C语言知识点简记

1.第一章

变量

- 标志符(变量)： 字母数字下划线，不能以数字开头，关键字不能做标识符
- 关键字--》略

赋值和初始化

- `int a`定义一个变量，只定义不赋值，系统乱给一个值
- `int a = 1000`初始化一变量
- `a= 2`为一个变量赋值（称为表达式），`=`为赋值运算符，可多次赋值，`int`是变量的类型,`a`是变量,存放数据
- `scanf("%d%d%d",&a&b&c)`%d%d%d要你输入的东西，不能有其他的东西
- `const int AMOUNT = 100`定义一个常量，`const`修饰符，不能被修改，一定要赋初值。`counst`常量大写
- `print("%d\n",10/3*3)`等于9，整数除`/`会将小数部分扔掉
- 浮点数，带小数点的数(小数点位置不固定)，如果，一个浮点数和一个整数运算，先将浮点数转化成浮点数，再运算，运算结果为浮点数。
- `getchar()`过滤掉回车,从标准输入流中读取下一个字符，也可以当输入字符串使用,一个一个字符匹配

  `putchar("\n")`打印换行符
- `(int)2.0`类型强制转换,去掉小数点
- 本地变量都没有初始值（不会初始化）

c语言的类型

- c语言的类型语言，但检查的强度不够严

变量类型	解释	输入输出
short	短整型	%d
int	整型	%d %u %lu %i %x%X%o
char	字符型	%c / %d
float	单精度浮点型	%f%F,%e%E科学计数法,%g%G, %a%A十六进制浮点
double	双精度浮点型	`scanf`输入用`%lf`, 输出`%lf,%f`都可以
long	长整型	%d
long long	c99	%ld
long double	c99	%lf
bool	_Bool布尔型c99	
char str[] = "xxx"	字符串	%s， `[^,]`
指针	以16的进制的形式输出地址	%p
unsigned long long	无符号的long long	%lu
unsigned int	无符号的整型	%u
enum?	自定义类型?	%n读入写入的个数

- `int`和`long`取决于编译器（cpu）,一个字长，32bit,64bit,反应到c就是int
- `signed默认0正1负`有符号,char -128~127
- `unsigned`无符号，不以高位时补码的形式看待，可以扩大表示范围，但不能在表示负数
- 如果一个字面量想要表达自己是unsigned,可以再后面加上`u/U`，如`255u`.同样的yong`l/L`表示long
- unsigned的初衷是做纯二进制运算，为了移位
- **整数越界**
  - **signed char**
  - `127 + 1 = -128`
  - `-128 - 1 = 127`
  - **unsigned char**
  - `255 + 1 = 0`
  - `0 - 1 = 255`
- `012 0x12`八进制`%o`，和十六进制`%x`
- 浮点数float 和double可以表示inf和NAN,float没有精度，带小数的字面量是double,不是float, `1.213f`表示float类型
- `char<short<int<float<double`大小范围
- `%.3f`保留三位小数，但会进行四舍五入

- 内存中表达形式：二进制（补码）、编码（浮点数）
- `sizeof(int)`运算符，输出类型的所占字节(bit)大小,一个字节8位。sizeof是静态的，不会处理（）的表达式
- 所有类型最终的意义是我们以什么样的方式去看待它，而不是表明他在计算机的内部是如何表达的
- 原码`0000 0001`1,`0000 0000`0??? ---- 补码`1111 1111`-1
- **字符类型**
  - char是一种整数，也是字符类型
  - `''`表示 49-->'1',ASCII编码
  - 逃逸字符(转义字符)`\b  \t  \n  \r  \"  \'  \\`
- 自动类型转换,当类型不同时,自动转换范围大的类型,`short char-->int`,`float-->double`
- 强制类型转换`(类型)值  (int)12.3`,强制类型转换,并不会改变原来的变量
- `bool`头文件`#include<stdbool.h>`

表达式

- 运算符,运算的动作
- 算子，运算的值，可以常数，可以变量，还可以是返回值
- `%`取余运算，必须整数

运算符的优先级

- 各种运算符的优先级
- 复合运算符`+= -+ *= += %=`
- 自增自减运算符`a++ b--`
- 条件运算符，返回值0或1
- 逻辑运算,短路运算
- 条件运算符(三目运算符)`a>b?a:b`
- 逗号运算符,for循环使用

2.第二章

判断

- `if(){};else if(){};else{}`
- `// /**/`注释
- 流程图

分支

- 条件分支
- 多路分支`switch-case`
- `switch(type){case 1:xx; break;case 2:xx; break;default:xx;break;}`type控制表达式只能是整数类型，case前面的值也只能是常数。

循环

- `while(条件语句){循环体};`条件满足时执行
- `do{循环体}while(条件语句);`不管条件满不满足，循环体先执行一次
- `rand()` 产生随机数
- `for(i=0;i<10;i++)` for循环

循环控制语句

- `continue`跳出本轮循环
- `break`跳出循环体
- `go lable ; lable: xxxx;`go语句

循环嵌套

3.第三章

函数

- 函数是一块代码，可以接收参数，并进行逻辑处理，然后返回得到的值。
- 定义函数`void sum(参数表){函数体}`，void不返回值，不能使用带值的return（代表返回值的类型），sum函数名
- `sum(x);`调用函数。，给出正确的参数顺序和数量。
- `return x;`出口，一个函数可以有多个出口，但多使用单一出口

函数的参数和变量

- c编译器自上而下顺序分析你的代码
- 函数的声明`void sum(参数表int);`，函数原型，声明不是一个函数，只有定义才是一个函数，声明和定义类型要一致,`int x`或`int`可以不写变量
- 函数的定义`void sum(参数表int x){函数体}`
- 调用函数时传递的值**类型可以不那么匹配**，编译器会进行类型转换，转换标准函数参数的定义类型。
- 再调用函数时，永远只能转值给函数，不能对实参的值进行操作。
- 局部变量，定义再语句块内，生存期是大括号{}，作用域是大括号{}。
- 局部变量不会被默认初始化，但参数进入函数内会被初始化。
- `void swap(void);`没有参数
- 不允许函数的嵌套定义
- 主函数也可以写成`int main(void)`，是一个程序的入口，也是一个函数

4.第四章

数组

- 类型 数组名[元素个数]
  - `int a[6];`
  - `char b[10];`
  - `double c[3];`
- 数组"不能"动态定义数组大小（c99支持）,数组是一个容器，所以的元素数据类型相同，一旦创建不能改变大小，内存中数组紧密排列。

- 编译器和运行环境不会检查数组下标越界问题。
- 可以创建`int a[0]`的数组，但没有任何用处。
- 数组创建时，不会自动初始化为0
- 访问数组：`a[0]`访问第一个元素
- 数组的初始化`int a[10] = {1,2,4,8}`未赋值的元素默认赋值为零
  - `int a[] = {1,2,4,8}`
  - `a[10] = {1,2,4,8,[9]=10}`
  - `sizeof(a)`计算数组的内存大小,字节
  - `sizeof(a)/sizeof(a[0])`
- 数组变量本身不能被赋值，要把一个数组的所有元素交给另一个数组，必须采用循环遍历。
- `int search(int key;int a[], int length){...}`当数组作为函数参数时，往往必须再用另一个参数来传入数组的大小。`search(key,a,length)`调用

二维数组

```c
#include<stdio.h>
int main(){
    int a[6][6];//定义
    //通过下标访问数组
    int b[2][2] = {{1,2},{3,4}};
    int a[][2] = {{1,2},{3,4},{5,6}}//三行两列，只有第一维的元素可以省略
}
```

- 二维数组，两层循环。
- 列必须给出，行数可以有编译器决定。
- `scanf("%d",&array[i][j])`输入

5.第五章

指针

- 定义指针变量`char *pa; int *pb;`定义一个指向字符型的指针变量,(保存地址的变量),`int *pb; <==> int* pb;` pb是一个指针,它指向一个int类型
- 取地址运算符`&`
- `char *pa = &a; int *pb = &f`正确的初始化，切记勿对指针变量赋值,用&运算符取出普通变量的地址给指针变量
- 取出地址的大小取决于编译器的架构(32bit还是64bit)
- `int a[10] &a == a == &a[0]`
- 指针作为参数时,`void fun(int *p);` 定义,`fun(&i)`调用,fun函数可以通过访问`i`变量
- 取值运算符`*` `printf("%c,%d\n",*pa,*pb)`,`*pa`取出`pa`存放的地址指向的值(目标地址里的内容)
- 函数可以通过指针可以改变变量的值(本质是通过地址操作)
- **左值**,出现在赋值号的左边,不是变量,是一个值(表达式),如`a[0] = 2 *p = 3`,`a[0]`,`*p`都是一个左值,`[]` `*`是一个运算符,左值是一个特殊的值
- 指针存放的都是地址，故指针变量的内存大小都是4个字节

- 指针的运算
  - `char a[10] = "qwerty"; char *p = a; printf("%c",*(p+1))` 指针加一，指向下一个数组(字符串)的元素

指针作用

- 1交换两个值
  - 函数要传回多个值时,用指针,传过去的值产生的结果有指针带回
- 2函数返回运算的状态，结果通过指针返回，通过返回特殊（-1，0）的不属于有效范围的值来表示出错。（原因函数可能出错）

指针和数组

- 函数参数表里的数组参数`void fun(int a[], int n)`，`a[]`其实是一个指针,也可以写成`void fun(int *a, int n)`,数组名（数组变量）本质上是一个特殊指针
  - 下列四种函数原型（声明）是等价的;
  - `int sum(int *ar, int n)`
  - `int sum(int *, int n)`
  - `int sum(int ar[], int n)`

    `int sum(int [], int n)`
- 数组名是存放一个地址，并且存放第一个元素的地址，`a == &a[0]`,`*a == a[0]`
- `int a[10],int *p = a;`无需用&取数组a的地址，因为a存放的就是一个地址，但数组的单元表达的是一个变量，需要用&取地址
- 数组变量是const的指针，所以不能被赋值。`int a[] --> int * const a`
- `int ia[10] = {0}`,`sizeof(ia)`可以输出ia数组的内存大小40，`sizeof(ia[0])`则是数组第一个元素的内存大小4,`sizeof(ia)/sizeof(ia[0])`可以求出ia数组的元素的个数10，一般用于遍历数组`for(i = 0; i <  sizeof(ia)/sizeof(ia[0]); i++)`。
- 另一种遍历数组的方法
  - ```c
    #include<stdio.h>
    int main(){
        char ca[] = {1,2,3,4,5,6,-1,};
        char *p = ca;//char *p = &ca[0];
        while(*p != -1){
            printf("%d\n",*p++);
        }
        return 0;
    }
    ```

指针和const

- **若指针是const**
  - `int * const q = &i`
  - 表示一旦得到了某个变量的地址，不能再指向其他变量
  - `*q = 26`可以对指针指向的变量修改，`q++错误`但对指针的内容不能再修改
- **若int是const**
    ```

    - `const int *p = &i`， `int const *p = &i`
    - 表示不能通过这个指针去修改那个变量（并不能使得那个变量成为const）
    - `i = 26 p = &j` 都可以，但 `*p = 26` 错误,(*p)是const
    - **作用**：要传递的参数类型比地址还大，这是常用的手段：既能用比较少的字节数传递值给参数，又能够避免函数对外面变量的修改
- **const数组**
    - `` `const int a[] = {1,2,3,4,5,6}; ``
    - 数组变量已经是const的指针了，这里的const表明数组的每个单元都是const int
    - 所以必须通过初始化进行赋值
    - 因为把数组传入函数时传递的是个地址，所以那个函数内部可以修改数组的值，为了保护数组不被函数破坏，可以设置参数为const. `int sum(const int a[], int length);`

指针的运算

- 指针操作数组，如`p+1取址 *(p+1)取值`
- 注意：若进行p+1或p-1操作不进行赋值运算（不给另外一个指针），不改变指针的指向（还是指向原来的地址），p++,p--同理
- 当我们用两个指针相加减`p1-p6`，给我们的不是两个指针的地址的差值，而是给我们地址的差值除以sizeof(type)，告诉我们可以放几个值
- `a = *p++` ++的优先级高，故不需要加括号，但a的值是加1以前的结果
- 指针可以的运算`< <= == > >= != ++ --`,地址大小的比较、加减，不能乘除。
- 当一个程序运行的时候，系统都会给从零地址开始，大小4G，一个虚拟内存，但0地址不让随便碰，所以指针不应该具有0值。但可以给指针赋初值0，`int *p = NULL 表示0地址`，来做一些特殊的事情，如返回的指针无效，指针没有被真正的初始化。
- 不论指向什么类型的指针，所以的指针的大小都是相同的，因为都是地址，但指向不同类型的指针不能直接相互赋值。（可以强制类型转换`int *p = &i; void* q = (void*)p;`，并没有改变i变量的类型，只是有一个void类型的指针q指向了变量i这片地址空间）
- `void* q;`表示指向一片地址空间，不知道指向什么类型

动态内存分配

- 输入多个不确定的变量，并记录时，我们用动态定义数组来实现，但在C99标准之前，我们用动态内存分配来实现。(运行时才分配的内存)
- ```
  #include<stdlib.h>
  int *a;
  a = (int*)malloc(n*sizeof(int));
  free(a);
  /**
  *   可以把a当作数组，如a[0]
  *   malloc()函数
  *   n*sizeof(int) 表示要多大的内存
  *   malloc()返回void类型，故需要类型转换
  *   详细见p36
  */
  ```
- `#include<stdlib.h>`
- `free(a)` 使用后要把申请内存还给系统，只能还最初始首地址。 同样的不是申请来的地址不能free().
- `free(NULL)` 是可以的，所以可以让定义指针p时可以让指针p指向NULL,如果指针p没有正确的malloc(),那我们可以正确的free指针p，`free(p)`
  ```

- 如果申请内存失败，返回0或NULL

指针数组和数组指针

- 数组名是一个地址，不可改变，指针是一个左值,可改变其值
- 指针**数组** `int *p1[5]`
- 数组**指针** `int (*p2)[5]`

指针和二维数组

- `*(array+i) = array[i]`

6.第六章

字符串

- 字符数组 `char word[] = {'h','e','l','l','0','!'};`
- 字符串 `char word[] = {'h','e','l','l','0','!','\0'};`
- `char str[] = "xxxxxx"`，自动在后面添加 `\0`
- 字符串时以0（整数0）结尾的一串字符，`0 || '\0'` 是一样的，但是和 `'0'` 不同
- 0标志字符串的结束，但它不是字符串的一部分，计算字符串长度时候不包含这个0
- 字符串以数组的形式存在，以数组或指针的形式访问，更多以指针的形式访问
- `#include<string.h>`
- **字符串变量**的不同表现形式
  - `char *str = "hello";` 定义了一个str指针，它指向了一个在代码段的 `"hello"` 字符数组
    - 上面的 `"hello"` 内存位置，在代码段，只可读，不可写，内存地址比本地变量小的多，并且如果还有一指针指向"hello"，他们指向的是同一个"hello"，地址相同。
    - 实际指针str是 `const char* str = "hello";`,编译器接受没有const的写法
    - 如果需要修改字符串，应该用数组：`char str[] = "hello";` 这时的 `hello` 就代码段的了，而是将代码段的 `hello` 拷贝到 `str[]` 创建的内存位置。
  - `char word[] = "hello";` "hello"字符串的字面量，"hello"会被编译器变成一个字符数组，这个数组的长度是6，结尾还有表示结束的0
  - `char line[10] = "hello";` line有10个空间，5个字符占了6个空间，还有结尾的0
- 如果要构造一个字符串用数组，如果要处理一个字符串用指针。
- `char *` 可以表示一个char类型的指针，也可以表示一个字符串，只有当他所指的字符数组以0结尾时，我们才说 `char *` 是一个字符串
- 两个相邻的字符串常量会被自动连接起来 `"abc" "def"` 等价于 `"abcdef"`

字符串的输入输出

- `char string[8]; scanf("%s",string); printf("%s",string);`，一个 `scanf` 读入一个单词（到空格、tab、回车为止）,无&取址符
- 这样会导致很容易导致输入越界 `scanf("%7s",string);`，`%7s` 表示最多只读7个字符，若有都输入的字符，交给下一个 `scanf`（不在以空格区分单词，而是以这个数字7区分）
- `char str[100] = ";"`，这是一个空字符串，`str[0] = '\0'`
- `char str[] = "";`，这个字符串长度只有1，`str[0] = '\0'`

字符串数组

- `char **a`，a是一个指针，指向另一个指针，那个指针指向一个字符（串）
- `char a[][3] = {"hello","world","nihao!"};`，是一个字符串数组
- `char *a[] = {"hello","world","nihao!"};`，相当于a[0]-->char*，是一个指针数组，数组的每一个元素都是一个指针，分别指向与之对应的字符串，如a[0]指向"hello"这个字符串。要加上`const`，不然会报警告,`const char *a[] = {...}`因为该字符串内存地址在代码段，只可读。

程序参数

- `int main(int argc,char const *argv[])`，argc数组元素的个数，argv[0]是命令本身，当使用Unix的符号连接时，反应符号连接的名字

字符串处理函数

单字的输入输出

函数	用法	解释
putchar	`int put char(int c);`	向标准输入写一个字符，返回int类型，返回写了几个字符，通常返回1，返回`EOF`表示写失败
getchar	`int getchar(void);`	从标准输入读入一个字符，返回类型是int是为了返回`EOF`

- `EOF`是一个宏，不同的宏有不同的值，`EOF`的默认1？，失败是-1.
- `Ctrl + c`，shell杀死程序，`Ctrl + z Winow ， Ctrl + d Linux`，shell给`EOF`赋值-1。
- ```c
  #include<stdio.h>

  int main(void){
      int ch;
      while((ch = getchar()) != EOF){
          putchar(ch);
      }
      printf("EOF");

      return 0;
  }
  ```

字符串函数

- 注意：不同的编译器或着操作系统可能返回值不同，用法不同
- 头文件`#include<string.h>`，引入标准库文件
- `size_t strlen(const char *s)`
  - strlen函数，返回s的字符串长度，不包含结尾的0
- `int strcmp(const char *s1,const char *s2);`
  - 比较两个字符(串)
  - 若s1等于s2,若字符串长度相等，且每一个字符都对应相等，则返回0
  ```

    - 若s1大于s2,依次比较s1s2的每个字符，若存在s1的字符>s2的字符，且是第一次出现，返回这两个字符的差值（正数），结束比较
    - 若s1小于s2,依次比较s1s2的每个字符，若存在s1的字符<s2的字符，且是第一次出现，返回这两个字符的差值（负数），结束比较
- `char *strcpy(char *restrict dst,const char *restrict src);`
    - 把src的字符串拷贝到dst
    - restrict，关键字，表明src和dst不能重叠(两个字符串不能有相同的部分)（c99）
    - 返回dst字符串。
    - 复制一个字符串`char *dst = (char*)malloc(strlen(src)+1);  strcpy(dst,src);`，`strlen(src)+1`是因为strlen()函数获取的长度不包含字符串结尾'\0'，将src字符串拷贝到dst,dst必须具有足够的空间
- `char *strcat(char *restrict dst,const char *restrict src);`
    - 把src拷贝到dst的后面，接成一个长的字符串，从'\0'开始，结束加'\0'
    - 返回dst，dst必须具有足够的空间
- `char * strchr(const char *s,int c);`
    - 在字符串s中从左边找c第一次出现的位置
    - 没有找到返回NULL,找到返回指针指向的那个字符位置
- `char * strrchr(const char *s,int c);`
    - 在字符串s中从右边找c第一次出现的位置，返回一个指针
    - 没有找到返回NULL,找到返回指针指向的那个字符位置
- `char *strstr(const char *s1,const char *s2)`
    - 在字符串中寻找单个字符
- `char *strcasestr(const char *s1,const char *s2)`
    - 在字符串中寻找字符串
    - 寻找时忽略大小写

字符串函数	语法	解释
`strlen()`	`size_t strlen(const char *s);`	获取长度(不包过字符最后的\0)
`strcmp()`	`int strcmp(const char *s1,const char *s2);`	比较两个字符(串)
`strncmp()`	`int strncmp(const char *s1,const char *s2,size_t n);`	安全版本，size_t n比较前几个字符
`strcpy()`	`char *strcpy(char *restrict dst,const char *restrict src);`	把src的字符串拷贝到dst
`strncpy()`	`char *strncpy(char *restrict dst,const char *restrict src,size_t n);`	安全版本，size_t n最多能拷贝多少字符
`strcat()`	`char *strcat(char *restrict dst,const char *restrict src);`	把src连接到dst的后面
`strncat()`	`char *strncat(char *restrict dst,const char *restrict src,size_t n);`	安全版本，size_t n最多能连接多少字符
`strchr()`	`char * strchr(const char *s,int c);`	在字符串s中从右边找c第一次出现的位置
`strrchr()`	`char * strrchr(const char *s,int c);`	在字符串s中从左边找c第一次出现的位置
`strstr()`	`char *strstr(const char *s1,const char *s2)`	在字符串中寻找单个字符
`strcasestr()`	`char *strcasestr(const char *s1,const char *s2)`	在字符串中寻找字符串

7.第七章

结构类型--枚举

- 常量符号化`const int red  = 0;`
- 用符号而不是具体的数字来表示程序中的数字，增加程序的可阅读性
- `enum COLOR{RED,YELLOW,GREEN};`，枚举，用枚举比定义独立的`const int`变量要方便
- 枚举是一种用户定义的数据类型，它用关键字`enum`，声明如下：
  - `enum [枚举类型名字]{name0,name1,name2,...,nameN};`
- 枚举类型名字通常不使用，使用的时大括号里的名字，因为他们就是符号常量，他们的类型是int,值依次是从0到n，如`enum color{red,yellow,green};`，即red值是0，yellow值是1，green值是2，red是枚举的变量，0是枚举值
- `color`相当于一个数据类型，实际上c语言内部枚举就是int,每个枚举变量可以当作int使用
- `enum COLOR{RED,YELLOW,GREEN,NumCOLORS};`，其中NumCOLORS是枚举的计数器，它的值就是枚举变量的个数，这样遍历枚举量就很舒服了

- 声明枚举量的时候可以指定值`enum COLOR{RED = 1,YELLOW,GREEN = 5};`，RED枚举量是1，YELLOW枚举量是2，GREEN枚举量是 5
- `enum COLOR color = 0`，现在给枚举类型的变量赋不存在的整数值也是可以的
- 如果我们需要定义一些排比的符号量，用枚举比const int 方便，枚举比宏（macro）好，因为枚举有int类型

结构类型-结构类型

- 结构，一个结构就是一个复合的数据类型，里面有各种类型的成员，用一个变量来表达多个变量类型
- ```c
  #include<stdio.h>

  int main(int argc,char const *argv[]){

      struct date{
          int month;
          int day;
          int year;
      };

      struct date today;

      today.month = 06;
      today.day = 3;
      today.year = 2023;

      printf("Today's date is %i-%i-%i.\n",today.year,today.month,today.day);

      return 0;
  }
  ```
- 上述代码中`struct date{...};`是声明的结构体，所定义的变量today的类型名字是`struct date`
- 结构体的声明还可以写成`struct{...}today1,today2;`，其中`today1,today2`是变量名，这种声明的结构体没有类型名字
- 同理还可以这样声明结构体`struct date{...}today1,today2;`声明一个结构叫做`struct date`，同时定义两个变量`today1,today2`
- 结构体和变量一样，在函数内声明的结构类型只能在函数内部使用，所以通常在函数外部声明结构类型，可被多个函数使用
- 声明了这样一个结构体后，并定义了结构类型变量`today`，那我们就可以通过`today`访问结构体里面的成员变量`month,day,year;`
- 因为结构变量（本地变量、局部变量）没有初始值，所以我们要初始化
- ```c
  //两种初始化的方法
  struct date today2 = {06,4,2023};
  struct date today3 = {.month = 06, .year = 2023};//.day默认赋值0
  ```
- **结构和数组的区别**
  - 结构的成员变量可以是不同的类型，数组的成员变量必须是同一种类型
  - 结构用`.`运算符和变量名访问成员变量，如`today.day student.firstName`

    - 数组用`[]`运算符和下标访问成员变量，如`a[0] = 10;`
    - 对于整个结构，可以做赋值、取地址、传递给函数参数
    - 要访问整个结构，可以直接用结构变量的名字`today = (struct date){06,4,2023}`这里相同当于`today.month = 06,...`，同样的`today1 = today2`，也可以赋值运算。
    - 而对于数组，除了声明的时候可以用`{}`初始化，其他时候不能用`{}`赋值，也不能给一个数组赋值另一个数组
- **结构指针**
    - 与数组不同，结构变量的名字不是结构变量的地址，必须使用`&`运算符
    - `struct date *pDate = &today;`

结构与函数

- 结构作为函数的参数`int numberOfDays(struct date d){...}`
- 整个结构可以作为参数的值传入函数
- 这时候是在函数的内新建了一个结构变量，并复制调用者的结构的值，函数接收不是你定义的结构，而是与结构具有相同值的结构`d`，也就是在函数中复制了一个结构`d`，它的值和原来结构体的值一致
- 函数也可以返回一个结构。
- 我们没有一个`%什么`可以用`scanf`或`printf`一次性输入输出，但是我们可以定义一个函数来接收这个结构输入输出
- 我们无法通过直接把结构作为参数传入函数的方式取去修改原来结构的值，c在函数调用时是传值的，所以传结构和数组不同

```c
void getStruct(struct point p){
    scanf("%d",&p.x);
    scanf("%d",&p.x);
    //printf("%d %d\n",p.x, p.y);
}
void output(struct point p){
    printf("%d %d",p.x, p.y);
}
```

- 【解决方法1】：我们可以创建一个结构类型的函数，返回输入的结构值，并通过赋值的方式修改原来结构的值
    - ```c
      struct point getStruct(void){
          struct point p;
          scanf("%d",&p.x);
          scanf("%d",&p.y);
          return p;
      }
      void output(struct point p){
          printf("%d %d",p.x, p.y);
      }
      ```
    - 这里结构体像Java里的类，而我们写的函数像java的`getXXX,putXXX`构造方法
- 【解决方法2】：我们可以通过传入指针的方式修改原来结构的值,不需要创建新的结构
    - **指向结构的指针**
      ```

```
struct date {
    int month;
    int day;
    int year;
} myday;

struct date *p = &myday;

(*p).month  = 1;
p->month = 1;//与(*p).month用法一致
```

- `struct date *p = &myday`，与数组不同，myday的值不是地址，故需要取址运算符`&`
- 语句`p->month`，其中`->`是运算符，语句表示指针p所指的结构变量myday的成员变量month

- `getStruct()`，是输入函数，返回传入的指针，`print()`是输出函数，无返回值

```
struct point *getStruct(struct point *p){
    scanf("%d",&p->x);
    scanf("%d",&p->y);
    return p;
}
void print(const struct point *p){
    printf("%d,%d\n",p->x,p->y);
}
```

结构中的结构

- **结构数组**:

```
struct date dates[100];//初始化一个数组
struct date dates[] = {{2023,06,04},{2023,06,05}};
```

- **结构中的结构**

```
struct point {
    int x;
    int y;
};
struct line {
    struct point ptOne;
    struct point ptTwo;
};
struct line l = {{1,2},{3,4}};
struct line *lp;
printf("l.ptOne.x = %i\n",l.ptOne.x); //(l.ptOne).x
printf("&lp->ptOne.x = %i",&lp->ptOne.x); //&((lp->ptOne).x)
```

- 注意：没有`&lp->ptOne->x`这种写法，因为ptOne不是指针

- **结构中的结构数组** 略

联合

类型定义

- **自定义数据类型** `typedef`
- `typedef`关键字，c语言提供一个叫typedef的功能，来给已有的数据类型起一个新名字。
- `typedef int Length;`，使得Length成为int类型的别名。
- `typedef *char[10] Strings`，使得Strings成为*char[10]的别名。
- 这样，Length这个名字就可以代替int出现在变量定义和参数声明的地方：`Length a,b,len;` `Length number[10];`
- ```
  typedef struct ADate{
      int month;
      int day;
      int year;
  } Date;
  Date d = {06,05,2023};
  ```
- 这里`typedef`相当于将`struct ADate{...};`替换成了`Date`，简化了复杂的名字，改善程序可读性，注意这里的`Date`相当于自定义的数据类型，不是**变量名**
- 也可以写成这样`typedef struct ADate Date;`
- 或着连`ADate`都不写`typedef struct {...} Date;`？

联合

- ```
  union Elt{
      int i;
      char c;
  }elt1, elt2;
  ```
- `sizeof(union ...)`是`MAX{sizeof(每个成员)}`,所谓联合是占据同一片内存空间，例如上述elt1变量占据4个字节
- 同一时间内只有一个成员变量有效，对第一个成员变量初始化
- [拓]：通常电脑cpu是x86小端，内存的存放方式是低位在前

8.第八章

全局变量

- 定义在函数外部的变量是全局变量，具有全局的生存期和作用域，他们与任何函数无关，在任何函数内部都可使用
- 没有做初始化的全局变量默认初始化为0，指针会得到NULL，
- 只能用编译时刻已知的值来初始化全局变量（就是不能给个未知数），他们的初始化发生在main函数之前
- 如果函数内部存在与全局变量同名的变量，则全局变量被隐藏

静态本地变量

- 在本地变量定义时加上`static`修饰符就成为静态本地变量
- `static int all = 100;`
- 当执行完函数离开后，静态本地变量会继续存在并保存其值
- 静态本地变量的初始化只会在第一次进入这个函数时做，以后再次进入函数保持上次离开的值
- 静态本地变量实际上就是特殊的全局变量，他们位于相同的内存区域
- 静态本地变量具有全局的生存期，函数内的局部作用域
- `static`意思是局部作用域，函数内部访问

返回指针的函数

- 返回本地变量的地址是危险的
- *返回全局变量或静态本地变量的地址是安全的，但是不使用全局变量来在函数间传递参数和结果，尽量避免使用全局变量或静态本地变量
- *全局变量或本地变量的函数是线程不安全的
- 返回在函数内malloc的内存是安全的，但是容易造成问题
- 最好的做法是返回传入的指针

宏定义

- **编译预处理指令**
  - `#`开头的是编译预处理指令，不是c语言的成分，但离不开它们
- `#define`用来定义一个宏
  - `#define PI 3.14159`
  - 不是语句，结尾没有分号，名字必须是一个单词，值可以是任何东西
  - 在编译之前，编译预处理程序（cpp）会把程序中的名字替换成值，完全的文本替换
- 如果宏值是其他宏的名字，也会被在一次的替换
- 宏超过一行，最后一行之前的行末尾加`\`
- 宏的值后面出现的注释不会被当作宏的值的一部分
- **没有值的宏**`#define _DEBUG`，这里宏用于条件编译，后面的编译预处理指令来检查这个宏是否已经被定义
- **预定义的宏**
  - `__LINE__`行号
  - `__FILE__`文件名，全路径路径
  - `__DATE__`编译时的日期
  - `__TIME__`编译时的时间
  - `__STDC__`用于指示当前编译器符合 ANSI C 或 ISO C 标准

带参数的宏

- `#define cube(x)  ((x)*(x)*(x))`，宏可以带参数
- 一切都要有括号，整个值要括号，参数出现的每个地方要括号
- 可以带多个参数`#define MIN(a,b) ((a)>(b)?(a):(b))`
- 也可以组合嵌套其他宏，宏结尾不要加分号
- 宏在大型程序中使用普遍，可以非常复杂，如'产生'函数，`##, #`，没有类型，不会检查，部分宏，如带参数的宏会被inline函数代替
- 其他的编译预处理指令：条件编译 、error、...

多个源代码文件

- mian()函数代码太长可以分成多个函数，一个源文件太长适合分成多个.c文件，如何操作?
  - 在编译软件中新建一个项目然后把源代码文件放在项目里，编译器会把所以源文件编译连接起来
- 一个.c文件是一个编译单元，编译器每次编译只处理一个编译单元，形成.o目标代码文件，然后连接形成可执行程序

头文件

- 把函数的原型(声明)放到一个头文件(以`.h`结尾)中，需要在调用这个函数的源代码文件中`#include`这个头文件,如`#include"max.h"`,就可以让编译器在编译的时候知道函数的原型
- `#include`是一个编译预处理指令，和宏一样，在编译之前就处理了，它把那个文件的全部文本内容原封不动的插入到他所在的地方，所以也不一定要在.c文件的前几行`#include`
- `#include`有两种形式指出插入文件`""  <>`
  - `""`要求编译器在当前目录下寻找，没有再去指定目录找
  - `<>`让编译器只在指定目录去找
  - 通过编译环境和编译命令行参数也可以寻找指定头文件的目录
- `#include`不是用来引入库的
- `stdio.h`只有printf的原型，printf的代码在另外的地方某个.lib(Win)或.a(Unix)
- 编译器默认引入所有的标准库
- `#include <stdio.h>`只是为了让编译器知道printf函数的原型，保证我们调用函数时给出参数值的正确的类型
- 在使用和定义这个函数的地方都应该#include这个头文件
- 一般的做法是任何.c都应对应同名的.h，把所有的对外公开的函数和原型和全局变量的声明都放进去
- 在函数的前面加上`static`就使得它成为只能在所在编译单元被使用的函数，在全局变量前面加上static就使得他成为只能在所在编译单元被使用的全局变量

声明

- 变量的定义`int i;`，变量的声明`extern int i;`,声明不能初始化
- 声明不产生代码，定义产生代码
- 规定只有声明可以被放在头文件中，同一个编译单元，同名的结构不能被重复声明（避免头文件重复声明同名结构）
- **标准头文件结构**

```
#ifndef __LIST_HEAD__   //#ifndef __ANYSUM_H__  文件名.h
#define __LIST_HEAD__

int anySum(int m,int n);
extern int m;

#endif
```

  - 运用条件编译和宏，保证在这个头文件在一个编译单元中只会被`#include`一次
  - `#pragma once`也有上述作用

9.第九章

文件

格式化的输入输出

- `printf("%[flags][width][.prec][hIL]type")`
- **[flags]**
Flag	含义	示例[width]
`-`	左对齐	`%-9d`
`+`	在前面放+	`%-+9d`
`(space)`	正数留空	
0	0填充	`%-+09d`
- **[width][.prec]**
[width][.pre]	含义	示例
`number`	最小字符数	`%9.2f`
`*`	下一个参数是字符数	`printf("%*d",6,123);`*是6
`.number`	小数点后的位数	`%9.2f`
`.*	下一个参数是小数点后的位数	
- **[hIL]**
[hIL]修饰类型	含义	示例
`hh`	单个字节	`%hhd`
`h`	short	`%hd`
`l`	long	`%ld`
`ll`	long long	`%lld`
`L`	long double	`%Ld`

`scanf("%[flag]type")`

- **flag**

flag	含义	flag	含义
`*`	跳过`*"%*d%d",&num`	`l`	long ,double
`数字`	最大字符数	`ll`	long long
`hh`	char	`L`	long double
`h`	short		

`scanf`返回读入的变量数，`printf`返回输出的字符数

文件的输入输出

- *用`>`和`<`做重定向
- **FILE**
- `FILE *fopen(const char *restrict path, const char *restrict mode);`
- `int fclose(FILE *stream);`
- `fscanf(FILE*,...);`
- `fprintf(FILE*,...);`向文件输出
- `sprintf(format,"%%%s",STR_LEN-1);`向字符串输出
- **打开文件的标准代码**
  - ```
    FILE* fp = fopen("file","r");
    if(fp){
        fscanf(fp,...);
        fclose(fp);
    }else{
        ....
    }
    ```
  - `fopen("file","r")`，第一个参数是文件名，是一个字符串，第一个参数打开方式，也是一个字符串，下面是更多打开方式
打开方式	解释
r	打开只读
r+	打开读写，从文件头开始
w	打开只写，如果不存在则新建，如果存在则清空
w+	打开读写，如果不存在则新建，如果存在则清空
a	打开追加，如果不存在则新建，如果存在则从文件尾开始
..x	只新建，如果文件已存在则不能打开
    ```

二进制文件

- 所有的文件最终都会都是二进制的，二进制文件是需要专门的程序读写的文件，文本文件的输入输出是格式化，可能需要经过转码
- Unix用文本文件来做数据存储和程序配置，Unix的shell提供读写文本的小程序
- Window用二进制文件做程序配置，更接近底层
- 文本的优势是方便人类读写，而且跨平台，缺点是程序的输入输出要经过格式化，开销大
- 二进制优点是程序的读写快，缺点是人类读写困难，而且不跨平台，int的大小不一致，大小端的问题
- **配置**：Unix用文本,window用注册表。**数据**：数据量大点的放数据库。**媒体**：只能二进制，程序通过第三方库来读写文件，很少直接读写二进制文件
- **二进制读写**
  - `size_t fread(void *restrict ptr,size_t size,size_t nitems,FILE *restrict stream);`
  - `size_t fwrite(const void *restrict ptr,size_t size,size_t nitems,FILE *restrict stream);`
  - 参数`ptr`是读或写的那个变量的内存地址（如：结构，里面存放读的数据或写的数据），参数`size`是这块内存的大小（一个结构变量的大小），参数`nitems`是有几块这样的内存，`stream`是文件指针
  - `FILE *restrict stream`指针是最后一个参数，返回的是成功读写的字节数
  - `size_t nitems`，二进制文件的读写一般是通过对一个结构变量的操作来进行，`nitems`用来说明这次读写几个结构变量
- **文件的定位**
  - `int fseek(FILE *stream,long offset,int whence);`，fseek定位stream文件，offset定位多少长度，whence从哪里地位，方向是什么
  - `long ftell(FILE *stream);`，获取根据fp的位置获取文件的大小
    - **whence**的三个参数
    - `SEEK_SET`：从头开始
    - `SEEK_CUR`：从当前位置开始
    - `SEEK_END`：从尾开始（倒过来）
- 二进制文件如`student.datda`不具有可移植性，解决方案一是不用int，使用typedef具有明确大小的类型，最佳方案二是使用文本

位运算

按位运算

- 按位运算的运算符

符号	解释	示例
`&`	按位的与	`5A & 8C` 结果 `08`，比较每一位，都是1取1，否则取0
`|`	按位的或	`5A | 8c` 结果 `DE`，比较每一位，有一个1取1，否则取0
`~`	按位取反	`5A` 的结果 `A5`，每一位取反
`^`	按位的异或	`66 ^ 66` 结果为 `00`，比较每一位，位相同取0，位不相同取1
`<<`	左移	``
`>>`	右移	``

- 按位与`&`的两种应用
  - 让某一位或某些位为0，如`x & 0xFE`结果是使x的最低位为0
  - 取一个数中的一段，如以32位int为例`0xAABBCCDD & 0x000000FF`,则取出该int的最后一个字节AA
- 按位或`|`的两种应用
  - 使得一位或几位位1，如`x & 0xFE`结果是使x的最低位为1
  - 把两个数拼接起来，如`0x00FF | 0xFF00`结果为`0xFFFF`
- 逻辑运算和位运算的区别：逻辑运算是先将非0值变成为1，在进行位运算

左移右移

- **左移**`i << j`
- `i`中的所有位向左移动`j`位，而右边填入0
- 所有小于int的类型，移位以int的方式来做，结果是int
- `x <<= n`等价于`x*=2^n`
- **右移**`i >> j`
- `i`中的所有位向左移动`j`位
- 对于unsigned的类型，左边填零
- 对于signed的类型，左边填入原来的最高位（保持符号不变）
- 所有小于int的类型，移位以int的方式来做，结果是int
- `x >>= n`等价于`x/=2^n`
- 移位的位数没有负数

*位段

- 把一个int的若干个位组合成一个结构

```
struct {
    unsigned int leading : 3;
    unsigned int FLAG1 : 1;
    unsigned int FLAG2 : 1;
    int trailing : 11;
}
```

- 可以直接用位段的成员名称来访问，具有不可移植性，所需位超过一个int时会采用多个int

10.第十章

可变数组

可变数组-顺序表

- 可变数组也就是**数据结构**中线性表的动态内存分配的方式实现
- 关于数据结构的线性表**动态内存分配**我现在有两种实现方式（其实有很多种写法，看你想怎么实现了）
- **【实现方法一】**
  - 动态内存分配-传结构体指针
  - ```
    //下述结构体是一个可变数组
    #ifndef _ARRAY_H_
    #define _ARRAY_H_

    typedef struct{
        int *arrary;
        int size;
    } Array;        //我们也常用 * Array 重命名一个结构体指针，但是不推荐用

    Array array_create(int init_size);      //创建可变数组
    void array_free(Arrray *a);     //free掉可变数组
    int array_size(const Array *a);     //返回可变数组的长度
    int * array_at(Array *a, int index);        //
    void array_inflate(Array *a,int more_size);     //增加可变数组的长度

    #endif
    ```
  - 为什么不推荐使用`*Array`重命名一个结构体指针?
    - 虽然当我们使用结构体指针的时候更方便
    - 但是当我们要在一个函数中定义`Array a;`的时候实际上是定义了一个结构体指针，我们无法定义一个局部结构体变量（所以单链表中typedef了一个结构体，一个结构体指针）
    - 同时`Array a;`也不符合C语言的常规写法，我们不容易想到这是一个结构体指针
  - 这里初始化`Array array_create(int init_size)`我们传进去的是长度，返回的是结构体
  - 增加可变数组的长度`void array_inflate(Array *a,int more_size);`我们传进去的是**结构体指针**和**要增加的长度**
- **【实现方法二】**
  - 动态内存分配-传结构体
  - ```
    #include<stdio.h>
    #include<stdlib.h>
    #define InitSize 10     //表长的初始定义

    typedef struct{
        int *data;
        int MaxSize;
        int length;
    } SqList;

    void InitList(SqList &L);
    void IncreseSize(SqList &L,int len);
    int Length(SqList &L);
    ```

```c
int LocateElem(SqList &L, int e);
int GetElem(SqList &L,int i);
bool ListInsert(SqList &L, int i, int e);
bool ListDelete(SqList &L, int i, int e);
void PrintList(SqList &L);
bool Empty(SqList &L);
void DestroyList(SqList &L);

void InitializeValue(SqList &L);
```

  - 这里初始化`void InitList(SqList &L);`传进去的是一个我们定义结构体
  - 增加可变数组的长度`void IncreseSize(SqList &L,int len);`我们传进去的是**结构体**和**要增加的长度**

- 基本操作的实现方式有很多，我们不能说那种方式最好，具体用哪种方法看生产环境
- 传指针相比传结构体更轻巧，而传结构体又避免了定义的繁琐

可变数组的缺陷

- 当可变数组很大的时候，每次增加新的长度，拷贝需要消耗很长时间
- 当我们有剩余内存的时候，容易内存申请失败，不过高效
- 如何解决--单链表

11.第十一章

链表

链表-单链表

- `main.c`文件

```c
#include <stdio.h>
#include <stdlib.h>
#include "node.h"
/*
*   @单链表-不带头节点
*/

int main(int argc, char *argv[]) {
    List list;
    list.head = NULL;
    int number;

    do{
        scanf("%d",&number);
        if( number != -1){
            add(&list,number);
        }
    }while( number != -1);

    print(&list);


    printf("deleteNode(&list,1) = %d\n",deleteNode(&list,1));
```

```
    printf("deleteNode(&list,4) = %d\n",deleteNode(&list,4));

    print(&list);

    freeNode(&list);

    print(&list);

    return 0;
}

//add节点
void add(List *pList,int number){
    //add to linked-list
    Node *p = (Node*)malloc(sizeof(Node));
    p->value = number;
    p->next = NULL;

    // find the last node
    Node *last = pList->head;
    if( last ){
        while( last->next ){
            last = last->next;
        }
        //attach
        last->next = p;
    }else{
        pList->head = p;
    }
}

//print链表
void print(List *pList){
    Node *p;
    for(p=pList->head;p;p=p->next){
        printf("%d\t",p->value);
    }
    printf("\n");
}

//deleteNode 删除节点 ，传入值，返回节点的值
int deleteNode(List *list,int e){
    Node *p,*q;
    int i = 0;
    for (q=NULL,p=list->head; p; q=p,p=p->next){
        if( p->value == e){
            if( q ){
                q->next = p->next;
            }
            else{
                list->head = p->next;
            }
            i = p->value;
            free(p);
            break;
```

```
        }
    }
    return i;
}

//free链表
void freeNode(List *list){
    Node *p,*q;
    for(p=list->head; p; p=q){
        q = p->next;
        free(p);
    }
    list->head = NULL;

}
```

- `node.h` 文件

```
#ifndef _NODE_H_
#define _NODE_H_


typedef struct _node{
    int value;
    struct _node *next;
} Node;

typedef struct _list{
    Node * head;
    Node * tail;
} List;

void add(List *pList,int number);
void print(List *pList);
int deleteNode(List *list,int e);
void freeNode(List *list);
#endif

/**
*   @@add的另外三种写法


//add节点
void add(Node * head,int number){
    //add to linked-list
    Node *p = (Node*)malloc(sizeof(Node));
    p->value = number;
    p->next = NULL;
    // find the last node
    Node *last = head;
    if( last ){
        while( last->next ){
            last = last->next;
        }
```

```
        //attach
        last->next = p;
    }else{
        head = p;
    }
}

//add节点
Node * add(Node * head,int number){
    //add to linked-list
    Node *p = (Node*)malloc(sizeof(Node));
    p->value = number;
    p->next = NULL;
    // find the last node
    Node *last = head;
    if( last ){
        while( last->next ){
            last = last->next;
        }
        //attach
        last->next = p;
    }else{
        head = p;
    }
    return head;
}

//add节点
void add(Node **pHead,int number){
    //add to linked-list
    Node *p = (Node*)malloc(sizeof(Node));
    p->value = number;
    p->next = NULL;
    // find the last node
    Node *last = *pHead;
    if( last ){
        while( last->next ){
            last = last->next;
        }
        //attach
        last->next = p;
    }else{
        *pHead = p;
    }
}

*/
```

> 第一版@2023.6.27
>
>
>
> 个人博客网站:**山外云海.cn**