500 道 C++ 面试（八股文）高频题实录

如了解就业干货，可扫码关注王道程序员公众号

如需以下领取就业资料可扫码添加

1. Java/C++/Python 面试题
2. 大厂校招面试题合集
3. 程序员简历模板
4. HR 面试注意事项
5. 2024 年值得去的互联网独角兽企业汇总

一、C/C++基础

1. 描述 C++ 程序的内存由哪几个部分组成，每个区域分别有什么作用和特点。（已完成）

一个 C++程序编译时内存分为 5 大存储区：栈区、堆区、全局区、文字常量区、程序代码区。

（1）栈区(stack)：由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

（2）堆区(heap)：一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收。注意它与数据结构 中的堆是两回事，分配方式倒是类似于链表。

（3）全局/静态区(static)：全局变量和静态变量的存储是放在一块的，在程序编译时分配。

（4）文字常量区：存放常量字符串。

（5）程序代码区：存放函数体（类的成员函数、全局函数）的二进制代码

2. 什么时候分配内存会产生内存碎片？（已完成）

内存碎片主要产生在以下几种情况下：

(1) 内存动态分配和释放：当程序频繁地申请和释放内存时，内存空间可能会被切割成多个小块，导致内存碎片。这是因为每次程序释放内存后，操作系统会将其标记为空闲，并尝试将其合并到一段连续的空闲内存中。但如果释放的内存块大小各异，且分散在内存的各个位置，就无法有效地合并，从而形成碎片。

(2) 内存分配算法：某些内存分配算法可能会导致碎片的产生。例如，为了避免频繁的内存分配和释放操作，操作系统可能会选择分配一块比实际需要更大的内存。虽然这种方式可以减少内存管理的开销，但也可能导致内存浪费和内存碎片。

(3) 多进程同时运行：在多进程环境中，每个进程都可能申请和释放内存。当一个进程释放内存时，其他进程可能会占用这块内存，从而导致内存空间的不连续，产生内存碎片。

此外，在使用如 malloc 等标准库函数进行内存分配时，如果频繁地分配和释放小块内存，也容易导致内存碎片问题。这是因为每次 malloc 调用都会在堆中分配一段大小合适的内存，而释放的内存块可能无法满足后续较大的内存分配需求，从而形成碎片。

因此，在编程过程中，应尽量避免频繁地申请和释放小块内存，优化内存分配策略，以减少内存碎片的产生。同时，可以使用一些工具和技术来检测和处理内存碎片，以提高内存的使用效率和程序的性能。

3. 负数的编码方式是什么？简述一下它的原理。（已完成）

负数的编码方式在计算机中主要是采用补码来表示的。补码编码方式不仅用于负数，也用于正数，它使得计算机中的加法运算能够同时处理正数和负数的加法，从而简化了计算机的电路设计。

补码的原理基于二进制数的特性以及计算机内部运算的需求。在计算机中，符号和数字一样，都必须用二进制数串来表示，因此正负号也必须用 0、1 来表示。通常，我们用最高的有效位来表示数的符号，0 表示正号，1 表示负号。

对于正数，其原码、反码和补码都是一样的。然而，对于负数，情况就有所不同了。负数的补码是通过取负数的绝对值的二进制表示（即原码），然后对其除符号位外的所有位取反（得到反码），最后再加 1 来得到的。这个加 1 的过程实际上是对反码进行加最低位的进位，如果最低位产生的进位需要向高位进位，则一直进到最高位为止。

采用补码表示负数的原理在于，这样可以使减法运算转化为加法运算。在计算机中，所有的减

法运算都可以转化为加法运算，这是因为减去一个数等于加上这个数的相反数（即负数）。而由于负数是用其正值的补码形式表示的，所以计算机在执行减法运算时，实际上是在执行加法运算。

总的来说，补码编码方式使得计算机能够统一处理加法和减法运算，简化了电路设计，提高了运算效率。

4. 浮点数的编码方式是什么？简述一下它的原理。(已完成)

浮点数的编码方式主要是遵循 IEEE 754 标准，这是一种在计算机中广泛使用的二进制浮点数算术标准。这种标准规定了浮点数的格式、精度以及指数与尾数的编码方式。

浮点数的原理是将一个实数表示为指数与尾数（或称为有效数字）的乘积形式，类似于科学计数法。具体来说，一个浮点数可以表示为：

数值 = (-1)^符号位 × (1 + 尾数) × 2^(指数-偏移量)

其中：

符号位用于表示数的正负，0 代表正数，1 代表负数。

指数用于表示小数点的位置，它决定了数值的范围。在 IEEE 754 标准中，指数部分采用了偏移量的表示方式，以避免指数为负数的特殊情况。偏移量是一个固定值，它等于指数所能表示的最大值的一半（对于单精度浮点数，偏移量是 127；对于双精度浮点数，偏移量是 1023）。

尾数是一个介于 1（包含）和 2（不包含）之间的二进制小数，它表示了数值的有效数字部分。在 IEEE 754 标准中，尾数部分包含了一个隐含的位（hidden bit），其值始终为 1，因此不需要显式存储。这样设计可以节省存储空间，并提高数值的精度。

通过调整指数和尾数的值，浮点数可以表示非常大或非常小的数值范围。这种表示方式既灵活又高效，使得浮点数成为计算机科学中一种非常重要的数值表示方法。

在实际的计算机系统中，浮点数的编码通常会将符号位、指数和尾数分别存储在不同的位段中。以单精度浮点数（float）为例，它在内存中占用 32 位，其中最高位是符号位，接下来 8 位用于存储指数，剩下的 23 位用于存储尾数。这种编码方式确保了浮点数的精度和范围能够满足大多数应用的需求。

需要注意的是，由于计算机内部存储空间的限制以及二进制表示的特性，浮点数的精度是有限

的。在进行浮点数运算时，可能会产生舍入误差或精度损失。因此，在使用浮点数时需要注意其精度限制，并结合具体业务场景进行考虑。

5. 可执行程序是如何生成的？（已完成）

可执行程序的生成过程通常包括四个主要步骤：预处理、编译、汇编和链接。下面将详细解释每个步骤的作用和目的。

(1) 预处理：这个步骤主要完成三个任务，即展开头文件、宏替换和去掉注释行。预处理器会读取源代码，并根据预处理器指令（如#include、#define 等）进行相应的处理。例如，#include 指令会告诉预处理器将指定的头文件内容插入到源代码中，而#define 指令则用于定义宏，预处理器会在编译前将所有的宏替换为其对应的值。

(2) 编译：编译器将预处理后的源代码转换为汇编代码。编译器会检查源代码的语法和语义错误，并将其转换为一种中间表示形式，即汇编代码。这个步骤中，编译器会进行诸如类型检查、变量作用域分析等操作，以确保源代码的正确性。

(3) 汇编：汇编器将编译生成的汇编代码转换为机器可以识别的二进制代码，也即是目标文件（.o 文件）。汇编器会将汇编指令转换为对应的机器码，并生成一个包含二进制代码的目标文件。

(4) 链接：链接器将多个目标文件以及所需的库文件链接成一个可执行文件。链接器会解析目标文件之间的符号引用，将各个目标文件中的代码和数据组合在一起，形成一个完整的可执行程序。此外，链接器还会处理静态链接和动态链接的问题，以确定程序在运行时如何加载和使用所需的库函数。

在整个过程中，还需要注意内存管理和优化的问题。例如，在链接阶段，静态链接器会复制被应用程序引用的目标模块，以减少可执行文件在磁盘和内存中的大小。同时，编译器和链接器也会进行各种优化操作，以提高生成的可执行程序的运行效率。

需要注意的是，不同的编程语言和编译器可能会有不同的实现方式和细节，但总体流程是相似的。此外，现代编译系统还可能包括其他步骤，如代码优化、调试信息生成等，以进一步提高生成的可执行程序的质量和性能。

6. 可执行程序是如何变成进程的？（已完成）

可执行程序转变为进程的过程涉及多个步骤和操作系统的参与。以下是这一过程的主要步骤：

(1) 加载到内存：当用户试图执行一个可执行程序时，操作系统首先会将该程序从磁盘加载到内存中。这是因为 CPU 只能直接访问内存中的数据，而不能直接访问磁盘中的数据。

(2) 分配资源：一旦程序被加载到内存，操作系统会为其分配必要的资源，如内存空间、CPU 时间片等。这些资源分配确保程序有足够的空间和环境来执行。

(3) 创建进程控制块（PCB）：操作系统为每个进程创建一个进程控制块（PCB），用于存储与进程相关的信息，如进程 ID、状态、寄存器信息、内存指针等。PCB 是操作系统管理进程的重要数据结构。

(4) 设置上下文：在进程开始执行之前，操作系统会设置进程的上下文，包括程序计数器、栈指针等寄存器的值。这些值决定了进程从哪里开始执行以及如何使用内存。

(5) 执行进程：一旦进程被设置好并放入调度队列中，操作系统会根据其调度算法选择适当的时机来执行该进程。当进程获得 CPU 时间片时，它开始执行，从程序入口点开始运行。

在这个过程中，操作系统扮演着关键的角色，它负责程序的加载、资源的分配、进程的管理和调度等任务。同时，进程的创建和管理也是操作系统实现并发执行和多任务处理的基础。

需要注意的是，不同的操作系统和平台可能会有不同的实现细节和步骤，但总体流程是相似的。此外，进程创建后，它会在操作系统的控制下与其他进程并发执行，共享系统资源，并通过进程间通信（IPC）机制进行交互。

7. 在 C 语言中如何调用 C++函数？

在 C 语言中直接调用 C++函数是不被允许的，因为 C++支持函数重载、类和对象等特性，而 C 语言并不支持这些。因此，C++编译器会对 C++代码进行“名字修饰”（name mangling），这会导致函数名在 C++编译器和 C 编译器中看起来不同。

然而，你可以通过一些方法间接地在 C 语言中调用 C++函数。一种常见的方法是使用“extern 'C'”链接规范。这个规范告诉 C++编译器，被修饰的函数应该以 C 语言的方式进行链接，因此不会进行名字修饰。

以下是一个简单的例子：

首先，你需要在 C++文件中创建一个函数，并使用 extern "C"来声明它：

```
// my_cpp_file.cpp
#include <iostream>

extern "C" {
    void my_cpp_function() {
        std::cout << "This is a C++ function called from C\n";
    }
}
```

然后，你可以在 C 文件中声明并调用这个函数：

```
// my_c_file.c
extern void my_cpp_function();  // 注意这里没有使用 extern "C"

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

在编译和链接时，你需要确保将 C++文件编译为对象文件，并将其与 C 文件的对象文件一起链接。例如，如果你使用 gcc，你可以这样做：

```
g++ -c my_cpp_file.cpp -o my_cpp_file.o
gcc -c my_c_file.c -o my_c_file.o
gcc my_cpp_file.o my_c_file.o -o my_program  -lstdc++
```

这里，-lstdc++选项用于链接 C++标准库，因为我们的 C++函数使用了 std::cout。如果你的 C++函数没有使用 C++标准库，那么可能不需要这个选项。

请注意，虽然这种方法可以在 C 语言中调用 C++函数，但它有一些限制。例如，你不能在 C 语言中调用 C++的类成员函数或模板函数，也不能直接处理 C++的复杂数据类型（如类、结构体等）。如果你需要在 C 和 C++之间进行更复杂的交互，可能需要考虑使用其他技术，如 C++/CLI 或 SWIG 等。

8. 请描述几种常见的 C/C++的缺陷和陷阱。

C/C++语言虽然强大且灵活，但在使用过程中也存在一些常见的缺陷和陷阱。以下是一些主要的例子：

(1) 内存管理问题：C/C++中的指针操作非常灵活，但也最容易出错。不正确的内存分配和释放可能导致内存泄漏、野指针等问题。例如，未初始化的指针在使用时可能指向任意内存地址，导致程序崩溃或数据损坏。此外，动态分配的内存如果忘记释放，会造成内存泄漏，长期运行可能导致系统资源耗尽。

(2) 类型安全问题：C 语言是一种弱类型语言，这意味着在类型转换时可能会出现精度丢失或错误。而 C++虽然提供了更严格的类型检查，但由于其支持操作符重载和模板等特性，如果使用不当也可能导致类型安全问题。

(3) 全局变量和静态变量的初始化顺序问题：在 C++中，不同源文件中的全局变量和静态变量的初始化顺序是不确定的。这可能导致依赖关系复杂的程序出现难以预测的行为。例如，如果一个全局变量在另一个全局变量之前初始化，而后者依赖于前者的值，那么程序的行为就可能是错误的。

(4) 未定义行为：C/C++标准中定义了一些未定义行为，即编译器对于这些行为的处理方式并不统一，可能因编译器或平台的不同而有所差异。例如，数组越界访问、空指针解引用等都是未定义行为，可能导致程序崩溃或产生不可预测的结果。

(5) 可移植性问题：虽然 C/C++在多种平台上都有广泛的应用，但由于不同平台的编译器、操作系统和硬件架构的差异，代码的可移植性可能受到影响。此外，为了优化性能或利用特定平台的特性，代码中可能包含一些与平台相关的代码，这也会降低代码的可移植性。

(6) 宏定义导致的问题：C/C++中的宏定义是通过预处理器在编译前进行文本替换的，如果宏定义不当或与其他代码产生冲突，可能导致难以调试的问题。例如，宏定义可能会改变代码的语义，或者由于宏的展开导致代码重复或逻辑错误。

为了避免这些缺陷和陷阱，开发者需要深入了解 C/C++语言的特性和规范，谨慎使用指针和动态内存分配，注意全局变量和静态变量的初始化顺序，避免未定义行为和平台相关代码，以及合理使用宏定义等特性。同时，使用合适的编程规范和工具进行代码审查和测试也是非常重要的。

9. 重写，重载，重定义这三者有什么区别？

重载（Overload）、覆盖（Override）和隐藏（Hiding）是 C++中常见的概念，它们在函数的行为和用途上有显著的区别。

1) 重载（Overload）:

重载发生在同一个类的作用域内。

允许有多个同名函数，但它们的参数列表（包括参数类型、个数或顺序）必须不同。

重载函数的返回类型可以相同也可以不同，但返回类型不是区分重载函数的依据。

重载是编译时多态性的体现，编译器会根据函数调用时提供的参数列表来确定调用哪个函数。

重载与函数的访问修饰符、返回值类型以及抛出的异常类型无关。

2) 覆盖（Override）:

覆盖发生在派生类与基类之间，是对基类中的虚函数进行重新定义。

派生类中的函数必须与基类中被覆盖的虚函数具有相同的函数名、参数列表和返回类型。

基类中的虚函数必须有 virtual 关键字。

覆盖是运行时多态性的体现，当通过基类指针或引用来调用虚函数时，会调用派生类中的覆盖版本。

覆盖用于实现多态性，允许子类在继承的基础上对父类的行为进行定制或扩展。

3) 隐藏（Hiding）:

隐藏也发生在派生类与基类之间，但基类中被隐藏的函数不一定是虚函数。

如果派生类定义了一个与基类同名的函数（不论参数列表是否相同），那么基类的函数将被隐藏。

即使基类的函数有 virtual 关键字，如果派生类中的函数参数列表与基类不同，也会发生隐藏而不是覆盖。

在隐藏的情况下，通过派生类对象只能访问到派生类中的函数，无法直接访问到被隐藏的基类函数。

隐藏可能导致意外的行为，因为程序员可能期望调用基类的函数，但实际上调用的是派生类的函数。

总结来说，重载、覆盖和隐藏的主要区别在于它们发生的场景、函数的行为以及多态性的体现方式。重载发生在同一个类内，通过参数列表的不同来区分函数；覆盖发生在派生类与基类之间，是对基类虚函数的重新定义，体现了运行时多态性；而隐藏可能导致基类函数被意外地屏蔽，需要谨慎使用。

10. 说一说 strcpy、sprintf 与 memcpy 这三个函数的不同之处

strcpy、sprintf 和 memcpy 是 C 语言中常用的字符串和内存操作函数，它们各自具有不同的用途和特性。下面是它们之间的主要不同之处：

1) strcpy => 函数原型

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

用途：复制字符串。

参数：

a. 源字符串（包含要复制的文本）。

b. 目标字符串（用于存储复制后的文本）。

行为：从源字符串开始，复制字符直到遇到空字符（'\0'），然后将空字符复制到目标字符串的末尾。这意味着 strcpy 会复制整个字符串，包括结尾的空字符。

注意事项：必须确保目标字符串有足够的空间来存储源字符串，否则可能会导致缓冲区溢出，这是一个常见的安全问题。strcpy 不会检查目标缓冲区的大小。

2) sprintf =》函数原型

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

用途：格式化字符串并将其存储在指定的缓冲区中。

参数：

目标字符串（缓冲区）。

格式字符串（包含要插入到目标字符串中的文本和格式说明符）。

可变数量的附加参数（用于替换格式说明符）。

行为：根据格式字符串和附加参数生成输出，并将该输出存储在目标字符串中。

注意事项：必须确保目标缓冲区有足够的空间来存储生成的字符串，否则可能导致缓冲区溢出。与 strcpy 类似，sprintf 也不会检查目标缓冲区的大小。为了避免缓冲区溢出，建议使用 snprintf，它允许指定目标缓冲区的大小。

3) memcpy=》函数原型

```
void *memcpy(void *dest, const void *src, size_t n);
```

用途：复制内存区域。

参数：

源内存地址。

目标内存地址。

要复制的字节数。

行为：从源地址开始，复制指定数量的字节到目标地址。它不关注所复制数据的内容或结构，只是简单地复制字节。

注意事项：必须确保目标内存区域有足够的空间来存储复制的字节，否则可能导致内存损坏。

源和目标内存区域可以重叠，而 memcpy 仍能正确复制数据。

总结：

(1) strcpy 专门用于复制字符串，直到遇到空字符。

(2) sprintf 用于格式化字符串并存储到缓冲区中，可以包含多种数据类型。

(3) memcpy 用于复制任意内存区域，不关心内容或结构，只复制指定数量的字节。

在使用这些函数时，都需要特别注意目标缓冲区的大小，以避免潜在的缓冲区溢出或内存损坏问题。对于 strcpy 和 sprintf，考虑使用更安全的替代函数（如 strncpy 和 snprintf）来减少这类风险。

11. strlen 和 sizeof 区别（已完成）

strlen 和 sizeof 在 C 语言中是两个不同的概念，它们在功能和用法上存在显著的差异。

首先，strlen 是一个函数，用于计算字符串的实际长度，即从字符串的开头到第一个空字符（'\0'）的位置。这意味着 strlen 只能用于字符串，并且会忽略字符串结束符。使用 strlen 函数时，需要包含头文件 string.h

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

另一方面，sizeof 是一个单目运算符，用于计算数据类型或对象在内存中所占的空间大小（以字节为单位）。它不仅可以用于数组、指针、类型、对象等，而且与初始化也有一定的关系。sizeof 的返回值是字符个数乘以每个字符所占的字节数。例如，对于字符串数组，其大小等于实际的字符个数加 1（包括结束符'\0'）。对于整型数组，其大小则是实际的字符个数乘以每个整型数据所占的字节数。

此外，在计算时间上，strlen 是在运行时计算的，而 sizeof 则在编译时就能确定其值。这是因

为 sizeof 是一个编译时运算符，它直接返回对象或类型所占的内存大小，而不需要在运行时进行计算。

总的来说，strlen 和 sizeof 的主要区别在于它们的定义、用途、计算时间和使用方式。strlen 主要用于计算字符串的长度，而 sizeof 则用于计算数据类型或对象在内存中的大小。

12. 二维数组是什么，函数指针是什么？（已完成）

二维数组是一种特殊的数据结构，用于存储表格形式的数据。在 C 或 C++等编程语言中，二维数组可以看作是一个数组的数组。例如，在 C 语言中，你可以声明一个二维数组如下：

```
int array[3][4];
```

这表示 array 是一个有 3 个元素的数组，每个元素又是一个包含 4 个整数的数组。你可以通过两个索引来访问数组中的元素，例如 array[i][j]，其中 i 是行索引，j 是列索引。

函数指针是一个指向函数的指针。在 C 或 C++中，每个函数都有一个地址，函数指针就是存储这个函数地址的变量。通过函数指针，你可以间接地调用函数。例如，在 C 语言中，你可以这样声明一个函数指针：

```
int (*func_ptr)(int, int);
```

这表示 func_ptr 是一个指向函数的指针，这个函数接受两个 int 类型的参数，并返回一个 int 类型的值。然后，你可以将这个函数指针指向一个具体的函数，例如：

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

func_ptr = add;
```

现在，func_ptr 就指向了 add 函数，你可以通过 func_ptr 来调用 add 函数：

```
int sum = func_ptr(2, 3);  // sum 的值为 5
```

函数指针在编程中有很多用途，例如回调函数。

13. 简述值传递、指针传递（地址传递）的区别（已完成）

值传递和指针传递在参数传递时的主要区别在于处理方式和内存管理的不同。

在值传递中，函数的形参是实参的拷贝，即函数内部会创建新的内存空间来存储这个拷贝。这意味着在函数内部对形参的任何修改都不会影响到实参，因为它们是存储在不同内存位置的两个独立变量。值传递是单向的，参数值只能传入函数，不能从函数传出。当函数内部需要修改某些值，并且不希望这个改变影响调用者时，通常使用值传递。

指针传递则不同，它本质上也是值传递，但传递的是变量的地址值，而非变量的实际内容。在函数内部，形参会接收到实参的地址，因此通过形参可以访问和修改实参的值。这是因为地址指向的是同一块内存空间，所以任何对形参的修改都会直接影响到实参。指针传递允许函数修改并返回修改后的值给调用者。

总的来说，值传递保证了函数内部对参数的操作不会影响到外部实参，而指针传递则允许函数直接修改外部实参的值。在选择使用哪种传递方式时，需要根据具体的编程需求和场景来决定。

14. C++中 const 关键字的作用？（已完成）

在 C++中，const 关键字主要用于定义常量，也就是值在初始化后不能被改变的量。以下是 const 关键字在 C++中的一些主要用途：

1) 定义常量：你可以使用 const 关键字来声明一个常量，一旦初始化后，其值就不能被改变。

```
const int kConstantValue = 10;
```

2) 指向常量的指针：指针所指内容的值不能修改，但是可以改变指向。

```
int value = 1, value2 = 2;
const int* ptrToConst = &value; // ptrToConst 是一个指向常量的指针
// 下面的代码是错误的，因为不能通过指向常量的指针来修改它所指向的值
// *ptrToConst = 30; // 错误：不能修改指向常量的指针所指向的值
ptrToConst = &value2; //ok
```

3) 指针常量：指针本身的值无法修改，指向不能修改, 但是可以修改指向的内容。

```
int value = 10;
int* const ptr = &value; // ptr 是一个指针常量，指向一个 int 类型的变量 value

// 下面的代码是错误的，因为不能改变常量指针 ptr 的值（即它所指向的地址）
// ptr = &anotherValue; // 错误：不能给指针常量赋新的地址
// 下面的代码是正确的，因为可以通过常量指针修改它所指向的值
*ptr = 20; // 正确：修改了 ptr 指向的值，现在 value 的值为 20
```

4) 函数参数：在函数参数中使用 const 可以确保传递给函数的参数在函数体内不会被修改。

函数参数：在函数参数中使用 const 可以确保传递给函数的参数在函数体内不会被修改。

5) 类数据成员：在类定义中，const 成员变量必须在构造函数的初始化列表中初始化，并且之后不能被修改。同时，const 成员函数不能修改类的任何成员变量（除非它们被声明为 mutable）。

```
class MyClass {
public:
    MyClass(int value) : m_value(value) {}

    void setValue(int value) {
        m_value = value;  // 错误，如果 m_value 是 const
    }

    int getValue() const {
        return m_value;
    }

private:
    const int m_value;
};
```

6) const 成员函数：是指那些不会修改调用它的对象状态的成员函数。通过在成员函数的声明或定义后添加 const 关键字，可以确保该函数不会修改任何成员变量（除非它们被声明为 mutable）。这有助于在逻辑上区分只读操作和修改操作，并允许编译器进行额外的检查以确保成员函数不会意外地修改对象。

```
class MyClass {
private:
    int value;
public:
    MyClass(int v) : value(v) {}
    // 非 const 成员函数，可以修改成员变量
    void setValue(int v) {
        value = v;
    }

    // const 成员函数，不会修改成员变量
    int getValue() const {
        return value;
    }
};

int main() {
    MyClass obj(10);
    obj.setValue(20); // 正确：调用非 const 成员函数
    int val = obj.getValue(); // 正确：调用 const 成员函数

    // 下面的代码是错误的，因为不能在一个 const 对象上调用非 const 成员函数
    // const MyClass constObj(30);
    // constObj.setValue(40); // 错误：不能在 const 对象上调用非 const 成员函数

    // 下面的代码是正确的，可以在 const 对象上调用 const 成员函数
    const MyClass constObj(30);
    int constVal = constObj.getValue(); // 正确：调用 const 成员函数
```

15. C++中 static 关键字的作用？（已完成）

在 C++中，static 关键字具有多种用途，根据它应用的上下文，它可以起到不同的作用。以下是 static 关键字在 C++中的主要作用：

1) 局部静态变量：当在函数内部声明一个静态局部变量时，该变量的生命周期会持续到程序结束，而不是当函数返回时结束。此外，静态局部变量只初始化一次，而不是每次函数被调用时都初始化。

```
void countCalls() {
    static int callCount = 0; // 只在第一次调用时初始化
    callCount++;
    std::cout << "Function has been called " << callCount << " times." << std::endl;
}
```

2) 静态全局变量：当在文件作用域中声明一个静态变量时（即不在任何函数内部），该变量的作用域被限制在定义它的文件中。这有助于防止全局变量的命名冲突，并隐藏了不需要在其他文件中访问的变量。

```
// file1.cpp
static int fileScopeStatic = 42; // 只在 file1.cpp 中可见
// file2.cpp
extern int fileScopeStatic; // 错误：file2.cpp 中不可见
```

3) 静态函数：当在文件作用域中声明一个静态函数时，该函数的作用域被限制在定义它的文件中。静态函数不能在其它文件中通过 extern 声明来访问。

```
// file1.cpp
static void staticFunction() { /* ... */ } // 只在 file1.cpp 中可见

// file2.cpp
extern void staticFunction(); // 错误：file2.cpp 中不可见
```

4) 静态数据成员：是 C++中类的一种特殊数据成员，它属于类本身，而不是类的任何特定对象。静态数据成员在内存中只占一份空间，并被该类的所有对象所共享。每个对象都可以访问这个静态数据成员，且静态数据成员的值对所有对象都是一样的。如果改变它的值，

则在各对象中这个数据成员的值都同时改变了。这样可以节约空间，提高效率。

5) 静态成员函数：静态成员函数只能访问静态成员变量和其他静态成员函数，而不能直接访问类的非静态成员变量或成员函数。这是因为非静态成员是依赖于对象的，而静态成员函数可以在没有对象的情况下被调用。

```
class MyClass {
public:
    static void staticFunction() {
        // 实现一些功能，不依赖于任何特定对象的状态
    }
};
int main() {
    MyClass::staticFunction(); // 直接通过类名调用静态成员函数
    return 0;
}
```

16. C++中 class 和 struct 的区别？（已完成）

在 C++中，class 和 struct 都可以用来定义类，它们的主要区别体现在成员的默认访问级别上。

1) 默认访问级别：

对于 class，其成员的默认访问级别是 private。这意味着，如果你在 class 中定义了一个成员，但没有明确指定其访问级别，那么它将被视为 private 成员，只能在类的内部访问。

对于 struct，其成员的默认访问级别是 public。这意味着，如果你在 struct 中定义了一个成员，但没有明确指定其访问级别，那么它将被视为 public 成员，可以在类的外部访问。

2) 继承时的默认访问控制：

当使用 class 进行继承时，默认的继承方式是 private 继承。

当使用 struct 进行继承时，默认的继承方式是 public 继承。

3) 语义上的区别：

在传统的 C++编程中，class 通常用于定义那些封装了数据并提供了数据操作的类，其中大部分成员是私有的。

struct 则更多地被用作数据结构的定义，其中成员大多是公共的，用于简单地组合数据。

17. string 的 SSO 模式(已完成)

String 的 SSO（Short String Optimization，短字符串优化）模式是一种针对字符串内存分配的优化策略。其核心思想是在字符串长度较小时，直接在栈上进行内存分配，以减少内存分配和拷贝的开销。具体来说，当字符串的长度小于或等于某个阈值（通常为 15 或 16 个字节）时，SSO 策略会直接在栈上分配一个小的缓冲区来存储整个字符串。这样，当需要拷贝这样的字符串时，只需要复制指针或者少量的数据，而不需要进行复杂的内存分配和拷贝操作。

这种优化策略的主要好处在于提高了小字符串的处理效率，减少了内存管理的开销。然而，当字符串长度超过阈值时，SSO 策略将不再适用，此时字符串将采用传统的堆内存分配方式。

需要注意的是，SSO 模式的具体实现可能因编译器和库的不同而有所差异。一些编译器和库可能采用了 SSO 策略来优化字符串处理，而另一些则可能采用其他策略。因此，在实际编程中，了解所使用的编译器和库对字符串的处理方式是非常重要的。

总的来说，SSO 模式是一种针对小字符串的优化策略，通过直接在栈上分配内存来减少内存分配和拷贝的开销，提高了字符串处理的效率。

18. C++中类型转换有哪几种？简述一下它们之间的区别。

在 C++中，类型转换（Type Casting）或类型转换操作符用于将一种数据类型的值转换为另一种数据类型。C++支持多种类型转换，主要可以分为四大类：静态转换（Static Cast）、动态转换（Dynamic Cast）、常量转换（Const Cast）和重新解释转换（Reinterpret Cast）。下面我将分别介绍它们并简述它们之间的区别。

1）静态转换（Static Cast）

静态转换是最常用的转换方式，用于非多态类型的转换，例如基本数据类型之间的转换、非多态类之间的转换、空指针和整型之间的转换、以及 void*到其他指针类型的转换等。静态转换在编译时进行，不执行任何运行时检查。

```
int a = 42;
double b = static_cast<double>(a);  // 基本数据类型之间的转换
```

2) 动态转换（Dynamic Cast）

动态转换主要用于在类继承体系中进行安全的向下和侧向转换。它用于执行运行时类型信息（RTTI）检查，以确保转换的安全性。如果转换不可能成功（例如，将一个基类指针转换为派生类指针，而该基类指针实际上并不指向派生类对象），则动态转换将返回空指针。

```
class Base {};
class Derived : public Base {};

Base* base = new Derived;
Derived* derived = dynamic_cast<Derived*>(base);  // 安全的向下转换
```

3) 常量转换（Const Cast）

常量转换用于添加或删除类型的 const 或 volatile 属性。它允许你修改一个本应为常量的对象，或者将一个非常量对象视为常量对象。但是，使用常量转换来修改一个实际上应为常量的对象是不安全的，并且可能导致未定义的行为。

```
const int a = 42;
int* b = const_cast<int*>(&a);  // 去除 const 属性，但修改*b 是不安全的
```

4) 重新解释转换（Reinterpret Cast）

重新解释转换提供了一种最低级别的转换方式，它告诉编译器将某个类型的位模式重新解释为另一种类型的位模式。这种转换基本上只是告诉编译器如何解释内存中的位，而不进行任何类型的检查或转换。因此，使用重新解释转换时必须格外小心，因为它很容易导致未定义的行为。

```
int a = 42;
char* b = reinterpret_cast<char*>(&a);  // 将 int 的地址重新解释为 char 的指针
```

区别总结：

1) 静态转换是最常用的转换方式，主要用于非多态类型的转换。
2) 动态转换用于在类继承体系中进行安全的向下和侧向转换，执行运行时类型检查。
3) 常量转换用于添加或删除类型的 const 或 volatile 属性，但使用时要小心，避免修改应为常量的对象。
4) 重新解释转换提供了最低级别的转换方式，只是重新解释内存中的位模式，使用时必须格外小心以避免未定义行为。

19. C++中函数指针和指针函数的区别？

在 C++中，函数指针和指针函数是两个不同的概念，它们有着本质的区别。

1) 函数指针是一个指针，它指向一个函数。这意味着你可以通过函数指针来调用函数。

如以下代码

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

void test() {
       int (*func_ptr)(int, int) = &add;//定义一个函数指针
       int sum = func_ptr(3, 4);  // sum 的值为 7
}
```

2) 指针函数：则是一个函数，这个函数的返回值是一个指针。它的声明格式类似于常规的函数声明，只不过它的返回类型是一个指针类型。例如：

```
int* getPointer() {
    static int x = 10;
    return &x;
}
```

20. void*的大小是多少？

占据一个指针的大小。

21.简述一下 malloc / free 的实现原理。

malloc 和 free 是 C 语言标准库中用于动态内存分配和释放的函数。它们的实现原理涉及操作系统的内存管理机制以及可能的数据结构来管理已分配和未分配的内存块。下面是对 malloc 和 free 实现原理的简述：

malloc 的实现原理

1) 请求内存大小：malloc 函数接受一个参数，表示请求的内存大小（以字节为单位）。
2) 查找空闲内存块：malloc 会维护一个或多个内存池（通常称为堆），这些内存池包含了可用的（未分配）和已分配的内存块。malloc 会遍历这些内存池，查找一个足够大的空闲内存块来满足请求。
3) 内存块分割：如果找到了一个比请求大小大的空闲内存块，malloc 可能会将其分割成两部分：一部分用于满足当前请求，另一部分保留为新的空闲内存块。
4) 内存块管理：malloc 通常会为分配的内存块添加一些额外的元数据，如内存块的大小、下一个空闲块的指针等，以便后续的内存分配和释放操作能够高效进行。
5) 返回指针：最后，malloc 返回指向分配的内存块的指针。如果无法找到足够的空闲内存来满足请求，malloc 会返回 NULL。

free 的实现原理

1) 获取内存块指针：

free 函数接受一个指向之前通过 malloc 分配的内存块的指针。

2) 查找内存块：

使用传递给 free 的指针，系统会在其维护的内存池结构中查找对应的内存块。这通常涉及遍历内存池，并使用内存块中的元数据来识别正确的内存块。

3) 合并空闲内存块：

如果已释放的内存块前后有其他的空闲内存块，free 可能会尝试将这些空闲内存块合并成一个更大的空闲内存块，以提高内存使用效率。

4) 更新内存池状态：

free 会更新内存池的状态，标记已释放的内存块为空闲，并可能更新相关的元数据（如空闲块的大小和位置）。

5) 防止悬挂指针：

虽然 free 释放了内存，但它并不清除指针本身的值。因此，程序员需要确保在调用 free 后不再使用该指针，以避免悬挂指针（dangling pointer）问题。

注意事项

线程安全：在多线程环境中，malloc 和 free 的实现必须确保线程安全，以避免竞态条件和数据不一致。

内存碎片：频繁的 malloc 和 free 调用可能导致内存碎片，即内存中有很多小的、不连续的空闲块，难以用于满足大的内存请求。一些实现会采用内存整理（compaction）或其他技术来减少碎片。

对齐和填充：为了性能和平台兼容性的原因，malloc 可能会为分配的内存块添加一些对齐和填充字节。

malloc 和 free 的具体实现可能会因操作系统、编译器和库的不同而有所差异，但它们的核心原理是相似的。这些函数为程序员提供了一种灵活的方式来管理程序的运行时内存需求。

22.为何 free(p) 的时候，只需要传递堆空间的地址就可以了?

23. malloc 申请内存后，怎么保证一定申请到了呢？你会申请完了后直接使用这片内存吗？

malloc 在 C 语言中用于在堆上动态分配内存。然而，malloc 并不保证总是能够成功申请到内存。当系统内存不足时，malloc 会返回 NULL。因此，为了保证在使用 malloc 申请内存后确实获得了所需的内存，你应该始终检查 malloc 的返回值。申请完之后，一般要进行初始化操作，否则容易出现问题。

以下是一个简单的示例，展示了如何在使用 malloc 后检查是否成功申请到了内存：

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

int main() {
    size_t size = 100; // 假设要申请的内存大小是 100 字节
    void *ptr = malloc(size); // 申请内存

    // 检查 malloc 是否成功
    if (ptr == NULL) {
        // malloc 失败，处理错误情况
        perror("Memory allocation failed");
        return EXIT_FAILURE;
    } else {
        // malloc 成功，使用 ptr 指向的内存
       memset(ptr, 0, size); //做初始化操作
        // ... 在这里对 ptr 指向的内存进行操作 ...

        // 不要忘记在不再需要内存时释放它
        free(ptr);
    }
    return EXIT_SUCCESS;
}
```

在上面的代码中，malloc 被用来申请 size 字节的内存。如果 malloc 返回 NULL，则通过 perror 函数打印出错误信息，并退出程序。如果 malloc 成功，则可以使用 ptr 指向的内存，并在不再需要时调用 free 来释放它。

请注意，即使 malloc 成功返回了一个非 NULL 的指针，也不意味着你可以无限制地写入这块内存。你应该始终确保不越界写入，即不超过申请的内存大小。越界写入可能导致程序崩溃、数据损坏或其他未定义行为。

此外，如果你的程序对内存的需求很大，或者需要在内存紧张的环境下运行，那么你应该考虑实现一些策略来管理内存，例如内存池（memory pooling）、提前分配大块内存并分割使用等。这些策略可以帮助减少 malloc 和 free 的调用次数，从而提高性能并减少内存碎片。

24. new / delete 与 malloc / free 的异同

new / delete 和 malloc / free 都是 C++ 中用于动态内存管理的操作符和函数，但它们之间存在一些重要的差异。

相同点：

1）两者都用于在堆上动态分配和释放内存。

2）在使用完毕后，都需要显式地释放分配的内存，以避免内存泄漏。

不同点：

1) 类型安全性：
    a) new 和 delete 是 C++ 操作符，它们是类型安全的。new 在分配内存时，还会调用对象的构造函数进行初始化；delete 在释放内存时，会调用对象的析构函数进行清理。这确保了对象的状态在生命周期内得到正确的管理。
    b) 相比之下，malloc 和 free 是 C 语言的标准库函数，它们在 C++ 中仍然可用，但它们是类型不安全的。malloc 仅仅分配指定大小的内存，并不会调用任何构造函数；free 也仅仅释放内存，不会调用析构函数。因此，在使用 malloc 和 free 时，需要额外注意对象的状态管理。
2) 异常安全性：
    a) new 操作符在内存分配失败时会抛出 std::bad_alloc 异常，这使得程序员能够更优雅地处理内存分配失败的情况。
    b) malloc 在内存分配失败时返回 NULL，需要程序员显式地检查返回值以处理错误。
3) 内存对齐：
    a) new 和 delete 考虑了对象的内存对齐要求，确保对象按照其类型的要求正确地对齐。
    b) malloc 和 free 则不保证特定的内存对齐，除非明确指定。
4) 效率：在某些情况下，malloc 和 free 可能比 new 和 delete 更快，因为它们不涉及构造函数和析构函数的调用。然而，这种差异通常在现代编译器和优化技术下变得不那么显著。
5) 自定义内存管理：
    a) new 和 delete 可以与自定义的内存分配器一起使用，以实现更复杂的内存管理

b) malloc 和 free 则通常与系统的默认内存分配器一起使用。

25. delete 与 delete[]的区别

delete 和 delete[] 都是 C++ 中用于释放动态分配的内存的操作符，但它们的主要区别在于它们所释放的对象类型：

1) delete：用于释放单个对象通过 new 分配的内存。例如，int* p = new int; 分配了一个整数对象的内存，随后应使用 delete p; 来释放这块内存。
2) delete[]：用于释放通过 new[] 分配的数组的内存。例如，int* arr = new int[10]; 分配了一个包含 10 个整数的数组的内存，随后应使用 delete[] arr; 来释放整个数组的内存。
3) 不正确地使用 delete 和 delete[] 可能会导致内存泄漏或未定义行为。使用 delete 释放一个通过 new[] 分配的数组，或者反之，都是错误的。

26. 指针和引用的区别

指针和引用是 C++ 中两种用于间接访问对象或变量的机制，但它们之间存在一些重要的区别：

1) 初始化：

指针可以不初始化，而引用必须初始化。

指针可以指向其他对象或重新赋值为 nullptr，而引用一旦初始化后，就不能再指向其他对象。

2) 内存分配：

指针本身是一个变量，存储了另一个变量的地址，因此需要为其分配内存。

引用只是为已存在的变量提供别名，不分配额外的内存。

3) 空值：

指针可以为 nullptr，表示它不指向任何对象。

引用没有空值的概念，它总是指向某个对象。

4) 操作：

可以对指针进行算术运算（如加、减），使其指向数组中的不同元素。

引用不能进行算术运算。

5) 函数参数：

当函数参数为指针时，可以修改指针本身的值（即它指向的地址），但通常用于修改指针指向的内容。

当函数参数为引用时，通常用于直接修改原始变量的值，而不是修改引用的地址。

6) 返回值：

函数可以返回指针，指向动态分配的内存或堆栈上的对象（但要注意堆栈对象在函数返回后可能不再有效）。

函数也可以返回引用，但通常用于返回类成员或全局变量的引用。

27. 引用作为函数返回时为什么不能返回局部变量？

引用作为函数返回时不能返回局部变量的原因与引用的本质和生命周期有关。

引用在 C++ 中是变量的别名，它并不拥有自己独立的存储空间，而是与所引用的对象共享同一块内存。因此，引用必须在其整个生命周期内都有效，即它所引用的对象必须始终存在。

局部变量是在函数内部定义的变量，它们的生命周期仅限于函数执行期间。当函数返回时，局部变量的存储空间会被释放，它们的值不再有效。

如果函数试图返回一个局部变量的引用，那么在函数返回后，这个引用所指向的局部变量已经不再存在，导致引用悬垂（dangling reference）。任何尝试通过这个悬垂引用访问或修改原始局部变量的行为都是未定义的，可能导致程序崩溃或其他不可预测的错误。

因此，为了避免引用悬垂和相关的错误，C++ 不允许返回局部变量的引用。如果需要在函数外部访问局部变量，应该考虑使用动态分配的内存（如使用 new 操作符）或将其设计为全局变量或类的成员变量。但需要注意的是，动态分配的内存必须在不再需要时显式释放，而全局变量和类成员变量则需要注意其生命周期和访问权限。

28. extern"C"的用法

extern "C"是 C++中的一个链接规范，用于告诉 C++编译器将随后的代码（函数或变量）以 C 语言的方式进行链接。这主要是为了解决 C++与 C 语言混合编程时的链接问题，因为 C++支持函数重载，所以编译器会对函数名进行名字修饰（name mangling），而 C 语言则不会。

使用 extern "C"的主要场景包括：

1) 当 C++代码需要调用 C 库中的函数时。

2) 当 C 代码需要调用 C++编写的函数时，并且这些函数不打算被重载。

示例：

```
// 在 C++代码中声明一个 C 风格的函数
extern "C" {
    void myFunction();
}

// 在 C 代码中定义这个函数
void myFunction() {
    // ... 函数实现 ...
}
```

29. 内联函数和宏定义的区别

内联函数和宏定义都是用于优化代码性能的工具，但它们之间存在一些重要的区别：

1) 内联函数：
   a) 内联函数是编译器优化的手段，它在编译时将函数体插入到每个调用点，从而避免了函数调用的开销。
   b) 内联函数是真正的函数，具有类型检查、作用域、参数列表等特性。
   c) 内联函数可以像普通函数一样进行调试。
   d) 内联函数如果定义不当，可能会导致代码膨胀，反而降低性能。
2) 宏定义：
   a) 宏定义是预处理指令，它在预处理阶段进行文本替换，不经过编译器类型检查。
   b) 宏定义没有作用域，定义后在整个文件内都有效（除非使用#undef 取消定义）。
   c) 宏定义可能导致一些难以追踪的错误，比如操作符优先级问题或多次求值问题。
   d) 宏定义通常用于定义常量或简单的计算。

30. 静态变量什么时候初始化

静态变量的初始化发生在程序开始执行之前，即在 main 函数被调用之前。对于局部静态变量，它只会在第一次进入其作用域时进行初始化，之后每次进入该作用域时都不会再次初始

化。对于全局静态变量，它在程序加载时就被初始化。

静态变量的初始化顺序如下：

1) 全局静态变量的初始化按照它们在文件中出现的顺序进行。
2) 局部静态变量的初始化在第一次进入其作用域时进行。

需要注意的是，静态变量的初始化是确定性的，即在程序执行过程中只发生一次。而且，静态变量的初始化顺序在跨编译单元（即不同的源文件）时可能会产生问题，因为不同编译单元中的全局变量初始化顺序是不确定的。

31. 动态编译与静态编译

动态编译（也称为动态链接）和静态编译（也称为静态链接）是两种不同的链接方式，它们影响着程序如何与库进行交互以及程序的最终大小。

1) 静态编译：
    a) 在静态编译中，所有需要的库和对象文件都被直接包含在最终的程序文件中。这意味着程序在运行时不需要再查找或加载任何外部库。
    b) 静态编译的程序通常较大，因为它包含了所有必要的代码和数据。
    c) 静态编译的程序可以在没有安装相应库的系统上运行，因为它自带了所有需要的库。
    d) 但如果库有更新或安全修复，静态编译的程序不会受益，除非重新编译。

2) 动态编译：
    a) 在动态编译中，程序只包含对外部库的引用，而不是库的实际内容。当程序运行时，它会加载这些外部库。
    b) 动态编译的程序通常较小，因为它不包含库的内容。
    c) 动态编译的程序依赖于系统上安装的库。如果库没有安装或版本不兼容，程序可能无法运行。
    d) 但动态编译的程序可以受益于库的更新和修复，因为它们在运行时加载的是系统上的库版本。

在选择使用静态编译还是动态编译时，需要考虑程序的部署环境、大小要求、以及是否需要与库的更新保持同步等因素。

32.拷贝构造函数的调用时机：

拷贝构造函数通常在以下几种情况下被调用：

使用一个已存在的对象去初始化一个同类的新对象时。

函数参数是类的对象时，以值传递的方式传递对象。

函数返回值是类的对象时，以值返回的方式返回对象。

33.inline 函数的使用？缺点是什么？

inline 函数是建议编译器将函数调用内联展开的函数，即直接将函数体插入到每个调用点，以减少函数调用的开销。使用 inline 关键字可以提示编译器内联。但缺点是，如果函数体很大或过于复杂，内联展开可能导致代码膨胀，反而降低性能。此外，内联并不是强制的，编译器会根据自己的优化策略决定是否内联。

34.为什么拷贝构造函数必须传引用不能传值？

拷贝构造函数如果传值，那么在调用拷贝构造函数时，会再次调用拷贝构造函数来复制参数，这会导致无限递归。因此，拷贝构造函数通常通过传引用或指针来避免这种情况。而引用可以防止传递过程中的拷贝，确保效率和正确性。

35.类中静态函数占用内存么：

类中的静态函数和普通函数一样，不占用类对象的内存空间。它们是在程序加载时分配到代码段（text segment）的，每个静态函数只有一份代码，所有对象共享这份代码。

36.构造函数初始化和列表初始化的区别：

构造函数初始化是在构造函数体内对成员变量进行赋值操作，而列表初始化是在构造函数的初始化列表中直接对成员变量进行初始化。列表初始化通常更高效，因为它可以在对象构造时就完成成员变量的初始化，避免不必要的临时变量和拷贝操作。

37.虚函数表的作用和存储的地址：

虚函数表是 C++中用于支持动态多态性的数据结构。它存储了类中所有虚函数的地址，并关联到类的一个实例上。每个有虚函数的类（包括从有虚函数的类继承而来的类）都会有一个虚函数表。虚函数表的地址通常存储在对象的内存布局中的固定位置，可以通过对象的指针或引用

来访问。

38.类指针初始化为空指针后调用成员函数会出问题吗？

类指针初始化为空指针后调用成员函数是危险的，因为这会导致运行时错误（通常是段错误），因为程序试图在空指针指向的内存地址上执行操作。在调用成员函数之前，必须确保指针指向一个有效的对象。

39.指针和引用的区别：

指针是一个变量，存储了另一个变量的地址，可以指向不同的对象，也可以为空。引用是变量的别名，必须在声明时初始化，并且一旦初始化后就不能再指向其他对象。引用没有自己的内存地址，它和所引用的对象共享同一块内存。

40.动态指针的判空：

对于动态分配的指针，可以使用 if (pointer == nullptr)或 if (!pointer)来判断指针是否为空。在 C++11 及以后的版本中，推荐使用 nullptr 代替传统的 NULL 或 0 来表示空指针。

41.构造函数可以设置成虚函数吗，为什么？

构造函数不能是虚函数。虚函数表是在对象构造完成后才与对象关联的，而构造函数在对象构造过程中就被调用，因此无法访问虚函数表。此外，构造函数的主要任务是初始化对象，而不是实现动态多态性，所以也没有必要设为虚函数。

42.new 和 malloc 之间有什么区别？

new 和 malloc 都是用于动态分配内存的函数，但它们在用法和特性上有一些区别：

new 是 C++运算符，而 malloc 是 C 语言的库函数。

new 分配内存时会调用对象的构造函数进行初始化，而 malloc 只是分配内存，不会进行初始化。

new 分配的内存在不需要时会自动调用析构函数并释放内存，而 malloc 分配的内存需要通过 free 函数来释放。

new 分配失败时会抛出异常，而 malloc 分配失败时返回 NULL。

new 在申请内存空间时不需要类型转化，是因为编译器会根据 new 后面跟的类型自动进行类型转换。底层实现上，new 会调用 malloc（或类似的函数）来分配内存，并调用构造函数进行初始化。

43.虚函数表里存放的内容是什么时候写进去的？

虚函数表里的内容是在编译阶段确定的。编译器会遍历类的继承层次结构，找出所有的虚函数，并将它们的地址按照一定顺序存放到虚函数表中。每个类对象都包含一个指向其类对应虚函数表的指针，这个指针是在对象构造时由编译器自动设置的。

44.虚函数和纯虚函数的区别：

虚函数是可以在基类中声明并在派生类中重写的函数，它支持动态多态性。

纯虚函数是在基类中声明为 virtual 且函数参数列表之后必须要加上= 0 的函数，它没有实现，派生类必须提供实现。包含纯虚函数的类称为抽象类，不能实例化。纯虚函数主要用于定义接口，实现多态性。

45.为什么析构函数一般写成虚函数

析构函数一般写成虚函数是为了确保当通过基类指针或引用来删除派生类对象时，能够正确调用派生类的析构函数，从而避免资源泄露和未定义行为。如果不将析构函数声明为虚函数，那么在删除派生类对象时，只会调用基类的析构函数，派生类的析构函数不会被调用，这可能导致派生类特有的资源（如动态分配的内存、打开的文件句柄等）无法被正确释放。

46.动态多态的实现过程和静态多态的实现过程

动态多态的实现过程：动态多态主要通过虚函数和虚函数表来实现。在基类中声明虚函数，并在派生类中重写该函数。当通过基类指针或引用来调用虚函数时，会根据对象的实际类型（即运行时类型）来确定调用哪个版本的函数。编译器在编译时插入额外的代码来查找虚函数表，并根据对象的虚函数表指针来确定正确的函数地址。这种机制允许在运行时根据对象的实际类型来改变行为，从而实现多态性。

静态多态的实现过程：静态多态主要通过函数重载和模板来实现。函数重载允许在相同的作用域内定义多个名称相同但参数列表不同的函数。在编译时，编译器会根据函数的参数类型和数

量来确定调用哪个版本的函数。模板则允许编写通用的代码，并在编译时根据模板参数的具体类型来生成特定类型的代码。这种机制在编译时就确定了函数或类的行为，因此称为静态多态。

47.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？

当从 vector 中删除元素时，迭代器可能会失效。具体来说，删除元素后，指向被删除元素及其之后元素的迭代器都会失效，因为删除操作可能导致 vector 内部元素的重新排列或内存重新分配。因此，在删除元素后，任何之前获取的迭代器都不应再使用。如果需要继续遍历 vector，应该使用新的迭代器或者通过其他方式获取有效的迭代器。

vector 的扩容策略通常是成倍增长，例如当需要更多空间时，可能会将容量翻倍。这种策略可以减少扩容操作的次数，从而提高性能。每次扩容都需要分配新的内存并将现有元素复制到新内存中，这是一个相对昂贵的操作。因此，通过成倍增长容量，可以减少这种操作的频率，从而提高整体性能。

关于释放空间，vector 在删除元素时通常不会立即释放其占用的所有内存。这是因为频繁地分配和释放小块内存可能导致性能下降。相反，vector 会保留一些额外的容量以应对未来的增长。这可以通过成员函数如 shrink_to_fit 来手动触发释放多余的空间，但请注意这并不保证一定会释放所有额外空间，因为实现可能会保留一些容量以优化性能。

48.map、set 是怎么实现的，红黑树是怎么能够同时实现这两种容器？为什么使用红黑树？

map 和 set 在 STL 中通常使用红黑树来实现。红黑树是一种自平衡的二叉搜索树，它通过对树进行着色和旋转操作来保持树的平衡，从而确保查找、插入和删除操作的时间复杂度为对数级别。

红黑树能够同时实现 map 和 set 这两种容器，是因为它们都需要存储键值对（对于 map）或唯一键（对于 set），并且需要按照键的顺序进行访问。红黑树的特性使得它能够在 O(log n)的时间复杂度内完成这些操作。对于 map，键作为红黑树的节点，值则作为节点的附加信息；对于 set，键本身就是红黑树的节点。

使用红黑树而不是其他数据结构（如哈希表或链表）来实现 map 和 set 的主要原因包括：

红黑树保证了元素的顺序性，这对于需要按照特定顺序访问元素的场景非常有用。

红黑树的平衡性确保了操作的时间复杂度为对数级别，这在处理大量数据时非常高效。

相对于哈希表，红黑树不需要计算哈希值，因此可以避免哈希冲突带来的性能开销。

相对于链表，红黑树在查找和插入操作上具有更好的性能。

49.模板和实现可不可以不写在一个文件里面？为什么？

C++中的模板通常需要在同一个文件中同时包含声明和定义，以便编译器能够正确地进行实例化和代码生成。这是 C++编译模型的限制所决定的。

50.请简述你了解及使用过的 C++11 的新特性

（提示：如 类型推导、智能指针、lambda 匿名函数等）

C++11 引入了许多新的语言特性和库，下面是其中一些我了解并使用过的特性：

- 类型推导：C++11 引入了 auto 关键字，可以根据变量的初始化表达式自动推导出变量的类型。这样可以简化代码，减少类型冗余，并且在使用模板时更加方便。
- 智能指针：C++11 引入了 unique_ptr、shared_ptr 和 weak_ptr 等智能指针类型，用于管理动态分配的内存资源。智能指针可以自动处理内存的释放，减少内存泄漏和悬空指针的风险，并提供更安全和方便的内存管理方式。
- lambda 表达式：是一种定义匿名函数的方式，可以在需要函数对象的地方使用，如算法函数、函数对象和回调函数等。lambda 表达式可以简化代码，使得函数对象的定义和使用更加灵活和便捷。
- 范围 for 循环：C++11 引入了范围 for 循环，可以方便地遍历容器或可迭代的对象。范围 for 循环可以简化遍历代码的编写，并且提供了更安全和易读的遍历方式。
- 移动语义：C++11 引入了右值引用和移动语义的概念。移动语义允许对象的资源（如动态分配的内存）在移动（如赋值和函数调用）时被转移而不是复制，从而提高性能和效率
- 初始化列表（Initializer Lists）：C++11 允许使用初始化列表来初始化对象，包括数组、容器和自定义类型等。初始化列表提供了一种简洁和直观的初始化方式，并且可以避免一些初始化相关的问题。

还有其他许多特性，如多线程支持、正则表达式库、委托构造函数等。这些新特性扩展了 C++的功能和灵活性，提高了编程效率和代码质量。

51.说一说你了解的关于 lambda 函数的全部知识

1、基本语法特性

[capture](params) opt -> retureType

{

    body;

};

其中 capture 是捕获列表，params 是参数列表，opt 是函数选项，retureType 是返回值类型，body 是函数体。

1.1、捕获列表

不能省略。捕获一定范围内的变量。

- [] 不捕捉任何变量

- [&] 捕获外部作用域中所有变量, 并作为引用在函数体内使用 (按引用捕获)

- [=] 捕获外部作用域中所有变量, 并作为副本在函数体内使用 (按值捕获)，拷贝的副本在匿名函数体内部是**只读的**

- [=, &foo] 按值捕获外部作用域中所有变量, 并按照引用捕获外部变量 foo

- [bar] 按值捕获 bar 变量, 同时不捕获其他变量

- [&bar] 按引用捕获 bar 变量，同时不捕获其他变量

- [&，bar] 其他变量按引用捕获，bar 变量按值捕获

- [this] 捕获当前类中的 this 指针，让 lambda 表达式拥有和当前类成员函数同样的访问权限；如果已经使用了 & 或者 =, 默认添加此选项

1.2、参数列表

和普通函数的参数列表一样，如果没有参数，参数列表可以省略不写。

1.3、opt 选项

可以省略。

mutable: 可以修改按值传递进来的拷贝（注意是能修改拷贝，而不是值本身）

exception: 指定函数抛出的异常，如抛出整数类型的异常，可以使用 throw();

1.4、返回类型 retureType

可以省略。通过返回值后置语法来定义的。

1.5、函数体 body

不能省略。函数的实现，这部分不能省略，但函数体可以为空

52. C++中的智能指针？三种指针解决的问题以及区别？

C++中有三种常见的智能指针：unique_ptr、shared_ptr 和 weak_ptr。

问题：unique_ptr 主要用于解决单一所有权的问题。在传统的指针使用中，我们需要手动释放动态分配的内存，容易忘记或出现异常时无法正常释放，导致内存泄漏。unique_ptr 通过独占资源的方式，确保只有一个指针指向资源，并负责在其生命周期结束时自动释放资源，避免了显式的内存管理。

区别：unique_ptr 不支持共享所有权，不能被复制，只能通过移动语义进行转移。这意味着 unique_ptr 在编译期间可以进行更严格的静态类型检查，避免了资源的重复释放和访问的问题。

问题：shared_ptr 主要用于解决共享所有权的问题。在某些情况下，我们需要多个指针共享一个资源，但需要确保资源在最后一个使用者释放后才被销毁。传统的指针使用中，手动管理共享资源的引用计数很容易出错，导致资源过早释放或内存泄漏。shared_ptr 使用引用计数的方式，跟踪资源的引用次数，确保在最后一个使用者释放后自动销毁资源。

区别：shared_ptr 允许多个指针共享同一个资源，每个指针都有一个关联的引用计数。当引用计数为零时，即没有任何指针指向资源时，资源会被销毁。shared_ptr 使用了动态分配的引用计数对象来跟踪引用计数，这增加了一定的开销，但提供了更灵活的资源共享方式。

问题：weak_ptr 主要用于解决弱引用和避免循环引用的问题。在某些情况下，我们需要引用一个资源，但不希望增加其引用计数，也不希望该引用影响资源的生命周期。此外，当存在循环引用（两个或多个对象相互持有 shared_ptr 导致引用计数无法为零）时，会导致内存泄漏，因为资源无法被正常释放。weak_ptr 可以以一种非拥有的方式引用共享资源，而不增加其引用计数，并且可以用于检测资源是否已被销毁。

区别：weak_ptr 是一种不拥有资源的指针，它只是提供对 shared_ptr 管理的资源的访问权。weak_ptr 不参与引用计数，也不会影响资源的生命周期。可以通过 weak_ptr 的 lock()成员函数获得一个 shared_ptr，如果资源还存在，则获得一个有效的 shared_ptr；如果资源已经被销毁，则获得一个空的 shared_ptr。

总结起来，unique_ptr 用于独占资源的所有权，shared_ptr 用于共享资源的所有权，而 weak_ptr 则用于解决弱引用和循环引用的问题。这三种智能指针在资源管理方式、所有权和性能方面有所不同，根据具体的需求选择合适的智能指针可以提高代码的可靠性和可维护性。

53. 假设有一个指针，如何做到多次使用，一次释放？

在 C++中，如果你希望在多次使用后只释放指针一次，你可以使用 std::shared_ptr 来管理资源的所有权。std::shared_ptr 是一种智能指针，它使用引用计数来跟踪资源的引用次数，当最后一个 shared_ptr 离开作用域时，资源会被自动释放。

54.分别写出在 if 语句的条件中，如何判断 bool a, int a, float a, char *a 是否为“零”。

bool 类型

```
bool a = false;
if (!a) {
    // a 为零的处理逻辑
}
```

int 类型

```
int a = 0;
if (a == 0) {
    // a 为零的处理逻辑
}
```

指针类型

```
char* a = nullptr;  // 或 char* a = NULL;
if (a == nullptr) {
    // a 为零的处理逻辑
}
```

在条件中判断浮点数是否为零时，使用==运算符进行比较可能存在精度问题。通常情况下，我

们建议使用比较运算符和一个适当的容差范围来判断浮点数是否近似于零，而不是直接比较是否等于零

```
float a = 0.0f;
float epsilon = 0.0001f;  // 容差范围

if (std::abs(a) < epsilon) {
    // a 近似于零的处理逻辑
}
```

55.以 int 变量为对象，设计一个类，给出所有成员函数的声明

类似格式的声明只需要写一个).

王道程序员

```
class IntWrapper {
public:
    IntWrapper();                       // 默认构造函数
    explicit IntWrapper(int value);     // 带参构造函数
    IntWrapper(const IntWrapper& other); // 拷贝构造函数
    ~IntWrapper();                      // 析构函数

    IntWrapper& operator=(const IntWrapper& other);  // 赋值运算符重载

    int getValue() const;               // 获取值
    void setValue(int value);           // 设置值

    IntWrapper operator+(const IntWrapper& other) const;  // 加法运算符重载
    IntWrapper operator-(const IntWrapper& other) const;  // 减法运算符重载
    IntWrapper operator*(const IntWrapper& other) const;  // 乘法运算符重载
    IntWrapper operator/(const IntWrapper& other) const;  // 除法运算符重载
    IntWrapper operator%(const IntWrapper& other) const;  // 取模运算符重载

    IntWrapper& operator+=(const IntWrapper& other);     // 加法赋值运算符重载
    IntWrapper& operator-=(const IntWrapper& other);     // 减法赋值运算符重载
    IntWrapper& operator*=(const IntWrapper& other);     // 乘法赋值运算符重载
    IntWrapper& operator/=(const IntWrapper& other);     // 除法赋值运算符重载
    IntWrapper& operator%=(const IntWrapper& other);     // 取模赋值运算符重载

    IntWrapper& operator++();           // 前置递增运算符重载
    IntWrapper operator++(int);         // 后置递增运算符重载
    IntWrapper& operator--();           // 前置递减运算符重载
    IntWrapper operator--(int);         // 后置递减运算符重载
```

56. 请解释 32 位/64 位系统具体指的是什么长度，对系统有何影响？

32 位和 64 位系统是指计算机处理器的寻址能力，也就是指处理器能够直接访问的内存地址的位数。在 32 位系统中，处理器的寻址能力为 32 位，即它可以直接寻址的内存地址范围为 2^32，约为 4GB。这意味着在 32 位系统中，每个内存地址都可以用 32 位的二进制数表示。而在 64 位系统中，处理器的寻址能力为 64 位，即它可以直接寻址的内存地址范围为 2^64，约为 18EB（1EB = 1 亿 GB）。这意味着在 64 位系统中，每个内存地址都可以用 64 位的二进制数表示。

系统的位数对计算机的性能和可用内存有一定影响，具体如下：

- 内存访问能力：64 位系统可以寻址更大的内存空间，因此可以支持更大的物理内存容量。这对于需要处理大量数据或运行内存密集型应用程序的任务来说是非常重要的。
- 整数运算能力：在处理大型整数、浮点数或执行复杂的数值计算时，64 位系统通常比 32 位系统更高效。64 位系统的寄存器和数据通路可以处理更大的整数和浮点数，提供更高的精度和性能。
- 应用程序兼容性：32 位程序无法直接在 64 位系统上运行，因为它们使用的是 32 位指令集和库。但是，大多数 64 位系统都提供了兼容性层，可以运行 32 位程序。相反，64 位程序可以在 64 位系统上运行，并且通常会利用 64 位体系结构的优势。
- 指针和数据大小：在 64 位系统上，指针的大小为 8 字节，而在 32 位系统上，指针的大小为 4 字节。这意味着在 64 位系统上，指针可以表示更大范围的内存地址。

57. 简述系统物理内存与虚拟内存之间的联系与区别。

联系：

1. 虚拟内存是建立在物理内存之上的一种机制。它允许操作系统将部分数据和代码存储在硬盘等辅助存储设备上，而不是完全依赖于物理内存。

2. 虚拟内存通过分页或分段等技术，将程序的地址空间划分为固定大小的块（页或段），这些块与物理内存中的页面或帧相对应。

3. 当程序访问虚拟内存中的数据时，操作系统会根据需要将相应的数据从辅助存储设备加载到物理内存中，以供程序使用。

区别：

1. 物理内存是计算机系统实际存在的硬件内存，由 RAM（随机存取存储器）组成。它是用于存储正在执行的程序和数据的实际内存资源。

2.虚拟内存是一种抽象的概念，是操作系统对物理内存的扩展和管理。它将程序的地址空间扩

展到比物理内存更大的范围，使得更大的程序能够运行，并提供了更好的内存管理和保护机制。

3.物理内存的访问速度通常比虚拟内存更快，因为它是直接与处理器连接的。虚拟内存的访问速度较慢，因为需要通过硬盘等辅助存储设备进行数据的交换。

4. 物理内存的容量是有限的，而虚拟内存的容量可以比物理内存大得多。虚拟内存通过将不常用的数据存储在磁盘上，可以支持更大的程序运行。

5. 物理内存是实际分配给每个进程的内存空间，而虚拟内存是每个进程视图的地址空间。不同的进程可以有不同的虚拟内存，但它们共享物理内存。

总的来说，物理内存是实际的硬件内存资源，而虚拟内存是操作系统对物理内存的抽象和管理方式。虚拟内存通过将不常用的数据存储在辅助存储设备上，扩展了系统的内存容量，并提供了更好的内存管理和保护机制。

58. 简述你熟悉的编译器的不同的优化级别，以及编译器优化一些基本的思想。

编译器的不同优化级别通常用于控制编译器在生成目标代码时应用的优化程度。

1. 无优化（-O0）：该级别关闭了编译器的所有优化。生成的代码与源代码的结构和逻辑一致，用于调试目的。

2.优化级别 1（-O1）：该级别启用一些基本的优化技术，如删除未使用的代码、常量折叠、简单的局部优化等。这些优化几乎不会增加编译时间，但可以提高代码的执行效率。

3.优化级别 2（-O2）：该级别在优化级别 1 的基础上应用更多的优化技术，如更复杂的局部优化、循环展开、内联函数等。这些优化可能会导致编译时间的增加，但可以进一步提高代码的执行效率。

4.优化级别 3（-O3）：该级别是最高级别的优化，应用了大多数可用的优化技术。它包括更激进的局部和全局优化、向量化、函数调用优化等。这些优化可能会显著增加编译时间，但可以在一定程度上提高代码的执行效率。

编译器优化的基本思想是在保持程序语义正确的前提下，通过各种优化技术改进代码的执行效率。

1. 常量传播和折叠：将程序中的常量表达式在编译时计算，并将其结果直接替换到代码中，以减少运行时的计算开销。

2. 代码消除：删除无法到达的代码或无效的代码，以减少程序的执行路径和运行时开销。

3.数据流分析和优化：通过分析程序中的数据流依赖关系，优化变量的使用方式，包括复制传

播、公共子表达式消除等，以减少不必要的计算和存储访问。

4. 循环优化：对循环进行变换和重组，包括循环展开、循环合并、循环分块等，以减少循环的开销和提高数据局部性。

5. 函数内联：将调用频率较高的函数直接展开到调用点，减少函数调用的开销。

6. 向量化：将标量计算转换为向量计算，利用 SIMD 指令集并行处理多个数据，提高代码的并行性和处理速度。

7. 代码重排和调度：通过重新排列和调度指令，优化指令的执行顺序，以减少数据相关性和提高指令级并行性。

8. 缓存优化：通过改进数据的访问模式和布局，减少缓存的不命中率，提高内存访问效率。

59. 函数 `bool less(float x, float y) { return *(int *)&x < *(int *)&y; }` 是否能正确计算 float 的大小关系，为何？对应了 c++的那种类型转换？

C++中的类型转换应该遵循类型系统的规则和语义。直接将 float 类型的指针转换为 int 类型的指针，然后通过对指针所指向的内存进行整数比较，违反了类型系统的规则。C++提供了一些合法的类型转换方式，如 static_cast、reinterpret_cast 和 dynamic_cast 等。可以使用这些类型转换操作符来进行类型转换，以保证类型的正确性和语义的一致性。在 C++中比较`float`或其他浮点数的大小关系应该使用<、>等比较运算符，或使用 std::less 等函数对象。这些方法会根据浮点数的规范进行正确的大小比较，考虑到浮点数的特殊规则，如 NaN（Not a Number）和处理非规范化的浮点数等。

60.有一个 char*p = {"abcdefg...z"} 的字符串，这个字符串强转成 int 型指针，再进行自增运算，求该值。

首先，将一个字符串常量赋值给 char* p 是不合法的，因为字符串常量是不可修改的（const），而 char*p`是一个非常量指针。正确的方式是使用 const char* p 来指向字符串常量。假设我们修正了代码为 const char* p = "abcdefghijklmnopqrstuvwxyz";，然后将该字符串指针强制转换为 int*类型，并进行自增运算，那么结果将依赖于系统的存储结构和指针的大小。在大多数系统上，指针的大小通常是 4 个字节或 8 个字节（32 位系统和 64 位系统）。当我们将一个指针强制转换为 int*类型时，指针的位模式会被解释为一个整数值。在这种情况下，p 指向的字符串的第一个字母是'a'。假设指针大小为 4 个字节，那么将 p 转换为 int*类型后，它将被解释为一个 32 位的整数值。自增运算符会增加该整数值的大小，而不是增加指针所指向的地址。需要注意的是，将指针类型强制转换为整数类型可能会导致未定义行为，因为这涉及到指针的解引用和对内存的访问。这种转换应该谨慎使用，并且在特定的情况下进行验

证和评估。

61. 谈一下模板 template。

Template 是 C++中的一项重要特性，它允许我们编写通用的代码，使得函数、类和数据结构可以在不同的类型上进行操作，从而实现代码的重用和泛化。模板提供了参数化类型的能力，可以根据实际需要在编译时生成具体的代码。C++中的模板主要有：函数模板和类模板。使用模板的主要优势是代码的泛化和重用。通过参数化类型，模板可以生成适用于不同类型的代码，避免了重复编写相似的代码。模板还提供了类型安全性，因为编译器可以在编译时进行类型检查和错误检查。此外，C++模板还支持模板特化和模板元编程等高级特性。模板特化允许我们为特定的类型提供特定的实现，而模板元编程则允许我们在编译时执行一些计算和逻辑，生成更复杂的代码。模板的代码在编译时进行实例化，因此会增加编译时间和生成的可执行代码的大小。此外，模板的错误信息可能较难理解，因为它们通常在实例化时才会报告错误。因此，在使用模板时，需要注意编写清晰和可读性高的代码，并进行适当的测试和类型检查。

62. 定义函数时，缺省函数的返回值类型，则函数的返回值类型为：

A float

B int

C char

D void

在 C 语言中，如果在函数定义时没有显式指定返回值类型，并且没有使用 C99 之后的函数原型，则函数的返回值类型被默认为 int。因此，答案是选项 B：int。

63. 要实现动态连编，必须使用（D）调用虚函数？

A 类名

B 派生类指针

C 对象名

D 基类指针

64. 面向过程和面相对象的区别？

C++既支持面向过程编程，也支持面向对象编程。下面是它们之间的一些区别：

1) 抽象单位：

- 面向过程：以过程或函数为基本的抽象单位，将问题划分为一系列的步骤和函数调用。
- 面向对象：以对象为基本的抽象单位，将问题划分为一组相互作用的对象，每个对象都有

自己的状态和行为。

2) 数据和行为的封装：

- 面向过程：数据和函数是分离的，函数对数据进行处理。

- 面向对象：数据和函数被封装在对象中，对象通过方法调用来处理自身的数据。

3) 继承和多态性：

- 面向过程：不支持继承和多态性。

-面向对象：支持继承和多态性。继承允许一个类继承另一个类的属性和方法，并添加或修改它们。多态性允许使用基类的指针或引用来调用派生类的方法。

4) 代码重用性：

- 面向过程：通过函数调用来实现代码的重用。

- 面向对象：通过继承和组合来实现代码的重用。继承允许子类重用父类的代码，组合允许一个对象包含其他对象作为其成员。

5) 设计思想：

- 面向过程：将问题分解为一系列的步骤和函数调用，强调过程和算法。

- 面向对象：将问题分解为一组相互作用的对象，强调对象之间的交互和消息传递。

需要注意的是，C++既支持面向过程编程，也支持面向对象编程，并且可以在同一个程序中结合使用这两种编程风格。这使得开发人员可以根据问题的性质和需求选择合适的编程范式，以实现代码的灵活性和可维护性。

65.面向对象的三大特性有哪些？

封装、继承、多态

66. 什么是多态？多态分为哪几种，多态的应用场景有哪些？

在 C++中，多态是面向对象编程的一个重要概念，指的是通过基类的指针或引用来调用派生类的方法。多态性允许不同的对象对同一消息做出不同的响应，提供了灵活性和可扩展性。

C++中的多态分为两种形式：

- 静态多态性：静态多态性是通过函数重载和运算符重载来实现的。在编译时，根据函数或运算符的参数类型和个数来确定调用哪个函数或运算符的版本。这种多态性在编译时就能确定，并且没有运行时的开销。

- 动态多态性：动态多态性是通过继承和虚函数来实现的。在运行时，根据对象的实际类型来确定调用哪个函数的版本。这种多态性在运行时才能确定，需要使用虚函数表来实现，因此会有一定的运行时开销。

多态的应用场景包括但不限于以下几个方面：

1) 继承和多态：通过基类和派生类之间的继承关系，可以使用基类的指针或引用来处理一组不同类型的对象，提高代码的灵活性和可维护性。
2) 虚函数和动态多态性：通过将基类的成员函数声明为虚函数，可以实现动态绑定，使得在运行时根据对象的实际类型调用正确的函数版本。
3) 抽象类和纯虚函数：抽象类是包含纯虚函数的类，它不能被实例化，只能作为基类使用。通过纯虚函数，可以定义接口和规范派生类必须实现的行为。
4) 虚析构函数：当基类指针指向派生类对象并进行删除时，如果基类的析构函数不是虚函数，可能会导致派生类对象的析构函数不被调用，造成资源泄漏。通过将基类的析构函数声明为虚函数，可以确保派生类的析构函数被正确调用。

67. 空类里有什么函数？

空类指的是没有任何成员变量和成员函数的类。在 C++中，空类不会自动拥有任何函数。然而，对于空类，编译器会自动生成默认的构造函数（无参构造函数）、拷贝构造函数、拷贝赋值运算符和析构函数。这些函数被称为合成函数。合成函数的实现是隐式的，它们并不占用空类的实例的大小。

68. A 继承 B、C 两个空类，对 A 进行强转成 B、C，地址空间有什么变化呢？

当类 A 继承自空类 B 和 C 时，对 A 进行从 B 到 A 和从 C 到 A 的强制类型转换并不会改变地址空间。由于空类没有成员变量和成员函数，其大小为 0 字节，因此类 A 的对象在内存中的布局只包含继承关系的相关信息，并不包含实际的成员。因此，强制类型转换后的地址仍然指向 A 对象的起始位置

69. public/priavate 继承的关系以及应用场景？

公有继承：通过公有继承，派生类（子类）可以访问基类（父类）的公有成员和保护成员，但不能直接访问基类的私有成员。公有继承可以实现代码重用和派生类与基类的接口一致性。公有继承还允许将派生类对象赋值给基类对象的指针或引用，实现多态性。

私有继承：通过私有继承，派生类可以访问基类的公有成员和保护成员，但不能直接访问基类的私有成员。私有继承将基类的接口隐藏在派生类中，可以用于实现继承实现的细节隐藏和实现继承接口的重定义。私有继承还限制了将派生类对象赋值给基类对象的指针或引用，因为派生类对象对外部是不可见的。

应用场景：

公有继承适用于"是一个"（is-a）的关系，派生类是基类的一种特殊类型。例如，`class Dog : public Animal`，其中 Dog 是 Animal 的一种特殊类型。

私有继承适用于"实现了一个"（implemented-in-terms-of）的关系，派生类通过继承基类的实现来实现自己的功能。例如，class Stack : private std::vector，其中 Stack 实现了一个栈的功能，但是使用了 vector 的实现细节，但对外部隐藏了 vector 的接口。

需要注意的是，除了公有继承和私有继承之外，C++还支持另一种继承关系，即保护继承（protected inheritance）。保护继承的特性介于公有继承和私有继承之间，派生类可以访问基类的保护成员，但不能直接访问基类的公有成员和私有成员。保护继承在实际应用中较少使用，通常在特定情况下才会使用。

70.C++的多态如何实现

继承 + 虚函数

71. 基类和派生类的构造函数和析构函数的执行顺序？

构造函数的执行顺序是从顶层基类开始，逐级向下执行，每个类的构造函数负责初始化自己的成员变量和执行自己的构造逻辑。

析构函数的执行顺序与构造函数相反，从派生类开始，逐级向上执行，每个类的析构函数负责执行自己的清理逻辑。

72. 谈谈深拷贝和浅拷贝，以及如何实现？

拷贝是指将一个对象的值复制到另一个对象，包括对象的成员变量。在浅拷贝中，如果对象中包含指针类型的成员变量，那么只会复制指针的值，而不会复制指针所指向的内容。这意味着原对象和拷贝对象会共享同一块内存，对其中一个对象的修改会影响另一个对象。

深拷贝是指创建一个新的对象，并将原对象的值复制到新对象中，包括对象的成员变量。在深拷贝中，如果对象中包含指针类型的成员变量，会为新对象分配一块新的内存，并将指针所指

向的内容复制到新的内存空间中。这样，原对象和拷贝对象拥有独立的内存空间，修改其中一个对象不会影响另一个对象。实现深拷贝通常需要自定义拷贝构造函数和拷贝赋值运算符。在这些函数中，需要对指针成员变量进行内存分配，并将原对象指针所指向的内容复制到新的内存空间中

73. string 的赋值操作是深拷贝还是浅拷贝？

深拷贝

在 C++中，std::string 的赋值操作是深拷贝（deep copy）。

当你对一个 std::string 对象进行赋值操作时，例如：

```
std::string str1 = "Hello";
std::string str2;
str2 = str1;
```

这里的赋值操作（str2 = str1;）会导致 str2 被赋予 str1 的内容的一个全新副本。这意味着 str2 会分配足够的内存来存储 str1 中的所有字符，并将这些字符复制到其新分配的内存中。因此，str1 和 str2 在内存中存储的是不同的字符数组，尽管它们的内容可能相同。

深拷贝确保了赋值操作后，两个 std::string 对象是完全独立的，对其中一个对象的修改不会影响到另一个对象。

相反，浅拷贝（shallow copy）仅复制对象的指针或引用，而不复制所指向的数据本身。这会导致两个对象共享同一块数据，一个对象对数据的修改会影响到另一个对象。幸运的是，std::string 的设计避免了这种情况，确保了赋值操作的正确性。

74. 什么时候重载赋值运算符与复制拷贝函数？

重载赋值运算符（operator=）：当一个类需要在对象赋值时执行特定的操作，而不仅仅是简单地复制成员变量的值时，可以重载赋值运算符。这样可以确保赋值操作符按照类的逻辑进行操作，而不是简单地进行浅拷贝。常见的情况包括动态内存分配、资源管理和对象状态的处理。重载赋值运算符允许自定义赋值行为。

复制构造函数：复制构造函数用于创建一个新对象，并将其初始化为已有对象的副本。当对象

被传递给函数或作为函数返回值时，复制构造函数会被调用。复制构造函数的目的是创建一个新对象，并确保该对象与原始对象具有相同的值。如果没有定义复制构造函数，编译器会提供一个默认的复制构造函数，它会按照成员变量的逐个复制来执行浅拷贝。

75. 什么地方需要用到拷贝构造函数？

上课讲的拷贝构造函数的三种调用时机即可。

1) 用一个已经存在的对象初始化另一个新对象
2) 当形参是对象，形参与实参进行结合时。
3) 当函数的返回值是对象，执行 return 语句时。

76. virtual()=0 是什么意思？

纯虚函数

77. 虚函数和虚继承是怎么实现？

1) 虚函数的实现

虚函数主要用于实现 C++中的多态性，即同一接口，不同实现。通过虚函数，我们可以用父类指针或引用来调用子类中的函数。

虚函数的实现主要依赖于两个关键概念：虚函数表和虚函数表的指针。

- 虚函数表（Virtual Table，简称 vtbl）：每个包含虚函数的类都会生成一个虚函数表。这个表是一个地址表，存储了类中所有虚函数的地址。
- 虚函数表的指针（Virtual Table Pointer，简称 vptr）：每个包含虚函数的类的对象都会包含一个指向其所属类的虚函数表的指针。这个指针通常放在对象内存的最前面。

当通过父类指针或引用来调用虚函数时，程序会首先通过 vptr 找到虚函数表，然后查找并调用对应的虚函数。这样，即使指针或引用实际上指向的是子类对象，也可以正确地调用子类的虚函数实现。

2) 虚继承的实现

虚继承主要用于解决 C++多重继承中的二义性和存储空间浪费问题。当一个类从多个路径继承同一个基类时，如果没有使用虚继承，那么在派生类中就会存在多个基类的拷贝，这既浪费了存储空间，又可能导致二义性。

虚继承的实现主要依赖于两个关键概念：虚基类指针和虚基类表。

- 虚基类指针：在派生类中，编译器会自动加入一个虚基类指针。这个指针指向虚基类表，占用了一定的存储空间（通常为 4 字节）。
- 虚基类表：虚基类表记录了派生类与其虚基类之间的偏移量。通过这个表，我们可以找到派生类中继承的虚基类的实际位置。

使用虚继承时，无论派生类从多少路径继承同一个基类，基类在派生类中只会有一个拷贝。当进行多重继承时，虚基类指针和虚基类表共同确保可以正确地找到并访问基类的成员。

78. 如果我有一块地址空间，我怎么在这个地址空间内调用构造函数？

分配内存、调用构造函数、构造函数执行

或者使用定位 new 表达式来完成。

79.sizeof(A) 是多少？

sizeof(A)的结果可能因编译器、编译选项和平台而异。它通常是编译器在满足对齐要求的情况下，根据类 A 的成员变量大小进行计算得出的，

```
class A {
 int a;
 short b;
 double c;
 virtual void fun() {}
 static int d;
};
```

二、STL

1. STL 包括哪些内容？

容器、算法、迭代器、函数对象、配置器、适配器

2. vector 底层实现

vector 的底层实现通常是基于动态数组或缓冲区。它使用连续的内存块存储元素，并支持随机

访问。当元素数量超过当前内存容量时，vector 会重新分配更大的内存块，并将原有元素复制到新的内存中。这种方式保证了 vector 在尾部添加和访问元素的高效性，但在插入或删除元素时可能需要进行内存的重新分配和数据复制。

3. vector 和 deque 的区别？

vector 和 deque 的区别在于内部存储结构和性能特征。vector 是基于动态数组的连续内存块，支持高效的随机访问，但在中间插入/删除元素时效率较低。deque 是双端队列，内部由多个分块的连续内存组成，支持高效的两端插入/删除操作，但对随机访问的性能较差。

4. vector 和 list 的本质区别？

vector 和 list 的本质区别在于底层数据结构和性能特征。vector 使用动态数组实现，支持高效的随机访问，但在插入/删除元素时可能需要进行内存的重新分配和数据复制。list 使用双向链表实现，支持高效的插入/删除操作，但对于随机访问的性能较差。

5. vector、list 在添加删除的效率方面有什么不同？

在添加和删除元素方面，vector 和 list 有不同的效率特点。对于 vector，尾部添加元素和访问元素是高效的，但在中间插入/删除元素时可能涉及内存重新分配和数据复制，效率较低。对于 list，插入和删除元素的效率很高，但在访问元素时需要遍历链表，效率较低。

6. 释放 vector 的内存的处理方式

vector 的内存释放是由其自身的析构函数负责处理。当 vector 对象被销毁时，其析构函数会自动释放内存。如果需要提前释放 vector 的内存，可以使用 vector<T>().swap(v)的方式，其中 v 为要释放内存的 vector 对象。

7. vector 迭代器失效的情况有哪些？

vector 迭代器失效的情况包括：

- 在插入或删除元素时，导致迭代器指向的元素或元素之后的位置失效。
- 在重新分配内存时，导致所有迭代器失效。

8. map 和 unordermap 的区别

map 是基于红黑树实现的有序关联容器，按照键排序。unordered_map 是基于哈希表实现的无序关联容器，不保持元素的顺序。主要区别在于内部数据结构和迭代器遍历的顺序。

9. stl 中的 unordered_map 和 unordered_set 有什么区别呢？

unordered_map 和 unordered_set 是基于哈希表实现的无序关联容器。区别在于 unordered_map 存储键值对，而 unordered_set 仅存储键。另外，unordered_map 和 unordered_set 的迭代器遍历顺序是根据哈希桶的顺序，而不是键的哈希值的顺序。

10. 自己实现 unordered_map 的话，你会考虑到什么问题呢？

自己实现 unordered_map 时需要考虑以下问题：

- 哈希函数的设计：选择合适的哈希函数以确保键的均匀分布。
- 哈希冲突的解决：处理哈希冲突的方法，如链地址法、开放寻址法等。
- 动态扩容：当元素数量增加时，需要动态扩展哈希表的大小，并重新哈希所有元素。
- 内存管理：包括内存分配和释放的策略。
- 迭代器的实现：实现迭代器以支持遍历和访问容器中的元素。

11. clear 和 erase 的区别？

clear 函数用于清空容器中的所有元素，但不释放容器的内存空间。erase 函数用于从容器中删除指定位置或指定范围的元素，并返回指向下一个有效元素的迭代器。erase 函数可以单个或批量删除元素，并可以选择

12. 迭代器失效问题。

迭代器失效是指迭代器在特定操作后不再指向有效的元素或位置。常见的迭代器失效情况包括：

- 在添加或删除元素时，导致迭代器失效。
- 在重新分配内存时，导致所有迭代器失效。
- 在对容器进行排序操作后，迭代器可能会失效。

13. swap 函数的作用？

swap 函数用于交换两个对象的内容。对于容器而言，swap 函数可以高效地交换两个容器的内

容，而不涉及元素的复制或移动操作。这在需要交换大型容器时可以带来性能上的优势。

14. 简述一下 STL 容器相关知识、特性等？

STL 容器具有以下一些共同的特性：

- 提供不同的数据结构和操作方式，以满足不同的需求。
- 通过迭代器提供统一的访问接口，支持遍历和操作容器中的元素。
- 部分容器支持动态内存管理，可以根据需要自动分配和释放内存。
- 提供丰富的成员函数和算法，用于对容器进行操作和处理。
- 具有不同的性能特征和适用场景，可以根据具体需求选择合适的容器。

15. stl 当中 vector、list、map 在内存中的数据结构有什么区别？

vector 在内存中的数据结构是连续的动态数组，通过指针访问元素；list 在内存中的数据结构是双向链表，每个节点包含元素和指向前后节点的指针；map 在内存中的数据结构是红黑树，根据键进行排序和查找。

16. erase 的返回值

> erase 函数的返回值是指向删除元素之后的位置的迭代器。如果删除的是最后一个元素，则返回 end()迭代器。如果删除的是多个元素，则返回删除范围的末尾位置的迭代器。如果删除的是单个元素，则返回删除元素的下一个位置的迭代器。

三、Linux

1. 说说 Linux 中常用的命令？

以下是一些在 Linux 系统中常用的命令：

ls: 列出目录内容。

cd: 更改当前工作目录。

pwd: 显示当前工作目录的路径。

mkdir: 创建新目录。

rm: 删除文件或目录。

cp: 复制文件或目录。

mv: 移动文件或目录。

touch: 创建空文件或更新文件的访问和修改时间戳。

cat: 查看文件内容。

less/more: 逐页查看文件内容。

grep: 在文件中搜索指定的字符串。

find: 在文件系统中搜索文件。

chmod: 更改文件权限。

chown: 更改文件所有者。

sudo: 以超级用户权限执行命令。

su: 切换用户身份。

ps: 显示当前运行的进程。

kill: 终止进程。

top/htop: 实时显示系统资源使用情况。

df: 显示文件系统磁盘空间使用情况。

du: 显示文件或目录的磁盘使用情况。

tar: 打包和解压文件。

ssh: 远程登录到其他计算机。

scp: 在本地主机和远程主机之间复制文件。

ping: 测试主机之间的连通性。

ifconfig/ip: 显示和配置网络接口信息。

date: 显示或设置系统日期和时间。

2. 创建软连接的命令是什么？

在 Linux 系统中，创建软链接的命令是 ln，使用方式如下：

```
ln -s  [target]  [link_name]
```

其中：

[target] 是要创建软链接指向的目标文件或目录的路径。

[link_name] 是新建的软链接的名称或路径。

例如，如果要创建一个指向 /usr/local/bin/example 的软链接，并将它命名为 example_link，可以执行以下命令：

```
ln -s /usr/local/bin/example example_link
```

这将在当前工作目录创建一个名为 example_link 的软链接，指向 /usr/local/bin/example。

3. /proc 文件夹下放的是什么？

/proc 文件夹是一个特殊的虚拟文件系统，它在 Linux 系统中用于提供关于运行中进程和系统内核状态的信息。在 /proc 文件夹下存放着一系列以数字命名的目录和一些特殊文件，这些目录和文件代表系统中运行的每个进程以及一些系统状态信息。

具体来说，/proc 中包含的信息包括但不限于以下内容：

进程信息：每个运行中的进程都有一个对应的以数字命名的目录，例如 /proc/1234，其中 1234 是进程的 PID（进程标识符）。在这个目录下可以找到关于该进程的各种信息，如命令行参数、环境变量、文件描述符、进程状态等。

系统信息：一些特殊文件和目录包含了关于系统硬件和内核状态的信息，如 /proc/cpuinfo 包含了 CPU 的信息，/proc/meminfo 包含了内存的信息，/proc/version 包含了内核版本信息等。

其他信息：还有一些其他的文件和目录，提供了一些与进程和系统相关的杂项信息，如 /proc/cmdline 包含了内核启动参数，/proc/sys 包含了一些系统参数和配置等。

总的来说，/proc 文件夹是一个非常有用的工具，可以用于获取系统状态信息、监控进程活动以及进行系统调优和诊断。

4. Linux 下有哪些文件类型？

在 Linux 系统中，文件类型可以分为以下几种：

普通文件（Regular File）：这是最常见的文件类型，包含了文本文件、二进制文件、图像文件等。普通文件可以通过文本编辑器或其他应用程序打开和编辑。

目录（Directory）：目录是一种特殊的文件类型，用于组织和存储其他文件和目录。它包含了其他文件和目录的列表。

符号链接（Symbolic Link）：也称为软链接，它是一个特殊类型的文件，包含了指向另一个文件或目录的路径。软链接类似于 Windows 中的快捷方式，可以跨文件系统指向不同位置的文件或目录。

设备文件（Device File）：设备文件用于与系统中的硬件设备或其他特殊设备进行通信。它们分为字符设备文件和块设备文件两种类型。字符设备文件用于顺序访问设备，如键盘、鼠标

等；块设备文件用于随机访问设备，如硬盘、闪存驱动器等。

管道（Named Pipe）：管道是一种用于进程间通信的特殊文件类型，它允许一个进程将输出发送到另一个进程的输入。

套接字（Socket）：套接字也是一种用于进程间通信的特殊文件类型，主要用于网络通信和本地进程间通信。套接字可以在不同计算机之间进行通信，也可以在同一台计算机的不同进程之间进行通信。

这些是在 Linux 系统中常见的文件类型，每种类型都有其特定的用途和特性。

5. Linux 查看内存、磁盘、端口、进程、线程命令有哪些？

在 Linux 中，可以使用以下命令来查看内存、磁盘、端口、进程和线程：查看内存：
free: 显示系统的内存使用情况，包括空闲内存、已用内存等。
top: 实时显示系统的资源使用情况，包括内存、CPU 等。
查看磁盘：
df: 显示文件系统的磁盘空间使用情况，包括磁盘总容量、已用空间、可用空间等。
du: 显示目录或文件的磁盘使用情况，包括该目录下所有文件的磁盘占用情况。
查看端口：
netstat: 显示网络状态信息，包括已经建立的连接、监听的端口等。
ss: 显示套接字信息，包括已经建立的连接、监听的端口等。
lsof: 显示打开的文件和进程信息，可以用来查看网络端口对应的进程。
查看进程：
ps: 显示当前运行的进程列表，包括进程的 PID、状态、CPU 使用情况等。
top: 实时显示系统的进程列表和资源使用情况，包括 CPU、内存等。
pgrep: 根据进程名查找进程的 PID。
查看线程：
ps -eLf: 显示所有线程的详细信息。
top: 在顶部的任务列表中显示线程信息。
这些命令可以帮助您监视和管理系统资源、进程和网络连接。

6. 是否在 linux 系统下用过 gdb 或者别的调试工具，对于 gdb 来说，用过哪些功能？

gdb 是 GNU Debugger 的缩写，是一个功能强大的调试工具，用于调试 C、C++ 等程序。以下是 gdb 的基本使用方法：
启动 gdb：
在终端中输入以下命令启动 gdb：

```
gdb [可执行文件路径]
```

如果程序需要参数，可以在启动 gdb 时传递参数。
设置断点：
在需要设置断点的代码行前面输入 break 命令：

```
break [行号]
```

或者可以设置在某个函数入口处：

```
break [函数名]
```

运行程序：
输入 run 命令运行程序，或者使用 r 的缩写：

```
run
```

执行程序：
输入 continue 命令继续执行程序，直到遇到下一个断点：

```
continue
```

单步执行：
使用 step 命令单步执行程序，进入函数内部：

```
step
```

下一步执行：
使用 next 命令执行下一行代码，不进入函数内部：

```
next
```

查看变量值：
使用 print 命令查看变量的值：

```
print [变量名]
```

查看堆栈信息：
使用 backtrace 命令查看当前的函数调用堆栈：

```
backtrace
```

退出 gdb：
输入 quit 命令退出 gdb：

```
quit
```

这些是 gdb 的一些基本用法，通过这些命令可以进行程序的调试、查看变量值、查看堆栈信息等操作。gdb 还支持更多的功能，如条件断点、查看内存内容等，可以通过查看 gdb 的文档或者使用 help 命令查看更多信息。

7. gdb 用法如果堆栈的信息不准，怎么办（不能运行，不能修改代码），可能是哪里出了问题

如果在使用 gdb 进行调试时，发现堆栈信息不准确，可能有以下几种可能性：

符号表丢失或不准确： 如果程序没有以调试模式编译，或者符号表被剥离了，gdb 可能无法正确解析函数名称和行号，从而导致堆栈信息不准确。这种情况下，可以尝试重新编译程序时启用调试选项，以确保生成正确的符号表。

优化级别过高： 在一些高级优化级别下编译的程序，可能会对代码进行优化，导致堆栈信息不准确。尝试使用更低的优化级别重新编译程序，以减少优化对堆栈信息的影响。

程序崩溃导致堆栈混乱： 如果程序发生了崩溃，堆栈信息可能会出现混乱。在这种情况下，gdb 可能无法正确解析堆栈信息。可以尝试在程序崩溃后立即使用 gdb 调试，以尽可能保留堆栈信息。

调试信息损坏： 如果程序或调试信息文件损坏，gdb 可能无法正确解析调试信息，从而导致堆栈信息不准确。在这种情况下，需要重新生成正确的调试信息文件，并确保它们与程序匹配。

如果无法修改代码或重新编译程序，可以尝试通过分析程序的行为和堆栈信息，结合其他调试工具或技术来确定问题的根本原因。例如，可以尝试使用其他调试工具进行检查，或者通过日志文件、核心转储文件等其他信息来了解程序的状态和运行情况。

8. 如果某个模块运行过程中突然崩溃，但是崩溃的几率不大，如何定位并解决这个问题？

定位和解决一个在某个模块运行过程中偶发性崩溃的问题可能是一个挑战，但是可以采取一些方法来尽可能地排除可能性并找到问题的根本原因：

记录日志信息： 在程序中添加适当的日志记录功能，包括在关键位置记录状态、变量值、函数调用等信息。当程序崩溃时，这些日志信息可以帮助您理解程序在崩溃之前的运行状态。

核心转储文件： 如果程序崩溃时生成了核心转储文件（core dump），可以使用 gdb 或其他调试工具分析核心转储文件，以了解程序崩溃时的堆栈信息、变量值等。这可以帮助您确定程序崩溃的位置和原因。

重现问题： 尝试重现问题，找出触发问题的特定条件或操作。可能需要多次尝试才能成功重现问题，但一旦找到了触发问题的方法，就可以更容易地调试和解决问题。

代码审查： 仔细审查模块的代码，查找可能导致崩溃的潜在问题，如内存泄漏、空指针引用、越界访问等。使用静态代码分析工具可以帮助发现一些常见的代码问题。

加强错误处理： 确保程序具有良好的错误处理机制，能够处理意外情况并给出合适的错误提示。避免使用不稳定的库或函数，尽量减少程序出错的可能性。

内存检查工具： 使用内存检查工具（如 Valgrind）来检查程序的内存使用情况，查找内存泄漏、内存越界等问题。这些问题可能导致程序偶发性崩溃。

通过以上方法的组合，可以尽可能地定位和解决模块偶发性崩溃的问题

9. 如果是在一个循环内出现问题，使用 gdb 调试需要等待很长时间，应该怎么处理？

在循环内出现问题，并且使用 gdb 调试时需要等待很长时间的情况下，可以考虑以下几种方法来加快调试过程：

添加条件断点： 可以使用 break 命令设置条件断点，只在满足特定条件时触发断点。这样可以避免每次循环都停下来，只在特定条件下才停止程序执行。例如：

break [行号] if [条件]跳过循环体： 可以使用 skip 命令跳过当前循环的执行，直接进入下一次循环。这样可以快速跳过循环体，进入下一次迭代，而不需要等待循环体执行完毕。

jump [下一个循环的行号]精简调试流程： 可以在循环内部添加更多的日志输出，以便在不使用 gdb 调试时也能获取循环内部的状态信息。这样可以减少在 gdb 下的调试时间，加快定位问题的速度。

分析问题根源： 尽量减少每次调试循环的次数，尝试找到问题的根本原因。可以通过日志、

核心转储文件等方法获取更多信息，并尝试在离线环境下进行分析，减少 gdb 调试的次数。

优化程序逻辑： 如果可能，可以考虑优化循环内部的逻辑，尽量减少循环体内部的计算量，以及可能导致长时间等待的操作，从而加快调试过程。

通过以上方法的组合，可以尽量减少在循环内部调试时的等待时间，加快定位问题的速度。

10.内存泄漏怎么检查，怎么避免？

内存泄漏是指程序在动态分配内存后未能正确释放内存，导致程序持续占用内存而不释放，最终耗尽系统资源的情况。下面是检查和避免内存泄漏的一些方法：

检查内存泄漏：

使用内存检查工具： 像 Valgrind、AddressSanitizer 等工具可以帮助检测内存泄漏，它们会在程序执行时跟踪内存分配和释放情况，并检查是否有未释放的内存块。

检查代码： 仔细检查代码，确保每次动态分配内存后都有相应的释放操作。尤其要注意在循环或条件分支中的内存释放操作是否正确执行。

使用 RAII（资源获取即初始化）： 在 C++ 中，使用 RAII 管理资源是一种良好的做法。通过使用智能指针、容器等 RAII 类型，可以确保资源在不再需要时自动释放。

检查文件描述符、句柄等资源的释放： 除了内存，还要确保其他资源（如文件描述符、网络连接等）在不再使用时正确释放。

避免内存泄漏：

正确释放内存： 在使用 malloc、calloc、new 等动态分配内存的地方，一定要在不再需要时使用 free、delete 等函数进行释放。

使用智能指针： 在 C++ 中，尽量使用智能指针（如 std::unique_ptr、std::shared_ptr）来管理动态分配的内存，以避免手动释放内存时的错误。

避免循环引用： 如果使用了智能指针的共享所有权机制（如 std::shared_ptr），要注意避免循环引用，否则可能导致内存泄漏。

及时释放资源： 及时释放不再需要的资源，尤其是在长时间运行的服务程序中，要确保定期清理资源，避免资源积累导致的内存泄漏。

注意异常情况处理： 在发生异常时，要确保及时释放已经分配的资源，以防止异常情况导致的资源泄漏。

通过以上方法，可以有效地检查和避免内存泄漏问题，提高程序的稳定性和可靠性。

11.什么是 coredump 文件？怎么调试？

Core dump 文件（核心转储文件）是在程序异常终止（如段错误、非法操作等）时由操作系统生成的一种文件，用于保存程序异常终止时的内存状态信息。它包含了程序崩溃时的内存映像、堆栈信息、寄存器状态等，可以帮助开发人员分析程序崩溃的原因。

调试 core dump 文件通常使用调试工具，如 GDB (GNU Debugger)。以下是使用 GDB 调试 core dump 文件的基本步骤：

启动 GDB：

在终端中输入以下命令启动 GDB，并指定 core dump 文件和相关的可执行文件：

```
gdb [可执行文件路径] [core dump 文件路径]
```

分析堆栈信息：

使用 bt 命令查看堆栈信息，显示导致程序崩溃的函数调用堆栈：

```
bt
```

查看寄存器状态：
使用 info registers 命令查看寄存器的状态，了解程序崩溃时 CPU 寄存器的值：

```
info registers
```

查看变量值：
使用 print 命令查看特定变量的值，了解程序崩溃时变量的状态：

```
print [变量名]
```

分析内存映像：
使用 x 命令查看内存映像，了解程序崩溃时内存的状态：

```
x/[长度格式] [内存地址]
```

定位问题：
结合堆栈信息、寄存器状态、变量值等信息，分析程序崩溃的原因，并定位代码中可能存在的问题。
通过分析 core dump 文件，可以在程序崩溃后重新调试程序，帮助开发人员定位和解决程序崩溃的问题。

12.什么时候用静态库和什么时候用动态库，两者有何区别？

静态库和动态库都是用于组织和管理代码的库文件，但它们在使用场景和工作原理上有所不同。以下是静态库和动态库的区别以及何时使用它们的建议：静态库（Static Library）：
使用场景：
当程序的规模较小，依赖的库文件不多时，通常使用静态库。
需要确保程序在不同环境中运行时能够完全独立，不依赖于外部库文件。
工作原理：
静态库会被链接到程序的可执行文件中，编译后的可执行文件会包含静态库的代码和数据。
在编译时，链接器会将静态库的目标文件直接复制到可执行文件中。
优点：
执行速度较快，因为所有代码都被链接到了可执行文件中，无需在运行时动态加载。
静态库不依赖于系统环境和外部库文件，可以确保程序在不同环境中的可移植性。
缺点：
占用内存较大，因为静态库的代码会被复制到每个可执行文件中。
更新静态库需要重新编译程序，可能会增加开发和部署的成本。
动态库（Dynamic Library）：
使用场景：
当程序依赖的库文件较多，或者需要在多个程序之间共享代码时，通常使用动态库。
需要实现代码的动态加载和共享，以减少内存占用和提高程序的灵活性。
工作原理：
动态库不会被直接链接到程序中，而是在程序运行时动态加载到内存中。
在程序执行过程中，通过动态链接器将动态库加载到内存中，并解析和链接其中的符号。

优点：

减少内存占用，因为动态库在内存中只有一份拷贝，多个程序可以共享同一个动态库。

方便更新和维护，因为可以通过更新动态库文件来更新多个程序的代码。

缺点：

执行速度较慢，因为需要在运行时动态加载和链接库文件。

对系统环境和库文件的依赖较强，可能导致不同环境下的兼容性问题。

总结：

静态库适用于小型项目或需要确保程序独立性的场景，而动态库适用于大型项目或需要共享代码的场景。

静态库将代码直接链接到可执行文件中，而动态库在程序运行时动态加载到内存中。

静态库执行速度快，但占用内存较大；动态库内存占用小，但执行速度较慢。

13.零拷贝技术有哪些？

零拷贝（Zero-Copy）技术是一种优化技术，旨在最小化数据在计算机系统中的复制次数，从而提高数据传输的效率和性能。以下是几种常见的零拷贝技术：

文件描述符传递（File Descriptor Passing）：在 UNIX 系统中，可以使用 sendfile() 系统调用将文件内容从一个文件描述符传输到另一个文件描述符，而无需将数据在用户空间和内核空间之间复制。这样可以避免数据的复制，提高数据传输效率。

直接内存访问（Direct Memory Access，DMA）：DMA 允许设备（如网络适配器、磁盘控制器等）直接访问系统内存，而无需 CPU 的介入。这样可以避免数据在内核空间和用户空间之间的复制，提高数据传输效率。

内存映射（Memory Mapping）：内存映射技术允许将文件映射到进程的地址空间，从而使得文件内容可以直接通过内存访问而无需通过 read/write 等系统调用进行数据传输。这样可以避免数据在内核空间和用户空间之间的复制。

零复制网络栈（Zero-Copy Networking Stack）：在网络传输中，零拷贝技术可以通过在内核空间和用户空间之间共享数据缓冲区来避免数据的复制。例如，使用 sendfile() 或 splice() 系统调用在网络套接字和文件描述符之间进行数据传输，或者使用 DMA 在网络适配器和内存之间进行数据传输。

这些零拷贝技术可以在不同的场景下提高数据传输的效率和性能，减少 CPU 的负载，提高系统的整体性能。选择合适的零拷贝技术取决于具体的应用场景和需求。

14.mmap 的应用场景有哪些？

mmap()（内存映射）是一种将文件或其他对象映射到进程的地址空间的系统调用。它的应用场景非常广泛，包括但不限于以下几个方面：

文件 I/O： mmap() 可以用于文件 I/O 操作，将文件映射到进程的地址空间中，使得文件的内容可以直接通过内存访问而无需通过 read/write 等系统调用进行数据传输。这样可以提高文件读写的效率，尤其是对于大文件和频繁读写的文件。

内存共享： 多个进程可以使用 mmap() 将同一个文件映射到它们各自的地址空间中，实现内存共享。这样可以节省内存空间，减少数据的复制，提高进程间通信的效率。

匿名内存映射： mmap() 还可以用于创建匿名内存映射区域，即不与任何文件关联的内存映射。这种技术通常用于创建共享内存区域，实现进程间通信或共享数据。

高性能计算： mmap() 在高性能计算领域中也有广泛的应用，例如在大规模数据处理、数据分析、图像处理等场景中，可以通过内存映射技术实现高效的数据访问和处理。

数据库： 许多数据库系统（如 SQLite）使用 mmap() 来管理数据库文件，以提高读取和写入

数据的性能，并简化文件管理和缓存管理。

总的来说，mmap() 是一种强大的系统调用，可以用于各种不同的应用场景，提高系统的性能和效率，简化代码实现。

15.linux 文件系统读入文件的过程。

在 Linux 文件系统中，读取文件的过程可以分为以下几个步骤：

文件路径解析： 当应用程序请求打开一个文件时，内核首先需要对文件路径进行解析，以确定文件的位置。这涉及到从根目录开始的目录树遍历，找到指定文件的 inode。

inode 查找： 一旦文件路径解析完成，内核根据文件的路径找到相应的 inode（索引节点）。inode 包含了文件的元数据信息，如文件类型、大小、权限等。

权限检查： 在读取文件之前，内核会对请求的文件进行权限检查，以确保当前用户有权读取该文件。

页缓存查找： 内核会检查页缓存（Page Cache），这是一块用于缓存文件数据的内存区域。如果文件的数据已经在页缓存中存在，并且没有过期，那么内核会直接从页缓存中读取数据，避免了对磁盘的访问。

文件系统读取： 如果文件数据不在页缓存中或已过期，内核将发起对文件系统的读取请求。根据文件的 inode 信息，内核会根据文件系统的布局和数据结构访问磁盘上的数据块，将文件的内容读取到内存中。

页缓存更新： 一旦文件数据被读取到内存中，内核会将数据写入页缓存，以备将来的读取请求。这样可以提高对相同文件的后续读取性能。

内存拷贝： 最后，文件的数据被拷贝到应用程序的地址空间中，以供应用程序使用。

需要注意的是，文件系统的具体实现和性能会受到许多因素的影响，包括文件系统类型、文件系统参数、磁盘性能等。上述步骤提供了一般情况下读取文件的过程，但在具体情况下可能会有所不同。

进程和线程

16.线程和进程区别？

17.多进程和多线程的区别是什么？换句话说，什么时候该用多线程，什么时候该用多进程？

线程（Thread）和进程（Process）是操作系统中的两个重要概念，它们之间有以下区别：资源分配：

进程是操作系统进行资源分配和调度的基本单位，每个进程拥有独立的地址空间、文件描述符、堆栈等资源。

线程是进程的执行单元，多个线程可以共享同一个进程的地址空间和其他资源，包括堆、全局变量、打开的文件等。

调度和切换：

进程之间的切换需要保存和恢复完整的上下文信息，包括内存映像、寄存器状态、打开的文件等，切换成本较高。

线程之间的切换只需要保存和恢复部分上下文信息，如寄存器状态和栈指针等，切换成本较低。

并发性：

不同进程之间的并发执行是通过操作系统的时间片轮转调度实现的，进程之间的通信需要使用进程间通信机制（如管道、消息队列、共享内存等）。

同一个进程内的多个线程可以并发执行，线程之间可以直接共享同一进程的地址空间和其他资源，因此线程之间的通信成本较低，可以通过共享内存等机制进行通信。

资源开销：

创建和销毁进程的开销比较大，因为需要分配和释放独立的地址空间、文件描述符等资源。

创建和销毁线程的开销相对较小，因为线程共享了进程的地址空间和其他资源，只需要分配和释放线程的栈空间、线程控制块等少量资源。

总的来说，进程和线程都是并发执行的基本单位，但线程比进程更轻量级，具有更低的切换开销和更高的并发性，因此在线程间共享数据和协同工作方面更加灵活和高效。然而，线程的共享资源可能会增加编程的复杂性和线程安全的难度，需要开发人员注意并发控制和同步机制的设计。

18.中断和异常的区别

中断（Interrupt）和异常（Exception）是计算机系统中用于处理异常事件的两种机制，它们之间的区别如下：引发原因：

中断是由外部设备或硬件触发的异步事件，例如定时器中断、I/O 设备完成等。

异常是由程序执行过程中出现的非预期事件或错误触发的，例如除零错误、内存访问越界等。

处理方式：

中断是一种与程序执行异步并发的机制，当中断事件发生时，CPU 会立即暂停正在执行的程序，并跳转到相应的中断处理程序进行处理。

异常是在程序执行过程中出现的错误或异常情况，CPU 会立即停止当前指令的执行，并跳转到异常处理程序进行处理。

处理程序：

中断处理程序通常是由操作系统或设备驱动程序提供的，用于响应中断事件并执行相应的处理逻辑。

异常处理程序通常是由操作系统或运行时环境提供的，用于捕获和处理程序执行过程中的错误或异常情况。

优先级：

中断通常具有不同的优先级，并且可以被屏蔽或禁用，以便系统能够在多个中断事件发生时进行适当的处理。

异常通常是在程序执行过程中即时触发的，其优先级可能较高，并且不能被屏蔽或禁用。

触发时机：

中断是由硬件事件或外部设备触发的，并且可以在任何时间发生。

异常是由程序执行过程中的错误或异常情况触发的，通常发生在特定的程序指令执行时。

总的来说，中断和异常都是处理计算机系统中异常事件的重要机制，但它们的引发原因、处理方式、处理程序和触发时机等方面有所不同。理解和处理好中断和异常对于保证系统的稳定性和可靠性至关重要。

19.进程间通信方式有哪些呢？

进程间通信（Inter-Process Communication，IPC）是不同进程之间进行数据交换和通信的一种机制。常见的进程间通信方式包括以下几种：管道（Pipe）：

管道是一种半双工的通信方式，用于在父进程和子进程之间进行通信。它是一种单向通信机

制，父进程可以向管道中写入数据，子进程可以从管道中读取数据，反之亦然。

命名管道（Named Pipe）：

命名管道是一种特殊的管道，它允许不相关的进程之间进行通信。与普通管道不同，命名管道在文件系统中有一个路径名，并且可以在多个进程之间共享数据。

消息队列（Message Queue）：

消息队列是一种通过消息传递进行通信的方式，允许进程之间发送和接收消息。每个消息都有一个特定的类型和长度，接收进程可以选择性地接收特定类型的消息。

信号量（Semaphore）：

信号量是一种用于进程间同步和互斥的机制。它可以用来控制多个进程对共享资源的访问，防止竞态条件和数据不一致问题。

共享内存（Shared Memory）：

共享内存是一种让多个进程共享同一块内存空间的方式，使得它们可以直接读写共享内存区域，而无需进行数据复制。共享内存是一种高效的进程间通信方式，但需要额外的同步机制来保证数据的一致性和完整性。

套接字（Socket）：

套接字是一种通用的进程间通信机制，用于在网络上进行通信。它可以用于在同一台主机上的不同进程之间进行通信，也可以用于在不同主机上的进程之间进行通信。

信号（Signal）：

信号是一种异步通信机制，用于向进程发送各种事件通知。进程可以注册信号处理函数来处理收到的信号，以便进行相应的处理。

以上是常见的进程间通信方式，每种方式都有其适用的场景和特点，开发人员可以根据具体的需求选择合适的进程间通信方式。

20.线程间通信的方式？

线程间通信（Thread Communication）是多线程编程中的重要概念，用于实现不同线程之间的数据共享和同步。常见的线程间通信方式包括以下几种：

共享内存：

多个线程可以共享同一块内存区域，通过读写共享内存来进行通信。但需要注意线程之间的竞态条件和数据一致性问题，需要使用互斥锁、读写锁等同步机制来保护共享内存的访问。

互斥锁（Mutex）：

互斥锁是一种用于多线程编程中的同步机制，用于保护共享资源的访问，防止多个线程同时访问共享资源导致的数据竞争和数据不一致问题。线程在访问共享资源之前需要先获取互斥锁，访问完成后释放互斥锁。

条件变量（Condition Variable）：

条件变量是一种同步机制，用于在线程之间进行通信和同步。它通常与互斥锁一起使用，当共享资源的状态满足特定条件时，线程可以通过条件变量等待通知，当条件不满足时，线程可以通过条件变量等待在条件上。当条件满足时，其他线程可以通过条件变量发送信号来通知等待的线程。

信号量（Semaphore）：

信号量是一种计数器，用于多线程编程中的同步和互斥。它可以控制同时访问共享资源的线程数量，防止资源的过度使用和竞争。线程可以通过信号量进行等待和释放，当信号量计数大于零时，线程可以继续执行，否则线程将被阻塞等待。

屏障（Barrier）：

屏障是一种同步机制，用于在多线程编程中实现多个线程的同步点。当所有线程到达屏障点时，它们将被阻塞，直到所有线程都到达后才能继续执行。屏障可用于实现线程间的同步和协调。

消息队列（Message Queue）：
消息队列是一种线程间通信的方式，用于在多个线程之间发送和接收消息。每个线程都可以通过消息队列发送消息，其他线程可以从消息队列中接收消息，并根据消息的内容执行相应的操作。
以上是常见的线程间通信方式，每种方式都有其适用的场景和特点，开发人员可以根据具体的需求选择合适的线程间通信方式。

21.Linux 程序运行找不到动态库.so 文件的三种解决办法

当 Linux 程序运行时找不到动态链接库（.so 文件）时，可以采取以下三种解决办法：
设置 LD_LIBRARY_PATH 环境变量：
使用 LD_LIBRARY_PATH 环境变量可以指定动态链接库的搜索路径。通过设置 LD_LIBRARY_PATH 变量，告诉系统在指定路径中查找动态链接库文件。例如：
export LD_LIBRARY_PATH=/path/to/library:$LD_LIBRARY_PATH
修改 /etc/ld.so.conf 文件并运行 ldconfig 命令：
在 /etc/ld.so.conf 文件中添加动态链接库所在路径，然后运行 ldconfig 命令更新动态链接库缓存。这样系统就会在运行时搜索指定路径中的动态链接库。例如：
echo "/path/to/library">> /etc/ld.so.conf ldconfig
使用软链接或修改程序链接路径：
可以使用 ln 命令创建动态链接库的软链接，将库文件链接到系统默认的库路径下，或者修改程序的链接路径，将库文件链接到程序所在目录。这样程序在运行时就可以找到相应的动态链接库。例如：
ln-s /path/to/library/libexample.so /usr/lib/libexample.so
或者修改程序的链接路径：
export LD_LIBRARY_PATH=/path/to/library:$LD_LIBRARY_PATH
通过以上方法，可以解决程序在运行时找不到动态链接库的问题，确保程序顺利执行。选择哪种方法取决于具体情况和个人偏好。

22.Linux 进程同步的机制

信号量（Semaphore）：
消息队列（Message Queue）：
管道（Pipe）：

23.什么是同步、异步？什么是阻塞、非阻塞？

- 同步和异步是指程序的执行方式。同步指的是调用者发出一个请求，被调用者进行处理，处理完毕后返回结果给调用者，期间调用者会一直等待；而异步则是调用者发出请求后不等待，而是继续执行其他操作，被调用者在处理完毕后通知调用者或者通过回调函数来处理结果。
- 阻塞和非阻塞则是指调用者在等待结果时的状态。阻塞是指调用者在等待结果时会被挂起，不能执行其他操作；非阻塞则是指调用者在等待结果时仍然可以执行其他操作，不会被挂起。

24.进程的状态

- 新建（New）：进程被创建但尚未执行。
- 就绪（Ready）：进程已经准备好执行，但还未开始执行。
- 运行（Running）：进程正在执行。
- 阻塞（Blocked）：进程因等待某事件而暂停执行。
- 终止（Terminated）：进程执行完毕。

25.什么是孤儿进程？什么是僵尸进程？怎么避免僵尸进程？

- 孤儿进程是指父进程退出，而子进程还在运行的进程。解决方法是让子进程被 init 进程接管。
- 僵尸进程是指进程已经终止，但其父进程尚未对其进行善后处理，导致其占用系统资源。解决方法是父进程调用 wait 或 waitpid 函数来等待子进程的终止，并处理子进程的退出状态。

26.结束进程的方式有哪些？

- main 函数当中执行 return
- 调用 exit() 系统调用
- 调用 _exit() 系统调用
- 调用 abort() 系统调用
- 接收到信号并处理
- 进程内的主线程终止

27.什么是会话(session)？

会话是进程组的集合，是指一组进程组成的环境，通常是一个用户登录到系统后启动的一系列进程。

28.守护进程与后台进程的区别

- 守护进程是在后台运行的进程，通常在系统启动时启动，并在系统关闭时结束。它们通常不受用户控制。
- 后台进程是由用户启动的，但是可以在后台运行而不影响用户的当前操作。通常可以通过将进程放入后台或使用 nohup 命令来启动后台进程

29.写时拷贝

写时拷贝是一种内存管理技术，用于在多进程共享内存时节省内存开销。当多个进程共享同一块内存时，只有当其中一个进程试图修改这块内存时，系统才会进行实际的拷贝操作，以确保每个进程拥有自己的独立副本。

30.自旋锁

自旋锁是一种同步机制，用于保护共享资源，避免多个线程同时访问导致数据不一致。当一个线程尝试获取自旋锁时，如果锁已被其他线程持有，该线程不会被挂起，而是一直循环等待，直到获取到锁为止。

31.谈一下对多线程的理解，如生产者 - 消费者问题。

多线程是指在一个程序中同时运行多个线程，每个线程执行不同的任务。生产者 - 消费者问题是指一个线程（生产者）生成数据，而另一个线程（消费者）消费数据的场景。需要注意线程之间的同步和互斥，以避免数据竞争和不一致的问题。

32.什么是死锁？死锁产生的条件？怎么解决死锁问题？

死锁是指多个进程或线程因竞争资源而造成的相互等待的状态。死锁产生的条件包括互斥、占有且等待、不可抢占和循环等待。解决方法包括资源预分配、破坏环路、加锁顺序等。

33.信号量处理耗费多长时间，信号量同步会有什么问题

信号量处理的时间取决于系统的实现方式和具体的操作系统，在 Linux 中大约 0.1 微秒 ~1 微秒左右。在信号量同步过程中，可能会出现死锁、饥饿等问题，需要合理设计同步机制以避免这些问题。

34.登录 shell 进程是如何启动的？shell 是如何调用系统调用的？

登录 shell 进程是由系统初始化时启动的，通常是由 init 进程启动的。当用户登录时，系统会为用户分配一个 shell 进程，并执行相应的登录脚本。Shell 通过调用系统调用来与操作系统进行交互，包括创建进程 fork、启动程序 exec 等。

35.sleep()调用后进程有哪些过程，在 sleep()的过程中进程占用 CPU 了吗？

sleep() 调用会使进程进入睡眠状态，直到指定的时间到达或接收到信号才会唤醒。在睡眠过程中，进程会释放 CPU，不会占用 CPU 资源。

36.线程池有什么好处？

线程池可以提高线程的利用率和系统的性能，减少线程创建和销毁的开销，提高系统的响应速度和吞吐量。

37.讲一下线程池？

线程池是一种管理线程的机制，它维护一定数量的线程，用于处理任务队列中的任务。通过线程池，可以有效地重用线程、控制并发线程数量，以及管理线程的生命周期。

38.什么是线程安全？

线程安全是指在多线程环境中，对共享资源的访问操作不会导致数据不一致或者产生竞态条件的状态。

39.多线程间共享数据，用什么方式来保存它们的安全性

- 使用互斥锁（mutex）来保护共享资源，确保在同一时刻只有一个线程能够访问共享资源。
- 使用条件变量（condition variable）来实现线程间的同步和通信。
- 使用原子操作来确保对共享资源的操作是原子的，不会被中断

40.什么是线程安全函数，工作中如何保证线程安全

线程安全函数是指在多线程环境中可以安全地调用的函数。线程安全函数通常尽量使用局部变量，对于共享资源会内部使用锁或其他同步机制来保证线程安全。

41.可重入函数是什么意思，为什么一定是线程安全的

可重入函数是指可以被多个线程同时调用而不会产生不正确的结果或者数据竞争的函数。可重入函数通常是线程安全的，因为它们不会引用全局变量或者共享资源。

网络编程

42.简述七层模型和四层模式。

- 七层模型是指 OSI 参考模型，包括物理层、数据链路层、网络层、传输层、会话层、表示层和应用层，每一层都有特定的功能和责任。
- 四层模型是指 TCP/IP 参考模型，包括网络接口层、网络层、传输层和应用层，它对应了 OSI 模型中的数据链路层、网络层、传输层和应用层。

43.请描述一下从输入 URL 到显示页面的全过程。

- DNS 解析：将 URL 解析为 IP 地址。
- TCP 连接：通过建立 TCP 连接与服务器进行通信。
- HTTP 请求：发送 HTTP 请求给服务器。
- 服务器处理请求：服务器接收到请求后处理并返回响应。

- 浏览器渲染：浏览器接收到响应后解析并渲染页面。

44.简述一下 socket 编程的流程。

- 创建 socket：调用 socket 函数创建一个套接字。
- 绑定地址：将套接字绑定到一个地址上。
- 监听连接：如果是服务器端，调用 listen 函数监听连接请求。
- 接受连接：如果是服务器端，调用 accept 函数接受客户端的连接请求。
- 发送和接收数据：通过 send 和 recv 函数发送和接收数据。
- 关闭连接：使用 close 函数关闭连接。

45.write 阻塞的原因有哪些？

- 接收缓冲区满：当接收方的接收缓冲区已满时，写操作将被阻塞，直到接收方读取数据释放空间。
- 网络拥塞：当网络拥塞时，发送数据可能会被阻塞，直到网络状况改善。

46.多路复用：select, poll, epoll 的区别？epoll 的底层是如何实现的？

- select 和 poll 采用轮询的方式来处理多个文件描述符，效率较低。
- epoll 使用事件通知的方式，当文件描述符就绪时，内核会通知应用程序，效率较高。
- epoll 底层采用红黑树来管理文件描述符，通过事件注册和就绪列表来实现高效的 I/O 多路复用。

47.epoll 边沿触发具体实现方式

- epoll 边沿触发（EPOLLET）是一种事件触发模式，只有在状态发生变化时才会触发事件。
- 实现方式是通过标记文件描述符为非阻塞模式，并使用边沿触发模式，当文件描述符就绪时，内核会触发事件并通知应用程序。

48.LT 和 ET 的区别，应用场景？

- LT（Level Triggered）是水平触发模式，当文件描述符处于就绪状态时会触发事件，如果应用程序没有处理完数据，则下次仍会触发事件。
- ET（Edge Triggered）是边缘触发模式，只有当文件描述符状态发生变化时才会

触发事件，如果应用程序没有处理完数据，则不会触发事件。

- LT 适用于单线程处理复杂的事件处理逻辑，而 ET 适用于处理多线程下高性能的网络编程。

8. 说说同步、异步、阻塞、非阻塞。

重复

49.调用 send 函数发送数据不全怎么办？

- 可以使用循环发送，直到发送完所有数据。
- 可以检查 send 函数的返回值，判断实际发送的字节数，然后继续发送未发送的数据。

50.1G 的文件从 A 机器发送到 B 机器，怎么发？(写代码实现)

51.什么是 TCP 的"粘包"问题？怎么解决？

粘包问题是指发送方连续发送的数据到达接收方时被粘在一起，导致接收方无法正确解析。解决方法包括使用消息边界（如添加消息长度字段）、使用特定分隔符（如换行符）、定时发送（如每次发送固定大小的数据）等。

52.tcp 和 udp 的区别？

- TCP 是面向连接的可靠传输协议，提供了数据的可靠性和顺序性，适用于需要可靠传输的场景；而 UDP 是面向无连接的不可靠传输协议，不保证数据的可靠性和顺序性，适用于实时性要求高的场景。
- TCP 提供流式传输，数据没有边界；而 UDP 提供数据报传输，数据有边界。

53.tcp 三次握手建立连接的过程？三次握手过程通信双方各自的状态？

- 第一次握手：客户端向服务器发送 SYN 报文，并进入 SYN_SENT 状态。
- 第二次握手：服务器收到 SYN 报文后，返回 SYN+ACK 报文给客户端，并进入 SYN_RECV 状态。
- 第三次握手：客户端收到 SYN+ACK 报文后，发送 ACK 报文给服务器，双方进入 ESTABLISHED 状态。

54.tcp 四次挥手的过程？四次挥手过程中通信双方各自的状态？

- 第一次挥手：客户端发送 FIN 报文给服务器，并进入 FIN_WAIT_1 状态。
- 第二次挥手：服务器收到 FIN 报文后，发送 ACK 报文给客户端，进入 CLOSE_WAIT 状态。
- 第三次挥手：服务器发送 FIN 报文给客户端，进入 LAST_ACK 状态。
- 第四次挥手：客户端收到 FIN 报文后，发送 ACK 报文给服务器，双方进入 CLOSED 状态。

55.简述一下 tcp 的超时机制。

TCP 的超时机制是为了检测连接是否断开，以及重传丢失的数据。超时机制包括重传超时（RTO）、持续超时（Retransmission Timeout）等。

56.tcp 通信过程的状态是如何变化的

TCP 通信过程中，双方会根据发送和接收的数据包不断切换状态，常见的状态包括 CLOSED、LISTEN、SYN_SENT、SYN_RECV、ESTABLISHED、FIN_WAIT_1、FIN_WAIT_2、CLOSE_WAIT、LAST_ACK、TIME_WAIT。

57.从实用的角度来讲，三次握手的真实目的？（从硬件的角度来看，每一次握手的意义?)

确认双方的通信能力： 通过三次握手，客户端和服务器可以确认彼此的通信能力，包括双方是否能够收发数据包、是否能够正确处理数据包等。

同步双方的初始序列号： 在 TCP 协议中，序列号用于标识每个数据包的顺序和唯一性。在三次握手的过程中，双方会交换初始序列号（Sequence Number），以便后续的数据传输能够正确地按序进行。

协商 TCP 连接参数： 除了确认通信能力和同步序列号外，三次握手还可以用于协商 TCP 连接的一些参数，如窗口大小、最大报文长度等，以便双方可以在连接建立后进行更高效的数据传输。

58.网络的七层模型？每一层的协议？

- 数据链路层：实现节点之间的数据传输，如 ARP、PPP。
- 网络层：实现数据在网络中的传输和路由选择，如 IP、ICMP。
- 传输层：提供端到端的数据传输服务，如 TCP、UDP。
- 会话层：管理应用之间的会话，如 TLS、SSL。
- 表示层：处理数据的表示格式，如 JPEG、ASCII。
- 应用层：提供应用程序与网络通信的接口，如 HTTP、FTP。

59.为什么 TIME_WAIT 状态需要经过 2MSL 才能返回到 CLOSE 状态

TIME_WAIT 状态是为了确保最后一个 ACK 报文能够到达，以防止对端收到了最后一个 ACK 后立即关闭连接，导致连接未完全关闭。2MSL（Maximum Segment Lifetime）是为了确保网络中所有的报文都已经消失，以防止新的连接误认为是之前的连接。

20. TCP 利用滑动窗口实现流量控制的机制？

60.如何根据 IP 获取对方的 MAC 地址？ARP 协议了解一下

可以使用 ARP（Address Resolution Protocol）协议来根据 IP 地址获取对应的 MAC 地址，发送 ARP 请求广播包，然后接收对方的 ARP 应答。

61.Proactor 和 Reactor 的区别和特点。

- Proactor 模型是在异步 I/O 模型的基础上进一步抽象出的设计模式，它将事件处理和 IO 操作分离，使用异步操作来处理 IO 事件，通常应用于高并发的场景。
- Reactor 模型是传统的同步 I/O 模型，它将事件处理和 IO 操作放在一个线程中，使用同步阻塞的方式来处理 IO 事件，通常应用于低并发的场景。

62.怎样加快大文件在网络中传输，根据滑动窗口与拥塞控制考虑。

TCP 利用滑动窗口实现流量控制，发送方根据接收方的反馈调整发送窗口大小，控制数据流量。接收方通过发送窗口大小来告知发送方自己的接收能力，发送方根据接收窗口大小调整发送数据的速率。接收方可以通过告知比较大的窗口的方式提升传输速率。

63.http 和 https 的区别？

- HTTP 是超文本传输协议，数据传输是明文的，不安全；而 HTTPS 是在 HTTP 的基础上加入了 SSL/TLS 加密传输，数据传输是加密的，更安全。
- HTTP 默认端口号是 80，而 HTTPS 默认端口号是 443。

64.HTTP 有哪些常用方法？HTTP 端口号？

- HTTP 常用方法包括 GET、POST、PUT、DELETE、HEAD、OPTIONS 等，用于定义客户端对服务器的请求操作。
- HTTP 默认端口号是 80，HTTPS 默认端口号是 443。

65.SSH 基于 TCP 还是 UDP？端口号？

SSH（Secure Shell）基于 TCP，使用 22 端口。

66.讲一下 WLAN？

WLAN（Wireless Local Area Network）是无线局域网，用于在局域网范围内无线传输数据，常见的协议有 WiFi 等。

67.网卡的中断、几级缓存、网络方面遇到瓶颈怎么解决？

68.什么时候会产生 time_wait，如果系统出现大规模 time_wait 怎么处理？

TIME_WAIT 状态是为了确保连接正常关闭，通常由连接终止方保持一段时间。如果系统出现大规模的 TIME_WAIT 状态，可以通过调整系统参数（如减少 TIME_WAIT 状态的超时时间）、优化代码（如将短连接改成长连接）等方式来处理。

四、数据结构与算法

链表

1. 红黑树的思想，红黑树的特点，红黑树和平衡二叉树的区别？红黑树的查找为什么是 O(logn)?

红黑树是一种自平衡的二叉查找树，它在每个节点上增加了一个额外的表示节点颜色的属性（红色或黑色），并且满足一定的性质来确保树的平衡性。红黑树的特点包括：节点颜色：每个节点要么是红色，要么是黑色。

根节点： 根节点是黑色的。

叶子节点（NIL 节点）： 叶子节点是指树的最底层，且不包含任何数据的节点。红黑树中的叶子节点通常被表示为 NIL 节点，并且是黑色的。

红色节点特性： 红色节点的父节点和子节点必须为黑色。

黑色节点特性： 从根节点到叶子节点的每条路径上，黑色节点的数量必须相同。

红黑树和平衡二叉树的区别在于：

平衡二叉树要求左右子树的高度差不能超过 1，因此可能需要进行更频繁的旋转和调整来保持平衡，而红黑树通过限制节点的颜色和路径上的黑色节点数量来实现自平衡。

红黑树的插入和删除操作相对平衡二叉树更加高效。

红黑树的查找复杂度为 O(logn)，这是因为红黑树是一种近似平衡的二叉查找树，它保证了从根节点到叶子节点的最长路径的长度不超过最短路径的两倍。因此，红黑树的高度最多是对数级别的。在红黑树中，查找操作从根节点开始，每次比较将搜索空间减半，因此查找的时间复杂度是 O(logn)。

2. 负载均衡算法

轮询（Round Robin）：将请求依次分配给每台服务器，按照事先设定的顺序轮流进行分配。优点是简单、均匀，适用于负载相对均衡的情况。

加权轮询（Weighted Round Robin）：在轮询的基础上，给每台服务器设置一个权重值，根据权重值的大小决定分配请求的比例。可以根据服务器的性能和负载情况动态调整权重值，以实现更灵活的负载均衡。

随机（Random）：随机选择一台服务器来处理每个请求。优点是简单，适用于负载较为均匀的情况，但不太适合对服务器性能要求较高的场景。

最少连接（Least Connections）：将请求分配给当前连接数最少的服务器。适用于处理连接持续时间不同的情况，能够有效地均衡负载，但需要实时统计每台服务器的连接数。

IP 哈希（IP Hash）：根据请求的 IP 地址进行哈希计算，将同一个 IP 地址的请求始终分配给同一台服务器。适用于需要保持会话一致性的场景，如用户登录状态。

3. 海量数据 Top K 问题

解决海量数据 Top K 问题的一种常见方法是使用堆（Heap）。具体步骤如下：

构建一个大小为 K 的小顶堆（或者大顶堆），堆中存放当前找到的前 K 个元素。

遍历数据集，将每个元素与堆顶元素进行比较：如果当前元素比堆顶元素大（或者小，取决于是找出最大还是最小的 K 个元素），则将堆顶元素替换为当前元素，并对堆进行一次调整，以保持堆的性质。

如果当前元素比堆顶元素小（或者大），则不做任何操作。

最终堆中剩余的 K 个元素就是所要找到的 Top K 元素。

4. 有损压缩和无损压缩算法

无损压缩算法：

无损压缩算法是指通过压缩数据，使其占用的存储空间更小，但压缩后的数据能够完全还原为原始数据，不会丢失任何信息。因此，无损压缩适用于那些要求数据完整性和精确性的场景，如文本文件、程序代码等。常见的无损压缩算法有：

Huffman 编码： 通过构建变长编码表，将频率较高的字符用较短的编码表示，从而实现压缩。

Lempel-Ziv 算法系列（如 LZ77、LZ78、LZW 等）：利用字符串重复出现的特点，使用短的代号来表示长字符串，从而实现压缩。

Run-Length Encoding（RLE）：将连续出现的重复字符序列用一个字符和其重复次数来表示，以减少重复字符的存储量。

有损压缩算法：

有损压缩算法是指通过舍弃部分数据的精确信息，以换取更高的压缩率。因此，有损压缩压缩后的数据不能完全还原为原始数据，可能会产生一定程度的信息损失。有损压缩适用于那些对数据的精确性要求不是特别高的场景，如音频、视频、图像等。常见的有损压缩算法有：

JPEG： 用于压缩图像文件的有损压缩算法，通过去除图像中的高频信息和颜色信息的精度来实现压缩。

MP3： 用于压缩音频文件的有损压缩算法，通过去除音频中的听觉上不敏感的频率和声音的精度来实现压缩。

H.264/AVC： 用于压缩视频文件的有损压缩算法，通过去除视频中的冗余信息、运动信息和空间信息的精度来实现压缩。

无损压缩和有损压缩算法各有其适用的场景，选择合适的压缩算法取决于数据的特点和应用需

求。

5. 三个有序的序列，查找公共的部分

要查找三个有序序列的公共部分，你可以采用归并排序中合并两个有序序列的思想。具体来说，你可以使用三个指针分别指向三个序列的起始位置，然后逐步比较并移动这些指针来找到公共的元素。

下面是一个简单的 C++示例代码，展示如何找到三个有序序列的公共部分：

王道程序员

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

std::vector<int> findCommonElements(const std::vector<int>& arr1, const
std::vector<int>& arr2, const std::vector<int>& arr3) {
    std::vector<int> commonElements;
    int i = 0, j = 0, k = 0;
    int n1 = arr1.size(), n2 = arr2.size(), n3 = arr3.size();

    while (i < n1 && j < n2 && k < n3) {
        if (arr1[i] == arr2[j] && arr2[j] == arr3[k]) {
            commonElements.push_back(arr1[i]);
            i++;
            j++;
            k++;
        } else if (arr1[i] < arr2[j]) {
            i++;
        } else if (arr2[j] < arr3[k]) {
            j++;
        } else {
            k++;
        }
    }

    return commonElements;
}

int main() {
    std::vector<int> arr1 = {1, 3, 5, 7, 9};
```

在这个例子中，findCommonElements 函数接受三个有序序列作为输入，并返回它们的公共部分。它使用三个指针 i、j 和 k 来分别遍历这三个序列。当三个指针所指向的元素都相等时，该元素就是公共部分的一部分，将其添加到结果向量 commonElements 中，并同时递增三个指针。如果某个序列的元素小于其他序列的元素，则只递增该序列的指针。

在 main 函数中，我们创建了三个示例序列并调用 findCommonElements 函数来找到它们的公共部分，并打印结果。

请注意，这个算法假设序列是升序排列的。如果序列是降序排列的，你需要相应地调整比较操作。此外，如果序列中有重复元素，并且你想要找到所有公共的重复元素，这个算法也可以处理。但是，如果你只想要找到每个公共元素出现一次，那么你可能需要在将元素添加到 commonElements 向量之前进行检查，以确保它不包含重复的元素。

6. 合并两个有序数组或链表

7. 实现一个栈，实现 O(1)时间复杂度求栈的最小元素

要实现一个栈，并且在 O(1)时间复杂度内求栈的最小元素，可以借助辅助栈的思想。具体步骤如下：
维护两个栈，一个用于正常的数据存储，另一个用于存储当前栈中的最小元素。
当入栈时，首先将元素压入正常的栈中，然后比较该元素与辅助栈中的栈顶元素的大小，如果小于等于辅助栈的栈顶元素，则将该元素也压入辅助栈中；否则不做操作。
当出栈时，同时从正常栈和辅助栈中弹出栈顶元素。
当需要获取栈的最小元素时，直接返回辅助栈的栈顶元素即可。

8. 100G 的文本，每行 80k 还是 80 字符，提示用多个机器，多进程，多线程，求出重复最多的行。一个机器内存 8G，计算每个机器大概分多少？能读取 100G 的文本吗？找重复率前十的文本。

9. 两数之和(leetocde 第一题)

10. 将奇数放于数组前面，偶数放于后面

11. 设计一个可以求最大值的栈

12. 余弦相似度算法

13. 如何判断单链表是否有环？

14. 反转单链表

15. 怎么判断单链表是否相交？

16. B 树和 B+树的特点以及应用场景？

17. hashmap 实现的思路和方法(类似于 STL 中的 unordered_map)

18. Hash 表处理冲突的方式有什么？

19. 说一下什么是一致性哈希

一致性哈希（Consistent Hashing）是一种特殊的哈希算法，主要用于解决分布式计算中的节点动态加入和移除对数据分布的影响问题。其基本原理是将哈希空间映射到一个环状空间，并将数据和节点都映射到该环上，从而实现了节点和数据的动态负载均衡。

当需要存储或查找数据时，一致性哈希算法通过哈希算法计算数据的哈希值，并将其映射到环上的一个位置。然后，算法会顺时针找到第一个大于等于该位置的节点，将数据存储在该节点上。这种方式确保了当节点加入或离开系统时，只有少量数据需要重新映射，大部分数据仍然可以在原来的位置上找到，从而减少了数据的迁移量。

一致性哈希算法的主要目标是尽可能小地改变已存在的服务请求与处理请求服务器之间的映射关系，以满足单调性的要求。在普通分布式集群中，服务请求与处理请求服务器之间可以一一对应，即固定服务请求与处理服务器之间的映射关系。然而，这种方式无法对整个系统进行负

载均衡，可能会导致某些服务器过于繁忙而无法处理新的请求。一致性哈希算法通过其独特的映射方式，有效地解决了这个问题。

总的来说，一致性哈希算法在分布式系统中具有广泛的应用，可以有效地处理节点动态变化带来的数据分布问题，提高系统的稳定性和性能。

20. 最短编辑距离问题

21. LRU 实现原理

22. 邻接矩阵是什么？

23. 贪心算法是什么，怎么保证最后的结果是最优的？

树

24. 用 C++实现二叉树的深度优先遍历

首先，定义二叉树节点的数据结构：

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

（1）先序

```
void preorderTraversal(TreeNode* root) {
    if (root == NULL) {
        return;
    }
    cout << root->val << " ";  // 访问根节点
    preorderTraversal(root->left);  // 递归遍历左子树
    preorderTraversal(root->right);  // 递归遍历右子树
}
```

（2）中序

```
void inorderTraversal(TreeNode* root) {
    if (root == NULL) {
        return;
    }
    inorderTraversal(root->left);  // 递归遍历左子树
    cout << root->val << " ";  // 访问根节点
    inorderTraversal(root->right);  // 递归遍历右子树
}
```

（3）后序

```
void postorderTraversal(TreeNode* root) {
    if (root == NULL) {
        return;
    }
    postorderTraversal(root->left);  // 递归遍历左子树
    postorderTraversal(root->right);  // 递归遍历右子树
    cout << root->val << " ";  // 访问根节点
}
```

25. 用 C++实现二叉树层次遍历

二叉树的层次遍历通常使用队列来实现。每一层的节点都依次入队，然后出队并访问该节点，同时将其左右子节点（如果存在）入队。这样，当队列为空时，我们就完成了一层的遍历。

以下是用 C++实现二叉树层次遍历的代码：

王道程序员

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

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

void levelOrder(TreeNode* root) {
    if (root == NULL) {
        return;
    }

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();
        cout << node->val << " ";

        if (node->left) {
            q.push(node->left);
        }

        if (node->right) {
```

在这个例子中，我们首先检查根节点是否为空，如果为空则直接返回。然后，我们创建一个队列 q，并将根节点入队。在 while 循环中，我们不断从队列中取出节点并访问它，然后将其左右子节点（如果存在）入队。当队列为空时，我们就完成了二叉树的层次遍历。

26. 二叉树镜像反转

二叉树的镜像反转（也称为二叉树的对称反转）是指将二叉树的左右子树互换。这可以通过递归或者迭代的方式实现。下面是使用递归的 C++实现方法：

首先，我们定义一个二叉树节点的数据结构：

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

然后，我们实现镜像反转函数：

千道程序员

```
void mirror(TreeNode* root) {
    if (root == NULL) {
        return;
    }

    // 交换左右子树
    TreeNode* temp = root->left;
    root->left = root->right;
    root->right = temp;

    // 递归反转左右子树
    mirror(root->left);
    mirror(root->right);
}
```

在上面的代码中，我们首先检查根节点是否为空，如果为空则直接返回。然后，我们交换当前节点的左右子树。最后，我们递归地对交换后的左右子树进行相同的操作，直到整个二叉树都被反转。

下面是一个完整的示例，包括如何创建二叉树和测试镜像反转函数：

千道程序员

```
#include <iostream>

using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

void mirror(TreeNode* root) {
    if (root == NULL) {
        return;
    }

    TreeNode* temp = root->left;
    root->left = root->right;
    root->right = temp;

    mirror(root->left);
    mirror(root->right);
}

// 辅助函数：用于创建二叉树
TreeNode* createTree() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
```

在这个示例中，我们首先创建了一个简单的二叉树，然后使用中序遍历打印出原始树的结构。接着，我们调用 mirror 函数来反转这个二叉树，并再次使用中序遍历打印出反转后的树的结构。你应该会看到反转前后的树结构是不同的，从而验证了 mirror 函数的正确性。

27. 假设数组为{3,5,7,4,6,2,1},构建一个二叉排序树,并输出排序结果，用 C++实现。

首先，我们需要定义二叉排序树（也叫二叉搜索树）的节点结构，然后实现构建二叉排序树的函数，最后使用中序遍历输出排序结果。下面是 C++的实现代码：

王道程序员

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

// 定义二叉排序树的节点结构
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// 插入一个节点到二叉排序树中
TreeNode* insert(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }
    if (val < root->val) {
        root->left = insert(root->left, val);
    } else if (val > root->val) {
        root->right = insert(root->right, val);
    }
    return root;
}

// 中序遍历二叉排序树，输出排序结果
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
```

在这个实现中，我们首先定义了一个 TreeNode 结构来表示二叉排序树的节点。然后，我们实现了 insert 函数来插入一个节点到二叉排序树中。在 insert 函数中，如果根节点为空，则直接创建一个新节点作为根节点；否则，根据待插入节点的值和根节点的值的大小关系，决定向左子树还是右子树递归插入。

接下来，我们实现了 inorderTraversal 函数来中序遍历二叉排序树，并输出排序结果。中序遍历的顺序是左子树-根节点-右子树，这样遍历的结果就是一个有序的数组。

在 main 函数中，我们定义了一个数组，并遍历数组中的每个元素，将它们插入到二叉排序树中。最后，我们调用 inorderTraversal 函数输出排序结果。

注意：在实际应用中，我们需要考虑二叉树的内存管理，避免内存泄漏。上面的示例代码中省略了释放二叉树内存的代码，实际使用时需要根据具体情况进行释放。

排序

28. 用 C++请实现冒泡排序

```
#include <iostream>
#include <vector>
void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // 交换 arr[j] 和 arr[j + 1]
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}

int main() {
    std::vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    bubbleSort(arr);
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

29. 用 C++请实现归并排序

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

void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<int> leftArr(n1);
    std::vector<int> rightArr(n2);

    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
```

30. 用 C++请实现快速排序

王道程序员

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

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

31. 快速排序算法思想？

快速排序的基本思想是：通过一次排序将待排序的数据分割成独立的两部分，其中一部分的所有数据都比另一部分的所有数据都要小，然后再按这种方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

具体步骤：

- 选择一个基准元素（pivot）。
- 将数组分成两个子数组：一个小于基准元素，另一个大于基准元素。
- 递归地对这两个子数组进行快速排序。

32.快排和堆排的应用场景。

1) 快速排序：

- 在平均情况下，快速排序是所有内部排序算法中速度最快的一种。因此，它非常适合处理大数据集，特别是那些几乎已经有序的数据集。
- 在需要频繁进行部分排序或者插入新元素并重新排序的场景中，快速排序也能很好地发挥其性能优势。

2) 堆排序：

- 堆排序是一种树形选择排序，它的主要优点是时间复杂度相对稳定，无论最好情况、最坏情况还是平均情况，时间复杂度都是 O(nlogn)。
- 当需要处理的数据量很大，且对稳定性没有特殊要求时，堆排序是一个很好的选择。
- 堆排序也常用于构建优先队列，如赫夫曼编码等。

33. 简述你所知道的 std::sort 实现方法。

std::sort 是 C++标准库中的排序函数，其实现通常依赖于具体编译器的优化和平台特性，因此没有一个固定的实现方法。但是，大多数实现会采用一种高效的排序算法，比如快速排序、归并排序或者它们的混合体（如归并排序的变种，称为 Introsort）。

在某些实现中，std::sort 可能会使用三分法（TimSort），这是一种混合排序算法，结合了归并排序和插入排序的特性，对于部分有序或已经有序的数据集特别有效。

此外，std::sort 通常还包含了对小数据集的优化，比如当数据量较小时，可能会直接采用插入排序等简单算法，以避免复杂算法带来的额外开销。

总的来说，std::sort 的实现方法旨在提供一种高效、通用且易于使用的排序功能，以适应各种实际应用场景。

34.写一个排序函数，时间复杂度不大于 O(nlogn)，如果有多线程时该排序算法可如何优化，或者无法多线程优化的原因。

写一个基于归并排序的排序函数，其时间复杂度为 O(nlogn)。归并排序是一种分治思想的排序算法，它将待排序的数组分割成两个子数组，分别进行排序，然后将两个有序子数组合并成一个有序数组。

关于多线程优化，归并排序由于其分治特性，天然地适合并行处理。在多线程环境下，可以将归并排序的递归过程分配给不同的线程执行，每个线程处理一部分数据。但是，要注意合并两个有序子数组时，必须确保两个子数组都已经排序完成，这可能需要同步机制来确保线程间的正确协作。

查找

35. 请设计一个目标字符串查询系统，输入一个目标字符串，找出磁盘上 40 亿个字符串中和目标字符串完全匹配的字符串，可以对 40 亿个字符串（数据量> 100GB，服务器可用内存为 8GB）进行预处理，

a. 如果目标是尽可能快的完成字符串搜索工作，应该如何设计。

b. 如果不仅需要找到目标字符串还需要返回按字典序排在该字符串后面的字符串数量，又该如何设计呢？

36. 实现二分查找算法

二分查找算法是一种在有序数组中查找某一特定元素的搜索算法。搜索过程从数组的中间元素

开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。

以下是用 C++实现二分查找算法的示例代码：

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

using namespace std;

int binarySearch(vector<int>& nums, int target) {
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            // 找到目标值，返回其索引
            return mid;
        } else if (nums[mid] < target) {
            // 目标值在右半部分
            left = mid + 1;
        } else {
            // 目标值在左半部分
            right = mid - 1;
        }
    }
    // 未找到目标值
    return -1;
}

int main() {
    vector<int> nums = {1, 3, 5, 7, 9};
    int target = 5;
    int result = binarySearch(nums, target);
    if (result != -1) {
```

这段代码首先定义了一个 binarySearch 函数，该函数接受一个整数向量 nums 和一个目标值 target 作为输入，然后返回目标值在向量中的索引（如果找到的话）。如果找不到目标值，则返回-1。

main 函数中，我们创建了一个有序的整数向量 nums 和一个目标值 target，然后调用 binarySearch 函数进行查找，并根据查找结果输出相应的信息。

Leetcode

37.【8，21，39，86，127，146】

其它题目

38. 有一个包含 4 个字节的数 a，将它的每个字节相加，结果对 b 取模，如果结果小于 c 则称之为有效数，统计输入的数字中包含有效数的个数。

输入：c b 10 个数 例：3 4 256 257 258 259 260 261 262 263 264 265

输出：有效数的个数 例：4

39. 如果一个字符串符合“辅音+元音(aeiou)+辅音(除了 r)+e"的格式，则称它为有效字符串。输入一组字符串，用空格区分，如果单词中不包含非字母元素，则将单词翻转，统计翻转后的每个单词中包含有效字符串的个数，输出其中的最大值。

输入：一组字符串 例：!maxe a ekekac

输出：最大值 例：2

分析：!maxe a ekekac 翻转后为!maxe a cakeke。cakeke 中有两个有效字符串 cake 和 keke，故输出 2。

40. solo 和 koko 是两兄弟，两人分一堆积木，每块积木都有自己的重量，弟弟 koko 要求积木的重量必须相等，否则就会哭（按照 koko 的标准），可是 koko 只会将数字转化为二进制后相加，而且总会忘记进位（每次都会忘记）。solo 应该如何分积木，才能使自己得到的积木重量尽可能大，如果不能则输出-1？

输入：积木个数 每个积木的重量　例：3 3 5 6

输出：solo 能拿到积木的最大重量　例：11

41. 给出字符串含 数字（0-9）和 ？ 例如（ ???1234?256??????3251? ）其中？可以为任何数，求字符串转成数后 %13 = 5 的情况数量。

给的要求:字符串长度达到 10^5 ，数字前段可以为 0

42. 已知某栅格三角网上三个顶点 A(x,y,z),B(x,y,z),C(x,y,z),求三角网范围内任意点的高程 z;

a)代码实现

b)简述思路

43. 实现全排列

五、数据库

MySQL

1. 数据库事务是什么？

数据库事务是数据库管理系统中的一个操作序列，它被视为一个逻辑单元，包含了一系列对数据库进行读写操作的步骤。这些操作要么全部执行成功，要么全部执行失败，保证数据库的一致性和完整性。

2. 数据库事务有哪些特性？

原子性（Atomicity）：事务中的所有操作要么全部成功，要么全部失败，不允许部分提交。
一致性（Consistency）：事务执行前后，数据库从一个一致的状态转变到另一个一致的状

态。
隔离性（Isolation）：事务的执行过程中，对其他事务是隔离的，互不干扰。
持久性（Durability）：一旦事务提交，其结果应该永久保存在数据库中，即使系统发生故障也不会丢失。

3. 事务的隔离级别有多少种，分别是什么？

数据库事务的隔离级别包括：
读未提交（Read Uncommitted）
读已提交（Read Committed）
可重复读（Repeatable Read）
串行化（Serializable）

4. 不可重复读和幻读区别是什么？可以举个例子吗

不可重复读：在一个事务中，同一查询在不同的时间点执行返回了不同的结果，通常是因为其他事务修改了数据。
幻读：在一个事务中，同一查询在不同的时间点执行返回了不同数量的记录，通常是因为其他事务插入或删除了数据。
举例：假设一个事务在 T1 时刻读取了某个表中的数据，然后在 T2 时刻再次读取同一数据，如果在这期间另一个事务修改了该数据并提交，就会导致不可重复读的问题。而幻读则是在 T1 时刻读取了某个范围的数据，然后在 T2 时刻再次读取同一范围的数据，如果在这期间另一个事务插入了新的数据或删除了已有的数据，就会导致幻读问题。

5. 什么是聚合索引 ？什么是非聚合索引

聚合索引：索引的叶子节点包含了数据行的全部数据，即索引和实际数据存储在一起。例如，MySQL 的 InnoDB 引擎中的主键索引就是一种聚合索引。
非聚合索引：索引的叶子节点只包含索引字段的值和指向数据行的指针，而不包含实际的数据。例如，普通索引或者唯一索引就是一种非聚合索引。

6. 数据库索引怎么用，适合什么场景，什么时候索引失效?

使用：索引通常用于加速查询操作，特别是针对经常用作查询条件的列。
适合场景：适合用于大型数据表，或者需要频繁查询的表。
失效情况：索引可能会失效的情况包括不满足索引列的顺序、使用函数或表达式进行查询、表数据过于庞大导致索引失效等

7. 如何对索引进行优化？

- 选择合适的索引列。
- 避免过度索引。
- 定期进行索引维护和优化。

- 使用覆盖索引。
- 注意索引列的顺序。
- 使用联合索引。
- 避免在索引列上使用函数或表达式。

8. 创建索引一定能加快检索速度吗，为什么？

创建索引可以加快检索速度，但并不是一定的。索引的效果取决于查询的条件、表的大小、索引的选择等因素。如果查询条件不是索引列、表数据较小或者索引选择不当，可能会导致索引的失效，甚至会降低查询性能。

9. 为什么 MySQL 索引要使用 B+树，而不是 B 树或者红黑树？

B+树相对于 B 树和红黑树有更高的查询效率和更好的磁盘读写性能。B+树的叶子节点存储了所有的数据，可以减少磁盘 I/O 次数，提高查询效率。此外，B+树的结构更适合数据库的范围查询和顺序访问操作。

10. 你知道哪些数据库结构优化的手段

- 使用合适的数据类型和字段长度。
- 合理设计数据库表结构，避免冗余和重复数据。
- 使用索引来加速查询。
- 适时进行数据库分表或分区。
- 定期清理和优化数据库。

11. B 树和 B+树区别

- B 树： B 树是一种多路搜索树，每个节点可以有多个子节点，适用于外部存储的数据结构，每个节点包含索引键和指向子节点的指针。

- B+树： B+树也是一种多路搜索树，每个节点可以有多个子节点，除了叶子节点外，其他节点不存储数据，只存储索引键和指向子节点的指针。所有叶子节点按顺序连接形成链表，方便范围查询。

12. 怎么判断一个查询是否是高效率的？

使用 explain 可以获取查询的执行计划。一个查询是否高效率取决于多个因素，包括：

执行时间：查询所需的执行时间越短，效率越高。
扫描行数：查询扫描的行数越少，效率越高。
是否利用索引：查询是否利用了适当的索引来加速检索。
是否使用合适的算法和数据结构：是否选择了合适的算法和数据结构来执行查询操作。
是否进行了适当的优化：是否对查询语句进行了优化，如避免使用不必要的表连接、避免使用全表扫描等。

13. 如何优化查询语句？

使用索引：确保查询语句中涉及的字段有合适的索引。
优化查询条件：避免在查询条件中使用函数操作、类型转换等，以提高索引的利用率。
避免全表扫描：尽量避免使用全表扫描，考虑合适的索引覆盖查询。
减少数据返回：只返回必要的数据，避免返回过多的数据。

14. MySQL 的约束有哪些？

主键约束（PRIMARY KEY）
外键约束（FOREIGN KEY）
唯一约束（UNIQUE）
非空约束（NOT NULL）

15. inner join, left join, right join, outer join 的区别？

INNER JOIN： 返回两个表中满足连接条件的行。
LEFT JOIN（或 LEFT OUTER JOIN）：返回左表中的所有行，以及右表中满足连接条件的行。
RIGHT JOIN（或 RIGHT OUTER JOIN）：返回右表中的所有行，以及左表中满足连接条件的行。
OUTER JOIN（或 FULL OUTER JOIN）：返回左表和右表中的所有行，如果某行在其中一个表中没有匹配，则另一个表中对应的列值为 NULL。

16. mysql 如何合并两个表?

MySQL 可以通过使用 JOIN 操作来合并两个表，根据需要选择 INNER JOIN、LEFT JOIN、RIGHT JOIN 或 OUTER JOIN 等不同的连接方式来进行合并。

17. 共享锁与独占锁。

- **共享锁（Shared Lock）**：允许多个事务同时对同一资源进行读取操作，但不允许任何事务对资源进行写入操作。

- **独占锁（Exclusive Lock）**：仅允许一个事务对资源进行写入操作，其他事务无

法同时进行读取或写入操作。

18. 乐观锁和悲观锁。

- **乐观锁**： 假设多个事务不会同时修改同一数据，只在提交时检查是否冲突，常用于并发度较高的场景。一般通过用户设计表时增加版本号字段来实现。

- **悲观锁**： 假设多个事务会同时修改同一数据，通过在读取数据时加锁来防止其他事务对数据的修改，常用于并发度较低的场景。

19. 了解过存储过程吗？

存储过程是一组预编译的 SQL 语句集合，存储在数据库中，可以像调用函数一样被调用。它可以接收参数、执行一系列 SQL 语句并返回结果，可以提高数据库的性能和安全性。

20. 了解过数据库视图吗？

数据库视图是一种虚拟的表，由一个或多个表的行和列组成，可以像表一样被查询。视图不存储实际的数据，而是根据查询语句动态生成结果集，可以简化复杂的查询操作，提高数据的安全性和可读性。

Redis

21. Redis 常见数据结构以及使用场景分别是什么？

字符串(Strings)、列表(Lists)、集合(Sets)、有序集合(Sorted Sets)、散列(Hashes): 键值对集合，适合存储对象。

22. Redis 持久化机制可以说一说吗？

RDB(Redis Database): 在指定的时间间隔内生成内存中数据的快照。

AOF(Append Only File): 记录每个写操作命令，服务器重启时会重新执行这些命令来恢复数据。

23. 了解 Redis 的线程模型吗？可以大致说说吗？

Redis 使用多路复用技术结合单线程模型，所有的命令序列化执行，但内部使用多线程来处理某些耗时的操作，如异步写入磁盘

24. 有没有读过 Redis 源码

是否读过 Redis 源码，这个问题针对个人经验，如果你读过，你可以分享你的观点和收获；如果没有，可以考虑阅读关键模块以增加理解。

25. C++中的 Map 也是一种缓存型数据结构，为什么不用 Map，而选择 Redis 做缓存？

为什么选择 Redis 而不是 C++中的 Map 作为缓存?虽然 C++中的 Map 也可以作为键值存储，但 Redis 提供网络访问、内置持久化、数据结构丰富性和高并发处理等功能，这使得 Redis 在分布式环境中作为缓存系统更为合适。

26. Redis 是如何部署的？

Redis 可以单独作为进程运行，也可以在容器中运行。它支持多种部署方案，包括单节点、主从复制、哨兵模式和集群模式，以提高可用性和分区容错性。

27. 简述一下主从复制和哨兵模式的原理？

主从复制: 主节点负责写操作，自动将数据同步到一个或多个从节点，从节点可以提供读服务或作为数据备份。

哨兵模式: 自动监控主从结构，实现故障转移。哨兵集群监视所有 Redis 服务器，并在主服务器故障时自动选择并切换到新的主服务

28. 解释一下 Redis 缓存雪崩，缓存穿透，缓存预热？

缓存雪崩: 大量缓存同时失效，导致数据库压力骤增。解决方法包括设置不同的过期时间、使用备用缓存等。

缓存穿透: 查询不存在的数据，导致每次都要访问数据库。可以通过布隆过滤器或缓存空对象来防止。

缓存预热: 在缓存启动后预先加载数据到缓存中，避免启动初期数据库访问负载

29. redis 的有序集合底层实现是什么？如果让你实现，你会怎么实现？

Redis 的有序集合(Sorted Sets)底层主要是通过跳表(Skip List)实现的。跳表是一种可以实现快速查找、插入、删除操作的数据结构，其性能可以与平衡树相媲美，同时保持了链表的灵活性。跳表通过在原始链表上增加多级索引来实现快速查找，这样可以在对数时间复杂度内完成查找，同时插入和删除操作也能够保持较低的复杂度。

如果我要实现一个类似 Redis 的有序集合，我会这样设计：

数据结构：使用跳表作为主要的数据结构，用于存储有序元素。结合哈希表，用于存储成员到分数的映射，这样可以在 O(1)的时间复杂度内找到任意成员的分数。

插入操作：在跳表中插入新的元素，同时更新索引。更新哈希表，记录成员到分数的映射。

删除操作：在跳表中删除元素，同时更新索引。在哈希表中删除成员的记录。

查找操作：通过跳表实现范围查找，以获取一个分数范围内的所有成员。通过哈希表直接获取单个成员的分数。

更新分数：首先在哈希表中更新成员的分数。在跳表中删除原始分数的成员，然后以新分数重新插入成员。

持久化和复制：类似于 Redis，提供 RDB 和 AOF 两种持久化选项，确保数据的安全性。支持主从复制，确保数据的可用性和一致性。通过这样的设计，可以实现一个高效、可持久化且支持复制的有序集合。

30. redis 失效时应该怎么处理，如果让你设计你方案，你会怎么设计？

当 Redis 失效时，可以采用以下设计方案确保系统的稳定性和数据的一致性：

故障检测与通知:实施定时健康检查，如使用 PING 命令检测 Redis 实例状态。一旦检测到 Redis 异常，立即通知运维人员并自动触发报警系统。

自动故障转移:利用 Redis Sentinel 或集群来监控 Redis 实例，自动进行主从切换。确保客户端支持自动重新连接到新的主节点。

读写分离与流量控制:在可能的情况下，通过主从复制实现读写分离，减轻主节点负担。实现流量控制机制，如请求限速，避免大量并发请求涌向数据库。

数据备份与恢复策略:定期执行数据备份，确保能够从最新的备份中恢复数据。准备明确的数据恢复流程和工具，以便快速恢复 Redis 服务。

降级策略:设计降级方案，如在 Redis 不可用时，可临时切换到其他缓存解决方案或直接从数据库读取数据。提前准备好降级脚本或程序，确保快速切换。

六、设计模式

1. 为什么用组合而不要用继承？

使用组合而不是继承的主要原因有以下几点：

1) 灵活性：组合比继承更灵活。通过组合，你可以动态地改变对象的行为，因为你可以在运行时改变其组件。而继承是静态的，一旦确定了类的继承关系，就难以更改。
2) 防止代码爆炸：过多的继承层次会导致代码复杂性增加，甚至可能引起代码爆炸。使用组合可以避免这种情况，因为你可以通过重用现有的组件来创建新的对象。
3) 控制复杂性：继承会将父类的所有细节暴露给子类，这可能导致子类变得复杂且难以维护。通过组合，你可以只暴露必要的接口，从而控制复杂性。
4) 实现接口复用：组合可以实现接口复用，通过对象组合，可以在多个不同的类中复用同一个接口的实现。

2. 手写单例模式

在 C++中实现单例模式，我们需要确保几个关键点：

1) 私有化构造函数和析构函数，以防止从类外部实例化对象。
2) 提供一个静态的公有方法，用于获取单例对象的指针或引用。
3) 在类中定义一个静态的指向单例对象的静态数据成员。
4) 确保单例对象的线程安全性，特别是在多线程环境下。

下面是使用懒汉式单例模式，并加入线程安全性的 C++代码实现：

```
#include <iostream>
#include <mutex>

class Singleton {
private:
    static Singleton* instance; // 静态成员变量，保存单例对象的指针
    static std::mutex mtx; // 互斥锁，用于线程同步

    Singleton() {} // 私有的构造函数
    ~Singleton() {} // 私有的析构函数

public:
    // 获取单例对象的静态方法
    static Singleton* getInstance() {
        if (!instance) { // 第一次检查实例是否存在
            std::lock_guard<std::mutex> lock(mtx); // 使用互斥锁确保线程安全
            if (!instance) { // 第二次检查实例是否存在，如果不存在才创建
                instance = new Singleton();
            }
        }
        return instance;
    }

    // 假设 Singleton 类有一个打印方法
    void printMessage() {
        std::cout << "Singleton object is being used." << std::endl;
    }
};
```

在这个实现中，我们使用了 std::mutex 来确保在多线程环境下 getInstance()方法的线程安全性。std::lock_guard 是一个 RAII（Resource Acquisition Is Initialization）风格的锁，它会在构造时自动加锁，在析构时自动解锁，从而简化了锁的管理。

此外，我们使用了双重检查锁定来避免不必要的锁竞争，只有在 instance 为 nullptr 时才进入同步块。

需要注意的是，在实际使用中，单例对象的生命周期管理也很重要。在上述示例中，我们使用了 new 来动态分配内存，但在程序结束时并没有显式释放它。在实际应用中，可能需要提供一个方法来释放单例对象占用的内存，或者在程序结束时使用某种机制（如智能指针或 atexit 函数）来自动处理。

3. 单例模式的构造函数？单例模式的创建过程？如何保证线程安全？

单例模式的构造函数通常是私有的，以确保无法从类外部实例化该类的对象。单例模式的创建过程通常包括一个静态的私有成员变量来保存单例对象的引用，以及一个公开的静态方法来获取这个单例对象。

保证线程安全的方法有多种，其中常见的有两种：

1) 饿汉式：在类加载的时候就完成初始化，所以类加载较慢，但获取对象的速度快。这种方式是线程安全的。
2) 懒汉式（双重检查锁定）：延迟初始化，在第一次调用 getInstance()方法时创建实例。为了避免在多线程环境下多次创建实例，使用双重检查锁定来确保线程安全。

4. 如何使用单例模式，有什么注意事项？

在 C++中使用单例模式时，你需要确保单例类的实例在整个程序运行期间只被创建一次，并且提供一个全局访问点来获取这个实例。以下是一些使用单例模式的注意事项：

注意事项

1) 线程安全：

如果你的程序是多线程的，那么必须确保单例的创建是线程安全的。通常，这可以通过在创建实例时使用互斥锁（mutex）或原子操作（atomic operations）来实现。

2) 防止拷贝：

单例类不应该允许拷贝构造和拷贝赋值操作，因此需要将这两个操作设为私有，并且不提供实现。这可以通过将拷贝构造函数和拷贝赋值操作符声明为 delete 来实现。

3) 防止继承：

单例类通常不应被继承，因为继承可能会破坏单例的特性。将类的继承设为私有或删除可以防止继承。

4) 懒汉式与饿汉式：

单例模式有两种常见的实现方式：懒汉式和饿汉式。懒汉式延迟创建实例，直到第一次使用时才创建；饿汉式则在类加载时就创建实例。饿汉式是线程安全的，但懒汉式需要额外的同步措施。

5) 资源管理：

单例对象的生命周期管理很重要。如果单例对象持有动态分配的资源（如内存、文件句柄等），则必须确保这些资源在程序结束时得到正确释放。这通常需要在单例的析构函数中处理。

6) 单例的访问方式：

提供一个静态方法来获取单例对象的引用或指针。这个方法应该是公有的，并且应该是线程安全的。

7) 避免全局对象：

在 C++中，全局对象的初始化顺序在不同的编译单元之间是不确定的。这可能会导致依赖于其他全局对象初始化的单例出现问题。尽量避免在全局范围内使用单例。

5. 如果使用单例模式时创建了多个对象，如何定位问题？

如果在使用单例模式时创建了多个对象，这通常意味着单例模式的实现存在问题。定位问题的步骤如下：

1) 检查构造函数：确保单例类的构造函数是私有的，以防止外部代码实例化该类。
2) 检查静态方法：确保获取单例对象的静态方法没有逻辑错误，并且确保在多线程环境下是线程安全的。
3) 调试和日志：使用调试工具逐步执行代码，并添加日志输出，以跟踪对象的创建过程。这有助于发现哪里创建了额外的实例。

6. 请简述一下适配器模式？

适配器模式是一种结构型设计模式，主要用于将一个类的接口转换成客户端所期望的另一个接

口，使得原本由于接口不兼容而不能一起工作的类可以协同工作。适配器模式主要包含三个角色：目标接口、适配器和被适配者。

目标接口是客户端所期望的接口，它定义了客户端可以调用的方法。适配器是将被适配者的接口转换成目标接口的中间件，它实现了目标接口，并持有一个被适配者的对象。被适配者是客户端所需要的对象，但其接口与目标接口不兼容。

适配器模式有两种主要类型：类适配器模式和对象适配器模式。在类适配器模式中，适配器通过继承实现目标接口，并持有被适配者的实例，在适配器的方法中调用被适配者的方法。在对象适配器模式中，适配器持有被适配者的实例，通过组合实现目标接口，并在适配器的方法中调用被适配者的方法。

在实际开发中，适配器模式经常用于将旧的接口转换成新的接口，或者将多个类的接口转换成一个统一的接口。适配器模式可以提高代码的复用性和扩展性，同时使得代码更加易于维护和测试。

然而，在使用适配器模式时，需要注意保持接口的一致性和简洁性，避免过度使用导致代码复杂度的增加。同时，需要仔细考虑适配器和被适配者之间的耦合度，避免过度耦合导致代码的可维护性降低。

下面是一个用 C++手动实现适配器设计模式的简单示例。在这个示例中，我们有一个旧式的打印机接口，而我们的新代码需要一个不同的接口。我们将创建一个适配器来桥接这两个接口。

首先，我们定义被适配者（OldPrinter）的接口和类：

```
// 被适配者：OldPrinter
class OldPrinter {
public:
    void printOldStyle(const std::string& message) {
        std::cout << "Old style printing: " << message << std::endl;
    }
};
```

然后，我们定义目标接口（NewPrinterInterface）：

```
// 目标接口：NewPrinterInterface
class NewPrinterInterface {
public:
    virtual ~NewPrinterInterface() {}
    virtual void printNewStyle(const std::string& message) = 0;
};
```

现在，我们创建适配器（PrinterAdapter），它实现了目标接口并持有被适配者的实例：

```
// 适配器：PrinterAdapter
class PrinterAdapter : public NewPrinterInterface {
private:
    OldPrinter oldPrinter;

public:
    PrinterAdapter() {}

    // 实现目标接口的方法
    void printNewStyle(const std::string& message) override {
        // 调用被适配者的方法，并可能进行转换
        oldPrinter.printOldStyle(message);
        // 这里可以添加其他转换逻辑
    }
};
```

最后，我们在客户端代码中使用适配器：

千道程序员

```
int main() {
    // 客户端原本期望使用 NewPrinterInterface 接口
    NewPrinterInterface* printer = new PrinterAdapter();

    // 客户端调用 printNewStyle 方法
    printer->printNewStyle("Hello, world!");

    // 释放内存
    delete printer;

    return 0;
}
```

在这个例子中，PrinterAdapter 类实现了 NewPrinterInterface 接口，并内部使用了 OldPrinter 类。当客户端调用 printNewStyle 方法时，适配器将其转换为对 OldPrinter 的 printOldStyle 方法的调用。这样，客户端就可以通过新的接口来使用旧的打印机了。

请注意，在这个简单的例子中，适配器没有进行太多转换工作。在实际应用中，适配器可能需要进行更复杂的转换，以确保客户端能够无缝地使用新的接口。同时，这个示例没有考虑异常处理、内存管理（如使用智能指针）和线程安全等高级话题，这些在实际项目中也是非常重要的。

7. 实现一个简单的观察者模式

观察者模式（Observer Pattern）是一种行为设计模式，它允许对象（观察者）订阅另一个对象（主题）的状态变化，并在主题状态发生改变时自动更新。下面是一个用 C++实现观察者模式的简单示例：

首先，我们定义一个 Subject 类，它代表被观察的主题，持有一个观察者的列表，并提供订阅和通知的方法：

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

// 观察者接口
class Observer {
public:
    virtual ~Observer() = default;
    virtual void update(const std::string& message) = 0;
};

// 主题类
class Subject {
private:
    std::list<std::shared_ptr<Observer>> observers;
    std::string state;

public:
    void attach(const std::shared_ptr<Observer>& observer) {
        observers.push_back(observer);
    }

    void detach(const std::shared_ptr<Observer>& observer) {
        observers.remove(observer);
    }

    void notifyObservers() {
        for (const auto& observer : observers) {
            observer->update(state);
```

然后，我们定义一个具体的观察者类 ConcreteObserver，它实现了 Observer 接口：

```
// 具体的观察者类
class ConcreteObserver : public Observer {
private:
    std::string name;
    Subject* subject;

public:
    ConcreteObserver(const std::string& name, Subject* subject)
        : name(name), subject(subject) {}

    void update(const std::string& message) override {
        std::cout << name << " received message: " << message << std::endl;
    }
};
```

最后，我们在 main 函数中创建主题和观察者，并演示观察者模式的使用：

```
int main() {
    // 创建主题
    Subject subject;

    // 创建观察者并订阅主题
    ConcreteObserver observer1("Observer 1", &subject);
    ConcreteObserver observer2("Observer 2", &subject);
    subject.attach(std::make_shared<ConcreteObserver>(observer1));
    subject.attach(std::make_shared<ConcreteObserver>(observer2));

    // 修改主题状态，并通知观察者
    subject.setState("New state");

    // 取消一个观察者的订阅
    subject.detach(std::make_shared<ConcreteObserver>(observer1));

    // 再次修改主题状态，这次只有 observer2 会收到通知
    subject.setState("Another new state");

    return 0;
}
```

这个简单的观察者模式实现中，Subject 类维护了一个观察者列表，并在状态改变时通知所有观察者。Observer 是一个接口，定义了观察者需要实现的方法。ConcreteObserver 是实现了 Observer 接口的具体类，它持有一个指向 Subject 的指针，并在 update 方法被调用时执行相应的操作。

在 main 函数中，我们创建了一个 Subject 和两个 ConcreteObserver 对象，并将观察者附加到主题上。然后，我们修改主题的状态，这会导致所有注册的观察者被通知。之后，我们取消了其中一个观察者的订阅，并再次修改主题状态，这次只有另一个观察者会收到通知。

8. 使用过的设计模式，应用场景，如何应用？阐述业务背景和应用的方式。

可以自行列举了。比如单例、代理、适配器、迭代器、观察者等

七、项目

网盘项目

1. 客户端发消息给服务器，服务器端是如何解析的？

客户端-服务端使用私有协议来规定消息的格式

2. 多个用户上传同一份文件该如何处理？

使用同步机制，优先第一个用户先上传，后续上传触发秒传

3. 秒传如何实现？

每个文件保有一个指纹信息（md5 或者 sha1），实际上传文件内容之前先校验

4. 断点续传如何实现？

传输文件之前先获取已经上传的文件大小，再客户端-服务端都是用 lseek 偏移后读写

5. 讲一下虚拟文件目录

使用数据库表结构模拟文件树结构

6. 前后端的通信方式知道哪几种？

使用 tcp 连接

7. 当你上传文件或更改文件时，如果出现问题，网络中断了；会不会导致数据库和文件对不上，怎么解决的？

传输完成以后，补充一个哈希校验码

8. 如何做 token 验证？

服务端提供 token 生成函数

客户端保存 token 值

9. 网盘项目的网络通信方式

tcp 和私有协议

10.有考虑过多线程同时上传一个文件的问题吗？

使用文件锁等同步机制

11.md5 的算法是自己实现的吗？

使用 openssl 库

12.文件如何和用户绑定？

数据库表中记录

13.连接使用的是长连接还是短链接？

长连接+超时踢人

14.长连接 socket 的参数是怎么设置的？

15.项目为什么要进行文件去重？

节约存储空间、提高上传速度

16.线程池是如何实现的？

线程池是一个经典的生产者消费者问题模型。

在提前创建 N 个线程之后，让每个线程阻塞在任务队列上。当任务到达时，其中一个线程去任务队列中获取任务即可。任务队列的实现中涉及到互斥锁和条件变量，来完成线程间的同步和互斥操作。

17.线程池在项目当中是怎么用的，分别有哪些线程，它们是怎么分工的？

主线程和工人线程

18.线程池中的线程数目是否会随并发量动态增加？

一般不会动态变化，可以手动扩容

19.如何确定线程池中线程的状态？

ps -elLF 命令

搜索引擎项目

20.一个网页的信息是通过什么形式存储的？

通过 xml 的形式进行存储的，设计了 docid、doctitle、docurl、doccontent 字段

21.Simhash 是什么，怎么使用的？

Simhash 是一种局部敏感哈希算法，主要用于处理海量文本数据，以检测文本的相似性。这种

算法的主要特点在于，即使两个字符串具有一定的相似性，经过哈希处理后，它们仍然能保持这种相似性，这在普通的哈希算法中是无法实现的。

用于网页去重，计算两篇网页的海明距离，判断是否为相同的网页。

22.介绍一下倒排索引

倒排索引是实现“单词-文档矩阵”的一种具体存储形式，通过倒排索引，可以根据单词快速获取包含这个单词的文档列表。这种索引方法主要用于全文搜索，是文档检索系统中最常用的数据结构。

与正排索引相比，倒排索引的查询效率更高。正排索引是以文档的 ID 为关键字，索引记录文档中每个字的位置信息，查找时需要遍历所有文档直到找出所有包含查询关键字的文档。而倒排索引以字或词为关键字进行索引，直接定位到包含该关键字的所有文档，因此查询速度更快。

23.余弦相似度算法。

余弦相似度的基本思想是，如果两个向量的夹角越小，那么它们就越相似。这是因为余弦函数在[0, π]的范围内是单调递减的，当两个向量的夹角趋近于 0 时，余弦值趋近于 1，表示两个向量非常相似；当夹角趋近于π时，余弦值趋近于-1，表示两个向量非常不相似。

在项目中用来进行网页排序。

24.任务队列是怎么实现的，阻塞还是非阻塞？

任务队列是阻塞式的。

```
using ElemType = function<void()>;
class TaskQueue
{
public:
    TaskQueue(size_t queSize);
    ~TaskQueue();

    //任务队列空与满
    bool empty() const;
    bool full() const;

    //添加任务与执行任务
    void push(ElemType &&value);
    ElemType pop();

    //唤醒所有等待在_notEmpty 条件变量上的线程
    /* void wakeupAllNotEmpty(); */
    void wakeup();

private:
    size_t _queSize;//任务队列的大小
    queue<ElemType> _que;//存放数据的数据结构
    MutexLock _mutex;//互斥锁
    Condition _notEmpty;//不空的条件变量
    Condition _notFull;//不满的条件变量
    bool _flag;//让工作线程唤醒的时候，可以退出来
};
```

25.负载均衡怎么做的？

采用 nginx 做反向代理，实现负载均衡。

Nginx 反向代理是一种服务器架构，其中 Nginx 服务器充当客户端和服务器之间的中介。在这种架构中，客户端向 Nginx 服务器发送请求，然后由 Nginx 服务器将这些请求转发到搜索引擎的服务器，这样客户端与 Nginx 之间的连接与实际的搜索引擎服务器之间的连接是分开的。

26.什么是最短编辑距离

最短编辑距离是指两个字符串之间，由一个字符串转化成另一个字符串所需要的最少编辑操作次数。编辑操作包括插入一个字符、删除一个字符或替换一个字符。编辑距离越小，说明两个字符串越相似。这种相似程度的计算方法在字符串匹配、自然语言处理、生物信息学等领域有着广泛的应用。

在计算最短编辑距离时，通常使用动态规划的思想。动态规划是一种解决问题的策略，它可以将一个问题分解为多个子问题，并存储子问题的解以避免重复计算，从而提高算法的效率。

27.双缓存轮换是怎么做到的，为什么不使用单缓存加锁？

单缓存加锁，在并发比较大的情况下会影响实时性。

双缓存的实现是一个线程对应两个缓存，一个缓存 A 进行正常查询，一个缓存 B 进行备份。当需要同步多线程中的缓存时，缓存系统整体对外的查询只提供读操作，具体实现是让每一个线程中备份缓存 B 只提供读操作，缓存 A 进行同步更新操作。当 A 完成同步操作后，再切换回来对外提供只读操作，再同步更新缓存 B。当 B 完成同步更新操作后，缓存系统再进行正常的查询（读写操作）。

28.LRU 算法的原理。 为什么使用 LRU 算法，还可以选用什么算法？

LRU 算法的原理是基于数据的访问历史来预测未来的访问模式。具体来说，LRU（Least Recently Used）算法将最近最少使用的数据移除，以便为最新读取的数据提供空间。其实现依赖于维护一个双向链表，该链表按照数据的访问时间从新到旧进行排序。当访问某个数据时，如果它在缓存中，就将其从链表中移除并重新插入到链表头部；如果不在缓存中，则从外部获取数据并将其插入到链表头部。通过这种方式，链表尾部的数据就是最近最少使用的，当需要淘汰数据时，就选择链表尾部的数据进行淘汰。

缓存实现采用 LRU 算法的原因主要有以下几点：

局部性原理：根据程序的执行特点，最近被访问过的数据在将来被再次访问的可能性较大。因此，通过淘汰最久未使用的数据，可以提高缓存的命中率，从而提升系统性能。

易于实现：LRU 算法实现相对简单，只需要维护一个双向链表和一些基本操作即可。

适用性广：LRU 算法适用于多种场景，特别是在缓存大小有限且访问模式较为稳定的情况下，表现尤为出色。

除了 LRU 算法外，还有其他算法可以用来进行缓存淘汰，例如：

FIFO（First In First Out）算法：按照数据进入缓存的先后顺序进行淘汰，最先进入的数据最先被淘汰。这种算法实现简单，但可能导致一些频繁访问的数据被过早淘汰。

LFU（Least Frequently Used）算法：根据数据的访问频率进行淘汰，访问次数最少的数据最先被淘汰。这种算法可以更好地适应访问模式的变化，但在处理访问频率相同的数据时可能不够准确。

近似 LRU 算法：为了平衡性能和准确性，有些实现采用了近似 LRU 算法。这种算法通过随机选择一部分数据进行 LRU 淘汰，而不是对所有数据进行全 LRU 运算，从而在一定程度上牺牲了准确性以提高算法执行效率。

29.缓存和数据库如何实现同步操作

数据库中的数据是只读的，不会更新数据库，所以也不需要将缓存与数据库进行同步。

如果数据库更新了，那就重新加载即可。

30.项目当中的任务队列具体会有哪些任务？

关键字推荐和网页搜索

31.搜索引擎项目用了几个进程几个线程池

项目可以采用多进程多线程架构，也可以采用单进程加线程池架构。

实际实现过程中采用的是单进程，具体采用的是 Reactor + 线程池（1 个）的架构。

Http 项目

32.介绍一下这个 workflow 编程范式是什么，怎么用的，以及这个框架能对项目起到什么作用？

workflow 是异步回调范式，项目的底层网络通信和任务调度由框架完成的。

33.有没有用什么二进制通信协议？

私有协议 protobuf

八、开放性问题

1. 离职原因

2. 大学学过的和软件开发相关的课程？

3. 互联网以后的发展

4. 有没有看过开源代码和较深的数据结构？

5. 给一个尺子，算一栋楼的高度

6. 1000 支试管，只有一支有毒，怎么用最少的小白鼠检测出有毒？

7. 最佩服的人

8. 最大的挫折

9. 有没有研究新技术？

10. 毕业论文是什么？你对毕业论文的理解(这个论文的方向为什么吸引你)？

11. 你有什么想问的？

12. C++了解到什么程度？

13. 你之前工作小组怎么构成的？都干了多久？

14. 你之前公司是做什么业务的？

15. 之前工作累吗？工作时间之类的问题？

16. 你过去的工资怎么构成的？

17. 了解过我们公司吗？

18. 了解 QT 客户端开发吗？

19. 是否对 OpenGL 有了解？

20. 工作压力主要来自哪儿？

21. 工作过程中最难忘的一件事儿？

22. 未来的职业规划？

23. 进入公司后，一年的规划。

24. 项目需求怎么来的？

25. 项目怎么展开的？

26. 代码怎么测试的？

27. 测试用例自己写吗？

28. 多久做一个项目？

29. 修复 bug 的过程？讲一个具体修复过的 bug？怎么发现的 bug？

30. 如果某个模块运行过程中内存增长速度过快，应该怎么处理？

31. 处理过的难度最高或挑战性最大的问题？

32. 有没有什么学习计划？使用这种学习计划多长时间了，会不会有完不成的情况？

33. 旁边有人不停打扰，怎么办？

34. 你在找工作时更看中什么？

35. 接受加班么？

36. 期望薪资多少呢？

37. 对这个行业有什么看法

38. C++ primer 中令你印象最深刻的是什么？

39. 是否会用 UML?

40. 上一家公司的总体规模

41. 上一家公司是什么样的开发，是敏捷式还是瀑布式还是别的类型的开发模式

42. 如何验证自己所写代码的质量？

43. 在以前的工作当中是否参与到某几款软件或者系统的设计工作/软件设计？

自由发挥~

项目相关

44. 介绍一下你的项目。

自我准备，提前写好稿子，多读几遍，达到能背下来的状态。

45.整个项目做了多长时间？项目多少个人？

项目组人数，担任的角色，工作的流程，如何提交给测试？

整体项目做了 3 个月左右。项目组 5 个成员，1 个 leader, 2 个后端，2 个前端。

46.如果需要在已有项目上新增功能，需要考虑哪些问题？

可以考虑从项目的架构上和扩展性方向进行输出

47.你的项目瓶颈在哪？怎么优化的？性能提升了多少？

可以从响应实时性、并发连接数、存储介质、网络带宽等角度进行说明。

48.使用过什么测试性能的工具？

Valgrind、Perf 可以了解一下。

49.项目中遇到的最大的问题是什么？

自行准备，可以将做项目过程中碰到的 bug 梳理一下，好在这里进行输出。

50.你们系统的性能如何？系统挂掉了怎么办？系统挂掉后，会不会有断流出现？

系统性能一般能满足需求了。

系统挂掉了，会自动重启的。不过也会有相应的日志进行记录，以供后续进行 bug 的定位。程序出错时也会产生 core dump 文件，通过 core dump 文件，来查找问题的原因。

系统挂掉后，客户端这边就需要重新请求了。

51.项目有没有做过压测，怎么做的？

甩锅疗法：压测这块不是我负责的。不过我有做过稳定性测试，服务器运行了 1 个星期，没有挂掉，跟 leader 沟通后，就交给 leader 进行处理了。如果是讨论项目的并发数，这个基本上够用了，几千个连接应该是绰绰有余了。

52.常用的云服务有用过吗？它的原理是？

云服务的原理主要基于虚拟化技术、分布式存储和网络技术等。

首先，虚拟化技术使得云服务提供商能够将物理硬件资源（如服务器、存储设备等）虚拟化成多个逻辑上的资源单元，从而提高了资源的利用率和灵活性。

其次，分布式存储技术使得大量数据能够在多个物理节点上进行存储和管理，从而实现了数据

的冗余备份和容灾恢复。

最后，网络技术则使得云服务能够通过网络进行远程访问和管理，从而为用户提供了便捷的服务体验。

53.私有云有没有用到集群

暂时没有采用。如果有准备集群的知识点，也可以输出。

九、面试真题

1. 每个 C/C++头文件头部都需要使用类似如下的语句，引入这种定义是为什么？（2 分）

```
#ifndef _C_CPP_basic_TEST_H
#define _C_CPP_basic_TEST_H


#endif
```

2. 当引用外部头文件时，有以下两种引用包含方式：

```
#include "myhead1.h"
#include <myhead2.h>
```

a) 预编译阶段，编译器会对#include 语句做如何的处理？（2 分）

b) 这两种引用方式的区别在于？（2 分）

c) 如果出现 file not found 的错误，该如何解决？（3 分）

3. 在 C++中，通常需要使用命名空间，比如 using namespace std;

或者直接使用前缀指定 std::function

a) C++为什么要使用命名空间？（2 分）

b) 什么时候用 using，什么时候用前缀的方式？（2 分）

4. #define 的用法细节

a) #define 和 typedef 有什么区别？（2 分）

b) 完成以下计算：

#define EQ(a) 0x12==a

EQ(12)=?　　（2 分）

#define ADD(a) a*2

ADD(1+1)=?　（2 分）

#define TH(a) (!a||a==-2)

TH(1)=?　（1 分）

TH(-2)=?　（1 分）

c) 用 #define 来写宏函数，如上面的 ADD(a)，有什么缺陷？该如何避免/改进（2 分）

5. 字节计算（32 位编译器）

```
struct A {
    unsigned int a;    //4
    int b;      //4
};
```

sizeof(struct A)=?　(2 分)

```
struct B {
    short a;//2
    int b;//4
};
```

sizeof(struct B)=?　（2 分）

```
const char* p = "string";
```

sizeof(p)=?　　（2 分）

6. 简述变量声明和定义的区别（3 分）

7. 在 C++文件中调用链接 C 语言代码函数，需要做什么特殊处理？（3 分）

8. 以下代码编译时会报错，请找出报错原因（3 分）

```
class A {
public:
    A():data(0){};
    ~A(){};

    int data;
    void inline inc(){data++;};
    static void dec(){data--;};
};
```

9. 简述值传递、指针传递（地址传递）的区别 （3 分）

10. static 有多种不同的用法，比如加在变量上、函数上、成员函数上等等，这些用法本质上有什么共同之处？（3 分）

11. 完成以下计算:

```
unsigned int sum = 0;
for (unsigned int i = 0; i < 10; i++) {
  if (2 == ++i)
      continue;
      if (i == 5)
     break;
   if (i)
            sum++;
}
```

跳出循环后 sum=? （5 分）

12. 为什么指针能指向函数，并能通过指针动态调用函数（多态），请简述其原理（3 分）

13. 完成以下计算 （32 位编译器）

王道程序员

```
void func() {
    unsigned char tmp[10] = {0};
    tmp[0] = 0;
    tmp[1] = 1;
    tmp[2] = 2;
    tmp[3] = 3;

    unsigned char *p = tmp;
    p++;
    //*p=?   （2 分）

    unsigned short *sp = (unsigned short *)tmp;
    sp++;
    //*sp=?   （2 分）
}
```

14. 在 C++中引入模板，相比于 C，其用处/优势在于？（2 分）

15. 完成以下计算

```
u32 k = 7, m = 0;
switch (k) {
case 1:
        m++;
  break;
case 7:
  m = 1;
default :
  m = 0;
}
```

m=? （4 分）

16. 举例三种常见的内存泄漏/溢出情景（每个独立有效情景 2 分， 6 分）

17. 完成以下计算（32 位编译器）

牛道程序员

```
unsigned short a = 0, b;

void func(void)
{
  /* 此时 b=? */ （1 分）
  unsigned short a = 2;
  a += 2;
  /* a=? */   （1 分）

  b = 21;
  b /= 2;
  /* b=? */   （1 分）

  unsigned short c = 0x0001;
  c = c << 8;
  /* c=? */   （1 分）

  unsigned short d;
  /* d 的内存分配在哪里? 此时 d 的值为? */ （各 1 分）

  short tmp = -100;
  d = tmp;
  /* 此时 d 的值为? */ （1 分）

  unsigned char e = 10;
  e = e > 0 ? 1 : 0;
  /* 此时 e 的值为? */ （1 分）

  u8 f = 0x01;
```

18. 简述 C/C++编译过程，必须涉及 `.h`, `.c`, `.cpp`, `.o`, `.a`文件（5 分）

19. 英语测试二（18 分）

Design goals

There are myriads of JSON libraries out there, and each may even have its reason to exist. Our class had these design goals:

Intuitive syntax. In languages such as Python, JSON feels like a first class data type. We used all the operator magic of modern C++ to achieve the same feeling in your code. Check out the examples below and you'll know what I mean.

Trivial integration. Our whole code consists of a single header file json.hpp. That's it. No library, no subproject, no dependencies, no complex build system. The class is written in vanilla C++11. All in all, everything should require no adjustment of your compiler flags or project settings.

Serious testing. Our class is heavily unit-tested and covers 100% of the code, including all exceptional behavior. Furthermore, we checked with Valgrind and the Clang Sanitizers that there are no memory leaks. Google OSS-Fuzz additionally runs fuzz tests against all parsers 24/7, effectively executing billions of tests so far. To maintain high quality, the project is following the Core Infrastructure Initiative (CII) best practices.

Other aspects were not so important to us:

Memory efficiency. Each JSON object has an overhead of one pointer (the maximal size of a union) and one enumeration element (1 byte). The default generalization uses the following C++ data types: std::string for strings, int64_t, uint64_t or double for numbers, std::map for objects, std::vector for arrays, and bool for Booleans. However, you can template the generalized class basic_json to your needs.

Speed. There are certainly faster JSON libraries out there. However, if your goal is to speed up your development by adding JSON support with a single header, then this library is the way to go. If you know how to use a std::vector or std::map, you are already set.

请翻译此程序的任选三个功能特性，每个 6 分。

十、 嵌入式相关

嵌入式相关面试题

千道程序员