    void func()
    {
        cout << "son :: not virtual function" << endl;
    }
};

int main()
{
    Son s;
    Base& b = s;    // 基类类型引用绑定已经存在的Son对象，引用必须初始化
    s.fun();        // son::fun()
    b.fun();        // son :: fun()
    // b.func();    // b的动态类型虽然是Son，但是无法调用b.func()

    return 0;
}
```

## 3.C++如何实现多态

### C++实现多态，一言以蔽之：

在基类函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据所指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

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

```

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

class Son1 : public Base{
public:
    virtual void fun() override{
        cout << " Son1::func()" <<endl;
    }
};

class Son2 : public Base{

};

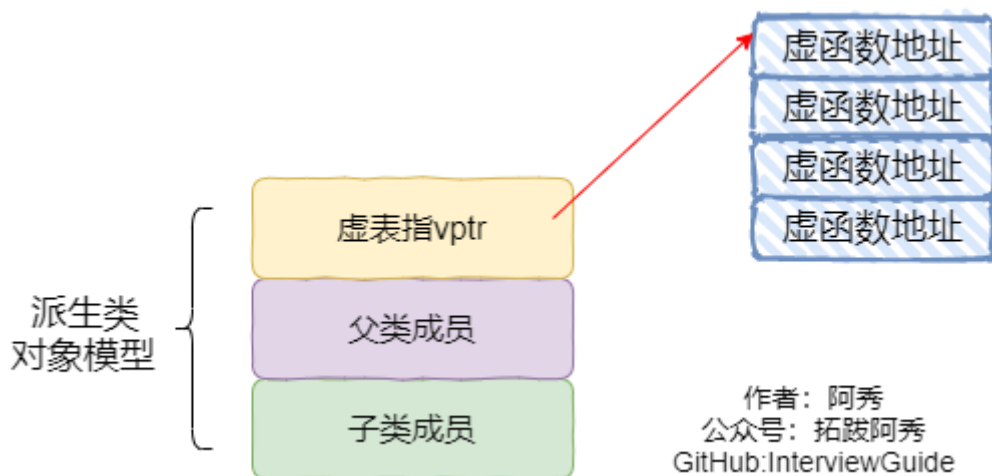
int main()
{
    Base* base = new Son1;
    base->fun();
    base = new Son2;
    base->fun();
    delete base;
    base = NULL;
    return 0;
}
// 运行结果
// Son1::func()
// Base::func()

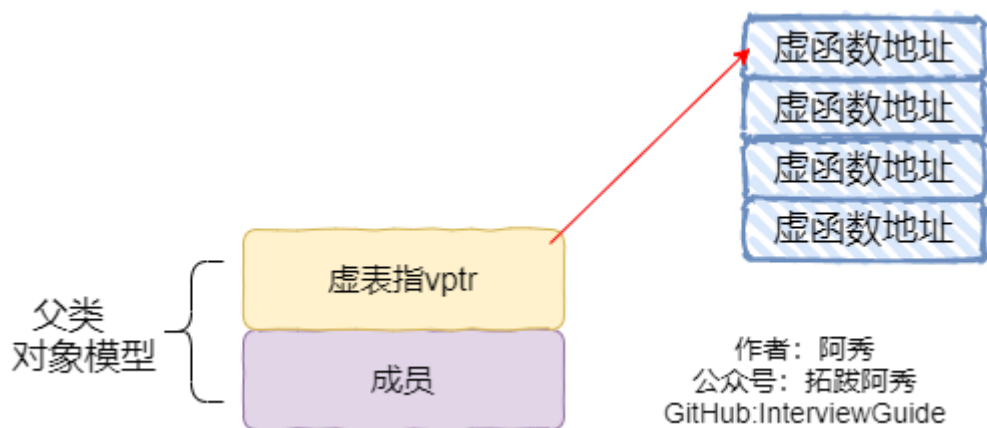
```

### 多态的底层原理：

虚函数表：类中含有virtual关键字修饰的函数时，编译器会自动生成虚表

虚函数表指针：在含有虚函数的类实例化对象时，对象地址的前四个字节存储的指向虚表的指针





### 多态实现过程:

1. 编译器在发现基类中有虚函数时, 会自动为每个含有虚函数的类生成一份虚表, 该表是一个一维数组, 虚表里保存了虚函数的入口地址
2. 编译器会在每个对象的前四个字节中保存一个虚表指针, 即vptr, 指向对象所属类的虚表。在构造时, 根据对象的类型去初始化虚指针vptr, 从而让vptr指向正确的虚表, 从而在调用虚函数时, 能找到正确的函数
3. 虚表的创建是在派生类定义对象时创建, 调用构造函数, 在构造函数中创建虚表, 并对虚表进行初始化。在构造子类对象时, 会先调用父类的构造函数, 此时编译器为父类对象初始化虚表指针, 另它指向父类的虚表; 当调用子类构造函数时, 为子类对象初始化虚表指针, 令它指向子类虚表
4. 当派生类对基类的虚函数没有重写时, 派生类的虚表指针指向的是基类的虚表; 当派生类对基类的虚函数重写时, 派生类的虚表指针指向的是自身的虚表; 当派生类中有自己的虚函数时, 在自己的虚表中将此虚函数地址添加在后面

这样指向派生类的基类指针在运行时, 就可以根据派生类对虚函数重写情况动态的进行调用, 从而实现多态性

## 七、函数/变量

### 1.在main执行之前和之后执行的代码可能是什么?

main函数执行之前, 主要就是初始化系统相关资源:

1. 设置栈指针
2. 初始化静态 `static` 变量和 `global` 全局变量, 即 `.data` 段的内容
3. 将未初始化部分的全局变量赋初值: 数值型 `short`, `int`, `long` 等为 0, `bool` 为 `FALSE`, 指针为 `NULL` 等等, 即 `.bss` 段的内容
4. 全局对象初始化, 在 `main` 之前调用构造函数, 这是可能会执行前的一些代码
5. 将main函数的参数 `argc`, `argv` 等传递给 `main` 函数, 然后才真正运行 `main` 函数
6. 执行被 `__attribute__((constructor))` 修饰的函数 (全局类对象的构造优先级更高)

main函数执行之后:

1. 全局对象的析构函数会在main函数之后执行;
2. 可以用 `atexit` 注册一个函数 (等级函数), 它会在main 之后执行;
3. 执行被 `__attribute__((destructor))` 修饰的函数 (全局类对象的析构优先级更高)

1. 给main函数传参-命令行下运行



```
int main(int argc, char* argv[]) {}
```

argc: 传参个数, 后面数组的大小

argv: 传入的字符串数组

默认argc=1, argv[0]为程序的名称, 所以我们设定的argc为num, 传入之后为num+1

## 2. GNU C的\_\_attribute\_\_机制:

\_\_attribute\_\_可以设置函数属性、变量属性和类型属性

被\_\_attribute\_\_((constructor))修饰的函数在main函数之前调用, 直接在函数定义之前加上即可

被\_\_attribute\_\_((destructor))修饰的函数在main函数之前调用

## 3. int atexit (void (\*func)(void));

程序终止时执行的函数,

参数要调用的函数, 该函数要求为无参无返回值

注册成功返回0, 否则返回非0

## 2.main函数的返回值有什么值得考究之处吗?

1. main()是程序运行的入口函数, 返回值类型必须是int (C语言可以是void, C++写void在可以)

2. 返回0表示程序正常退出 (最后的return 0可以省略不写), 非零表示异常退出, 关于非零代码没有明确的标准

```
// 两种合法用法
```

```
int main()
```

```
int main(int argc char* argv[])
```

## 3.声明和定义区别?

**变量的声明和定义:**

声明是仅仅告诉编译器, 有个某类型的变量会被使用, 但是编译器并不会为它分配任何内存, 而定义要在定义的地方为其分配存储空间; 相同变量可以在多处声明 (外部变量extern), 但只能在一处定义

**函数的声明和定义:**

声明: 一般在头文件里, 对编译器说: 这里我有一个函数叫function() 让编译器知道这个函数的存在; 定义: 一般在源文件里, 具体就是函数的实现过程写明函数体。

## 4.初始化和赋值的区别

- 对于简单类型来说, 初始化和赋值没什么区别
- 对于类和复杂数据类型来说, 初始化会调用拷贝构造函数进行对象构造, 赋值会调用类重载的赋值运算符=函数, 两个函数 (拷贝构造函数和重载的赋值运算符=函数做的事情可能不一样)。举例如下:

```
class A{
public:
    int num1;
    int num2;
public:
    A(int a=0, int b=0):num1(a),num2(b){};
    A(const A& a){};
    //重载 = 号操作符函数
    A& operator=(const A& a){
        num1 = a.num1 + 1;
```

```

        num2 = a.num2 + 1;
        return *this;
    };
};
int main(){
    A a(1,1);
    A a1 = a; //拷贝初始化操作，调用拷贝构造函数
    A b;
    b = a; //赋值操作，对象a中，num1 = 1, num2 = 1; 对象b中，num1 = 2, num2 = 2
    return 0;
}

```

## 5.C和C++的类型安全

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。

“类型安全”常被用来形容编程语言，其根据在于该门编程语言是否提供保障类型安全的机制；有的时候也用“类型安全”形容某个程序，判别的标准在于该程序是否隐含类型错误。

类型安全的编程语言与类型安全的程序之间，没有必然联系。好的程序员可以使用类型不那么安全的语言写出类型相当安全的程序，相反的，差一点儿的程序员可能使用类型相当安全的语言写出类型不太安全的程序。绝对类型安全的编程语言暂时还没有。

### (1) C的类型安全

C只在局部上下文中表现出类型安全，比如试图从一种结构体的指针转换成另一种结构体的指针时，编译器将会报告错误，除非使用显式类型转换。

然而，C中相当多的操作是不安全的。

- printf格式输出

```

#include <stdio.h>

int main(){
    printf("整型输出: %d\n",10);    // 整型输出: 10
    printf("浮点型输出: %s\n",10);  // 浮点型输出: 0.000000
    return 0;
}

```

// 上述代码中，使用%d控制整型数字的输出，没有问题，但是改成%f时，明显输出错误，再改成%s时，运行直接报segmentation fault错误

- malloc函数的返回值

malloc是C中进行内存分配的函数，它的返回类型是void即**空类型指针**，常常有这样的用法 `char* pStr=(char*)malloc(100*sizeof(char))`，这里明显做了显式的类型转换。

类型匹配尚且没有问题，但是一旦出现 `int* pInt=(int*)malloc(100*sizeof(char))` 就很可能带来一些问题，而这样的转换C并不会提示错误。

### (2) C++的类型安全

如果C++使用得当，它将远比C更有类型安全性。相比于C语言，C++提供了一些新的机制保障类型安全：

1. 操作符new返回的指针类型严格与对象匹配，而不是void\*
2. C中很多以void\*为参数的函数可以改写为C++模板函数，而模板是支持类型检查的；

3. 引入const关键字代替#define宏替换，它是有类型、有作用域的，而#define宏替换只是简单的文本替换
4. 一些#define宏可被改写为inline函数，结合函数的重载，可在类型安全的前提下支持多种类型，当然改写为模板也能保证类型安全
5. C++提供了dynamic\_cast关键字，使得转换过程更加安全，因为dynamic\_cast比static\_cast涉及更多具体的类型检查。

## 6.全局变量和局部变量有什么区别？

**生命周期不同：**全局变量随主程序创建而创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在；

**使用方式不同：**声明后全局变量在程序的各个部分都可以用到；局部变量分配在堆栈区，只能在局部使用。

**内存位置不同：**，全局变量分配在全局数据段，在程序开始运行的时候被加载；局部变量则分配在堆栈里面

## 7.形参与实参的区别？

1. 形参在调用的时候才会分配内存，调用结束立即释放；实参在函数调用之前就会分配内存，用实参来给形参进行初始化
2. 形参和实参是不同的变量，作用域不同，互不影响
3. 如果形参是指针或引用，那么形参和实参指向的是同一内存地址，形参指针或引用可以改变地址的值

## 8.静态变量什么时候初始化

1. 静态局部变量和全局变量一样，数据都存放在静态存储区（全局区域），所以在主程序之前，编译器已经为其分配好了内存（没有进行初始化，只会初始化一次，但可以多次赋值）
2. 在C语言中，初始化发生在代码执行之前，编译阶段分配好内存之后，就会进行初始化，C语言中无法使用变量对静态局部变量进行初始化
3. 在C++中，初始化时在执行相关代码时才会进行初始化，主要是由于C++引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。所以C++标准定为全局或静态对象是有首次用到时才会进行构造。在程序结束，按照构造顺序反方向进行逐个析构。所以在C++中是可以使用变量对静态局部变量进行初始化的。

## 9.C++中新增了string，它与C语言中的 char \*有什么区别吗？它是如何实现的呢？

string继承自basic\_string，其实是对 char\* 进行了封装，封装的string包含了 char\* 数组，容量，长度等属性。

string可以进行动态扩展，在每次扩展的时候另外申请一块原空间大小两倍的空间（2\*n），然后将原字符串拷贝过去，并加上新增的内容。

## 10.你知道const char\* 与string之间的关系是什么吗？

1. string 是c++标准库里面其中一个，封装了对字符串的操作，实际操作过程我们可以用const char\* 给string类初始化

2. string、const char\* 和 char\* 之间的转换关系：

1.string转const char\*

```
string s = "abc";  
const char* c_s = s.c_str();
```

2. const char\* 转string，直接赋值即可

```
const char* c_s = "abc";  
string s=c_s;  
string s(c_s);
```

3. string 转char\*

```
// char* c = s; // 不能直接转，需要先将string转成const char*，再用strcpy  
string s = "abc";  
const int len = s.length();
```

```
char* c = new char[len+1]; // 注意此处的+1  
strcpy(c,s.c_str());
```

4. char\* 转string

```
// char* c = "abc"; // 错误，const char* 不能直接赋值给char*，因为他们存的都是地址，  
const不能变  
string s(c);  
string s=c;
```

5. const char\* 转char\*

```
const char* cpc = "abc";  
char* pc = new char[strlen(cpc)+1];  
strcpy(pc,cpc);
```

6. char\* 转const char\*，直接赋值即可

```
char* pc = "abc";  
const char* cpc = pc;
```

// 总结

1. char\*和const char\*转string，直接初始化，或者赋值即可

```
string s=c; // c为char*和const char*都可以  
string s(c);
```

2. char\*转const char\*直接赋值

3. const char\*转char\*，需要先new (strlen+1)，再用strcpy

4. string转const char\* 利用函数s.c\_str()

5. string转char\*先将string转成const char\*，再用strcpy

## 11.对象复用的了解，零拷贝的了解

### 对象复用

对象复用其本质是一种设计模式：Flyweight享元模式。

通过将对象存储到“对象池”中实现对象的重复利用，这样可以避免多次创建、销毁重复对象的开销，节约系统资源。

### 零拷贝

零拷贝就是一种避免CPU将数据从一块存储空间拷贝到另外一块存储空间的技术。零拷贝技术可以减少数据拷贝和共享总线操作的次数。

在C++中，vector的一个成员函数**emplace\_back()**很好地体现了零拷贝技术，它跟push\_back()函数一样可以将一个**尚未构造的元素**插入容器尾部，区别在于：**使用push\_back()函数需要调用拷贝构造函数和转移构造函数，而使用emplace\_back()插入的元素原地构造，不需要触发拷贝构造和转移构造，只需要调用初始化构造函数，效率更高。**举个例子：

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

struct Person
{
    string name;
    int age;
    //初始构造函数
    Person(string p_name, int p_age) : name(std::move(p_name)), age(p_age)
    {
        cout << "初始构造函数" << endl;
    }
    //拷贝构造函数
    Person(const Person& other) : name(std::move(other.name)), age(other.age)
    {
        cout << "拷贝构造函数" << endl;
    }
    //转移构造函数
    Person(Person&& other) : name(std::move(other.name)), age(other.age)
    {
        cout << "转移构造函数" << endl;
    }
};

int main() {

    vector<Person> e;

    e.emplace_back("Jane", 23); // 初始化构造

    vector<Person> p;
    p.push_back(Person("Mike", 36)); // 先调用初始化构造，构造出临时对象；然后调用拷贝构造
```

```

Person bob("bob", 23);
Person alice("Alice", 18);

// 左值：拷贝构造，因为要拷贝一个对象给vector
e.emplace_back(bob);

// 参数：初始化构造，直接原地构造一个对象给vector，
// 如果用p.push_back(Person("Mike", 36))，这会先构造一个临时对象
e.emplace_back("Jane", 23);

// 右值：转移构造
e.emplace_back(move(alice));

system("pause");
return 0;
}

```

## 12.怎样判断两个浮点数是否相等？

对两个浮点数判断大小和是否相等不能直接用==来判断，会出错（不是报错）！对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值！浮点数与0的比较也应该注意。

计算机表示浮点数(float或double类型)都有一个精度限制，对于超出了精度限制的浮点数，[计算机](#)会把它们的精度之外的小数部分截断。

```

float a=10.222222225, b=10.222222229

```

数学上 a 和 b 是不相等的，但在32 位计算机中它们是相等的。

应该通过`abs( fa - fb) < 0.000001`来判断是否相等

1. 判断是否相等不能用 `==` 或 `!=`
2. 判断大小关系可以用 `>` 和 `<`

## 13.当程序中有函数重载时，函数的匹配原则和顺序是什么？

1. 名字查找
2. 确定候选函数
3. 寻找最佳匹配

## 14.如何在不使用额外空间的情况下，交换两个数？你有几种方法

```

1) 算术
x = x + y;
y = x - y;
x = x - y;

2) 异或-x和y不能是同一个元素（占用的内存不能是同一块）
x = x^y;
y = x^y;
x = x^y;

```

## 15.你知道strcpy和memcpy的区别是什么吗？

```
#include <string.h>
```

```
char*strcpy (char*dest, const char*src)
```

- 1.返回dest，复制之后的首地址
- 2.只适用于字符串复制，遇到'\0'结束复制，会复制'\0'，要求原字符串以'\0'结尾
- 3.目标空间要足够大，否则可能会造成溢出

```
char * strncpy(char *dest, const char *src, size_t n)
```

- 1.复制字符串的前n个字符（最多）
- 2.src 和 dest 所指的内存区域不能重叠，且 dest 必须有足够的空间放置n个字符
- 3.当src的长度小于n时，dest的剩余部分将用'\0'填充
- 4.strncpy()不会向dest追加结束标记'\0'

```
#include <string.h>
```

```
void * memcpy ( void * destination, const void * source, size_t num )
```

- 1.将num字节值从源指向的位置直接复制到目标内存块
- 2.不检查'\0'，精确地复制num字节
- 3.为避免溢出，目标参数和源参数所指向的数组的大小应至少为 num 个字节，并且不应重叠

```
#include <stdio.h>
```

```
int sprintf(char *str, const char *format, ...);
```

- 1.根据参数format字符串来转换并格式化数据，然后将结果输出到str指定的空间中，直到 出现字符串结束符 '\0'为止
- 2.str: 字符串首地址，format: 字符串格式，用法和 printf() 一样
- 3.成功返回实际格式化的字符个数，失败返回 - 1

strcpy和memcpy的区别？

- 1.复制的内容不同，strcpy只能复制字符串，而memcpy可以复制任意内容，例如字符数组、整型、结构体、类等
- 2.复制的方法不同，strcpy不需要指定长度，它遇到被复制字符串的结束符"\0"才结束，所以容易溢出；memcpy则是根据其第3个参数决定复制的长度
- 3.用途不同，通常在复制字符串时用strcpy，而需要复制其他类型数据时则一般用memcpy
- 4.效率memcpy最高，strcpy次之，sprintf的效率最低

## 16.成员函数里memset(this,0,sizeof(\*this))会发生什么

```
void *memset(void *s, int c, size_t n);
```

- 1.将某一块内存中的全部设置为指定的值，按[字节]进行填充，他部分类型，全部按照字节进行填充
- 2.s指向要填充的内存块
- 3.c是要被设置的值，由于是按照字节进行填充，所以，c的可取范围是0-255，多出的部分会截断，一般设为0
- 4.n是要被设置该值的字节数
- 5.返回类型是一个指向存储区s的指针

memset(this, 0, sizeof (\*this));将整个对象的内存全部置为0。下面的情况不能使用：

- 1.类含有虚函数表：这么做会破坏虚函数表，后续对虚函数的调用都将出现异常；
- 2.类中含有C++类型的对象：类对象会在构造函数体代码执行之前进行初始化，使用memset会破坏类对象初始化后的内存

## 八、C++11新特性

### 1.C++ 11有哪些新特性?

1. nullptr替代 NULL
2. 引入了 auto 和 decltype 这两个关键字实现了类型推导
3. 范围的for循环
4. 类和结构体的中初始化列表
5. Lambda 表达式 (匿名函数)
6. std::forward\_list (单向链表)
7. 右值引用和move语义

### 2.auto、decltype和decltype(auto)的用法

#### (1) auto

C++11新标准引入了auto类型说明符，用它就能让编译器替我们去分析表达式所属的类型。

**auto 让编译器通过初始值来进行类型推演。从而获得定义变量的类型，所以说 auto 定义的变量必须有初始值。**

```
//1. 普通：类型
int a = 1, b = 3;
auto c = a + b; // c为int型

//2. const类型
const int i = 5;
auto j = i;      // 变量i是顶层const，会被忽略，所以j的类型是int
const auto l = i; // 如果希望推断出的类型是顶层const的，那么就需要在auto前面加上const
auto k = &i;     // 变量i是一个常量，对常量取地址是一种底层const，所以k的类型是const int*

// 3. 引用和指针类型
int x = 2;
int& y = x;
auto z = y;      // z是int型不是int& 型
auto& p1 = y;    // p1是int&型
auto p2 = &x;    // p2是指针类型int*
// 对于const和&，auto会当成普通的赋值，不会保留const和&
// 如果对const常量取地址，则auto会推导出常量指针
```

#### (2) decltype

获得表达式类型，但不用表达式来初始化变量，例如用某表达式的类型当作函数的返回值类型，这种情况不能用auto，可以使用decltype

**它的作用是选择并返回操作数的数据类型。在此过程中，编译器只是分析表达式并得到它的类型，却不进行实际的计算表达式的值。**

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

//普通类型
```



```

decltype(func()) sum = 5; // sum的类型是函数func()的返回值的类型int，但是这时不会实际调用函数func()
int a = 0;
decltype(a) b = 4; // a的类型是int，所以b的类型也是int

//不论是顶层const还是底层const，decltype都会保留
const int c = 3;
decltype(c) d = c; // d的类型和c是一样的，都是顶层const
int e = 4;
const int* f = &e; // f是底层const
decltype(f) g = f; // g也是底层const

//引用与指针类型
//1. 如果表达式是引用类型，那么decltype的类型也是引用
const int i = 3, &j = i;
decltype(j) k = 5; // k的类型是 const int&

//2. 如果表达式是引用类型，但是想要得到这个引用所指向的类型，需要修改表达式：
int i = 3, &r = i;
decltype(r + 0) t = 5; // 此时是int类型

//3. 对指针的解引用操作返回的是引用类型
int i = 3, j = 6, *p = &i;
decltype(*p) c = j; // c是int&类型，c和j绑定在一起

//4. 如果一个表达式的类型不是引用，但是我们需要推断出引用，那么可以加上一对括号，就变成了引用类型了
int i = 3;
decltype((i)) j = i; // 此时j的类型是int&类型，j和i绑定在了一起

```

### (3)decltype(auto)

decltype(auto)是C++14新增的类型指示符，可以用来声明变量以及指示函数返回类型。

“=”号右边的表达式替换掉auto，再根据decltype的语法规则来确定类型。

```

int e = 4;
const int* f = &e; // f是底层const
decltype(auto) j = f; // j的类型是const int* 并且指向的是e

```

## 3.C++中NULL和nullptr区别

1. C语言的NULL被定义成 (void\*)0，C++的NULL被定义成整数0；nullptr是C++11的新增关键字

```

#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif

```

2. C++将NULL定义为0，当遇到函数重载时（一个参数为指针，另一个为int），如果NULL当作实参传入参数，将会调用int型参数的函数。

nullptr的引入用于解决这一问题，nullptr可以明确区分指针类型和整型，可以转换成任意的指针类型，但不能转换成整型

但当重载函数是以不同指针类型当作重载参考时，直接传入nullptr将无法区分调用哪个函数，这是需要将nullptr显示转成相应类型

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

// 1.NULL将会调用第二个重载函数
void fun(char* p) {
    cout << "char*" << endl;
}
void fun(int p) {
    cout << "int" << endl;
}

// 2.直接使用nullptr, 将无法区分
void func(char* p)
{
    cout<< "char* p" <<endl;
}
void func(int* p)
{
    cout<< "int* p" <<endl;
}

int main()
{
    fun(NULL); // 会调用第二个重载函数
    func(nullptr); // 报错，有多个匹配
    func((char*)nullptr); // 调用第一个func
    return 0;
}
```

## 4.智能指针

### (1) 原理

智能指针是一个类，是根据RAII（Resource Acquisition Is Initialization，资源获取即初始化）机制对普通指针进行封装，负责自动释放动态分配的对象，防止内存泄漏。

将动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源。

### (2) auto\_ptr

// 主要是为了解决“有异常抛出时发生内存泄漏”的问题

auto\_ptr:

1. 需要包含头文件<memory>

2. 用法: auto\_ptr<类型> 变量名(new 类型)

3. 三个常用函数:

(1) get() 获取智能指针托管的指针地址

(2) release() 取消智能指针对动态内存的托管, 并返回被托管的指针

(3) reset() 重置智能指针托管的内存地址, 如果地址不一致, 原来的会被析构掉(默认传

nullptr, 取消原指针的托管)

4. 注意:

(1) 尽可能不要将auto\_ptr 变量定义为全局变量, 没有意义

(2) 不要把`auto_ptr`智能指针赋值给同类型的另外一个智能指针，复制或者赋值都会改变资源的所有权

(3) **C++11** 后`auto_ptr`已经被“抛弃”，已使用`unique_ptr`替代! **C++11**后不建议使用`auto_ptr`

#### 5. `auto_ptr` 被**C++11**抛弃的主要原因

- (1) 复制或者赋值都会改变资源的所有权
- (2) 在STL容器中使用`auto_ptr`存在着重大风险，因为容器内的元素必须支持可复制和可赋值
- (3) 不支持数组的内存管理（析构函数中没有`delete[]`）

### (3) `unique_ptr`

`unique_ptr`:**C++11**用更严谨的`unique_ptr` 取代了`auto_ptr`

#### 1. `unique_ptr`特性:

- (1) 基于排他所有权模式：两个指针不能指向同一个资源，否则会报错（会`delete`两次）
- (2) 无法进行左值`unique_ptr`拷贝构造，也无法进行左值复制赋值操作，但允许临时右值赋值构造和赋值（不同于`auto_ptr`），被赋值的智能指针原来的管理的指针被释放，接管赋值得到的指针
- (3) 保存指向某个对象的指针，当它本身离开作用域时会自动释放它指向的对象。
- (4) 在容器中保存智能指针是安全的，但插入/赋值对象时需使用右值，用户直到后果
- (5) 支持对象数组的操作，析构函数中由`delete[]`

#### 2. 三个常用函数:

- (1) `get()` 获取智能指针托管的指针地址，或者直接用智能指针变量名（和普通指针一样，`auto_ptr`不能这样用）
- (2) `release()` 取消智能指针对动态内存的托管,并返回被托管的指针
- (3) `reset()` 重置智能指针托管的内存地址，如果地址不一致，原来的会被析构掉（默认传`nullptr`，取消原指针的托管）

#### 3. 构造

//自定义一个内存释放器，默认为`delete`

```
class DestructInt {
public:
    void operator()(int* p) {
        cout << "自定义内存释放器" << endl;
        delete p;
    }
};

void test04() {
    //1.空的unique_ptr
    unique_ptr<int>t1;

    //2.非空unique_ptr
    unique_ptr<int>t2(new int(5));

    //3.指向空数组
    unique_ptr<int[]> t3;

    //4.指向非空数组
    unique_ptr<int[]> t4(new int[5]);

    //5.空的unique_ptr，接受一个D类型的删除器D，使用D释放内存
    unique_ptr<int, DestructInt>t5;

    //6.定义unique_ptr，接受一个D类型的删除器D，使用删除器D来释放内存
    unique_ptr<int, DestructInt>t6(new int(6));
}
```

### (4) `shared_ptr`

shared\_ptr的原理：

采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

1. 智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针
2. 每次创建类的新对象时，初始化指针并将引用计数置为1
3. 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数
4. 对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数
5. 调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

shared\_ptr：

unique\_ptr具有排他性，也就是同一内存空间，只能由一个unique\_ptr变量管理。

1. use\_count()：获得当前托管指针的引用计数

2. 构造

```
void test02() {  
    //空的shared_ptr  
    shared_ptr<Person> sp1;  
    Person* person1 = new Person(1);  
    sp1.reset(person1); // 托管person1  
  
    //可以用原生指针，也可以用智能指针  
    Person* p = new Person(2);  
    shared_ptr<Person> sp2(p);  
    shared_ptr<Person> sp3(sp1);  
  
    //空数组  
    shared_ptr<Person[]> sp4;  
  
    shared_ptr<Person[]> sp5(new Person[5]{ 3, 4, 5, 6, 7 });  
  
    //带删除器  
    shared_ptr<Person> sp6(NULL, DestructPerson());  
  
    //  
    shared_ptr<Person> sp7(new Person(8), DestructPerson());  
}
```

3. 初始化

```
void test03() {  
    //方式一：构造函数  
  
    //方式二：  
    shared_ptr<int> up3 = make_shared<int>(2);  
    shared_ptr<string> up4 = make_shared<string>("字符串");  
    shared_ptr<Person> up5 = make_shared<Person>(9);  
  
}
```

}

4. 赋值

```
void test04() {  
    shared_ptr<int> up1(new int(10)); //int(10) 的引用计数为1  
    shared_ptr<int> up2(new int(11)); //int(11) 的引用计数为1
```

```

        //int(10)的引用计数减1,计数归零内存释放, up2共享int(11)给up1, int(11)的引用计数为2
        up1 = up2;
    }
5. 主动释放对象
void test05() {
    shared_ptr<int> up1(new int(10));
    up1 = nullptr; //int(10) 的引用计数减1,计数归零内存释放
    up1 = NULL; //作用同上

    //没有release函数
}
6. 重置
void test06() {
    shared_ptr<int> p(new int(10));
    p.reset(); //将p重置为空指针, 所管理对象引用计数减1

    int* p1 = new int(5);
    p.reset(p1); //将p重置为p1 (的值), p管控的对象计数减1, p接管对p1指针的管控
    //p.reset(p1, d); //将p重置为p1 (的值), p 管控的对象计数减1并使用d作为删除器(仿函数)
    //p1是一个指针
}
//仿函数, 内存删除
class DestructPerson {
public:
    void operator() (Person* pt) {
        cout << "DestructPerson..." << endl;
        delete pt;
    }
};
7. 交换
void test07() {
    shared_ptr<int> p1(new int(10));
    shared_ptr<int> p2(new int(5));

    swap(p1, p2); //交换p1和p2管理的对象, 原对象的引用计数不变
    p1.swap(p2); //交换p1和p2管理的对象, 原对象的引用计数不变
}

```

注意: 使用shared\_ptr时要注意避免对象交叉使用智能指针的情况 (你中有我, 我中有你)! 否则会导致内存泄露!

解决方法: weak\_ptr

```

class Boy {
public:
    Boy() {
        cout << "Boy 构造函数" << endl;
    }

    ~Boy() {
        cout << "~Boy 析构函数" << endl;
    }

    void setGirlFriend(shared_ptr<Girl> _girlFriend) {
        this->girlFriend = _girlFriend;
    }
}

```

```

private:
    shared_ptr<Girl> girlFriend;
};

class Girl {
public:
    Girl() {
        cout << "Girl 构造函数" << endl;
    }

    ~Girl() {
        cout << "~Girl 析构函数" << endl;
    }

    void setBoyFriend(shared_ptr<Boy> _boyFriend) {
        this->boyFriend = _boyFriend;
    }

private:
    shared_ptr<Boy> boyFriend;
};

void useTrap() {
    shared_ptr<Boy> spBoy(new Boy());
    shared_ptr<Girl> spGirl(new Girl());

    //陷阱用法
    spBoy->setGirlFriend(spGirl);
    spGirl->setBoyFriend(spBoy);
    //此时boy和girl的引用计数都是2，函数调用结束后，引用计数都为1
}

```

#### (5) 智能指针shared\_ptr代码实现:

```

template<typename T>
class SharedPtr
{
public:
    // 1.默认构造函数
    SharedPtr(T* ptr = NULL):_ptr(ptr), _pcount(new int(1))
    {}

    // 2.初始化构造函数
    SharedPtr(const SharedPtr& s):_ptr(s._ptr), _pcount(s._pcount){
        (*_pcount)++;
    }

    // 3.重载赋值运算符
    SharedPtr<T>& operator=(const SharedPtr& s){
        if (this != &s)
        {
            if (--(*this->_pcount) == 0)
            {
                delete this->_ptr;
                delete this->_pcount;
            }
        }
    }
}

```

```

        _ptr = s._ptr;
        _pcount = s._pcount;
        *(_pcount)++;
    }
    return *this;
}

// 4.重载*运算符
T& operator*()
{
    return *(this->_ptr);
}

// 5.重载->运算符
T* operator->()
{
    return this->_ptr;
}

// 6.析构函数
~SharedPtr()
{
    --(*(this->_pcount));
    if (*(this->_pcount) == 0)
    {
        delete _ptr;
        _ptr = NULL;
        delete _pcount;
        _pcount = NULL;
    }
}

private:
    T* _ptr;
    int* _pcount; // 指向引用计数的指针
};

```

## (6) weak\_ptr

```

// 弱指针的使用
1. 只可以从一个shared_ptr或另一个weak_ptr对象构造
2. 构造和析构不会引起引用计数的增加或减少（只引用不计数）
3. 没有重载*和->但可以使用 lock 获得一个可用的 shared_ptr 对象
4. use_count() 获得引用计数
5. expired函数：判断当前智能指针是否还有托管的对象，有则返回false，无则返回true

void test01() {
    shared_ptr<Boy> spBoy(new Boy());
    shared_ptr<Girl> spGirl(new Girl());

    //1.弱指针的使用
    // 只可以从一个shared_ptr或另一个weak_ptr对象构造
    weak_ptr<Girl> wpGirl_1;           // 定义空的弱指针
    weak_ptr<Girl> wpGirl_2(spGirl);    // 使用共享指针构造
}

```

```

wpGirl_1 = spGirl; // 允许共享指针赋值给弱指针

//2.弱指针也可以获得引用计数
cout << "spGirl \t use_count = " << spGirl.use_count() << endl;
cout << "wpGirl_1 \t use_count = " << wpGirl_1.use_count() << endl;

//3.弱指针不支持 * 和 -> 对指针的访问
//wpGirl_1->setBoyFriend(spBoy);
//(*wpGirl_1).setBoyFriend(spBoy);

//4.在必要的使用可以转换成共享指针
shared_ptr<Girl> sp_girl;
sp_girl = wpGirl_1.lock();
cout << sp_girl.use_count() << endl;
// 使用完之后, 再将共享指针置NULL即可
sp_girl = NULL;

//5.expired函数: 判断当前智能指针是否还有托管的对象, 有则返回false, 无则返回true
shared_ptr<int>sp(new int(100));
weak_ptr<int>wp = sp;
if (wp.expired()) {
    cout << "没有托管的对象了" << endl;
}
else {
    cout << "还有托管的对象" << endl;
}
}

```

```

// 弱指针解决shared_ptr循环引用的问题
// 将其中一个类中的share_ptr改成weak_ptr, 因为weak_ptr不会增加应用计数, shared_ptr循环引用
的问题得以解决
class Girl;

class Boy {
public:
    Boy() {
        cout << "Boy 构造函数" << endl;
    }

    ~Boy() {
        cout << "~Boy 析构函数" << endl;
    }

    void setGirlFriend(shared_ptr<Girl> _girlFriend) {
        this->girlFriend = _girlFriend;

        // 在必要的使用可以转换成共享指针
        shared_ptr<Girl> sp_girl;
        sp_girl = this->girlFriend.lock();

        cout << sp_girl.use_count() << endl;
        // 使用完之后, 再将共享指针置NULL即可
        sp_girl = NULL;
    }
}

```



```

private:
    weak_ptr<Girl> girlFriend; // 这里是
};

class Girl {
public:
    Girl() {
        cout << "Girl 构造函数" << endl;
    }

    ~Girl() {
        cout << "~Girl 析构函数" << endl;
    }

    void setBoyFriend(shared_ptr<Boy> _boyFriend) {
        this->boyFriend = _boyFriend;
    }

private:
    shared_ptr<Boy> boyFriend;
};

```

#### (7) 总结:

1. auto\_ptr主要是为了解决“有异常抛出时发生内存泄漏”的问题，但是它存在复制或者赋值都会改变资源的所有权的问题。C++11不建议使用auto\_ptr，C++14禁止使用auto\_ptr
2. unique\_ptr也不支持支持左值复制赋值操作，但允许临时右值赋值构造和赋值，让用户显示直到复制复制的后果。unique\_ptr具有排他性，也就是同一内存空间，只能由一个unique\_ptr变量管理。
3. shared\_ptr通过采用应用计数器的方法，允许多个智能指针指向同一个对象，解决了unique\_ptr的问题。但是存在循环引用的问题，导致两个指针指向的内存都无法释放。
4. weak\_ptr可以打破循环引用（只引用，不计数，两个类中一个是shared\_ptr，一个是weak\_ptr），weak\_ptr是配合shared\_ptr提出的

## 5.使用智能指针管理内存资源，RAII是怎么回事？

1. RAII全称是“Resource Acquisition Is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源
2. 因为C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在RAII的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定
3. 智能指针（std::shared\_ptr和std::weak\_ptr）即RAII最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记delete造成的内存泄漏。

## 6.智能指针出现循环引用怎么解决？

循环引用：两个对象互相使用一个shared\_ptr成员变量指向对方。shared\_ptr只有引用计数为0时才会析构对象，释放内存，但是循环引用会使得最终引用计数为1，不会析构对象，从而造成内存泄漏

weak\_ptr专门用于解决shared\_ptr循环引用的问题，weak\_ptr不会修改引用计数，即其存在与否并不影响对象的引用计数器。

由于弱引用不会改变引用计数，所以可以在其中一个类中使用weak\_ptr，当需要用到该指针时，再将其转成shared\_ptr（由于weak\_ptr没有重载\*和->，需要使用使用lock获得一个可用的shared\_ptr 对象），从而打破循环，解决了循环引用的问题

## 7.说一说你了解的关于lambda函数的全部知识

### (1) lambda表达式的作用：

lambda表达式，在调用或作为函数参数传递的位置处定义匿名函数对象（是一个右值的函数对象）的便捷方法。

### (2) 原理：

1. 编译器会把一个Lambda表达式自动生成一个匿名类，并在类中重载函数()运算符（匿名函数对象），运行时，这个lambda表达式会返回一个匿名实例化对象（右值）
2. 采用值捕获时，相当于在匿名类中声明成员变量，同时用捕获来的变量初始化对应的成员变量
3. 默认情况下，由匿名类中重载的()运算符是一个const成员函数，所以采用值捕获的值不能修改，mutable可以去除const属性
4. 采用引用捕获时，编译器可以直接使用该引用而无须在匿名类中将其声明为成员变量，唯一需要注意的是，变量将由程序负责确保执行时引用的对象确实存在。

```
[capture] (parameters) mutable ->return_type {statement};
```

#### 1. [capture]，捕获列表，不可省

- (1) []：不捕获任何变量
- (2) [var]：以值传递的方式捕获var，如果var为=，表示以值传递方式捕获所有父作用域的变量，须在lambda表达式定义之前出现。在类中定义的lambda表达式不能直接捕获类成员变量，需要捕获this指针
- (3) [&var]：以引用传递捕捉变量var，如果[&]，表示以引用传递方式捕捉所有父作用域的变量（包括this）
- (4) 举例：[this]：以值传递方式捕捉当前的this指针；[=, &a, &b]：以引用传递的方式捕捉变量a和b，以值传递方式捕捉其它所有变量；[&, a, this]：以值传递的方式捕捉变量a和this，引用传递方式捕捉其它所有变量
- (5) 注意：不要重复传递变量，即通过引用传递，又通过值传递；捕获的变量的值是在lambda创建时拷贝，而不是调用时拷贝；以引用的方式捕获，需要保证匿名对象调用时，引用存在

#### 2. (parameters)，参数列表，没有可以连同()省略

#### 3. mutable，去除重载operator()的const属性，可以省略。使用时，参数列表不可省略

#### 4. ->return\_type，返回值类型，可以连同->省略

#### 5. {statement}，函数体，函数体可以使用的变量

- (1) 捕获的变量
- (2) 参数列表中的变量
- (3) 函数体内声明的局部变量
- (4) 类成员变量，需要捕获this
- (5) 静态变量/全局，不需要捕获

```
// 一个使用所有部分的例子
int main() {
    int a = 10;
    int b = 2;
    auto f = [&a, b](int c)mutable ->int{b = 1; return a + b + c; };
    a = 20;
    cout << f(5) << endl;// 26

    return 0;
}
// 注意lambda表达式后加不加()的区别，加()表示调用函数对象，不加()表示声明函数对象
```

### (3) 优缺点

#### 1. 优点:

- a)可以直接在需要调用函数的位置定义短小精悍的函数，而不需要预先定义好函数
- b)使用Lambda表达式变得更加紧凑，结构层次更加明显、代码可读性更好

#### 2. 缺点:

- a)Lambda表达式语法比较灵活，增加了阅读代码的难度
- b)对于函数复用无能为力

## 九、STL模板库/模板

### 1.什么是STL?

C++stl广义上讲可以分为三类：容器、算法、迭代器，容器和算法通过迭代器进行无缝连接。可以细分为六大组件：容器、算法、迭代器、仿函数、适配器、空间配置器

- 1. 容器：STL内部封装好的数据结构，以类模板的方式实现，如...
- 2. 算法：是一种函数模板，如sort、serach等，包括不同容器的特定算法

函数模板是一组函数的抽象描述，不是真的函数，函数模板不会编译成任何目标代码；模板函数是函数模板的实例化，这些模板函数再程序运行时进行编译和链接，然后产生目标代码。

- 3. 迭代器：泛型指针，算是一种智能指针，重载了 `operator*`、`operator->`、`operator++`、`operator--` 等指针相关操作的类模板。所有STL容器都附带自己的迭代器
- 4. 仿函数：重载operator()的类，行为类似函数，具有可适配性，如greater、less
- 5. 适配器：用来修饰容器、仿函数或迭代器接口。如queue和stack，它们的底部完全借助deque，所有操作都由底层的deque供应。改变仿函数接口称为仿函数适配器，改变容器接口称为容器适配器；改变迭代器接口称为迭代器适配器
- 6. 分配器：负责空间配置与管理。是一个实现了动态空间配置、空间管理、空间释放的class template

## 2.解释一下什么是trivial destructor

“trivial destructor”一般是指用户没有自定义析构函数，而由系统生成的析构函数；

用户自定义了析构函数，称之为“non-trivial destructor”，这种析构函数要将在堆区申请的空间显式的释放掉，否则会造成内存泄漏；

## 3.迭代器：++it、it++哪个好，为什么

1. 前置++返回引用；后置++返回临时对象。临时对象的产生会降低效率
2. 实现代码

```
class MyInteger {
    friend ostream& operator<<(ostream& out, MyInteger myint);

public:
    MyInteger() { m_Num = 0; }

    //前置++
    MyInteger& operator++() {
        //先++
        m_Num++;
        //再返回
        return *this;
    }

    //后置++
    MyInteger operator++(int) {
        //先返回
        MyInteger temp = *this; //记录当前本身的价值，然后让本值加一，达到先返回再加一
        //再++
        m_Num++;
        return temp;
    }

private:
    int m_Num;
};

ostream& operator<<(ostream& out, MyInteger myint) {
    out << myint.m_Num;
    return out;
}

//前置++，先++在返回
void test01(){
    MyInteger myint;
    cout << ++myint << endl;
    cout << myint << endl;
}

//后置++，先返回 再++
void test02() {
```

```
MyInteger myint;
cout << myint++ << endl;
cout << myint << endl;
}
```

## 4.左值引用和右值引用

### (1) 左值和右值

左值：是一个表示数据的表达式（如变量名），可以获取它的地址+可以对它赋值（const常量是左值，但不能对其赋值）。左值既可以出现在赋值号的左边/右边

右值：也是一个表示数据的表达式（如字面常量，表达式返回值，传值返回的函数返回值）。右值不能取地址，只能位于赋值号的右边

1. 纯右值：一般意义上的右值
2. 将亡值：跟右值引用相关的表达式。右值是将亡值，将亡值不一定是右值，也有可能是move的返回值

**左值引用就是对左值的引用（别名）；右值引用就是对右值的引用（别名）**

### (2) 注意

1. 左值引用可以连续左值引用，右值引用不可联系右值引用（右值引用后该变量为左值，不能使用右值引用了）
2. 左值引用只能引用左值，不能引用右值；**但是const左值引用既可引用左值，也可引用右值**
3. 右值引用只能引用右值，不能引用左值；但是右值引用可以move以后的左值
4. 通过右值引用延长变量值的生命期
5. 引用本质上都是用来减少拷贝，提高效率。左值引用解决大部分的场景；右值引用是左值引用一些盲区的补充
6. 引用类型的唯一作用是限制接收的类型，左值引用和右值引用都是左值；

### (3) 左值引用的使用场景

左值引用做参数和做返回值都可以提高效率，但是不要返回局部变量的引用，因为局部变量出了作用域会被销毁（右值引用也不可以）

### (4) 右值引用的使用场景

右值引用，一般使用在需要深拷贝的类中，需要实现移动构造和移动赋值，利用移动构造和移动赋值在传参和传返回值过程中转移资源，减少拷贝

## 5.完美转发

所谓完美转发是指，将一个参数a传递给函数f，在f内部再将参数a传递给另一个函数h，在传递给h时保持参数a原生类型属性

**std::forward 完美转发在传参的过程中保留对象原生类型属性，如果原来的值是左值，经std::forward处理后该值还是左值；如果原来的值是右值，经std::forward处理后它还是右值。**

```
// 左值引用
void process(int& x) {
    cout << "lvalue:" << x << endl;
}
```

```

// 右值引用
void process(int&& x) {
    cout << "rvalue:" << x << endl;
}

// 不完美转发
void forward1(int&& x) {
    process(x); // x退化为左值，调用第一个process
}

// 完美转发
void forward2(int&& x) {
    process(forward<int>(x)); // 调用第二个process
}

```

模板中的&&不代表右值引用，表示万能引用，既可以接收左值，又可以接收右值。因此可以用一个函数模板来接收函数参数，然后再将利用forward实现完美转发

```

// 左值引用
void process(int& x) {
    cout << "lvalue:" << x << endl;
}

// 右值引用
void process(int&& x) {
    cout << "rvalue:" << x << endl;
}

// 完美转发
template<typename T>
void test(T&& x) {
    process(forward<T>(x)); // 调用第二个process
}

int main() {
    test(2); // 右值引用

    int a = 10;
    test(a); // 左值引用

    system("pause");
    return 0;
}

```

## 6.vector

### (1) vector原理-序列式容器

1. vector底层是一段连续的内存空间，用三个迭代器表示，这三个迭代器分别是指向vector容器对象的起始字节位置、最后一个元素的末字节位置（size）和整个vector容器对象内存空间（capacity）的末字节位置
2. 支持对象的随机存取，时间复杂度为O(1)；因为内存空间是连续的，所以在任意位置插入、删除元素会造成存储元素的拷贝，时间复杂度为O(n)；在末尾插入、删除的时间复杂度为O(1)

3. 当vector总有效元素个数 (size) 与空间容量 (capacity) 相等时, vector内部会触发扩容机制: 开辟新空间、拷贝元素、释放旧空间
4. 对vector的任何操作, 一旦引起空间重新配置, 指向原vector的所有迭代器就都失效了

## (2) vector使用

```
// 1. 构造
vector<T> v; // 采用模板实现类实现, 默认构造函数
vector(n, elem); // 构造函数将n个elem拷贝给本身, 如果省略elem, 则默认为0
vector(const vector &vec); // 拷贝构造函数
vector(v.begin(), v.end()); // 将v[begin(), end())区间中的元素拷贝给本身

// 2. 赋值
vector& operator=(const vector &vec); // 重载等号操作符
assign(beg, end); // 将别的vector对象[beg, end)区间中的数据拷贝赋值给本身
assign(n, elem); // 将n个elem拷贝赋值给本身

// 3. 容量和大小
empty(); // 判断容器是否为空
capacity(); // 容器的容量
size(); // 返回容器中元素的个数
resize(int num, elem); // 重新指定容器的长度为num, 若容器变长, 则以elem填充新位置,
// 如果容器变短, 则末尾超出容器长度的元素被删除
// 如果resize之后的容器大小超过了capacity则会进行扩容,
// 否则capacity不变

// 4. 插入和删除
push_back(ele); // 尾部插入元素ele
pop_back(); // 删除最后一个元素
insert(const_iterator pos, ele); // 迭代器指向位置pos插入元素ele
insert(const_iterator pos, int count, ele); // 迭代器指向位置pos插入count个元素ele
erase(const_iterator pos); // 删除迭代器指向的元素, 返回被删除元素后一个元素的位置
erase(const_iterator start, const_iterator end); // 删除迭代器[start, end)之间的元素
clear(); // 删除容器中所有元素
// 删除元素不会改变capacity, 插入元素在size>capacity时会扩容

// 5. 数据存取
at(int idx); // 返回索引idx所指的数据
operator[]; // 返回索引idx所指的数据
front(); // 返回容器中第一个数据元素
back(); // 返回容器中最后一个数据元素

// 6. 互换容器
swap(vec); // 将vec与本身的元素互换
// 可以达到实用的收缩内存效果, 举例:
vector(vec).swap(vec); // 将vec中多余内存清除, vector(vec)调用拷贝构造函数构造出一个临时对象(size与vec的相同, capacity=size), 然后用这个对象与vec交换, 从而达到收缩内存的作用
vector().swap(vec); // 清空vec的全部内存;

int main ()
{
    vector<int> vec (100,100);
    vec.push_back(1);
    vec.push_back(2);
```

```

cout <<"vec.size(): " << vec.size() << endl;          // 102
cout <<"vec.capacity(): " << vec.capacity() << endl;    // 200

vector<int>(vec).swap(vec); //清空vec中多余的空间，相当于vec.shrink_to_fit();

cout <<"vec.size(): " << vec.size() << endl;          // 102
cout <<"vec.capacity(): " << vec.capacity() << endl;    // 102

vector<int>().swap(vec); //清空vec的全部空间

cout <<"vec.size(): " << vec.size() << endl;          // 0
cout <<"vec.capacity(): " << vec.capacity() << endl;    // 0

return 0;
}

// 7.手动指定capacity，可以减少扩容次数
reserve(int len); // 如果len<=capacity，那么capacity不变，否则变为len

```

### (3) vector扩容机制

// 1.vector是如何进行扩容的？  
开辟新空间 --> 拷贝元素 --> 释放旧空间

每次扩容新空间不能太大，也不能太小，太大容易造成空间浪费，太小则会导致频繁扩容而影响程序效率，一般是2倍/1.5倍进行扩容

// 2.如何避免扩容导致效率低  
在插入元素之前，预估vector存储元素的个数，提前将底层容量开辟好即可（使用reserve函数）

// 3.为什么不以等长个数方式进行扩容  
如果以等长个数进行扩容，那么push\_back的平均时间复杂度为 $O(N)$   
当前容量为0，假设要插入n个元素，每次扩容的长度为k  
需要进行 $n/k$ 向上取整次扩容，第i次扩容需要将 $(i-1)*k$ 个元素拷贝到新空间，这么多次扩容求和即为需要拷贝的次数，除以n就是插入每个元素需要拷贝的平均次数，结果是  $O(n)$  量级

// 4.为什么以倍数方式进行扩容  
如果以倍数方式进行扩容，那么push\_back的平均时间复杂度为  $O(1)$   
当前容量为0，假设要插入n个元素，每次扩容为原来的m倍  
需要扩容  $\log_m(n)$  次，第i次扩容需要将 $m^{i-1}$ 个元素拷贝到新空间，这么多次扩容求和即为需要拷贝的次数，除以n就是插入每个元素需要拷贝的平均次数，结果是  $O(1)$  量级

// 5. 为什么选择1.5倍或者2倍方式扩容，而不是3倍、4倍  
以小于2倍的方式进行扩容可以复用之前已经释放的空间；  
以更大倍数扩容可能造成更大的空间浪费

linux下是按照2倍的方式扩容的，而windows vs下是按照1.5倍的方式扩容的

最佳扩容倍数

满足：  $X(n-2)+X(n-1)=X(n)$ ，斐波那契数列。。。

当n取无穷大时，扩容倍数为黄金分割比1.618

扩容倍数的选择使空间和时间权衡的结果，空间分配多，平坦时间复杂度就低，但浪费空间也多



## (4) 其他问题

// 1. **vector**如何释放空间?

**vector**占用的内存空间只增不减, 删除元素只是将**size**变小, **capacity**并没有改变, 占用的内存空间也不变, 无法保证内存的回收

如果需要空间动态缩小, 可以考虑使用**deque**

如果使用**vector**, 可以用 **swap()** 来释放多余内存或者清空全部内存

## 7.list

### (1) 实现原理

1. list底层是由一个带头结点的双向循环链表实现, 不支持随机存取

2. 时间复杂度: 插入 $O(1)$ 、查找 $O(N)$ 、删除 $O(1)$

### (2) list使用

// 1.list构造函数

```
list<T> lst;           // list采用模板类实现, 对象的默认构造形式:
list(beg, end);        // 构造函数将[beg, end)区间中的元素拷贝给本身。
list(n, elem);         // 构造函数将n个elem拷贝给本身。
list(const list &lst); // 拷贝构造函数。
```

// 2.赋值和交换

```
assign(beg, end);      // [beg, end)区间中的数据拷贝赋值给本身
assign(n, elem);       // n个elem拷贝赋值给本身。
list& operator=(const list &lst); // 重载等号操作符
swap(lst);             // lst与本身的元素互换。
```

// 3.大小操作

```
size();               // 返回容器中元素的个数
empty();              // 判断容器是否为空
resize(num, elem);    // 重新指定容器的长度为num, 若容器变长, 则以默认值填充新位置。如果容器变短, 则末尾超出容器长度的元素被删除。
```

// 4.插入和删除

```
push_back(elem);      // 在容器尾部加入一个元素
pop_back();           // 删除容器中最后一个元素
push_front(elem);     // 在容器开头插入一个元素
pop_front();          // 从容器开头移除第一个元素
insert(pos, elem);    // 在pos位置插elem元素的拷贝, 返回新数据的位置
insert(pos, n, elem); // 在pos位置插入n个elem数据, 无返回值
insert(pos, beg, end); // 在pos位置插入[beg, end)区间的数据, 无返回值
clear();              // 移除容器的所有数据
erase(beg, end);       // 删除[beg, end)区间的数据, 返回下一个数据的位置
erase(pos);            // 删除pos位置的数据, 返回下一个数据的位置, 删除位置的迭代器失效, 其他迭代器不变
remove(elem);          // 删除容器中所有与elem值相等的元素
```

// 5.数据存取

```
front();             // 返回第一个元素
back();              // 返回最后一个元素
```

// 6.反转和排序

```
reverse();           // 反转链表
```

```
sort();           // 链表排序
```

### (3) 其他

1. `list` 插入都不会使原有的迭代器失效，删除只会使被删除的迭代器失效，其他元素迭代器不失效
2. `vector` 插入元素可能引起扩容，使得所有迭代器失效；删除元素会使删除元素及之后的元素失效

## 8. 容器内部删除一个元素

1. 调用 `erase` 函数根据迭代器位置删除一个/区间元素，返回删除元素的下一个元素的迭代器；
2. `vector/deque` 元素删除元素会使删除元素之后元素原来的迭代器失效，返回的新的迭代器；
3. 关联式容器还可以通过 `key` 来删除元素，删除成功返回 1，否则返回 0（删除的 `key` 不存在）；
4. 关联式容器可以通过 `erase(it++)` 来更新 `it`，和 `it=erase(it)` 效果一样，而 `vector/deque` 不可以，因为删除元素之后的迭代器已经失效了

## 9. 迭代器失效情况

以 `vector` 为例

### 插入元素：

插入位置之前的迭代器不失效，之后的位置全部失效；如果插入元素引起了扩容，则所有迭代器均失效

### 删除元素：

删除位置之前的迭代器不失效，之后的位置全部失效；删除元素不会引起扩容

`deque` 和 `vector` 类似；`list` 仅删除位置的迭代器失效；`map/set` 底层是红黑树，删除节点不会影响其他节点的迭代器；`unordered_set/map`，无序，迭代器意义不大，`rehash` 之后迭代器全部失效，删除元素不会使得其他迭代器失效

## 10. 红黑树

1. 二叉搜索树：左子树上的节点值都小于根节点值；右子树的节点值都大于根节点值，左右子树也是搜索二叉树
2. 节点是红色/黑色
3. 叶子节点是空节点，都是黑色
4. 红色节点的父/子节点都是黑色
5. 从根节点到叶子节点（空）的所有路径上不能由两个连续的红色节点
6. 从任意节点到叶子节点（空）的所有路径都包含相同数目的黑色节点

通过上述约束，红黑树可以保证最长路径不超过最短路径长度的两倍，从而达到近似平衡的效果。添加/查找/删除的时间复杂度都是  $O(\log n)$

## 11.set/multiset

### (1) 实现原理

set属于关联式容器，key和value相同，set会对所有元素进行自动排序。底层是通过红黑树实现的。不允许有相同的key。添加/查找/删除的时间复杂度都是 $O(\log n)$

### (2) 使用

```
// 1.构造
set<T> st;           // 默认构造函数
set(const set &st);  // 拷贝构造函数

// 2.赋值
set& operator=(const set &st); // 重载等号操作符

// 3.set大小和交换
size();           // 返回容器中元素的数目
empty();          // 判断容器是否为空
swap(st);         // 交换两个集合容器

// 4.set插入和删除
insert(elem);     // 在容器中插入元素。
clear();          // 清除所有元素
erase(pos);       // 删除pos迭代器所指的元素，返回下一个元素的迭代器。
erase(beg, end);  // 删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
erase(elem);      // 删除容器中值为elem的元素，成功返回1，失败返回0

// 5.set查找和统计
find(key);        // 查找key是否存在，若存在，返回该键的元素的迭代器；若不存在，返回set.end();
count(key);        // 统计key的元素个数（对于set，结果为0或者1）

// 6.set和multiset区别
set不可以插入重复数据，而multiset可以
```

## 12.map/multimap

### (1) 实现原理

map属于关联式容器，存储的元素是pair<key,value>，map会根据key进行自动排序。底层是通过红黑树实现的。不允许有相同的key。添加/查找/删除的时间复杂度都是 $O(\log n)$

### (2) 使用

```
// 1.构造
map<T> st;           // 默认构造函数
map(const map &st);  // 拷贝构造函数

// 2.赋值
map& operator=(const map &st); // 重载等号操作符

// 3.map大小和交换
size();           // 返回容器中元素的数目
empty();          // 判断容器是否为空
swap(st);         // 交换两个集合容器
```

```

// 4.map插入和删除
insert(elem);           // 在容器中插入元素
mp.insert(pair<int, string>(1, "student_one"));
mp.insert(make_pair(1, "student_one"));
mp.insert({1, "student_one"});
mp[1]="student_one";

clear();                // 清除所有元素
erase(pos);             // 删除pos迭代器所指的元素，返回下一个元素的迭代器。
erase(beg, end);        // 删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
erase(elem);            // 删除容器中值为elem的元，成功返回1，失败返回0

// 5.map查找和统计
find(key);              // 查找key是否存在,若存在，返回该键的元素的迭代器；若不存在，返回map.end();
count(key);             // 统计key的元素个数（对于map，结果为0或者1）

// 6.map和multimap区别
map不可以插入相同key的元素，而multimap可以

```

// set/map的自定义类型排序，默认是从小到大

// 1.在类中重载<和>

在自定义数据类型中重载<（从小到大排序）和>（从大到小排序），根据具体排序需要，重载一个即可。  
注意这两种方式重载运算符函数时需要用const修饰！

```

set(int,less<int>);      // 从小到大
set(int,greater<int>);  // 从大到小

```

// 2.利用仿函数

定义一个重载小括号运算符的类，在实例化set/map对象时将该【类名】作为参数传入

// 3.函数指针

```

class Person {
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    string m_Name;
    int m_Age;

    // 1.在类中重载<
    bool operator<(const Person& other) const { // 两个const都不能少
        return other.m_Age < this->m_Age;
    }
};

// 2.仿函数
class myCompare {
public:
    bool operator()(const Person& p1, const Person& p2) const { // 两个const都不能少
        return p1.m_Age < p2.m_Age;
    }
};

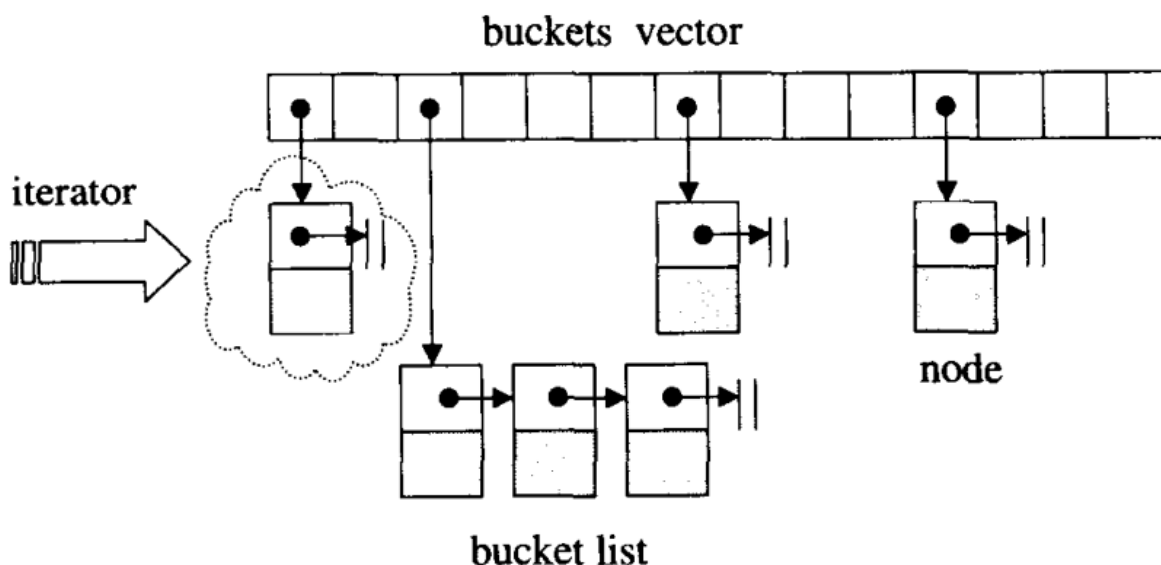
```

```
// 3. 函数指针
bool mycom(const Person& p1, const Person& p2) { // const可以不用，最好有
    return p1.m_Age < p2.m_Age;
}

void test03() {
    // set<Person, less<Person>>s;
    // set<Person, myCompare>s;
    set<Person, decltype(mycom)*>s(mycom); // 注意这里的用法
    Person p1("张三", 18);
    Person p2("李四", 15);
    Person p3("王五", 17);
    Person p4("赵六", 19);
    s.insert(p1);
    s.insert(p2);
    s.insert(p3);
    s.insert(p4);
    for (set<Person>::iterator it = s.begin(); it != s.end(); it++) {
        cout << "姓名: " << (*it).m_Name << " 年龄: " << (*it).m_Age << endl;
    }
    cout << endl;
}
```

## 13.STL中hashtable的实现?

1. 哈希表是一种利用哈希函数进数据存储的数据结构，通过键（Key）映射到哈希表的一个位置来存取，以加快查找的速度。
2. STL中的hashtable使用的是**开链法**解决hash冲突问题。用vector（其中每一个元素也称为一个bucket）作为存储链表的容器。
3. hashtable设计bucket（vector的长度）上，内置了28个质数（53...），在创建hashtable时，会根据存入的元素个数选择大于等于元素个数的质数作为hashtable的容量（vector的长度），bucket中所维持的链表的最大长度等于hashtable的容量。
4. 扩容：如果插入hashtable的元素个数超过了bucket的容量，就要进行重建table操作，即找出下一个质数，创建新的buckets vector，重新计算元素在新hashtable的位置。



hashtable中解决冲突有哪些方法?

## 1.线性探测

使用hash函数计算出的位置如果已经有元素占用了，则向后依次寻找，找到表尾则回到表头，直到找到一个空位

## 2.开链

每个表格维护一个list，如果hash函数计算出的格子相同，则按顺序存在这个list中

## 3.再散列

发生冲突时使用另一种hash函数再计算一个地址，直到不冲突

## 4.二次探测

使用hash函数计算出的位置如果已经有元素占用了，按照 $1^2$ 、 $2^2$ 、 $3^2$ ...的步长依次寻找，如果步长是随机数序列，则称之为伪随机探测

## 5.公共溢出区

一旦hash函数计算的结果相同，就放入公共溢出区

# 14.unordered\_map

### (1) 实现原理

unordered\_map是一个关联式容器，底层是通过hashtable实现，通过hash函数计算元素位置，具有快速查询的功能 ( $O(1)$ )

关联性：键值对 key-value，`(*it).first`为 key，`(*it).second`为 value

无序性：内部无序

键唯一性：不存在两个一样的键

### (2) 使用

```
// 1. 构造
void test01(){
    unordered_map<string, string>first;// 默认构造
    unordered_map<string, string>second({ {"apple","red"}, {"lemon","yellow"} });
}; // 通过数组构造
    unordered_map<string, string>third(second);// 拷贝构造
    unordered_map<string, string>fourth(third.begin(), third.end());// 通过区间的方式
进行构造
}

// 2. 大小
.size(); 返回unordered_map的大小
.empty(); 空返回true，不为空返回false

// 3. 插入和删除
void test02() {
    pair<string, string>mypair("orange", "orange");
    unordered_map<string, string>first({ {"apple","red"}, {"lemon","yellow"} });
    unordered_map<string, string>second = { {"banana","yellow"}, {"pear","white"} };
}; // 初始化，重载了=

// 插入
first.insert(mypair); // 复制插入
first.insert(make_pair<string, string>("strawberry", "red"));// 移动插入
```

```

first.insert(second.begin(), second.end()); //范围插入
first.insert({ "peach", "pink" }); //数组插入(可以用二维一次插入多个元素，也可以用一维
插入一个元素)
first["strawberry"] = "pink"; //数组形式插入

//删除
first.erase(first.begin()); //通过位置，返回下一个迭代器
first.erase("peach"); //通过key，成功返回1，失败返回0

//清空
first.clear();
}

// 4. 修改查找交换
void test03() {
    first.at("apple") = "pink"; //at()
    first["lemon"] = "blue"; //数组的方式

    //查找，找到返回迭代器，失败返回end()
    unordered_map<string, string>::iterator it = first.find("apple");

    //交换
    first.swap(second);
}

```

## 15.unordered\_set

### (1) 实现原理

unordered\_set是一个关联式容器，底层是通过hashtable实现，通过hash函数计算元素位置，具有快速查询的功能 ( $O(1)$ )。与unordered\_map类似，只不过unordered\_set的key和value是相同的

### (2) 使用

```

void test01() {
    //默认构造
    unordered_set<int> us1;
    us1.insert(1);
    us1.insert(1); //相同元素，unordered_set会自动去重，只保留一个
    cout << us1.size() << endl; //插入了相同元素，因此size()的值不是2.而是1

    //拷贝构造
    unordered_set<int> us2(us1);

    //通过数值构造
    unordered_set<int> us3({ 1, 2, 5, 5, 6, 8 });

    //通过迭代器构造
    unordered_set<int> us4(us3.begin(), us3.end());
}

//插入和删除: insert&erase
void test02() {

```

```

unordered_set<int>us1({ 1,2,5,5,6,8 });

//插入
us1.insert(10);
us1.insert(10);//会自动去重

//删除
us1.erase(8);//根据键值删除

unordered_set<int>::iterator pos = us1.find(10);//find() 查找元素，找到返回迭代器，
找不到返回end()

if (pos != us1.end()) {
    us1.erase(pos);//通过迭代器删除
}

//遍历
//方式1
for (auto a : us1) {
    cout << a << " ";
}
cout << endl;
//方式2
for (unordered_set<int>::iterator it = us1.begin(); it != us1.end(); it++) {
    cout << *it << " ";
}
cout << endl;

//count() 查找相应值的个数，因为unordered_set不允许重复，则结果为0或1
cout << us1.count(11) << endl;

//容器大小size()
cout << us1.size() << endl;

//清空容器: clear()
us1.clear();

//判断容器是否为空: empty()
cout << us1.empty() << endl;

//交换两个容器: swap()
unordered_set<int>temp{ 11,12,13 };
us1.swap(temp);

for (auto a : us1) {
    cout << a << " ";
}
cout << endl;
}

```



## 16.STL中unordered\_map和map的区别

1. unordered\_map和map都是存储的key-value的值（根据key计算哈希值），可以通过key快速索引到value。不同的是unordered\_map不会根据key进行排序
2. 使用时map的key需要定义operator<; unordered\_map需要定义hash函数并重载operator==，但是很多系统内置的数据类型都自带这些
3. 如果需要内部元素自动排序，使用map，否则使用unordered\_map
4. map底层是红黑树实现；unordered\_map底层是hashtable实现

## 17.deque

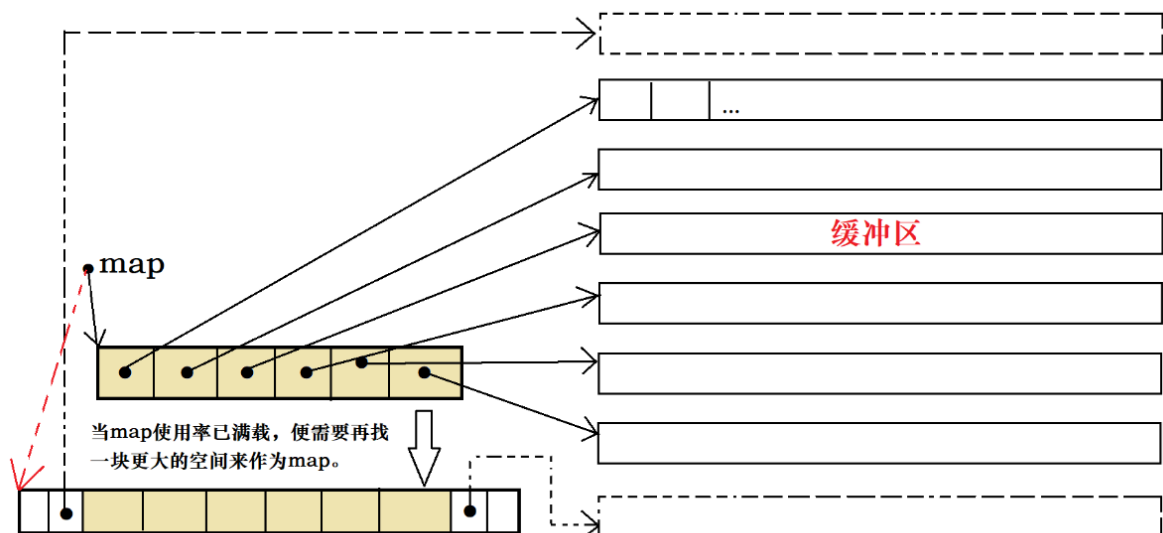
### (1) 实现原理

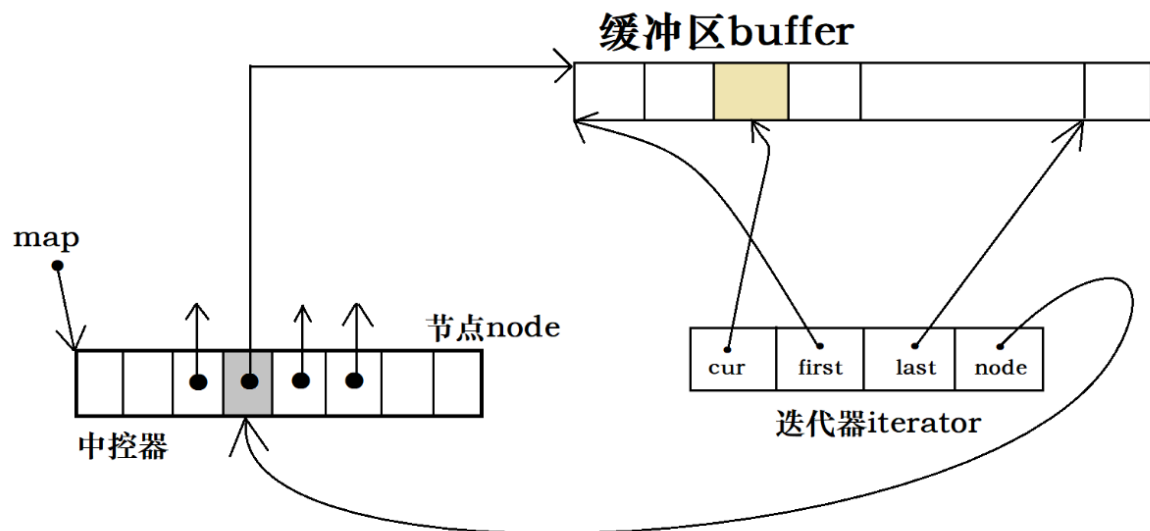
deque（双端队列）是一种双向开口的连续性空间（分段连续）。由中控器（map）和缓存区buffer组成。map是一小块连续空间，其中每个元素（此处称为一个节点，node）都是指针，指向另一段连续空间（大小相等，默认512bytes），称为缓冲区，而缓冲区才是deque的储存空间主体。

deque可以在常数时间对头尾进行元素操作

deque没有容量的概念，它是动态地以分段连续空间组合而成，可以随时增加一段新空间连接起来

通过迭代器维护这些定量连续的空间，提供随机访问的结构，维持整体连续的假象。deque迭代器的“++”、“--”操作是远比vector迭代器繁琐，其主要工作在于缓冲区边界，如何从当前缓冲区跳到另一个缓冲区，当然deque内部在插入元素时，如果map中node数量全部使用完，且node指向的缓冲区也没有多余的空间，这时会配置新的map（2倍于当前+2的数量）来容纳更多的node，也就是可以指向更多的缓冲区。在deque删除元素时，也提供了元素的析构和空闲缓冲区空间的释放等机制。





[https://blog.csdn.net/qq\\_38289815](https://blog.csdn.net/qq_38289815)

## (2) 使用

```
// 1.构造, 同vector
void test01() {
    deque<int>d1;// 默认构造
    for (int i = 0; i < 10; i++) {
        d1.push_back(i);
    }

    deque<int>d2(d1.begin(), d1.end());// 范围构造

    deque<int>d3(10, 100);

    deque<int>d4(d3);
}

// 2.赋值
void test02() {
    deque<int>d1;
    for (int i = 0; i < 10; i++) {
        d1.push_back(i);
    }

    //operator=赋值
    deque<int>d2;
    d2 = d1;

    //assign赋值
    deque<int>d3;
    d3.assign(d1.begin() + 1, d1.end());

    deque<int>d4;
    d4.assign(10, 100);
}

// 3.大小
void test01() {
    deque<int>d1;
    for (int i = 0; i < 10; i++) {
```

```

        d1.push_back(i);
    }

    if (d1.empty()) {
        cout << "d1为空" << endl;
    }

    //从新指定大小
    //d1.resize(15); //默认值0填充
    d1.resize(15, 1);
}

// 4. 插入/删除
push_back();
push_front();
pop_back();
pop_front();

insert(iter, ele); // 在iter位置插入ele
insert(iter, num, ele); // 在iter位置插入num个ele
insert(iter1, iter2_beg, iter2_ed); // 在iter1位置插入[iter2_beg, iter2_ed)元素

erase(iter); // 返回下一有效迭代器的位置
erase(iter_beg, iter_ed);

clear(); // 清空

// 5. 存取, 同vector
[];
at();
front();
back();

// 可以直接使用sort进行排序, 但效率较低, 可以转成vector
sort(d.begin(), d.end());

```

## 18. 常见容器性质总结

1. vector 底层数据结构为数组，支持快速随机访问
2. list 底层数据结构为双向链表，支持快速增删
3. deque 底层数据结构为一个中央控制器和多个缓冲区，支持首尾（中间不能）快速增删，也支持随机访问
4. stack 底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时
5. queue 底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时（stack和queue其实是适配器，而不叫容器，因为是对容器的再封装）
6. priority\_queue 的底层数据结构一般为vector为底层容器，堆heap为处理规则来管理底层容器实现
7. set 底层数据结构为红黑树，有序，不重复
8. multiset 底层数据结构为红黑树，有序，可重复
9. map 底层数据结构为红黑树，有序，不重复
10. multimap 底层数据结构为红黑树，有序，可重复

- 11. unordered\_set 底层数据结构为hash表，无序，不重复
- 12. unordered\_multiset 底层数据结构为hash表，无序，可重复
- 13. unordered\_map 底层数据结构为hash表，无序，不重复
- 14. unordered\_multimap 底层数据结构为hash表，无序，可重复

## 19.说一下STL每种容器对应的迭代器

容器	迭代器
vector、deque	随机访问迭代器
stack、queue、priority_queue	无
list、(multi)set/map	双向迭代器
unordered_(multi)set/map、forward_list	前向迭代器

## 20.STL中stack和queue的实现

stack（栈）是一种先进后出（First In Last Out）的数据结构，只有一个入口和出口，都是栈顶，除了获取栈顶元素外，没有其他方法可以获取到内部的其他元素，不提供迭代器。底层由deque/list实现，默认使用deque。

queue（队列）是一种先进先出（First In First Out）的数据结构，只有一个入口和一个出口，分别位于最底端和最顶端，出口元素外，没有其他方法可以获取到内部的其他元素，不提供迭代器。底层由deque/list实现，默认使用deque。

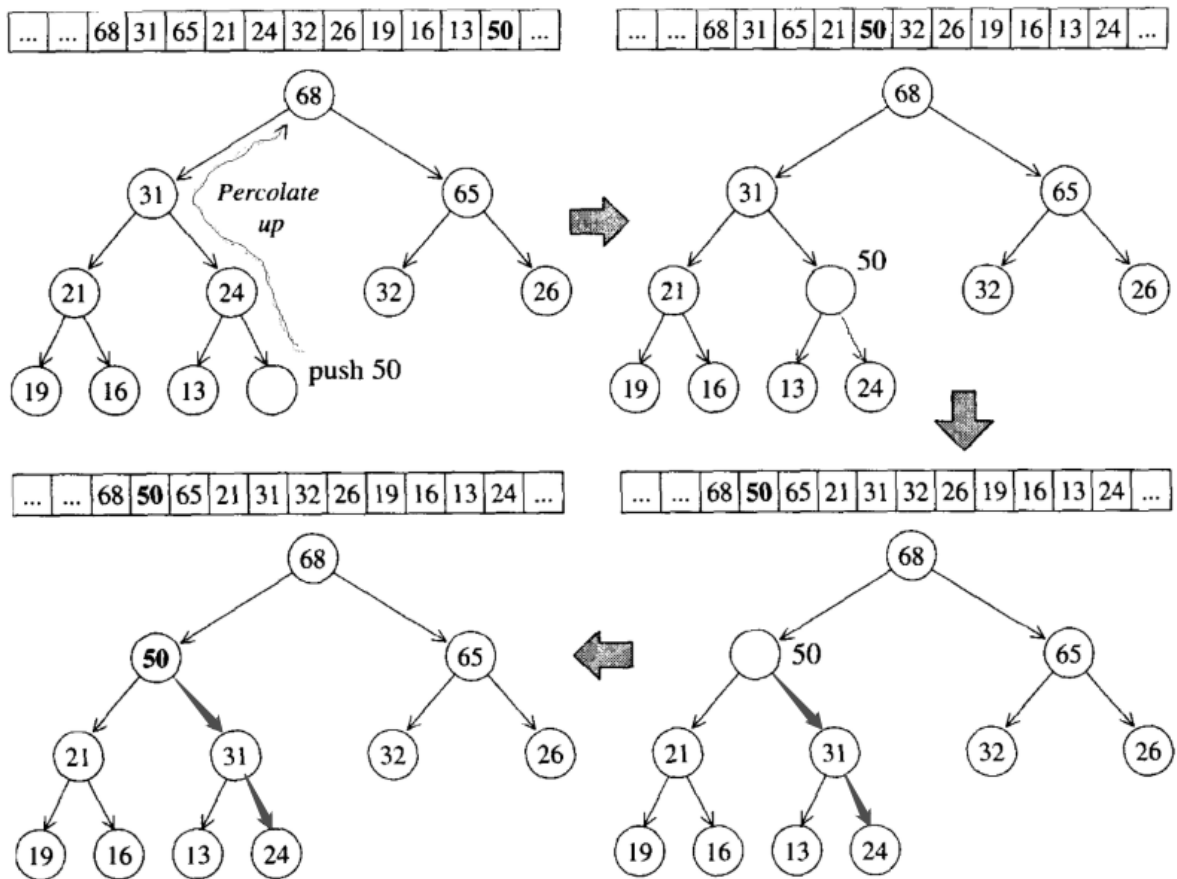
```
// stack和queue用法相同：  
push();  
pop();  
empty();
```

## 21.STL中的heap的实现

heap（堆）并不是STL的容器组件，而是priority\_queue（优先队列）的底层实现机制。heap本质上是一个完全二叉树，并且根节点值最大。由于完全二叉树的性质，底层可以用一个vector来实现（从根节点开始从上到下，从左到右，依次编号0 1...，与数组下标对应，i的左右孩子分别为2i+1和2i+2，i的父节点为(i-1)/2)

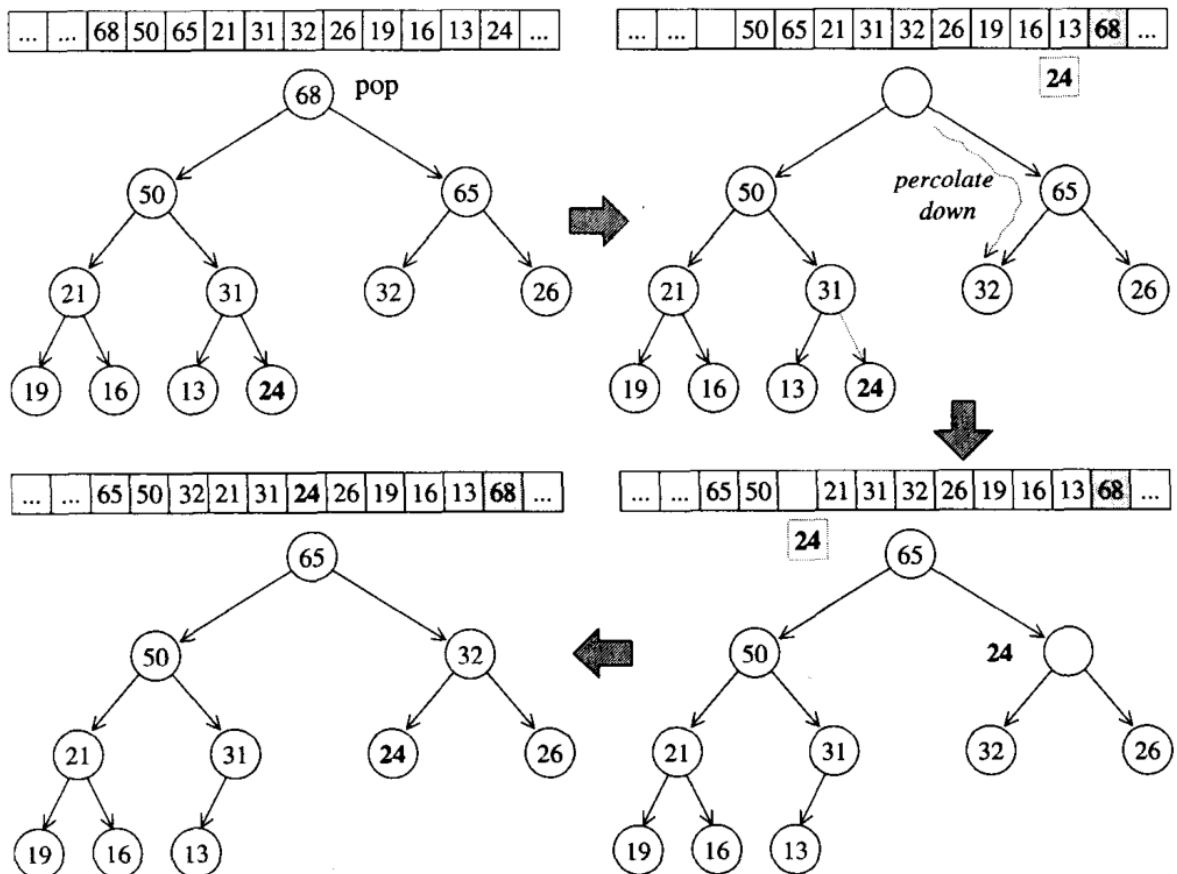
**push\_heap插入算法：**

新插入的元素位于完全二叉树的【最后】一个节点，然后与父节点比较大小，一直向上，直到新插入元素比父节点小为止，否则与父节点交换



### pop\_heap算法:

弹出根节点，其实并没有真正弹出，只是将其移动到vector的最后位置。过程是将根节点与最后位置的子节点交换，然后向下调整这个子节点的位置，使其移动到合适的位置



## 22.STL中的priority\_queue的实现

priority\_queue, 优先队列, 底层是由heap实现, 在插入元素时, 元素会自动排序

```
// 1.头文件
#include <queue>      //优先队列

// 2.用法
priority_queue< type, container, function >
    type: 数据类型
    container: 实现优先队列的底层容器, 必须是数组形式的实现容器, 如vector、deque, 不能是list
    function: 元素之间的比较方式
    默认情况下 (省略后面两个参数) 是以vector为容器, 以 operator< 为比较方式, 所以在只使用第一个参数时, 优先队列默认是一个最大堆, 每次输出的堆顶元素是此时堆中的最大元素

// 3.成员函数
empty();
size();
pop();
top();
push();

// 4.大根堆
priority_queue<int> big_heap;
priority_queue<int,vector<int>,less<int> > big_heap2;

// 5.小根堆
priority_queue<int, vector<int>, greater<int> > small_heap;
    注意使用less<int>和greater<int>, 需要头文件functional (也可以没有)
```

## 23.C++模板是什么, 你知道底层怎么实现的?

模板技术是C++泛型编程的基础, 提高代码的复用性。提供一个模板, 让不同类型的代码可以根据模板代码生成与其对应的代码。提供了两种模板机制: 函数模板和类模板

### (1) 函数模板

```
// 1.函数模板的格式
template<typename T>      // typename可以换成class, 可以有多个模板参数
T Add(T left, T right)
{
    return left + right;
}

// 2.函数模板的实例化
void test01() {
    int a = 10;
    int b = 20;
    double d = 1.2;
    Add(a, b); // 隐式实例化, 让编译器根据实参推演模板参数的实际类型

    // 在模板中, 编译器一般不会进行类型转换操作
    // a的类型为int, d的类型为double, 但是模板参数列表中只有一个T,
    // 编译器无法确定此处应该将T确定为int还是double
```

```

Add(a, d); // 编译不通过

// 两种方法解决上述问题：强制类型转换、使用显式实例化
Add((double)a, d); // 强制类型转换

Add<double>(a, d); // 显式实例化
}

// 3. 模板参数的匹配原则
// 非模板函数和函数模板可以重名，相同条件下会优先调用非模板函数，而不会从该模板中产生一个实例
// 如果模板可以产生一个匹配更好的函数，会选择模板
int Add(int a, int b) {
    return a + b;
}

void test02() {
    int a = 10;
    int b = 20;
    Add(a, b); // 调用非模板函数

    Add(double(a), double(b)); // 调用实例化的模板函数
}

// 模板函数不允许自动类型转换，但普通函数可以进行自动类型转换

// 4. 声明与定义可以分离，声明和定义都要给出模板参数声明
template<typename T> // 声明
T Add(T left, T right);

template<typename T> // 定义
T Add(T left, T right)
{
    return left + right;
}

// 5. 使用默认类型参数
template<typename T=int>
// 可以推导出参数类型就是用推导出的
// 无法推导出参数类型，那么编译器就会用默认模板参数
// 无法推导出模板参数类型也没有默认的就会报错

```

### (3) 类模板

```

// 1. 类模板个数
template <class NameType, class AgeType> // 语句1
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        m_Name = name;
        m_Age = age;
    }

    void showPerson() {
        cout << "m_Name:" << m_Name << " m_Age:" << m_Age << endl;
    }
}

```

```

    }
private:
    NameType m_Name;
    AgeType m_Age;
};

// 2.成员函数类内声明，类外定义
template <class NameType,class AgeType>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        m_Name = name;
        m_Age = age;
    }

    void showPerson(); // 类内声明
private:
    NameType m_Name;
    AgeType m_Age;
};

template <class NameType,class AgeType> // 类外定义
void Person<NameType, AgeType>::showPerson() {
    cout << "m_Name:" << m_Name << " m_Age:" << m_Age << endl;
}

// 3.类模板(C++11之后函数模板也可以)在模板参数列表中可以有默认参数
// 将语句1改为
template <class NameType=string,class AgeType=int> // 默认参数放在最右边
Person<>p("猪八戒", 999); // <>不可省

// 类模板不会进行自动类型推导

```

#### (4) 模板的特化

编写单一的模板，它能适应多种类型的需求，使每种类型都具有相同的功能，但对于某种特定类型，如果要实现其特有的功能，单一模板就无法做到，这时就需要模板特例化

模板特化：对单一模板提供的一个特殊实例，它将一个或多个模板参数绑定到特定的类型或值上

```

// 1.模板函数特化：
// 必须为原函数模板的每个模板参数都提供实参
// 且使用关键字template后跟一个空尖括号对<>，表明将原模板的所有模板参数提供实参
// 函数特化的本质是模板的实例化，而非重载它，特例化不影响参数匹配，参数匹配都以最佳匹配为原则
// 模板和特例化应该声明在同一个头文件中，模板在前，特例化版本在后

//泛型版本
template <class T>
int compare(const T &v1, const T &v2){
    if(v1 < v2) return -1;
    if(v2 > v1) return 1;
    return 0;
}

```



```
//为实参类型 const char * 提供特化版本
template <>      // 空模板形参表
int compare<const char *>(const char * const &v1, const char * const &v2){
    return strcmp(v1, v2);
}
```

// 2. 类模板的特化，分为全特化和偏特化

// 基础

```
template <class T1, class T2>
class Date{
public:
    Date(){
        cout << "Date<T1, T2>" << endl;
    }
private:
    T1 _d1;
    T2 _d2;
};
```

// 全特化：对类模板参数列表的类型全部都确定（明确指定）

```
template<>
class Date<int, double>{
public:
    Date(){
        cout << "Date<int, double>" << endl;
    }
private:
    int _d1;
    double _d2;
};
```

// 偏特化： 堆类模板的参数列表中部分参数进行确定化分为部分特化和参数进一步限制

// 部分

// 偏特化/半特化，仍然是一个模板

```
template<class T1>
class Data <T1, char>//只要第二个类型是char就匹配这个
{
public:
    Data() { cout << "Data<T1, char>" << endl; }
private:
};
```

// 对模板参数更进一步的条件限制

// 偏特化/半特化：不一定指的是特化部分参数，而是对模板参数类型的进一步限制

```
template<class T1, class T2>
class Data <T1*, T2*>{
public:
    Data() { cout << "Data<T1*, T2*>" << endl; }
private:
};
```

```
template<class T1, class T2>
class Data < T1&, T2&>{
```

```

public:
    Data() { cout << "Data<T1&, T2&>" << endl; }
private:
};

template<class T1, class T2>
class Data < T1&, T2* >{
public:
    Data() { cout << "Data<T1&, T2*>" << endl; }
private:
};

```

## (5) 分文件编写

分离编译：一个程序（项目）由若干个源文件共同实现，而每个源文件单独编译生成目标文件，最后将所有目标文件链接起来形成单一的可执行文件的过程称为分离编译模式

一个项目创建三个文件，一个头文件（Func.h）用于进行函数声明，一个源文件（Func.cpp）用于对头文件中声明的函数进行定义，最后一个源文件（Test.cpp）用于调用头文件当中的函数

**模板是不支持分离编译的，会出现链接错误，即不能将模板的声明写到.h文件中，定义写到.cpp文件中**

原因如下：



程序的编译过程分为四步：预处理、编译、汇编、链接。预处理阶段会进行头文件展开，生成.i文件，然后各个.i文件（预处理之后的.cpp文件）**分别独立**进行编译、汇编，生成目标文件（.o文件）。

最后在链接阶段，因为之前的过程没有对模板进行实例化，没有函数地址，在链接的时候，调用模板的地方无法通过名字在别的文件中找到模板的地址，也就无法实现模板的分离编译

**解决办法：**

1. 将模板的声明和定义写在一个文件中（.hpp文件）

## (6) 优缺点

优点：

1. 模板复用了代码，节省资源，更快的迭代开发，C++的标准模板库（STL）因此而产生
2. 增强了代码的灵活性

缺陷：

1. 模板会导致代码膨胀问题，也会导致编译时间变长
2. 出现模板编译错误时，错误信息非常凌乱，不易定位错误

## 24.模板类和模板函数的区别是什么？

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。即**函数模板允许隐式调用和显式调用而类模板只能显示调用**。在使用时类模板必须加，而函数模板不必。

## 25.模板会写吗？写一个比较大小的模板函数

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

template <typename T1,typename T2>
T1 m_max(const T1& a, const T2& b) {

    return a > b ? a : b;
}

int main()
{
    int a = 1;
    double b = 2.1;

    cout << m_max(a, b) << endl;

    return 0;
}
```

// 其实该模板有个比较隐晦的bug，那就是a、b只有在能进行转型的时候才能进行比较，否则 `a > b` 这一步是会报错的。这个时候往往需要对于 `>` 号进行重载，这代码量瞬间上来了。

# 十、其他

## 1.C++中标准库是什么？

### (1) C++标准库简介

1. C++标准库是类库和函数的集合（不是C++语言标准的一部分）
2. C++标准库是编译器厂商根据C++标准委员会官方的ISO规范并将其转化为代码。必须依赖其不同操作系统所提供的系统调用接口，因此每个平台都有其自己的C++标准库实现
3. C++标准库中定义的类和对象都位于std命名空间中
4. C++标准库的头文件都不带.h后缀，但C++编译器会附带提供C语言兼容库，C语言兼容库头文件带.h后缀，如#include <stdio.h>
5. C++标准库中包含一个涵盖C库功能的子库，通常头文件以c开头，如#include <cmath>

### (2) C++标准库组成

C++标准库是一组C++模板类，提供了通用的编程数据结构和函数，如链表、堆、数组、算法、迭代器等C++组件。C++标准库包含了C标准库，并在C++标准中进行了定义。

C++ 标准库包含了三个部分：

1. C 标准库的 C++ 版本；
2. C++ IO 库；
3. C++ STL

C++标准库的内容分为10类：

1. 语言支持，头文件，头文件描述，< new>：支持动态内存分配
2. 流输入/输出，< iostream>支持标准流cin、cout、cerr和clog的输入和输出，还支持多字节字符标准流wcin、wcout、wcerr和wclog。
3. 诊断功能，< stdexcept>：定义标准异常
4. 工具函数，< ctime>：支持系统时钟函数
5. 字符串处理，< string>：为字符串类型提供支持和定义，包括单字节字符串(由char组成)的string和多字节字符串
6. 容器类模板，< vector>
7. 迭代器，< iterator>
8. 算法，< algorithm>：提供一组基于算法的函数，包括置换、排序、合并和搜索
9. 数值操作，< numeric>：在数值序列上定义一组一般数学操作，例如accumulate
10. 本地化，< locale>：提供的本地化包括字符类别、排序序列以及货币和日期表示

## 2.你知道Debug和Release的区别是什么吗？

两种编译模式：

1. 调试版本，包含调试信息，不进行任何优化，便于程序员调试。Debug模式下生成两个文件，除了.exe或.dll文件外，还有一个.pdb文件，该文件记录了代码中断点等调试信息；
2. 发布版本，不对源代码进行调试，编译时对应用程序的速度进行优化，使得程序在代码大小和运行速度上达到最优，Release模式下只生成一个文件.exe或.dll文件
3. 实际上，Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本

## 3.什么是一致性哈希？

### (1) 哈希函数

哈希函数/散列函数是一个不可逆的单向映射，将任意长度的输入信息映射成较短的哈希值。

哈希函数特点：

1. 输入范围无限；输出范围有限
2. 相同的输入有相同的输出
3. 不同的输入可能有相同的输出（哈希碰撞）
4. 离散性：将不同的输入（可能很相近）离散到输出域的不同位置上；均匀性：输出均匀分布在输出域上的。离散性/均匀性越好，就认为哈希函数越好

## (2) 普通哈希算法

在经典的分布式缓存的应用场景中，普通哈希的做法是：假设有n台服务器(编号从0..n-1)，以图片的名称作为key进行缓存

1.  $\text{hash}(\text{图片名称}) \% n$
2. 根据计算结果将该图片存放到对应的服务器中

普通哈希存在的缺陷：当服务器数量改变时，所有缓存在一定时间内失效，当应用无法从缓存中获取数据时，则会向后端服务器请求数据，后端服务器将会承受巨大的压力，整个系统很有可能被压垮

## (3) 一致性哈希

一致性哈希算法是一种特殊的哈希算法，目的是为了克服传统哈希分布在服务器节点数量变化时大量数据迁移的问题。

原理：

1. 普通哈希时对服务器数量进行取模，一致性哈希是对  $2^{32}$  取模
2. 一致性哈希算法将整个哈希值空间（取模）按照顺时针方向组织成一个虚拟的圆环，称为 Hash 环；
3. 将服务器进行哈希（IP或主机名），确定在哈希环上的位置
4. 使用相同的哈希函数对数据进行哈希，确定在哈希环上的位置，从此位置顺时针寻找，遇到的第一台服务器就是该数据缓存的服务器

优点：

1. 当服务器数量放生变化时，只有一部分缓存失效，具有较好的容错性和可扩展性
2. 增加节点时，只有增加节点位置逆时针旋转到最近的一个节点之间的缓存失效
3. 移除节点时，只有移除节点位置逆时针旋转到最近的一个节点之间的缓存失效

hash环的倾斜：

一致性哈希算法在服务节点太少的情况下，容易因为节点分布不均而造成数据倾斜问题，也就是被缓存的对象大部分集中缓存在某一台服务器上，从而出现数据分布不均匀的情况，这种情况就称为hash环的倾斜。

虚拟节点机制可以解决哈希环倾斜问题：对每个服务器多次计算哈希值，得到多个节点位置（虚拟节点），每个实际物理节点对应多个虚拟节点，虚拟节点越多，hash环上的节点就越多，缓存被均匀分布的概率就越大，hash环倾斜所带来的影响就越小，同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射。

# 4.C++从代码到可执行程序经历了什么？

## (1) 预编译

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下：

1. 删除所有的#define，展开所有的宏定义
2. 处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”
3. 处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件
4. 删除所有的注释，“//”和“/\*\*/”

5. 保留所有的#pragma编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重复引用
6. 添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译错误或警告时能够显示行号

## **(2) 编译（词法分析、语法分析、语义分析及优化，生成相应的汇编代码文件）**

把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。

1. 词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分割成一系列的记号
2. 语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树
3. 语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分析的语义，相对应的动态语义是在运行期才能确定的语义
4. 优化：源代码级别的一个优化过程
5. 目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示
6. 目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等

## **(3) 汇编**

将汇编代码转变成机器可以执行的指令(机器码文件)。汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表——翻译过来，汇编过程由汇编器as完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Linux下)、xxx.obj(Window下)。

## **(4) 链接**

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

### **静态链接：**

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据，并把他们和应用程序的其它模块组合起来创建最终的可执行文件。

缺点：

1. 浪费空间，每个可执行程序中对所有需要的目标文件都需要一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一目标文件在内存中存在多个副本的情况；
2. 更新困难，当库文件代码修改时，需要重新进行汇编连接形成可执行文件

优点：

1. 运行速度快，可执行文件中具备所有执行程序所需要的东西，执行的时候速度快

### **动态链接：**

动态链接的基本思想是把程序按照模块拆分成相互独立的部分，在程序执行时才将他们链接在一起，形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件

共享库：即使多个程序都依赖同一个库，该库也不会像静态链接那样在内存中存在多个副本，而是多个程序在执行时共享一个副本

优点：

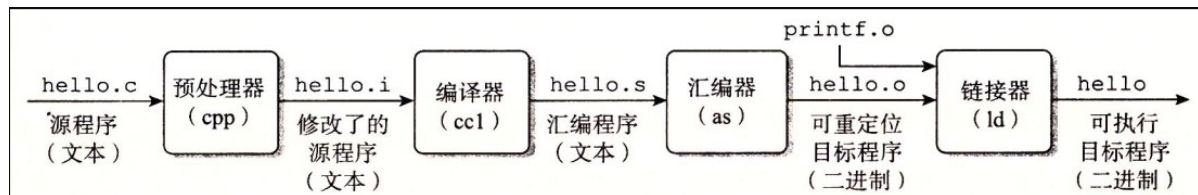
1. 更新方便，更新时只需要替换原来的目标文件，无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标

缺点：

1. 性能损耗，因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失（5%）

## 5.hello.c 程序的编译过程

```
#include <stdio.h>
int main()
{
    printf("hello, world\n");
    return 0;
}
// 在Unix系统上，由编译器把源文件转换为目标文件
gcc -o hello hello.c
```

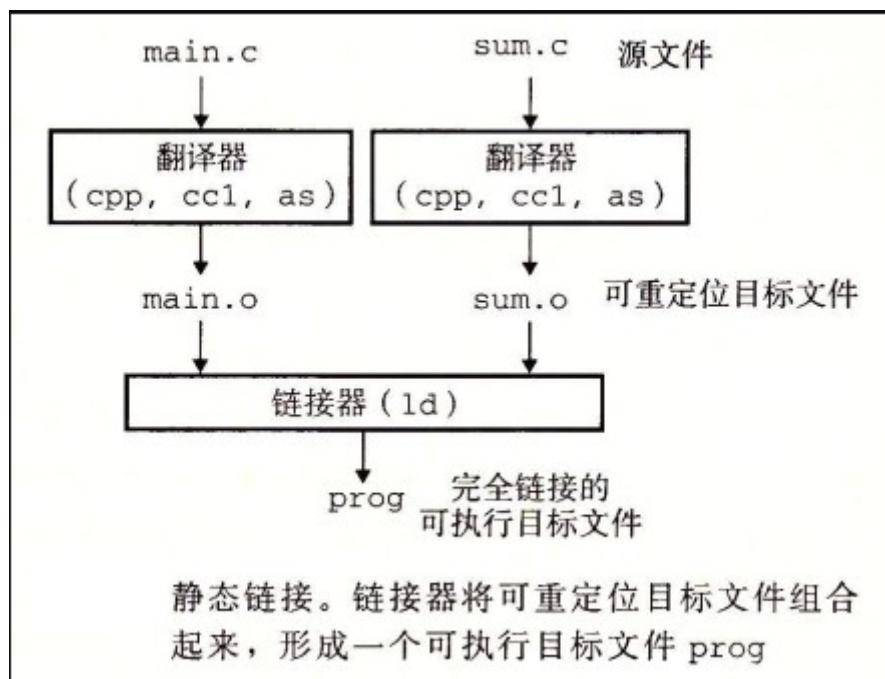


1. 预处理阶段：处理以 # 开头的预处理命令；
2. 编译阶段：翻译成汇编文件；
3. 汇编阶段：将汇编文件翻译成可重定位目标文件；
4. 链接阶段：将可重定位目标文件和 printf.o 等单独预编译好的目标文件进行合并，得到最终的可执行目标文件。

### 静态链接

静态链接器以一组可重定位目标文件为输入，生成一个完全链接的可执行目标文件作为输出。链接器主要完成以下两个任务：

1. 符号解析：每个符号对应于一个函数、一个全局变量或一个静态变量，符号解析的目的是将每个符号引用与一个符号定义关联起来
2. 重定位：链接器通过把每个符号定义与一个内存位置关联起来，然后修改所有对这些符号的引用，使得它们指向这个内存位置



### 动态链接

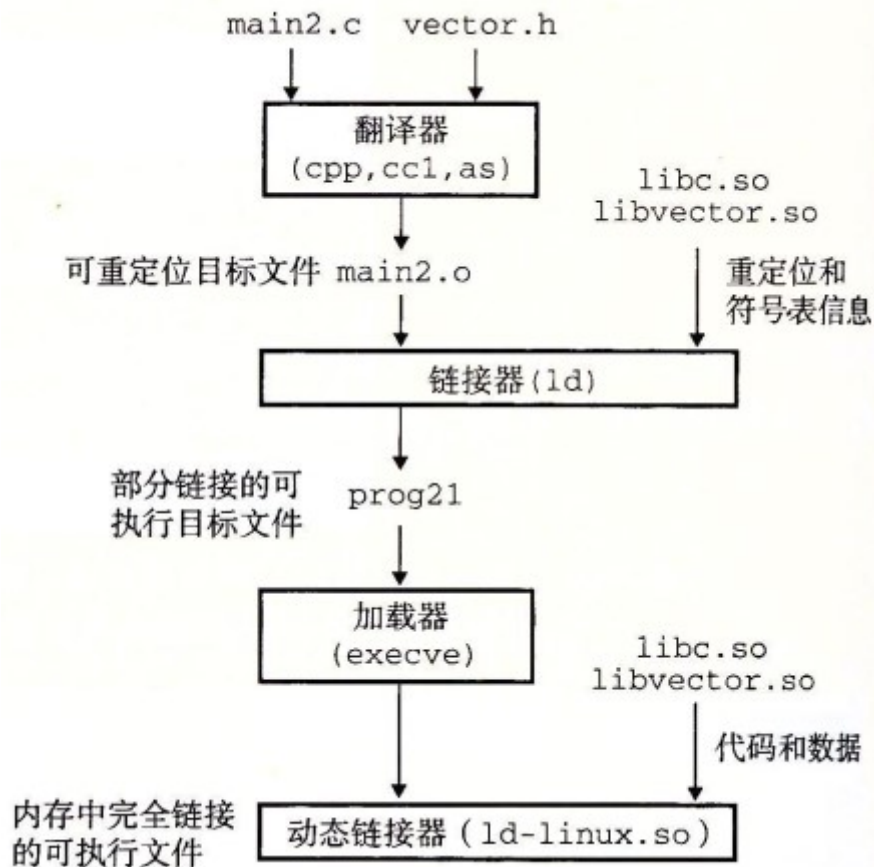
静态库有以下两个问题：

1. 当静态库更新时那么整个程序都要重新进行链接；
2. 对于printf这种标准函数库，如果每个程序都要有代码，这会极大浪费资源

共享库是为了解决静态库的这两个问题而设计的，在Linux系统中通常用.so后缀来表示，Windows系统上它们被称为 DLL。它具有以下特点：

1. 在给定的文件系统中一个库只有一个文件，所有引用该库的可执行目标文件都共享这个文件，它不会被复制到引用它的可执行文件中；
2. 在内存中，一个共享库的.text段（已编译程序的机器代码）的一个副本可以被不同的正在运行的进程共享





源代码 - ->预处理 - ->编译 - ->优化 - ->汇编 - ->链接->可执行文件

#### 1. 预处理

读取c源程序，对其中的伪指令（以#开头的指令）和特殊符号进行处理。包括宏定义替换、条件编译指令、头文件包含指令、特殊符号。预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。i.预处理后的c文件，.ii.预处理后的C++文件。

#### 1. 编译阶段

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。.s文件

#### 1. 汇编过程

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个C语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。.o目标文件

#### 1. 链接阶段

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够诶操作系统装入执行的统一整体。

## 6.动态编译与静态编译

1. 静态编译，编译器在编译可执行文件时，把需要用到的动态链接库中的对应部分提取出来，链接到可执行文件中去，使可执行文件在运行时不需要依赖于动态链接库；

2. 动态编译的可执行文件需要附带一个动态链接库，在执行时，需要调用其对应动态链接库的命令。所以其优点一方面是**缩小了执行文件本身的体积**，另一方面是**加快了编译速度，节省了系统资源**。缺点是哪怕是很简单的程序，只用到了链接库的一两条命令，也**需要附带一个相对庞大的链接库**；二是如果其他计算机上**没有安装对应的运行库**，则用动态编译的可执行文件就**不能运行**。

## 十一、待补充

---

62.写C++代码时有一类错误是 coredump，很常见，你遇到过吗？怎么调试这个错误？

gdb调试，待补充

68.函数调用原理（栈、汇编）

待补充

参考66、76、85

72.你知道printf函数的实现原理是什么吗？

待补充，87

73.cout和printf有什么区别？

6.简单说一下traits技法

7.

7.STL的两级空间配置器

8.21

11.STL迭代器如何实现

13.

16.如何在共享内存上使用STL标准库？

15.

17.将字符串“hello world”从开始到打印到屏幕上的全过程？

26.