C

0.1 初识

```c
#include <stdio.h>

int main(){

    printf("hello world");
    return 0;
}
// main函数是程序的入口,有且只有一个.
// include <stdio.h>是包含一个头文件,在包含内声明的函数都可以在本程序中使用了(printf函数).
// printf:输出函数.
// gcc test.c -o test && ./test  编译并且运行.
```

0.2 关键字

1. 44个关键字

传统的 C 语言（ANSI C）有 32 个关键字：

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

1999年，ISO 发布 C99，添加了 5 个关键字：

inline	restrict	_Bool	_Complex	_Imaginary

2011年，ISO 发布 C11，添加了 7 个关键字：

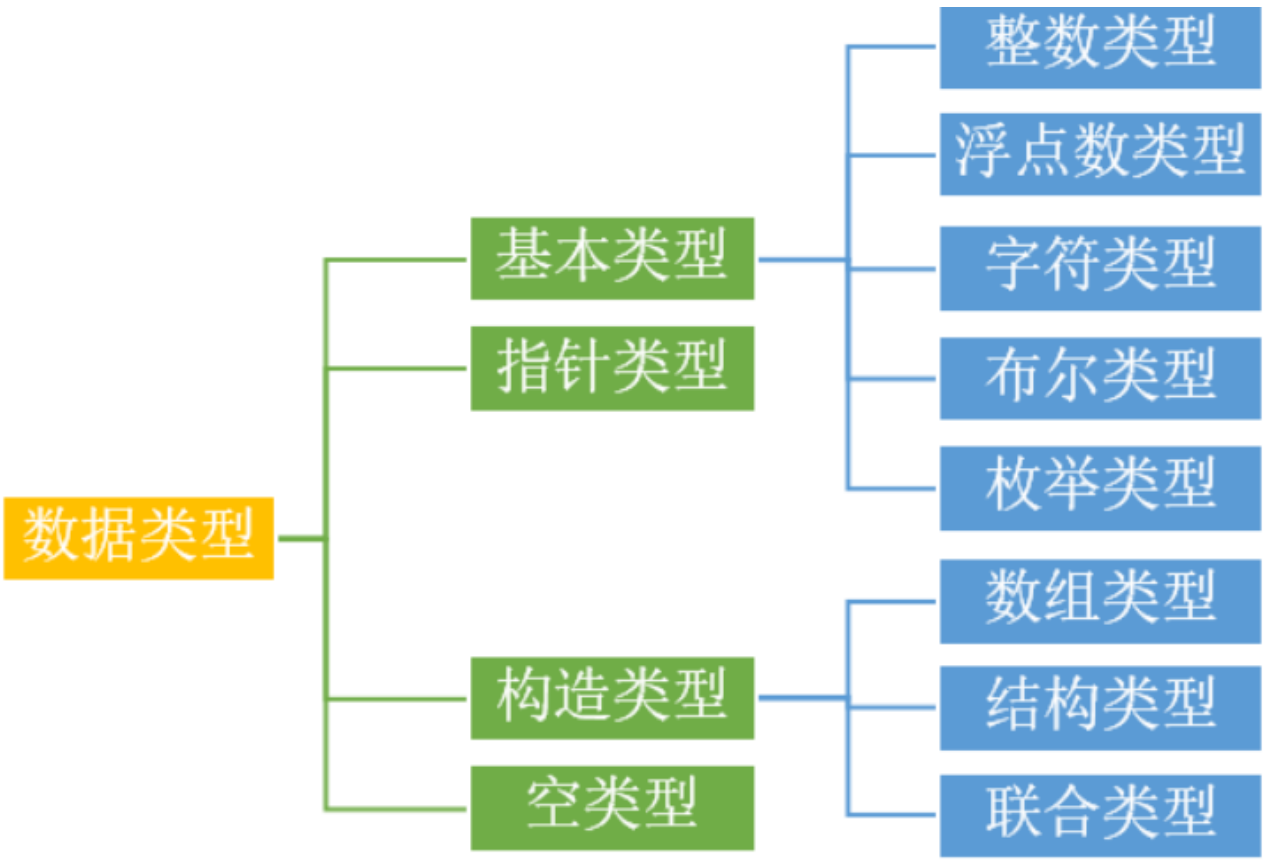
_Alignas	_Alignof	_Atomic	_Static_assert	_Noreturn	_Thread_local	_Generic

0.3 宏定义

```
1. 宏定义
        实质是定义符号常量,格式:
        #defind 标识符 常量
        解释:defind将以后所出现的该标识符全部替换成常量
        #defind URL 'baidu.com'
2. 将变量变成常量
        利用const关键字修饰
        const int a = 1;  //只读而不可写入
```

```c
#include <stdio.h>

#defind NAME "lee"

int main(){
    printf("名字是%s",NAME);
    return 0;
}

// NAME后必须是双引号
```

0.4 数据类型




```
1. 整数类型
        short int                2字节              -2^15 ~ 2^15-1
        int                      4字节              -2^31 ~ 2^31-1
        long int                 4字节              -2^31 ~ 2^31-1
```

```
        long long int               8字节               -2^63 ~ 2^63-1
2. 浮点数
        float                       4字节
        double                      8字节
        long double                 16字节
3. 字符类型
         char                       1字节               -2^7 ~ 2^7-1
4. 布尔类型
         _Bool                      1字节               -2^7 ~ 2^7-1
5. 字符串定义(2种)
          C中没有字符串类型,定义语法:
          char name[] = "lee"  // 必须是双引号,系统自动加'\0'结束、
          取值: printf("%c",*name); //l
          char *name = "lee"; // 字符串指针
          取值: printf("%c",*name) // L

          C语言并没有提供字符串类型,指针一旦指向某个字符串的首字符地址就可以获取字符串了
```

0.5 运算符优先级

```
1. 优先级1
        [],(),.,->,变量名++,变量名--
2. 优先级2
        负号,强制类型转化,++变量名,--变量名,*(取值运算),&(取址运算),!(逻辑非),~(按位取反),sizeof
3. 优先级3
        /,*,%
4. 优先级4
        +,-,<<,>>
5. 优先级6
        比较运算符
6. 优先级7
        ==,!=
7. 优先级8
        位运算符(与,异或,或)
8. 优先级9
        &&,||
9. 优先级10
        三目运算符
         (a>b) ? a:b
         a是真值,b是假值
10. 优先级10
        赋值运算符(=),增强运算符
```

0.6 流程控制

```
1. 分支(3个)
        if(){}-else{}.
        if(){}-else if(){}-else{}.
        switch(表达式){
           case 常量表达式1:
                代码块或语句;
                break;
```

```
            default:
             语句  }
2. 循环(3个)
        for(int i = 0;i < 10;i++){
           printf("hello");
        }
        while(布尔表达式){
           //循环体
        }
        do{
          //循环体
        }while(表达式)
```

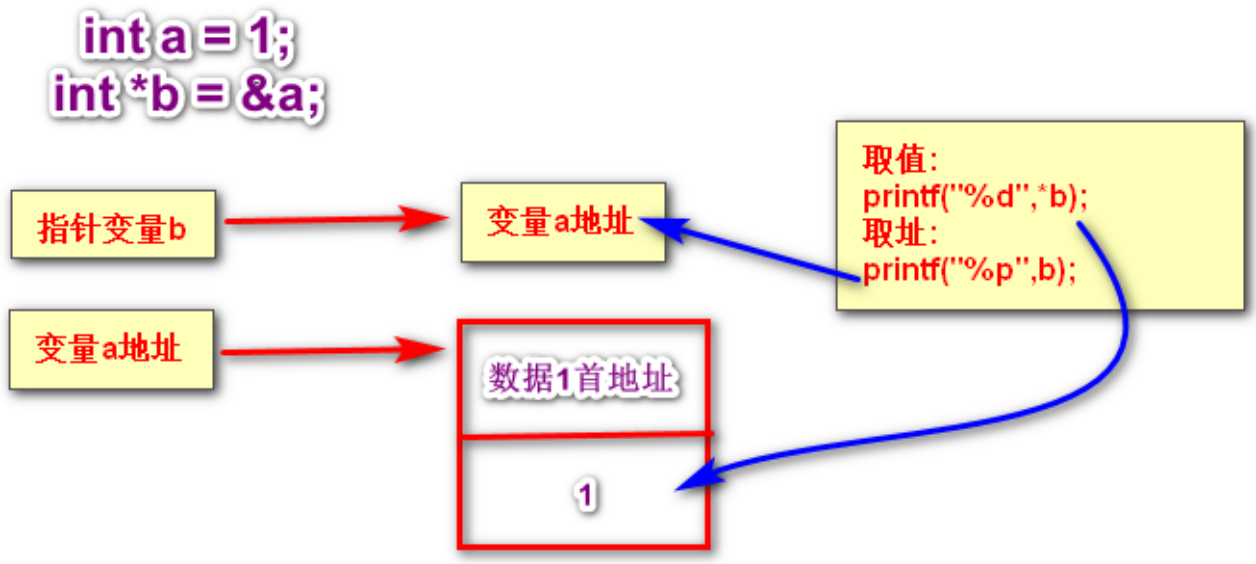
0.7 数组

```
1. 定义
        类型 数组名[元素个数]
        int a[6]; //a数组能存放6个整型值
        存储类型相同的一组数据的结构.
2. 特性
        int a[10] = {0}; //全部为0
        int d[] = {1,2,3}    //偷懒,此时长度默认为3
        ***数组可以动态定义:
        int n;
        scanf("%d",n);
        char s[n+1];
3. 字符串数组定义
        char strs[] = "liyang";
4. 二维数组定义
        类型 数组名[常量表达式][常量表达式]
        int a[6][6];  //6行6列
        char b[4][5]; //4行5列
        int a[3][2] = {
               {1,2},
               {3,4},
               {5,6}
         };
```

0.8 指针

```
1. 指针变量
        指针变量就是一个名字(标识符),指针变量存放的指针的地址.
2. 类型
        指针变量也有类型,该类型就是存放的地址指向数据的类型.一个指针占4个字节,来存放一个地址.
3. 例子
        pa -> 11000
        11000 : 10000
        10000 : 'F'

        pa是指针变量,所在的地址是11000,11000所在的地址存放的值是10000这个地址,10000地址的存放的值是
        真实的数据.
4. 定义指针
```

```
        类型名 *指针变量名
        int *pb = 某地址; //定义了一个指向整型的指针变量
5. 指针变量一定存放某个地址
        int a = 1;
        int *b = &a;    //&取址运算,b指针指向a

        b---->变量a的地址
        变量a的地址---->真实数据的地址和1
```



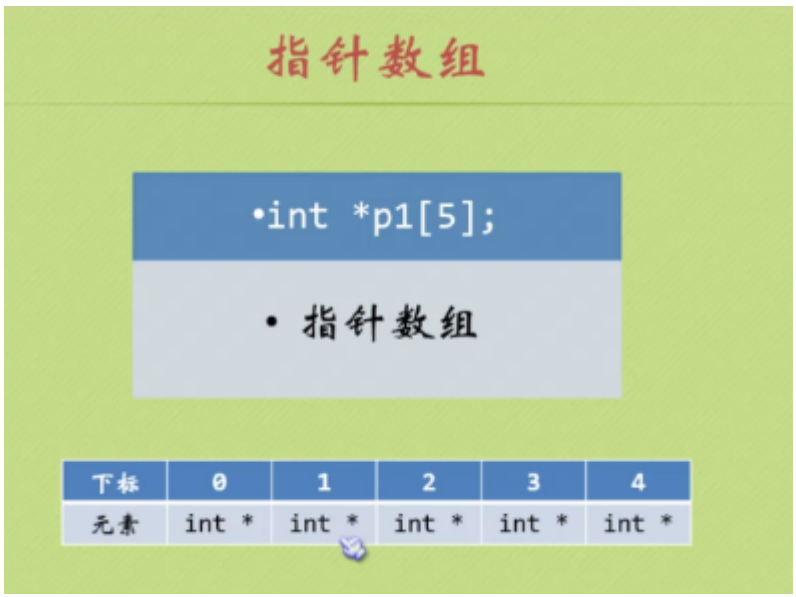

```
6. ***区别
          指针变量就是一个左值,而数组名实质上是一个地址
```

```
#include <stdio.h>

int main(){
    char str[] = "liyang"; //数组名本身就是数组所在的地址
    char *target = str;
    int count = 0;
    while(*target++ != '\0'){    //*取值运算符顺序在++后面,target代表地址,地址加1在取值
        count++;
    }
    printf("长度是%d",count);

    return 0;
}
```

```
char str[] = "liyang";
char *target = str;
// target指向的是数组所在的地址,该地址就是首元素的地址
// *target++ :
target地址加1在去取值得到每个元素值
```

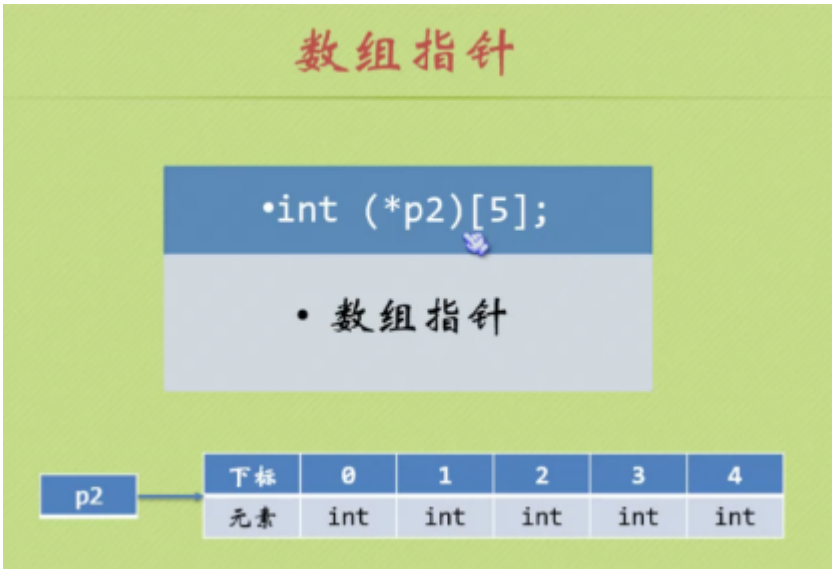

```
int main(){
    char str[] = "liyang";
    printf("数组的地址是:%p\n",str);
    printf("首元素地址是:%p",&str[0]);
    return 0;
}

// 数组的地址是:000000000062FE10
// 首元素地址是:000000000062FE10
```

0.9 指针数组



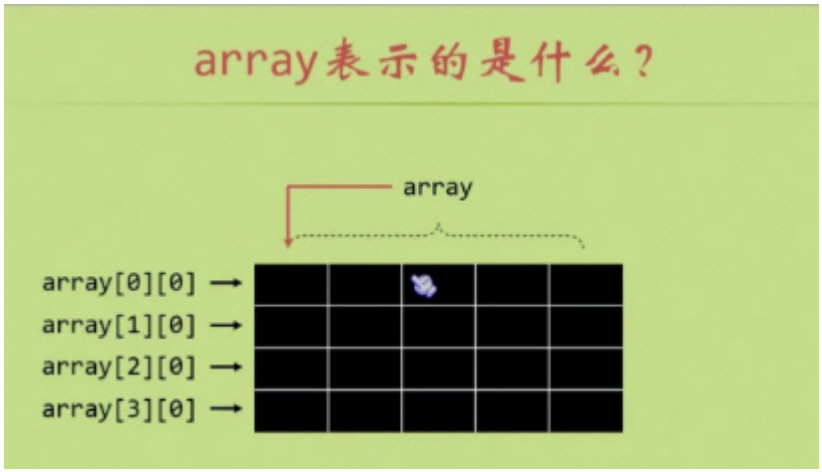
```
1. 理解
        指针数组是数组,该数组的每一个元素是指针
2. 定义
        int a = 1;
        int b = 2;
        int *p[2] = {&a,&b};
3. 遍历取值
        for(i = 0;i<5;i++){
        printf("%d\n",*p1[i]);     //先[]后*
        }
```

0.9 数组指针




```
1. 理解
        数组指针是指针,指向的是一个数组
2. 定义
        int (*p)[5]
        p是一个指针,指向长度为5的数组.
3. 取值
        int temp[3] = {1,2,3}
        int (*p)[3] = &temp;
        for(int i = 0;i<3;i++){
          printf("%d",*(*p+1));
        }
```

1.0 指针和二维数组




```
1. 定义
        int array[4][5] = {...};    //4行5列
        int *p = array;             //array表示第一行的地址---第一组5个元素数组的地址
        每一行地址依次:array,array + 1,array + 2,...
2. 取值
        *(array+1) == array[1]
```

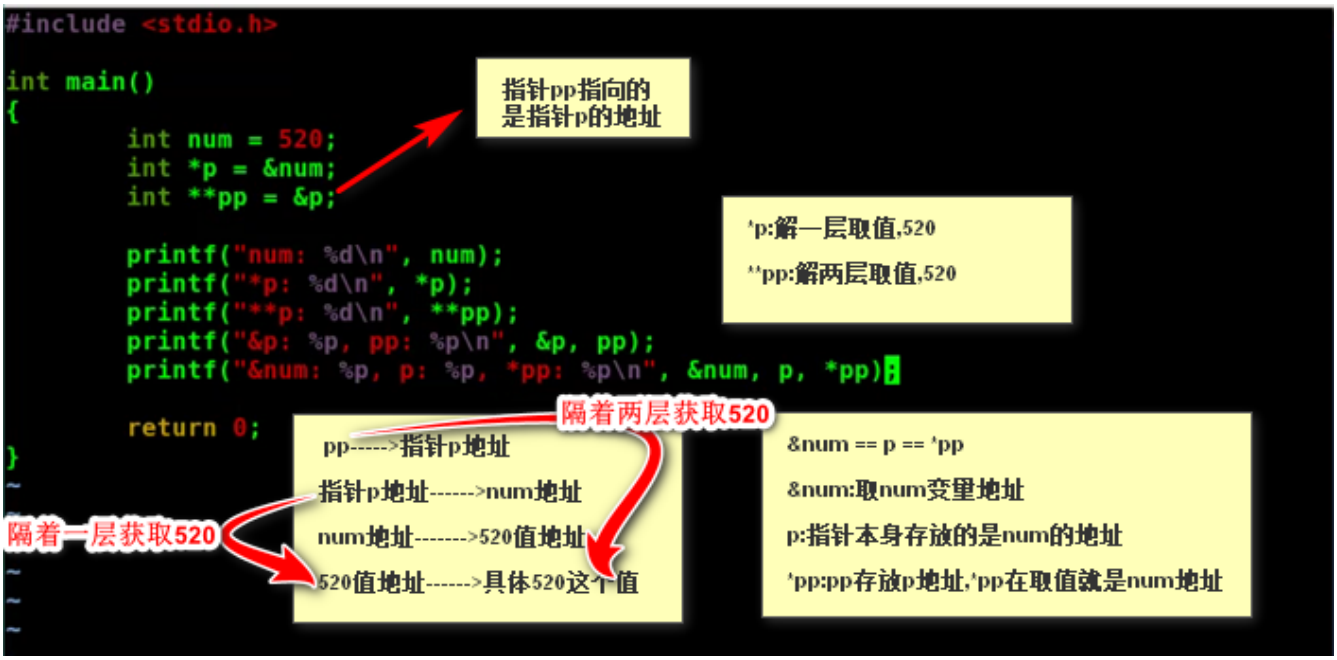
1.1 void和null指针

```
1. void
        void:无类型,void指针:通用指针,可以指向任何类型的数据.
```

```
        void *p;
2. null
        c语言中该指针不指向任何数据就是null指针.是一种编程习惯,当指针不知道指向谁时可以
        初始化为null.
        null指针解引用是非法的.对指针进行解引用先检查是否为null.
        int *a = NULL;
3. 指针类型转化
        int a = 1;
        int *b = &a;
        void *c;
        c = b;
        printf("%d",*(int *)c);  // 1
```

1.2 指向指针的指针

```
1. 定义
        int **pp = &p; //pp指针指向的是指针的地址
```



1.3 mybook[2]

```
1. 定义
        char *books[] = {
            "Python",
            "C语言"
        };
        char **mybook[2];  //指针数组中存在着两个指向指针的指针元素
        mybook[0] = &books[0];
        mybook[1] = &books[1];
        printf("%s",*mybook[0]); // Python
2. 解释
        mybook是存放指针的指针数组
```

1.4 指针和常量

```
1. 指向常量的指针
        const int a = 1;  // 将变量变成常量,只读但不可写
        const int *b = &a; // 指向常量的指针必须用const修饰,const在最前面
        指向常量的常量指针指针本身指向可以变,但是指向的那个值是不可变的
        int c = 2;
        b = &c; // 成功
        *b = c; // 报错
2. 常量指针
        const int a = 1;
        int c = 2;
        int * const b = &c;  // const在*后面就是常量指针
        指向非常量的常量指针指针本身指向不可变,但是指向的值可以变
        *b = a; // 报错
        b = &a; // 成功
3. 指向常量的常量指针
        指针本身不可以被修改同时指向的值也不可以被修改
        int c = 2;
        const int a = 1;
        const int * const b = &a; // 指向常量的常量指针
        *b = c; // 报错
        b = &c; // 报错
4. 注
        指针只要指向常量就必须用const修饰.
        const 在最前面是修饰指针是指向常量的指针(此指针可以指向常量可以指向变量).
        const 在*后面修饰指针是常量指针(此种条件只能指向变量).
        const 在前也在后就是指向常量的常量指针(此指针可以指向常量可以指向变量).
5. 指向“指向常量的常量指针”的指针
        const int a = 1;
        const int * const b = &a;
        const int * const *bb = &b;
        bb里面存放的是b的地址.
        printf("%d",**bb); // 1
```

1.5 函数

```
1. 定义
        类型名 函数名(参数列表){
              函数体
        }
        ps:如果类型名是void,则函数体不返回任何类型东西,有类型名则必须返回该类型的东西.
2. 规范
        模块化程序设计.
        先定义后调用.
        每个函数都有独立的作用域.
3. 流程
        可以先声明,在调用,最后定义也是可以的.
        void a();  //声明.
        int main(){
            a();  // 调用.
            return 0;
        };
        void a(){ // 最后定义.
```

```
            printf("hello");
        }
4. 参数列表
        01.指针可以当作参数:
        void a(int *a,int *b){
            printf("%d,%d",*a,*b); // 指针解引用取值.
        }
        int main(){
            int a1 = 1;
            int a2 = 2;
            a(&a1,&a2); //传址.
        }
        02.传数组
        void get_array(int a[10]){ // 形参看上去是数组但是实质是接受了一个地址.
                                   // a的实际长度是4字节,指针长度.
            a[6] = 66;
            for(int i = 0;i<10,i++){
                printf("%d\n",*(a+i));
            }

        }
        int main(){
            int a[10] = {1,2,3,4,5,6,7,8,9,0};  // a的实际长度是40字节.
            get_array(a); // 此时实质是将数组第一个元素的地址传递给了形参.
            // 在main函数中打印a数组此时a[6]就变成了66.
        }
        03.可变参数--待续
        #include <stdarg.h>  // 必须引入头文件
        // 三个变量:va_list,va_start,va_arg,va_end
        int sum(int n,...){ // 第一个参数n指定后面有多少个参数,...是占位符
            int

        }
```

1.6 函数和指针结合

```
1. 指针函数
         指针函数是函数.
         char *getWord(char c){
            switch(c){
                case 'A':return "正确";
            }

         }
         char a = 'A';
         getWord(a);  // getWord是指针函数,返回的值是"正"的首地址.
         char *s;
         s = getWord(a);

         printf("地址是:%p\n",*getWord(a));
         printf("地址是:%p\n",*s);
2. 函数指针
          函数指针是指针,是指向一个函数的指针.
```

```
int (*p)(参数列表){函数体}

int square(int num){
   return num * num;
}
int main(){
   int num = 2;
   int (*fq)(int); // 函数指针,参数是int.
   fq = square; // 让函数指针指向该square函数.  fq = &square  取出函数地址给fq也行
                // 函数名相当于函数地址.
   printf("%d\n",(*fq)(num));
}
```

3. 函数指针作为参数

```
int add(int nmu1,int num2){
    return num1 + num2;
}
int sub(int num1,int num2){
    return num1 - num2;
}
int cal(int (*fp)(int,int),int num1,int num2){
    return (*fp)(num1,num2);
} // 形参是函数指针,实参应该传函数地址

int main(){
    int res1,res2;
    res1 = cal(add,1,2);
    res2 = cal(suvb,1,2);
    printf("%d\n",res1);
    printf("%d\n",res2);
}
```

4. 函数指针作为返回值

```
int add(int num1,int num2){
    return num1 + num2;
}

int sub(int num1,int num2){
    return num1 - num2;
}

int cal(int (*fq)(int,int),int num1,int num2){
    return (*fq)(num1,num2);
}

int (*select(char op))(int,int){
    switch(op){
        case '+':return add;
        case '-':return sub;
    }
}

int main(){
    int num1,num2;
    char op;
    int (*fp)(int,int);
    printf("输入一个式子如(1+2):");
    scanf("%d%c%d",&num1,&op,&num2);
    fp = select(op);
    printf("%d %c %d = %d\n",num1,op,num2,cal(fp,num1,num2));
}
```

target:
根据输入的符号自动计算数值

int (*fp)(int,int):
是函数指针,表明指向一个函数,指向谁???

int (*select(char op))(int,int){...}
括号优先级高:
先看:select(char op),实质就是一个函数,形参是char类型参数,返回该函数指针
int(*)(int,int):是函数指针

int (*select(char op))(int,int){...} 实质就是一个int类型并带有两个int形参的函数指针
select就会返回int类型并带有两个int形参的函数指针

fp = select(op)

所以fp就指向了select返回的函数(指针).

cal:return (*fq)(num1,num2)---->获取了函数指针再去调用对应的函数

```
输入一个式子如(1+2):3+5
3 + 5 = 8

--------------------------------
Process exited after 6.145 seconds with return value 0
请按任意键继续. . .
```

1.7 局部和全局变量

1. 全局变量
    如果不对全局变量进行初始化,则会初始化为0.
2. 屏蔽
    同名局部变量会屏蔽全局变量
3. 作用域
    代码块作用域
    文件作用域:在代码块之外声明的标识符都具有文件作用域,从声明开始到结束,函数名就是例子
    原型作用域:函数参数列表
    函数作用域
4. 定义和声明
    声明:告诉编译器别报错,在后面有定义
    定义:为变量申请了内存空间进行存储了

1.8 编译器流程

1.9 生存期和存储期

1. 静态存储期
    具有文件作用域的变量属于静态存储期,程序关闭才会释放
2. 自动存储期
    属于代码块作用域的变量是自动存储期,代码库块结束就会释放
3. 存储类型
    存储类型是指存储变量值的内存类型
    5种:
    auto:自动变量,代码块中声明的变量默认就是auto.
    register:寄存器变量,将一个变量声明为寄存器变量,有可能被cpu存放于寄存器中(空间有限)
        不能使用取址运算操作符.
    static:静态局部变量,属于静态存储期,程序结束才释放,生存期变化作用域不变化.
    extern:
    typedef:为数据类型定义别名

2.0 递归

1. 概念
    函数调用自身就是递归
    递归必须要有递归结束条件
    把一个大问题分成若干个小问题,小问题和大问题解决的方式一样(分治法)
2. 递归实现阶乘

```
#include <stdio.h>

long fact(int num){
    long result;
    if(num > 0){
        result = num * fact(num - 1);
    }
    else{
        result = 1;
    }
    return result;
}
```

```
int main(){
    int num,result;
    printf("请输入一个整数:\n");
    scanf("%d",&num);
    result = fact(num);
    printf("结果是:%d",result);
    return 0;
}
```

2.1 汉诺塔

```
// froms,temp,to
int hnt(int num,char a,char b,char c){
    if(num == 1){
        printf("%c------>%c\n",a,c);
    }
    else{
        hnt(num-1,a,c,b);
        printf("%c------>%c\n",a,c);
        hnt(num-1,b,a,c);
    }
}

int main(){
    int num;
    printf("请输入一个整数:");
    scanf("%d",&num);
    hnt(num,'A','B','C');
}
```

2.2 快速排序

```
1. 思想核心
        通过一趟排序将待排序的数据分割成独立的两部分,
        其中一部分的所有元素别另外一部分的元素小,
        然后分别对这两部分继续进行排序,重复以上步骤(递归的体现).
2. 例子
             5,2,9,4,7,8,6,3,0,1
        索引  0,1,2,3,4,5,6,7,8,9
        找分界点: (0 + 9) / 2 = 4     索引4是分界点对应元素是7,7就是基准点.
        找第一部分大于等于7的元素,找第二部分小于等于7的元素
        索引:i<j
        5,2,9,4,7,8,6,3,0,1
        i                 j

        5,2,9,4,7,8,6,3,0,1
            i             j          9和1互换

        5,2,1,4,7,8,6,3,0,9          7和0互换
                i       j
```

```
        5,2,1,4,0,8,6,3,7,9          8和3互换
                  i   j

        5,2,1,4,0,3,6,8,7,9          i==j,第一次循环结束了
                    i
                    j
        i++,j--
        i:7,j:6

        5,2,1,4,0,3,6       8,7,9
        基准点:4             基准点:7
```

```
void quick_sort(int array[], int left,int right){
    int i = left, j = right;
    int temp;
    int pivot;

    pivot = array[(left + right) / 2];

    while(i <= j){

        //从左到右找到大于等于基准点的元素
        while(array[i] < pivot){
            i++;
        }

        //从右到左找到小于等于基准点的元素
        while(array[j] > pivot){
            j--;
        }

        //如果i<=j则互换元素
        if(i <= j){
            temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }

    }

    if(left < i){
        quick_sort(array,left,i);
    }

    if(j < right){
        quick_sort(array,j,right);
    }
}


int main(){
    int array[] = {73,108,111,101,78,105,115,104,67,46,99,111,109};
    int i,length;
```

```
    length = sizeof(array) / sizeof(array[0]);
    printf("开始排序:\n");
    quick_sort(array,0,length-1);

    printf("排序后的结果是:\n");

    for(i = 0;i < length; i++){
        printf("%d ",array[i]);
    }

    return 0;

}
```

2.3 动态内存管理

```
1. malloc
        #include <stdlib.h>
        void *malloc(size_t size);
        是指针函数,作用是申请size字节大小的内存块,返回值是指向该内存块的指针,类型是void.
2. free
        void free(*ptr);
        free是释放,因为malloc申请的内存都存在于堆上,如果不及时释放内存,会造成内存泄漏.
        free(指向内存块的指针).
```

```
#include <stdlib.h>

int main(){
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if(ptr == NULL){
        printf("申请失败");
        exit(1);
    }

    printf("请输入一个数字:\n");
    scanf("%d",ptr);

    printf("输入的整数是%d",*ptr);
    free(ptr);
    return 0;
}
```

```
3. 内存泄漏
        导致内存泄漏两种情况:
        隐式内存泄漏,内存块没有及时free函数释放
        丢失内存块地址
```

```
int main(){
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    printf("请输入一个数字:\n");
    scanf("%d",ptr);
    int num = 5;
    ptr = &num;   // 指针从新指向了其他值,内存块的地址只有ptr知道,所以丢失了
    free(ptr);
    return 0;
}
```

4. 申请一大块内存地址

```
int main(){
    int num,i;
    printf("请输入个数:\n");
    scanf("%d",&num);
    int *ptr = NULL;
    ptr = (int *)malloc(num * sizeof(int));

    for(i = 0;i < num;i++){
        printf("请输入第%d个整数:",i + 1);
        scanf("%d",&ptr[i]);
    }

    printf("录入的整数是:\n");
    for(i = 0;i < num;i++){
        printf("%d ",ptr[i]);
    }
    free(ptr);
    return 0;
}
```

5. malloc并不会初始化内存空间,不会清0
        string.h标准快提供函数功能
        memset:使用一个常量字节来填充内存空间(0).
        提供一个高效的接口来处理内存数据.

```
#include <string.h>
#define N 10

int main(){
    int *ptr = NULL;
    int i;

    ptr = (int *)malloc(N * sizeof(int));
    if (ptr == NULL){
        exit(1);
    }
```

```
    memset(ptr,0,N * sizeof(int));  // 指向内存块指针,初始化值,内存尺寸
    for(i = 0;i < N;i++){
        printf("%d ",ptr[i]);
    }

    free(ptr);


    return 0;
}
```

```
6. calloc
        void *calloc(size_t nmemeb, size_t size);
        calloc函数在内存中动态申请nmemb个长度为size的连续内存空间(总空间尺寸:nmemeb * size)
        这些内存空间全部被初始化为0.
        该方法是对memset的优化.
```

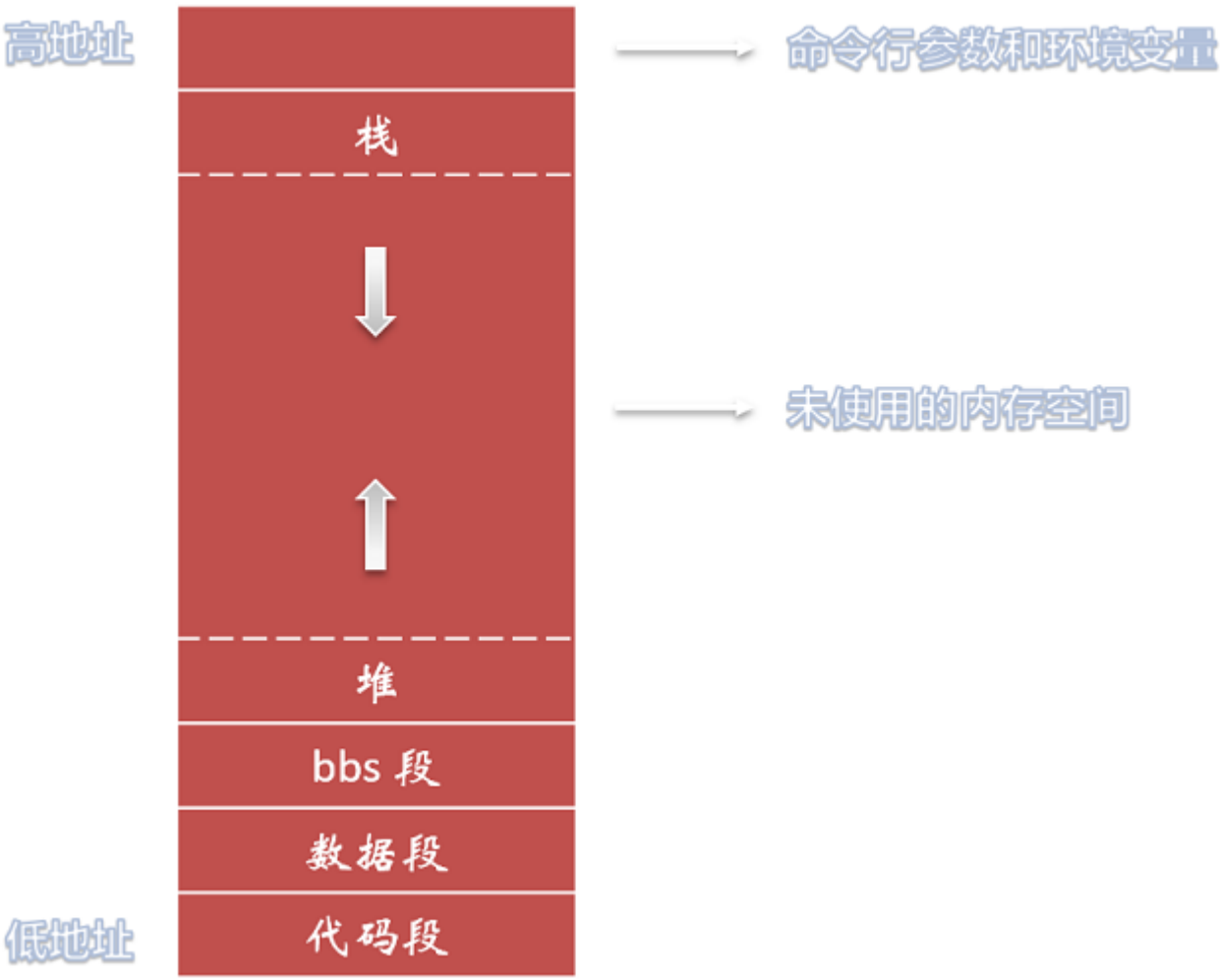
```
int main(){
    int num,i;
    printf("请输入个数:\n");
    scanf("%d",&num);
    int *ptr = (int *)calloc(num,sizeof(int));  // 内存总尺寸是num*4字节,并且内存数值初始化为0

    for(i = 0;i < num;i++){
        printf("第%d个数是:\n",i+1);
        scanf("%d",&ptr[i]);
    }

    printf("您录入的数是:\n");
    for(i = 0;i < num;i++){
        printf("%d ",ptr[i]);
    }

    free(ptr);
    return 0;

}

// int *ptr = (int *)calloc(8,sizeof(int));
// 等价于:
// int *ptr = (int *)malloc(8 * sizeof(int));
// meset(ptr,0,8 * sizeof(int));  初始化为0
```

```
7. 对内存空间进行扩展(以前申请的不够用)
        思路:重新申请新的内存空间,将原来的数据搬过去.
        void *realloc(void *ptr,size_t size);
        该函数将移动内存空间的数据并返回新的指针.
        原先的ptr指针不能指向栈(局部变量),必须指向堆.
        如果ptr参数为NULL,realloc则调用malloc函数
        如果size参数为0,并且ptr指针不为NULL,则相当于调用free函数
```

```
// memcpy函数也可以重新分配内存

int main(){
    int *ptr1 = NULL;
    int *ptr2 = NULL;

    // 第一次申请内存
    ptr1 = (int *)malloc(10 * sizeof(int));

    //第二次申请内存
    ptr2 = (int *)malloc(20 * sizeof(int));

    // 将ptr1的10个数据拷贝到ptr2中
    memcpy(ptr2,ptr1,10);
    free(ptr1);
    free(ptr2);

    return 0;
}
```

```
// 根据用户每输入一个整数取重新申请内存空间,直到输入-1则停止,最后打印出所有数据
int main(){
    int i,num;
    int count = 0;
    int *ptr = NULL;

    do{
        printf("请输入一个数(输入-1结束):\n");
        scanf("%d",&num);
        count ++;

        ptr = (int *)realloc(ptr,count * sizeof(int)); //第一次ptr为NULL,相当于调用malloc函数
        if(ptr == NULL){
            exit(1);
        }

        ptr[count - 1] = num;
    }while(num != -1);

    for(i = 0;i < count;i++){
        printf("%d ",ptr[i]);
    }

    free(ptr);

    return 0;
}
```

2.4 内存布局



1. bss
   存放未初始化的全局变量和未初始化的局部静态变量，区段中的数据在程序运行前将被自动初始化为数字0.
2. 数据段
   存放已经初始化的全局变量和局部静态变量.
3. 代码段
   代码段通常是指用来存放程序执行代码的一块内存区域，字符串常量存在于此.

2.5 堆栈

1. 堆概念
   存放进程中被动态分配的内存段，大小不固定，可动态扩展或缩小.
   malloc函数将新分配的内存动态添加到堆里.
   free函数将被释放的内存从堆中被删除.
2. 栈概念
   栈是函数执行的内存区域，局部变量，函数参数，函数的返回值都是在栈里.
3. 区别
   堆是由程序员手动申请和释放
   栈是由系统自动分配和释放
4. 生存周期
   堆生存周期：由动态申请到程序员主动释放为止，不同函数之间均可自由访问.
   栈生存周期：由函数调用开始到函数返回时结束，函数之间的局部变量不能互相访问.
5. 地址

堆和其他区段一样,从低地址向高地址发展.堆容易跨字节,相比于栈不环保.
栈相反,由高地址向低地址发展.栈是连续存放的.

```
// 验证函数可以自由访问堆
int *func(void){
    int *ptr = NULL;
    ptr = (int *)malloc(sizeof(int));
    if(ptr == NULL){
        exit(1);
    }
    *ptr = 666;
    return ptr;
}

int main(){
    int *ptr = NULL;
    ptr = func();

    printf("%d\n",*ptr);
    free(ptr);

    return 0;
}
```

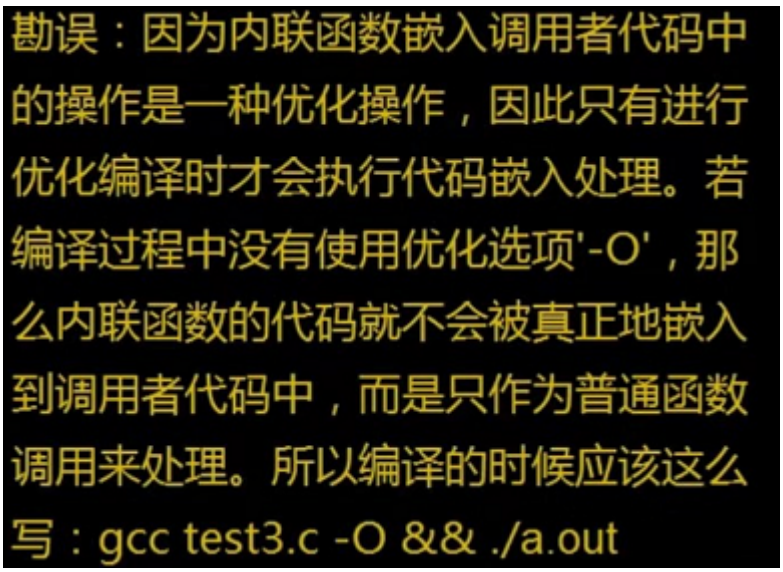
2.6 高级宏定义

```
1. 真理
        不管宏定义多么复杂,它都是发挥替换作用.
        编译器不会对宏定义进行语法检查.
        宏定义不加分号,不是语句.
        #undef来终止宏定义.
        宏定义允许嵌套.
2. 不带参数的宏定义
        #define PI 3.14
        // 宏定义嵌套
        #define V PI*3
3. 带参数的宏定义
        #define MAX(x,y) (((x) > (y)) ? (x):(y))  // 括号必须带着
        int x=1,y=2;
        MAX(x,y);
4. 和函数区别
        函数参数需要类型,但是宏定义不用,因为宏定义只是单纯的替换.
5. #
        #define STR(s) # s    // #会将后面的参数变成一个字符串
        printf("%s",STR(LiYang));
6. ##
        #define TOGETHER(x,y) x ## y  // ##是记号连接运算符
        printf("%d\n",TOGETHER(2,5)); // 25
7. 可变参数
        #define SHOWLIST(...) printf(# __VA_ARGS__)  // #变成字符串,__VA_ARGS__将所有参数给
                                                     // SHOWLIST()
        SHOWLIST(Li,520,3.14\n); // Li,520,3.14
```

```
        // 可变参数允许空参数:
        #define PRINT(format,...) printf(# format,## __VA_AGRS__)
        PRINT(num = %d\n,520);
        PRINT(LIYANG\n); //不给可变参数传参数,为空.只给format
```

2.7 内联函数

```
1. 内联函数
        提高函数的调用效率,因为函数是基于栈内存的,每次调用都是申请和释放.
        语法:在函数定义之前加inline
             inline int square(){
                     ;
             }
2. 优势
        内联函数不会调用来调用去的,直接在调用出展开,不会有栈空间的申请.但是会增加
        代码编译时间,因为每一处都要替换,编译器有时候也很聪明,即使不写也会自动变成内联.
```



2.8 结构体

```
1. 作用
        将多种不同类型的数据存放在一起.
2. 声明
        struct 结构体名称{
          结构体成员1;
          结构体成员2;
          ...
        };
3. 声明例子
        struct Book{
          char title[128];
          float price;
          unsigned int date;
        };
4. 定义
        // 声明
        struct Book{
```

```
          char title[128];
          float price;
          unsigned int date;
        } book; // 定义在这里也可以,两种方法.
        struct Book book; //定义了该结构体
5. 引用结构体变量
        book.title = "C语言"; // 用点进行引用
6. 初始化结构体变量
        struct Book book = {.price = 48.8};  //可以不按顺序
        记:
        struct A{
          char a;
          int b;
        } a = {'x',520};  // 声明定义初始化结合

        ***结构体所占的字节数受对齐的影响.
```

2.9 结构数组和指针

```
```