【C++】C语言基础部分知识点总结 （指针，函数，内存，关键字，预处理等）（秋招篇）

今天一定要洛必达 于 2023-08-10 15:01:27 发布

文章目录

前言

秋招笔记汇总篇之**C语言基础**
笔者是拿chatgpt写的，所以可能部分答案存在一定出路（3.5版本GPT有些缺陷），大部分答案我是写完了之后校正过一遍，有出入的地方还望各位同学指出。

2023.8.7 首次更新

讲一下32位系统常用数据类型的字节大小（stm32f103为例）

1）8位数据类型：

uint8_t（无符号8位整数）： 1字节

int8_t（有符号8位整数）： 1字节

char：1字节

2）16位数据类型：

uint16_t（无符号16位整数）：2字节

int16_t（有符号16位整数）：2字节

uint_fast16_t：2字节

int_fast16_t：2字节

3）32位数据类型：

uint32_t（无符号32位整数）：4字节

int32_t（有符号32位整数）：4字节

uint_fast32_t：4字节

int_fast32_t：4字节

float：4字节

4）64位数据类型（通常不适用于 STM32F103）：

uint64_t（无符号64位整数）：8字节

int64_t（有符号64位整数）：8字节

uint_fast64_t：8字节

int_fast64_t：8字节

double：8字节

讲一些C/C++中常见的库

stdio.h：提供了标准输入输出函数，如printf和scanf。

stdlib.h：提供了一些常用的函数，如内存分配和释放函数malloc和free。

unistd.h：提供了对POSIX操作系统API的访问，包括进程控制、文件操作等。

string.h：提供了字符串操作函数，如字符串复制函数strcpy和字符串比较函数strcmp。

arpa/inet.h：提供了一些与网络编程相关的函数和数据结构，如IP地址转换函数inet_pton和inet_ntop。
（在Windows平台上，网络编程相关的功能通常由**winsock2.h库**提供）

什么是易变变量?

“易变变量”（volatile variable）是一个在编程中使用的修饰符，用于告诉编译器该变量可能在程序的执行过程中发生不可预测的变化，因此编译器不应该进行某些优化，以确保对变量的访问和操作的正确性。
在C和C++中，编译器通常会对变量进行优化，以提高代码的执行效率。然而，有些情况下，变量的值可能会在程序执行期间被外部因素改变，这种情况下编译器的优化可能会引发问题。例如：

1）硬件寄存器访问：当变量实际上是与硬件寄存器相关联的，寄存器的值可能会在程序执行期间发生变化，但编译器可能不知道这种情况。

2）多线程环境：在多线程环境中，一个线程可能会修改另一个线程使用的变量，但编译器可能无法检测到这种可能性。

3）中断处理：在嵌入式系统中，中断可能会随时修改变量的值，但编译器不一定能够识别这种情况。

通过将变量声明为 volatile，编译器会知道这个变量的值可能会在未经通知的情况下发生变化，因此它不会进行某些优化，比如将变量的值缓存在寄存器中而不是从内存中读取。
在C语言中，使用 volatile 关键字声明易变变量，例如：

```
volatile int sensorValue;
```

在C++中，同样也可以使用 volatile 关键字，但在C++11之后，推荐使用 std::atomic 类或其他多线程编程机制来处理多线程环境中的易变数据。
需要注意的是，虽然 volatile 可以告诉编译器不进行特定的优化，但它并不能完全解决多线程或并发编程的问题。在多线程环境中，除了使用 volatile 外，还需要使用适当的同步机制来确保数据的正确性和一致性。

代码的转化和构建通常会经历哪几个步骤：（预处理，编译，汇编，链接）

在 C++ 程序的开发过程中，代码的转化和构建通常会经历以下几个步骤：

1）预处理（Preprocessing）：这是第一个阶段，预处理器会处理源代码中的预处理指令（以 # 开头的指令）。例如，#include 指令会将头文件的内容插入到源代码中，宏展开会将宏替换为其定义内容，条件编译会根据条件判断是否编译某段代码等。预处理之后生成一个被预处理后的源代码文件。

2）编译（Compilation）：在这一阶段，编译器会将预处理后的源代码翻译成汇编语言，这个汇编语言是特定于目标机器架构的中间表示。编译器会检查语法错误、类型错误和其他静态错误，并将代码转换为汇编指令。

3）汇编（Assembly）：汇编器会将汇编代码翻译成机器码指令，这些指令是可以被计算机硬件直接执行的指令。生成的是目标文件（通常以 .obj 或 .o 扩展名为后缀），其中包含了机器码指令、符号表等信息。

4）链接（Linking）：在多文件项目中，不同的源文件可能会产生多个目标文件。链接器会将这些目标文件以及可能的库文件合并在一起，解决符号引用（函数、变量等）的问题，生成可执行文件。这个过程包括地址分配、重定位、符号解析等步骤。

可执行文件生成：最终阶段，链接器生成可执行文件，该文件可以在操作系统上运行，并执行程序的功能。

介绍一下C/C++ 常用的关键字

C语言常用关键字：

break: 跳出循环或switch语句。

case: 在switch语句中标记不同的情况。

char: 声明字符型变量或数据类型。

const: 声明常量。

continue: 跳过当前循环中的剩余语句，进行下一次循环。

default: 在switch语句中定义默认情况。

do: 开始一个do-while循环。

double: 声明双精度浮点型变量或数据类型。

else: 条件不满足时执行的分支。

enum: 声明枚举类型。

extern: 声明外部变量。

float: 声明浮点型变量或数据类型。

for: 开始一个for循环。

goto: 跳转到指定标签处的代码位置。

if: 条件判断语句。

int: 声明整型变量或数据类型。

long: 声明长整型变量或数据类型。

register: 请求将变量存储在寄存器中（不一定会生效）。

return: 从函数中返回值。

short: 声明短整型变量或数据类型。

signed: 声明有符号类型。

sizeof: 返回数据类型或变量的字节大小。

static: 声明静态变量或函数，限制作用域。

struct: 声明结构体类型。

switch: 开始一个switch语句。

typedef: 定义类型别名。

unsigned: 声明无符号类型。

void: 声明无类型，通常用于函数的返回类型。

volatile: 声明易变变量，用于可能被优化器优化的变量。

while: 开始一个while循环。

C++语言常用关键字：

C++继承了C语言的关键字，还引入了一些新的关键字，用于支持面向对象编程和其他功能。

class: 声明类。

delete: 删除动态分配的对象。

explicit: 声明禁止隐式类型转换的构造函数。

friend: 声明友元函数或类。

inline: 声明内联函数。

namespace: 声明命名空间。

new: 动态分配内存。

operator: 声明重载运算符函数。

private: 类中的私有成员访问修饰符。

protected: 类中的受保护成员访问修饰符。

public: 类中的公共成员访问修饰符。

template: 声明模板。

this: 指向当前对象的指针。

throw: 抛出异常。

try: 异常处理的起始块。

catch: 捕获并处理异常。

typeid: 返回表达式的类型信息。

virtual: 声明虚函数或虚继承。

wchar_t: 宽字符数据类型。

介绍一些const的使用方法

在C和C++中，const 是用来定义常量的关键字，它可以应用于变量、函数参数、函数返回值等地方，用于表示数据的不可修改性。

常量变量声明：使用 const 关键字可以声明一个常量变量，其值在声明后不能被修改。

```
const int num = 10;
```

常量指针：使用 const 可以将指针声明为常量，这意味着指针指向的值不能通过该指针修改。

```
const int *ptr; // 常量指针，指向的值不可修改
int const *ptr; // 同样是常量指针的声明方式
```

指针常量：使用 const 可以将指针本身声明为常量，这意味着指针本身不能被修改，但可以通过指针修改其指向的值。

```
int value = 5;
int *const ptr = &value; // 指针常量，指针本身不可修改，但指向的值可修改
```

常量指针常量：指针和指向的值都不能被修改。

```
const int *const ptr = &value; // 常量指针常量，指针和指向的值都不可修改
```

函数参数中的const：将函数参数声明为 const 表示在函数内部不能修改该参数的值。

```
void printValue(const int num) {
    // num 只能被读取，不能被修改
}
```

函数返回值中的const：将函数的返回值声明为 const 表示调用者不能修改返回的值。

```
const int getValue() {
    return 42;
}
```

const 关键字的使用可以帮助编译器检测和防止对数据的不必要修改，从而增加程序的可靠性和安全性。注意，const 并不是绝对的保护，因为通过类型转换等手段，仍然有可能修改 const 数据。

#define const typedef三者区别

define、const、typedef 是在C和C++中用于定义、声明和创建别名的关键字，它们各自有不同的作用和用法。

1）define（宏定义）：
#define 是C和C++中的预处理指令，用于创建宏定义。宏定义可以用来在代码中替换文本，以达到代码复用和简化的目的。宏定义不会分配内存，只是在编译之前将指定的文本替换成宏的内容。

```
#define PI 3.14159
```

在这个例子中，所有的 PI 出现的地方都会被替换成 3.14159。

2）const（常量）：
const 是一个关键字，用于声明一个不可修改的变量，即常量。它可以用于修饰变量、指针、函数参数等，以确保其值不会被修改。

```
const int MAX_VALUE = 100;
```

在这个例子中，MAX_VALUE 被声明为一个常量，其值不能被修改。

3）typedef（类型定义）：
typedef 是用来创建类型别名的关键字。它可以用来定义一个已有类型的别名，使得代码更易读、更具可维护性。它通常用于定义复杂的类型，如结构体、指针等。

```
1 typedef int MyInt;
```

在这个例子中，MyInt 被定义为 int 的别名，以后可以使用 MyInt 来代替 int。

---

define、const、typedef 在许多方面都有不同的用途和特性。以下是它们在各个方面的区别：

1)作用范围：

define 是预处理指令，它在编译之前将指定的文本替换为宏的内容。它的作用范围是全局的，会影响文件中所有使用该宏的地方。

const 是关键字，用于声明一个常量，它的作用范围可以是变量、函数参数等。常量在程序运行时不可修改。

typedef 是用于创建类型别名的关键字，它可以在局部或全局范围内定义类型别名。

2）内存分配：

define 不涉及内存分配，它只是在编译前进行文本替换。

const 可以用于声明常量，它们会在内存中分配存储空间。

typedef 不涉及内存分配，它只是为已有类型创建一个别名。

3）类型检查：

define 不执行类型检查，它只是简单的文本替换。

const 可以用于声明常量，但它不会影响类型检查。

typedef 可以用于创建类型别名，它不会影响类型检查。

4）代码可读性：

define 可以用来创建宏，但宏展开后可能会使代码难以阅读和理解。

const 可以用于声明常量，它有助于提高代码的可读性和维护性。

typedef 可以用于创建类型别名，它有助于在代码中使用更具有描述性的类型名。

5）错误处理：

define 无法提供错误处理机制，因为它只是简单的文本替换，可能导致难以调试的问题。

const 声明的常量在编译时可以受到类型检查和错误处理机制的保护。

typedef 可以提高代码的清晰度，减少因类型错误引起的问题。

6）启动时间

#define在预处理阶段工作，剩下两个在编译阶段

链接的静态链接和动态链接分别是啥

链接是将多个目标文件合并成一个可执行文件或者共享库的过程。在链接的过程中，有两种常见的链接方式：静态链接和动态链接。

1）静态链接（Static Linking）：
静态链接是将所有的目标文件和库文件的代码、数据合并到一个单独的可执行文件中。在静态链接中，链接器会将程序使用到的所有库函数的代码都复制到最终生成的可执行文件中。这意味着可执行文件本身包含了所有需要的代码，因此它可以独立运行，不需要依赖外部的库文件。静态链接的优点是简单，不依赖于系统环境，但缺点是可执行文件会变得较大，占用更多的磁盘空间。

2）动态链接（Dynamic Linking）：
动态链接是在编译时和链接时，将程序所需的库文件的引用信息嵌入到可执行文件中，但实际的库函数代码会保留在磁盘上的共享库文件中（**如 .dll 或 .so**）。在程序运行时，操作系统会在内存中加载这些共享库，并将库函数代码映射到程序的地址空间中。动态链接的优点是可以减小可执行文件的大小，多个程序可以共享同一个库的实例，节省内存，同时库的更新只需要替换共享库文件而不需要重新编译整个程序。然而，动态链接需要系统中存在相应版本的共享库，否则程序无法运行。

总之，静态链接将所有的代码和数据都包含在可执行文件中，而动态链接只在运行时加载所需的库文件，使得程序更为灵活和节省资源。选择静态链接还是动态链接取决于项目的需求和优化策略。

常见的预处理指令有哪些

预处理指令是在编译源代码之前由预处理器处理的特殊命令。它们用于在编译过程中进行宏替换、条件编译、文件包含等操作。以下是一些常见的预处理指令：

#define：用于定义宏，可以是简单的文本替换，也可以带参数的宏。

#include：用于包含其他文件，通常是头文件，可以将其他文件的内容插入到当前文件中。

#ifdef 和 #ifndef：用于条件编译，根据宏的定义与否来决定是否编译一部分代码。

#else 和 #elif：用于在条件编译中，指定在条件不满足时或满足其他条件时要编译的代码。

#endif：用于结束条件编译块。

#undef：用于取消已定义的宏。

#pragma：用于发出特定编译器的指令，用于控制编译器的行为。

#error：用于在预处理时生成编译错误消息。

#warning：用于在预处理时生成编译警告消息。

defined() 函数样式宏：用于检查某个宏是否已定义。

预处理指令基本都不进行指令检查

什么是类型检查?

类型检查是编程语言中的一个重要概念，它涉及到在编译或运行时验证变量、表达式、函数参数等的数据类型是否符合预期。类型检查有助于捕获代码中可能出现的类型错误，提高代码的健壮性和可靠性。
编程语言通常分为静态类型语言和动态类型语言，类型检查在这两类语言中的行为有所不同：

静态类型语言（Static Typing）：
在静态类型语言中，**类型检查发生在编译阶段。**在编写代码时，你需要明确声明变量的数据类型，编译器会在编译过程中检查变量和表达式的数据类型是否一致。如果发现类型不匹配的错误，编译器会在编译时报告这些错误，从而避免了在运行时可能出现的类型相关问题。
例如，在C++中：

```
int x = 5;
double y = 3.14;
int result = x + y; // 编译时会报类型不匹配的错误
```

动态类型语言（Dynamic Typing）：
在动态类型语言中，类型检查通常发生在运行时。变量的数据类型可以在运行时根据赋值操作或表达式推断得出。因此，类型错误通常在实际执行代码时才会被发现。
例如，在Python中：

```
x = 5
y = 3.14
result = x + y  # 在运行时才会发现类型错误
```

类型检查的好处包括：

类型安全性：通过检查数据类型，可以防止不正确的类型操作，避免潜在的错误。

提前发现错误：静态类型检查可以在编译阶段捕获类型错误，避免在运行时产生不确定的行为。

增强代码可读性：明确的类型声明可以使代码更易于理解，降低误解和错误的可能性。

然而，严格的类型检查也可能导致一些灵活性的降低，因此在选择编程语言和处理类型检查时，需要根据项目需求和开发团队的偏好做出权衡。

define预处理指令存在什么问题?

#define 是C和C++中用于定义宏的预处理指令，它允许在代码中进行文本替换。虽然宏定义在某些情况下很有用，但也存在一些缺点：

没有类型检查： **宏是简单的文本替换，没有类型检查机制。如果宏的参数使用不当，可能会导致类型错误，但这些错误只能在编译时或运行时被发现。**

举个例子
当宏的参数使用不当时，可能会导致类型错误，而这些错误只能在编译时或运行时才会被发现。以下是一个例子来说明这一点：
假设有一个简单的宏定义，用于计算两个数的平方：

```
#define SQUARE(x) (x) * (x)

然后在代码中使用这个宏进行计算：
int main() {
    int result = SQUARE(5 + 3); // 预期计算结果是 64 (8 * 8)
    return 0;
}
```

在这个例子中，代码在宏调用中使用了一个表达式 5 + 3。根据宏的展开规则，宏定义会简单地进行文本替换，即 (5 + 3) * (5 + 3)。然而，这并不是预期的结果。
因为宏没有类型检查，它只是进行了简单的文本替换，上述代码实际上会被展开为：

```
int result = (5 + 3) * (5 + 3);
```

这会导致错误的结果，因为 (5 + 3) * (5 + 3) 实际上是 5 + 3 * 5 + 3，而不是预期的平方运算。这个错误只有在运行时才会被发现，导致了意外的计算结果。
相比之下，如果使用内联函数来实现相同的功能，就能避免这种类型错误：

```
inline int square(int x) {
    return x * x;
}

int main() {
    int result = square(5 + 3); // 正确计算结果为 64
    return 0;
}
```

在这个例子中，使用了内联函数 square 来实现平方计算，这样就能在编译时进行类型检查，避免了上述类型错误。这突显了宏的一个潜在问题，即没有类型检查机制，可能导致不易发现的错误。

可读性差： 复杂的宏定义可能会导致代码变得难以阅读和理解。在宏替换后，代码可能会变得冗长，导致难以跟踪代码逻辑。

调试困难： 宏展开后的代码在调试过程中可能会让人困惑。调试器显示的是展开后的代码，而不是宏定义本身，这可能使定位问题变得复杂。

命名空间污染： 宏是在全局命名空间中展开的，这可能导致宏名称与其他变量、函数等发生冲突，从而引起不必要的问题。

不方便调试： 由于宏是在预处理阶段展开的，因此在错误消息和警告中很难跟踪到原始宏定义的位置，这可能导致调试困难。

难以维护： 修改宏定义可能会涉及到多处使用该宏的地方，因此如果需要对宏进行修改，可能需要修改多处代码，增加了维护的复杂性。

可能引发副作用： 某些宏可能会在展开时多次计算其参数，这可能导致预期之外的副作用。

局限性： 宏定义只能进行简单的文本替换，无法实现一些复杂的编程概念，如函数调用、作用域等。

为了避免上述问题，现代C++通常推荐使用内联函数、常量、枚举等替代宏定义，因为这些更加类型安全、可读性更好，而且可以避免宏带来的很多问题。当然，在某些情况下，宏仍然是一种有效的工具，但在使用时应该慎重考虑其潜在的缺点。

面试中的#define 题 （让写几个标准宏函数）

在面试中，经常会遇到要求你写一些标准宏函数的题目，以考察你对预处理指令和宏的理解。以下是几个常见的标准宏函数示例：

最大值和最小值宏函数：

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

交换两个变量的值：

```
#define SWAP(a, b) do { \
    typeof(a) temp = (a); \
    (a) = (b); \
    (b) = temp; \
} while (0)
```

这里使用了 do { … } while (0) 结构，以确保宏在使用时不会出现意外的问题。

计算数组长度：

```
#define ARRAY_LENGTH(arr) (sizeof(arr) / sizeof((arr)[0]))
```


定义常量：

```
#define PI 3.14159265359
```

字符串化宏（将宏参数转换为字符串）：

```
#define STRINGIFY(x) #x
```

例如，STRINGIFY(test) 将会被替换为 “test”。

连接宏（将两个宏参数连接为一个标识符）：

```
#define CONCAT(a, b) a ## b
```

例如，CONCAT(prefix, _value) 将会被替换为 prefix_value。

讲一下头文件中常见的#ifndef #endif

在C和C++中，头文件（header file）是一种用于存放声明和定义的文件，通常用于存放函数、变量、类等的声明以及常量的定义。为了避免头文件被重复包含，防止重复定义和编译错误，经常会使用预处理指令来确保头文件只被包含一次。其中，常见的预处理指令包括 #ifndef、#define 和 #endif，它们一起用于创建所谓的“头文件保护”（header guards）。
头文件保护的作用是防止同一个头文件被重复包含，避免引发重复定义错误。下面是它们的用法和解释：

#ifndef：这个指令是 “if not defined” 的缩写，用于检查一个标识符是否已经被定义过。如果标识符未被定义，预处理器会执行接下来的代码，否则会跳过。

#define：这个指令用于定义一个标识符。通常在 #ifndef 后面使用，以确保标识符不存在时定义它。这个标识符通常是和头文件相关的唯一标识符，用于头文件保护。

#endif：这个指令用于结束一个条件编译块，对应于 #ifndef。在这个指令之后的代码会被正常处理。

举个例子，假设有一个头文件 myheader.h，它的内容如下：

```
#ifndef MYHEADER_H
#define MYHEADER_H

// 此处放置头文件的声明和定义

#endif
```

这里的 MYHEADER_H 是一个自定义的标识符，用来唯一标识这个头文件。当第一次包含 myheader.h 时，MYHEADER_H 尚未被定义，因此 #ifndef 会通过，进而 #define MYHEADER_H 定义了这个标识符。如果在其他地方再次包含 myheader.h，由于 MYHEADER_H 已经被定义，#ifndef 将不会通过，因此头文件的内容不会再次

被包含。
这样做可以防止由于头文件重复包含而引发的重复定义错误，确保每个头文件只被包含一次。这在大型项目中特别重要，因为它可以帮助管理代码结构和依赖关系。

举例介绍一下预处理指令#pragma

#pragma 是一种预处理指令，用于向编译器发出特定的指令或命令，以控制编译器的行为。它通常用于实现编译器特定的功能或优化，或者在特定情况下进行设置。然而，需要注意的是，#pragma 指令的行为可能会因编译器而异，因此在跨平台或跨编译器的项目中需要小心使用。
一个常见的使用场景是在编译网络相关代码时，特别是在使用 Winsock2 库（Windows套接字库）的情况下。Winsock2 是用于在 Windows 操作系统上进行网络编程的库，它需要一些特定的设置和初始化。
以下是一个可能的示例，展示如何在使用 Winsock2 时使用 #pragma 指令：

```c
#include <stdio.h>
#include <winsock2.h>

#pragma comment(lib, "ws2_32.lib") // 告诉编译器链接 ws2_32.lib 库

int main() {
    WSADATA wsaData;

    // 初始化 Winsock2 库
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        printf("Failed to initialize Winsock2.\n");
        return 1;
    }

    // 在这里进行网络编程操作

    // 清理并释放 Winsock2 库资源
    WSACleanup();

    return 0;
}
```

在这个例子中，#pragma comment(lib, “ws2_32.lib”) 用于告诉编译器链接 ws2_32.lib 库，这是使用 Winsock2 所必需的。这个指令的作用是在编译过程中自动将所需的库链接到生成的可执行文件中，而无需手动添加链接选项。
需要注意的是，#pragma 指令的行为可能因编译器而异，而且在可移植性要求高的情况下，最好避免过于依赖于特定的 #pragma 指令。在编写代码时，最好提供适当的注

释来解释 #pragma 指令的用途和目的，以便其他开发人员理解代码的含义。
这里也可以不写这条指令，直接在visual stdio里链接器里链接

举例介绍一下预处理指令#error

#error 是C和C++预处理器中的一个指令，用于在编译过程中生成错误消息并停止编译。它通常用于条件编译的情况下，以便在满足特定条件时发出错误信息。
以下是一个使用 #error 指令的示例：

```
#include <stdio.h>

#define DEBUG_MODE 0

#if DEBUG_MODE
    // 执行一些调试操作
#else
    #error "DEBUG_MODE is not defined! Please define it to enable debugging."
#endif

int main() {
    // 其他代码
    return 0;
}
```

在这个示例中，我们定义了一个名为 DEBUG_MODE 的宏，并根据其值进行条件编译。如果 DEBUG_MODE 的值为非零（即为真），则会执行一些调试操作。如果 DEBUG_MODE 未定义或其值为零，则会使用 #error 指令生成错误消息，并在编译时停止。
如果在编译上述代码时，DEBUG_MODE 没有被定义，编译器将会生成以下错误消息：
error: “DEBUG_MODE is not defined! Please define it to enable debugging.”

这样，开发人员就会在编译时得到明确的错误信息，提醒他们需要定义 DEBUG_MODE 宏以启用调试模式。
需要注意的是，#error 指令只是在预处理阶段生成错误消息，它并不会影响到编译生成的目标代码。因此，它通常用于在预处理期间检查条件，并在需要时提供有用的错误提示。还有什么我可以帮助你的吗?

讲一下静态变量和静态函数

静态变量和静态函数是 C++ 程序中的两种特殊类型。**它们的作用范围在于当前文件内，即在同一个源文件中。**

1）静态变量（Static Variables）：
静态变量是在函数内部或者在全局作用域内使用 static 关键字声明的变量。静态变量在程序的整个生命周期内都存在，但作用范围仅限于声明它的函数内部（如果在函数内部声明）或者当前源文件内的全局作用域（如果在全局作用域内声明）。

在函数内部声明的静态变量将会在函数调用时被初始化，但只会在第一次进入函数时被初始化一次。之后每次函数调用时，该变量的值会保持上一次函数调用结束时的值。这在某些需要保留状态的情况下很有用。
在全局作用域内声明的静态变量将在程序运行时初始化，并且只能在当前源文件内使用。这意味着其他源文件无法直接访问该静态变量。

```
void foo() {
    static int counter = 0; // 静态局部变量
    counter++;
}

static int globalStaticVar = 5; // 静态全局变量，只在当前源文件内可见
```

2）静态函数（Static Functions）：
静态函数是在函数声明前加上 static 关键字的函数。静态函数与静态变量类似，也是只在当前源文件内可见的。静态函数的作用范围仅限于当前源文件，其他源文件无法直接调用它。

使用静态函数的主要优点是可以将函数的作用范围限制在当前源文件，避免与其他文件中的函数命名冲突，并且可以提高代码的模块性和封装性。

```
static void staticFunction() {
    // 这是一个静态函数，只在当前源文件内可见
}
```

总之，静态变量和静态函数都在作用范围上有限制，只在当前文件内有效。这些特性可以在一定程度上帮助组织代码并控制作用域。

将变量和函数声明为静态的，限制它们的作用范围在当前文件内（同一个源文件中）带来了一些好处和用途：

1）模块性和封装性：通过将变量和函数的作用范围限制在单个源文件内部，可以实现更好的模块性和封装性。每个源文件可以看作一个独立的模块，只暴露必要的接口给其他模块使用，减少了不同模块之间的直接依赖，从而提高了代码的可维护性和可扩展性。

2）避免命名冲突：静态变量和静态函数的作用范围被限制在当前文件内，这意味着你可以在不同的文件中使用相同的变量名或函数名，而不会产生命名冲突。每个文件内的命名都相对独立，减少了全局命名空间中的潜在冲突。

3）信息隐藏：静态变量和静态函数只在当前文件内可见，不会被其他文件直接访问。这种机制可以隐藏实现细节，避免外部代码直接操作内部状态，从而提高了代码的安全性和稳定性。

4）代码优化：编译器在优化时可以更容易地对局部作用域内的代码进行分析和优化，因为它知道这些变量和函数不会被其他文件引用，从而提高了代码的执行效率。

5）控制访问权限：静态变量和静态函数可以通过将它们声明在私有的类或命名空间中，实现对外部的访问控制。这有助于实现面向对象编程中的封装和信息隐藏。

需要注意的是，虽然静态变量和静态函数在当前文件内有效，但并不适用于多个源文件之间的通信。如果需要在不同的源文件之间共享变量或函数，就需要使用 extern 关键字来声明外部变量或函数。静态变量和静态函数主要用于在单个源文件内部组织代码和管理作用域。

const结合静态变量和静态函数

const 可以结合静态变量和静态函数使用，用于表达对它们的属性和行为的约束。在C和C++中，静态变量和静态函数在不同的上下文中具有不同的含义，下面我将分别解释如何使用 const 来结合它们。

1）静态变量 + const：

```
void func() {
    static const int value = 10;
    // value 是一个静态常量变量，在多次调用 func() 时值不会改变
}
```

这里的 value 是一个静态常量变量，意味着它只会在第一次调用 func() 时初始化，然后在后续调用中保持不变。const 保证了它的值不会在函数执行期间被修改。**只在声明所在作用域内可见**

2）静态函数 + const：

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

static int multiply(int a, int b) {
    return a * b;
}

static const int (*operation)(int, int) = add;
// operation 是一个指向静态函数的指针，指向 add 函数
```

在这个示例中，operation 是一个指向静态函数的指针，通过将其声明为指向函数的指针，我们可以在运行时选择要调用的函数。const 在这里表示，通过 operation 指针不能修改函数的内容。
const 关键字在这里的作用是限制通过函数指针修改函数代码，而不限制调用函数本身。

讲一下extern关键字

extern 是 C++ 编程语言中的一个关键字，用于声明外部变量和函数。它告诉编译器某个变量或函数是在其他文件中定义的，并且在当前文件中只是一个声明，实际的定义在其他地方。**extern 关键字在链接阶段起作用**，帮助编译器和链接器正确处理在不同文件中定义和引用的标识符，以确保程序能够正确地运行和链接。
主要用途有两个：

1）声明外部变量：
当你需要在一个源文件中引用另一个源文件中定义的全局变量时，可以使用 extern 关键字在当前文件中进行声明，以便编译器知道该变量是在其他地方定义的。这样，编译器就不会报告变量未定义的错误。

```
// File1.cpp
int globalVar = 42;

// File2.cpp
extern int globalVar; // 声明外部变量
int main() {
    // 在这里可以使用 globalVar
    return 0;
}
```

2）声明外部函数：
当你需要在一个源文件中调用另一个源文件中定义的函数时，可以使用 extern 关键字在当前文件中声明该函数，以便编译器知道函数是在其他地方定义的。

```
// MathFunctions.cpp
int add(int a, int b) {
    return a + b;
}

// Main.cpp
extern int add(int a, int b); // 声明外部函数
int main() {
    int result = add(3, 5); // 调用另一个文件中的函数
    return 0;
}
```

总之，extern 关键字用于在一个文件中声明在其他地方定义的变量或函数，使得不同的源文件可以协同工作，避免了重复定义和重复声明的问题。它在 C++ 程序中起到了连接不同源文件的作用。

指针的大小是?

在大多数现代计算机体系结构中，指针的大小（即占用的字节数）通常取决于计算机的体系结构和操作系统的位数。
在一个典型的32位操作系统中，指针的大小通常为**4字节（32位）**。这意味着指针可以存储一个32位的内存地址，用于访问内存中的数据。
在一个64位操作系统中，指针的大小通常为**8字节（64位**）。这样的指针可以存储一个64位的内存地址，使计算机能够在更大的内存范围内寻址。

指针p++,其值怎么变化的

假设我们有一个 int 类型的数组，我们将一个指针 p 指向数组的第一个元素，然后执行 p++ 操作，观察地址值的变化。
假设数组起始地址为 0x1000，int 类型占用4字节。

初始状态：
数组起始地址: 0x1000
p 指向地址: 0x1000
*p 的值: 42 （这里假设数组第一个元素的值为 42）

执行 p++ 操作后：
数组起始地址: 0x1000
p 指向地址: 0x1004 （增加了 4 个字节，即一个 int 类型的大小）
*p 的值: 57 （这里假设数组第二个元素的值为 57）

介绍一下指针之间的加减法

假设有两个指针 ptr1 和 ptr2，可以进行如下操作：

```
int arr[] = {1, 2, 3, 4, 5};
int *ptr1 = &arr[1]; // ptr1指向数组的第二个元素（值为2）
int *ptr2 = &arr[4]; // ptr2指向数组的最后一个元素（值为5）

ptrdiff_t diff = ptr2 - ptr1; // 计算ptr2和ptr1之间的偏移量
printf("Offset between ptr2 and ptr1: %td\n", diff);
```

在上述示例中，ptrdiff_t 是用来保存指针之间偏移量的整数类型，它可以是有符号整数类型，用于确保能够表示负的偏移量。
需要注意的是，进行指针之间的加法和减法运算的两个指针必须指向同一个数组（或数组的最后一个元素的下一个位置），否则结果将是未定义的。这种操作常用于计算在数组中的元素之间的距离，以及在动态分配的内存块中计算元素的偏移量等情况。
在C和C++中，指针之间的加法是不允许的，因为指针加上另一个指针没有定义的意义。指针的加法运算只能与一个整数值进行，而不是与另一个指针进行。

讲一下指针常量和常量指针的区别

在 C/C++ 编程中，指针常量和常量指针是两个不同的概念。它们分别涉及到指针本身的不可变性和指针所指向的内容的不可变性。

1）指针常量（Pointer to Constant）：
指针常量是一个指针，其指向的值（也就是指针所指向的内存地址）是不可变的。这意味着一旦指针被初始化，它将始终指向同一个地址，无法通过这个指针来修改所指向的值。

```
const int *ptr;  // ptr 是一个指向常量的指针
```

在这个例子中，ptr 是一个指向整数常量的指针。这表示您不能通过 ptr 来修改所指向的整数的值，但可以改变 ptr 指向的位置。

2）常量指针（Constant Pointer）：
常量指针是一个指针，其本身的值（即存储的内存地址）是不可变的。这意味着一旦指针被初始化，它将始终指向同一个地址，无法通过这个指针来修改其指向的位置。

int x = 10;

```
int *const ptr = &x;  // ptr 是一个常量指针
```

在这个例子中，ptr 是一个常量指针，它指向整数 x。这表示您不能通过 ptr 来改变它所指向的位置，但可以修改 x 的值。
综上所述，指针常量注重指针所指向的值的不可变性，而常量指针注重指针本身的不可变性。这两个概念可以结合使用，例如 const int *const ptr 表示一个即不可修改指针本身，也不可修改所指向值的指针。

讲一下野指针和空指针，它们怎么发生的

野指针（Dangling Pointer）和空指针（Null Pointer）是指针在不同状态下的两个概念。

1）野指针（Dangling Pointer）：
野指针是指一个指针仍然存在，但它指向的内存地址已经无效或者已经被释放。这种情况可能发生在以下几种情况下：

指针指向的内存被释放，但指针本身没有被置为 NULL。

指针指向的局部变量在其作用域结束后被销毁，但指针没有被及时置为 NULL 或重新分配。

使用野指针可能会导致不可预测的结果，包括程序崩溃或错误的数据访问。为了避免野指针问题，建议在释放内存后将指针设置为 NULL。

2）空指针（Null Pointer）：
空指针是指针变量明确地指向空地址，即没有有效的内存地址与之关联。在 C/C++ 中，通常使用宏定义 NULL 或关键字 nullptr 来表示空指针。空指针在一些情况下是有用的，特别是在以下情况下：

初始化指针，以防止野指针的问题。

检查指针是否已经被初始化，避免在未初始化状态下使用。

在 C 中，使用 NULL 表示空指针：

```
int *ptr = NULL;  // ptr 是一个空指针
```

在 C++11 及以后的标准中，使用 nullptr 表示空指针：

```
int *ptr = nullptr;  // ptr 是一个空指针
```

总结：

野指针是指向无效内存的指针，可能导致不可预测的行为。

空指针是明确指向空地址的指针，可以用来避免未初始化指针的问题。

1）野指针的发生：

指针指向已释放的内存： 当您释放了一块内存后，但忘记将指向该内存的指针置为 NULL，该指针就会成为野指针。

指针指向超出作用域的局部变量： 如果一个指针指向了一个已经超出其作用域的局部变量，那么当您尝试访问该指针时，它会成为野指针，因为局部变量的内存已经无效。

指针引用了一个临时变量： 如果一个指针引用了一个临时变量（例如函数返回的临时对象），当临时变量超出作用域时，指针将成为野指针。

指针操作不当： 在进行指针操作时，如果没有正确更新指针的值，可能会导致指针变成野指针。

2）空指针的发生：

未初始化的指针： 如果一个指针没有被初始化，它的值可能是随机的，可能指向任意内存地址，这就是空指针。

显式设置为空指针： 有时您可能会将指针显式设置为 NULL 或 nullptr，以确保它不指向任何有效的内存。

介绍一下函数指针

函数指针是C和C++中的一种特殊类型的指针，**它允许你存储和操作函数的地址**。通过函数指针，你可以将函数作为参数传递给其他函数、在运行时选择要调用的函数，或者实现回调机制等。以下是关于函数指针的一些基本概念和用法：

函数指针的声明和定义：
函数指针的声明形式类似于函数原型，但将函数名替换为指针变量名，同时需要指定函数的参数列表和返回类型。
returnType (*pointerName)(parameterType1, parameterType2, …);

例如，声明一个可以指向返回整数的函数的指针：

```
int (*funcPtr)(int, int);
```

初始化函数指针：
函数指针可以被初始化为指向一个具体的函数。这需要注意函数的签名必须与函数指针声明的类型匹配。

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

int (*funcPtr)(int, int) = add;  // 初始化函数指针指向 add 函数
```

使用函数指针调用函数：
通过函数指针可以调用指向的函数，就像调用函数本身一样。只需像调用函数一样使用指针名称，并传入所需的参数。

```
int result = funcPtr(3, 5);  // 调用 add 函数，返回 8
```

函数指针作为函数参数：
函数指针常用于将函数作为参数传递给其他函数，这在实现回调机制时非常有用。

```
void performOperation(int (*operation)(int, int), int a, int b) {
    int result = operation(a, b);
    printf("Result: %d\n", result);
}

performOperation(add, 3, 5);  // 调用 performOperation，传递 add 函数
```

函数指针在编写高度灵活的代码、实现插件系统、动态选择函数等方面非常有用。它们允许在运行时根据需要选择不同的函数实现，从而提高代码的可扩展性和适应性。

介绍一下二级指针和应用场景

二级指针（Double Pointer）是指一个指向指针的指针，也称为指向指针的指针。它在一些编程语言中用于处理复杂的数据结构和操作，特别是在涉及动态内存分配和多级指针的情况下。
让我们通过一个简单的例子来理解二级指针的概念。假设我们有一个整数指针 p，它指向一个整数变量，而我们又有一个指向指针 p 的二级指针 pp。这就是一个二级指针的示例。

```
int x = 42;
int *p = &x;
int **pp = &p;
```


在这个示例中：

x 是一个整数变量，存储了值 42。

p 是一个指向整数的指针，指向变量 x 的地址。

pp 是一个指向指针 p 的指针，也就是一个二级指针。它指向指针 p 的地址。

我们可以通过二级指针来访问和修改指针 p 的值，进而访问和修改变量 x 的值。示例如下：

```
printf("x = %d\n", **pp);  // 输出 x 的值，即 42
**pp = 57;  // 修改 x 的值为 57

printf("x = %d\n", *p);   // 输出修改后的 x 的值，即 57
```

在实际编程中，二级指针常常用于以下情况：

多级数据结构，例如链表的头指针，树的节点等。
动态内存分配和释放，例如在函数内部分配内存，然后通过二级指针将分配的内存传递给调用者。
在函数中修改指针变量的值，以便在函数外部保留修改后的指针值。

字符串和字符数组的关系

字符串（String）和字符数组（Character Array）在C和C++中是密切相关的概念，但它们有一些不同之处。

字符串： 字符串是由一系列字符组成的序列，以空字符 '\0'（ASCII码为0）作为结尾。C和C++中的字符串实际上是字符数组，只不过最后一个字符是空字符，用于标识字符串的结束。字符串在C语言中没有作为一种原生的数据类型存在，而是通过字符数组来表示。

字符数组： 字符数组是一组连续的字符元素，存储在内存中的相邻位置。字符数组可以用于存储字符串，其中最后一个字符是空字符，以便标识字符串的结束。

以下是一个简单的例子，展示了字符串和字符数组的关系：

```
#include <stdio.h>

int main() {
    // 字符数组的初始化
    char charArray[] = {'H', 'e', 'l', 'l', 'o', '\0'};

    // 字符串的初始化
    char str[] = "Hello";

    printf("charArray: %s\n", charArray);
    printf("str: %s\n", str);
```

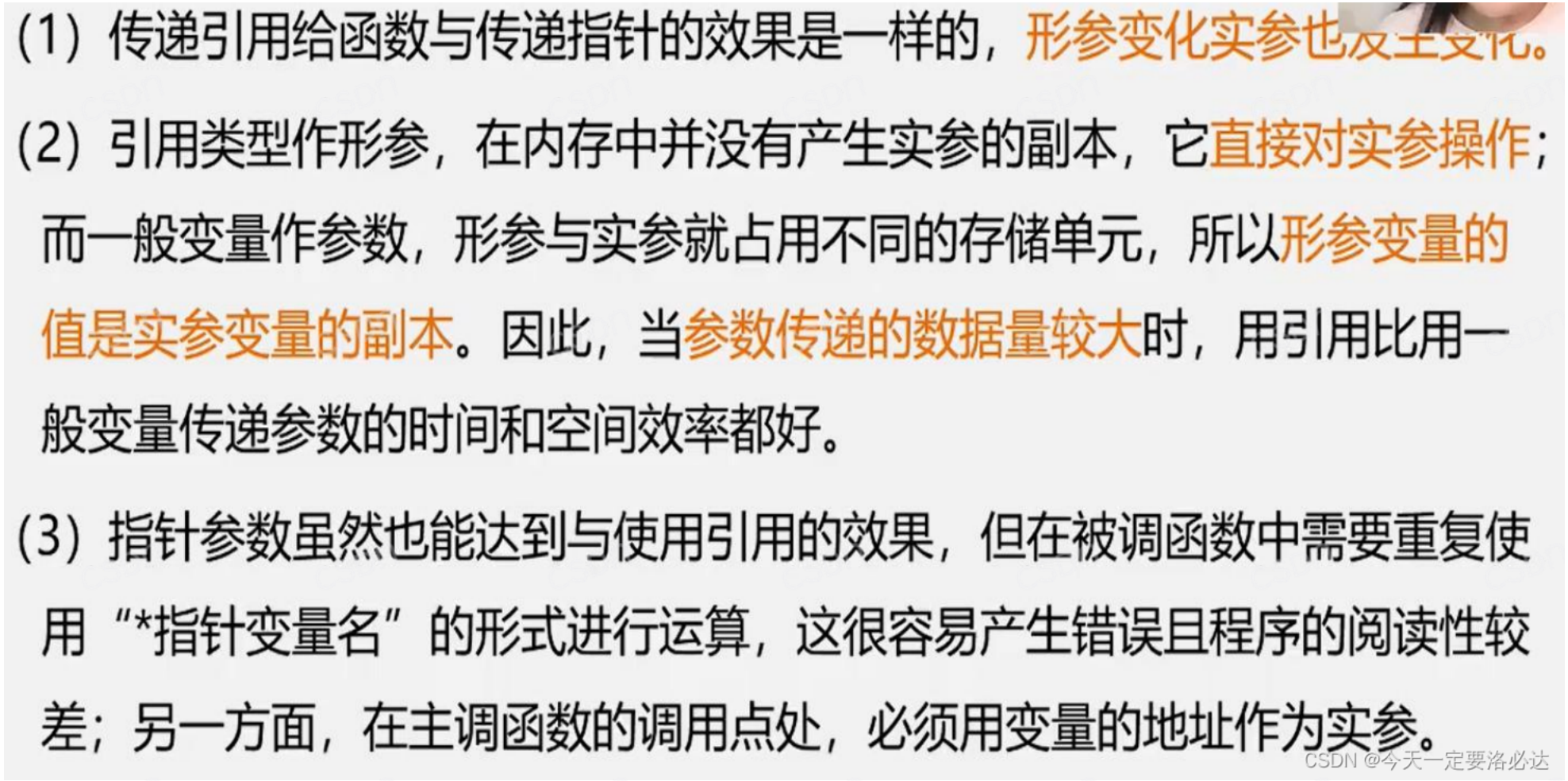
```
12
13      return 0;
14  }
```

在这个例子中，charArray 和 str 都是字符数组，存储了相同的字符串 “Hello”，但是用法稍有不同。str 在初始化时使用了字符串字面值，编译器会自动在其末尾添加一个空字符，而 charArray 需要手动添加空字符来表示字符串的结束。
需要注意的是，字符串是以空字符结尾的，因此在操作字符串时要注意空字符的存在。在C和C++中，有许多用于操作字符串的标准库函数，如 strlen、strcpy、strcmp 等。这些函数可以用来处理字符数组，从而实现对字符串的操作。
字符串本质上是字符数组当我们处理字符串时，实际上是在操作一个字符数组，只不过该数组以空字符结尾，以便我们能够识别字符串的结束

C/C++中参数传递的方式有几种?

参数传递有2种方式：
1）传值
2）传地址：包括传引用（by reference）和传指针（by pointer），传数组名等。

优缺点：参考来自数据结构 青岛大学



数组作为函数参数，有几种写法?

数组名在大多数情况下被视为指向数组首元素的指针。在将数组作为函数参数传递时，有以下几种写法：

1）传递数组的指针：
在函数参数中，可以将数组名作为指针传递，函数将接收一个指向数组首元素的指针。这是最常见的方式，但是函数无法获知数组的大小，因此需要额外传递数组的大小作为参数。

```
1
2 void function(int arr[], int size) {
3     // 使用 arr 来访问数组元素
4 }
5
```

```
int main() {
    int myArray[5] = {1, 2, 3, 4, 5};
    function(myArray, 5);
    return 0;
}
```

2）使用指针表示数组：
可以使用指针形式的数组来传递，这样就可以不必传递大小信息。这种情况下，函数需要知道数组的结束标志，例如在 C 字符串中使用的 NULL 终止符。

```
void function(int *arr) {
    // 使用 arr 来访问数组元素，直到遇到某个终止条件
}

int main() {
    int myArray[5] = {1, 2, 3, 4, 5};
    function(myArray);
    return 0;
}
```

直接使用引用方式传递数组（&）可能会导致失去数组的大小信息，从而不方便在函数内部访问数组的元素。
但是在力扣（LeetCode）等编程竞赛和面试中，使用引用方式将向量（vector）作为函数参数传递

```
void modifyVector(std::vector<int>& vec) {
    for (int i = 0; i < vec.size(); ++i) {
        vec[i] *= 2; // 修改向量元素的值
    }
}
```

do while(0)有何优势?

do while(0) 是一种常见的编程技巧，它的主要优势在于它可以创建一个代码块，使得可以在一个语句中执行多个操作，同时又能保持代码的结构清晰和可读性。
以下是 do while(0) 的一些优势：

可以在一个语句中执行多个操作：do while(0) 可以将多个语句组合在一个代码块中，这样可以在一个地方执行多个操作，而不需要使用额外的函数或条件语句。

保持代码结构清晰和可读性：使用 do while(0) 可以避免使用大括号来创建代码块，从而使代码结构更加简洁和清晰。这样可以提高代码的可读性，减少错误和调试的难度。

方便使用宏定义：do while(0) 通常与宏定义一起使用，可以方便地定义复杂的宏，而不会引起语法错误。如果不这么使用的话，在宏展开时，可能会出现问题。使用 do while(0) 可以确保宏定义的正确性，并且可以在宏定义中使用多个语句。

eg:

```c
#include <stdio.h>

#define PRINT_NUM(x) printf("The number is: %d\n", x) \
                     printf("This is a complex macro.\n")

int main() {
    int num = 10;
    PRINT_NUM(num);

    return 0;
}
```

在这个例子中，我们定义了一个名为 PRINT_NUM 的宏，它包含了两个 printf 语句。然而，如果我们尝试编译这段代码，会得到以下错误：
error: expected ‘;’ before ‘printf’
printf(“This is a complex macro.\n”)

这是因为宏定义没有将多个语句组合在一个代码块中，导致在宏展开时缺少了分号。使用 do while(0) 可以解决这个问题，确保宏定义的正确性。

```c
#include <stdio.h>

#define PRINT_NUM(x) do { \
                        printf("The number is: %d\n", x); \
                        printf("This is a complex macro.\n"); \
                     } while(0)

int main() {
    int num = 10;
    PRINT_NUM(num);

    return 0;
}
```

swap(a,b)和swap(&a,&b)的区别

swap(a, b) 和 swap(&a, &b) 的区别在于参数的传递方式。

swap(a, b)：这种方式将变量 a 和 b 的值作为参数进行传递。在函数内部，使用的是参数的副本，即函数会创建 a 和 b 的副本，并对副本进行操作。**这样，函数内部的交换操作不会影响到原始的 a 和 b 变量。**

swap(&a, &b)：这种方式将变量 a 和 b 的地址作为参数进行传递。在函数内部，使用指针来操作传入的地址，从而直接修改原始的 a 和 b 变量的值。**这样，函数内部的交换操作会影响到原始的 a 和 b 变量。**

总结起来，swap(a, b) 是值传递，函数内部操作的是参数的副本，而 swap(&a, &b) 是地址传递，函数内部直接操作原始变量的地址。因此，通过 swap(&a, &b) 可以实现对原始变量的直接修改，而 swap(a, b) 则不会改变原始变量的值。

i++ 和++i 的区别

i++ 和 ++i 都会将 i 的值增加 1，但是它们的主要区别在于它们的"副作用"发生的时间和返回值。
i++ 是后置递增运算符。它首先返回 i 的当前值，然后再将 i 的值增加 1。例如，如果 i 的初始值为 5，那么表达式 j = i++ 会将 j 的值设置为 5，然后再将 i 的值增加 1。

```
int i = 5;
int j = i++;  // j becomes 5, i becomes 6
```

而 ++i 是前置递增运算符。它首先将 i 的值增加 1，然后返回新的 i 值。例如，如果 i 的初始值为 5，那么表达式 j = ++i 会将 i 的值增加 1，然后将 j 的值设置为 6。

```
**int i = 5;
int j = ++i;  // i and j both become 6**
```

简单来说，如果你只是想增加 i 的值，并不关心递增操作发生前后的值，那么 i++ 和 ++i 的效果是一样的。但是，如果你在一个表达式中使用了它们，并且对递增操作的执行时间有所依赖，那么就需要注意它们之间的这个区别了。
此外，在一些情况下，++i 可能比 i++ 更有效率，因为 i++ 需要创建一个临时变量来存储原始的 i 值。然而，现代编译器通常会对这种情况进行优化，所以在实践中这个差异可能并不明显。

if ， if， if 和 if， else if， else 的区别

if if if 和 if else if else 在逻辑上有明显的区别，它们分别是：

if if if:
这是多个连续的独立的 if 语句，每个 if 都会独立地进行条件判断和执行。**这意味着无论前面的 if 是否执行，后面的 if 都会被尝试执行。**每个 if 语句都是独立的条件分支。

例如：

```
int x = 10;

if (x > 5)
    cout << "x is greater than 5" << endl;

if (x > 7)
    cout << "x is greater than 7" << endl;
```

```
8
9  if (x > 9)
10     cout << "x is greater than 9" << endl;
```

如果 x 的值是 10，以上代码将输出：
x is greater than 5
x is greater than 7
x is greater than 9

if else if else:
这是多个连续的条件语句组成的链式条件结构。每个条件会依次被检查，**只有满足条件的第一个分支会被执行**，而其它分支将被忽略。这个结构允许在多个条件中选择一个满足的分支。

例如：

```
1  int x = 10;
2
3  if (x > 9)
4      cout << "x is greater than 9" << endl;
5  else if (x > 7)
6      cout << "x is greater than 7" << endl;
7  else if (x > 5)
8      cout << "x is greater than 5" << endl;
9
10 如果 x 的值是 10，以上代码将输出：
11 x is greater than 9
```

这是因为只有第一个条件 x > 9 成立，因此只有第一个分支被执行，后续的 else if 分支都不会被考虑。
综上所述，if if if 是独立的多个条件分支，而 if else if else 是链式条件结构，只有满足的第一个条件分支会被执行。你可以根据具体的逻辑需求选择使用哪种结构。

介绍一下内联函数和宏函数，同时讲一下它们的区别

内联函数（Inline Function）：

内联函数是一种在编程中用于优化代码执行效率的方法。**它通过将函数调用处的代码直接嵌入到调用位置，避免了函数调用的开销，不用寻址直接展开** 内联函数通常适用于函数体较短的情况，因为代码膨胀可能会导致可读性和维护性的问题。
在C++中，你可以使用 inline 关键字来声明内联函数。例如：

```
1  inline int add(int a, int b) {
2      return a + b;
```

```
}
```

在这个例子中，add 函数被声明为内联函数。编译器会在调用 add 函数时，将函数体的代码直接插入到调用位置。这样可以减少函数调用的开销，提高执行效率。
宏函数（Macro Function）：

宏函数是一种在预处理阶段展开的替代文本。通过宏，你可以在代码中定义一些简单的代码片段，并在代码编译前将它们替换成对应的文本。宏函数在代码展开时不进行类型检查，因此需要特别小心以避免潜在的错误。
在C/C++中，你可以使用 #define 来定义宏函数。例如：

```
#define ADD(a, b) (a + b)
```

在这个例子中，ADD 被定义为一个宏函数，它会在代码展开时将 (a + b) 替换为实际的参数相加。需要注意的是，宏函数会简单地进行文本替换，不会进行类型检查。
区别总结：

1）内联函数是编译器在编译时展开的，而宏函数是在预处理阶段展开的。

2）内联函数进行类型检查，宏函数不进行类型检查。

3）内联函数可以进行调试，宏函数在展开时难以进行调试。

4）内联函数可能导致代码膨胀，宏函数也可能导致文本替换导致的代码膨胀。

5）内联函数适合较短的函数体，宏函数适合简单的代码片段。

内联函数本质上是一个函数 宏函数本质上宏定义

介绍一下C语言和C++中IO流的函数

在C语言和C++中，I/O（输入输出）操作是非常重要的，用于从程序中获取输入和将输出显示给用户或其他设备。在C和C++中，I/O操作通常使用函数来实现。下面我将为你介绍一些常用的C语言和C++中的I/O函数。
C语言中的I/O函数：

printf()：用于格式化输出数据到标准输出（通常是终端窗口）。

scanf()：用于从标准输入（通常是键盘）读取格式化的数据。

getchar() 和 putchar()：分别用于从标准输入读取一个字符和向标准输出输出一个字符。

gets() 和 puts()：分别用于从标准输入读取一行字符串和向标准输出输出一行字符串。gets() 存在安全性问题，应该避免使用。

C++中的I/O流函数（使用iostream库）：

C++引入了更为灵活和面向对象的I/O机制，使用iostream库进行输入输出操作。这里是一些常用的函数：

cin：用于从标准输入读取数据。

cout：用于向标准输出输出数据。

cerr：用于输出错误信息到标准错误输出（通常也是终端窗口），不会缓冲。

clog：类似于cerr，但是会缓冲输出。

getline()：从输入流中读取一行字符串。

setw()：用于设置输出字段的宽度。

setprecision()：用于设置浮点数输出的精度。

ifstream：用于读取文件。

ofstream：用于写入文件。

fstream：用于同时读写文件。

这些函数和流提供了更高层次的抽象，允许更容易地进行格式化输出和输入，并且对于面向对象的编程来说非常有用。
请注意，这只是一个简要的概述，实际使用时可以查阅相关的文档以获取更多详细信息。

讲一下typeid函数

typeid 是C++中的一个操作符，用于获取表达式的类型信息。它通常用于运行时获取对象或表达式的类型，以便进行类型检查或其他操作。typeid 主要用于运行时的类型识别和类型比较。
以下是一个简单的使用示例：

```
#include <iostream>
#include <typeinfo>

class Base {
    virtual void foo() {} // 为了演示多态
};

class Derived : public Base {
};

int main() {
    int i = 42;
```

```
    double d = 3.14;
    std::string str = "Hello";
    Base base;
    Derived derived;

    std::cout << "i的类型: " << typeid(i).name() << std::endl;
    std::cout << "d的类型: " << typeid(d).name() << std::endl;
    std::cout << "str的类型: " << typeid(str).name() << std::endl;
    std::cout << "base的类型: " << typeid(base).name() << std::endl;
    std::cout << "derived的类型: " << typeid(derived).name() << std::endl;

    return 0;
}
```

在这个示例中，我们使用了 typeid 来获取不同变量和对象的类型信息。需要注意的是，typeid 返回的类型信息可能不是人类可读的名称，因为它取决于编译器的实现。在不同的编译器和平台上，可能会有不同的类型名表示。
除了获取类型信息外，typeid 还可以用于比较两个类型是否相同。例如：

```
if (typeid(obj1) == typeid(obj2)) {
    // obj1和obj2的类型相同
}
```

请注意，typeid 通常在多态情况下才会用到，因为它在基类和派生类之间的类型比较上特别有用。

讲一下C++中的内存四区

当谈论C++中的内存四区时，通常指的是堆、栈、全局区（也称为数据区或BSS段）、以及代码区（也称为文本区或代码段）。这些区域在程序的执行和内存管理中扮演着不同的角色。以下是它们的具体解释：

1）堆（Heap）：

功能：**用于动态分配内存，以供在运行时管理对象的存储。**

内存分配：通过运行时的内存分配函数，如**new和malloc**等，来在堆上分配内存。

内存释放：手动调用delete和free来释放已分配的堆内存。如果没有释放，会导致内存泄漏。

特点：堆内存的分配和释放可能相对较慢，适用于需要灵活管理大小和生命周期的数据。

2）栈（Stack）：

功能：**用于管理函数调用的局部变量、函数参数和返回地址等信息。**

内存分配：在函数调用时自动分配，函数调用结束后自动释放。

特点：栈内存分配和释放非常高效，适用于快速的函数调用和局部变量。

3）全局区/数据区（Global/Static Memory）：

功能：存储全局变量、静态变量以及初始化为零或未初始化的静态数据。

内存分配：在程序启动时分配，全局和静态变量在整个程序生命周期内存在。

特点：全局和静态变量的作用域和生命周期长，适用于需要全局访问的数据。

一些系统可能会将全局变量和静态变量分为不同的段，比如"BSS段"（存储未初始化的全局变量和静态变量）和"数据段"（存储已初始化的全局变量和静态变量）。

4）代码区/代码段（Code Area/Text Segment）：

功能：存储程序的机器码指令，即可执行代码。

内存分配：在程序加载到内存时由操作系统进行分配。

特点：代码区是只读的，包含程序的执行逻辑，但不能被修改。函数的机器码指令存储在这里，供程序执行。

这个分区方式是按黑马程序员视频里来的

new和malloc函数用法介绍一下（函数原型和返回值等）

new 和 malloc **分别是在C++和C语言**中用于动态分配内存的函数，但它们有一些区别。下面是它们的用法介绍以及函数原型和返回值：
1）new 操作符：

new 是C++中的操作符，用于在堆上动态分配内存，并返回指向新分配内存的指针。它还可以自动调用构造函数来初始化对象。
用法示例：

```
int *ptr = new int; // 分配一个整数大小的内存，并将指针指向它
double *arr = new double[10]; // 分配一个包含10个双精度浮点数的数组
```


函数原型：

```
new type;        // 分配一个 type 类型的内存并返回指向它的指针
new type[size];  // 分配一个包含 size 个 type 类型元素的数组并返回指针
```

注意：使用 new 分配的内存需要使用 delete 运算符进行手动释放，以避免内存泄漏。
2）malloc 函数：

malloc 是C语言中的库函数，也可以在C++中使用，用于在堆上动态分配内存。它不会调用构造函数，返回的指针指向一块未初始化的内存。
用法示例：

```
int *ptr = (int *)malloc(sizeof(int)); // 分配一个整数大小的内存，并将指针指向它
double *arr = (double *)malloc(10 * sizeof(double)); // 分配一个包含10个双精度浮点数的数组
```

前面的括号是类型强制转换

函数原型：

```
void *malloc(size_t size); // 分配 size 大小的内存并返回指向它的指针
```

注意：使用 malloc 分配的内存需要使用 free 函数进行手动释放，以避免内存泄漏。
总结：

new 操作符是C++的一部分，能够自动调用构造函数，分配的内存必须使用 delete 运算符释放。

malloc 函数是C语言的库函数，可以在C++中使用，分配的内存需要使用 free 函数释放。

在C++中，建议使用 new 和 delete 运算符，因为它们能够更好地与构造和析构函数配合使用。如果需要与C代码兼容，可以使用 malloc 和 free。

讲一下 堆栈在各个方面的不同之处

当涉及堆和栈时，有许多方面可以区分它们。以下是堆栈在各个方面的不同之处的详细介绍：

1. 分配和释放方式：

堆：堆内存的分配和释放是显式的，程序员需要手动分配和释放内存。分配使用 new 或者 malloc，释放使用 delete 或者 free。

栈：栈内存的分配和释放是隐式的，由系统自动管理。在函数调用时，栈上的局部变量被分配，函数返回后，这些变量自动被释放。

2. 生命周期：

堆：堆上分配的内存的生命周期可以在整个程序运行期间，直到**显式释放**。对象在堆上创建后，需要手动调用构造函数进行初始化，并在不再需要时手动释放。

栈：栈上的局部变量的生命周期与其所在的函数调用相关。变量在进入作用域时分配，**在退出作用域时自动销毁，无需手动释放**。因此，栈上的生命周期通常较短。

3. 内存管理：

堆：堆内存的管理比较灵活，适用于动态数据结构和需要动态分配内存的情况。然而，由于需要手动管理内存，容易出现内存泄漏和悬空指针等问题。

栈：栈内存的管理由系统自动完成，对程序员来说更加方便，无需手动释放内存。然而，栈内存分配有限，通常用于局部变量和函数调用。

4. 内存分配效率：

堆：堆内存的分配和释放涉及更复杂的操作，**可能会导致内存碎片的产生**。因此，堆的分配和释放相对较慢。

栈：栈内存的分配和释放是系统**自动管理**的，因此非常高效。函数的频繁调用和局部变量的管理在栈上完成效率较高。

5. 使用场景：

堆：适用于需要灵活管理对象生命周期的情况，例如动态数据结构（链表、树等）、大型对象等。

栈：适用于函数调用和局部变量的管理，特别适合频繁调用的情况。

综上所述，堆和栈在分配方式、生命周期、内存管理和适用场景等方面有着明显的不同。根据程序的需求，选择适当的内存管理方式对于代码的正确性、性能和可维护性都至关重要。

每个线程都有自己的栈空间

在多线程编程中，每个线程都需要自己的栈空间来管理函数调用和局部变量。这是因为线程在执行过程中需要维护自己的执行上下文，包括函数调用链和局部变量值。线程的栈空间通常会在创建线程时分配，并在线程终止时释放。

对于一个函数而言 函数参数压栈的顺序是?

在许多计算机体系结构中，函数参数的压栈顺序通常是从右到左（或从后向前）的。这意味着较后面的参数会被先压入栈，而较前面的参数会被后压入栈。这种顺序是为了支持一些编程语言的可变参数函数（例如C语言的变参函数）以及函数调用约定。
让我们以C语言为例，假设有一个函数 foo，其原型为：

```
1 int foo(int a, int b, int c);
```


当调用 foo 函数并传递参数时，参数的压栈顺序可能如下：

首先，参数 c 会被压入栈，因为它在参数列表中位于最右侧。

接下来，参数 b 会被压入栈，其次是参数 a。

所以，在栈中，参数的布局可能如下：

```
±--------+
| … |
±--------+
| a | <-- 栈顶
±--------+
| b |
±--------+
| c |
±--------+
```

需要注意的是，这种参数压栈顺序并不是绝对的，可能会因为编译器、体系结构以及编程语言的不同而有所变化。但是，从右到左的顺序在许多情况下是一种常见的约定。在某些体系结构中，也可能使用从左到右的顺序。如果您在特定环境下使用不同的编程语言或体系结构，请查阅相应的文档以了解参数压栈的具体顺序。

讲一下内存泄漏和解决办法

内存泄漏是指在程序运行时，分配给程序的内存没有被正确释放，导致系统中的可用内存逐渐减少，最终可能导致程序崩溃或系统性能下降。内存泄漏通常发生在程序没有正确管理动态分配内存（如堆内存）的情况下。以下是一些常见的内存泄漏情况以及解决办法：

1. **未释放堆内存： 最常见的内存泄漏是在动态分配内存后没有及时释放。这可能是因为程序员忘记了调用 free()（C/C++）或 delete（C++）来释放已经不再需要的内存。**
   解决办法： 在不再需要使用动态分配的内存时，务必调用适当的释放函数来释放内存。在 C/C++ 中使用 free()，在 C++ 中使用 delete，或者使用智能指针等自动内存管理工具。
2. 引用计数问题： 引用计数是一种内存管理技术，但它可能导致内存泄漏，特别是在循环引用的情况下。如果两个对象互相引用，并且它们的引用计数不正确地管理，那么它们可能永远不会被释放。（**这个是共享指针（shared_ptr）导致的）**
   解决办法： 在循环引用的情况下，可以使用弱引用（weak references）来避免循环引用导致的内存泄漏。同时，可以考虑使用更高级的内存管理技术，如垃圾回收机制。
3. 遗漏关闭资源： 除了内存外，还有其他资源如文件、网络连接等需要在不再使用时及时关闭。
   解决办法： 在不再需要使用资源时，确保调用适当的关闭函数来释放资源，如关闭文件句柄、网络连接等。
4. 对象池未释放： 在使用对象池技术时，如果没有正确地将对象返回到池中，就可能导致内存泄漏。
   解决办法： 使用对象池的情况下，务必确保在使用完对象后将其返回到池中，以便重复使用。
5. 内部缓存未清理： 如果程序使用了内部缓存，但没有定期清理不再需要的缓存项，可能会导致内存泄漏。
   解决办法： 确保在使用内部缓存时，定期检查并清理不再需要的缓存项。

6. 使用工具进行检测： 使用内存分析工具可以帮助检测和诊断内存泄漏问题，例如 Valgrind（C/C++）、VisualVM（Java）等。这些工具可以帮助定位内存泄漏的来源并提供解决方案。
7. **使用智能指针： 在支持智能指针的编程语言中，使用智能指针可以更方便地管理动态分配内存，避免手动释放内存的问题。** C++里具体讲了智能指针

在编写程序时，养成良好的内存管理习惯以及使用适当的工具进行检测是减少内存泄漏问题的关键。及时发现和解决内存泄漏可以提高程序的稳定性和性能。

什么是内存对齐

内存对齐是一种规则，它要求你把数据放在特定的位置上，以便计算机可以更高效地读取和处理这些数据。

想象你有一排存储盒子，每个盒子可以存储一定数量的数据。不同类型的数据（比如整数、浮点数、字符等）需要不同数量的盒子来存储。内存对齐告诉你如何将这些数据放在盒子中，以便计算机可以更快速地读取它们。

例如，假设你有一个整数（int），它需要4个盒子来存储。内存对齐规定，你应该从一排盒子的起始位置开始放置这个整数，这样整数的第一个盒子刚好在一个特定的内存地址上。这就是4字节对齐。如果你不按照这个规则，让整数的第一个盒子不在正确的位置，计算机可能会花更多时间来获取这个整数，影响效率。

内存对齐确保数据按照一种规律放置，使计算机可以更有效地访问它们，提高程序的性能。所以，你可以把内存对齐想象成一种优化策略，让计算机更智能地使用内存中的数据。

举个例子，考虑以下结构体：

```
struct MyStruct {
    char a;    // 1字节
    int b;     // 4字节
    double c;  // 8字节
};
```

在这个例子中，char 类型需要1字节，int 类型需要4字节，double 类型需要8字节。根据不同的编译器和体系结构，内存对齐的规则可能会有所不同，但通常情况下，编译器会对结构体成员进行对齐，以使整个结构体的大小是其成员中最大对齐要求的倍数。

假设这里的编译器使用了4字节的对齐规则，结构体的大小将会是 **1字节（char）+ 3字节填充 + 4字节（int）+ 8字节（double）= 16字节** 这样，结构体的内存布局会保证每个成员都在适当的对齐边界上，以便更高效地访问。

总之，结构体的内存对齐确保结构体成员在内存中以有效的方式存储，同时遵循特定的对齐规则，以提高程序的性能和效率。不同的编译器和设置可能会导致不同的内存对齐行为。

命名空间是干嘛的 常用的命名空间有哪些

在编程中，命名空间（Namespace）是一种包含了变量、函数、类等标识符的容器。命名空间的主要目的是避免名称冲突。例如，你可能在自己的程序中定义了一个名为 “max” 的函数，但同时编程语言的标准库也可能有一个名为 “max” 的函数。如果没有命名空间，这两个函数就会产生冲突。但是，如果你将你的函数定义在一个命名空间中，那么你就可以通过命名空间来调用你的函数，而不是标准库中的函数。

在不同的编程语言中，命名空间的概念和使用方式可能会有所不同。以下是一些常见的命名空间类型：

Python：在 Python 中，模块就是一种命名空间，模块内的函数和类都是在这个命名空间中。你可以通过 “import” 语句来导入一个模块，然后使用模块名作为命名空间来调用模块内的函数或类。例如，“os” 和 “sys” 是 Python 的标准库中的两个常用模块。

C++：在 C++ 中，你可以使用 “namespace” 关键字来定义一个命名空间，然后在命名空间内定义函数、类等。你可以使用 “::” 运算符来访问命名空间内的成员。例如，“std” 是 C++ 标准库的命名空间。

Java：在 Java 中，包（package）是一种命名空间。你可以使用 “package” 关键字来定义一个包，然后在包内定义类。你可以使用 “.” 运算符来访问包内的类。例如，“java.util” 和 “java.io” 是 Java 标准库中的常用包。

在 C++ 中，最常用的命名空间无疑是标准库的命名空间，也就是 std。这个命名空间包含了 C++ 标准库中的大多数函数和类。例如，std::cout、std::endl、std::vector、std::string 等等。
此外，一些常用的库也会定义自己的命名空间。例如：

Boost 库：Boost 是一个广泛使用的 C++ 库，提供了很多复杂的功能。Boost 库的组件通常在 boost 命名空间下。例如，boost::asio 是用于异步 I/O 的库，boost::filesystem 是用于文件系统操作的库。

Eigen 库：Eigen 是一个用于线性代数、矩阵和向量操作、数值分析和解决微分方程的 C++ 库。它的所有组件都在 Eigen 命名空间下。

std::placeholders：这是 std 命名空间中的一个子命名空间，用于 std::bind 函数。例如，std::placeholders::_1、std::placeholders::_2 等。

请注意，命名空间的使用取决于你的代码引用了哪些库。如果你的代码使用了其他库，那么可能会有其他的命名空间。

include<>和include""的区别

在C和C++编程中，#include 指令用于包含外部的头文件，以便在当前源文件中可以使用头文件中声明的函数、变量和其他定义。在 #include 指令中，尖括号 <> 和双引号 “” 是两种不同的方式，用于指定头文件的位置。

#include <>：
这种形式的 #include 用于包含系统提供的标准库头文件。编译器会在系统标准库的路径中查找指定的头文件。

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

在这里，stdio.h 和 stdlib.h 都是系统提供的标准库头文件，编译器会从标准库路径中查找并包含这些头文件。

#include “”：
这种形式的 #include 用于包含用户自定义的头文件或项目中的其他文件。编译器会首先在当前源文件所在的目录中查找指定的头文件，如果没有找到，则会在编译器设置的其他路径中查找。

```
#include "myheader.h"
#include "utils/functions.h"
```

在这里，myheader.h 和 utils/functions.h 都是自定义的头文件，它们应该与源文件位于同一目录或正确配置的路径下。

总结一下，#include <> 用于包含系统标准库头文件，而 #include “” 用于包含用户自定义的头文件。编译器会根据这两种不同的方式来查找并包含指定的头文件。在使用时，要根据情况选择适当的形式。