C和C++的区别

C和C++的区别

- 基础语法没用太大的区别?
- C++ 多出一下语法和关键字，引用、new/delete、auto、explicit等
- C++ 多了重载和虚函数
- C++ 面向对象的部分，类、继承、对象
- C++ 和 C 的设计思想和应用场景。处于对内存和执行效率的考虑，C优点是精简和高效，多用于操作系统和内核驱动；C++ 设计之初的目的就是把 C 繁杂的实现过程抽象成类，并且进行实例化管理，更适用做大型软件。C 更注重逻辑实现，C++ 更适合程序的整体设计。

基础语法

C++11 特性

- 类型推导：auto && decltype
- 左值右值：
- 类别初始化：
- function && bind && lambda
- 模板改进：
- 并发：
- 智能指针：
- 基于范围的for循环
- 委托构造函数
- 继承构造函数
- nullptr
- final && override
- default && delete
- explict
- constexpr

C++ 从代码到可执行函数经历了什么？（预编译、编译、汇编、链接（静态链接、动态链接））

- 预编译：处理以 # 号开头的预编译命令，#define、#if、#include，删除注释、添加行号和文件标识等；
- 编译：把预编译生成的 xxx.i 或者 xxx.ii 文件进行一系列的词法分析、语法分析、语义分析及优化之后生成相应的汇编文件；
- 汇编：把汇编代码翻译成机器码文件，生成 xxx.o 或者 xxx.obj 文件
- 链接：静态链接、动态链接
- 可执行程序

---

hello.c 的编译过程？（静态链接、动态链接）

1. 预处理阶段：处理以 # 号开头的预处理命令；

2. 编译阶段：翻译成汇编文件；
3. 汇编阶段：将汇编文件翻译成重定位目标文件；
4. 链接阶段：将可重定位目标文件和有关文件链接起来生成可执行文件；

Windows 和 Linux 环境下内存分布情况

用户空间内存，从低到高，分别是 7 种不同的内存段

- 程序文件段（.text）：包括二进制可执行代码；
- 常量区：存放常量字符串，程序推出后由 OS 释放；
- 全局区（静态区）：
    - 已初始化数据段（.data）：包括静态变量；
    - 未初始化数据段（.bss）：包括未初始化的静态变量；
- 堆段：包括动态分配的内存，从低地址开始增长；
- 文件映射段：包括动态库，共享内存等，从低地址开始向上增长（跟硬件和内核版本有关）；
- 栈段：包括局部变量和函数调用的上下文。

C++ 中类数据成员和成员函数内存分布情况?

- 函数都放在代码区；
- 全局数据和静态数据在全局数据区，局部变量放在栈区；

---

main执行前和执行后的代码可能是什么?

- main 函数执行之前，注意是为了初始化系统相关资源：
    - 设置栈指针；
    - 初始化静态 static 变量和 global 全局变量，即 .data 段的内容；
    - 将未初始化部分的全局变量赋初始值，即 .bss 内容（数值型 short、int、long 为 0，bool 型为 false、指针为 null 等等）
    - 全局对象初始化，在 main 之前调用构造函数，这是可能执行前的一些代码；
    - 将 main 函数参数 argc、argv 等传递给 main 函数，然后才真正运行 main 函数；
    - attribute(constructor) //构造函数
- main 函数执行之后：
    - 全局的析构函数会在 main 执行；
    - 可以注册一个 atexit 函数，它会在 main 之后执行；
    - attribute((destructor))

C++ 中 struct 和 class 的区别?

1. C++ 的 struct 和 calss 的主要区别是成员默认权限不同，struct 默认公有的，class 默认私有的；

C 中 struct 和 C++ struct 的区别?

1. C 中 struct 是用户自定义数据类型（UDT）；C++ 是struct 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，且 struct 中只能是一些变量的集合体，可以封装数据却不能隐藏数据，而且成员不能是函数；
3. 在 C++ 中 struct 增加了访问权限，且和类一样有成员函数，成员访问默认说明符是 public（为了和 C 兼容）
4. struct 作为类的一种特例用来自定义数据结构，一个结构标记声明后，在 C 中必须在结构标记前加上 struct，才能做到结构类型名（typdef struct class{};）；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；

形参和实参的区别?

- 新参变量只有在被调用的时候才会分配内存单元，在调用结束时即可释放内存单元，因此，形参只有在函数内部有效，函数调用结束返回主调函数后则不能使用该形参变量
- 实参可以是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，他们都必须是确定的值，以便把这些值传递给形参，因此应预先用赋值、输入等方法使实参获得确定的值
- 实参和形参在数量上，类型上、顺序上应该保持严格一致，否则会发生"类型不匹配错误"
- 函数调用中发生的数据传送是单向的，即只能把实参的值传递给形参，而不能把形参的值反向的传递给实参，因此在函数调用过程中，形参的值发生变化不会影响到实参
- 当形参和实参不是指针类型时，在函数运行时，形和参实参是不同的变量，他们在内存中位于不同的位置，形参将实参传递过来的内容复制一份，而在该函数运行结束时形参被释放，而实参内容不会改变

结构体内存对齐问题?

- 结构体成员每个成员的相对结构体首地址的偏移量是对齐参数的整数倍，首位元素是对齐参数的 0 倍；
- 结构体变量所占空间大小是对齐参数的整数，如有需要在最后一个成员末尾填充字节达到该要求；

内存对齐的理解以及理解?

- 分配内存的顺序是按照变量声明的顺序
- 每个变量相对于起始位置的偏移量必须是该变量类型的整数倍，不是整数倍的空出内存，直到偏移量为整数倍为止
- 最后整个结构体的大小必须是里面变量类型最大值的整数倍

如何获取结构体成员相对于结构体开头的字节偏移量?

- (unsigned long)(&(type.Member)) - (unsigned long)(&(type))

判断结构体比较是否相等?

- 重载 "==" 操作符
- 对结构体变量一个个比较

- 指针直接比较，如果保存的是同一个实例，则两个指针保存的地址是一样的

1. 如果是想对结构体内容进行比较，就重载 == 操作符，在函数中对结构体变量一一比较；
2. 如果是想判断是否是同一个结构体的话，直接对两个指针进行比较，如果是同一个实例的话，则两个指针保存的地址是一样的；
3. 不能使用 memcmp 的内存比较函数，因为结构体会有内存对齐和内存补齐。

指针和引用的区别?

- 指针是一个实体，需要分配内存；引用只是变量的别名，不需要分配内存空间；
- 引用在定义时候必须进行初始化，并且不能改变；指针在定义的时候不一定要进行初始化，并且指向的空间可变；
- 有多级指针，但是没有多级引用。只能有一级引用；
- 指针和引用的自增运算结果不一样；（指针是指向下一个空间，引用是引用的变量值加 1 ）；
- sizeof 引用得到的结果是引用所指向的变量（对象）的大小；sizeof 指针得到的指针本身的大小；
- 使用指针前最后做类型检查，防止野指针的出现；
- 引用底层是用指针实现的；
- 作为参数时，传指针实际上是传值，传递的值是指针的地址；传引用的实质是传地址，传递的是变量的地址；

值传递、指针传递、引用传递的区别和效率?

- 值传递：有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象，或是大的结构体对象，将耗费一定的时间和空间。（传值）

- 指针传递：同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为 4 字节的地址。（传值，传递的是地址值）

- 引用传递：同样有上述数据的拷贝过程，但是针对的地址，相当于为该数据所在的地址起了一个别名

- 从效率上说，指针传递和引用传递的效率要比值传递的效率要高，一般主张使用引用传递，代码逻辑上更加紧凑，清晰；

将引用作为函数参数的好处是什么?

- 传递引用给函数与传递给指针的效果是一样的：此时被调函数的形参就成为原来主调函数的实参变量或者对象的一个变量的一个别名使用，所以在被调函数中对形参变量的操作就是对其目标对象（在主调函数中）的操作；
- 使用引用传递函数的参数，在内存中没有产生实参的副本，它是直接对实参操作；
    - 使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配内存单元，形参变量是实参变量的副本；
    - 如果传递的是对象，还将调用拷贝拷贝函数，因此当参数传递的数据较大时，用引用比一般变量传递参数的效率和所占空间都好
- 使用指针作为函数参数虽然也能达到和引用的效果，但是在被调函数中同样要给形参分配存储单元，且需要重复使用 "*指针变量名"的形式进行运算，这样容易产生错误且程序的阅读性较差；另一方面在主调函数的调用点处，必须用变量的地址作为实参，而引用更容易使用，也更清晰。

C++ 指针参数传递和引用参数传递有什么区别？底层原理是什么？

- 指针参数传递的本质是值传递，它所传递的是一个地址值；
    - 值传递的过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存以存放由主调函数传递进来的实参值，从而形成一个实参的一个副本（替身）
    - 值传递的特定是，被调函数对形式参数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（形参指针变了，实参指针不会变）。
- 引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放是由主调函数放进来的实参变量地址；
    - 被调函数对形参（本体）的任何操作都会被处理成间接寻址，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的实参变量。
- 引用传递和指针传递是不同的
    - 虽然它们都是在被调函数栈空间上的一个局部变量，但是任何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；
    - 而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想要通过指针参数传递来改变主调函数中的相关变量（地址—），那就得使用指向指针的指针或者指针引用了；
- 从编译的角度来说，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以及变量所对应的地址
    - 指令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（与实参名字不同，地址相同）
    - 符号表生成之后就不会再更改，因此指针可以改变其指向的对象（指针变量中的值可以该），而引用对象则不能修改

你什么时候用指针当参数，什么时候用引用当参数，为什么？

- 使用引用参数两个原因：
    - 程序员能修改调用函数中的数据对象；
    - 通过传递引用而不是整个数据对象，可以提高程序的运行速度
- 一般使用的原则：
    - 对于使用引用的值而不做修改的函数：
        - 如果数据对象很小，如内置数据类型或者小型结构，则按照值传递；
        - 如果数据对象是数组，则使用指针（唯一的选择），并且指针声明为指向 const 的指针；
        - 如果的数据对象是较大的结构，则使用 const 指针或者引用，以提高程序的效率。可以节省结构所需的时间和空间；
        - 如果数据对象是类对象，则使用 const 引用（传递类对象参数的标准方式按照引用传递）；
    - 对于修改函数中数据的函数：
        - 如果数据是内置数据类型，使用指针；
        - 如果数据对象是数组，则只能使用指针；
        - 如果数据对象是结构，则使用引用或者指针；

在汇编层面上解释一下引用？

- 引用在底层是用指针，更准确的是用地址实现的
- 但从高级语言一级来说，指针和引用没有关系，引用就是另一个变量的别名

- 引用占据一个内存空间，存储一个地址，一般是4个字节

---

const 的用法？（常量指针和指针常量的区别，C++ 中顶层 const 和底层 const）

- const 修饰变量、数组、指针、函数参数、函数返回值、引用都是对修饰对象的声明为只读属性，不允许修改 const 所修饰的对象
- const 修饰类成员函数、类限制成员函数和类对类成员变量的修改
- 常量指针：是 const 修饰的常量的指针，int const *p/const int* p，*p的值不可变
- 指针常量：是 const 修饰的不能改变指向的指针，int* const p，p值不可变

---

static 的作用？（静态变量什么时候初始化）

- 不考虑类时：

    1. 修饰全局变量作用是隐藏：修饰全局变量和函数时隐藏它们的全局可见性，限制它们只能在该文件所在的编译模块中使用；
    2. 修饰局部变量的作用是改变局部变量的生存周期：静态变量在函数内定义，始终存在，只进行一次初始化。作用区域与局部变量相同，生存周期等同于程序的生存周期；
    3. 默认初始值：static 修饰的变量没有初始化的话存储在全局未初始化区域，存储在该区域的变量统一默认为 0；

- 考虑类时： 4. 修饰类成员变量：该变量只与类关联，不与类的对象关联。定义是要分配空间，不能在类声明中初始化，必须在类定义外部初始化，初始化时不必标识为 static ； 可以被非 static 成员函数任意访问； 5. 修饰成员函数：该函数没有 this 指针，无法访问类的非 static 成员函数和非 static 成员变量；不能被声明为 const、虚函数、volatile；可以被非 static 成员函数任意访问；

---

strlen 和 sizeof 的区别？

1. sizeof 是运算符，在编译时就获得了结果；strlen 是字符处理的库函数；
2. sizeof 的参数可以是任何数据的类型或任何数据（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；
3. 因为 sizeof 值在编译时就已经得到了，所以它不能用来得到动态分配存储空间的大小；

---

extern"C" 的用法？

- 功能：extern "C" 的作用是为了实现 C/C++ 混编；
- 原因：C++ 为了支持重载功能，编译器在编译阶段会对函数名进行“再次重命名”-由原有的函数名和各个参数的数据类型构成一个新的函数名；但是 C 的编译器不支持重载，也就没有了这步额外的步骤；这个结果也就意味着在 C 和 C++ 进行混合编译时，会造成编译器在程序链接阶段无法找到函数具体的实现，导致链接失败。
- 解决方法：而 extern "C" 就是为了解决这个问题的，extern "C" 修饰的变量和函数都是按照 C 语言方式进行编译和链接的；

宏定义（define）和 typedef 的区别?

- 宏主要是用于定义常量及书写复杂的内容；typedef 主要是定义类型别名；
- 宏替换发生在预处理阶段，属于文本插入替换；typedef 是编译的一部分；
- 宏不检查类型； typedef 会检查数据类型；
- 宏不是语句，不在结尾处加分号； typedef 是语句，要加分号标识结束；
- 注意对指针的操作；typedef char *p_char* 和 *define p_char char* 差别巨大

宏定义（define）和函数的区别?

- 宏定义在预编译期间完成替换，之后被替换的文本参与编译，相当于直接插入了代码，不存在调用步骤，执行速度更快；函数调用在运行时需要跳转到具体调用函数；
- 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；
- 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；
- 宏定义在最后不需要加分号；

宏定义（define）和 const 的区别?

- 编译阶段
  - define 在编译的预处理阶段起作用，而const 在编译、运行时起作用
- 安全性
  - 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含住所有内容，不然很容易出现错误；
  - const 常量有数据类型，编译器可以对其进行类型安全检查
- 内存占用
  - 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；const 在程序运行时只有一份备份，且可以执行常量折叠，能将复杂的表达式计算出结果放到常量表中；
  - 宏定义的数据没有分配内存，只是插入替换；const 定义的变量只是值不能改变，但要分配内存空间；

宏定义（define）和内联函数的区别?

- define 是关键字，inline 是函数
- 宏定义在预处理阶段进行文本替换，inline 函数在编译阶段进行替换
- inline 函数有类型检查，相比于宏定义更安全

变量声明和定义的区别？（声明和定义的区别？）

- 声明仅仅是把变量声明的位置提供给编译器，并不分配内存空间；定义是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extern），但只能在一处定义；

不同的指针类型？（数组名和指针， a 和 &a 之间的区别）

- 数组名和指针
    - 二者都可以通过增减偏移量来访问数组中的元素；
    - 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的操作；
    - 当数组名当作形参传递给调用函数后，就退化成了一般指针了，也就有了自增自减的操作了。但sizeof运算符不能再得到原数组的大小了；

野指针和悬空指针?

- 都是指向无效（无效是指"不安全不可控"的）内存区域的指针，访问行为将会导致为定义行为;
- 野指针指的是没有初始化的指针；悬空指针是指向的区域已经被释放了的指针;
- 解决方案：
    - 野指针：定义指针变量要及时初始化，要么置空；
    - 悬空指针：指针释放操作后要及时置空；

初始化和赋值的区别?

- 初始化的发生过程在编译阶段，赋值发生在函数或者程序运行时；
- 对于基本数据类型来说，两者的差别不大，但是对于用户自定义数据类型区别很大：
    - 对于初始化操作，程序调用了类的构造函数；
    - 对于赋值操作，程序需要重载 = 号，调用拷贝构造函数

什么是内存池，如何实现？（待完善）

- 一次性向操作系统申请一大堆内存，在此之上构建需要的对象，用完之后统一返还给操作系统。这样做最大的好处是避免了频繁的 new/delete 开销和带来的内存碎片问题。

什么是内存泄漏？如何检查和避免？（内存泄露后果，如何检测，解决方案？）（待完善）

- 内存泄漏一般指堆内存泄漏。用动态存储分配函数动态开辟的空间，在使用完毕后未释放，结果导致一直占据该内存单元。直到程序结束；
- 避免内存泄漏的几种方法：
    1. 计数法：使用一个变量来记录内存申请和释放的次数，在使用内存分配函数时加一，在释放内存时减一，在程序执行完检测该变量的值，如果不为0则表示内存泄漏；
    2. 一定要把基类的虚构函数声明为虚函数；
    3. 对象数组的释放要用到 delete[]；
    4. 有 new 就要有 delete， 有 malloc 就要有 free

--

C++ 函数调用压栈过程?

- 当函数从入口函数 main 开始执行时，编译器会将我们操作系统状态的运行状态，main 函数的返回地址，main 的参数，main 函数中的变量 依次进行压栈；

- 当 main 函数开始调用函数 func 时，编译器此时会将 main 函数的运行状态进行压栈，再将 func 函数的返回地址、func 函数的参数从右向左、func 函数定义变量依次压栈；
- 当 func 函数调用 func2 函数时，编译器会将调用 func 函数的运行状态压栈，再将 func2 函数的返回地址、func2 函数的参数从右向左依次压栈；

总结： 函数压栈的顺序是当前程序运行的运行状态、调用函数的返回地址、调用函数的参数从右向左、调用函数中定义变量依次压栈。

--

C++ 将临时变量作为返回值时的处理过程?

- 函数调用结束后，函数返回值被临时存储到寄存器中，此时该值就和堆栈没有关系了。当函数退出时，临时变量可能被销毁，但被保存到寄存器的返回值和临时变量的生命周期已经没有关系了。

补充： C 语言中规定，16bit程序中，返回值保存在 ax 寄存器中，32bit 程序中，返回值保持在 eax 寄存器中，如果是 64bit 返回值，edx 寄存器保存高 32bit，eax 保存低 32 bit

---

全局变量和局部变量有什么区别？（全局变量和 static 变量区别？）

- 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，甚至是局部循环体内部存在，退出时即被销毁；
- 使用方式不同:通过声明后全局变量在程序各个部分都可以用到；局部变量分配在堆栈上，只能局部使用；
- 内存分配位置不同：全局变量分配在全局数据段并且在程序运行时被加载，局部变量分配在堆栈中，使用时才会被分配内存空间。

全局变量和 static 变量的区别

+ 全局变量的作用域是整个整个源程序，静态全局变量的作用域仅限本文件，静态局部变量的作用域仅限局部函数中；
+ 当整个源程序是由一个文件组成的话，静态全局变量和全局变量没有根本差别

函数指针是什么?

- 概念：函数指针指向的是特殊的数据类型，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，而函数名称不是其类型的一部分；
- 函数指针的声明方法：
    - int (*pf)(const int&, const int&); 其中pf就是一个函数，指向所有返回值为 int 型，并带有两个 const int& 参数的函数
- 函数指针的作用：函数和数据项很相似，函数也有指针，我们希望在同一个函数通过使用相同的参数在不同的时间产生不同的效果；
- 函数指针的使用：一个函数名就是一个指针，它指向函数的代码。一个函数的地址就是该函数的进入点，也是调用函数的地址。函数调用可以通过函数名，也可以通过指向函数的指针来调用。函数指针还允许将函数作为变元传递给其他函数；

回调函数的作用?

- 作用就是为了分离代码，降低程序的耦合性

数组和指针的区别?

- 数组在内存中是连续存放的，开辟一块连续的内存空间；数组所占存储空间是 sizeof(数组名)；数组大小：sizeof(数组名)/sizeof(数组元素数据类型)；
- 用运算符 sizeof 可以计算出数组的容量(字节数)。对于指针，使用 sizeof 得到的一个指针变量的字节数，不是指针所指的内存容量；
- 编译器为了简化对数组的支持，实际上是利用指针实现了对指针的支持，具体来说，就是将表达式的数组元素引用转换成为指针加偏移量的引用；
- 在向函数传递参数时，如果实参是一个数组，那么用于接收的形参为对应的指针，也就是传递过去的是数组的首地址而不是整个数组，能够提高效率；
- 在使用下标的时候，两则的用法相同，都是原地址加上下标值，不过数组的原地址就是数组首元素的地址是固定的，指针的原地址就不是固定的。

函数调用过程中栈的变化，返回值和参数变量哪个先入栈?

- 先定义的变量先入栈，先是函数形参入栈，然后是按照临时变量定义的顺序入栈，返回值和参数变量入栈的顺序是根据定义顺序，谁先被定义谁先入栈；

---

printf 函数实现原理？（待完善）

lambda函数的全部知识？（待完善）

将字符串 “hello world” 从开始到打印屏幕上的全过程？()

cout 和 printf 有什么区别?

- cout 是 C++ 标准库的控制台输入流类，需要配合 << 使用，而 printf 是 C 的控制台输出函数；
- printf 是变参函数，没有类型检查，不安全；cout 是通过运算符重载实现的，较为安全；
- printf 只能输入基本数据类型，cout 可以通过重载函数的方式去实现自定义类的输出；

---

ifdef、endif 代表着什么?

- ifdef、endif 实现了条件编译的需求；
- 使用 #define、#ifndef、#ifdef、#endif 避免头文件的重定义

---

strcpy 和 memcpy 的区别?

- 复制内容不同。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；
- 复制的方法不同。strcpy 不需要指定长度，它遇到'\0'时自动结束，容易产生溢出；而 memcpy 根据第三个参数决定复制的长度；
- 用途不同。复制字符串时一般使用 strcpy，复制其他类型时使用 memcpy；

程序在执行 int main(int argc, char *argv[]) 时的内存结构?

- 参数的含义是程序在命令行下运行时，需要输入 argc 个参数，每个参数都是 char 类型输入，依次存 argv[[] 在数组中，所有参数在指针 char* 指向的内存中，数组中元素个数是 argc 个，第一个参数为程序的名称

const char* 和 string 之间是什么关系?

- string 是 C++ 标准库，封装了对字符串的操作，const char* 是一个指向字符常量的指针。
- 其中 char* 、const char* 和 string 可以相互转化

strcpy、sprintf、memcpy 这三个函数的不同之处?

- 操作对象不同：stcpy的两个操作对象是字符串；sprintf 操作源可以是多种数据类型，目的操作对象是字符串；memcpy的两个对象

Debug 和 release 的区别?

- Debug(调试版本)，包含调试信息，容量要比 release 大，没有进行优化，生成文件除了 .exe 或者 .dll 外还有一个 .pdb，该文件记录了代码中断点等调试信息；
- Release（发布版本）：编译时对应用程序进行优化，使代码在运行速度和代码大小上都是最优的。不生成 .pdb 文件
- 实际上，Debug 和 Release 没有本质的区别，它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。实际上我们也可以对这些选项进行修改，从而得到优化后的调试版本或者带有跟踪语句的发布版本；

main 函数的返回值有什么考究之处的?

- 程序运行过程入口点 main 函数，即 main() 函数返回值类型返回值必须是 int，这样返回值才能传递给激活者(如操作系统)表示程序正常退出；
- main(int args, char **argv) 参数的传递一般使用 getopt() 函数去处理；

strcpy 函数和 strncpy 函数的区别？哪个更安全?

- 函数原型：

```
char *strcpy(char* strDest, const char* strSrc)
char *strncpy(char* strDest, const char* strSrc, int pos)
```

- strcpy 函数：如果 dest 所指的内存空间不够大，可能会造成缓冲溢出（buffer overflow）的错误；
  strncpy函数： 用来复制源自符的前 n 个字符，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

- 如果 sizeof(dest) > pos > sizeof(strSrc),则将 strSrc 全部拷贝到目标中，尾后自动加上 '\0'；如果 pos < sizeof(strSrc) ，则将源字符串按指定查昂都拷贝到目标字符串中，不包括 '\0'；如果 pos > sizeof(dest)，运行报错；

一致性哈希?

- 一致性哈希是一种哈希算法，就是在移除或者增加一个结点时，能够尽可能小的改变已存在 key 的映射关系；一致性哈希将整个哈希值空间组织成一个虚拟的圆环，一致性哈希的基本思想就是使用相同的hash算法将数据和节点都映射到一个圆形哈希空间中，按照顺时针的方向，将数据绑定到离它最近的一个节点上去；

浅拷贝和深拷贝的区别?

- 浅拷贝只是拷贝一个指针，并没有开辟一个新地址，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；
- 深拷贝不仅是拷贝值，还会开辟一块新的空间去存放新的值，即使原先的对象被释放掉，释放内存也不会影响深拷贝的值。

动态编译和静态编译?

- 静态编译：编译器在编译可执行文件时，把需要用到的对应动态链接库中的部分提取出来，连接到可执行文件中，使可执行文件在运行时不需要依赖动态链接库；
- 动态编译：动态编译的可执行文件需要附带一个动态链接库，在执行时需要调用对应动态链接库的命令。所以优点是一方面缩小了可执行文件的体积，另一方面加快了编译速度，节省了系统资源。缺点是无论程序大小，都需要附带一个相对庞大的链接库，二是移植性较差，如果其他计算机没有安装对应的运行库，那么动态编译的可执行文件就无法运行；

关键字

---

malloc 和 free 实现的原理?

- malloc/free 函数在底层是由 brk、mmap、munmap 这些系统调用实现；
- malloc 是从堆中申请内存。操作系统中有一个记录空闲内存地址的链表，操作系统收到程序的申请时，会遍历该链表，然后找到第一个空间大于所申请空间的堆节点，然后把该节点从链表中删除，并把该节点的空间分配给程序；

---

malloc、realloc、calloc 区别?

- calloc 会对申请的空间初始化为0， 但是其他两个不会；
- malloc 申请的空间必须用 memset 初始化；
- realloc 是对已有的内存空间进行调整，可能会重新开辟内存空间并且释放原有内存空间。

---

new/delete 和 malloc/free 的区别？（new 和 malloc 的区别？）

- 类型不同：new/delete 是关键字，需要编译器支持；malloc/free 是库函数，需要头文件支持；
- 参数不同：new 申请内存分配时无需指定内存块大小，编译器会根据类型信息自行计算。而 malloc 则需要显示地指出所需内存的大小；
- 返回结果不同：new 内存申请成功后返回时对象类型的指针；malloc 返回的是 void*，需要通过强制类型转换将 void* 转换成需要的类型；
- 内存分配失败返回结果不同：new 内存返回失败会抛出异常，malloc 内存分配失败返回 NULL；
- 申请用户自定义类过程不同：new 会调用 operator new 函数，申请足够的内存，然后调用该类型的构造函数，进行初始化成员变量，最后返回自定义类型指针，delete 会先调用析构函数，然后再调用 operator delete 函数释放内存。malloc/free 只能实现动态的申请和释放内存，无法强制要求对其做自定义类型对象构造和析构函数。

---

new 和 delete 是如何实现的？（delete 是如何知道要释放内存的大小？）

- new 简单类型直接调用 operator new 分配内存；
- new 复杂结构时表达式会调用 operator new(operator new[]) 函数，分配一块足够大的、原始的、未命名的内存空间；
- 编译器运行相应的构造函数构造这些对象，并为其传入初始值；
- 对象被分配了空间并沟站完成，返回一个指向该对象的指针；
- delete 简单数据类型时只是调用了 free 函数，且此时 delete 和 delete[] 等同；
- delete 复杂类型时先调用析构函数，再调用 operator delete；

---

malloc 申请的空间能用 delete 释放吗?

- 理论上来说是可以的，但是由于 malloc/free 操作对象都是明确大小的，而且是不能用在动态类上，与此同时 new 和 delete 会自动进行类型检查和大小，malloc/free 不能执行构造函数和析构函数，所以不建议使用 delete 去释放 malloc 分配的空间，因为这样不能保证每个 C++ 程序运行时都正常；

---

delete 和 delete [] 的区别？（delete、delete []、alloctor 有什么作用？）

- delete 和 delete[] 的区别主要是在于对非内部数据对象的处理上，delete 只能调用一次非内部数据对象的析构函数，而 delete[] 会调用数组的每一个成员的析构函数；
- new 的机制是将内存分配和对象构造组合在一起，delete 也是将对象的析构函数和内存释放组合在一起了，allocator 是将这两部分分开了，allocator 申请一部分内存，但是不进行初始化，只有当需要的时候才进行初始化操作；

C++ 有几种类型的 new?

- palin new:最常用的 new 函数，申请空间失败抛出异常
- nothrow new：申请失败返回 NULL

- placement new：在一块已经开辟的空间上构造对象，需要显示的调用析沟函数去销毁，不能使用 delete，否则可能会造成内存泄漏或者运行错误

C++ 中 NULL 和 nullptr 的区别?

- NULL 是宏定义，而 nullptr 是关键字；
- C 语言中的 NULL 被定义为 (void*)0, C++ 中 NULL 被定义为 0；所以在传入 NULL 参数时会把 NULL 当作整数零来看待，为了和调用参数时指针函数区分开，引入 nullptr
- nullptr 可以明确的区分整型和指针类型，根据环境自动的转换成相应的指针类型，但不会被转换成任何整形，所以不会造成参数传递错误；

---

final 和 override 关键字?

- override 指定了子类的这个虚函数是重写父类的，如果该函数父类没有或者没有被声明为虚函数，是要被报错的。
- final 阻止某个类被继承或者某个虚函数被重写；

---

auto、decltype 和 decltype(auto) 的用法?

- auto： C++ 新标准引入 auto 说明符，用它就能让编译器替我们去分析表达式所属类型。auto 让编译器通过初始值进行类型推演，从而获得定义变量的类型，auto 必须要有初始值
- decltype：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，并不去计算实际的表达式的值。
- decltype(auto):C++14 新增的类型指示符，可以用来生命变量以及指示函数返回类型。在使用时会将"="号左边的表达式替换调 auto，再根据 decltype 的语法规则来确定类型；

C++ 四种强制转换 reinterpret_cast/const_cast/static_cast/dynamic_cast 用法?

- reinterpret_cast：用于类型之间进行强制转换；
- const_cast：用来修改类型的 const 和 volatile 属性；
- static_cast：派生类指针或引用转换成基类；
- dynamic_cast：基类向派生类转换比较安全，反之不安全，但是转换失败会返回 nullptr，以此来做判断

---

static_cast 比 C 语言的转换强在哪里?

- 更加安全
- 更加明显，能够一眼能看出什么类型转换成什么类型，容易找出程序中的错误；可以清楚辨别出每个显示的强制转换；可读性要好；

C++ 左值引用、右值引用?

- 左值：表示可以获取地址的变量，它能出现在赋值语句的左边，对该表达式进行赋值，如果修饰符 const 修饰左值时，可以取地址但是不能赋值；

- 右值：表示无法获取地址的对象，有常量值、函数返回值、lambda 表达式等，无法获取地址不代表其值不能被改变，当定义了右值的右值引用时就可以更改右值。
- 左值引用：传统 C++ 中引用被称为左值引用；
- 右值引用：右值引用关联到右值，右值引用指向特定位置，也就是说，右值虽然无法获取地址，但是右值引用可以获取地址，该地址表示临时对象的存储对象；
    - 通过右值引用的声明，右值的生命周期与右值引用类型变量的声明一样长，只有该变量还存在，该右值临时量就会一直存在下去；
    - 右值引用独立于左值和右值，意思就是右值引用类型的变量可能是左值也可能是右值；
    - T&& t在发生自动类型推断时，它是左值还是右值取决于它的初始化；

---

volatile 关键词作用是什么?

- volatile 关键字是一种修饰符，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、硬件或者线程等。遇到这个关键字声明的变量，编译器对访问变量的代码就不再进行优化，从而可以提供特殊地址的稳定访问。
- 声明语法：int volatile vInt;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，而且读取的数据立刻保存。
- volatile 用在如下地方：
    - 中断服务程序中修改的供其它程序检测的变量需要加 volatile；
    - 多任务环境下各任务间共享的标志应加 volatile；
    - 存储器映射的硬件寄存器通常也要加 volatile，因为每次对它读写都可能有不同意义

---

mutable 关键字的作用?

- 为了突破 const 限制设置的。可以用来修饰一个类的成员变量。被 mutable 修饰的变量，将永远处于可变的状态。即使 const 函数也可以改变这个变量的值；

重载、虚函数

C++ 中重载、重写（覆盖）、和隐藏的区别?

- 重载（overload）：在同一范围定义中，存在参数类型、数量和顺序不同，但是函数名相同的两个或者以上函数；他们的关系称之为重载；
- 重写（override）：派生类中的函数覆盖基类的同名函数，要求基类函数和必须是虚函数，且必须与基类的虚函数有相同的参数个数，参数类型和返回值类型；
- 隐藏（hide）：派生类的函数会隐藏其父类同名函数（只需要函数名相同）；

什么是虚函数？虚函数和纯虚函数的区别?

- 虚函数是 C++ 用于实现多态的机制定义的函数，实现通过基类指针访问派生类的需求；
- 虚函数和纯虚函数的区别：
    - 纯虚函数没有具体函数体，并且是用 “=0” 来表示
    - 含有纯虚函数的类不能实例化，并且继承的子类也必须实现继承的纯虚函数后才能实例化
    - 纯虚函数在虚函数表中的值为 0

静态函数能定义为虚函数吗？常函数呢？（常函数待补充）

- 不能，static 成员不属于任何类对象或类实例，即使给此函数加上 virtual 也是没有意义的；
- 静态和非静态成员函数的一个主要区别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，因为它是类的一个成员，并且 vptr 指向保存虚函数地址的 vtable。对于静态成员来说它没有 this 指针，所以就无法访问 vptr。
- 这就是为什么静态函数不能为虚函数的原因，虚函数的调用关系是 this-> vptr -> vtable -> virtual function

---

哪些函数不能定义为虚函数?

1. 友元函数，它不是类的成员函数；
2. 全局函数；
3. 静态成员函数（没有this指针）；
4. 构造函数、拷贝构造函数；

虚函数的代价?

- 带有虚函数的类，每个类都会产生一个虚函数表，用来存储指向虚函数成员的指针；
- 带有虚函数的类的每一个对象，都会有一个指向类虚表的指针，会增加对象的空间大小；
- 不能再是内联函数，因为内联函数在编译阶段会被替代，而虚函数表示等待，在运行阶段才能确定到底采用哪个函数，虚函数不能是内联函数；

基类的虚函数表存储在内存的什么位置（哪个区）？虚表指针 vptr 的初始化位置?

- 基类的虚函数表存储在内存的只读数据段，也就是内存模型中常量区；
- vptr 虚表指针是在类在进行实例化时，构造函数的执行时进行初始化，存在对象的前四个字节；

静态类型和动态类型、静态绑定和动态绑定的介绍

- 静态类型：对象在声明时采用的类型，在编译期既已确定；
- 动态类型：通常是指一个指针或引用目前所指对象的类型，是在运行期才决定的；
- 静态绑定：绑定的是静态类型，所对应的函数或属性依赖于静态类型，发生在编译期；
- 动态绑定：绑定的动态类型，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；

总结：非虚函数一般都是静态绑定，而虚函数都是动态绑定；

引用是否能实现动态绑定，为什么可以实现?

- 可以，引用在创建的时候必须要初始化，在访问虚函数时，编译器会根据所绑定的对象类型决定要调用哪个函数。主要只能调用虚函数。

重载运算符?

1. 重载运算符只能重载已有的运算符；对于一个重载的运算符，其优先级和结合律与内置的一致才可以；不可以改变运算符操作个数；
2. 重载方式为两种：成员运算符和非成员运算符，成员运算符比非成员运算符少一个参数；下标运算符、箭头运算符必须是成员运算符；
3. 引入运算符重载是为了实现类的多态性；
4. 当重载的运算符是成员函数时，this 绑定在左侧运算符对象。成员运算符函数的参数数量比运算符对象的数量少一个；至少含有一个类类型的参数；
5. 从参数的个数推断到底定义的是哪种运算符，当运算符既是一元运算符又是二元运算符(+,-,*,&);
6. 下标运算符必须是成员函数，下标运算符通常以访问元素的引用作为返回值，同时最好定义下标运算符的常量版本和非常量版本；
7. 箭头运算符必须是类的成员，解引用通常也是类的成员；重载的箭头运算符必须返回类的指针；

函数中有重载时，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

什么是隐式转换，如何消除隐式转换?

1. 隐式转化意思是不需要用户干预，编译器私下进行的类型转换行为。
2. 基础数据类型的隐式转换发生在小精度到大精度的转换中；比如 char=>int,int->long；自定义对象中子类对象可以隐式转换为父类对象；
3. C++ 提供了explicit 关键字，在构造函数声明的时候加上可以禁止隐式转换；
4. 如果构造函数只接受一个参数，则它实际上定义了转换为此类类型的隐式转换机制。可以通过将构造函数声明为 explicit 加以制止专类转换，explicit 关键字只对一个参数的构造函数有效，需要多个实参的构造函数不能用于执行隐式转换，所以无需为这些构造函数指定 explicit

---

类、对象、继承

介绍面对对象的三大特性，并举例说明?

- 封装"把对象的服务和属性结合成一个独立的服务单元，并尽可能的隐藏内部的实现细节；
- 继承：特殊类拥有一般类的属性和服务。通过继承可以利用已有的数据结构去构造新的数据结构；
- 多态：指的是同一操作作用于不同对象产生的不同响应。简单的概括为：一个接口多种实现；

是什么是类的继承?

- 类与类之间的关系：
    - 包含关系：一个类是另一个类的成员；
    - 使用关系：一个类使用另一个类的成员函数，实现方式是使用友元函数或者传递函数指针；
    - 继承关系：关系具有传递性；

- 概念：一个类包含另一个类的属性和方法，继承的被称为子类或者派生类，被继承的被称为父类或者基类；
- 特点：子类拥有父类的所有属性和方法，子类也可以定义的属性和方法，子类对象可以当作父类对象使用；
- 访问控制：public、private、protected

空类大小是多少?

- C++ 空类大小不为0，不同编译器设置不一样，vs设置为 1；
- C++ 标准指出，不允许一个对象（包括类对象）的大小为 0，不同的对象不能具有相同的地址；
- 带有虚函数的类大小不为 1，因为会有一个 vptr 指向虚函数表，具体大小根据指针大小确定；
- C++ 中要求对于类的每个实例都要有唯一的地址，那么编译器自动为空类分配一个字节大小，这样就可以保证每个实例都有一个独一无二的内存地址；

什么是成员列表初始化？什么时候必须要用初始化成员列表？调用过程：

- 概念：在类的构造函数中，不在函数体内对成员变量进行赋值，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；
- 效率：用初始化列表会快一些的原因是，对于类型它少一次构造函数的调用过程，而在函数体内赋值则会多一次调用，对于内置数据类型没有差别；
- 必须使用：
    - 初始化一个引用成员变量时；
    - 初始化一个 const 成员变量时；
    - 调用一个基类的构造函数，而基类构造函数拥有一组参数时；
    - 当调用一个成员类的构造函数，而它的构造函数有一组参数时；
- 调用过程：编译器会一一操作初始化列表，以适当的声明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前。成员列表的初始化顺序是按照类中成员声明顺序决定的，不是初始化列表中排列顺序决定的；

C++ 有哪几种构造函数?

- 默认构造函数
- 初始化构造函数
- 拷贝构造函数
- 移动复制构造函数
- 委托构造函数
- 转换构造函数
- 默认构造函数和初始化构造函数在定义类的对象，完成对象的初始化工作；
- 复制构造函数用于复制文本类的对象；
- 转换构造函数用于将其他类型的变量，隐式转换为本类对象；

什么时候调用拷贝构造函数?

- 用一个类的实例化区初始化另外一个对象的时候;
- 函数的参数时类的对象时（非引用传递）
- 函数的返回值是函数体内局部对象的类的对象时，此时虽然返回了 NVR（Named return Value）优化，但是由于返回方式是值传递，所以会在返回值的地方发生拷贝构造函数--- linux g++ 中不会发生;

移动构造函数?

- 使用一个对象初始化另一个对象后，如果该对象不再使用后，重复使用该对象的地址空间，避免新的空间重复分配，降低构造的构造成本，这是移动构造函数的设计初衷;
- 拷贝构造函数中对于指针需要采用深拷贝，但是移动构造函数中，对于指针采用的是浅拷贝。然后在析构时对于第一个指针置为null，而不回收空间即可;
- 移动构造函数的参数和拷贝构造函数的不同在于:
    - 拷贝构造函数的参数是一个左值引用，但是移动构造函数的初值是一个右值引用。
    - 这就意味着移动构造函数的参数是一个右值或者是一个将亡值。只有用一个右值，或者将亡值初始化另一个对象时，才会调用移动构造函数。

构造函数的几种关键字?

- default:显示的要求编译器生成合成构造函数，防止在调用时相关构造函数类型没有定义而报错;
- delete:删除构造函数、赋值运算符函数等;
- =0： 定义纯虚函数;

类成员初始化方式？构造函数的执行顺序？为什么成员初始化类列表会快一些?

- 赋值初始化：通过在函数体内进行赋值初始化；初始化列表：在冒号后使用初始化列表进行初始化;

- 两者区别在于：函数体内进行初始化是在所有的数据成员被分配内存后才进行的，列表初始化是给数据成员分配内存空间时就进行初始化，也就是说分配一个数据成员，只要冒号后面有此数据成员的赋值表达式（此表达式必须是括号赋值表达式），那么分配了内存空间后在进入函数体之前给数据成员赋值，也就是说初始化这个数据成员时函数体还未执行。

- 一个派生类的构造函数执行顺序如下:

    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（多个普通基类也是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（按照初始化顺序）；
    4. 派生类自己的构造函数。

- 赋值初始化在构造函数中作赋值的操作，而成员初始化列表时纯粹的初始化操作。C++ 赋值操作是会产生临时对象的，临时对象的出现会降低程序的效率。

拷贝初始化和直接初始化的区别?

- 当用于类类型对象时，初始化的拷贝形式和直接形式有所不同：
  - 直接初始化直接调用与实参匹配的构造函数，拷贝拷贝初始化总是调用拷贝构造函数。
  - 拷贝初始化首先使用制定构造函数创建一个临时对象，然后用拷贝构造函数将那个临时对象拷贝到正在创建的对象。
- 为了提高效率，允许编译器跳过创建临时对象这一步，直接调用构造函数构造要创建的对象，这样就完全等价于直接初始化了，但是要区别两种情况：
  - 当拷贝构造函数为 private 时，拷贝初始化会出错；
  - 使用 explicit 修饰构造函数时，如果构造函数存在隐式转换，编译时会报错；

类的对象存储空间？

- 非静态成员的数据类型大小之和；
- 编译器加入的额外成员变量（如指向虚函数表的指针）
- 为了边缘对齐优化加入的 padding。
- 空类（无静态数据成员）的对象 size 为 1，当作为基类时，size 为 0；

类对象大小受到哪些因素影响？

1. 类的非静态成员变量大小，静态成员不占据类的空间，成员函数也不占据类的空间大小；
2. 内存对齐另外分配的空间大小，类中的数据也是需要进行内存对齐操作的；
3. 虚函数的话，会在类对象插入 vptr 指针，加上指针大小；
4. 当该类是某类的派生类，那么派生类继承的基类部分数据成员也会存在在派生类中的空间中，对派生类的类对象大小进行扩展；

类什么时候会析构？

- 对象声明周期结束，被销毁时；
- delete 指向对象的指针时，或者 delete 指向对象的基类类型指针，而其基类析构函数是虚函数时；
- 类包含类时，外部类析沟函数被调用时，内部类的析构函数也会被调用；

析构函数的作用？以及是如何起作用的？

- 构造函数值是起初始化作用，但是实例化一个对象的时候，可以通过实例去传递参数，从主函数传递到其他函数里面，这样就使其他的函数里有值了；
- 析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务的处理，可以是释放对象分配的内存。
  - 特点是和构造函数同名，但在函数前加 ~ ；
  - 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，当撤销对象时，编译器也会自动调用析构函数；
  - 每一个类中必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成的默认的析构函数，一般析构函数定义为类的公有属性；

为什么析构函数一般写成虚函数？

- 为了避免内存泄漏，当基类指针指向派生类对象时，如果删除该基类指针，就会调用指针指向的派生类析构函数，而派生类的析构函数又会去自动调用基类的析构函数，这样整个派生类的对象就完全被释放了；
- 如果析构函数不声明为虚函数，则编译器实施静态绑定，这样整个派生类的对象完全被释放。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，这样就会造成派生类对象析构不完全，造成内存泄漏。
- 在实现多态时，当用基类操作派生类时，在析构时防止只析构基类而不析构派生类的状况发生，所以要将基类的析构函数声明为虚函数。

构造函数能否声明为虚函数或者纯虚函数，析构函数呢？（构造函数为什么不能是虚函数，析构函数为什么要是虚函数）

- 析构函数可以声明为虚函数，并且在一般情况下都要定义为虚函数。
- 只有在基类析构函数定义为虚函数时，调用操作符 delete 销毁指向对象的基类指针，才能准确调用派生类的析构函数，才能准确销毁数据；
- 析构函数可以是纯虚函数，含有纯虚函数的类是抽象类，此时不能被实例化，但派生类中可以根据自身需求重新改写基类的纯虚函数；
- 构造函数不能定义为虚函数，在构造函数中可以调用虚函数，不过此时调用的是真正构造类中的虚函数，而不是子类的虚函数，因为此时子类尚未构造好。

构造函数一般不定义为虚函数的原因？

- 创一个对象时，需要确定对象的类型，而虚函数的是在运行时动态确认其类型的。在构造一个对象时，由于一个对象还没有创建成功，编译器无法知道对象的实际类型；
- 虚函数的调用需要虚函数表指针 vptr，而该指针存放在对象的内存空间中，若构造函数声明为虚函数，那么由于对象还未创建，没有内存空间，更没有虚函数表 vtable 地址来调用虚构造函数；
- 虚函数的作用在于通过父类的指针或者引用它的时候能够变成调用子类的成员函数，而构造函数是在创建对象时自动调用，不可能通过父类或者引用去调用，因此规定构造函数不能是虚函数。

构造函数和析构函数可以调用虚函数吗？为什么？（构造函数或者虚构函数中可以调用虚函数吗？）

- 在 C++ 中，提倡不在构造函数和析构函数中调用虚函数；
- 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身定义类型定义的版本；
- 因为父类对象会在子类之前进行构造，此时子类部分的数据还未初始化，此时调用子类的虚函数是不安全的，故而 C++ 不会进行动态联编；
- 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，在调用子类的析构函数，所以在调用基类的析构函数时，派生类对象数据成员已经被销毁了，这个时候再调用子类的虚函数已经没有意义了

构造函数和析构函数的执行顺序，以及它们的内部都做了什么？

- 构造函数顺序

1. 基类构造函数。如果有多个基类，则构造函数调用顺序是某个类在类派生表中的顺序，而不是它们在成员初始化表中的顺序。
2. 成员类对象构造函数，如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。
3. 派生类的构造

- 析构函数顺序
  1. 调用派生类的析构函数；
  2. 调用成员类对象的析构函数；
  3. 调用基类的析构函数；

虚析构函数的作用？父类的析构函数是否要设置为纯虚函数？

- 父类的析构函数声明为虚函数是为了防止内存泄漏。
- 当派生类中申请了空间，如果没有把基类的析构函数声明为虚函数，删除指向派生类的基类指针时，就只会调用基类的析构函数，而不回去调用派生类的析构函数，导致内存泄漏的问题；
- 纯虚析构函数一定得定义，因此每个派生类的析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层的析构函数，因此缺乏任何一个基类析构函数的定义，都会导致链接失败，所以最好不要把析构函数定义为纯虚析构函数。

构造函数和析构函数可否抛出异常？

- C++ 只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当，在构造函数发生异常，控制权又转出构造函数之外，因此当对象的构造函数发生异常，对象的析构函数不会被调用，因此会发生内存泄漏；
- 用 auto_pre 对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危险，不再需要在析构函数中手动释放资源；
- 如果控制权基于异常的因素里离开析构函数，而此时正有另一个异常正处于作用状态，C++ 会调用 terminate 函数让程序结束；
- 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是它没有完成它应该执行的每一件事。

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，但是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，属于重载 "=" 号运算符的范畴，"="两端的对象都是已存在的

拷贝构造函数和赋值运算符重载的区别？

- 拷贝构造函数是函数，赋值运算符是运算符重载；
- 拷贝构造函数是直接构造一个新的对象，所以在初始化对象前不需要检查源对象和新建对象是否相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有内存分配则需要把内存是放掉；

- 形参传递是调用拷贝构造函数 （调用的被赋值对象的拷贝构造函数），并不是所有出现 "=" 的地方都是使用赋值运算符的。如果等号左边的对象不存在，使用的就是拷贝构造函数；

注意：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

C++ 多态是如何实现的?

1. 编译器在发生基类中有虚函数时，会自动为每个含有虚函数的类生成一个虚表，该表时一个一维数组，虚表中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区初始化虚指针 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. 所谓的合适时机就是在派生类定义对象时，程序会自动调用构造函数，在构造函数中创建虚表并对虚表初始化。在构造子类对象时，会先调用父类的构造函数，此时编译器只能找到父类，并为父类对象初始化虚表指针，令它指向父类的虚表，当调用子类的构造函数时，为子类对象初始化虚表指针，令它指向子类的虚表；
4. 当派生类没有对基类的虚函数重写时，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到后面；
5. 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的情况进行动态的调用，从而实现多态性。

构造函数、析构函数、虚函数是否能声明为内联函数?

- 语法上没有错误，因为 inline 只是一个建议，编译器不一定会真正内联；
- 析构函数和构造函数声明为内联函数是没有意义的：
    - class 中的函数都是默认内联的，编译器也是选择性的 inline；
    - 编译器在构造函数和析构函数中添加额外的操作（申请/释放内存，构造/析构对象），导致构造函数和析构函数并不是看上去那么内联；
- 虚函数的 inline 是看情况的：
    - 如果是指向派生类的指针调用被声明为 inline 的虚函数不会被内联展开，但是对象本身调用不复杂的虚函数时，会内联展开；

---

public/protected/private 区别和访问、继承权限？（C++ 类成员的访问权限和继承权限问题？）

- public 的变量和函数在类的内部外部都可以访问；
- protected 的变量和函数只能在类的内部和其派生类中访问
- private 修饰的元素只能在类的内部访问
- public：基类成员在派生类中的访问权限保持不变；
- protected：基类的公有成员在派生类的访问权限变为保护权限，其他不变；
- private：基类的所有成员在派生类中都变成私有权限

什么是 trivial destructor?

- 默认的，由系统自动生成的析构函数；

什么是对象复用？什么是零拷贝？

- 对象复用：本质是设计模式中的享元模式，通过将对象存储到“对象池”中是先对对象的重复利用，避免多次重复创建对象的开销，节约系统资源；
- 零拷贝：一种避免 CPU 将数据从一块存储拷贝到另一块存储的技术，可以减少数据拷贝和共享总线操作的次数；

关于 this 指针?

- 概念：
  - this 指针是类的指针，指向对象的首地址；
  - this 指针只能在成员函数中使用，在全局函数、静态成员函数中无法使用；
  - this 指针只有在成员函数中定义才有意义，且存储位置因编译器不同而不同；
- 作用：
  - 指向当前所在对象，通过它可以访问到当前对象的所有成员；
- 使用： this 是一个常指针，使用方式等同于常指针；
- 特点：
  - this 只能在成员函数中使用，全局函数、静态函数都不能使用 this。实际上成员函数默认第一个参数为 T* const this;
  - this 在成员函数开始前构造，在成员函数结束后清楚，生命周期等同于函数的参数。当调用一个类的成员函数时，编译器将类的指针作为函数 this 参数传递进去；
  - 编译器会对 this 作优化，因此 this 指针传递效率较高；

- this 指针何时创建的?
  - this 指针在成员函数开始执行前构造，在成员函数执行结束后清除；
- this 指针存放在哪里？堆、栈、全局变量、还是其他?
  - this 存放位置会因为编译器的不同而不同，栈、寄存器、全局变量都有可能。在汇编级别里，一个值只会以三种形式存在，立即数，寄存器值，内存变量值。不是放在寄存器中就是放在内存中。
- this 指针是如何传递类中的函数的？绑定还是函数参数的首参数就是 this 指针？那么 this 指针是如何找到“类实例后函数的”?
  - 大多编译器是通过 ecx（计数寄存器）传递this指针；
  - 在调用之前，编译器会把对应的对象地址放到 eax 中，this 是通过函数参数的首参来传递的。
- this 指针是如何访问类中的变量的?
  - this 指针实际上就是对象的首地址，通过偏移量来访问类中的变量的；
- 只有获得一个对象后，才能通过对象使用 this 指针。如果我们知道一个对象 this 指针的位置，可以直接使用吗?
  - this 指针只有在成员函数中有定义，获得一个对象无法知道该对象的this指针的位置，所以就算知道this指针的位置，也没法使用。
- 每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调试函数?
  - 不会，只有虚函数才会创建函数表。通过 this 指针，用来指向不同的对象，从而确保不同对象之间调用相同的函数可以互不干扰；

在成员函数中 delete this 会出现什么问题？对象还可以使用吗?

- 在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码所有内容，类的成员函数单独放在代码段。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再调用它。当调用 delete this 时，类对象内存空间被释放。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；
- 为什么是不可预期的问题？(待补充)
- 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 本质是"未将被释放的内存调用一个或多个析构函数，然后释放内存"，于是 delete this 回去调用析构函数，析构函数中又去调用 delete this，形成无限递归，造成堆栈溢出，系统崩溃；

this 指针调用成员变量时，堆栈会发生变化吗?

+ 当在类的非静态成员函数中访问类的非静态成员，编译器会自动将对象的地址作为隐含参数传递给函数，这个隐含参数就是 `this` 指针。
+ 对于类中个成员的访问都是通过 `this` 指针去调用的。
+ 当 `this` 指针调用成员变量时，该变量会入栈（C++的main函数也是一个函数调用，在堆栈中）。

什么是虚拟继承?

+ 由于 `C++` 支持多继承，除了公有继承、保护继承、私有继承三种方式以外，还支持虚拟继承；
+ 在虚拟继承的情况下，无论基类被继承多少次，都是只存在一个实体，虚拟继承基类的子类中，子类会在增加某种形式的指针，或者指向虚基类子对象，或者指向一个相关的表格；表格中存放的不是虚基类子对象的地址，就是其偏移量，此类指针被成为 `bptr`。如果既继承 `vptr`,又继承 `bptr`，某些编译器会将其优化成一个指针；

类是如何实现只能静态分配和动态分配的?

- 静态分配：把 new/delete 运算符重载设为 private 属性。
- 动态分配：把构造函数、析构函数设为 protected 权限，再用在类来动态创建；

如果想将某个类作为一个基类，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

什么情况会自动生成默认构造函数?

1. 带有默认构造函数的类成员函数，如果一个类没有任何构造函数，但它含有一个成员对象，而后者有默认构造函数，那么编译器就为该为该类合成一个默认构造函数。该合成函数只有在构造函数真正被需要的时候才会发生；如果一个类中有多个成员类对象的话，那么该类的每一个构造函数必须调用每一个成员对象的默认构造函数并且必须按照类对象在该类中的声明顺序进行；
2. 带有默认构造函数的基类，如果一个没有构造函数的派生类派生自一个带有默认构造函数基类，那么该派生类会合成一个古朴造函数调用上一层基类的默认构造函数；
3. 带有一个虚函数的类；
4. 带有一个虚基类的类；
5. 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。

抽象基类为什么不能创建对象？

- 在面向对象程序设计中，很多情况下基类本身生成对象是不合理的，并且为了实现多态，需要在基类中定义虚函数。为了解决这个联动问题，就引入了纯虚函数；
- 将函数定义为纯虚函数后，该类就被成为抽象类，在该类中就无法创建对象了。
- 所以说不是抽象基类不能创建对象，而是我们定义一个不能创建对象的类，取名为抽象基类；

继承机制中对象直接如何转换？指针和引用之间如何转换？

- 向上类型转换：将派生类指针或引用转换为基类的指针或引用称为向上类型转换，向上类型转换会自动进行，并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乡下类型转换，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上动态类型时别技术。
- RTTI 技术，用 dynamic_cast 进行向下类型转换；

C++ 的组合和继承有什么区别和优缺点？

- 继承：
  - 优点：子类可以重写父类的方法来方便地实现对父类函数进行扩展；
  - 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，所以无法在运行期改变从父类继承的方法的行为；
    3. 如果对父类的方法做了修改的话，则子类必须作出相应的修改，所以说子类和父类是一种高耦合，违背了面向对象的思想；
- 组合：设计类时把组合的类加入到该类中作为自己的成员变量
  - 优点：
    1. 当前对象只能通过所包含的那个对象区调用其方法，所以所包含的对象的内部细节对当前对象是不可见的；
    2. 当前对象与包含的对象是一个低耦合关系，如果修改所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，可以通过 set 方法给所包含的对象赋值；
  - 缺点：

1. 容易产生过多的对象
2. 为了组合多个对象，必须仔细对借口进行定义

静态成员和普通成员是什么?

- 生命周期：静态成员变量从类加载开始到类被卸载，一直存在；普通成员只有在类创建对象后才开始存在，对象结束生命周期结束；
- 共享方式：静态成员变量是全类共享；普通成员变量是每个对象单独享有；
- 定义位置：普通成员定义在栈或堆中，而静态成员定义在静态全局区；
- 初始化位置：普通成员变量在类中初始化，静态成员在类外初始化；
- 默认实参：可以使用静态成员变量作为默认实参；

虚继承?

- 为了解决多继承时命名冲突和冗余数据问题，产生了虚继承，使得派生类中只保留一份间接基类的成员；
- 在继承方式前面加上 virtual 关键字就是虚继承；
- 虚继承的目的就是为了让某个类做出声明，承诺愿意共享它的基类，其中这个被共享的基类就是虚基类。

虚函数内存结构，棱形继承的虚函数内存结构呢?

- 虚函数内存结构（当基类有虚函数时）：
    1. 每个类都有虚函数和虚表；
    2. 如果不是虚继承，那么子类将父类的虚指针继承下来，并指向自身的虚表（发生在对象构造时期），有多少个虚函数。虚表里面的项就有多少，多重继承时，可能存在多个基类虚表和虚指针；
    3. 如果是虚继承，那么子类会有两份虚指针，一份是指向自己的虚表，另一份指向虚基表，多重继承时虚继承与虚基表指针有且只有一份；

多继承的优缺点，作为一个开发者如何看待多继承?

- C++ 运行一个派生类指定多个基类，这样的继承结构被称为多重继承；
- 多重继承的优点很明显，就是对象可以调用多个基类中的接口；
- 如果派生类所继承的多个基类有相同的基类，而派生类的对象需要调用这个祖先类的接口方法，就会容易出现二义性；
- 加上全局符确定调用哪一份拷贝；
- 使用虚拟继承，使得多重继承类可以只拥有虚基类的一份拷贝；

如果有一个空类，它会默认添加什么函数?

- 默认构造函数；
- 默认拷贝构造函数；
- 默认析构函数；

- 重载赋值运算符函数；

拷贝构造函数为什么必须引用而不能传值?

1. 拷贝构造函数的作用就是用来复制对象的，在使用这个的对象的实例来初始化这个对象的一个新的实例；
2. 参数过程：将地址传递和值传递统一起来，归根结底传递的是"值"（地址也是值，只不过通过它可以找到另一个值）
    - 值传递：对于内置数据类型的传递，直接复制拷贝到形参（注意形参是函数内部局部变量）；
    - 对于类类型的传递，首先要调用拷贝构造函数来初始化形参（局部变量）；
    - 引用传递：无论是内置类型还是类类型，传递引用或指针最终都是传递的地址值，而地址总是指针类型（属于简单类型），显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用（对于类类型） **注意：当拷贝构造函数采用值传递时，会调用类的拷贝构造函数来初始化形参，会产生无限递归调用，造成内存溢出）**

如何设计一个类来计算子类的个数?

1. 为类设计一个 static 静态变量 count 作为计数器；
2. 类定义结束后初始化 count；
3. 在构造函数中对 count + 1；
4. 设计拷贝构造函数，在进行拷贝构造函数中进行 count + 1；
5. 设计复制构造函数，在进行复制函数中对 count + 1；
6. 在析构函数中对 count - 1；

什么时候合成构造函数?

1. 如果一个类没有任何构造函数，但它含有一个有默认构造函数的成员对象，编译器会为该类合成一个默认构造函数，因为不合成一个默认构造函数那么该成员对象的构造函数不能调用；
2. 没有任何构造函数的类派生自一个带有默认构造函数的基类，那么需要给该派生类合成一个构造函数，只有这样的基类的构造函数才能被调用；
3. 带有虚函数的类，虚函数的引入需要进入虚表，指向虚表的指针，该指针在构造函数中初始化，所以没有构造函数的话该指针无法被初始化；
4. 带有一个虚基类的类； **注意：**
5. 并不是任何没有构造函数的类都会合成一个构造函数；
6. 编译器合成出来的构造函数并不会显示设定类内的每一个成员变量；

什么时候需要合成拷贝构造函数?

- 有三种情况会以一个对象的内容作为另一个对象的初始：
    1. 当一个对象用等号做显示的初始化操作；
    2. 当对象被当作参数交给某个函数时；
    3. 当函数传回一个类对象时；
- 如果一个类没有拷贝构造函数，但是有一个含有拷贝构造函数的类类型成员变量，那么此时编译器需要为该类合成一个拷贝构造函数；

- 如果一个类没有拷贝构造函数，但是该类继承含有拷贝构造函数的基类，那么编译器会为该类合成一个拷贝构造函数；
- 如果一个类没有拷贝构造函数，但是该类声明或继承了虚函数，此时编译器会为该类合成了一个拷贝构造函数；
- 如果一个类没有拷贝构造函数，但是该类含有虚基类，此时编译器会为该类合成一个拷贝构造函数；

成员初始化列表会在什么时候用到？它的调用过程是什么？

1. 当初始化一个引用成员变量时；
2. 初始化一个 const 成员变量时；
3. 当调用一个基类的构造函数，而构造函数拥有一组参数时；
4. 当调用一个成员类的构造函数，而它拥有一组参数时；
5. 调用过程：编译器会一一操作初始化列表，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前，列表中项目顺序是由类中的成员声明顺序决定的，而不是初始化列表中的排列顺序决定的；

构造函数的执行顺序是什么？

1. 在派生类构造函数中，所有的虚基类以及上一层基类的构造函数调用；
2. 对象 vptr 被初始化；
3. 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 vptr 被设定以后才做；
4. 执行程序提供的代码；

一个类中的全部构造函数的扩展过程是什么？

1. 记录在成员初始化列表中的数据成员初始化操作会被放在构造函数的函数体内，并与成员的声明顺序为顺序；
2. 如果一个成员并没有出现在成员初始化列表中，但它有一个默认构造函数，那么这个默认构造函数必须被调用；
3. 如果类中有虚表，那么它必须被设定初值；
4. 所有上一层的基类构造函数必须被调用；
5. 所有虚基类的构造函数必须被调用；

哪些函数不能是虚函数？

1. 构造函数：构造函数初始化对象，派生类必须知道基类函数做了什么，才能进行构造；当有虚函数时，每个类有一个虚表，每个对象有一个虚表指针，虚表指针在构造函数中初始化；
2. 内联函数：内联函数表示在编译阶段进行函数体的替换，而虚函数意味着在运行期间进行类型确认，所以内联函数不能是虚函数；
3. 静态函数，静态函数不属于对象，属于类，静态成员函数没有 this 指针，因此静态函数被设置为虚函数是没有意义的；

4. 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来说没有虚函数的说法；
5. 普通函数：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一说；

--

如何阻止一个类被实例化的方法?

1. 把类定义为抽象类
2. 将构造函数声明为 private；

如何禁止程序自动生成拷贝构造函数?

- 为了阻止编译器默认生成拷贝构造函数和拷贝复制函数，我们需要手动去重写这两个函数，某些情况下，为了避免调用拷贝构造函数和拷贝赋值函数，我们需要把他们设置为 private，防止被调用；
- 类的成员函数和友元函数还是可以调用 private 函数，如果这个 private 只声明不定义，则会产生一个链接错误；
- 针对上述两种情况，我们可以定义一个 base 类，在 base 类中将拷贝构造函数和拷贝赋值函数设置成 private，那么派生类中编译器就不会自动生成这两个函数，且由于 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；

成员函数中 memset(this, 0, sizeof(this)) 会发生什么?

1. 有时候类里面定义了很多 C 语言类型的变量，我们希望在构造函数中将他们初始化为 0， 可以使用 memset(this, 0 ,sizeof(this))，将整个对象的内存全部置为 0，但是在以下几种情况下不能这么使用：
    - 类中含有虚函数表：这样做会破坏虚函数表，后续对虚函数表调用会出现问题；
    - 类中含有 C++ 对象成员：如果对象成员的构造函数中分配了内存，这样做会破坏对象的内存；

为什么友元函数必须在内部声明?

- 因为编译器需要在编译阶段知道哪些数据或者函数能够访问类的私有成员；

独立特性（范式编程，异常机制、元编程、STL标准库)

迭代器失效的情况?

- 迭代器失效就是原本可以访问到容器内迭代器的元素在插入或删除的操作后无法再继续访问到了
- 插入元素：
    - 尾后插入：size < capacity 时，尾迭代器失效（未重新分配空间）； size == capacity 时，所有迭代器均失效（需要重新分配空间）
    - 中间插入：size < capacity 时，插入元素之后的所有迭代器失效； size == capacity 时，所有迭代均失效；
- 删除元素：

- 尾后删除：只有尾迭代器失效；
- 中间删除：删除位置之后所有迭代器失效；
- list 链表删除节点后只有当前迭代器失效，erase 返回下一个有效迭代器；
- map/set 等关联容器底层是红黑树，删除节点不会影响其他节点的迭代器，使用递增方式获取到下一个迭代器(mmp.erase(iter++));

C++ 异常处理方法？（C++ 是如何处理多个异常的）

- 程序执行时由于程序员疏忽或者系统资源紧张等因素都有可能导致异常，常见的异常有 数组下标越界，除法计算时除数为 0，动态分配空间不足；如果不及时对这些异常进行处理，程序多数情况下会崩溃；
- try、throw、catch 关键字；
- 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，指出所能抛出异常列表；
- C++ 标准异常类 exception；

C/C++ 类型安全?

- C 的类型安全：C 只在局部上文表现出类型安全，比如两个不同结构体指针相互转换时，编译器会报错，除非使用显示类型转换；
- C++ 类型安全：C++ 提供一些新的机制来保证类型安全：
  - 操作符 new 返回的指针类型严格与对象匹配，而不是 void*；
  - 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；
  - 引入 const 关键代替 #define constans，它是有类型，有作用域的，而 #define constants 只是简单的文本替换；
  - 一些 #define 宏可被改写成 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，当然改写成模板也能保证类型安全；
  - C++ 提供 dynamic_cast 关键字，使得转换过程更加安全，因为 dynamic_cast 比 static_cast 涉及更多的类型检查；

C++ 模板是什么，底层是如何实现的?

- 编译器并不是把函数模板处理成能够处理任意类的函数；编译器从函数模板通过具体类型产生不同的函数；编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译；
- 这是因为模板函数要被实例化以后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件只有声明，没有定义，则编译器无法实例化该模板，最终导致链接错误；

模板函数和模板类的用法（模板类和模板函数的区别）？（待修改）

- 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示地指定，即函数模板允许隐式调用和显示调用，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；

智能指针的作用

- 普通指针容易造成堆内存泄漏，（忘记释放）二次释放，程序发生异常时内存泄漏等问题；使用智能指针对堆内存的管理会更加严谨；

智能指针的原理、常用的智能指针及实现?

- 原理：智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏，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；
- shared_ptr：
- unique_ptr:
- weak_ptr:
- auto_ptr:

---

auto_ptr 作用（待修改）？

1. auto_ptr 的出现，主要是为了解决"有抛出异常时发生的内存泄漏"问题；抛出异常，将导致指针所指向的位置得不到释放而导致内存泄漏；
2. auto_ptr 构造时取得某个对象的控制权，在析构时释放该对象。实际上是创建一个 auto_ptr 类型的局部变量，在局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏；
3. auto_ptr 的构造函数是 explicit ，阻止了一般指针隐式转换为 auto_ptr 的构造，所以不能直接将一般指针赋值给 auto_ptr 类型的对象，必须用 auto_ptr 的构造对象创建对象；
4. 由于 auto_ptr 对象析构时会删除它所用的指针，所以使用时避免多个 auto_ptr 对象管理同一个指针；
5. auto_ptr 内部实现，析构函数删除对象用的是 delete 而不是 delete[]，所以 auto_ptr 不能管理数组；
6. auto_ptr 支持所拥有的指针类型之间的隐式转换；
7. 可以通过 * 和 -> 运算符对 auto_ptr 所使用的指针进行提领操作；
8. T* get():获得 auto_ptr 所拥有的指针； T* release()：释放 auto_ptr 的所有权，并将所使用的指针返回；

智能指针的循环引用?

- 循环引用指的是使用多个智能指针 shared_ptr 时，出现了指针之间的相互指向，从而形成环的情况，类似于死锁现象，在这种情况下智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏；

智能指针出现循环引用怎么解决?

- 弱指针专门用来解决 shared_ptr 循环引用的问题，weak_ptr 不会修改引用计数，即其存在与否不影响对象的引用计数器；
- 循环引用就是两个对象互相使用一个 shared_ptr 成员变量指向对方。弱引用并不对对象的内存进行管理，在功能上类似于普通指针；
- 弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存；

智能指针管理内存资源，RAII是什么?

- RAII 全程为 "Resource Acqusition is Initialization", 直译过来就是 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；

- C++ 语言机制把保证了当一个对象创建的时候，自动调用构造函数，当对象超出了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；
- 智能指针(std::shared_ptr 和 std::unique_ptr)即 RAII 最具代表的实现，使用智能指针，可以实现自动的动态管理。

手写实现智能指针类?

- 智能指针是一个数据类型，一般使用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动基类 smartPointer<T*>对象的引用计数，一旦T类型对象的引用计数为0，就释放该对象；
- 除了指针对象以外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为1，需要一个构造函数，新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存；
- 通过复写赋值运算符，才能将一个旧的智能指针赋值给另外一个指针，同时旧的引用计数减1，新的引用加1；
- 一个构造函数、拷贝构造函数、复制构造函数、析构哈桑纳湖、移走函数；

C++ 11 有哪些新特性?

- nullptr 代替 NULL；
- 引用了 auto/decltype 两个关键字实现了类型推导；
- 基于范围的 for 循环 for(auto &i : res){};
- 类和结构体中的初始化列表；
- lambda 表达式（匿名函数）；
- std:forward_list 单向链表；
- 右值引用和 move 语义；

为什么模板类一般都放在一个 .h 文件中?

- 模板定义特殊：由 template<...> 处理的任何东西都意味着编译器在当时不为它分配空间，它一致处于等待状态，直到被一个模板实例告知。在编译器和链接器的某一处，有一机制能去掉指定模板的多重定义；所以为了容易使用，几乎总是在头文件中防止全部的模板声明和定义；
- 在分离式编译的环境下，编译器编译某一个 .cpp 文件时，并不知道另一个 .cpp 文件的存在，也不会去查找，（当遇到未决符号时，它会寄希望于链接器）。这种模式在没有模板的情况下运行良好，但遇到模板时就无法起作用了，因为模板仅在需要的时候才会实例化；
- 所以当编译器只看到模板声明时，它不能实例化该模板，只能创建一个具有外部链接的符号并期待链接器能够将符号的地址决议出来；
- 然而当实现该模板的 .cpp 文件中没有用到模板的实例时，编译器不回去是咯话，所以整个工程的 .obj 文件中就找不到一行模板实例的二进制代码，于是链接器也没用了。

模板和实现可不可以不写一个文件中？为什么？（待补充）

迭代器 ++it、it++ 哪个好，为什么?

- 前置返回一个引用，后置返回一个对象；
- 前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低；

C++ 中标准库是什么?

- C++ 标准库分为两部分：
  - 标准函数库：这个库是通用的、独立的、不属于任何类的函数组成，函数库继承自 C 语言；
  - 面向对象库类：这个库是类及其相关函数的集合；
- 输入/输入 IO、字符串和字符处理、数学、时间、日期和本地化、动态分配、其他、宽字符函数
- 标准的 C++ IO 类，String 类、数值类、STL 容器类、STL函数对象、STL迭代器、STL分配器、本地化库、异常处理类、杂项支持类；

写一个比较大小的模板?

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

template<typename type1, typename type2>
type1 Max(type1 a, type2 b){
    return a > b ? a : b;
}

int main(){
    cout << "Max = " << Max(5.5, 'a') << endl;
    return 0;
}
```

STL 的 hashtable 实现

- hashtable 使用的是开链法解决冲突的；
- hashtable 中使用的是自定义的 hashtable_node 数据结构组成的表；
- 内置了28个质数作为初始时的长度；

STL 的 traits 技法? （待补充）

STL 的两级空间配置器? （一级配置器，二级配置器，一级分配器，二级分配器）（待完善）

- 为了避免频繁的在堆上开辟释放内存，造成外部碎片浪费内存空间，设置了二级空间配置器；
- 当开辟空间 <= 128bytes 时，则视为开辟小块内存，则调用二级空间配置器；

---

vector 和 list 的区别与应用，怎么找到某 vector 和 list 的倒数第二个元素?

- vector 数据结构：
  - vector 拥有一块连续的内存空间，并且起始位置不变，因此可以高效的进行随即存取，时间复杂度为O（1）；
  - 插入和删除时会造成内存块的拷贝，时间复杂度为O（n）；
  - 数组内存空间不够时，会重新申请一块空间进行内存拷贝；

    - vector 可以实现动态增长，支持对数组高效率的访问和在数组尾部删除和插入操作，在中间和头部删除和插入不易；
    - 和数组最大的区别在于不需要使用者去考虑容量问题，包括在扩容的时候；
- list 数据结构:
    - list 是由双向链表实现的，内存不连续，只能通过指针访问数据，所以随即存储没有效率，时间复杂度为 O(n);
    - 链表支持高效的插入和删除；
    - list 是一个双链表结构，支持对链表的双向遍历。每个节点包括三个信息：元素本身，指向前一个元素的节点和指向后一个元素的节点。可以高效的对元素任意位置进行访问和插入删除等操作；
    - 由于涉及到对额外指针的维护，开销较大；
- 区别：
    - vector随即访问效率高，但删除和插入时（不包括尾部）需要挪动数据，不易操作。list访问需要遍历整个链表，随即访问效率低，但是对数据的插入和删除操作比较方便；
    - list 是单向的，vector 是双向的，vector迭代器在使用后就失效了，而 list 的迭代器在使用后还可以继续使用；
- 访问倒数第二个元素：
    - vector: `int mySize = vec.size(); vec.at(mySize-2);`
    - list: 双指针，相距两个位置，遍历一边走完，慢指针的位置就是到处第二个元素的位置；

STL 中 vector 删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间?

- size() 函数返回已用空间大小；capacity() 返回宗空间大小；capacity()-size()则是剩余的可用空间大小。但size()和capacity()相等时，说明 vector 目前的空间已被用完，如果再添元素的话，则会引起 vector 空间的动态增长；
- 由于动态增长会引起重新分配内存空间、拷贝原空间、释放原空间，这些过程会降低程序效率。因此可以使用 reserve(n) 预先分配一块较大的指定大小的内存空间，这样当指定大小的内存空间未使用完时，是不会重新分配内存空间的，这样可以提升效率，只有当 n > capacity 时，调用 reserve(n) 才会改变 vector 的容量；
- resize() 成员函数只改变了元素的数目，不改变 vector 的容量；

总结:

1. 空的 vector 对象，size() 和 capacity() 都为 0；
2. 当空间大小不足时，新分配的空间大小是原空间大小的两倍；
3. 使用 reserve() 预分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率；
4. 当 reserve() 分配的空间比原空间小时，是不会引起重新分配的；
5. resize() 函数只改变容器的元素数目，未改变容器大小；
6. 用 reserve(size_type) 只是扩大了 capacity 的值，这些内存空间还可能是'野'的，如果此时使用'[]'来访问，则可能越界，而 resize(size_type new_size) 会真正使容器具有 new_size 个对象；

- 两倍扩容是因为每次扩展的新尺寸必然刚好大于之前分配的总和，也就是说之前分配的内存空间不可能被使用，这样对内存不友好，最好把增长因子设定在（1，2）；对比可以发现采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定指定大小的容量只能达到 O(n) 的时间复杂度，因此使用成倍的扩容方式；

STL 容器内部删除一个元素?

1. 顺序容器：erase迭代器不仅使所指向的被删除的迭代器失效，而且使被删除元素之后的所有迭代器失效（list除外），所以不能使用 erase(it++)的方式，但是 erase 的返回值是下一个有效迭代器; `it = c.erase(it);`
2. 关联容器：erase迭代器只是被删除元素的待迭起失效，但返回值是 void,所以要采用 erase(it++) 的方式删除迭代器；

STL 迭代器如何实现?

- 迭代器是一种抽象的设计理念，通过迭代器可以在不了解容器内部原来的情况下，遍历该容器，除此之外 STL 中的迭代器一个最重要的作用就是作为容器与 STL 算法的粘合剂；
- 迭代器的作用就是提供一个遍历容器内部所有元素的接口，因此迭代器内部必须保存一个与容器相关联的指针，然后重载各种运算操作来遍历，其中最重要的就是 * 和 -> 运算符，以及 ++ 、-- 等可能需要重载的运算符重载。
- 最常见的迭代器相应类型有五种：value type、difference type、pointer、reference、iterator catagoly；

---

map、set 如何实现的，红黑树是怎么能够同时使用这两种容器的？为什么使用红黑树?

- 他们的底层都是以红黑书的结构实现的，因此插入删除等操作都在 O(logn)时间内完成的，因此可以完成高效的插入删除操作；
- 定义一个模板参数，根据参数的不同决定它是哪种容器。
- map 和 set 需要自动排序，红黑树能实现且时间复杂度低；

如何在共享内存中使用 STL 标准库？（待补充）

map 插入有几种方式?

- 用 insert 函数插入 pair 数据

```
mapData.insert(pair<int,string>(1,"one"));
```

- 用 insert 函数插入 value_type 数据

```
mapData.insert(map<int,string>::value_type (1,"one"));
```

- 用 insert 函数插入 make_pair 数据

```
mapData.insert(make_pair(1,"one"));
```

- 用数组插入数据：

```
mapData[1] = "one";
```

STL 的 unordered_map(hash_map) 和 map 的区别？hash_map 如何解决冲突和扩容?

- 区别是 unordered_map 底层是用 hash 实现的， map 是用红黑树实现的，所以表现出现的不同是 unordered_map 存储元素是无序的，而 map 是有序的，但是因为排序是需要消耗时间的，所以 map 消耗的资源平均要大一些；
- hash_map 底层使用的是 hash_table ，而 hash_table 使用的开链法避免冲突的；
- 当向容器内部添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值，即当前数组的长度乘以加载因子的值的时候，就自动扩容了；

---

vector 越界访问下标，map 越界访问下标？ vector 删除元素会不会释放空间?

- 通过下标访问 vector 中的元素是不会做边界检查的，即使下标越界，程序也不会报错，而是返回这个地址存储的值；如果想要在访问 vector 数组时进行边界检查，可以使用 at 函数；
- map 的 下标运算符 [] 作用是：将 key 作为下标去执行查找，并返回相应的值，如果不存在这个 key，就将一个具有该 key 和 value 的节点插入到 map 中；
- erase() 函数，只能删除内容，不能改变容量大小
    - erase成员函数，它删除了itVect 迭代器指向的元素，并且返回要被删除的 itVect 之后的迭代器，迭代器相当于一个智能指针；
    - clear() 函数，只能清空内容，也不能改变容量的大小；

---

map 中的 [] 和 find 有什么区别?

- 下标运算符 []：将关键码作为下标去执行查找，并返回相应的值，如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项插入到 map 中；
- find 函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器；

STL 中的 list 和 queue 区别?

- list 不能够像 vector 一样以普通指针作为迭代器，因为其节点不保证在存储空间中的连续存在；
- list 插入操作和删除操作都不会造成原有 list 迭代器失效；
- list 不仅是一个双向链表，还是一个环状双向链表，所以它只需要一个指针；
- list 不会有容量扩充的操作，所以插入前的所有迭代器在插入操作后仍然有效；
- deque 是一种双向的连续线性空间，可以在头尾两端分别做元素的插入和删除操作，
- deque 和 vector 最大的差异在于 deque 可以在常数时间内对头尾做元素的插入和删除操作，二是 deque 是动态的以分段连续空间组合而成，不必有重新分配空间的设计；

STL 的 allocator，deallocator

- 第一级配置器直接使用 malloc()、free()、relloc()，第二级配置器视情况采用不同的策略：当配置区块超过 128bytes 时，视为足够大，调用第一级配置器；当配置器区块小于 128bytes 时，为了降低额外负担，使用复杂的内存池整理方式，而不再使用一级配置器；
- 第二级配置器主动将任何小额区块的内存需求量上调至 8 的倍数，并维护 16 个 free-list，各自管理大小为 8~128bytes 的小额区块；
- 空间配置函数 allocate()，首先判断区块大小，大于 128 直接调用第一级配置器，小于 128 就检查对应的 free-list。如果 free-list 之内有可用区块，就直接拿过来用，如果没有可用区块就将区块大小调整为 8 的倍数，然后调用 refill()，为 free-list 重新分配空间；
- 空间释放函数 deallocate()，该函数首先判断区块大小，大于 128byetes 时直接调用一级配置器，小于 128bytes 时就找到对应的 free-list 然后释放内存；

STL 的 hash_map 扩容发生了什么?

- hash table 表格内的元素被称为桶，而由桶所链接的元素被称为节点，其中存入桶元素的容器是vector容器。
- 向前操作：首先尝试从目前所指的节点出发，前进一个位置，由于节点被安置在 list 内，所以利用节点的 next 指针即可轻易的完成向前操作，如果目前正巧是 list 尾端，就调至下一个bucket身上，那正是指向下一个 list 的头部节点；

常见的容器性质总结?

1. vecotr：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，支持首尾快速增删，也支持随即访问；双向队列；
4. stack：底层一般使用 list 或 deque 实现，封闭头部即可；适配器，对容器的再封装
5. queue：底层一般使用 list 或者 deque 实现，等比头部即可；适配器，对容器的再封装
6. priority_queue：底层数据结构一般为 vector 容器，堆 heap 为处理规则实现底层容器管理；
7. set：底层结构为红黑树，有序，不重复；
8. multiset：底层结构为红黑树，有序，可重复；
9. map：底层结构为红黑树，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 hahs 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，不重复；

---

vector 的增加删除是怎么做到的，为什么是 1.5 或者是 2?

1. 新增元素：vector 通过一个连续的数组存放元素，如果集合已满，则在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的数据，再插入新数据；
2. 对 vector 的任何操作，一旦引起空间重新分配，指向原 vector 的所有迭代器都失效了；
3. 初始时刻 vector 的capacity 为 0，塞入第一个元素后 capacity 增加 1；
4. 不同的编译器实现扩容的方式不一样， vs2015 是以1.5倍扩容，GCC 是以 2 倍扩容；

对比可以发现采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到O(n)的时间复杂度，因此采用成倍的方式扩容 对于扩容的考虑：

1. 考虑可能产生的对空间的浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式就只有两种，2倍和1.5倍；
2. 以2倍的方式扩容，导致下一次申请的内存必然会大于之前分配的内存的总和，导致之前分配的内存不能再被使用，所以增长倍数因子最好再(1,2)；
3. 向量容器 vector 的成员函数 pop_back() 可以删除最后一个元素；
4. 而函数 erase() 可以删除由一个 iterator 指出的元素，也可以删除一个指定范围的元素；
5. 还可以采用通用算法 remove() 来删除 vector 容器中的元素；
6. 不同的是 remove 一般情况下不会改变容器的大小，而 pop_back 和 erase 等成员函数会改变容器的大小；

- 说一下 STL 每种容器对于的迭代器?
- STL 的 vector 实现
- STL 的 slist 实现
- STL 的 list 实现
- STL 的 deque 实现
- STL 的 stack 和 queue 实现
- STL 的 heap 实现
- STL 的 priority_queue 实现
- STL 的 set 实现
- STL 的 map 实现

set 和 map 的区别，multimap 和 multiset 的区别?

1. set 只提供数据接口，但是会将这个元素分配到 key 和 value，而且它的 compare_function 用的是 identity() 函数，这个函数输入什么输出什么，这样就实现了 set 的机制，set 的 key 和 value 是一样的，其实它保存了两份元素。
2. map 则提供了两种数据类型的接口，分别放在 key 和 value 的位置上，它的比较函数 function 采用的是红黑树的 compare_function()， 保存的确实时两份元素；
3. 它们两个的 insert 都是采用红黑树的 insert_unique() 独一无二的插入；
4. multimap 和 map 的唯一区别：multimap 调用的是红黑树的 insert_equal(),可以重复插入，而 map 调用的是独一无二的 insert_unique(), multiset 和 set 也一样，底层实现都是一样的，只是在插入的时候调用的方法不一样；

STL 的 unordered_map 和 map 的区别和应用场景?

- map 支持键值的自动排序，底层机制是红黑树，红黑树的查询和维护信息时间复杂度均为 O(logn)，但是空间占用比较大，因为每个节点都要保持父节点、子节点以及颜色的信息；

- unordered_map 底层机制是哈希表，通过 hash 函数计算元素的位置，其查询的时间复杂度为 O(1),维护时间与bucket桶所维护的list长度有关，但是建立 hash 表耗时较长；
- 从两者的底层机制和特点可以看出：map 适用于有序数据的应用场景，unordered_map 适用于高效查询的引用场景；
