函数

函数的定义

格式：

```
return_type function_name(parameter list)

{

    function body

}
```

*return_type：指函数返回值的类型；当定义一个不需要返回值的函数时为void

*parameter list：定义了函数中的参数。这里的参数是形式参数；而当函数被调用时主调函数中的实际参数会将自身传递给形式参数。

*形参与实参：形参只在定义函数时使用，目的是接受函数被调用时接受的实际参数。实际参数则是主调函数中已经被赋值的参数。在调用过程中，实参必须被赋值，且数据类型与个数需要与形参完全对应。

*形参占用内存的问题：形参只有被调用时才分配内存单元，调用结束后立即释放。在程序不调用该函数时形参无法被使用也不占用内存。

*特别地，当被定义函数的参数是指针类型变量时,在调用该函数的过程中，传给函数的是实参的地址，在函数体内部使用的也是实参的地址，即使用的就是实参本身，所以在函数体内部可以改变实参的值。

*main函数定义方式解释：https://blog.csdn.net/u011846123/article/details/88203094

函数的声明

```
return_type function_name( parameter list );
```

需要注意的是，在括号内书写参数时，类型是必须的但名称可以省略。 //(int a,int b)和(int,int)都是合法的表述

有关变量

*在程序中，局部变量和全局变量的名称可以相同，但是在函数内，如果两个名字相同，会使用局部变量值，全局变量不会被使用。同样的，在函数内部如果形式参数被用作局部变量，其优先级仍然高于全局变量。

*https://www.runoob.com/w3cnote/cpp-static-usage.html

递归算法

```
void recursion()
{
   statements;
   ... ... ...
   recursion(); /* 函数调用自身 */
   ... ... ...
}

int main()
{
   recursion();
}
```

数组作变量

*数组中的元素就是下标变量，与普通变量没有区别，作为函数实参调用，单向地将值传送给形参。

*当数组名作为变量时传送地址名；（数组内第一个元素所对应的地址）/同时要求形参和实参都需要是类型相同的数组。（例如一维数组，二维数组，字符数组）

这么做的原因是形参数组并不实际存在，因此编译器不会为其分配内存单元。在形参数组取得实参数组的地址单元之后，由于具有相同的地址，实际上形参与实参数组会调用同一个数组并共用内存。

在函数调用声明中实现这一点比较简单，只需要把变量名称加上数组格式即可：

```
return_type function_name(int array[i]);
```

然后在主调函数调用语句中用array代替array[i]；

相似的情形出现在类似于scanf这样的函数中：

scanf("%d",&a[i])：使用下标访问赋值是需要指定其地址，&起指定地址的作用；

```
int a[10];
for(; ; ){
    scanf("%d", a + i);
}
```

这一段表示从首地址a开始每次地址增加i，而scanf获取的输入值则被赋给地址序号为'a+i'的数组元素。其因为直接访问地址则不需要'&'。

指针

基础操作

初始化：int *pointer;

赋值：poiner= &i;

指针即为变量存在的时候存储该变量的内存地址；而指针变量则为人为定义的专门用来存放其他变量的地址的变量。

在定义指针变量的时候，'*'表明这是一个指针变量。

```
int main() {
    int i=2000;
    int *pointer;
    pointer = &i;
    printf("%d\n", ? pointer);
    return 0;
}
```

在这段代码中，'?'处填写不一样的运算符获得的结果也是不一样的。当?取*的时候,此时pointer表示取pointer所存储的指针对应的变量的值，所以输出结果为2000；当？取空格时，意为直接以十进制打印pointer所存储的值，即为变量i的地址1172306932，实际地址为0x8959dff7f4;当？取&时，取址运算符则会直接取变量pointer的地址打印为67106800，实际值为0x291a3ff89c。

*在定义函数的时候：例如int function(int arr[],int n)，在编译过程中会默认把数组当成指针变量处理，即 * arr。此时传递的实参为数组首元素的地址。

有关数组的调用和声明

一共有四种方式

1.形参和实参都用数组名：

```
void main()
{
    int a[10];
    f(a,10) ;
}
void f(int x[],int n)
{
    ……
}
```

2.形参使用指针变量，实参用数组名：

```
void main()
{
    int a[10];
    f(a,10) ;
}
void f(int *a,int n)
{
    ……
}
```

3.形参和实参都用指针变量：

```
void main()
{
    int a[10],*p=a ;//此处*为定义p为指针变量。
    f(p,10);
}
void f(int *x,int n)
{
    ……
}
```

4.实参为指针变量，形参为数组名：

```
void main()
{
    int a[10], *p=a ;
    f(p,10);
}
void f(int x[],int n)
{
    ……
}
```

有关二维数组指针

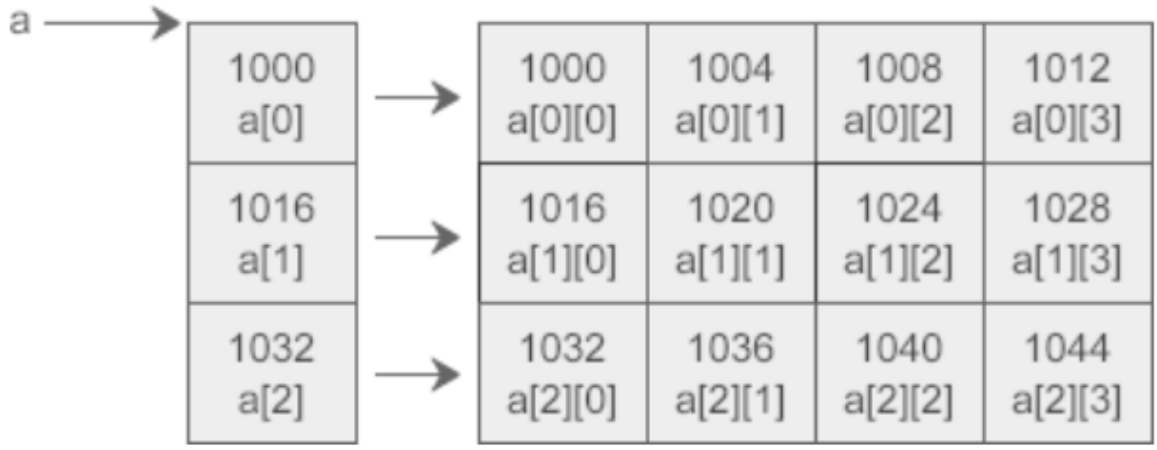
对于二维数组a[m] [n]，(m行n列），其包含三个int [4]，即三个包含4个元素的一维数组。

所以，对二维数组的指针初始化，需要对第一行，也就是第一行所对应的一维数组，代码如下：

```
int (*p)[4] = a;
```

[ ]的优先级高于*，( )是必须要加的，如果赤裸裸地写作int *p[4]，那么应该理解为int *(p[4])，p 就成了一个指针数组，而不是二维数组指针。

一般地，二维数组结构如下：（在排序中行的优先级大于列，但事实上一切二维数组在计算机语言中都会被当成线性的一维数组处理）



对指针进行加法（减法）运算时，它前进（后退）的步长与它指向的数据类型有关，p 指向的数据类型是int [4]，那么p+1就前进 4×4 = 16 个字节，p-1就后退 16 个字节，这正好是数组 a 所包含的每个一维数组的长度。也就是说，p+1会使得指针指向二维数组的下一行，p-1会使得指针指向数组的上一行。这一点在debug时体现在汇编语言中指针p地址的增加上。

关于取某一个元素的值/地址：

a[0] [0]:(*(p)) /表示地址/　*(*(p)) /表示取值/

类似的，*(*(p+a)+b)表示第a行第b列；

指针数组和二维数组指针的区别

定义的不同：

```
int *(p1[5]);  //指针数组，可以去掉括号直接写作 int *p1[5];
int (*p2)[5];  //二维数组指针，不能去掉括号
```

指针数组和二维数组指针有着本质上的区别：指针数组是一个数组，只是每个元素保存的都是指针。二维数组指针是一个指针，它指向一个二维数组。

字符串指针调用中的一个问题

声明一个函数用来将一个字符串的内容复制并覆盖另一个字符串：

当使用数组名时运行正常，而使用指针时编译成功但输出异常；

```
#include <stdio.h>

int main(){
    void copystr(char *from,char *to);
    char *a = "Viking1";
    char *b = "Viking2";
    copystr( from: a, to: b);
    printf( format: "%s\n%s",a,b);
}
void copystr(char *from,char *to)
{
    for(;*from!='\0';from++,to++)
    {
        *to = *from;
    }
    *to = '\0';
}
```

```
运行报错：Process finished with exit code -1073741819 (0xC0000005)
编译报错：SIGSEGV (Segmentation fault)
```

解决方法：将*b改为b[]。

分析：

Segmentation fault:内存泄露/溢出： 当一个进程执行一个无效的内存引用，或发生断错误时，会触发SIGSEGV信号，内核默认的动作就是终止该进程.一般出现在动用了非法指针的时候。

反汇编查看源代码：

```
        {
            *to = *from;
0043188F  mov         eax,dword ptr [to]
00431892  mov         ecx,dword ptr [from]
00431895  mov         dl,byte ptr [ecx]
00431897  mov         byte ptr [eax],dl
        }
```

断点出现在最后一步mov,这四句话表达的意思是：

将内存地址[to]中数据赋值给eax，[from]中数据赋值给ecx，将ecx内数据传递给寄存器dl，寄存器dl再将数据发送给eax。说明数据赋值给eax时发生错误。

*dword ptr[]：即表示一个指向四字节数据的指针，方括号中填写内存地址；若将dword换为byte则说明是一个字节，word是两个字节。

*dl：一种寄存器；

观察使用char命令定义字符串/指针时汇编语言的差别：

指针（char *p)：

```
dword ptr [a],offset string "Viking1" (0437B30h)
```

offset：将数值回送变量或标号的偏移地址值；

即为前面定义的四字节地址指针a被后面字符串的偏移地址(0437B30h)填充。

字符串名（char a[]）：

```
00E045CF  mov         eax,dword ptr [string "Viking1" (0E07B30h)]
00E045D4  mov         ecx,dword ptr ds:[0E07B34h]
00E045DA  mov         dword ptr [a],eax
00E045DD  mov         dword ptr [ebp-0Ch],ecx
```

ds:数据寄存器；

第一行不解释；第二行表示将ds寄存器存储的地址值直接存入ecx；第三行表示将eax内数据赋值给a所在地址；第四行表示为字符串预留空间（？）；

由此可见字符串名和指针定义的内部实现是完全不同的，这就是错误出现的原因。

上述实例中问题出现的原因就是，在语句char *b = "Viking2"中，由于b作为指针变量指向字符串的地址，b作为局部变量被存储在栈中，而"Viking2"则被作为字符串保存在静态存储区中。静态存储区的数据只能读不能写，所以覆写"Viking2"时就会出现错误。

C语言中的内存区域划分
1. 栈区（stack）:存放函数形参和局部变量（auto类型），由编译器自动分配和释放
2. 堆区（heap）:该区由程序员申请后使用，需要手动释放否则会造成内存泄漏。如果程序员没有手动释放，那么程序结束时可能由OS回收。
3. 全局/静态存储区：存放全局变量和静态变量（包括静态全局变量与静态局部变量），初始化的全局变量和静态局部变量放在一块，未初始化的放在另一块
4. 文字常量区：常量在统一运行被创建，常量区的内存是只读的，程序结束后由系统释放。
5. 程序代码区：存放程序的二进制代码，内存由系统管理

参考文章：https://wenku.baidu.com/view/70607d5602f69e3143323968011ca300a6c3f6de.html

其它的一些特殊的指针变量

**p：二级指针。表示p所指向的地址内存放了一个指向另一个变量指针。

int *p()：p为带回一个指针的函数，该指针指向整型数据；

int(*p)()：p为指向函数的指针，函数返回一个整型值；

参考：https://www.zhihu.com/question/399008520

int main(int argc,char* argv[])：argc表示程序内部发送给main函数的参数个数；argv[]表示指针数组，其中每一个数组都表示一个字符指针，每个字符指针指向命令行中每个参数中的第一个字符。

特殊的C注释方式

#if #endif

如果为#if 0，则二者之间内容被注释；

如果为#if 1,则二者之间内容正常执行

在数组作为函数传参时的一些注意事项

数组被当作一个指针变量被传入被定义函数中，指针变量保存的是数组首元素a[0]的地址。函数通过该地址找到整个数组并正常进行操作，在被定义函数中对数组中任意一个元素进行操作都是可行的。

以下传参方式均可行：

```
int f(int *a);
int f(int *);
int f(int *a[]);
int f(int a[]);
int f(int *[]);
```

数组变量是一种特殊的指针。由于其拥有不需要*定义即为指针变量的特征，可以通过这种方式来输出指针变量。反之，一个数组的首元素的值也可以通过指针变量输出（和下面的代码示例正好相反）

```
int *a = &b;
printf("",*a);
printf("",a[0]);
```

这两种printf语句的输出等价。

需要注意的是，数组变量是const类型，初始化后不可以被赋值。（常量指针）. 可以理解为：

```
int a[]= <=>int *const a
```

常量指针一旦被定义，在程序寿命过程中就不可以被再次改变。这里的不可被改变指的是指针变量和指向的地址之间的对应关系不可被改变：

```
int * const q =&i;//正向对指针变量赋值，可行
*q =26//反向通过对指针变量赋值改变指针变量指向的变量值，可行
q++//改变了指针变量指向的变量地址，不可行
```

这种情况是指针变量不可改变，还有另一种情况是指针变量指向一个const类型：

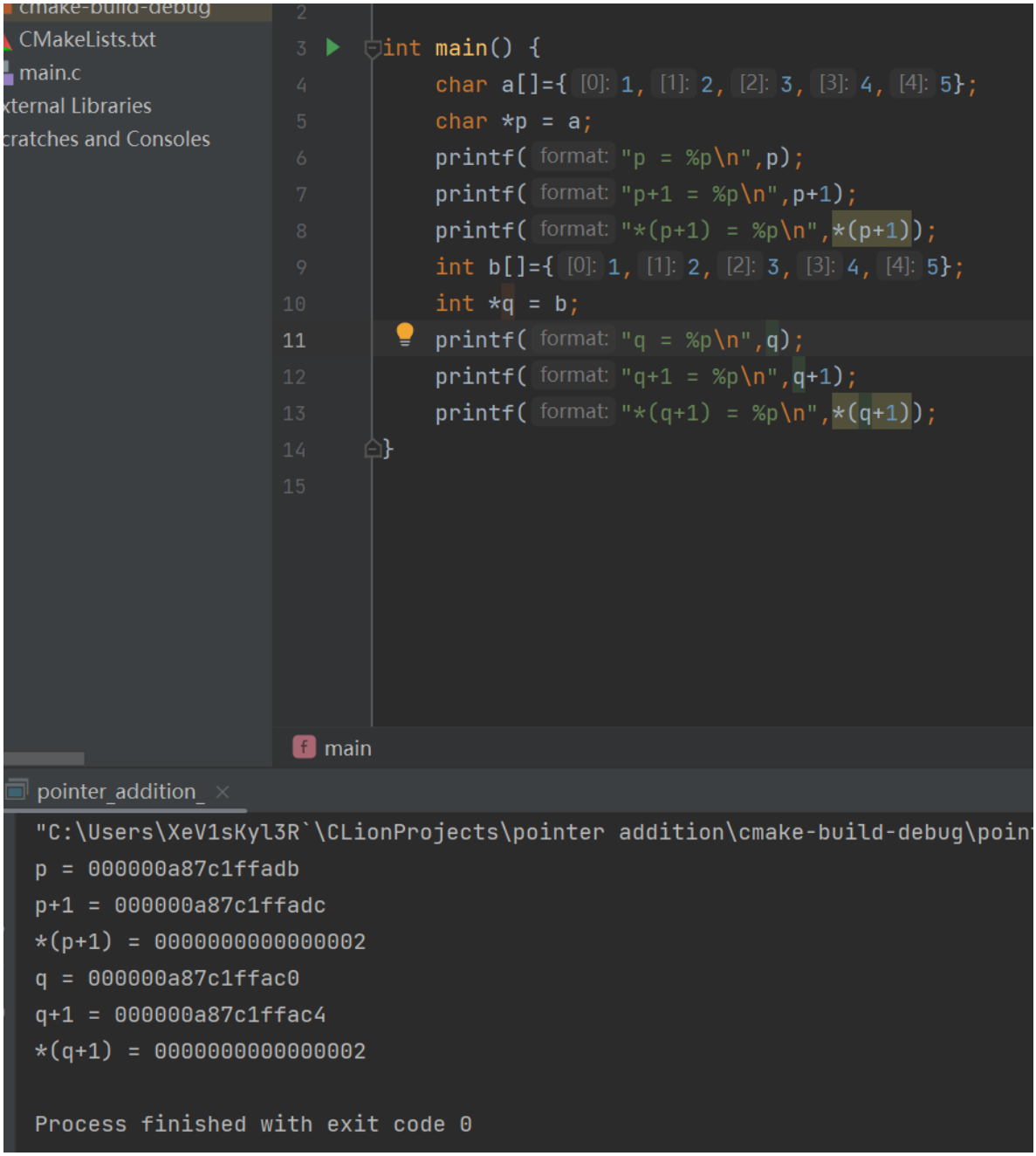
```
const int *p =&i;//使通过p修改i的途径被封死
*p =26;//不可行，因为不能通过p修改i
i = 26;//可行
p=&j;//可行
*p =&j//不可行，这里我理解为const的东西其实就是*p而不是p，也就是不是实际的变量p而是一种抽象的指向实现。
```

这里其实挺抽象的……,实际上并不是使指向的变量变成一个不可被改变的常量，const只是限定变量i不可以通过指针变量p修改，通过其他方式修改是可行的。同样的，如果把*p指向的东西修改成其他变量也一样可以运行。

指针（数组） 的计算

通过一个代码来比较一下指针计算的一些区别：

```
#include <stdio.h>

int main() {
    char a[]={1,2,3,4,5};
    char *p = a;
    printf("p = %p\n",p);
    printf("p+1 = %p\n",p+1);
    printf("*(p+1) = %p\n",*(p+1));
    int b[]={1,2,3,4,5};
    int *q = b;
    printf("q = %p\n",q);
    printf("q+1 = %p\n",q+1);
    printf("*(q+1) = %p\n",*(q+1));
}
```



从这里可以发现这几件事情：

1.字符串中char类型每个元素占据一个地址，int类型占据四个地址。

2.如果使用指针的加减法，则*(p+n)表示在一段连续含有变量的地址段中选取p后面第n个数据（注意，这样的加减法只在面向一片连续空间时才有意义）。

3.当两个指针相减的时候，输出的是两个指针指向的地址之差n/sizeof(parameter)（除以该字符串变量类型所对应的长度）

递增：*p++中++优先级比*高，所以会先运行++，++作为后缀则先取p的值，然后让p递增1（此时结果为p），再取*,则地址仍然对应p。（先取地址对应的值，再让地址递增）

4.char *的定义方式在定义完成之后不可以修改；而char[]则可以被修改。char *指的是某个指向字符的指针，可能指向字符数组，只有其指向的字符数组以\0结尾时才可以说指向的是字符串。

空指针和指针的类型变换

空指针void *在不知道指向什么类型的变量的时候使用。可以通过(void *)p来强制把指针变量p的类型转化为空指针。

比如可以这样转换指针的类型：

```
int *p=&i;
void *q=(void*)p
```

为什么一般强制类型转换的时候必然有一边是void空指针呢？因为指针是一种强类型（联系上方的代码），因为指针不仅需要表示变量的地址，同时也需要来引用所定位的地址处的内容。

这里贴一段微软官方对C语言指针类型转换的说明文档：

> 指向值的一个类型的指针可以转换为指向另一类型的指针。 但是，由于对齐需求和存储中不同类型的大小，结果可能是未定义的。 指向对象的指针可转换为指向其类型要求小于或等于严格存储对齐的对象的指针，然后再次返回而不做更改。
>
> 指向 `void` 的指针可转换为/自指向任何类型的指针，且不受限制或不丢失信息。 如果结果转换回原始类型，则将恢复原始指针。
>
> 如果指针转换为另一个类型相同但具有不同的或其它限定符的指针，则新指针与旧指针相同（新限定符强加的限制除外）。
>
> 指针值也可以转换为整数值。 根据以下规则，转换路径取决于指针的大小和整型的大小：
>
> 如果指针的大小大于或等于整型的大小，则指针的行为类似于转换中的无符号值，除非它无法转换为浮点值。
>
> 如果指针小于整型，则指针首先转换为与整型大小相同的指针，然后转换为整型。
>
> 相反，整型可以基于以下规则转换为指针类型：
>
> 如果整型与指针类型的大小相同，则转换只会促使整数值被视为指针（无符号整数）。
>
> 如果整型类型的大小与指针类型的大小不同，则使用表从带符号整型类型转换和从无符号整型类型转换中给定的转换路径，首先将整型转换为指针的大小。 然后将其视为一个指针值。
>
> 值为 `0` 的整型常量表达式或强制转换为类型 `void*` 的此类表达式可以通过类型强制转换、赋值或与任何类型的指针进行比较来进行转换。 这将产生与同一类型的另一个 `null` 指针相等的 `null` 指针，但此 `null` 指针与指向函数或对象的任何指针不相等。 常数 `0` 以外的整数可以转换为指针类型，但结果是不可移植的。

动态内存分配

malloc()：动态分配内存。向malloc申请的空间大小以字节为单位。返回的类型是void *（也就是空指针）需要将类型转换为自己需要的类型。

可以通过(void *)p来强制把指针变量p的类型转化为空指针。

语法如下：

```
(int *)malloc(n*sizeof(int))
```

与之相对的是，分配的内存功能完成之后需要free()函数释放内存。需要注意的是free()只能是申请到的内存的首地址。

字符串函数

strlen()：返回字符串长度（这里的字符串长度不包括\0的位置，也就是看到几个字母就返回字母的数量）（相比之下sizeof()包括了\0的位置）

函数定义实现：

```
int len(const char* s)
{
    int idx = 0;
    while(s[idx] != '\0')
    {
        idx++;
    }
    return idx;
}
```

strcmp()：比较两个字符串,字符串相等返回0，不相等则返回正/负数

函数定义实现：

```
int cmp(const char* s1,const char* s2)
{
    int idx = 0;
    while(s1[idx]==s2[idx]&&s1[idx]!='\0')
    {
        idx++;
    }
    return s1[idx]-s2[idx];
}
//也可以使用指针进行遍历比较
int cmp(const char* s1,const char* s2)
{
    while(*s1==*s2 && *s1!='\0')
    {
        s1++;
        s2++;
    }
    return *s1-*s2;
}
```

strcpy()：将一个字符串的内容拷贝到另一个字符串内。

函数定义实现：

```
int cpy(char *dst,const char *src)
    //dst作为即将被赋值的指针变量肯定不可以是const类型
{
    while(src[idx] != '\0')
    {
        dst[idx]=src[idx];
        idx++;
    }
    dst[idx] = '\0';
    return dst;
}
//使用指针也可以，但与上面的strcmp有一些不同，因为需要返回*dst的首地址，而*dst++改变了dst的地址，所以需要再引入一个指针变量记录*dst的初始首地址：
int cpy(char *dst,const char *src)
```

```
{
    char *ret = dst;
    while( *src != '\0')
    {
        *dst++ = *src++;
    }
    *dst = '\0';
    return ret;
}
```

strcat(char *restrict dst,const char *restrict src):将一个字符串拷贝到另一个字符串的末尾之后形成一个新的长字符串。

```
int cat(char *dst, const char *src)
{
    char *ret = dst;
    while (*dst != '\0')
    {
        dst++;
    }
    while (*src != '\0')
    {
        *dst = *src;
        dst++;
        src++;
    }
    return ret;
}
//使用数组遍历一样可行
char mycat(char *dst, char *scr)//传递参数
{
    int i = 0;//定义变量
    int len = strlen(dst);//计算a的长度
    while(scr[i] != '\0')//循环的条件是当b的末尾不为0时
     {

        dst[len] = scr[i];//让a的末尾接上b的开头
        len++;
        i++;
     }
}
```

strcat和strcpy及其类似函数其实是危险的，因为在使用过程中并没有定义可用的最大字符串空间，故可能造成字符串溢出导致的越权访问。strncat/strncpy则解决了这个问题，在调用时规定了最大字符串空间。

strchr():在字符串中寻找单个字符

strstr():在字符串中寻找一个字符串

枚举

enum 类型名{num0,num1,.....,num n}

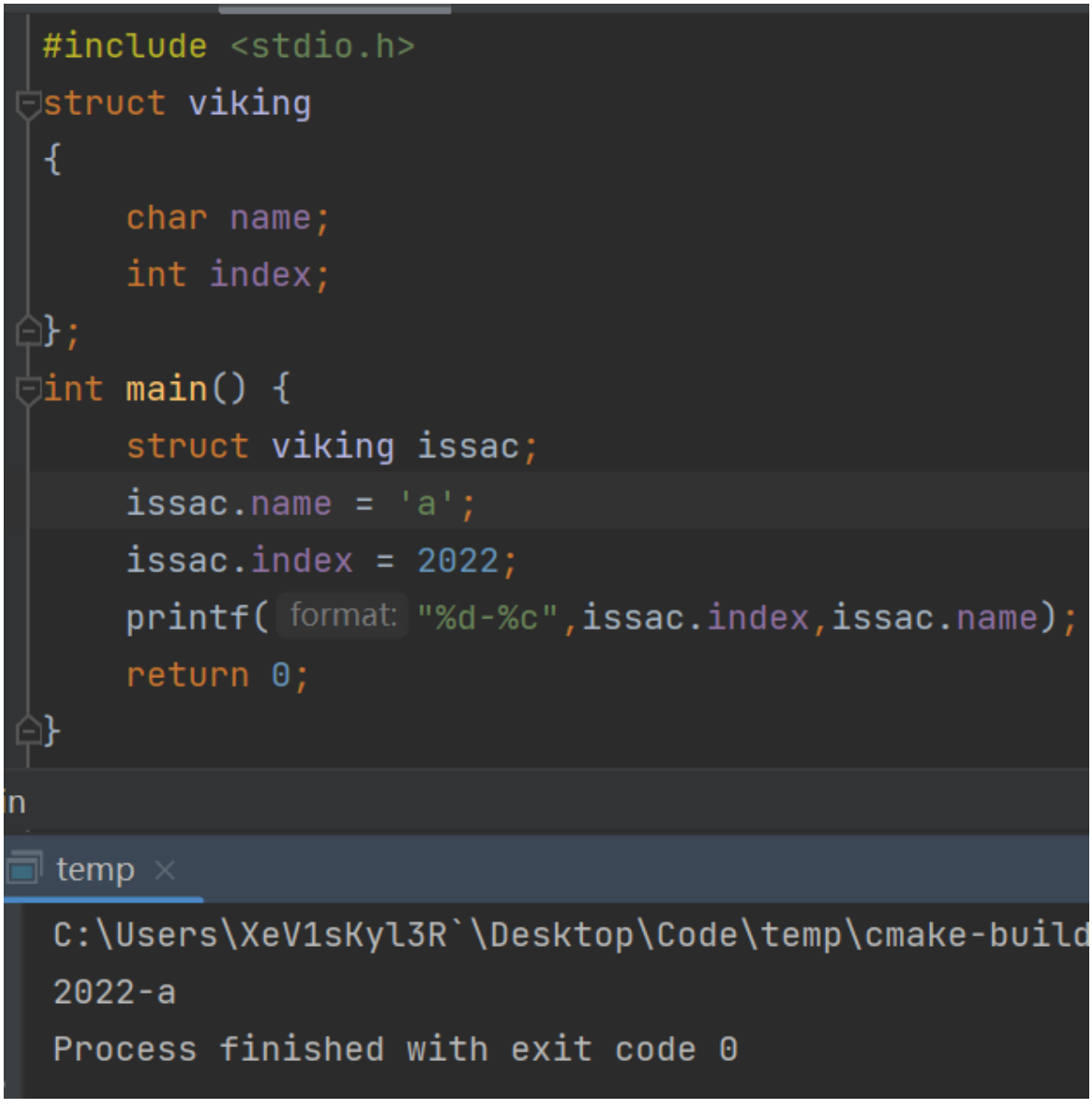
一般来说这个类型名不会被使用，需要使用的是大括号内的名字，而这些名字默认的数据类型为int，对应的值默认从0向后递增（但也可以通过num0 = k的方式定义第一个数据的值）可以将其理解为简化的一组连续值的宏定义。

*枚举的量可以作为值

*枚举类型可以加上enum关键字作为数据类型，但它实际上是以int（unsigned int)输入输出的。

结构体

结构体的声明和简单调用实例：




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

tag 是结构体标签。

member-list 是标准的变量定义，比如 int i; 或者 float f，或者其他有效的变量定义。

variable-list 结构变量，定义在结构的末尾，最后一个分号之前，您可以指定一个或多个结构变量。（也可以在其他函数里声明，则为局部变量，只在该函数内生效）

对结构变量的赋值：

```
variable-list1 = (struct struct-tag1){1,2,3}
```

和数组不同的是，结构变量的名字不是结构变量的地址，如果需要使用指针则必须使用&运算符。

```
struct struct-tag *p = &variable-list1;
```

结构体的输入

需要注意的是，结构体不能像数组一样直接指针对应地址，而是复制原有的结构体内容。

```
#include <stdio.h>
struct point
{
    int a;
    int b;
};
struct point getstruct();
void output(struct point);
int main()
{
    struct point y = {0,0};
    y = getstruct();
    output(y);
}
struct point getstruct()
{
    struct point p;
    scanf("%d",&p.a);
    scanf("%d",&p.b);
    return p;
}
void output(struct point p)
{
    printf("%d,%d",p.a,p.b);
}
```

当然，也可以直接定义一个指向结构变量的指针。

```
struct point
{
    int a;
    int b;
}pointer;
struct date *p = &pointer;
(*p).a = 1;
// p->a = 1;
```

通过这种方式可以对结构的传递进行优化：

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

```
struct point
{
    int a;
    int b;
}y;
struct point* getstruct(struct point*);
void output(struct point);
void sprint(const struct point *p);
int main()
{
    struct point y = {0,0};
    getstruct(&y);
    output(y);
    output(*getstruct(*y));
    sprint(getstruct(&y));
}
struct point* getstruct(struct point *p)
{

    scanf("%d",&p->a);
    scanf("%d",&p->b);
    return p;
}
void output(struct point p)
{
    printf("%d,%d",p.a,p.b);
}
void sprint(const struct point *p)
{
    printf("%d,%d",p->a,p->b);
}
```

结构体也可以是数组：

```
struct date dates[100];
struct date dates[] = {
    {1,2,3}{1,2,3}
}
```

共用体

union的定义方式和struct类似，但是其区别是，只定义了一个变量，但是该变量可以有不同的变量类型。共用体占用的内存应足够存储共用体中最大的成员。（typeof())

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

类型定义

```
typedef A B;
//A是原有的类型名字，B替代了A成为新的名字
```

除了使C原有的变量类型所指更加明确，这样做还可以使结构等本来复杂的类型简化：

```
typedef struct Adate{
    int month;
    int day;
    int year;
}Date;
Date d = {1,2,3}
```

预处理器和头文件

预处理器从本质上就是文本替换工具，例如：

```
#define A 50
//在编译之前定义宏A即可将代码中所有的A替换成50
```

使用比较多的预处理器有：

#define	定义宏
#include	包含一个源代码文件
#undef	取消已定义的宏
#ifdef	如果宏已经定义，则返回真
#ifndef	如果宏没有定义，则返回真
#if	如果给定条件为真，则编译下面代码
#else	#if 的替代方案
#elif	如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个 #if……#else 条件编译块
#error	当遇到标准错误时，输出错误消息
#pragma	使用标准化方法，向编译器发布特殊的命令到编译器中

一个很有意思的DEBUG嵌套：

```
#ifdef DEBUG
   /* Your debugging statements here */
#endif
//只有在debug被定义的时候才会执行debug语句
```

预处理器也有一些特殊的运算符：

"\"宏延续运算符，当一个宏语句太长，一行放不下的时候需要在第一行结尾使用过该运算符。

##运算符：标记粘贴，允许在宏定义中两个独立的标记合并为一个。

defined:用来确定一个标识符是否已经被定义（如果定义过则返回假，反之返回真）

宏定义本身也可以带参数来模拟函数。

头文件一般包含自定义函数的原型，在.c源文件中通过#include进行包含。函数的代码并不在头文件之中，#include能做到的只有让编译器知道函数的结构，保证调用时传入和传出的参数值类型正确。

示例：（该头文件可以保证在一个编译单元中只被包含一次）

```
#ifndef __LISTHEAD__
#define __LISTHEAD__

#include "node.h"

typedef struct _list
{
    Node* head;
    Node* tail;
}
#endif
```

文件读写

详见：https://www.runoob.com/cprogramming/c-file-io.html

位运算

位运算符：

```
若x==1,y==1,则(x&y) = 1,否则(x&y) = 0
1.让某些位为0: x & 0x()FE // FE(16)=1111 1110 （可以让最后一位成为0）
2.取一个数中的某一部分： x & 0x()FF //FF(16)=1111 1111 （可以让FF对应段不变）
```

```
若x==1或y==1,则(x|y)=1,否则(x|y)=0
1.使某些位为1： x|0x01
2.拼接数字：0x00FF|0xFF00
```

```
~x = 1 - x
把1变成0，0变成1
```

```
如果x == y,则(x^y)=0,反之(x^y)=1
```

```
i << j ：将i向左移动j个位置，在右边填充j个0
```

可变数组示例

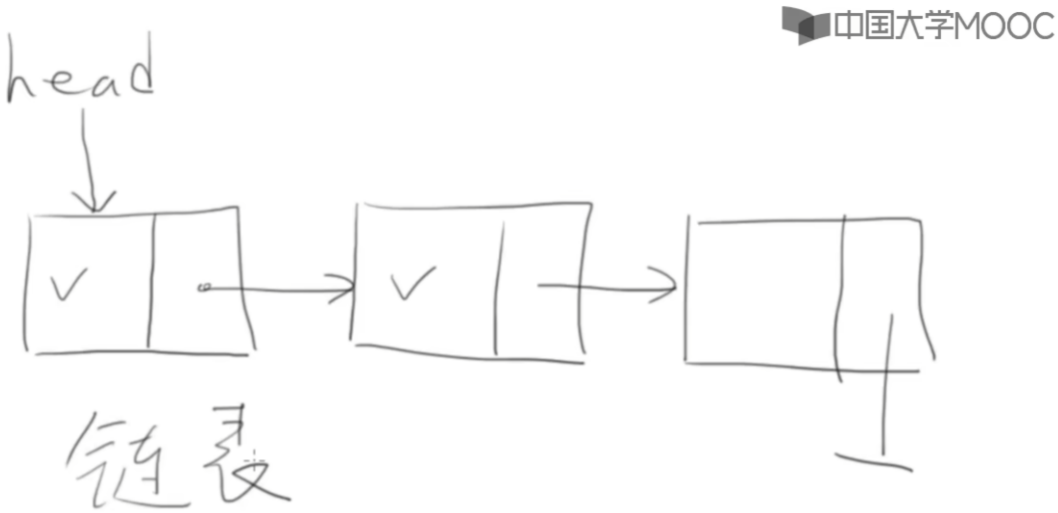
```
//array.h

#ifndef _ARRAY_H_
#define _ARRAY_H_
```

```
typedef struct{
    int *array;
    int size;
}Array;
void f()
{
    Array a
}
Array array_create(int int_size);
void array_free(Array *a);
int array_size(const Array *a);
int* array_at(Array *a,int index);
void array_inflate(Array *a, int more_size);

#endif

//array.c

Array array_create(int init_size);
{
    Array a;
    a.size = init_size;
    a.array = (int *)malloc(sizeof(int)*init_size);
    return a;
}
void array_free(Array *a)
{
    free(a->array);
    a->size = 0;
}
int array_size(const Array *a);
{
    return a->size;
}
int* array_at(Array *a,int index);
{
    if(index >= a->size)
    {
        array_inflate(a,index-a->size+1);
    }
    return &(a->array[index])
}
void array_inflate(Array *a, int more_size);
{
    int *p = (int*)malloc(sizeof(int)(a->size + more_size));
    int i;
    for(i=0;i<a->size;i++)
    {
        p[i] = a->array[i];
    }
    free(a->array);
    a->array = p;
    a->size += more_size;
int main(int argc ,char const *argv[])
{
    Array a = array_create(100);
    printf("%d\n",array_size(&a));
    *array_at(&a,0) = 10;
```

```
    printf("%d\n",*array_at(&a,0));
    int number;
    int cnt =0;
    while(number != -1)
    {
        scanf("%d",&number);
        if(number != -1)
        {
              *array_at(&a,cnt++) = number;
        }
        //scanf("%d",array_at(&a,cnt++));
    }
    array_free(&a);
}
```

链表

链表其实是一种特殊的结构体，每个单一结构体都包含一部分，其作用是以指针链接到下一个结构体上：




```
typedef struct _node{
    int value;
    struct _node *next;
}Node;
//链表的定义
Node * head = NULL;
int number;
do{
        scanf("%d", &number);
        if(number != -1){
            Node *p = (Node*)malloc(sizeof(Node));
            p->value = number;
            p->next = NULL;
            Node * last = head;
            if(last){
                while(last->next){
                    last = last->next;
                }
                last->next = p;
```

```
            }
            else{
                head = p;
            }
        }
    }while(number!=-1);
//向链表内添加新的元素
```

文档：https://blog.csdn.net/qq_61672347/article/details/125701955