C Learning

程序概览和基本概念

C语言标准：ANSI C -> C99 -> C11

示例程序

```c
#include <stdio.h>
int main()
{
   int num;
   num = 6;
   printf("HelloWorld!\n");
   printf("My faveourite number is %d", num);
   return 0;
}
```

基本概念

1. `#include <stdio.h>`：#include 称为C预处理器指令，导入头文件<stdio.h>
2. `int main()`：main函数是一个程序的入口，必须要有，但实际上在调用main函数还有系统底层的函数
3. `// /* */`：单行注释和跨行注释
4. `int num num = 6;`
    1. 变量/标识符（Variable/Identifier）的声明和初始化，标识符在使用之前必须要进行声明和初始化，C99之前变量的声明一定要放在块顶部，C99标准允许其放在任意位置
    2. 命名变量时必须要用字母或下划线开始，不能用数字和其他字符，通常_或__开头的变量是留给操作系统和C库函数的
    3. int是一个数据类型，是一个关键词（keyword），不能被用作变量名
5. `\n`为转义序列 excape sequence
6. `%d`为占位符
7. `return 0;`返回值，函数需要一个返回值，即使不写这条语句函数也会默认返回0，但写这条语句可以养成一种统一的代码风格
8. 关键字和保留标识符

数据和C

数据类型关键字

1. 整数和浮点数

2. 整数
3. 浮点数

C语言基本数据类型

1. int类型

2. 其他整数类型

3. 使用字符：char类型

    1. char类型用于存储字符，但由于在计算机中使用ASCII码来存储字符，因此char实际上是整型。每个char占用一个bit位
    2. ASCII码
        1. 65D-90D 大写字母；97D-122D 小写字母
        2. 0D NULL；32D 空格；48D-57D 1-9
    3. 用单引号括起来的单个字符被称为字符常量 character constant `char grade ='A'`
    4. C语言语法中没有规定char是有符号还是无符号，但为了保持统一，大部分编译器（包括vscode）中char都是**有符号**的
    5. 转义序列 escape sequence

转义序列	含义
\a	警报(ANSIC)
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\\	反斜杠
\'	单引号
\"	单引号
?	问号
\0oo	八进制
\xhh	十六进制

4. _Bool类型

5. 可移植类型：stdint.h和inttypes.h

6. float、double和long double

7. 负数和虚数类型

8. 其他类型

9. 类型大小：sizeof是C语言的内置运算符，以字节位单位给出指定类型的大小，C99和C11提供%zd转换说明匹配sizeof的返回类型。注意，sizeof返回的是size_t无符号整型

字符串和格式化输入/输出

字符串 Character string 简介

1. char类型数组和null字符
2. 使用字符串
3. 计算数组大小和元素个数
    1. `sizeof(str)/sizeof(str[0])` sizeof将会包括字符串末尾'\0'之前的所有char字节：即`char str[100]="I love Jiaxing."; sizeof(str);`其结果为100
    2. strlen()从第一个字符开始计算字符串中字符数，直到遇到空字符，然后返回空字符前字符总个数：即`char str[100]="I love Jiaxing。"; strlen(str);`其结果为15（包含空格和标点符号）

常量和C预处理器

1. C预处理器：`#define NAME value` 变量的值在程序中可能会被改变，因此用预处理器
2. const限定符
    1. int const *p：不能修改*p所指向的值，但可以通过修改p地址
    2. int* const p：不能修改指针p，即p指向的地址不可修改，但*p指向的地址所村粗的指可以修改
3. 明示常量

格式化输入输出I/O函数：printf()和scanf()

1. printf()函数

    1. 转换说明 Conversion specification：格式字符串中的转换说明一定要与后面的每个项相匹配，若忘记这个基本要求会导致严重的后果；若没有匹配项，则根据系统和编译器不同结果也不同

        - 转换说明表格

转换说明	**输出**
%a和%A	浮点数、十六进制数和p计数法
%c	单个字符
%d	有符号十进制整数
%i	有符号十进制整数，和%d相同
%f	浮点数
%e和%E	浮点数，e计数法
%g和%G	根据值的不同自动选择
%o	无符号八进制整数
%p	指针

转换说明	输出
%s	字符串
%u	无符号十进制整数
%x和%X	无符号十六进制整数，使用十六进制数0f和0F
%%	打印一个百分号

- printf()的转换说明修饰符

修饰符	含义
数字	最小字段宽度：即打印几个字符
.数字	精度
h	和整形转换说明一起使用，表示short int或unsigned short int类型的值
hh	和整型转换说明一起使用，表示signed char或unsigned char类型的值
j	和整型转换说明一起使用，表示intmax_t或untmax_t类型的值，这些类型定义在stdint.h中
l	和整形转换说明一起使用，表示long int或unsigned long int类型的值
ll	和整形转换说明一起使用，表示long long int或unsigned long long int类型的值（C99）
L	和浮点转换一起使用，表示long double类型的值
t	和整形转换一起使用，表示ptrdiff_t类型的值
z	和整形转换一起使用，表示size_t类型的值（C99）

- printf()中的标记

标记	含义
-	待打印项左对齐（默认为右对齐）
+	有符号值为正，则在值前面显示加号；为负则显示负号
空格	有符号值为正，则在值前面显示前导空格（不显示任何符号），若为负，则在值前面显示减号
#	把结果转换为另一种形式，若是%o，则以0开始；若是%x或%X，则以0x或0X开始
0	用前导0代替空格填充字段宽度

2. 转换说明的意义：把给定的值翻译成要求的值并打印出来。程序把传入的值放入栈的内存区域，计算机根据变量类型（不是根据转换说明）把这些值放入栈中。printf()函数根绝转换说明（不是根据变量类型）从栈中读取值

    - 使用不匹配的转换说明将造成严重的后果，例如%lf用成了%f

- **printf()和scanf()中对于浮点数的处理是不同的：printf()中没有float的转换类型，所有的float都会被强制转换为double，因此统一用%f，%Lf用于long double。而在scanf()中%f为float型，%lf为double型**

3. printf()**返回打印字符的个数**，如果有输出错误，则返回一个负值

4. 打印长字符串和数字不需要转换说明

2. scanf()函数
    1. scanf()函数使用空白，即**换行符、制表符和空格**把输入分成多个字段
    2. scanf()函数每次读取一个字符，***除了%c模式外跳过开头所有的空白字符开始读取第一个非空白字符***，并保存非空白字符直到再次遇到空白。当scanf()把字符串放进指定数组中时，他会在字符序列的末尾加上'\0'，让数组中的内容称为一个C字符串
    3. 程序在下一次读取输入时，先读到的是上一次读取丢弃放入缓冲区的字符
    4. 格式字符串中的普通字符：除空格字符外的普通字符必须与输入字符串严格匹配。如`scanf("%d,%d", &a, &b)`意味着必须使用逗号
    5. scanf()的返回值：返回成功读取的项数，如果没有读到数输入错误返回0，如果退出则返回EOF

getchar()和putchar()

1. 这两个函数只用于处理字符，所以它们比更通用的scanf()和printf()函数更快、更简洁，同时它们也不需要转换说明。
2. ctype.h系列的字符函数

运算符、表达式和语句

操作符

1. 算术操作符：+-*/%
2. 移位操作符：<< >>
    1. 逻辑移位
    2. 算数移位
3. 位操作符：&|^
4. 赋值操作符：= -= += *= /=
5. 单目操作符
6. 关系操作符：> == != <
7. 逻辑操作符：&& ||
8. 条件操作符：exp1 ? exp2 : exp3
9. 逗号表达式
10. 下标引用、函数调用和结构成员

优先级

1. ();->;++ --;* &;sizeof;加减乘除取余;<< >>;> < == != && ||;= += -= *= /=;
2. 只有对所有编译器唯一确定运算顺序的运算才是正确的，在一个语句里用太多的自加自减会引起混乱。仅仅有操作符的优先级和结合性是无法唯一确定某些表达式的计算路径的

类型转换

1. 自动类型转换

    1. 算术转换：类型转换从高到低顺序：long double, double, float, unsignedlong long, long long, unsigned long, long, unsigned int, int。算术转换时内存中二进制表示的值没有变，知识计算机解释值的方式变了
    2. 升级和降级 Promotion and demotion `double x=2; double y=x+3/2`这里输出为3.000000
    3. 整型提升：当传参和计算时，char和short被转换为int， float转换为double：整型提升是按照变量的数据类型的符号位来提升的
        1. 有符号数
            1. 负数的整型提升：高位补充1
            2. 正数的整型提升：高位补充0
        2. 无符号数：高位补充0
2. 强制类型转换 (int)a

循环

1. while循环是最早出现的
2. for循环是后期产生的，将控制变量的初始化、条件判断、循环条件的调整集成到一起
3. do while循环至少进行一次，使用场景有限

分支和跳转

if语句：0表示假，其余为真（包括负数）

1. 多重选择 else if
2. else与if的配对：else与离它最近的if配对，除非最近的if被花括号括起来了
3. 多层嵌套的if语句

循环辅助

1. continue：执行到该语句时，会跳过本次迭代的剩余部分，并开始下一轮迭代。若continue语句在嵌套循环内，则只会影响包含该语句的内层循环
2. break：终止（跳出）当前循环，并继续执行下一阶段的代码。若是在嵌套循环内，break只会跳出当前层循环，一个break只能跳一层

多重选择：switch和break

1. 程序根据整形表达值跳转至相应得case标签处，然后执行剩余的**所有语句**，除非遇到break语句跳出。表达式和case标签都必须是**整数值**（包括char），标签必须是常量或完全由常量组陈给的表达式。如果没有case标签与表达式匹配，控制则转至标有default的语句（如果有的话）；否则将转至执行紧跟在switch语句后面的语句

2. switch语句中的break：如果不使用break将无法实现分支，因为如果不用break将会顺序执行

3. default：如果所有的case标签都不匹配，那么将会跳过这个switch结构，因此最好写上一个default语句。

```
#include <stdio.h>
int main()
{
    int day = 0;
```

```
    switch(day)
    {
       default:
          printf("Wrong Input!\n");
          break;
       case 1:
       case 2:
       case 3:
       case 4:
       case 5:
          printf("Workday.\n");
          break;
       case 6:
       case 7:
          printf("Weekend.\n");
          break;
    }
    return 0;
 }
```

4. 如果表达式是浮点型或者范围，那么就不能使用switch，用if else结构方便

goto语句：尽量避免使用goto语句，会扰乱程序的执行，在跳出深层循环嵌套时可以使用

字符输入/输出和输入验证

单字符I/O：getchar()和putchar()

getchar()和putchar()每次只处理一个字符。这很方便计算机处理字符数据，且这是绝大多数文本处理程序所用的核心方法。

```
//连续输入
char ch;
while ((ch = getchar()) != EOF)
   putchar(ch);

//或者下面这种写法
while (~(ch = getchar()))
   putchar(ch);
```

缓冲区 Buffer

1. 为什么要有缓冲区？把若干字符作为一个块进行传输比逐个发送这些字符节约时间。如果打错字符也可以修改，这样按下回车时，传输的就是确认过的输入。但也不是所有的输入都要求有缓冲IO，比如游戏等交互程序就要求快速响应的无缓冲IO。C语言标准规定了把缓冲IO作为标准的原因是一些计算器不允许无缓冲输入。
2. 分类
    1. 完全缓冲IO：当缓冲区被填满时才刷新缓冲区
    2. 行缓冲IO：当出现换行符时刷新缓冲输入

结束键盘输入

1. 文件、流和键盘输入
    1. C语言有很多直接操作文件系统的库函数。从底层来看，C可以直接创建并调用操作系统的函数称为底层IO（low-level IO），但因为计算机系统的多样性，所以不可能为这些底层IO函数创建标准库。因此在较高层面上C可以通过标准IO包来处理文件，这涉及创建用于处理文件的标准模型和一套标准IO函数，在这一层面上，具体的C只需要四线负责处理不同系统的差异，以便用户使用统一的界面
    2. C程序处理的是流而不是直接处理文件。**流 Stream是一个实际输入或输出映射的理想化数据流。这意味着不同属性和不同种类的输入，由属性更统一的流来表示**打开文件的过程就是把流与文件相关联，而且读写都通过流来完成。因此可以用处理文件的形式来处理键盘输入。程序读文件时要能检测文件的末尾才知道应在何处停止。因此，C的输入函数内置了文件结尾检测器
2. 文件结尾
    1. 大部分操作系统可以使用内嵌的Ctrl+Z字符来标记文件结尾，getchar()读取文件检测到结尾时返回一个特殊的值，即EOF（End of File），实际上EOF被预处理为-1

重定向和文件

创建更友好的用户界面

1. 使用缓冲输入：使用 while((ch=getchar())!='\n')吸收换行符
2. 混合数值和字符输入

```
#include <stdio.h>
void display(char cr, int lines, int width)
{

}
```

输入验证

1. 分析程序
2. 输入流和数字

菜单浏览

1. 任务
2. 使执行更顺利
3. 混合字符和数值输入

函数

数组和指针

一维数组

1. 初始化：`int char[] = {0};`

1. 当初始化列表中的值少于数组元素个数时，编译器会把剩余的元素都初始化化0，即如果部分初始化数组，剩余的元素就会被初始化为0；而如果不初始化数组，数组元素和未初始化的普通变量一样，其中存储的都是垃圾值。**[]中的数字如果省略，那么编译器自动匹配数组大小和初始化列表中的项数。**
2. C不允许把数组作为一个单元赋给另一个数组，除初始化以外也不允许使用花括号列表的形式赋值

2. 指定初始化器（C99）designated initializer

```
int arr[6] = {[5] = 212};
```

3. 数组边界：使用数组时不允许下标越界，越界结果根据不同编译器表现不同

4. 数组元素编号从0开始

多维数组（以二维数组为例）

1. `test[3][2]`的意思是一个三行二列的二维数组，`test[0]`的意思是第一行首元素的地址
2. 初始化：`int test[3][2] = {{1,2,3,4,5}, {6,7,8,9,10}}`行的个数可以省略，但列的个数一定不能省，因为内存中是将二位数字从第一行到最后一行以大端或者小端逐次排放的，如果缺少列的个数将无法确定排放方式

指针和数组

1. **数组名是数组首元素的地址**，即`arr == &arr[0];`
2. 指针运算：指针+1后的地址是下一个元素的地址，而不是下一个字节的地址，即`arr+1 = &arr[1]`。这也是为什么必须声明指针类型的原因之一，内存中存储不同类型的数据的方式不同，每一个单位的数据也就有不一样的地址，因此也就有其所对应的指针类型。只知道开头地址是不够的，必须要知道指针的类型

函数、数组和指针

1. 使用指针形参
2. 指针表示法和数组表示法：两种方法本质上都是指针操作，arr[i] == *arr+i。数组表示法是为了让使用者更好理解

保护数组中的数据

1. 对形式参数使用const
2. const的其他内容
3. 有时需要把数组设置为只读，这样程序只能从数组中检索值而不能修改数组。提高程序的鲁棒性。

指针和多维数组

变长数组 Variable-Length Array（C99，C11中改为可选特性）

复合字面量 Compound literal

字符串和字符串函数

表示字符串和字符串I/O

1. 在程序中定义字符串

    1. 字符串字面量：字符串常量属于静态存储类别 static storage class，这说明如果在函数中使用字符串常量 ，该字符串只会被储存一次，在整个程序的生命期内存在，即使函数被调用多次

    2. 字符串数组和初始化

    3. 数组和指针区别

        - 数组形式：字符串在程序运行时被载入到静态存储区中，程序在没有运行到相关代码时不会在栈区创建数组，数组形式意味着在栈区开辟了一个字符串常量的临时拷贝。可以进行arr+i操作，但不能进行++arr的操作。同时如果是定义在函数中，该数组是一个自动变量，该拷贝在函数结束时栈区被销毁，可能会造成野指针问题
        - 指针形式：创建了一个指针变量，其指向堆区中字符串常数的首地址，可以进行++arr的操作

```
char* GetMemory(void)
{
    char p[] = "hello world";
    return p;
}

int main()
{
    char* str = NULL;
    str = GetMemory();
    printf(str);
    return 0;
}
```

运行后不会打印"hello world"，只会打印乱码。虽然"hello world"字符串定义在了GetMemory()中，但字符串常量是静态变量保存于内存的静态区中，在GetMemory()退出后并不会被销毁，其定义在GetMemory()中只意味着其位于GetMemory()的定义域中。但问题在于虽然"hello world"字符串常量不会被销毁，但char p[]意味着开辟了新的内存空间给p数组，而其是"hello world"的一份临时拷贝，在GetMemory()退出时被销毁。因此返回的p指向了一个被系统回收的区域，即野指针问题。将数组写成指针形式可以规避这个问题。

```
char* GetMemory(void)
{
    char* p = "hello world"; // *p指向的空间是静态区不会被销毁
    return p;
}

int main()
{
    char* str = NULL;
    str = GetMemory();
```

```
    printf(str);
    return 0;
}
```

4. 字符串数组

2. 指针和字符串

字符串输入

1. 分配空间：必须为输入的字符串提供足够的空间。和直接存储字符串不同，存储输入的字符串时编译器不会自动计算字符串的长度
2. gets()函数 `char* gets(char* buffer);`
    1. 读取整行输入，直至遇到换行符，然后丢弃换行符，存储其余字符，并在这些字符的末尾添加一个\0使其称为一个C字符串
    2. gets()函数只有一个参数，是无法判断输入的内容是否放的进数组中的，容易产生缓冲区溢出，利用其甚至可以造成系统安全上的漏洞，因此不建议使用
3. gets()的替代品
    1. 单字符IO：getchar()
    2. fgets() `char* fgets(char* string, int n, FILE* stream );`
        - fgets()被设计用于处理文件输入
        - fgets()第二个参数用于控制读入字符的最大数量
        - 与gets()丢弃换行符不同，fgets()会保留换行符，因此要与fputs()配对使用
    3. gets_s()
    4. sget_s()
4. scanf()函数：scanf()使用%s来读取字符串，但它读到空白字符就停止了，因此scanf()更像是用来读取一个单词，而非一整句话。scanf()的典型用法就是读取并转换混合数据类型为某种标准形式

字符串输出

1. puts()
2. fputs()
3. prinf()

自定义输入/输出函数

字符串函数

1. 求字符串长度：strlen()函数

```
size_t my_strlen(const char* str) //因为这个函数不会修改str, const增加鲁棒性
{
   assert(str != NULL) //防范野指针
   int count;
   while (*str++ != '\0')
      {
         count++;
      }
```

```
    return count;
}
```

- 字符串已经以'\0'作为结束标志，strlen函数返回的是在字符串中'\0'前面出现的字符个数（不包含'\0'）
- 注意函数的返回值为size_t，是无符号的，要用%zu格式打印

```
    if (strlen("abc")) - strlen("qwerty") > 0)
        printf(">\n);
    else
        printf("<=\n");
```

结果将会打印>，因为strlen返回无符号整型size_t，结果虽然是负数，但会被强制转换为大于0的无符号整型。如果要使用的话可以进行强制类型转换`if (int)(strlen("abc")) - (int)strlen("qwerty") > 0)`

2. 长度不受限制的字符串函数

    1. strcpy()函数

```
char* my_strcpy(char* dest, const char* src)
{
    assert(dest && src)
    char* ret = dest //保存起始位置
    while(*dest++ = *src++)
    {
        ;
    }
    return ret
}
```

- src必须以'\0'结束，strcpy()中的'\0'拷贝到dest
- src只用于读取不修改，因此设为const src以增强其鲁棒性
- dest必须足够大，以确保能存放字符串

    2. strcat()函数 将src追加到dest后面

```
char* my_strcat(char* dest, const char* src)
{
    assert(dest && src);
    char* ret = dest;
    //1.用strlen找到dest中开始拷贝的位置，即第一个'\0'的位置
    while (*dest)
    {
        dest++; //如果判断条件里写*dest++，则当dest已经为'\0'时，dest还会再++一次
        //此时dest就会将'\0'包括进去，那么即使之后的代码有效，在打印时由于打印
```

```
到'\0'就不再打印了，所以打印无效
    }
    //2.用strcpy将src从头开始将dest拷贝到目标位置，src的'\0'被覆盖
    while (*dest++ = *src++)
    {
        ;
    }
    return ret;
}

//第一步的另一种写法
while (*dest++)
{
        ;
}
dest--;

int main()
{
    char arr1[20] = "hello ";
    char arr2[] = "bit";
    my_strcat(arr1, arr2);
    printf("%s\n", arr1);
    return 0;
}
```

- dest从'\0'开始被src首字符开始覆盖，src的'\0'也被一同拷贝
- 设计思路：先用strlen找到dest'\0'的位置（即开始拷贝的位置），然后用strcpy将src拷贝到dest之前找到的位置
- 用my_strcat函数，字符串自己给自己追加的时候会造成死循环，某些C语言库中的strcat函数解决了这个问题

3. strcmp()函数

```
int my_strcmp(const char* s1, const char* s2)
{
    assert(s1 && s2);
    while (*s1 == *s2)
    {
        if (*s1 == '\0')
        {
            return 0; //相等
        }
        s1++;
        s2++;
    }
    //if (*s1 > *s2) //不相等
    //return 1;
    //else
    //return -1;
    return *s1 - *s2;
```

```
}

int main()
{
    char str1[] = "abcq";
    char str2[] = "abc";

    int ret = my_strcmp(str1, str2);

    if (ret > 0)
    {
        printf(">\n");
    }
    else if (ret == 0)
    {
        printf("=\n");
    }
    else
    {
        printf("<\n");
    }
    return 0;
}
```

- "abc" < "abcdef"或者arr1 < arr2这么写是在比较首元素地址的大小
- C语言标准规定，若str1>str2，则返回大于0的数字，<则返回小于0的数字，=返回0
- strcmp函数比较的不是字符串的长度，而是比较字符串中对应位置上的字符的ASCII码大小，如果相同，就比较下一对，直到不同或者都遇到'\0'

3. 长度受限制的字符串函数

    1. strncpy

```
char *strncpy( char *strDest, const char *strSource, size_t count );
```

- 限制了操作的字符个数
- str2长度小于count时不够的时候其余位置会拿'\0'填充
- src也会将自己末尾的'\0'一同拷贝到dest

    2. strncat

- src也会将自己末尾的'\0'一同拷贝到dest
- 可以自己给自己追加了，不会像strcat一样造成死循环

    3. strncmp

4. 字符串查找

    1. strstr：找子串，返回str2在str1中第一次出现的位置，若没有找到则返回NULL

```
char* my_strstr(const char* str1, const char* str2)
{
   assert(str1 && str2);
   //滚动匹配
   const char* s1 = str1; //加上const和str1保持一致，否则权限被放大了(指向同一个地址，一个可以被修改，一个不能被修改)
   const char* s2 = str2;

   const char* cur = str1; //记录开始匹配的位置
   while (*cur)
   {
      //匹配失败重置
      s1 = cur;  //匹配失败s1重置到cur当前位置
      s2 = str2; //匹配失败s2重置到str2开始位置

      while (*s1 && *s2 && (*s1 == *s2)) //前提条件是*s1, *s2不为零且两者相等
      {
         s1++;
         s2++;
      }
      if (*s2 == '\0') //s2被找完了，也就是说s2匹配成功
      {
         return (char*)cur; //返回这一次匹配开始的起始位置，强转为返回类型char*
      }
      cur++; //匹配失败cur前进一位
   }
   return NULL; //找不到
}

int main()
{
   char str1[] = "abcdefjlkjjl\0XXXXX";
   char str2[] = "cdef";
   char* ret = my_strstr(str1, str2);
   if (NULL == ret)
   {
      printf("Cannot find the string.\n");
   }
   else
   {
      printf("%s\n", ret);
   }
   return 0;
}
```

- 还可以用KMP算法实现

2. strtok:查找自定义分隔符（token）

```
char *strtok( char *strToken, const char *sep);
```

- sep参数是个字符串，定义了用作分隔符的字符集合
- strToken为一个字符串，里面包含了0个或者多个被sep分割的字符串段
- strtok函数的第一个参数
- strtok()找到str中的下一个标记，并将其用'\0'结尾，返回一个指向这个标记的指针。strtok()会改变被操作的字符串，所以在使用strtok函数切分的字符串一般都是临时拷贝的内容并且可修改
- strtok函数的第一个参数不为NULL时，函数将找到str中第一标记，strtok函数将保存它在字符串中的位置；strtok函数的第一个参数为NULL时，函数将在同一个字符串中被保存的位置开始，查找下一个标记

```
printf("%s\n", strtok(arr, sep)); //只找第一个标记
printf("%s\n", strtok(NULL, sep)); //是从上一次保存好的位置开始继续往后找
printf("%s\n", strtok(NULL, sep)); //函数内部有一个静态指针变量保存字符串位置，
//函数调用结束后不会被销毁，可以下一次调用时被用到
printf("%s\n", strtok(NULL, sep));
```

- 不区分分隔符的出现顺序，相同的分隔符只要写一个
- 实际使用不可能手动写n次`printf("%s\n", strtok(NULL, sep))`，要写成循环的形式，具体使用方式如下代码所示

```
int main()
{
    char arr[] = "wj.feng@tum.de";
    char buf[30] = { 0 };

    strcpy(buf, arr); //strok会修改原数据，因此使用buf拷贝
    const char* sep = "@."; //不区分分隔符的出现顺序，相同的分隔符只要写一个
    char* str = NULL;
    for (str = strtok(buf, sep); str != NULL; str = strtok(NULL, sep))
    {
        printf("%s\n", str);
    }

    return 0;
}
```

5. 错误信息报告： strerror

```
// strerror 头文件：#include <errno.h>
// 全局变量：errno（错误码）比如说404就是一种错误码
int main()
{
```

```
    printf("%s\n", strerror(0));
    printf("%s\n", strerror(1));
    printf("%s\n", strerror(2));
    printf("%s\n", strerror(3));

    int* p = (int*)malloc(INT_MAX);
    if (p == NULL)
    {
        printf("%s\n", strerror(errno)); //库函数malloc出错时会把错误码放到errno
里
        //errno是全局变量，会被更新的
        perror("malloc"); //与strerror（不打印）使用场景不同
        //perror是直接打印错误码对应的字符串，可以加上自定信息（如"malloc"）
        return 1;
    }
    return 0;
}
```

6. 内存操作函数：str类函数只能用于字符型类型，其他数据如int类数组就不能用

    1. memcpy

```
//void * memcpy ( void * destination, const void * source, size_t num
);
void* my_memcpy(void* dest, const void* src, size_t count)
//void* 可以用来接收任意类型的指针，但时候时必须要进行强制转换
{
    assert(dest && src);
    void* ret = dest;
    while (count--)
    {
        *(char*)dest = *(char*)src;
        dest = (char*)dest + 1;
        src = (char*)src + 1;
        //或者这么写，但这可能有编译器计算路径不确定的问题，还是不要用了
        //((char*)dest)++; //++优先级高于强制类型转换
        //((char*)src)++;
    }

    return ret;
}
```

        - 函数从src位置开始往后复制count个字节的数据到dest
        - 这个函数在遇到'\0'的时候不会停下来
        - 不能用于src和dest有重叠的情况，复制情况未定义，要用memmove

    2. memmove

```c
//第一种写法，前->后/后->前/后->前
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include <stdlib.h>
#include <assert.h>
//void * memmove ( void * destination, const void * source, size_t num
);
void* my_memmove(void* dest, const void* src, size_t count)
{
   assert(dest && src);
   void* ret = dest;
   //1
   if (dest < src)
   {//前->后
      while (count--)
      {
         *(char*)dest = *(char*)src;
         dest = (char*)dest + 1;
         src = (char*)src + 1;

         //++(char*)dest; //在某些编译器下可能会有问题
         //++(char*)src;
      }
   }
   else
   {//后->前
      while (count--)
      {
         *((char*)dest + count) = *((char*)src + count);//以count=20为
例，则第一个之间相差19个字节
      }
   }
   return ret;
}

int main()
{
   int arr1[10] = { 1,2,3,4,5,6,7,8,9,10 };
   int sz = sizeof(arr1) / sizeof(arr1[0]);
   //int arr2[10] = { 0 }; //创建一个临时空间不是一个好的写法，因为很难给定
到底给多少大小，因此还是用自己的空间
   ////my_memcpy(arr2, arr1, 20);
   //my_memcpy(arr1 + 2, arr1, 20); //从3开始复制 期望结果: 1 2 1 2 3 4 5
8 9 10
   ////实际结果：1 2 1 2 1 2 1 8 9 10
   ////memcopy不适合重叠内存拷贝（自己拷贝到自己），要用memmove
//my_memmove(arr1 + 2, arr1, 20);
   my_memmove(arr1, arr1 + 2, 20);

   int i = 0;
   for (i = 0; i < sz; i++)
```

```
    {
        printf("%d ", arr1[i]);
    }

    return 0;
}
```

```
//第二种写法，前->后/后->前/前->后
void* my_memmove(void* dest, const void* src, size_t count)
{
    assert(dest && src);
    void* ret = dest;
    if (dest > src && dest < ((char*)src + count))
    {//后->前
        while (count--)
        {
            *(char*)dest = *(char*)src;
            dest = (char*)dest + 1;
            src = (char*)src + 1;
        }
    }
    else
    {//前->后
        while (count--)
        {
            *((char*)dest + count) = *((char*)src + count);
        }
    }
    return ret;
}
```

- 既然memmove的功能比memcpy的功能强大，为什么还不废除memcpy？因为memcpy早于memmove出现，出于兼容早期版本C语言等目的是不能随便废除memcpy函数的
- 同时相比于my_memcpy，为了方便使用，VS编译器库函数中的memcpy也实现了重叠拷贝

3. memset：内存设置

```
//memset 初始化内存单位
//void* memset(void* dest, int c, size_t count);
int main()
{
    int arr[] = { 1,2,3,4,5 };
    memset(arr, 0xFF, 20); //以字节为单位来初始化内存单元的
    return 0;
}
```

4. memcmp

```c
//memcmp
//int memcmp(const void* ptr1, const void* ptr2, size_t num); //因为要兼容所有数据类型，所以用了void*，因此这里是一个字节一个字节进行比较的
//形参与实参数据类型不一致时，强制转换为形参的数据类型void*
int main()
{
   int arr1[5] = { 1,2,3,4,5 };
   int arr2[5] = { 1,2,3,4,0x11223305 };
   int ret = memcmp(arr1, arr2, 18); //1，逐字节比较
   int ret = memcmp(arr1, arr2, 17); //0
   printf("%d\n", ret);
   return 0;
}
```

字符串示例：字符串排序

1. 排序指针而非字符串
2. 选择排序（冒泡排序）

存储类别、连接和内存管理

存储类别

C语言提供了多种不同的模型或存储类别 storge class 在内存中存储数据。从硬件来看，将数据占用的内存称为对象 object，其用存储期 storage duration来描述；从软件来看，程序需要一种方法来访问对象，即声明变量，其用作用域 scope和链接 linkage来描述标识符。

1. 作用域 Scope

    1. 局部作用域/块作用域 Local scope：块是用一对花括号括起来的代码区域，例如整个函数体，或者在for、while、do while和if语言的代码块中，一但变量离开当前定义的块时，其内存会被释放，局部变量被销毁。外部函数不能使用内部函数定义的变量，而内部函数可以使用外部函数的变量，即内部函数变量的作用域包含了外部函数

    2. 函数作用域 Function scope只用于goto语句中的标签

    3. 函数原型作用域 Function prototype scope用于函数原型中的形参名（变量名）：函数圆形作用域的范围是从形参定义处到原型声明结束，这意味着在除变长数组外的函数声明中，形参的变量名无关紧要（因为该作用域不会影响到块作用域），只有形参的数据类型有用

    4. 全局作用域/文件作用域 Global scope：定义在函数外面，整个c文件中都可使用

    5. 在全局作用域和局部作用域中有相同名字的变量时，**采用就近原则，其有不同的地址**。如以下的代码输出结果为1：test中的a为局部变量，在test函数结束后被销毁，打印的a为全局变量a

```c
#include <stdio.h>

int a = 1;
void test()
{
```

```
        int a = 2;
        a += 1;
}

int main()
{
        test();
        printf("%d\n",a);
        return 0;
}
```

2. 链接

    1. C变量有3种链接属性：外部链接、内部链接或无链接
    2. 具有块作用域、函数作用域或函数原型作用域的变量都是无链接变量，这意味着这些变量仅属于定义他们的块、函数或原型
    3. 具有文件作用域的变量可以是外部链接或内部链接。
        - 一个C文件称为一个翻译单元
        - 内部链接/文件作用域：仅限于一个翻译单元
        - 外部链接/全局作用域或程序作用域：延伸至其他翻译单元

3. 存储期：描述了内存中存储的对象可以被访问的生存期

    1. 自动变量

        - 声明在块或函数头中的任何变量都属于自动存储类别，其具有自动存储期、块作用域和无链接性
        - 没有{}的块（C99）
        - 自动变量不会初始化，除非显式初始化它。若没有初始化则分配一个随机值
        - 如果内层块中声明的变量与外层块中的变量同名时，内层块会隐藏外层块的定义，即**采用就近原则，其有不同的地址**

    2. 寄存器变量：寄存器变量也是自动变量，但其存储在寄存器中，无法获取其地址

    3. 静态变量：静态变量和外部变量在程序载入内存时已执行完毕，在程序结束后不会被销毁，保留在静态区中，如字符串常数

        - 块作用域的静态变量：在块中定义时用static类别说明符
        - 外部链接的静态变量/外部变量
            - 外部链接的静态变量又称为外部变量 external variable，其具有文件作用域、外部链接和静态存储期
            - 把变量的定义性声明 defining declaration 放在所有函数外面便创建了外部变量
            - 外部变量是可以被所有C文件访问的，若要使用的外部变量被定义在其他C文件中，要使用extern类别说明符再次声明
            - 外部变量如果没有初始化会被自动初始化为0
        - 内部链接的静态变量
            - 具有文件作用域、内部连接和静态存储器
            - 需要使用static修饰，只能在同一个文件中被访问，在其他文件中无法使用extern访问

```
int traveler = 5; //定义式声明 defining declaration
static int stayhome = 1;

void test(void)
{
   extern traveler x; //引用式声明 referencing declaration
   //使用定义在其他C文件中的外部变量，并不是该文件中的外部变量
   //...
}
```

4. 多文件：只有当程序由多个翻译单元组成时，才能体现出内部链接和外部链接的重要性
    - C通过在一个文件中进行定义式声明，然后在其他文件中进行引用式声明来实现共享
    - 如果外部变量定义在一个文件中，那么其他文件在使用该变量必须用extern进行引用声明。也就是说在某文件中对外部变量进行定义式声明只是单方面允许其他文件使用该变量，其他文件在用extern接收前不可以使用该外部变量
5. 存储类别和函数：和变量相同有外部函数（默认）和静态函数，除非用static声明，否则默认为extern。C99新增了内联函数
6. 存储类别的选择

存储类别	存储期	作用域	链接	声明方式
自动变量	自动	块	无	块内
寄存器变量	自动	块	无	块内，使用关键字 register
静态外部链接	静态	文件	外部	所有函数外
静态内部链接	静态	文件	内部	所有函数外，使用关键字static
静态无连接	静态	块	无	块内，使用关键字static

随机数和静态变量

掷骰子

动态内存开辟

1. 为什么存在动态内存分配

   我们已经掌握的内存开辟方式有

   ```
   int val = 20; //在栈空间上开辟四个字节
   char arr[10] = {0}; //在栈空间上开辟10个字节的连续空间
   ```

   但是上述的空间开辟的方式有两个特点：

    - 空间开辟大小是固定的

- 数组在申明的时候，必须指定数组的长度，它所需要的内存在编译时分配 但是对于空间的需求不仅仅是上述的情况，很多时候我们到底需要多少空间，是随着程序的运行逐渐变化的，因此静态的空间开辟就不能满足需要了，需要进行动态内存开辟

2. 动态内存分配函数

    1. malloc()和free()

    ```
    #include <stdlib.h>
    void* malloc (size_t size);
    void free (void* ptr);
    ```

    malloc()向内存申请一块连续可用的匿名空间，并返回指向这块空间的指针

    - 若开辟成功，返回开辟成功空间的地址
    - 若开辟失败，则返回一个NULL指针，因此malloc的返回值一定要做检查。否则如果开辟失败而不知道，会产生对野指针解引用等问题
    - 返回值的类型是void*，所以mallloc()并不知道开辟空间的类型，具体在使用的时候使用者自己进行强制转换

    ```
    int* ptr = (int*) malloc(40); //开辟40个字节的int型空间
    int* p = ptr; //free()要从开辟空间的起始地址开始，所以不能直接用ptr
    if (p == NULL)) //检查动态内存空间开辟是否成功
    {
       perror("malloc");
       return 1;
    }

    int i = 0;
    for(i=0; i<10; i++)
    {
       *p = i;
       p++;
    }

    free(ptr); //释放空间
    ptr = NULL; //ptr释放后变成野指针，置空
    ```

    - 若参数size未定，则malloc()的行为是标准未定义的，由编译器决定

    free()用来释放回收动态开辟的内存

    - 若参数ptr指向的空间不是动态开辟的，则free()的行为未定义
    - 若参数ptr是NULL，则函数什么事都不做

    2. calloc

```
void* calloc (size_t num, size_t size);
```

calloc()的功能是为num个大小为size的元素开辟一块空间，并且把空间中的每个字节初始化为0；其与malloc()的区别只在于calloc()会在返回地址之前把申请的空间的每个字节初始化为全0

3. realloc

realloc()让动态内存开辟更加灵活。有时我们发现过去申请的空间太小了，有时我们又觉得太大了，那为了合理的使用内存，我们一定会对内存的大小进行灵活的调整。realloc()就可以做到对动态开辟内存大小的调整

```
void* realloc (void* ptr, size_t size);
```

- ptr为要调整的内存地址
- size为调整之后新的内存空间的大小
- 若开辟成功，则返回新空间的地址，否则返回NULL
- 扩容时有两种情况
    - 原有空间之后有足够大的空间：直接扩展内存，原空间数据存放的数据不变
    - 原有空间之后的空间已经被用掉了，没有足够大的空间用来扩容
        - 在堆空间上找一个新的可用连续匿名空间，返回其地址
        - 拷贝原空间的数据到扩容的地址上
        - free原空间

```
int* p = (int*)malloc(40);

if (p == NULL)
{
   perror("malloc");
   return 1;
}

int* ptr = (int*)realloc(p, 80); //将空间扩容为存储80字节的int数据的空间
if (ptr != NULL)
{
   p = ptr;
}

free(p);
p = NULL;
```

3. 常见的动态内存错误

1. 对NULL指针的解引用操作

```

```

```
void test()
{
   int* p = (int*)malloc(INT_MAX); //需要开辟的内存过大，malloc开辟失败，
返回NULL
   *p = 20;
   free(p);
}
```

解决办法：对malloc函数的返回值进行判断后再使用

2. 对动态内存开辟空间的越界访问

```
void test()
{
   int i = 0;
   int* p = (int*)malloc(10 * sizeof(int));
   if (p == NULL)
   {
      exit(EXIT_FAILURE);
   }

   for (i=0; i<=10; i++) //i<=10 访问11个int，造成越界
   {
      *(p+i) = i;
   }

   free(p);
}
```

3. 对非动态开辟内存使用free()

```
int a = 10;
int* p = &a;
free(p);
```

4. 使用free()释放一块动态内存的一部分

```
int* p = (int*)malloc(40);
p++;
free(p);
```

5. 对同一块动态内存多次释放

```
int* p = (int*)malloc(40);
free(p);
free(p);
```

6. 动态开辟内存忘记释放，造成内存泄漏 Memory leak

```
void test()
{
   int* p = (int*)mallloc(40);
   if (p != NULL)
   {
      *p = 20;
   }
}

int main()
{
   test();
   while(1);
   return 1; //没有free()
}
```

4. 柔性数组/动态数组/伸缩性数组（C99） Flexible array

1. 柔性数组的特点

1. 结构中的柔性数组成员前面必须至少有一个其他成员，且柔性数组一定要放在结构体最后
2. sizeof返回的这种结构大小不包括柔性数组的内存
3. 包含柔性数组成员的结构用malloc()进行内存的动态分配，并且分配的内存应该大于结构的大小，以适应柔性数组的预期大小

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

struct S
{
   int num;
   int arr[];
}

int main()
{
   //柔性数组用malloc()初始化
   struct S* ps = (struct S*)malloc(sizeof(struct S) + 40);
   if (ps == NULL)
   {
      perror("malloc\n");
      return 1;
```

```
    }
    ps->num = 100;

    int i = 0;
    for(i=0; i<10; i++)
    {
       ps->arr[i] = i;
    }

    for(i=0; i<10; i++)
    {
       printf("%d ", ps->arr[i]);
    }

    //realloc()扩容
    struct S* ptr = (struct S*)realloc(ps, (sizeof(struct S) + 80));
    if (ptr == NULL)
    {
       perror("realloc\n");
       return 1;
    }
    else
       ps = ptr;

    //释放
    free(ps);
    ps = NULL;

    return 0;
}
```

2. 柔性数组的使用

3. 柔性数组的优势

文件输入/输出

与文件进行通信

标准I/O

文件I/O

随机访问

标准I/O的机理

其他标准I/O函数

结构和其他数据形式 Struct

结构的声明

1. 结构声明与初始化

```
struct tag
{
   int a; //结构布局
   char b;
   float c;
   int* ptr;
}a, *pa; //结构变量
```

    1. 结构体变量的初始化：`struct tag A = {3, 'a', 3.50, ptr}`
    2. 访问结构成员：使用.运算符 `tag.a; tag.b; tag.c; tag.ptr`
    3. 匿名结构体：不写tag名，只能用一次

2. 用->操作符访问结构体指针：`struct tag * A; A->a = 5;`

3. 结构体内存对齐

    1. 对齐规则
        - 第一个成员在于结构体变量偏移量为0的地址处
        - 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处
            - 对齐数=编译器默认对齐数与结构体成员大小的较小值
            - VS默认为8， Linux中则没有对齐数，结构体成员自身的大小就是对齐数
        - 结构体总大小为最大对齐数（每个成员变量都有一个对齐数）的整数倍
        - 如果嵌套了结构体的情况，嵌套的结构体对齐到自己的最大对齐数的整数倍出，结构体的整体大小就是所有最大对其书（含嵌套结构体的对齐数）的整数倍
    2. 为什么存在内存对齐
        - 平台原因（移植原因）：不是所有的硬件平台都能访问任意地址上的任意数据的，某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常
        - 性能原因：数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因是为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次
    3. 修改默认对齐数：`#pragma pack()`

链式访问

联合体 Union

枚举 Enum

位操作

C按位运算符 bitwise operator

1. 按位逻辑运算符
2. 移位运算符

位字段 bit field

对齐特性（C11）

C预处理器和C库

明示常量 Manifest constant: #define

在#define中使用参数

1. 用宏参数创建字符串：#运算符
2. 预处理器粘合剂：##运算符
3. 变参宏：...和__VA_ARGS__

宏和函数的选择

1. 宏和函数相比较的优点
    1. 宏通常被用来进行较简单的计算
    2. 函数的调用需要进行栈帧空间的开辟和销毁。调用一个简单函数所需要的栈空间的开辟和销毁所需要的内存和时间比其实际用于用于运算的内存和时间大得多
    3. 更为重要的是函数的参数必须声明为特定的类型，所以函数只能在类型合适的表达式上使用，反之宏可以适用于任何的数据类型
2. 宏和函数相比较的缺点
    1. 每次使用宏的时候，一份宏定义的代码都会被插入到程序中。除非宏比较短，否则可能大幅度增加程序的长度
    2. 宏是没法调试的
    3. 宏由于类型无关，也就不够严谨
    4. 宏可能会带来运算优先级的问题，程序容易出错

```
//malloc()开辟动态内存
//1.一般方法，不方便多次开辟
int* p = (int*)malloc(10 * sizeof(int));

//2.预定义器进行替换
#define MALLOC(num, type) (type*)malloc(num * sizeof(type))

int* p2 = MALLOC(10, int);
char* p3 = MALLOC(20, char);
```

文件包括 #include

其他指令

1. #undef指令：#undef用于取消已有的#define指令。有时候为了安全起见，想使用一个名称但是不能确定前面是否已经用过的时候可以用#undef来取消
2. 条件编译
3. 预定义宏
4. #line和#error
5. #pragma
6. 泛型选择（C11）

内联函数（C99） Inline

_Noretun函数（C11）

断言