

简介

- C语言相关的理论知识笔记

C 简介

- C 语言是一种通用的高级语言，最初是由丹尼斯·里奇在贝尔实验室为开发 UNIX 操作系统而设计的。C 语言最开始是于 1972 年在 DEC PDP-11 计算机上被首次实现。
- 在 1978 年，布莱恩·柯林汉（Brian Kernighan）和丹尼斯·里奇（Dennis Ritchie）制作了 C 的第一个公开可用的描述，现在被称为 K&R 标准。
- UNIX 操作系统，C 编译器，和几乎所有的 UNIX 应用程序都是用 C 语言编写的。由于各种原因，C 语言现在已经成为一种广泛使用的专业语言。
- 优点
 - 易于学习
 - 结构化语言
 - 它产生高效率的程序
 - 它可以处理底层的活动
 - 它可以在多种计算平台上编译
- 关于 C
 - C 语言是为了编写 UNIX 操作系统而被发明的。
 - C 语言是以 B 语言为基础的，B 语言大概是在 1970 年被引进的。
 - C 语言标准是于 1988 年由美国国家标准协会（ANSI，全称 American National Standard Institute）制定的。
 - 截至 1973 年，UNIX 操作系统完全使用 C 语言编写。
 - 目前，C 语言是最广泛使用的系统程序设计语言。
 - 大多数先进的软件都是使用 C 语言实现的。
 - 当今最流行的 Linux 操作系统和 RDBMS（Relational Database Management System：关系数据库管理系统）MySQL 都是使用 C 语言编写的。
- 为什么要使用 C
 - C 语言最初是用于系统开发工作，特别是组成操作系统的程序。由于 C 语言所产生的代码运行速度与汇编语言编写的代码运行速度几乎一样，所以采用 C 语言作为系统开发语言。下面列举几个使用 C 的实例：
 - 操作系统
 - 语言编译器
 - 汇编器
 - 文本编辑器
 - 打印机
 - 网络驱动器
 - 现代程序
 - 数据库
 - 语言解释器

- 实体工具

C 程序结构

- C 程序主要包括以下部分
 - 预处理器指令
 - 函数
 - 变量
 - 语句 & 表达式
 - 注释

C 基本语法

- C 程序由各种令牌组成，令牌可以是关键字，标识符，常量，字符串值，或者是一个符号。
- 分号 ；
 - 在 C 程序中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。
- 注释
 - C 语言有两种注释方式
 - 以 // 开始的单行注释，这种注释可以单独占一行
 - /* */ 这种格式的注释可以单行或多行。
 - 您不能在注释内嵌套注释，注释也不能出现在字符串或字符值中
- 标识符
 - C 标识符是用来标识变量、函数，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 _ 开始，后跟零个或多个字母、下划线和数字（0-9）。
 - C 标识符内不允许出现标点字符，比如 @、\$ 和 %。C 是区分大小写的编程语言。因此，在 C 中，Manpower 和 manpower 是两个不同的标识符
- 关键字
 - 下表列出了 C 中的保留字。这些保留字不能作为常量名、变量名或其他标识符名称
 - auto 声明自动变量
 - break 跳出当前循环
 - case 开关语句分支
 - char 声明字符型变量或函数返回值类型
 - const 定义常量，如果一个变量被 const 修饰，那么它的值就不能再被改变
 - continue 结束当前循环，开始下一轮循环
 - default 开关语句中的"其它"分支
 - do 循环语句的循环体
 - double 声明双精度浮点型变量或函数返回值类型
 - else 条件语句否定分支（与 if 连用）
 - enum 声明枚举类型
 - extern 声明变量或函数是在其它文件或本文件的其他位置定义
 - float 声明浮点型变量或函数返回值类型

- for 一种循环语句
- goto 无条件跳转语句
- if 条件语句
- int 声明整型变量或函数
- long 声明长整型变量或函数返回值类型
- register 声明寄存器变量
- return 子程序返回语句（可以带参数，也可不带参数）
- short 声明短整型变量或函数
- signed 声明有符号类型变量或函数
- sizeof 计算数据类型或变量长度（即所占字节数）
- static 声明静态变量
- struct 声明结构体类型
- switch 用于开关语句
- typedef 用以给数据类型取别名
- unsigned 声明无符号类型变量或函数
- union 声明共用体类型
- void 声明函数无返回值或无参数，声明无类型指针
- volatile 说明变量在程序执行中可被隐含地改变
- while 循环语句的循环条件
- C99 新增关键字
 - _Bool _Complex _Imaginary inline restrict
- C11 新增关键字
 - _Alignas _Alignof _Atomic _Generic _Noreturn _Static_assert _Thread_local
- C 中的空格
 - 只包含空格的行，被称为空白行，可能带有注释，C 编译器会完全忽略它。
 - 在 C 中，空格用于描述空白符、制表符、换行符和注释。空格分隔语句的各个部分，让编译器能识别语句中的某个元素（比如 int）在哪里结束，下一个元素在哪里开始

C 数据类型

- 在 C 语言中，数据类型指的是用于声明不同类型的变量或函数的一个广泛的系统。变量的类型决定了变量存储占用的空间，以及如何解释存储的位模式
- C 中的类型可分为以下几种：
- 基本数据类型
 - 它们是算术类型，包括整型（int）、字符型（char）、浮点型（float）和双精度浮点型（double）。
 - 枚举类型：
 - 它们也是算术类型，被用来定义在程序中只能赋予其一定的离散整数值的变量。
 - void 类型：
 - 类型说明符 void 表示没有值的数据类型，通常用于函数返回值。
 - 派生类型：
 - ：包括数组类型、指针类型和结构体类型。
- 数组类型和结构类型统称为聚合类型。函数的类型指的是函数返回值的类型

- char 1 字节 -128 到 127 或 0 到 255
- unsigned char 1 字节 0 到 255
- signed char 1 字节 -128 到 127
- int 2 或 4 字节 -32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647
- unsigned int 2 或 4 字节 0 到 65,535 或 0 到 4,294,967,295
- short 2 字节 -32,768 到 32,767
- unsigned short 2 字节 0 到 65,535
- long 4 字节 -2,147,483,648 到 2,147,483,647
- unsigned long 4 字节 0 到 4,294,967,295
- void 类型
 - void 类型指定没有可用的值。它通常用于以下三种情况下
 - 函数返回为空
 - C 中有各种函数都不返回值，或者您可以说它们返回空。不返回值的函数的返回类型为 `void`。例如 `void exit (int status);`
 - 函数参数为空
 - C 中有各种函数不接受任何参数。不带参数的函数可以接受一个 `void`。例如 `int rand(void);`
 - 指针指向 `void`
 - 类型为 `void *` 的指针代表对象的地址，而不是类型。例如，内存分配函数 `void *malloc(size_t size);` 返回指向 `void` 的指针，可以转换为任何数据类型。
- 类型转换
 - 类型转换是将一个数据类型的值转换为另一种数据类型的值
 - C 语言中有两种类型转换
 - 隐式类型转换：隐式类型转换是在表达式中自动发生的，无需进行任何明确的指令或函数调用。它通常是将一种较小的类型自动转换为较大的类型，例如，将 `int` 类型转换为 `long` 类型或 `float` 类型转换为 `double` 类型。隐式类型转换也可能导致数据精度丢失或数据截断
 - 显式类型转换：显式类型转换需要使用强制类型转换运算符（`type casting operator`），它可以将一个数据类型的值强制转换为另一种数据类型的值。强制类型转换可以使程序员在必要时对数据类型进行更精确的控制，但也可能会导致数据丢失或截断

C 变量

- 变量，只不过是程序可操作的存储区的名称。C 中每个变量都有特定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可应用于变量上。
- 变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。大写字母和小写字母是不同的，因为 C 是大小写敏感的。基于前一章讲解的基本类型，有以下几种基本的变量类型：
 - `char`: 通常是一个字节（八位），这是一个整数类型。
 - `int`: 整型，4 个字节，取值范围 -2147483648 到 2147483647

- float: 单精度浮点值。单精度是这样的格式，1位符号，8位指数，23位小数
- double: 双精度浮点值。双精度是1位符号，11位指数，52位小数
- void: 表示类型的缺失。
- C 中的变量定义
 - 变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：
 - type variable_list;
 - type 表示变量的数据类型，可以是整型、浮点型、字符型、指针等，也可以是用户自定义的对象
 - variable_list 可以由一个或多个变量的名称组成，多个变量之间用逗号分隔，变量由字母、数字和下划线组成，且以字母或下划线开头。
- 变量的初始化
 - 在 C 语言中，变量的初始化是在定义变量的同时为其赋予一个初始值。变量的初始化可以在定义时进行，也可以在后续的代码中进行。
 - 初始化器由一个等号，后跟一个常量表达式组成，如下所示：
 - type variable_name = value;
 - 其中，type 表示变量的数据类型，variable_name 是变量的名称，value 是变量的初始值。
 - 后续初始化变量：
 - 在变量定义后的代码中，可以使用赋值运算符 = 为变量赋予一个新的值。
 - type variable_name; // 变量定义
 - variable_name = new_value; // 变量初始化
 - 需要注意的是，变量在使用之前应该被初始化。未初始化的变量的值是未定义的，可能包含任意的垃圾值。因此，为了避免不确定的行为和错误，建议在使用变量之前进行初始化。
- 变量不初始化
 - 在 C 语言中，如果变量没有显式初始化，那么它的默认值将取决于该变量的类型和其所在的作用域。
 - 对于全局变量和静态变量（在函数内部定义的静态变量和在函数外部定义的全局变量），它们的默认初始值为零。
 - 以下是不同类型的变量在没有显式初始化时的默认值：
 - 整型变量（int、short、long等）：默认值为0。
 - 浮点型变量（float、double等）：默认值为0.0。
 - 字符型变量（char）：默认值为'\0'，即空字符
 - 指针变量：默认值为NULL，表示指针不指向任何有效的内存地址
 - 数组、结构体、联合等复合类型的变量：它们的元素或成员将按照相应的规则进行默认初始化，这可能包括对元素递归应用默认规则。
 - 需要注意的是，局部变量（在函数内部定义的非静态变量）不会自动初始化为默认值，它们的初始值是未定义的（包含垃圾值）。因此，在使用局部变量之前，应该显式地为其赋予一个初始值。
 - 总结起来，C 语言中变量的默认值取决于其类型和作用域。全局变量和静态变量的默认值为 0，字符型变量的默认值为 \0，指针变量的默认值为 NULL，而局部变量没有默认值，其初始值是未定义的
- C 中的变量声明

- 变量声明向编译器保证变量以指定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。
- 变量的声明有两种情况：
 - 一种是需要建立存储空间的。例如：`int a` 在声明的时候就已经建立了存储空间。
 - 另一种是不需要建立存储空间的，通过使用`extern`关键字声明变量名而不定义它。例如：`extern int a` 其中变量 `a` 可以在别的文件中定义的
 - 除非有`extern`关键字，否则都是变量的定义。
 - `extern int i;` //声明，不是定义
 - `int i;` //声明，也是定义
- C 中的左值(Lvalues)和右值(Rvalues)
 - C 中有两种类型的表达式：
 - 左值 (lvalue)：指向内存位置的表达式被称为左值 (lvalue) 表达式。左值可以出现在赋值号的左边或右边
 - 右值 (rvalue)：术语右值 (rvalue) 指的是存储在内存中某些地址的数值。右值是不能对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边
 - 变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边

C 常量

- 常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量
- 常量可以是任何的基本数据类型，比如整数常量、浮点常量、字符常量，或字符串字面值，也有枚举常量
- 常量就像是常规的变量，只不过常量的值在定义后不能进行修改
- 常量可以直接在代码中使用，也可以通过定义常量来使用
- 整数常量
 - 整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：`0x` 或 `0X` 表示十六进制，`0` 表示八进制，不带前缀则默认表示十进制
 - 整数常量也可以带一个后缀，后缀是 `U` 和 `L` 的组合，`U` 表示无符号整数 (unsigned)，`L` 表示长整数 (long)。后缀可以是大写，也可以是小写，`U` 和 `L` 的顺序任意
- 浮点常量
 - 浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。
 - 当使用小数形式表示时，必须包含整数部分、小数部分，或同时包含两者。当使用指数形式表示时，必须包含小数点、指数，或同时包含两者。带符号的指数是用 `e` 或 `E` 引入的。
- 字符常量
 - 字符常量是括在单引号中，例如，`'x'` 可以存储在 `char` 类型的简单变量中

- 字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）
- 在 C 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符 (\n) 或制表符 (\t) 等。下表列出了一些这样的转义序列码：
- 字符串常量
 - 字符串字面值或常量是括在双引号 " " 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符
 - 您可以使用空格做分隔符，把一个很长的字符串常量进行分行
 - 字符串常量在内存中以 null 终止符 \0 结尾
- 定义常量
 - 在 C 中，有两种简单的定义常量的方式：
 - 使用 #define 预处理器：#define 可以在程序中定义一个常量，它在编译时会被替换为其对应的值
 - 使用 const 关键字：const 关键字用于声明一个只读变量，即该变量的值不能在程序运行时修改
- #define 预处理器
 - 下面是使用 #define 预处理器定义常量的形式：
 - #define 常量名 常量值
- const 关键字
 - 您可以使用 const 前缀声明指定类型的常量，如下所示：
 - const 数据类型 常量名 = 常量值;
- 请注意，把常量定义为大写字母形式，是一个很好的编程习惯。
- #define 与 const 区别
 - #define 与 const 这两种方式都可以用来定义常量，选择哪种方式取决于具体的需求和编程习惯。通常情况下，建议使用 const 关键字来定义常量，因为它具有类型检查和作用域的优势，而 #define 仅进行简单的文本替换，可能会导致一些意外的问题。
 - #define 预处理指令和 const 关键字在定义常量时有一些区别：
 - 替换机制：#define 是进行简单的文本替换，而 const 是声明一个具有类型的常量。#define 定义的常量在编译时会被直接替换为其对应的值，而 const 定义的常量在程序运行时分配内存，并且具有类型信息。
 - 类型检查：#define 不进行类型检查，因为它只是进行简单的文本替换。而 const 定义的常量具有类型信息，编译器可以对其进行类型检查。这可以帮助捕获一些潜在的类型错误。
 - 作用域：#define 定义的常量没有作用域限制，它在定义之后的整个代码中都有效。而 const 定义的常量具有块级作用域，只在其定义所在的作用域内有效。
 - 调试和符号表：使用 #define 定义的常量在符号表中不会有相应的条目，因为它只是进行文本替换。而使用 const 定义的常量会在符号表中有相应的条目，有助于调试和可读性。

C 存储类

- 存储类定义 C 程序中变量/函数的存储位置、生命周期和作用域

- 这些说明符放置在它们所修饰的类型之前
- 下面列出 C 程序中可用的存储类：
 - auto
 - register
 - static
 - extern
- auto 存储类
 - auto 存储类是所有局部变量默认的存储类
 - 定义在函数中的变量默认为 auto 存储类，这意味着它们在函数开始时被创建，在函数结束时被销毁

```
{  
    int mount;  
    auto int month;  
}
```

- 上面的实例定义了两个带有相同存储类的变量，auto 只能用在函数内，即 auto 只能修饰局部变量。
- register 存储类
 - register 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个字），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）
 - register 存储类定义存储在寄存器，所以变量的访问速度更快，但是它不能直接取地址，因为它不是存储在 RAM 中的。在需要频繁访问的变量上使用 register 存储类可以提高程序的运行速度

```
{  
    register int miles;  
}
```

- 寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。
- static 存储类
 - static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值
 - static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内
 - 全局声明的一个 static 变量或方法可以被任何函数或方法调用，只要这些方法出现在跟 static 变量或方法同一个文件中
 - 静态变量在程序中只被初始化一次，即使函数被调用多次，该变量的值也不会重置
- extern 存储类

- `extern` 存储类用于定义在其他文件中声明的全局变量或函数。当使用 `extern` 关键字时，不会为变量分配任何存储空间，而只是指示编译器该变量在其他文件中定义
- `extern` 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 `extern` 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置
- 当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 `extern` 来得到已定义的变量或函数的引用。可以这么理解，`extern` 是用来在另一个文件中声明一个全局变量或函数
- `extern` 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候

C 运算符

- 运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C 语言内置了丰富的运算符，并提供了以下类型的运算符：
 - 算术运算符
 - 关系运算符
 - 逻辑运算符
 - 位运算符
 - 赋值运算符
 - 杂项运算符
- 算术运算符
 - `+` 把两个操作数相加 `A + B` 将得到 30
 - `-` 从第一个操作数中减去第二个操作数 `A - B` 将得到 -10
 - `*` 把两个操作数相乘 `A * B` 将得到 200
 - `/` 分子除以分母 `B / A` 将得到 2
 - `%` 取模运算符，整除后的余数 `B % A` 将得到 0
 - `++` 自增运算符，整数值增加 1 `A++` 将得到 11
 - `--` 自减运算符，整数值减少 1 `A--` 将得到 9
- 关系运算符
 - `==` 检查两个操作数的值是否相等，如果相等则条件为真。(`A == B`) 为假。
 - `!=` 检查两个操作数的值是否相等，如果不相等则条件为真。(`A != B`) 为真。
 - `>` 检查左操作数的值是否大于右操作数的值，如果是则条件为真。(`A > B`) 为假。
 - `<` 检查左操作数的值是否小于右操作数的值，如果是则条件为真。(`A < B`) 为真。
 - `>=` 检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。(`A >= B`) 为假。
 - `<=` 检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。(`A <= B`) 为真。
- 逻辑运算符
 - `&&` 称为逻辑与运算符。如果两个操作数都非零，则条件为真。(`A && B`) 为假。
 - `||` 称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。(`A || B`) 为真。
 - `!` 称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。!
(`A && B`) 为真。
- 赋值运算符
 - `=` 简单的赋值运算符，把右边操作数的值赋给左边操作数 `C = A + B` 将把 `A + B` 的值赋给 `C`

- `+=` 加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数 `C += A` 相当于 `C = C + A`
- `-=` 减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数 `C -= A` 相当于 `C = C - A`
- `*=` 乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数 `C *= A` 相当于 `C = C * A`
- `/=` 除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数 `C /= A` 相当于 `C = C / A`
- `%=` 求模且赋值运算符，求两个操作数的模赋值给左边操作数 `C %= A` 相当于 `C = C % A`
- `<<=` 左移且赋值运算符 `C <<= 2` 等同于 `C = C << 2`
- `>>=` 右移且赋值运算符 `C >>= 2` 等同于 `C = C >> 2`
- `&=` 按位与且赋值运算符 `C &= 2` 等同于 `C = C & 2`
- `^=` 按位异或且赋值运算符 `C ^= 2` 等同于 `C = C ^ 2`
- `|=` 按位或且赋值运算符 `C |= 2` 等同于 `C = C | 2`
- 杂项运算符 \mapsto `sizeof` & 三元
 - `sizeof()` 返回变量的大小。 `sizeof(a)` 将返回 4，其中 `a` 是整数。
 - `&` 返回变量的地址。 `&a;` 将给出变量的实际地址。
 - `*` 指向一个变量。 `*a;` 将指向一个变量。
 - `?:` 条件表达式 如果条件为真？则值为 X：否则值为 Y

C 判断

- 判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）
- C 语言把任何非零和非空的值假定为 `true`，把零或 `null` 假定为 `false`。
- `?:` 运算符(三元运算符)
 - 我们已经在前面的章节中讲解了 条件运算符 `?:`，可以用来替代 `if...else` 语句。它的一般形式如下：
 - `Exp1 ? Exp2 : Exp3;`
 - 其中，`Exp1`、`Exp2` 和 `Exp3` 是表达式。请注意，冒号的使用和位置。
 - `?` 表达式的值是由 `Exp1` 决定的。如果 `Exp1` 为真，则计算 `Exp2` 的值，结果即为整个表达式的值。如果 `Exp1` 为假，则计算 `Exp3` 的值，结果即为整个表达式的值。

C 循环

- 有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。
- 编程语言提供了更为复杂执行路径的多种控制结构。
- 循环语句允许我们多次执行一个语句或语句组

C 函数

- 函数是一组一起执行一个任务的语句。每个 C 程序都至少有一个函数，即主函数 `main()`，所有简单的程序都可以定义其他额外的函数。

- 您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是每个函数执行一个特定的任务来进行的。
- 函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。
- C 标准库提供了大量的程序可以调用的内置函数。例如，函数 `strcat()` 用来连接两个字符串，函数 `memcpy()` 用来复制内存到另一个位置。
- 函数还有很多叫法，比如方法、子例程或程序，等等
- 定义函数
 - C 语言中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
    body of the function
}
```

- 在 C 语言中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：
 - 返回类型：一个函数可以返回一个值。`return_type` 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，`return_type` 是关键字 `void`。
 - 函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
 - 参数：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
 - 函数主体：函数主体包含一组定义函数执行任务的语句。
- 函数声明
 - 函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。
 - 函数声明包括以下几个部分：
 - `return_type function_name(parameter list);`
- 调用函数
 - 创建 C 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。
 - 当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。
 - 调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值
- 函数参数
 - 如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数。
 - 形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。
 - 当调用函数时，有两种向函数传递参数的方式：
 - 传值调用 该方法把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数不会影响实际参数。
 - 引用调用 通过指针传递方式，形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作。


- 默认情况下，C 使用传值调用来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的实际参数。

C 作用域规则

- 任何一种编程中，作用域是程序中定义的变量所存在的区域，超过该区域变量就不能被访问。C 语言中有三个地方可以声明变量：
 - 在函数或块内部的局部变量
 - 在所有函数外部的全局变量
 - 在形式参数的函数参数定义中
- 局部变量
 - 在某个函数或块的内部声明的变量称为局部变量。它们只能被该函数或该代码块内部的语句使用。局部变量在函数外部是不可知的。
- 全局变量
 - 全局变量是定义在函数外部，通常是在程序的顶部。全局变量在整个程序生命周期内都是有效的，在任意的函数内部能访问全局变量。
 - 全局变量可以被任何函数访问。也就是说，全局变量在声明后整个程序中都是可用的。
 - 在程序中，局部变量和全局变量的名称可以相同，但是在函数内，如果两个名字相同，会使用局部变量值，全局变量不会被使用。
- 形式参数
 - 函数的参数，形式参数，被当作该函数内的局部变量，如果与全局变量同名它们会优先使用
- 全局变量与局部变量在内存中的区别：
 - 全局变量保存在内存的全局存储区中，占用静态的存储单元；
 - 局部变量保存在栈中，只有在所在函数被调用时才动态地为变量分配存储单元。

C 数组

- C 语言支持数组数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。
- 所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素
- 数组中的特定元素可以通过索引访问，第一个索引值为 0
- C 语言还允许我们使用指针来处理数组，这使得对数组的操作更加灵活和高效
- 声明数组
 - 在 C 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示
 - `type arrayName [arraySize];`
 - 这叫做一维数组。arraySize 必须是一个大于零的整数常量，type 可以是任意有效的 C 数据类型
- 初始化数组

- 在 C 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：
 - `double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};`
-  中指定的元素数目
- 如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数
- 访问数组元素
 - 数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内，跟在数组名称的后边
- 获取数组长度
 - 数组长度可以使用 `sizeof` 运算符来获取数组的长度，例如：

```
int numbers[] = {1, 2, 3, 4, 5};
int length = sizeof(numbers) / sizeof(numbers[0]);
```

- 使用宏定义：

```
#include <stdio.h>

#define LENGTH(array) (sizeof(array) / sizeof(array[0]))

int main() {
    int array[] = {1, 2, 3, 4, 5};
    int length = LENGTH(array);

    printf("数组长度为: %d\n", length);

    return 0;
}
```

- 数组名
 - 在 C 语言中，数组名表示数组的地址，即数组首元素的地址。当我们在声明和定义一个数组时，该数组名就代表着该数组的地址
 - 例如，在以下代码中：
 - `int myArray[5] = {10, 20, 30, 40, 50};`
 - 在这里，`myArray` 是数组名，它表示整数类型的数组，包含 5 个元素。`myArray` 也代表着数组的地址，即第一个元素的地址
 - 数组名本身是一个常量指针，意味着它的值是不能被改变的，一旦确定，就不能再指向其他地方
 - 我们可以使用 `&` 运算符来获取数组的地址，如下所示

```
int myArray[5] = {10, 20, 30, 40, 50};
int *ptr = &myArray[0]; // 或者直接写作 int *ptr = myArray;
```

- 在上面的例子中，`ptr` 指针变量被初始化为 `myArray` 的地址，即数组的第一个元素的地址。

- 需要注意的是，虽然数组名表示数组的地址，但在大多数情况下，数组名会自动转换为指向数组首元素的指针。这意味着我们可以直接将数组名用于指针运算
- C 中数组详解
 - 在 C 中，数组是非常重要的，我们需要了解更多有关数组的细节。下面列出了 C 程序员必须清楚的一些与数组相关的重要概念
 - 多维数组 C 支持多维数组。多维数组最简单的形式是二维数组。
 - 传递数组给函数 您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
 - 从函数返回数组 C 允许从函数返回数组。
 - 指向数组的指针 您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。
 - 静态数组与动态数组 静态数组在编译时分配内存，大小固定，而动态数组在运行时手动分配内存，大小可变。

C enum(枚举)

- 枚举是 C 语言中的一种基本数据类型，用于定义一组具有离散值的常量，它可以让数据更简洁，更易读
- 枚举类型通常用于为程序中的一组相关的常量取名字，以便于程序的可读性和维护性
- 定义一个枚举类型，需要使用 `enum` 关键字，后面跟着枚举类型的名称，以及用大括号 `{}` 括起来的一组枚举常量。每个枚举常量可以用一个标识符来表示，也可以为它们指定一个整数值，如果没有指定，那么默认从 0 开始递增
- 枚举语法定义格式为：
 - `enum 枚举名 {枚举元素1,枚举元素2,.....};`
- 注意：第一个枚举成员的默认值为整型的 0，后续枚举成员的值在前一个成员上加 1。我们在这个实例中把第一个枚举成员的值定义为 1，第二个就为 2，以此类推
- 枚举变量的定义
 - 前面我们只是声明了枚举类型，接下来我们看看如何定义枚举变量
 - 我们可以通过以下三种方式来定义枚举变量
 - 先定义枚举类型，再定义枚举变量

```
enum DAY
{
    MON=1, TUE, WED, THU, FRI, SAT, SUN
};
enum DAY day;
```

- 定义枚举类型的同时定义枚举变量

```
enum DAY
{
```

```
    MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;
```

- 省略枚举名称，直接定义枚举变量

```
enum
{
    MON=1, TUE, WED, THU, FRI, SAT, SUN
} day;
```

- 在C语言中，枚举类型是被当做 int 或者 unsigned int 类型来处理的，所以按照 C 语言规范是没有办法遍历枚举类型的。
- 不过在一些特殊的情况下，枚举类型必须连续是可以实现有条件的遍历

C 指针

- 学习 C 语言的指针既简单又有趣。通过指针，可以简化一些 C 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。所以，想要成为一名优秀的 C 程序员，学习指针是很有必要的
- 正如您所知道的，每一个变量都有一个内存位置，每一个内存位置都定义了可使用 & 运算符访问的地址，它表示了在内存中的一个地址。
- 什么是指针
 - 指针也就是内存地址，指针变量是用来存放内存地址的变量。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：
 - `type *var_name;`
 - 在这里，`type` 是指针的基类型，它必须是一个有效的 C 数据类型，`var_name` 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针
 - 所有实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，对应指针的值的类型都是一样的，都是一个代表内存地址的长的十六进制数
 - 不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同
- 如何使用指针
 - 使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值
- C 中的 NULL 指针
 - 在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值是一个良好的编程习惯。赋为 NULL 值的指针被称为空指针。
 - NULL 指针是一个定义在标准库中的值为零的常量
 - 在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，它表明该指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。
 - 如需检查一个空指针，您可以使用 if 语句，如下所示：

```
if(ptr)      /* 如果 p 非空，则完成 */
if(!ptr)     /* 如果 p 为空，则完成 */
```

- C 指针详解

- 在 C 中，有很多指针相关的概念，这些概念都很简单，但是都很重要。下面列出了 C 程序员必须清楚的一些与指针相关的重要概念：
- 指针的算术运算 可以对指针进行四种算术运算：++、--、+、-
- 指针数组 可以定义用来存储指针的数组。
- 指向指针的指针 C 允许指向指针的指针。
- 传递指针给函数 通过引用或地址传递参数，使传递的参数在调用函数中被改变。
- 从函数返回指针 C 允许函数返回指针到局部变量、静态变量和动态内存分配。

C 函数指针与回调函数

- 函数指针

- 函数指针是指向函数的指针变量
- 通常我们说的指针变量是指向一个整型、字符型或数组等变量，而函数指针是指向函数
- 函数指针可以像一般函数一样，用于调用函数、传递参数
- 函数指针变量的声明
 - `typedef int (*fun_ptr)(int,int);` // 声明一个指向同样参数、返回值的函数指针类型

- 回调函数

- 函数指针作为某个函数的参数
- 函数指针变量可以作为某个函数的参数来使用的，回调函数就是一个通过函数指针调用的函数
- 简单讲：回调函数是由别人的函数执行时调用你实现的函数。
- 以下是来自知乎作者常溪玲的解说：
 - 你到一个商店买东西，刚好你要的东西没有货，于是你在店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。在这个例子里，你的电话号码就叫回调函数，你把电话留给店员就叫登记回调函数，店里后来有货了叫做触发了回调关联的事件，店员给你打电话叫做调用回调函数，你到店里去取货叫做响应回调事件。

C 字符串

- 在 C 语言中，字符串实际上是使用空字符 `\0` 结尾的一维字符数组。因此，`\0` 是用于标记字符串的结束
- 空字符（Null character）又称结束符，缩写 NUL，是一个数值为 0 的控制字符，`\0` 是转义字符，意思是告诉编译器，这不是字符 0，而是空字符
- 其实，您不需要把 `null` 字符放在字符串常量的末尾。C 编译器会在初始化数组时，自动把 `\0` 放在字符串的末尾
- C 中有大量操作字符串的函数：
 - `strcpy(s1, s2);`
 - 复制字符串 `s2` 到字符串 `s1`。
 - `strcat(s1, s2);`
 - 连接字符串 `s2` 到字符串 `s1` 的末尾。
 - `strlen(s1);`

- 返回字符串 s1 的长度。
- strcmp(s1, s2);
 - 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回小于 0；如果 s1>s2 则返回大于 0。
- strchr(s1, ch);
 - 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。
- strstr(s1, s2);
 - 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

C 结构体

- C 数组允许定义可存储相同类型数据项的变量，结构是 C 编程中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项
- 结构体中的数据成员可以是基本数据类型（如 int、float、char 等），也可以是其他结构体类型、指针类型等。
- 定义结构
 - 结构体定义由关键字 struct 和结构体名组成，结构体名可以根据需要自行定义
 - struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下

```
struct tag {  
    member-list  
    member-list  
    member-list  
    ...  
} variable-list ;
```

- tag 是结构体标签。
- member-list 是标准的变量定义，比如 int i; 或者 float f;，或者其他有效的变量定义
- variable-list 结构变量，定义在结构的末尾，最后一个分号之前，您可以指定一个或多个结构变量。下面是声明 Book 结构的方式

```
struct Books  
{  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
} book;
```

- 在一般情况下，tag、member-list、variable-list 这 3 部分至少会出现 2 个
- 如果两个结构体互相包含，则需要对其中一个结构体进行不完整声明，如下所示：

```
struct B;    //对结构体B进行不完整声明
```

```
//结构体A中包含指向结构体B的指针
struct A
{
    struct B *partner;
    //other members;
};

//结构体B中包含指向结构体A的指针，在A声明完后，B也随之进行声明
struct B
{
    struct A *partner;
    //other members;
};
```

- 访问结构成员
 - 为了访问结构的成员，我们使用成员访问运算符（.）。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。您可以使用 `struct` 关键字来定义结构类型的变量。
- 结构作为函数参数
 - 您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似
- 指向结构的指针
 - 您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似。
 - 现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 `&` 运算符放在结构名称的前面
 - 为了使用指向该结构的指针访问结构的成员，您必须使用 `->` 运算符
- 结构体大小的计算
 - C 语言中，我们可以使用 `sizeof` 运算符来计算结构体的大小，`sizeof` 返回的是给定类型或变量的字节大小
 - 对于结构体，`sizeof` 将返回结构体的总字节数，包括所有成员变量的大小以及可能的填充字节。
 - 结构体的大小可能会受到编译器的优化和对齐规则的影响，编译器可能会在结构体中插入一些额外的填充字节以对齐结构体的成员变量，以提高内存访问效率。因此，结构体的实际大小可能会大于成员变量大小的总和，如果你需要确切地了解结构体的内存布局和对齐方式，可以使用 `offsetof` 宏和 `attribute((packed))` 属性等进一步控制和查询结构体的大小和对齐方式。

C 共用体

- 共用体是一种特殊的数据类型，允许您在相同的内存位置存储不同的数据类型。您可以定义一个带有多成员的共用体，但是任何时候只能有一个成员带有值。共用体提供了一种使用相同的内存位置的有效方式。
- 定义共用体
 - 为了定义共用体，您必须使用 `union` 语句，方式与定义结构类似。`union` 语句定义了一个新的数据类型，带有多个成员。`union` 语句的格式如下：

```
union [union tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more union variables];
```

- `union tag` 是可选的，每个 `member definition` 是标准的变量定义，比如 `int i`; 或者 `float f`; 或者其他有效的变量定义。在共用体定义的末尾，最后一个分号之前，您可以指定一个或多个共用体变量，这是可选的。下面定义一个名为 `Data` 的共用体类型，有三个成员 `i`、`f` 和 `str`：

```
union Data
{
    int i;
    float f;
    char str[20];
} data;
```

- 现在，`Data` 类型的变量可以存储一个整数、一个浮点数，或者一个字符串。这意味着一个变量（相同的内存位置）可以存储多个多种类型的数据。您可以根据需要在一个共用体内使用任何内置的或者用户自定义的数据类型。
- 共用体占用的内存应足够存储共用体中最大的成员。例如，在上面的实例中，`Data` 将占用 20 个字节的内存空间，因为在各个成员中，字符串所占用的空间是最大的
- 访问共用体成员
 - 为了访问共用体的成员，我们使用成员访问运算符（`.`）。成员访问运算符是共用体变量名称和我们要访问的共用体成员之间的一个句号。您可以使用 `union` 关键字来定义共用体类型的变量。

C 位域

- C 语言的位域（bit-field）是一种特殊的结构体成员，允许我们按位对成员进行定义，指定其占用的位数。
- 如果程序的结构中包含多个开关的变量，即变量值为 `TRUE/FALSE`，如下：

```
struct
{
    unsigned int widthValidated;
    unsigned int heightValidated;
} status;
```

- 这种结构需要 8 字节的内存空间，但在实际上，在每个变量中，我们只存储 0 或 1，在这种情况下，C 语言提供了一种更好的利用内存空间的方式。如果您在结构内使用这样的变量，您可以定义变量的宽度来告诉编译器，您将只使用这些字节。例如，上面的结构可以重写成

```
struct
{
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status;
```

- 现在，上面的结构中，status 变量将占用 4 个字节的内存空间，但是只有 2 位被用来存储值。如果您用了 32 个变量，每一个变量宽度为 1 位，那么 status 结构将使用 4 个字节，但只要您再多用一个变量，如果使用了 33 个变量，那么它将分配内存的下一段来存储第 33 个变量，这个时候就开始使用 8 个字节。
- 位域的特点和使用方法如下：
 - 定义位域时，可以指定成员的位域宽度，即成员所占用的位数。
 - 位域的宽度不能超过其数据类型的大小，因为位域必须适应所使用的整数类型。
 - 位域的数据类型可以是 int、unsigned int、signed int 等整数类型，也可以是枚举类型。
 - 位域可以单独使用，也可以与其他成员一起组成结构体。
 - 位域的访问是通过点运算符 (.) 来实现的，与普通的结构体成员访问方式相同。
- 位域声明
 - 有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位二进制即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为"位域"或"位段"。
 - 所谓"位域"是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。
 - 典型的实例：
 - 用 1 位二进制存放一个开关量时，只有 0 和 1 两种状态。
 - 读取外部文件格式——可以读取非标准的文件格式。例如：9 位的整数
- 位域的定义和位域变量的说明
 - 位域定义与结构定义相仿，其形式为：

```
struct 位域结构名
{
    位域列表
};
```

- 其中位域列表的形式为：
 - type [member_name] : width ;
- 下面是有关位域中变量元素的描述：

- type 只能为 int(整型), unsigned int(无符号整型), signed int(有符号整型) 三种类型, 决定了如何解释位域的值。
- member_name 位域的名称。
- width 位域中位的数量。宽度必须小于或等于指定类型的位宽度。
- 带有预定义宽度的变量被称为位域。位域可以存储多于 1 位的数
- 对于位域的定义尚有以下几点说明：
 - 一个位域存储在同一个字节中, 如一个字节所剩空间不够存放另一位域时, 则会从下一单元起存放该位域。也可以有意使某位域从下一单元开始。

```
struct bs{
    unsigned a:4;
    unsigned :4;    /* 空域 */
    unsigned b:4;    /* 从下一单元开始存放 */
    unsigned c:4
}
```

- 在这个位域定义中, a 占第一字节的 4 位, 后 4 位填 0 表示不使用, b 从第二字节开始, 占用 4 位, c 占用 4 位
- 位域的宽度不能超过它所依附的数据类型的长度, 成员变量都是有类型的, 这个类型限制了成员变量的最大长度, :后面的数字不能超过这个长度
- 位域可以是无名位域, 这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k{
    int a:1;
    int :2;    /* 该 2 位不能使用 */
    int b:3;
    int c:2;
};
```

- 从以上分析可以看出, 位域在本质上就是一种结构类型, 不过其成员是按二进位分配的
- 位域的使用
 - 位域的使用和结构成员的使用相同, 其一般形式为：

```
位域变量名.位域名
位域变量名->位域名
```

- 位域允许用各种格式输出。

C typedef

- C 语言提供了 `typedef` 关键字，您可以使用它来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 `BYTE`
 - `typedef unsigned char BYTE;`
- 在这个类型定义之后，标识符 `BYTE` 可作为类型 `unsigned char` 的缩写，例如：
 - `BYTE b1, b2;`
- 按照惯例，定义时会大写字母，以便提醒用户类型名称是一个象征性的缩写，但您也可以使用小写字母，如下：
 - `typedef unsigned char byte;`
- 您也可以使用 `typedef` 来为用户自定义的数据类型取一个新的名字。
- `typedef` vs `#define`
 - `#define` 是 C 指令，用于为各种数据类型定义别名，与 `typedef` 类似，但是它们有以下几点不同：
 - `typedef` 仅限于为类型定义符号名称，`#define` 不仅可以为类型定义别名，也能为数值定义别名，比如您可以定义 1 为 `ONE`
 - `typedef` 是由编译器执行解释的，`#define` 语句是由预编译器进行处理的。

C 输入 & 输出

- 当我们提到输入时，这意味着要向程序填充一些数据。输入可以是以文件的形式或从命令行中进行。C 语言提供了一系列内置的函数来读取给定的输入，并根据需要填充到程序中
- 当我们提到输出时，这意味着要在屏幕上、打印机上或任意文件中显示一些数据。C 语言提供了一系列内置的函数来输出数据到计算机屏幕上和保存数据到文本文件或二进制文件中
- 标准文件
 - C 语言把所有的设备都当作文件。所以设备（比如显示器）被处理的方式与文件相同。以下三个文件会在程序执行时自动打开，以便访问键盘和屏幕。
 - 标准输入 `stdin` 键盘
 - 标准输出 `stdout` 屏幕
 - 标准错误 `stderr` 您的屏幕
 - 文件指针是访问文件的方式，本节将讲解如何从键盘上读取值以及如何把结果输出到屏幕上
 - C 语言中的 I/O (输入/输出) 通常使用 `printf()` 和 `scanf()` 两个函数
 - `scanf()` 函数用于从标准输入（键盘）读取并格式化，`printf()` 函数发送格式化输出到标准输出（屏幕）。
- `getchar()` & `putchar()` 函数
 - `int getchar(void)` 函数从屏幕读取下一个可用的字符，并把它返回为一个整数。这个函数在同一个时间内只会读取一个单一的字符。您可以在循环内使用这个方法，以便从屏幕上读取多个字符
 - `int putchar(int c)` 函数把字符输出到屏幕上，并返回相同的字符。这个函数在同一个时间内只会输出一个单一的字符。您可以在循环内使用这个方法，以便在屏幕上输出多个字符
- `gets()` & `puts` 函数

- `char *gets(char *s)` 函数从 `stdin` 读取一行到 `s` 所指向的缓冲区，直到一个终止符或 EOF。
- `int puts(const char *s)` 函数把字符串 `s` 和一个尾随的换行符写入到 `stdout`
- `scanf()` 和 `printf()` 函数
 - `int scanf(const char *format, ...)` 函数从标准输入流 `stdin` 读取输入，并根据提供的 `format` 来浏览输入
 - `int printf(const char *format, ...)` 函数把输出写入到标准输出流 `stdout`，并根据提供的格式产生输出
 - `format` 可以是一个简单的常量字符串，但是您可以分别指定 `%s`、`%d`、`%c`、`%f` 等来输出或读取字符串、整数、字符或浮点数。还有许多其他可用的格式选项，可以根据需要使用

C 文件读写

- 一个文件，无论它是文本文件还是二进制文件，都是代表了一系列的字节。C 语言不仅提供了访问顶层的函数，也提供了底层（OS）调用来处理存储设备上的文件
- 打开文件
 - 您可以使用 `fopen()` 函数来创建一个新的文件或者打开一个已有的文件，这个调用会初始化类型 `FILE` 的一个对象，类型 `FILE` 包含了所有用来控制流的必要的信息。下面是这个函数调用的原型：
 - `FILE *fopen(const char *filename, const char *mode);`
 - 在这里，`filename` 是字符串，用来命名文件，访问模式 `mode` 的值可以是下列值中的一个
 - `r` 打开一个已有的文本文件，允许读取文件。
 - `w` 打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。如果文件存在，则该会被截断为零长度，重新写入。
 - `a` 打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。
 - `r+` 打开一个文本文件，允许读写文件。
 - `w+` 打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。
 - `a+` 打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。
 - 如果处理的是二进制文件，则需使用下面的访问模式来取代上面的访问模式：
 - `"rb"`, `"wb"`, `"ab"`, `"rb+"`, `"r+b"`, `"wb+"`, `"w+b"`, `"ab+"`, `"a+b"`
- 关闭文件
 - 为了关闭文件，请使用 `fclose()` 函数。函数的原型如下
 - `int fclose(FILE *fp);`
 - 如果成功关闭文件，`fclose()` 函数返回零，如果关闭文件时发生错误，函数返回 EOF。这个函数实际上，会清空缓冲区中的数据，关闭文件，并释放用于该文件的所有内存。EOF 是一个定义在头文件 `stdio.h` 中的常量
 - C 标准库提供了各种函数来按字符或者以固定长度字符串的形式读写文件
- 写入文件
 - 下面是把字符写入到流中的最简单的函数：
 - `int fputc(int c, FILE *fp);`

- 函数 `fputc()` 把参数 `c` 的字符值写入到 `fp` 所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 `EOF`。您可以使用下面的函数来把一个以 `null` 结尾的字符串写入到流中：

- int `fputs(const char *s, FILE *fp);`

- 函数 `fputs()` 把字符串 `s` 写入到 `fp` 所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回 `EOF`。您也可以使用 `int fprintf(FILE *fp, const char *format, ...)` 函数把一个字符串写入到文件中。尝试下面的实例：

- 读取文件

- 下面是从文件读取单个字符的最简单的函数：
- `int fgetc(FILE *fp);`
- `fgetc()` 函数从 `fp` 所指向的输入文件中读取一个字符。返回值是读取的字符，如果发生错误则返回 `EOF`。下面的函数允许您从流中读取一个字符串：
- `char *fgets(char *buf, int n, FILE *fp);`
- 函数 `fgets()` 从 `fp` 所指向的输入流中读取 `n - 1` 个字符。它会把读取的字符串复制到缓冲区 `buf`，并在最后追加一个 `null` 字符来终止字符串。
- 如果这个函数在读取最后一个字符之前就遇到一个换行符 `'\n'` 或文件的末尾 `EOF`，则只会返回读取到的字符，包括换行符。您也可以使用 `int fscanf(FILE *fp, const char *format, ...)` 函数来从文件中读取字符串，但是在遇到第一个空格和换行符时，它会停止读取

- 二进制 I/O 函数

- 下面两个函数用于二进制输入和输出：

```
size_t fread(void *ptr, size_t size_of_elements,
              size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
              size_t number_of_elements, FILE *a_file);
```

C 预处理器

- C 预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器只不过是一个文本替换工具而已，它们会指示编译器在实际编译之前完成所需的预处理。我们将把 C 预处理器（C Preprocessor）简写为 CPP。
- 所有的预处理器命令都是以井号（`#`）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一列开始。下面列出了所有重要的预处理器指令：
 - `#define` 定义宏
 - `#include` 包含一个源代码文件
 - `#undef` 取消已定义的宏
 - `#ifdef` 如果宏已经定义，则返回真
 - `#ifndef` 如果宏没有定义，则返回真
 - `#if` 如果给定条件为真，则编译下面代码
 - `#else #if` 的替代方案
 - `#elif` 如果前面的 `#if` 给定条件不为真，当前条件为真，则编译下面代码

- `#endif` 结束一个 `#if.....#else` 条件编译块
- `#error` 当遇到标准错误时，输出错误消息
- `#pragma` 使用标准化方法，向编译器发布特殊的命令到编译器中
- 预处理器示例
 - `#define MAX_ARRAY_LENGTH 20`
 - 这个指令告诉 CPP 把所有的 `MAX_ARRAY_LENGTH` 定义为 20。使用 `#define` 定义常量来增强可读性。

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

- 这些指令告诉 CPP 从系统库中获取 `stdio.h`，并添加文本到当前的源文件中。下一行告诉 CPP 从本地目录中获取 `myheader.h`，并添加内容到当前的源文件中。

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

- 这个指令告诉 CPP 取消已定义的 `FILE_SIZE`，并定义它为 42

```
#ifndef MESSAGE
    #define MESSAGE "You wish!"
#endif
```

- 这个指令告诉 CPP 只有当 `MESSAGE` 未定义时，才定义 `MESSAGE`。

```
#ifdef DEBUG
    /* Your debugging statements here */
#endif
```

- 这个指令告诉 CPP 如果定义了 `DEBUG`，则执行处理语句。在编译时，如果您向 `gcc` 编译器传递了 `-DDEBUG` 开关量，这个指令就非常有用。它定义了 `DEBUG`，您可以在编译期间随时开启或关闭调试。

- 预定义宏
 - ANSI C 定义了许多宏。在编程中您可以使用这些宏，但是不能直接修改这些预定义的宏
 - `__DATE__` 当前日期，一个以 "MMM DD YYYY" 格式表示的字符常量。
 - `__TIME__` 当前时间，一个以 "HH:MM:SS" 格式表示的字符常量。
 - `__FILE__` 这会包含当前文件名，一个字符串常量。
 - `__LINE__` 这会包含当前行号，一个十进制常量。
 - `__STDC__` 当编译器以 ANSI 标准编译时，则定义为 1。

- 预处理器运算符

- C 预处理器提供了下列的运算符来帮助您创建宏：
 - 宏延续运算符 (\)
- 一个宏通常写在一个单行上。但是如果宏太长，一个单行容纳不下，则使用宏延续运算符 (\) 。
例如：

```
#define message_for(a, b) \  
    printf("#a " and " #b ": We love you!\n")
```

- 标记粘贴运算符 (##)
 - 宏定义内的标记粘贴运算符 (##) 会合并两个参数。它允许在宏定义中两个独立的标记被合并为一个标记。例如：

```
#include <stdio.h>  
  
#define tokenpaster(n) printf ("token" #n " = %d", token##n)  
  
int main(void)  
{  
    int token34 = 40;  
  
    tokenpaster(34);  
    return 0;  
}
```

- 这个实例演示了 token##n 会连接到 token34 中，在这里，我们使用了字符串常量运算符 (#) 和标记粘贴运算符 (##) 。
- defined() 运算符
 - 预处理器 defined 运算符是用在常量表达式中的，用来确定一个标识符是否已经使用 #define 定义过。如果指定的标识符已定义，则值为真（非零）。如果指定的标识符未定义，则值为假（零）。下面的实例演示了 defined() 运算符的用法：

```
#include <stdio.h>  
  
#if !defined (MESSAGE)  
    #define MESSAGE "You wish!"  
#endif  
  
int main(void)  
{  
    printf("Here is the message: %s\n", MESSAGE);  
    return 0;  
}
```

- 参数化的宏
 - C++ 一个强大的功能是可以使用参数化的宏来模拟函数。例如，下面的代码是计算一个数的平方

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

- 我们可以使用宏重写上面的代码，如下：
 - `#define square(x) ((x) * (x))`
- 在使用带有参数的宏之前，必须使用 `#define` 指令定义。参数列表是括在圆括号内，且必须紧跟在宏名称的后边。宏名称和左圆括号之间不允许有空格。例如：

```
#include <stdio.h>  
  
#define MAX(x,y) ((x) > (y) ? (x) : (y))  
  
int main(void)  
{  
    printf("Max between 20 and 10 is %d\n", MAX(10, 20));  
    return 0;  
}
```

C 头文件

- 头文件是扩展名为 `.h` 的文件，包含了 C 函数声明和宏定义，被多个源文件中引用共享。有两种类型的头文件：程序员编写的头文件和编译器自带的头文件。
- 在程序中要使用头文件，需要使用 C 预处理指令 `#include` 来引用它。前面我们已经看过 `stdio.h` 头文件，它是编译器自带的头文件。
- 引用头文件相当于复制头文件的内容，但是我们不会直接在源文件中复制头文件的内容，因为这么做很容易出错，特别在程序是由多个源文件组成的时候。
- A simple practice in C 或 C++ 程序中，建议把所有的常量、宏、系统全局变量和函数原型写在头文件中，在需要的时候随时引用这些头文件。
- 引用头文件的语法
 - 使用预处理指令 `#include` 可以引用用户和系统头文件。它的形式有以下两种：
 - `#include`
 - 这种形式用于引用系统头文件。它在系统目录的标准列表中搜索名为 `file` 的文件。在编译源代码时，您可以通过 `-I` 选项把目录前置在该列表前。
 - `#include "file"`
 - 这种形式用于引用用户头文件。它在包含当前文件的目录中搜索名为 `file` 的文件。在编译源代码时，您可以通过 `-I` 选项把目录前置在该列表前。
- 只引用一次头文件
 - 如果一个头文件被引用两次，编译器会处理两次头文件的内容，这将产生错误。为了防止这种情况，标准的做法是把文件的整个内容放在条件编译语句中，如下：

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

- 这种结构就是通常所说的包装器 `#ifndef`。当再次引用头文件时，条件为假，因为 `HEADER_FILE` 已定义。此时，预处理器会跳过文件的整个内容，编译器会忽略它
- 有条件引用
 - 有时需要从多个不同的头文件中选择一个引用到程序中。例如，需要指定在不同的操作系统上使用的配置参数。您可以通过一系列条件来实现这点，如下：

```
#if SYSTEM_1
    # include "system_1.h"
#elif SYSTEM_2
    # include "system_2.h"
#elif SYSTEM_3
    ...
#endif
```

- 但是如果头文件比较多的时候，这么做是很不妥当的，预处理器使用宏来定义头文件的名称。这就是所谓的有条件引用。它不是用头文件的名称作为 `#include` 的直接参数，您只需要使用宏名称代替即可：

```
#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H
```

- `SYSTEM_H` 会扩展，预处理器会查找 `system_1.h`，就像 `#include` 最初编写的那样。`SYSTEM_H` 可通过 `-D` 选项被您的 `Makefile` 定义。
- 在有多个 `.h` 文件和多个 `.c` 文件的时候，往往会用一个 `global.h` 的头文件来包括所有的 `.h` 文件，然后在除 `global.h` 文件外的头文件中包含 `global.h` 就可以实现所有头文件的包含，同时不会乱。方便在各个文件里面调用其他文件的函数或者变量。

```
#ifndef _GLOBAL_H
#define _GLOBAL_H
#include <fstream>
#include <iostream>
#include <math.h>
#include <Config.h>
```

C 强制类型转换

- 强制类型转换是把变量从一种类型转换为另一种数据类型。例如，如果您想存储一个 long 类型的值到一个简单的整型中，您需要把 long 类型强制转换为 int 类型。您可以使用强制类型转换运算符来把值显式地从一种类型转换为另一种类型，如下所示：
 - (type_name) expression
- 类型转换可以是隐式的，由编译器自动执行，也可以是显式的，通过使用强制类型转换运算符来指定。在编程时，有需要类型转换的时候都用上强制类型转换运算符，是一种良好的编程习惯。
- 整数提升
 - 整数提升是指把小于 int 或 unsigned int 的整数类型转换为 int 或 unsigned int 的过程。
- 常用的算术转换
 - int -> unsigned int -> long -> unsigned long -> long long -> unsigned long long -> float -> double -> long double
 - 常用的算术转换不适用于赋值运算符、逻辑运算符 && 和 ||。

C 错误处理

- C 语言不提供对错误处理的直接支持，但是作为一种系统编程语言，它以返回值的形式允许您访问底层数据。在发生错误时，大多数的 C 或 UNIX 函数调用返回 1 或 NULL，同时会设置一个错误代码 errno，该错误代码是全局变量，表示在函数调用期间发生了错误。您可以在 errno.h 头文件中找到各种各样的错误代码。
- 所以，C 程序员可以通过检查返回值，然后根据返回值决定采取哪种适当的动作。开发人员应该在程序初始化时，把 errno 设置为 0，这是一种良好的编程习惯。0 值表示程序中没有错误。
- errno、perror() 和 strerror()
 - C 语言提供了 perror() 和 strerror() 函数来显示与 errno 相关的文本消息。
 - perror() 函数显示您传给它的字符串，后跟一个冒号、一个空格和当前 errno 值的文本表示形式。
 - strerror() 函数，返回一个指针，指针指向当前 errno 值的文本表示形式。
- 被零除的错误
 - 在进行除法运算时，如果不检查除数是否为零，则会导致一个运行时错误。
- 程序退出状态
 - 通常情况下，程序成功执行完一个操作正常退出的时候会带有值 EXIT_SUCCESS。在这里，EXIT_SUCCESS 是宏，它被定义为 0。
 - 如果程序中存在一种错误情况，当您退出程序时，会带有状态值 EXIT_FAILURE，被定义为 -1。

C 递归

- 递归指的是在函数的定义中使用函数自身的方法

- C 语言支持递归，即一个函数可以调用其自身。但在使用递归时，程序员需要注意定义一个从函数退出的条件，否则会进入死循环。
- 递归函数在解决许多数学问题上起了至关重要的作用，比如计算一个数的阶乘、生成斐波那契数列，等等

C 可变参数

- 有时，您可能会碰到这样的情况，您希望函数带有可变数量的参数，而不是预定义数量的参数。
- C 语言为这种情况提供了一个解决方案，它允许您定义一个函数，能根据具体的需求接受可变数量的参数。
- 声明方式为：
 - `int func_name(int arg1, ...);`
- 其中，省略号 ... 表示可变参数列表。
- 下面的实例演示了这种函数的使用：

```
int func(int, ... ) {  
    .  
    .  
    .  
}  
  
int main() {  
    func(2, 2, 3);  
    func(3, 2, 3, 4);  
}
```

- 请注意，函数 `func()` 最后一个参数写成省略号，即三个点号 (...)，省略号之前的那个参数是 `int`，代表了要传递的可变参数的总数。为了使用这个功能，您需要使用 `stdarg.h` 头文件，该文件提供了实现可变参数功能的函数和宏。具体步骤如下：
 - 定义一个函数，最后一个参数为省略号，省略号前面可以设置自定义参数。
 - 在函数定义中创建一个 `va_list` 类型变量，该类型是在 `stdarg.h` 头文件中定义的。
 - 使用 `int` 参数和 `va_start()` 宏来初始化 `va_list` 变量为一个参数列表。宏 `va_start()` 是在 `stdarg.h` 头文件中定义的。
 - 使用 `va_arg()` 宏和 `va_list` 变量来访问参数列表中的每个项。
 - 使用宏 `va_end()` 来清理赋予 `va_list` 变量的内存。
- 常用的宏有：
 - `va_start(ap, last_arg)`：初始化可变参数列表。`ap` 是一个 `va_list` 类型的变量，`last_arg` 是最后一个固定参数的名称（也就是可变参数列表之前的参数）。该宏将 `ap` 指向可变参数列表中的第一个参数。
 - `va_arg(ap, type)`：获取可变参数列表中的下一个参数。`ap` 是一个 `va_list` 类型的变量，`type` 是下一个参数的类型。该宏返回类型为 `type` 的值，并将 `ap` 指向下一个参数。
 - `va_end(ap)`：结束可变参数列表的访问。`ap` 是一个 `va_list` 类型的变量。该宏将 `ap` 置为 `NULL`。
- 现在让我们按照上面的步骤，来编写一个带有可变数量参数的函数，并返回它们的平均值：

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

```
double average(int num,...)
{
    va_list valist;
    double sum = 0.0;
    int i;

    /* 为 num 个参数初始化 valist */
    va_start(valist, num);

    /* 访问所有赋给 valist 的参数 */
    for (i = 0; i < num; i++)
    {
        sum += va_arg(valist, int);
    }
    /* 清理为 valist 保留的内存 */
    va_end(valist);

    return sum/num;
}

int main()
{
    printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
    printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

C 内存管理

- 本章将讲解 C 中的动态内存管理。C 语言为内存的分配和管理提供了几个函数。这些函数可以在 `<stdlib.h>` 头文件中找到。
- 在 C 语言中，内存是通过指针变量来管理的。指针是一个变量，它存储了一个内存地址，这个内存地址可以指向任何数据类型的变量，包括整数、浮点数、字符和数组等。C 语言提供了一些函数和运算符，使得程序员可以对内存进行操作，包括分配、释放、移动和复制等。
- `void calloc(int num, int size);` 在内存中动态地分配 `num` 个长度为 `size` 的连续空间，并将每一个字节都初始化为 0。所以它的结果是分配了 `numsize` 个字节长度的内存空间，并且每个字节的值都是 0。
- `void free(void *address);` 该函数释放 `address` 所指向的内存块,释放的是动态分配的内存空间。
- `void *malloc(int num);` 在堆区分配一块指定大小的内存空间，用来存放数据。这块内存空间在函数执行完成后不会被初始化，它们的值是未知的。
- `void *realloc(void *address, int newsize);` 该函数重新分配内存，把内存扩展到 `newsize`。
- 注意：void * 类型表示未确定类型的指针。C、C++ 规定 void * 类型可以通过类型转换强制转换为任何其它类型的指针。
- 重新调整内存的大小和释放内存

- 当程序退出时，操作系统会自动释放所有分配给程序的内存，但是，建议您在不需要内存时，都应该调用函数 `free()` 来释放内存。
- 或者，您可以通过调用函数 `realloc()` 来增加或减少已分配的内存块的大小
- C 语言中常用的内存管理函数和运算符
 - `malloc()` 函数：用于动态分配内存。它接受一个参数，即需要分配的内存大小（以字节为单位），并返回一个指向分配内存的指针。
 - `free()` 函数：用于释放先前分配的内存。它接受一个指向要释放内存的指针作为参数，并将该内存标记为未使用状态。
 - `calloc()` 函数：用于动态分配内存，并将其初始化为零。它接受两个参数，即需要分配的内存块数和每个内存块的大小（以字节为单位），并返回一个指向分配内存的指针。
 - `realloc()` 函数：用于重新分配内存。它接受两个参数，即一个先前分配的指针和一个新的内存大小，然后尝试重新调整先前分配的内存块的大小。如果调整成功，它将返回一个指向重新分配内存的指针，否则返回一个空指针。
 - `sizeof` 运算符：用于获取数据类型或变量的大小（以字节为单位）。
 - 指针运算符：用于获取指针所指向的内存地址或变量的值。
 - `&` 运算符：用于获取变量的内存地址。
 - `*` 运算符：用于获取指针所指向的变量的值。
 - `->` 运算符：用于指针访问结构体成员，语法为 `pointer->member`，等价于 `(*pointer).member`。
 - `memcpy()` 函数：用于从源内存区域复制数据到目标内存区域。它接受三个参数，即目标内存区域的指针、源内存区域的指针和要复制的数据大小（以字节为单位）。
 - `memmove()` 函数：类似于 `memcpy()` 函数，但它可以处理重叠的内存区域。它接受三个参数，即目标内存区域的指针、源内存区域的指针和要复制的数据大小（以字节为单位）

C 命令行参数

- 执行程序时，可以从命令行传值给 C 程序。这些值被称为命令行参数，它们对程序很重要，特别是当您想从外部控制程序，而不是在代码内对这些值进行硬编码时，就显得尤为重要了。
- 命令行参数是使用 `main()` 函数参数来处理的，其中，`argc` 是指传入参数的个数，`argv[]` 是一个指针数组，指向传递给程序的每个参数。
- 应当指出的是，`argv[0]` 存储程序的名称，`argv[1]` 是一个指向第一个命令行参数的指针，`*argv[n]` 是最后一个参数。如果没有提供任何参数，`argc` 将为 1，否则，如果传递了一个参数，`argc` 将被设置为 2
- 多个命令行参数之间用空格分隔，但是如果参数本身带有空格，那么传递参数的时候应把参数放置在双引号 "" 或单引号 ' 内部。让我们重新编写上面的实例，有一个空间，那么您可以通过这样的观点，把它们放在双引号或单引号 "" 或 ' 内部。让我们重新编写上面的实例，向程序传递一个放置在双引号内部的命令行参数：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    printf("Program name %s\n", argv[0]);

    if( argc == 2 )
    {
        printf("The argument supplied is %s\n", argv[1]);
    }
}
```



```
else if( argc > 2 )
{
    printf("Too many arguments supplied.\n");
}
else
{
    printf("One argument expected.\n");
}
}
```

C 排序算法

- 冒泡排序
 - 冒泡排序（英语：Bubble Sort）是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序（如从大到小、首字母从A到Z）错误就把他们交换过来。

```
#include <stdio.h>
void bubble_sort(int arr[], int len) {
    int i, j, temp;
    for (i = 0; i < len - 1; i++)
        for (j = 0; j < len - 1 - i; j++)
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
}
int main() {
    int arr[] = { 22, 34, 3, 32, 82, 55, 89, 50, 37, 5, 64, 35, 9, 70 };
    int len = (int) sizeof(arr) / sizeof(*arr);
    bubble_sort(arr, len);
    int i;
    for (i = 0; i < len; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

- 选择排序
 - 选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

```
void selection_sort(int a[], int len)
{
    int i, j, temp;

    for (i = 0 ; i < len - 1 ; i++)
    {
```

```

    int min = i; // 记录最小值，第一个元素默认最小
    for (j = i + 1; j < len; j++) // 访问未排序的元素
    {
        if (a[j] < a[min]) // 找到目前最小值
        {
            min = j; // 记录最小值
        }
    }
    if(min != i)
    {
        temp=a[min]; // 交换两个变量
        a[min]=a[i];
        a[i]=temp;
    }
    /* swap(&a[min], &a[i]); */ // 使用自定义函数交换
}

/*
void swap(int *a,int *b) // 交换两个变量
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
*/

```

• 插入排序

- 插入排序（英语：Insertion Sort）是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用in-place排序（即只需用到 $O(1)$ 的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

```

void insertion_sort(int arr[], int len){
    int i,j,temp;
    for (i=1;i<len;i++){
        temp = arr[i];
        for (j=i;j>0 && arr[j-1]>temp;j--){
            arr[j] = arr[j-1];
        }
        arr[j] = temp;
    }
}

```

• 希尔排序

- 希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。希尔排序是非稳定排序算法
- 希尔排序是基于插入排序的以下两点性质而提出改进方法的
 - 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率

- 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位

```
void shell_sort(int arr[], int len) {
    int gap, i, j;
    int temp;
    for (gap = len >> 1; gap > 0; gap = gap >> 1)
        for (i = gap; i < len; i++) {
            temp = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > temp; j -= gap)
                arr[j + gap] = arr[j];
            arr[j + gap] = temp;
        }
}
```

- 归并排序
 - 把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。
 - 可从上到下或从下到上进行。
- 迭代法

```
int min(int x, int y) {
    return x < y ? x : y;
}

void merge_sort(int arr[], int len) {
    int* a = arr;
    int* b = (int*) malloc(len * sizeof(int));
    int seg, start;
    for (seg = 1; seg < len; seg += seg) {
        for (start = 0; start < len; start += seg + seg) {
            int low = start, mid = min(start + seg, len), high = min(start + seg + seg, len);
            int k = low;
            int start1 = low, end1 = mid;
            int start2 = mid, end2 = high;
            while (start1 < end1 && start2 < end2)
                b[k++] = a[start1] < a[start2] ? a[start1++] : a[start2++];
            while (start1 < end1)
                b[k++] = a[start1++];
            while (start2 < end2)
                b[k++] = a[start2++];
        }
        int* temp = a;
        a = b;
        b = temp;
    }
    if (a != arr) {
        int i;
        for (i = 0; i < len; i++)
            b[i] = a[i];
        b = a;
    }
}
```

```
    free(b);
}
```

- 递归法

```
void merge_sort_recursive(int arr[], int reg[], int start, int end) {
    if (start >= end)
        return;
    int len = end - start, mid = (len >> 1) + start;
    int start1 = start, end1 = mid;
    int start2 = mid + 1, end2 = end;
    merge_sort_recursive(arr, reg, start1, end1);
    merge_sort_recursive(arr, reg, start2, end2);
    int k = start;
    while (start1 <= end1 && start2 <= end2)
        reg[k++] = arr[start1] < arr[start2] ? arr[start1++] :
arr[start2++];
    while (start1 <= end1)
        reg[k++] = arr[start1++];
    while (start2 <= end2)
        reg[k++] = arr[start2++];
    for (k = start; k <= end; k++)
        arr[k] = reg[k];
}
void merge_sort(int arr[], const int len) {
    int reg[len];
    merge_sort_recursive(arr, reg, 0, len - 1);
}
```

- 快速排序
 - 在区间中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。
- 迭代法

```
typedef struct _Range {
    int start, end;
} Range;
Range new_Range(int s, int e) {
    Range r;
    r.start = s;
    r.end = e;
    return r;
}
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
void quick_sort(int arr[], const int len) {
```

```

    if (len <= 0)
        return; // 避免len等於負值時引發段錯誤 (Segment Fault)
    // r[]模擬列表,p為數量,r[p++]為push,r[--p]為pop且取得元素
    Range r[len];
    int p = 0;
    r[p++] = new_Range(0, len - 1);
    while (p) {
        Range range = r[--p];
        if (range.start >= range.end)
            continue;
        int mid = arr[(range.start + range.end) / 2]; // 選取中間點為基準點
        int left = range.start, right = range.end;
        do
        {
            while (arr[left] < mid) ++left; // 檢測基準點左側是否符合要求
            while (arr[right] > mid) --right; // 檢測基準點右側是否符合要求

            if (left <= right)
            {
                swap(&arr[left], &arr[right]);
                left++; right--; // 移動指針以繼續
            }
        } while (left <= right);

        if (range.start < right) r[p++] = new_Range(range.start, right);
        if (range.end > left) r[p++] = new_Range(left, range.end);
    }
}

```

- 递归法

```

void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}

void quick_sort_recursive(int arr[], int start, int end) {
    if (start >= end)
        return;
    int mid = arr[end];
    int left = start, right = end - 1;
    while (left < right) {
        while (arr[left] < mid && left < right)
            left++;
        while (arr[right] >= mid && left < right)
            right--;
        swap(&arr[left], &arr[right]);
    }
    if (arr[left] >= arr[end])
        swap(&arr[left], &arr[end]);
    else
        left++;
}

```

```
    if (left)
        quick_sort_recursive(arr, start, left - 1);
    quick_sort_recursive(arr, left + 1, end);
}
void quick_sort(int arr[], int len) {
    quick_sort_recursive(arr, 0, len - 1);
}
```