C程复习

- ## 基础语法
  - ### 标识符
    - 由字母，数字，下划线构成
    - 不能以数字开头
    - 区分大小写
    - 不能与关键字重名
    - 空白字符：`'\n'`、`'\t'`、`' '`
  - ### `main()`函数
  - ### 输入输出流
    - `printf ()`
      - 位数控制：
        - `%md`：右对齐
        - `%ms`：最多输出m个字符的字符串
        - `%-md`：左对齐
        - `%+md`：正数显示`+`
        - `%0md`：左边如果有空格用`0`补齐
      - 小数位数控制：`%.nf`（四舍五入）
      - 各种数据类型的输出：
        - 整数
          - `%d`：带符号十进制整数
          - `%ld`：long型
          - `%o`：八进制（无前缀`0`）
          - `%x`或`%X`：十六进制（无前缀`0x`）

            要输出八进制或十六进制前缀，`%#x`或`%#o`
            - 八进制和十六进制应该没有负数，如果输出负数，则会输出它的补码
          - `%u`：无符号十进制整数
        - 浮点数
          - `%f`：小数形式，默认显示前6位
          - `%lf`：double 型（输入时需要，输出时只要`%f`即可）
          - `%e`或`%E`：指数形式，默认显示前6位
          - `%g`或`%G`

        - 字符型
            - `%c`（`''`）
            - `%s`（`""`）
        - 地址（指针）
            - `%p`：**十六进制**地址
        - 注意保持变量与相对应的格式字符串类型一致，否则无法正常输出
        - 返回值：成功返回输出的字符数，失败返回 `EOF` (-1)
- `scanf()`
    - &（除了指针（字符串）都要加上）
        - 看一个特殊例子 `char c[100]; scanf("%c", c);`
            - 貌似不太对的样子，但它能读入 `c[0]` 位置的字符
    - 位数控制
    - 各种数据类型的输入
    - 格式控制字符串
        - 注意一个附加格式字符：`*`，它表示跳过该输入项，不传递任何值
        - 格式控制字符串中如包含一个空格或 `/n`，则它可以匹配一个或多个空白字符
    - 返回值为成功读入的项个数，什么都没读到返回值为0
        - 但如果 `scanf()` 第一次读到的是 `EOF`，那么返回值为 `-1`（如果是后面读到 `EOF`，返回值还是正常的）
    - 一种情况：如果对 `double` 型变量，输入时使用 `%f`，则无法正常读取该变量，该变量最终会等于0.00...
- `getchar()`
    - 格式：`char ch = getchar();`
    - 与 `scanf()` 混用时注意：如果之前用 `scanf()` 读入一个数字，后面想再用 `getchar()` 读入一个字符，中间往往会有空白字符，因此在读字符之前，先用 `getchar()` 把空白字符读掉，再读字符
- `putchar()`
    - 格式：`putchar(ch);`
    - 返回值：成功返回该字符，失败返回 `EOF`
- `gets()`
    - 格式：`gets(s);`
    - 换行符停止，替换为 `'\0'`
    - 返回值：成功返回 `s`，失败返回 `NULL`
- `puts()`
    - 输出字符串并换行（把 `'\0'` 转换成 `'\n'`）

- 返回值：成功返回换行符，失败返回 `EOF`

变量与数据类型

变量

按作用域分类

- 局部变量
  - 函数内或语句块内（被花括号包起来的语句）定义
  - 没有加 `static`，若没赋初值，默认为随机数
  - `main()` 中定义的变量只能在 `main()` 中使用
  - 不同函数可以使用同名变量
  - 函数形参也是局部变量
- 全局变量
  - 函数外部定义，且只能定义一次
  - 没有赋初值的话，会自动赋一个默认值（0,0.0或 `NULL`）
  - 若与局部变量重名，会优先调用同名局部变量
  - 可通过 `extern` 被其他文件使用，注意重复定义的问题
  - 尽量不要使用（有诸多缺点）

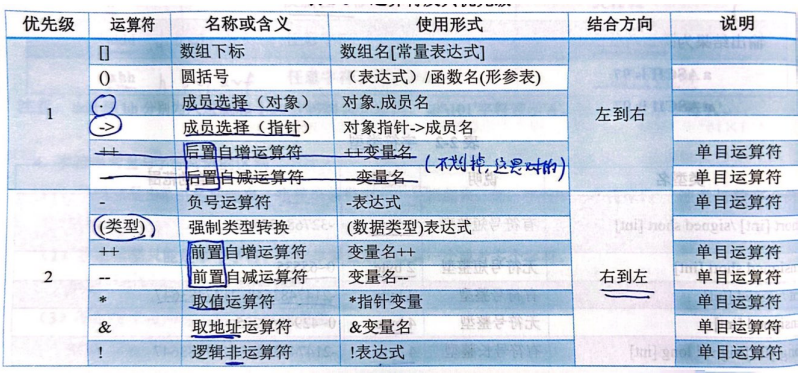
按生命周期分类

- **自动** `auto(默认省略)`
  - 平时定义的局部变量默认为自动变量
  - 动态存储，若没有初始化，其值不确定；其所在函数结束后即被释放
- 寄存器 `register`（现在不必考虑）
  - 类似 `auto`
- **静态** `static`（重点）
  - 格式：`static 类型名 变量表;`
  - 静态存储，若没初始化，其值为默认值（0）
  - 生命周期：从整个程序开始，到整个程序结束
  - 静态局部变量
    - 用于函数内部
    - 在被编译时赋初值，且仅赋一次
    - 作用域：仅函数内部
  - 静态全局变量
    - 函数外
    - 作用域为整个文件
    - 不会被其他文件使用

        - 没有要求尽量少用
        - 还可以用在函数上，具体见“函数”一节
    - 外部 `extern`
        - 单个文件
        - 多文件（常用）
            - 一个文件定义该外部变量，其余文件使用 `extern` 引用该变量，避免重复定义的问题

- **数据类型**
    - **整数**
        - long，后缀 `l` 或 `L` （一般为4B）
        - int（一般为4B）
        - unsigned (int)，后缀 `u` 或 `U` （可以与 `l` 或 `L` 组合）
        - short（一般为2B）
        - char（一般为1B）[-128, 127]
        - 看这个例子： `int a = (1ll<<31);`
            - 它的结果是？—— `-2^31`
            - 解析： `int` 的取值范围为(以4B为例) `-2^31 ~ 2^31-1` ， `1ll<<31` 的值本来应该为 `2^31` ，但这个值超出范围。而事实上“上下限是连通的”（即超出上限会回到下限，超出下限回到上限），所以是 `-2^31`
            - 补充： `for (int i = 0; i>=0; i++)` ，看似是个死循环，实则当 `i` 的值超过 `2^31-1` 时， `i` 就回到下限（是负数 `-2^31` ），可以退出循环的
    - **字符**
        - char（一般为1B）[-128, 127]
        - ASCII字符（本质是整数）
            - 常见的：‘0’：48，‘A’：65，‘a’：97，‘ ’：32
        - 重点：转义字符
            - 常见： `\n` 、 `\t` 、 `\r` 、 `\a` 、 `\b` 、 `\f` 、 `\'` 、 `\"` 、 `\\`
            - `\xhh` ——用十六进制ASCII码表示任意字符
            - `\ddd` ——用八进制ASCII码表示任意字符（如果题目中出现8、9这些数字，那肯定有问题）
                - 注意： `'\0'` 、 `0` 、 `'\00'` 、 `'\000'` 表示同一个字符
                - `\xhh` 、 `\ddd` 的大小不会超过2^7-1
    - **浮点数**
        - float（一般为4B）
        - double（一般为8B）

        - 注意指数形式的表达：**[±][整数部分][.][小数部分][e/E±n][后缀]**，数字部分不可全部省略，且后缀部分一定是整数
        - 虽然浮点数的最大精度有限，但并不是说精度范围外的数字都是随机产生的，它们在计算机内按照一定的规则存储（只是看起来这些数字超出我们的预期而已）
            - 所以多次运行同一程序，这样超精度范围的数的结果保持一致
    - **数据的存储**
        - **整数**
            - 一般以二进制补码的形式存储
            - 原码、反码、补码的关系
                - 正数原码、反码、补码相同，符号位为0
                - 负数
                    - 原码：符号位为1，其余各位表示数值的绝对值
                    - 反码：符号位为1，其他各位对原码取反（二进制）
                    - 补码：反码+1
            - 进制间的转换
        - **浮点数（了解）**
            - 符号位、阶码（指数）、尾数
            - 以4字节的float型为例，它的符号位共1位，指数共8位，尾数共23位（1.00001*2^3中，指数为3，尾数为1.00001）
    - **类型转换**
        - **自动类型转换**
            - 有一套转换规则，这里先空着
            - 运算类型不一致（但不同类型的指针不会做自动类型转换）
            - 函数实参形参类型不一致
        - **强制类型转换**
            - 格式：`（类型名） 表达式`
- **运算符与表达式**
    - **运算符**
        - **优先级表(熟记！！！)**



优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	
	()	圆括号	（表达式）/函数名(形参表)		
	.	成员选择（对象）	对象.成员名		
	->	成员选择（指针）	对象指针->成员名		
	++	后置自增运算符	++变量名		单目运算符
	--	后置自减运算符	--变量名 (不划掉，这是对的)		单目运算符
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		单目运算符
	++	前置自增运算符	变量名++		单目运算符
	--	前置自减运算符	变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符

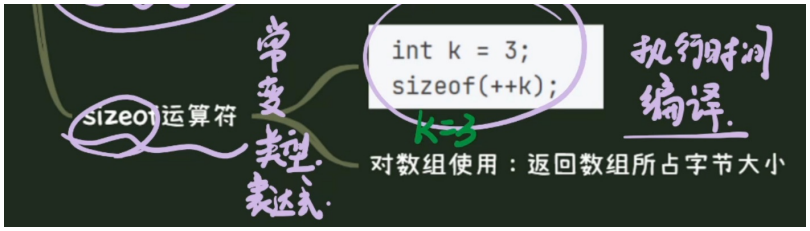
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数（取模）	整型表达式/整型表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!= 表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2: 表达式 3	右到左	三目运算符
14 所有的赋值	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	*=	乘后赋值	变量*=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	|=	按位或后赋值	变量|=表达式		
15	,	逗号运算符	表达式,表达式,...	左到右	从左向右顺序运算

注：单目运算符是指所需运算量为一个的运算符；双目运算符指所需运算量为两个的运算符。

- 结合关系
- i++ 与 ++i
  - `i++`：先用 `i`，再加一
  - `++i`：先加一，再用 `i`
  - 注意：在 `if(…)`、`while(…)`、`for(…)` 语句中的 `…` 部分出现 `i++` 表达式时，进入主体语句（循环体）后 `i` 的值已经加上1了
- 算术运算
  - `+`、`-`、`*`、`/`、`%`
  - 其中 `%` 只能用于两个整数，余数正负取决于被除数
  - `/` 两边如果都是整数，则进行整除运算。所以想对两个整数进行实除的话，要么用强制类型转换（变量），要么将常量改写成形如 `5.0` 的样子
- 位运算(全部转化成二进制再计算)
  - 与 `&`（有0出0，全1出1）

- 或 `|` （有1出1，全0出0）
- 取反 `~` （1->0，0->1）
- 异或 `^` （相同为0，不同为1）
- 左移 `<<`
  - `num << n == num * pow(2,n)`
  - 移出左边界的所有位都丢失，右侧新增加0
- 右移 `>>`
  - `num >> n == num / pow(2,n)`
  - 正数为左移的相反情况
  - 负数则一律补1（因为符号位为1），而不是0（这是其中一种情况，要取决于编译器的实现）

- **逻辑表达式**
  - **本质上是 0 和 非0 的判断**
  - **短路（重要）**
    - `||` 左边表达式为1，则不判断（计算）右边的表达式
    - `&&` 左边表达式为0，则不判断（计算）右边的表达式
- **三目运算符（条件表达式）**
  - 格式：`（表达式1）？（表达式2）：（表达式3）`
  - 整个条件表达式的值会进行自动类型转换（如果出现2和3类型不一致的情况）
- **赋值表达式**
  - 右结合
  - 优先级倒二
  - 复合赋值运算符
- **逗号表达式**
  - 格式：`表达式1，表达式2，.....，表达式n`
  - 从左往右算，返回最后一个表达式的值
  - 优先级倒一
- sizeof 运算符
  - 例子：



- 对数组使用：返回数组的长度
  - 比如 `int a[10] = {1, 2, 3}; printf("%d", sizeof(a)); // 输出结果：10`
  - 小技巧，如果数组（这里假设该数组为 `int a[]` ）初始化时没有标出长度，可以用 `sizeof(a) / sizeof(int)` 计算它的元素个数
- 对结构体使用 `sizeof` ，因为字节对齐，所以最终结果会大于等于理论值。如果选择题中遇到一个选项说 `==` ，则要视其他选项而定。
- 对指针使用 `sizeof` ，其结果一定是 `4或8`
- 在自定义函数（一个形参是数组 `int a[]` ，其实是 `int *` 指针）对这个形参 `a` 使用 `sizeof` ，返回的不是数组的大小，而是指针的大小
- 例子：
  - `sizeof(5.0)` 的结果为：8（浮点数默认double型）

分支结构

if、else、else if

- `else` 的匹配——与最近的没有与 `else` 匹配的 `if` 匹配
- 注意题目中判断语句表达式的等号问题（ `==` 或 `=` ）

switch

- `switch` 后可跟数值或字符常量、变量、表达式
- `case` 后只能跟整型常量（常量表达式），并且每个 `case` 的常量必须保证不同
  - 貌似变量前加 `const` 好像没问题（我不确定，还没试过）
- 注意 `break` ——如果 `case` 语句中没有 `break` ，则会继续执行后面的语句

条件

- 本质是0或非0

循环结构

基本语法

for

- 三个语句的执行顺序： `for (表达式1;表达式2;表达式3)`
- 表达式1 -> 表达式2 -> 循环体 -> 表达式3 -> 表达式2（真） -> 循环体 -> ..... -> 表达式3 -> 表达式2（假） -> 退出循环
- 三个表达式均可省略，但分号不可省（**表达式2**省略，循环条件永远为真）

        - 每个表达式内可采用逗号表达式，实现多种运算
    - while
        - `while (1)` 表示死循环
    - do while
        - 至少循环1次
    - break
        - 跳出1层循环
        - 还可用于switch语句中
    - continue
        - 结束本轮循环，进入下一轮循环，而不是结束整个循环
        - 跳过高亮部分的“循环体”那一块
    - goto（了解，可用于退出多层循环，平时别用）
- **for 执行后变量的值**
    - **是否 break 结束的判断**
- **嵌套循环的分析**
- **分析方法：单步执行**
- **注意判断条件中的 `=` 和 `==` 问题**

数组

- **数组定义**
    - 格式：`类型 数组名[整数常量表达式]`
    - **数组名**是个地址常量，表示首元素的地址
    - 下标从0开始，不要让下标越界！（编译器不会报错）
- **数组初始化**
    - **数组的初值**
        - 全部初始化——可以省略长度
            - `int a[] = {1, 2, 3, 4, 5};`
        - 不初始化（自动变量）——随机数
        - 部分初始化——前面几个元素会赋初值，后面的元素均为0，长度不能省略
            - `int a[5] = {1, 2, 3};  // a[3] == a[4] == 0`
        - 静态 / 全局——不初始化，所有元素默认为0
            - `static(extern) int a[3]; //  a[0] == a[1] == a[2] == 0`
    - **隐式定义长度（即省略长度）：当所有元素都赋了初值时** √

    - **显示定义长度：常量**
    - **二维数组**
        - 数组的数组
        - 部分初始化
        - 省略数组第一维（行）长度的初始化
            - `int a[][2] = {{1, 2}, {3, 4}, {5. 6}};`
        - **不能省略第二维（列）**
        - 一维初始化
            - `int a[3][2] = {1, 2, 3, 4, 5, 6};`
        - 省略第一维长度的一维初始化
    - **数组存放是线性的（二维也一样）**
- **数组的使用**
    - `数组名[下标]`——该下标可以是常量，也可以是变量
    - 不能直接对整个数组赋值（除初始化）
    - 常见用法：排序（选择、冒泡等）、查找（二分查找）
- **字符串·**
    - **本质：字符数组**
        - 这是个一维字符数组，但它不是字符串：`char s[5] = {'H', 'a', 'p', 'p', 'y'};`
        - 以下几种情况都是**字符串**
            - `static char s[6] = {'H', 'a', 'p', 'p', 'y'};`
                - 部分初始化，后面的元素都是0，正好充当字符串的结尾
            - `static char s[6] = {'H', 'a', 'p', 'p', 'y', 0};`
            - `static char s[6] = {'H', 'a', 'p', 'p', 'y', '\0'};`
            - `static char s[6] = "Happy";`
            - `static char s[6] = {"Happy"};`
        - 字符串 = 有效字符 + `'\0'`（无视`'\0'`后面的东西），用双引号括起来
        - **字符串常量**
            - 实质：一个指向该字符串首字符的指针常量
            - 可以使用下标[]，像访问数组元素一样访问每个字符
    - **由字符指针和 '\0' 决定长度**
        - `printf()` 输出字符串时直到遇到`'\0'`时停止
    - **数组长度与字符串长度**
    - **字符数组与字符指针**

- 以这个为例：`char sa[] = "Hello world"; char * sp = "Hello world";`
- **字符数组**
  - 只能改变单个字符的值，不能对整体进行赋值
    - `sa[6] = 'W'; // 合法`
    - `sa = "hello"; // 非法`
    - 只能用`strcpy()`函数赋值：`strcpy(sa, "hello");`
- **字符指针**
  - 可以用赋值语句直接改变它的值，让它指向新的字符串
    - `sp = "hello"; // 合法`
  - 注意：为避免引用未赋值的指针所造成的危害，在定义指针时，可以先这样做：`char *p = NULL;`

- **字符串操作函数**
  - **输入输出**
    - `scanf()`和`printf()`用`%s`占位符，字符串名前无需加`&`
    - 注意`scanf()`遇空白字符（前面提到过）结束字符串输入，并自动添加`'\0'`；而`printf()`遇`'\0'`结束
    - `char * gets(char *s);`：读一个字符串，遇回车停止，因此可以读入空格，并自动添`'\0'`
    - 若成功读入，返回值为字符串首字符地址，否则返回`NULL`
    - `int puts(char *s);`：输出字符串，遇到`'\0'`将其换成`'\n'`，即输出时自动换行
    - 若成功输出，返回`'\n'`，否则返回`EOF`
  - 以下函数需要使用`#include <string.h>`
  - **长度**
    - `unsigned int strlen(char * s);`返回字符串有效字符个数，即字符串中第一个`'\0'`之前（不包括自身）的所有字符个数
  - **复制（赋值）**
    - `char * strcpy(char * s1, char * s2);`：将`s2`的内容复制到`s1`上，`s1`原本内容会被覆盖
    - 确保`s1`长度不小于`s2`
  - **连接**
    - `strcat(char * s1, char *s2`：将`s2`接至`s1`末尾，此时`s1`原本的`'\0'`被放置在新的`s1`末尾
    - 确保`s1`足够大
    - 不允许使用算术运算符连接两个字符串（~~python √~~）

- 比较
  - `int strcmp(char * s1, char * s2);`：比较两个字符串的ASCII值大小
  - 规则：逐位比较，直到遇到不同的字符或 `'\0'` 为止。（ASCII值大的，或长度更长的，则为更大的字符串）
  - 返回值
    - 0：`s1 == s2`
    - 正数：`s1 > s2`
    - 负数：`s1 < s2`
  - 不能使用关系运算符比较大小（~~python √~~）

指针

- 作用：访问内存和操作地址，间接访问变量；函数传递指针节省内存空间，提升运行效率
- 大小：4字节（32位系统）或8字节（64位系统）
- 常规内容
  - 定义
    - `类型名 *指针变量名`
    - 可以在同一行初始化多个指针，但要记得在每个指针变量前加 `*`
    - 注意：指针类型与它所指向变量的类型必须相同
    - 若指针未初始化，不能用它来进行scanf()等操作
  - 指针与下标访问的等价表达
    - `int a[10] = {1, ...(懒得写), 10}, *p; p = a;`，此时有以下关系
      - `*(a + i) == a[i], *(p + i) == p[i]`
      - `p + i == &a[i] == &p[i]`
  - 基本操作
    - 取地址运算符 `&`，得到变量的地址
      - `int a = 3, *p; p = &a;`
    - 间接访问运算符 `*`
      - 还是上面的例子，此时 `*p == a`
      - 若修改 `a` 或 `*p` 的值，两者均同时变化
      - `&*p == &a, *&a == a`
    - 指针与常数
      - `p++` / `++p`：指针加1，即往后移动指针所指类型所占内存大小长度（单位：Byte）（的1倍）的距离（当然两者略有区别）
      - `p+=3` 可以移动 3倍 的类型大小的距离，而 `p+3` 不会改变指针大小，除非 `p = p + 3;`

        - 也可以减一个常数，代表从当前位置向前移动
        - 注意下面的混搭关系：
            - `*p++`：先用`p`所指向的变量，再对`p`加一，此时`p`已不再指向原来的位置
            - `(*p)++`：先用`p`所指向的变量，再对`p`加一，`p`的指向的位置不变
            - 记住：`++`的优先级高于`*`
    - 指针之间
        - 赋值
            - 相同类型的两个指针
            - 赋值好的指针才能正常引用，否则后果很严重，可以先给指针赋一个空指针`NULL`
            - 不能将数值直接赋给指针
        - 减法
            - 使用前提：两个指针指向同一个数组或字符串
            - 结果：两个指针间的”距离“（元素个数）
        - 使用关系运算符比较大小
- **字符指针（见“字符串”一节）**

- **进阶内容**
    - **指针数组**
        - 一个数组，每个元素都是指针
        - 格式：`类型名 *数组名[数组长度];`，比如`char *s[3] = {"red", "yellow", "blue"};`
        - 常用例子：一个字符串（字符指针）数组，（`printf("%x", s[1]);`打印十六进制地址）
    - **数组指针**
        - 格式：`类型名 (*数组名)[数组长度];`
        - 例子：
            - `int a[2][3] = {1,2,3,4,5,6}; int (*p)[3];  p = a + 1;`
            - 此时`*p == a[1] == &a[1][0]`
    - **多级指针**
        - 二级指针：`类型名 **变量名;`
        - 比如：`int a = 3, *p, **pp; p = &a; pp = &p;`
        - 二维数组名本质上是个二级指针常量
        - 比如：`int b[3][2]={(略)}, i; pp = b`
        - 此时`*(p + i) == b[i] == &b[i][0],  **(p + i) == b[i][0]`

- 二维数组中的指针等价关系（参考）

二级指针		一级指针			数组元素		
a	&a[0]	*a	a[0]	&a[0][0]	**a	a[0][0]	*(a[0]+0)
		*a+1	a[0]+1	&a[0][1]	*(*a+1)	a[0][1]	*(a[0]+1)
		*a+j	a[0]+j	&a[0][j]	*(*a+j)	a[0][j]	*(a[0]+j)
a+1	&a[1]	*(a+1)	a[1]	&a[1][0]	**(a+1)	a[1][0]	*(a[1]+0)
		*(a+1)+1	a[1]+1	&a[1][1]	*(*(a+1)+1)	a[1][1]	*(a[1]+1)
		*(a+1)+j	a[1]+j	&a[1][j]	*(*(a+1)+j)	a[1][j]	*(a[1]+j)
a+i	&a[i]	*(a+i)	a[i]	&a[i][0]	**(a+i)	a[i][0]	*(a[i]+0)
		*(a+i)+1	a[i]+1	&a[i][1]	*(*(a+i)+1)	a[i][1]	*(a[i]+1)
		*(a+i)+j	a[i]+j	&a[i][j]	*(*(a+i)+j)	a[i][j]	*(a[i]+j)

  - 对于二级指针变量，它可以指向同类型一级指针，但不能直接指向同类型变量，这会导致类型不匹配或未定义行为
- **函数指针**
  - 函数名本身为代表函数的入口地址，可用函数指针指向它
  - 格式：`类型名 (* 变量名)(形参表)`
  - 当函数指针与某个函数的返回值类型和形参表相同时，可以使用赋值语句
    - `int f(int, int); int (*pfun)(int, int); pfun = f;`
    - 调用：`f(3, 5) == (*pfun)(3, 5)`
  - 函数指针更多用来作为函数的参数
- **动态内存分配**
  - 优点：可以提高使用内存效率，减少内存空间浪费
  - 常用函数
    - 动态内存**分配**函数 `void * malloc(unsigned int size);`
      - 功能：在内存中分配一块 `size` 长度（一般用 `sizeof(类型名) * 所需元素个数（可以是变量）` 来计算）的空间（里面不可能是指针）
      - 分配成功返回该空间起始位置的地址，否则返回 `NULL`
      - 应当将该返回值通过强制类型转换赋给一个指针，比如 `int *p, n = 5; p = (int *) malloc(n * sizeof(int));`
    - `void * calloc(unsigned n, unsigned size);`，与 `malloc()` 的不同
      - 形参表略有不同（`calloc()` 的 `n * size` == `malloc()` 的 `size`）
      - 分配完内存后将所有元素全部初始化0
      - 其他没什么大的区别
    - 动态内存**释放**函数 `void free(void * prt);`
      - 切记：一定与 `malloc()` 搭配使用，程序结束前一定要用到这个函数！！！
    - `void realloc(void *ptr, unsigned size);`（不重要）
- **链表（重点，但现阶段不需深入了解）**
  - 一般通过结构（嵌套）和指针实现

        - 每个节点包含一个数据域 `data` 和一个指针域 `next` （一般指向下一个节点，最后一个节点指针域为 `NULL` ）
        - 类型：单向、双向、循环
        - 头指针 `head` 是链表的入口，十分重要
        - 需要使用**动态内存分配**
        - 优点：内存使用效率高，插入、删除操作方便
        - 缺点：查找慢
        - 常用操作：创建，遍历，插入，删除

- **函数**
    - **分类：**
        - 库函数（记得相应的头文件，这里就不列出来了），自定义函数
        - 无参函数，有参函数
    - **组成部分**
        - **返回值（返回类型）**
            - 无返回值使用 `void` ，这时函数只能单独使用，不能用于各种运算等
            - 漏写的话默认 `int` （应该不会出现这种情况吧……）
            - 如果要返回指针，该指针应该为主调函数或静态变量的地址，而不是该函数内的指针。可以返回0（空指针 `NULL` ）
        - **函数名**
            - 两个函数不能重名（无论是定义还是声明），否则视为函数重复定义，这是不行的
        - **参数列表**
            - 无参数使用 `void` ，但括号不能省
            - 每个形参的类型必须分别写明
    - **函数原型（声明），函数定义的位置**
        - 不能嵌套定义函数，但允许函数之间相互调用
        - 推荐先写原型，再写定义
        - 原型后面有分号！！！定义没有
    - **函数调用**
        - **实参（函数调用）与形参（函数定义中的参数表）**
            - 形参在被调用时才会分配内存，函数结束即刻释放
        - **值传递**
            - 单向数据传递（实参 -> 形参），即形参的改变不影响实参的值
            - 简单说是实参的值“复制”给形参，两者占用不同的内存空间

- **指针（地址）传递**
  - 形参改变影响实参
  - 因此可用数组名作为函数参数，改变形参数组元素就能改变实参数组元素
    > 最好再传一个表示数组大小的参数
  - 可通过这个方法“返回”多个变量
  - 常用例子：两个数交换的函数
  - 某个形参形如 `int a[]`，则它其实是个指针，而非整个数组
- 注意：如果参数表有多个参数，则会从最后一个参数开始逆序传递（即先计算最后一个参数（表达式），再往前计算）
  - 不论是自定义函数还是库函数（比如 `printf()` ）都遵循这个原则
- **递归调用**
  - 递归表达式
  - 终止条件
  - 函数递归调用自身时，每次调用都会的到一个与以前变量集合不同的新的变量集合

- return 语句
  - 允许函数中有多个 `return`，但每次调用只使用一个 `return`，且只能 `return` 一个值
  - 无返回值可以不写
- **内部函数与外部函数**
  - **内部**
    - 格式：`static 返回类型 函数名(形参表);`
    - 只能在本文件使用该函数，其他文件无法使用
  - **外部**
    - 格式：`extern 返回类型 函数名(形参表);`
    - 可被其他文件调用
    - 一般的函数默认为外部函数，所以 `extern` 往往会省略

- **结构**
  - **定义结构**
    - **常规定义**
      - `struct 结构名{类型名 结构成员1; 类型名 结构成员2; ...(略) };`
      - 注意后面的分号
      - 可嵌套定义，当然被嵌套的结构要先定义
      - `struct 结构名 == 类型名`
      - 一般：`struct student info; //这里假设struct student 已定义`

        - 混合定义：`struct student{(略)}info; //这里sturct student 刚定义`
    - **无名结构定义(慎用！)**
        - `struct {(略)}info;`
    - **初始化**
        - `struct student info = {"qzy", 0, 0, 0};`
        - 和数组类似，初始化严格按顺序进行，如果部分初始化，则未被初始化的部分会被赋上默认值 `0`
- **访问成员**
    - `.` **访问**
        - 嵌套结构使用多个 `.`
    - `->` **访问（指针）**
        - `struct student info = {(略)}, *p; p = &info`
        - `info.num == (*p).num == p->num`
- **结构的整体赋值（前提：两个结构类型完全相同）**
- **结构数组**
- **传参**
    - 整个结构（简单，但效率低）
    - 结构指针（效率高，并且结构数组只能这么传）
- **位字段结构（不考）**
- **联合 `union` （不考）**

编译预处理

- **宏定义 `#define`**
    - 本质：简单的字符替换，不进行语法检查，不分配内存空间
    - 所以记得多加括号 （否则会影响运算顺序）
    - **不带参**
        - 格式：`#define 宏名 字符串`
        - 有效范围：从定义开始，至整个程序结束，可以用 `#undef` 提前终止
        - 可以嵌套使用
        - 可作为符号常量
    - **带参**
        - 格式：`#define 宏名(形参表) 字符串`
        - 注意：宏名与括号间**没有空格**

- 与有参函数的区别：
  - 函数调用在**运行**时处理，分配临时内存；宏展开在**编译**时进行，不分配内存
  - 函数要定义类型（实参、形参、返回值）；宏**不考虑类型**
  - 宏展开“**占空间不占时间**”，函数“占时间不占空间”
    > 空间：代码长度，时间：运行时间
- 记得多加括号 （否则会影响运算顺序）
- 可作为简单的函数

- **其他：**
  - 无需分号
  - 可以使用 `'\'` 在句末进行替换

- **文件包含** `#include`
  - `#include <stdio.h>`

    通常使用系统提供的**标准头文件**
  - `#include "max.c"`
  - `#include "max.h"`

    先在**当前工作文件夹**寻找被包含的文件，若找不到，再到系统文件夹中寻找
  - 本质：在编译预处理时将被包含文件的全部内容复制并插入至 `#include` 命令处
  - 可嵌套使用
  - 被包含文件的**全局变量**在本文件中生效，而 `static` 的变量和函数不会生效
- **条件编译**
  - 注：这里的 `#else` 可省略

```
#ifdef 标识符
    程序段1
#else
    程序段2
#endif
//功能：如果标识符定义过，编译程序段1，否则编译程序段2

#ifndef 标识符
    程序段1
#else
    程序段2
#endif
//功能：如果标识符未定义，编译程序段1，否则编译程序段2

#if 表达式
    程序段1
#else
    程序段2
#endif
//当表达式的值非0，编译程序段1，否则编译程序段2
```

  - 这里还漏了 `#elif` 的情况，类比 `else if`
  - 注意条件编译与条件语句if-else完全不同，后者的分支都会被生成到目标代码中，而前者只有其中一条分支被生成到目标代码中，另一段被舍弃。且#if 后不是程序表达式，只能是宏名（因为编译预处理无法计算表达式，只有程序运行时才做计算）

- 头文件的 `#define` 保护
  - 如图

```
//header.h
#ifndef  _NAME_
#define  _NAME_
//头文件内容
#endif
```

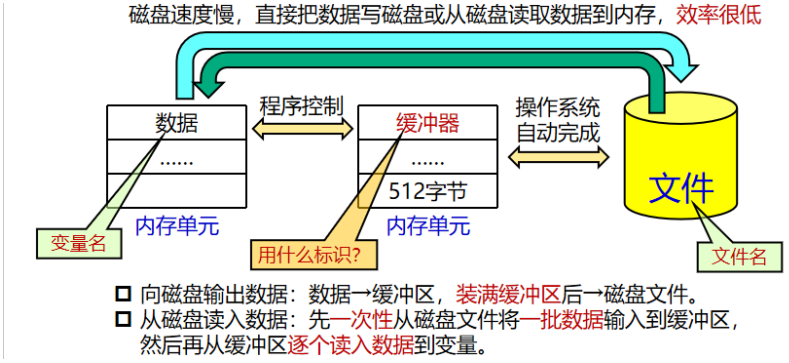
  - 作用：可以保证函数只定义一次
- 好处：1. 目标代码精简，不包含无关代码；2. 系统代码保护性更好

多文件编程

- `.h` 文件一般可以存放函数和变量的声明，存放全局统一的宏定义，包含编译器自带的头文件；但不能存放函数与变量的定义，因为这会导致函数重复定义的错误
- 在编译时，编译器将所有 `.c` 文件链接起来，生成 `.o` 文件，然后一起编译
- `.c` 文件不一定包含 `main()` 函数
- 头文件不是单独编译的，它通过 #include 将整个文件“复制”到源程序中，然后再编译

文件

- c语言把文件看作数据流，并将数据按一维方式组织存储
- 文件类型
  - 文本文件
    - ASCII字符
    - 可用记事本查看
    - 有单独的文本行，会自动生成行结束标记（EOF）
  - 二进制文件
    - 机器代码
    - 目标文件（ `.o` ）、可执行文件（ `.exe` ）
    - 不会分成单独行，无行结束标记
- 缓冲文件系统



  - 要考这个的话，我******

- 打开文件
  - 文件指针
    - 对于每一个要操作的文件，要在程序中定义一个指向 `FILE` 类型（本质是通过 `typedef` 改变名称的一个 `struct` ）的指针变量，比如 `FILE *fp;`
    - 文件指针指向文件类型结构，通过 `fp->curp` 可以指示文件缓冲区中数据存取的位置（了解即可）
    - 一个指针变量只能指向一个文件
  - 格式：
    - `FILE *fopen(const char * filename, const char * mode);`
    - 比如：`fp = fopen("file.txt","r");`
    - 注：`filename` 中表路径的斜线必须使用双反斜杠 `\\` ，比如C:\\one\\file.c（一个反斜杠是转义字符，所以要双反斜杠）（绝对路径）
      - 貌似双反斜杠可以用斜杠‘/’代替
      - 相对路径：
        - `/` 表示根目录
        - `..` 表示上级目录
    - 如果成功打开文件，则返回 `FILE` 结构地址，否则返回 `NULL` (空指针)
  - 操作类型

mode	处理方式	指定文件不存在	指定文件存在	含义
r	只读	出错	正常打开	以读方式打开一个文本文件
w	只写	建立新文件	文件原有内容丢失	以写方式打开一个文本文件
a	追加	建立新文件	在文件原有内容后追加	以追加方式打开一个文本文件
rb	只读	出错	正常打开	以读方式打开一个二进制文件
wb	只写	建立新文件	文件原有内容丢失	以写方式打开一个二进制文件
ab	追加	建立新文件	在文件原有内容后追加	以追加方式打开一个二进制文件
r+	读写	出错	正常打开	以读/写方式打开一个文本文件
w+	读写	建立新文件	文件原有内容丢失	以写/读方式打开一个文本文件
a+	读写	建立新文件	在文件原有内容后追加	以读/写方式打开一个文本文件
rb+	读写	出错	正常打开	以读/写方式打开一个二进制文件
wb+	读写	建立新文件	文件原有内容丢失	以写/读方式打开一个二进制文件
ab+	读写	建立新文件	在文件原有内容后追加	以读/写方式打开一个二进制文件

    - `b` 表示针对二进制文件，没有 `b` 表示针对文本文件
    - `r+` 打开文件不会清空， `w+` 会清空，两者均可完成读写操作
- 关闭文件

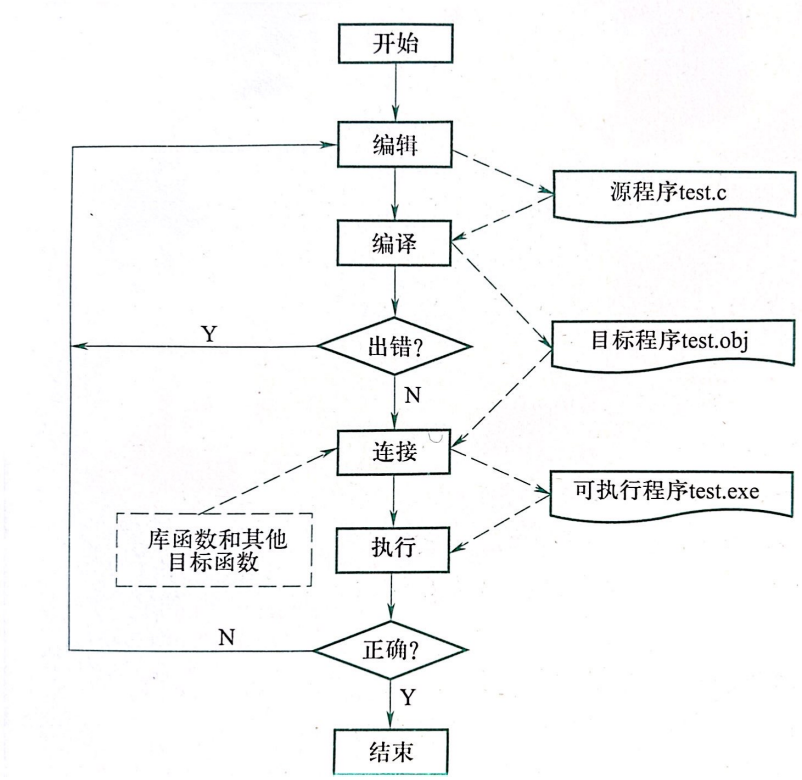
- 格式：`int fclose(FILE *fp);`
- 作用
  - 缓冲区写入磁盘，确保文件操作正常完成
  - 解除`FILE`指针与磁盘文件的关系

    即`flose()`后的`fp`可以用来指向别的文件了
  - 释放文件缓冲区
- 返回值
  - **0：正常关闭文件**
  - 非0：无法正常关闭文件
- 常检测返回值判断是否异常关闭
- 打开文件后记得一定要关上（`fclose()`）

- 读写文件函数
  - **注意参数顺序！！！**
  - 字符读写
    - `int fgetc(FILE *fp);`
      - 读取一个字符，成功返回ASCII码，否则返回`EOF`
    - `int fputc(int ch, FILE *fp);`
      - 写入一个字符，成功返回ASCII码，否则返回`EOF`
  - 字符串读写
    - `char * fgets(char *s, int nsize, FILE *fp);`
      - 读取一个以换行符为结束标志的字符串，成功返回字符串首地址，否则返回`NULL`
      - 会读入换行符，不读入`EOF`，末尾自动添加`'\0'`
      - 所以**读入字符串最大长度为**`nsize - 1`
    - `int fputs(char *s, FILE *fp);`
      - 写入一个字符串，成功返回最后1个字符，否则返回`EOF`
      - 注意，`'\0'`不写入文件！也不会像`puts()`那样自动添加换行符
  - 格式化读写
    - `int fscanf(FILE *fp, char *format, ......);`
      - 操作与`scanf()`类似，实际上`scanf(...) == fscanf(stdin, ...)`
      - 返回实际读取的个数，出错或到结尾返回`EOF`
    - `int fprintf(FILE *fp, char *format,......);`
      - 操作与`printf()`类似，实际上`printf(...) == fprintf(stdout, ...)`
      - 返回写入的字节数

- 数据块读写（多用于二进制文件）
  - `unsigned fread(void *buffer, size_t size, size_t count, FILE *fp);`
    - `buffer` 是一个指向待读入**数据块首地址**的指针，`size` 是每个数据块的**大小**（Byte），`count` 表示最多允许读取的数据块**个数**
    - 从 `fp` 所指文件中读取数据块并存储到 `buffer` 所指内存中，函数返回实际读取到的数据块个数
    - 例：`fread(fa,4,5,fp);` 从 `fp` 所指文件中每次读入4个字节（一个 `int` 整数）送入数组 `fa`（`int fa[]`），读取5次，即 `fa` 现在有5个元素
  - `unsigned fwrite(const void * buffer, size_t size, size_t count, FILE * fp);`
    - 将 `buffer` 所指内存中的数据块写入 `fp` 所指文件
- 文件随机访问函数
  - 重定位文件首函数 `rewind(FILE * fp);`
    - 使指针指向打开文件时文件读写位置指针所指向的位置
    - 不一定是文件的开头，比如使用 `"a"` 模式打开
  - 指针移动控制函数 `fseek(FILE * fp, long offset, unsigned int from);`
    - `offser` 为偏移量（单位为字节），使用常量时应添加 `L` 后缀。值为正时从当前位置向后计算，为负则向前计算
    - `from` 表示起始位置，共3种
      - 0：文件首部，可用 `SEEK_SET`
      - 1：当前位置，可用 `SEEK_CUR`
      - 2：文件尾部，可用 `SEEK_END`
  - 获取指针当前位置函数 `long ftell(FILE * fp);`
    - 成功运行返回相对于文件开头（`fp` 第一次指向文件时的位置）的偏移量（单位还是字节），否则返回 `-1L`（可能是指针未定义）
  - 读写错误检查函数 `ferror(FILE * fp);`
    - 出现错误返回1，没错返回0（相当于 `NULL`）
  - 出错标记清除函数 `clearer(FILE * fp);`
- 文件检测函数
  - `int feof(FILE *fp);`
  - 作用：判断 `fp` 是否到达末尾，是的话返回1，否则返回0
  - 细节：当文件指针指向文件末尾时，并没有立即设置 `EOF`，只有再调用1次读文件操作，才会设置，此后调用 `feof()` 才会返回1，所以慎用 `feof()`
- 顺序读取与随机读取
- EOF

    - 值为-1
    - NULL值为0
- 杂项
    - 编译



    - 类型定义语句 `typedef`
        - 格式：`typedef 类型 新类型名;`
        - 新类型名一般用大写表示
        - 例子：
            - `typedef int INTEGER; //更麻烦了.....`
            - `typedef int NUM[10]; NUM a; //此时的NUM表示一个包含10个整型元素的数组`
            - `NUM b[20] == int b[20][10]`
        - 本质：别名定义就是在定义前面加上typedef，然后把原来变量名的位置换成需要的别名
        - 优点：对于复杂的类型名可以化简
    - 随机数
        - 调用 `#include <stdlib.h> (换行) #include <time.h>`
        - 先用 `srand(time(0));` 生成随机种子，否则 `rand()` 只生成伪随机数
        - `rand() % n` 将产生 0 ~ n - 1 的随机整数
    - 命令行（应该不考吧，但还是得看看.....）
    - 一些坑

> - 缩进：本来是为了美观，但命题人有可能故意利用缩进，形成错觉来误导我们。所以要注意花括号的匹配，分号等问题