1 数组

首先，数组是一连串同类型对象的集合，存储在**一整块固定大小连续的内存**中，在编译后的代码没有数组一说，就是一块连续的内存.

1.1 一维数组 (single-dimension arrays)

声明:

```
char p[10];
type name[size];
```

声明时中括号内是元素个数，但是使用时索引从 0 开始.C/C++ 没有数组边界检查，如果时间超出边界可能会写到别的变量甚至程序的内存区域.

一维数组的内存大小：每个元素的字节大小乘元素个数.

total_bytes = sizeof (base_type) * size

1.2 指向数组的指针

首先，虽然指针和数组关系很近，但是，**指针不等于数组!**

数组**在需要值的表达式中**有**退化**行为，单独使用数组名不加索引时，编译器会把他当作数组中第一个元素的指针来用. a == &a[0] 但是这仅是数组名退化，数组的本体还在.

三种永远不会退化的行为（保持数组的身份）:

(1) **sizeof (a)** sizeof 的 a 本身，所以大小是数组长度.

(2) **取地址 &a** 求的是数组 a 的指针，值和 &a[0] 相同，应该用 **int (*)[5]** 类型的指针接收

(3) 作为定义本身

本质上，数组下标索引是通过指针运算规则实现的.

```
int a[5];
a[3];
*(a+3);等价

x[y] == y[x] == *(y+x);最最最本质
a[3] = 3[a];
```

因为 a 先退化成 int * 类型（也就是首元素的指针），再 a 加 3（指针 +3 是值加 3 个指针类型长度），最后 *(a+3) 求该指针的指向对象的值.

而 &a 是取数组 a 的地址，而不是 &&a[0]，因为 &a[0] 是首个元素的指针，再取地址就变成指针的指针了，没意义，而且 & 运算符作用于对象，a 就是对象了不需要退化. 因此，&a 类型是 int (*)[5]，自增一次加整个数组的长度. **需要注意的是，地址是不能被赋值的，只有指针变量能被赋值. 所有指针的大小都相同，指针类型不同是指向对象不同. 所以 a 和 &a 只是值相同**

我们把单独数组名 a 就叫做 a，而把 &a 叫做指向数组的指针，或者数组指针（注意指针数组是指针类型构成的数组）.

1.3 传递一维数组进函数

首先，不可能把整个数组值传递进函数，因为太慢了. 所以传递数组进函数就是传递数组第一个元素的指针进函数. **指针那块细说，声明参数是本质是声明一个新的指针变量，所以传递 &a[0]/a 给这个指针变量. 因此函数内部对这个指针变量的自增不会改变外面，但是若对指向的对象操作就会改变外面**.

三种声明函数参数为数组的方式:

```
1  void fun (int a[10]);//1.有长度数组
2  void fun (int a[]);//2.无长度数组
3  void fun (int *a);3.指针
```

因为 C/C++ 没有边界检查，所以声明参数时随便写数组长度也可以，但是因为本质都是传递指针进入，所以写 3 最常见.

调用时直接传递指针即可，就是数组首元素指针，写数组名.

1.4 以空字符结尾的字符串

空字符 = NULL = 0 = '\0' = 0x00

在很多头文件中都有宏定义 NULL

其实有好多种字符串，但是 C 语言只有以空字符结尾的字符串. **字符串就是类型为 char 的数组**，但是因为以空字符结尾，所以声明长度时要多一个元素.

字符串常量需要用双引号括起来，而在使用（例如初始化）的时候不需要结尾加一个\0，因为编译器会自动加上.

重点：几乎所有字符串处理函数，都是传递给函数首元素指针，并处理到空字符停止.

初始化方式:

```
1  char str1[10] = "Hello";//1.是字符串变量/字符数组
2  char *str2 = "Hello";//2.是指向字符变量的指针
3  char str3 = {'h','e','l','l','o','\0'};//3
4  str1 = "ABC";//str1根本不能作为左值被赋值，也别提退不退化了!
5  str2 = "ABC";//str2本身就是指向字符类型的指针变量，可以被赋值
```

先说第三种，就是类似数组初始化的方式，不推荐，因为后面得手动加上'\0'

再说一下，**"string" 是一个字符串常量，就相当于 str[7]，所以直接用字符串就相当于用 str（退化/不退化）. 它也在内存中有自己的位置，也可以访问，但是不能更改.** 前两种（重要!），第一种声明了一个字符串变量（字符数组）直接把字符串写在栈上的一块大小固定的连续内存，基本上所有操作都可以进行. 第二种声明了一个字符指针变量，而"Hello" **被放到了字符串表 (string table)**，str2 只是指向这个字符串的首元素.

注意，修改字符串表的常量是未定义行为，程序直接崩溃. 而使用 sizeof 对 str1 是字符串长度，对 str2 是指针变量的长度 4/8. **没有区别的点是都可以传递进字符串处理函数 but 见下文**.

非常重要！承接上文，char *str2 只声明了一个指向字符变量的指针变量，而 char str1[10] 是分配了一块连续的大小为 10 个 char 的可以使用内存！如果想用 gets/scanf 读取存入 str2，正常来说是不可以的，因为 **(1) 如果不对 str2 初始化，char *str2 是野指针！还没有给他赋值，所以不能直接用 scanf 和 gets，因为不知道这个指针是指向哪里的，而再自增后又不知道破坏了哪个地址存**

储的值.(2) 如果对 str2 初始化了，str2 存的地址是只读区的，更不能用 scanf 和 gets 来读取改变字符串表的对象了.

给字符串赋值的的方式:

1. 初始化时大括号赋值：声明成字符数组或者字符指针都可以.
2. 初始化后赋值：字符数组 str1 不可以再被一个字符串常量赋值，因为在这里不退化，还是保持数组的身份. 而字符指针 str2 可以被赋值，因为赋的是字符串常量首元素的指针.
3. strcpy 赋值: 可以直接把一个字符串赋值进字符串变量, 也可以给字符指针赋值（可以个屁, 要是野指针直接写入别的内存了, **所以用字符串函数处理也慎重**）.

1.5 二维数组

首先，二维数组不是“二维”，而是每个元素都是数组的数组. 其在内存中表示可以看成一个矩阵方便理解，但是本质还是一**块大小固定的连续内存**. 左索引代表行，右索引代表列，真实内存表示是**一行连接着下一行的线性内存, 所以列索引变化比行索引快，因为是连续的挨着近**.

把每个元素看成一个一维数组，再看退化后与指针的关系.

1. **int a[3][4];** 二维数组，是一个有 3 个元素的数组，每个算是都是一个长度为 4 的 int 类型数组.
2. **a** 不退化时类型是 int [3][4]，退化后（表达式中）类型是第一个元素的指针，第一个元素的类型是一个长度为 4 的一维 int 数组，所以 a 的类型是 int (*)[4];
3. **a[0]** 不退化时是一个一维数组（二维数组的第一行），类型是 int [4]，退化后（表达式中）类型是第一个元素的指针，第一个元素类型是 int，所以 a[0] 类型是 int *
4. **&a[0][0]** 显然是二维数组第一个元素（一维数组）的一个元素（整型）的指针，类型是 int * 等价于退化后的 a[0].
5. **&a** 是 a 这个数组（二维数组）的指针，类型是 int (*)[3][4];

访问数组元素时编译器的本质操作，下面仍旧有那个神级操作，也符合小白老师的秘诀：**一个 [ ] 等价于一个 ***

```
int a[3][4];
a[i][j] == *(*(a+i)+j);等价

a[i][j] == i[a][j] == *(*(a+i)+j) == *(*(i+a)+j)
```

还是跟一维数组一样，拆解，a 先退化为 int (*)[4] 类型，也就是第一行的指针，然后偏移 i（**指针运算跨 4 个 int**），变成了第 i 行的指针，加 * 变成这个指针指向的对象，也就是第 i 行. 然后再退化，变成 int * 类型，也就是第 i 行的首元素的指针，偏移 j（**指针运算跨 1 个 int**），变成第 i 行第 j 列的指针. 最后加 * 变成第 i 行第 j 列的值.(**全是指针运算，数组不存在了 (doge)**)

内存大小：total_bytes = sizeof (base_type) * 行数 * 列数（没啥可说的）

二维数组作为参数传递进函数，就是第一行的指针传递进函数. 为什么不能把第一行的第一个元素的指针传递进去呢，因为这样的话就没法在函数里二维索引了（麻烦），就真把内存看成线性去操作了. 调用的话就是用第一行的指针，下面两种都可以.

三中声明方式:

```
int a[3][4];
void fun (int a[3][4]);1.闲的
void fun (int a[][4]);2.也行
```

```
    void fun (int (*a)[4]);3.最常用

    fun (a);
    fun (&a[0]);都可以
```

因为本质就是传递一行的指针，所以你即使跟函数说有几行也没用，因此可以省去行数，最本质也最常用的方式就是声明一个指向一行的指针，因为我们说过编译器视角下数组不存在，所以在函数内使用也可以索引. 至于为什么要明确每行有多少元素，是因为如果不告诉函数长度是多少，就无法通过索引找到下一行了.

1.6 指针索引

如果指针变量指向数组的元素，那指针变量就可以被索引.

```
    int a[5];
    int *p=a;
    *(a+3) == *(p+3);
    a[3] == p[3];

    int a[10][10];
    a[i][j];//直接索引
    *(*(a+i)+j);//最本质
    (*(a+i))[j];//*(a+i)是第i行，退化为第i行第一个元素的指针，然后通过指针索引，就是第i行第j列的元素.
    *(a[i]+j);//a[i]是第i行，退化为第i行第一个元素的指针，偏移j个指针，得到第i行第j列元素的指针，*求对象.
    *((int *)a+i*10+j);a是数组退化为第一行的指针，强制类型转换为指向int的指针，就是指向第一个元素，因为线性内存，偏移行号*列数+列号个指针，再求值.
    int a[m][n];
    a[i][j] == *((int *)a + i*n+ j);
    //好ex!!!
```

根据上述最最最本质的公式 x[y]=*(x+y)，得出来指针 p 后面加上索引就是就是数组 a 中第三个元素，二维数组同理.**p 等价于 &p[0]**

如果顺序遍历一个数组的话指针自增变化比索引快的多，但是随便访问时速度都差不多.

数组索引就算基指针 + 偏移. 小白老师秘诀：一个 * 等价与一个 [ ]

1.7 多维数组 (multidimensional arrays)

还是一样，只有一块大小固定的连续内存，可以理解为多维. 但是不常用，因为计算索引的时候太慢了. 别的性质和数组都一样，声明为形参时就别写指针形式了（太麻烦），写成数组形式，第一维数（最左边的）可以不写.

1.8 动态内存分配数组（allocated array）

先看一个例子：

```
    int n=10;
```

```
    int arr[n];//老的编译器会报错,c99以后不会

    char *a;
    a = malloc (n);
    for (int i=0; i<n; i++){
       a[i]='6';//之前说的,本质是*(a+i), 所以就直接当作数组用了.
    }
    int *p = malloc (n*sizeof (long));
```

经常会有不知道数组明确大小但是也得声明数组的情况，这就得引入两个函数.

malloc(memory allocation) 会分配长度为要求**字节**的可用连续内存块，并且返回内存块首字节的地址（返回值为通用指针）. 所以这个内存块可以当作数组来用（用一个指针变量来接收地址，类型自己决定）. 因为单位是字节，所以要用什么数组记得乘上一个 sizeof(类型).

free() 用来释放 alloc 分配的动态内存,参数为内存块的首元素指针. 其实小程序不用也可以,因为程序结束的时候操作系统会默认释放分配的动态内存. 如果随便写了个指针给 free() 可能堆 (heap) 就崩了，然后程序崩了.

函数原型：void *malloc (size_t num_bytes);

void free (void *p);

两个函数都在 stdlib 库中，size_t 是一个无符号整型，在 stdlib 中定义，能够存储足够大的无符号整数. 如果内存不足无法分配，malloc 会返回 NULL, **因此，建议在分配后检验一下以免出错.**

```
    #include <stdlib.h>
    int *p;
    if ((p=(int *)malloc(100))==NULL){
       printf ("Out of memory.\n");
       exit (EXIT_FAILTURE);
    }//(int *)的cast可以不写
```

1.9 初始化数组

对于 C89 数组大小必须是常量，但是 C99 以后就可以用变量了. 声明后面接等号，然后中括号内是用逗号分隔的变量. **未初始化的数组中存的是垃圾值，若只初始化部分元素，则其余元素默认为 NULL**. 对于多维数组可以把每个维度的元素再用中括号括起来嵌套，叫做**聚合初始化 (aggregate initialization)**.

声明时若初始化，一维数组可以不写大小，二维数组可以不写行数，多维数组可以不写第一维数（最左边），编译器会自动分配.

sizeof 字符串时算上 NULL，因为求的是大小 (bytes)，而 strlen 时不算 NULL，因为求的是长度.

2 指针

2.1 基本认知

首先, 这里内存中以 byte 为单位, 每个 byte 有自己的编号, 编号大小通常是 4/8bytes（32 位/64 位编译器），因此，**地址是内存中 byte 的编号**. 而 C 语言中的对象都是存在内存中，根据类型决定

用几个 bytes，所说的指针是一个变量，他指向一个对象（变量、数组、函数指针、结构等），因此，**指针 = 地址 + 类型解释，也就是说知道一个指针变量，就知道了一个对象的类型和地址**. 需要注意的是，指针变量的值是该对象所在的最小的地址.**C 语言是不能操作裸地址的**，所以地址不能被改变、赋值、运算，而指针可以，所以对一切内存的访问都是通过指针.

2.2 声明指针变量

声明指针变量就是正常声明变量的格式再名字面前加 *，注意，**这个 * 作用于这个对象（对象名），而非作用于变量类型**. 正因如此**声明指针变量时 * 后不能用逗号分开共享前面的 *，但是如果用 typedef 就可以了，因为定义了一个类型叫 xx 指针**. 指针的类型就是它指针所指向对象的类型，这个信息同时储存在指针变量中. 正如前面所说，指针的大小是固定的（sizeof 出来），因为他的值只是指向的对象的第一个地址，而非全部地址. type *name;

2.3 指针操作

先说两个指针运算符（一元）:

- & 作用于对象，通常叫做取地址. 但很重要的是，C 语言本身不对裸地址操作，所以应该是**生成一个“指向对象的指针”，&x 返回的是 x 的指针！（不是地址）更深度一点就可以说成是把一个对象映射成指向这个对象的指针**.
- * 作用于指针变量，将一个指针映射成这个指针指向的对象. 通常叫做取这个地址存储的值，但本质不是取值，而是**按指针的类型解释内存，是读/写 sieof(T) 个 bytes**.

在说赋值前先说一下**通用指针 (void *)**,void 在函数中就是表示“无类型”，而在指针中也是如此，void * 就是声明一个不携带类型信息的指针，只有一个值，只知道内存从哪开始. **但是不可以用 void 声明一个变量，只有这 3 种用法**. 通用指针的大小和其他指针一样由编译器决定. 通用指针可以接收任何任何对象的地址（不需要 cast），因为所有对象指针都隐式转换成 void *，也可以把通用指针赋值给任何指针变量. 但是使用前要 cast 一下不然光对这地址什么也干不了. 同时，因为通用指针不携带类型信息，所以不能对它 * 和运算.

现在说指针赋值:

- 合法：同类型相互赋值，任何类型赋值给通用指针，通用指针赋值给任何类型.
- 不合法：不同类型的指针之间，指向的对象是 const 不能被赋值给比人（就是不能 &const 变量）.

然后说指针运算，指针只有两种运算:

1. **指针 ± 整数:** 根据指针基类型来运算，每变化一个单位，指针值就变化基类型大小个 byte，按所指对象大小变化. 加就是向大编号内存偏移，减就是向小编号内存偏移.

```
        p + n * sizeof(*p);//p+n的数值变化

        int a = 10;
        int *p = &a;
        p + 3;//p + 3 * sizeof(*p)
              //p + 3 * siezof(int)
              //p + 3 * 4;

        int a[5];
```

```
        int * + 1;//跳4个bytes, 1个int
        int (*)[5] + 1;//跳5个int
```

2. **两个指针相减：**同类型/类型兼容才能相减，但是说句实话，**两个指针指向同一数组中的元素，相减才有意义.** 其余虽然都合法但是未定义行为没用（因为只有数组是一块连续内存，其余单独声明的时候都是分散的），**结果返回两个指针之间差几个元素，而不是 bytes.** 数值上也就是指针值（地址）相减再除基类型大小.

指针比较也只在两个指针指向同一个数组中元素时才有意义，原因同指针减法，一个指针比另一个大表示它的值（地址/内存的编号）大，数组后面的元素地址大于前面的元素.

2.4 常量指针

之前说过 const 修饰符表示一个变量不能被修改，除了初始化时/硬件操作. 这里我们说到**当 const 修饰符用在声明函数参数为指针时，可以避免函数内部操作改变外部该指针指向的对象**. 如果写了 const，函数内也有改变外部对象操作的代码，编译会超时. **很多标准库函数处理都 const 了指针变量，但是注意 scanf 没有 const，因为要是 const 了还咋读取?**

2.5 指向字符串的指针

说实话前面在数组那块都说的差不多了,这里就再强调几点,大部分字符串处理函数参数是 const char *str，接收字符串指针，并且是常量不改变函数外部的字符串. 同时记得如果顺序遍历数组，指针自增快得对，还有字符串都是以 NULL 结尾的，因此可以将循环条件写成字符串中的每个元素，遇到 NULL 结束.

```
int str[100];
while (*str){
   str++;
   .
   .
   .
}
```

2.6 指针数组

因为数组是相同类型对象的集合，指针是变量，也是对象，所以可以作为数组的元素. 声明时建议加上括号，虽然 [ ] 优先级本来就比 * 高，但是可以让可读性高一点.

```
int *x[10];
int *(x[10]);

void fun (int *x[10]);
void fun (int *x[]);
void fun (int **x);
```

声明方式还是 3 种都性，这个写成数组形式可能会清楚一点.

实用层面可在创建一个指针数组，每个指针都指向一个字符串，用来存储字符串信息. 其实 argv 就是这样的，只是参数传递时数组退化成首元素的指针，就是指针的指针.

```
    char *str[] = {
       "Hello",
       "I am",
       "Jupiter Traveller",
       "Who are you"
    };
```

2.7 指针的指针 (Multiple Indirection)

就是字面意思，因为指针是一个变量，所以也存储在内存中，有自己的地址，注意，指针的指针指向的对象是指针，也就是说它的类型是它指向的指针，而非指向的指针的类型

```
    int x;
    int *p = &x;
    int **a = &p;指向的类型是整形指针
    int *(*a);首先a是个指针，指向一个整形指针
    printf ("%d", **a);
```

同时，想访问 x 的值需要用两次 * 来到 a 指向的对象指向的对象 x.

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

    int a = 10;
    int b = 20;
    &a = &b;
```

好复杂啊啊啊，那就一次整明白吧：

1. 先想想什么是赋值，右侧是一个表达式，左侧是一个**对象，没错是对象**, 那就很明白了，&a 是一个表达式，一个临时出现的值，不是一个对象，就更别说被赋值了.
2. 本来想直接取 a 的地址的地址，但是不对啊. 两种思路：(1)&& 是逻辑中的二元操作符与运算，这么理解 &&a 只有一个运算对象，肯定错了.(2) 那我认为是 &(&a) 不行吗，对还是不行，**因为上面说了 & 生成的不是对象更不是变量，内存中都没有地储存他更没法再取地址了.**

总的来说，& 作用于对象，而结果不是对象. 其实 && 取的是控制流的入口地址（是 gcc 的扩展不是标准 C 先不用看）.

2.8 初始化指针

很重要！很重要！很重要！一定要初始化不要忘了.

```
   int *p;
   *p=10;
   printf ("%d", *p);

   int *p=NULL;
   *p=114514;
```

先介绍两种指针：

1. 野指针 (wild pointer)：未初始化，变量 p 存储的地址为随机地址.windows 系统中一些地址不能被用户访问，只能 windows 自己访问，若用户想要访问，windows 捕获信息，停止程序。如果在该地址写入东西了，那程序甚至操作系统都有可能崩了. 解决方案：**在引用 *p 前，先对 p 本身赋值.**
2. 空指针 (null pointer)：若指针变量 p=NULL(0)，则称 p 是空指针，空指针就是 0 地址.**C 语言规定，0 地址处的内存单元不能存放任何类型的变量**. 因此空指针用来表达一个不存在对象的地址. 同样如果对空指针指向的对象赋值可能会崩. **但是用空指针在程序里有很多便利，比如标志指针数组结束，可在遍历时直接以数组元素为条件.**

2.8.1 函数的指针

函数也是一段程序, 是一段可执行代码, 所以**不是对象**. 但函数在内存中是有自己的物理位置的, 所以函数也有指针, 在编译时，函数会有一个入口点被编译出，运行时调用就是进入这个地址，因此函数的指针就是这个函数入口点的内存地址.

声明函数指针

```
int add (int a, int b);
int (*p) (int , int ) = add;
add == &add;
*add == add;
```

首先 p 是指针，然后 p 指向一个函数，函数的返回类型为 int，参数情况是有两个 int.（***p 不能没有括号，因为括号优先级比 * 高，否则就变成声明返回指针的函数了.**）用函数名当表达式会立刻变成函数的指针，**这不是退化，而是类型转换，因为函数不是对象!!! 是可执行实体.** 所以单独用函数名，加 *，加 & 全都是函数的指针. 总结一下声明，**p 是一个指针，然后指向一个签名是这样的函数**. 还得注意一下函数指针不能运算哦～

用函数指针调用函数：

```
int add (int a, int b);
int (*p) (int , int );
p = add;//p == &add;
add (1, 1);
(&add) (4, 5);
p (1, 4);
(*p) (6, 6);
```

全都可以，调用随便到已经不知天地为何物了.

声明函数指针为参数：

```
int comapre (int a, int b);
void sort (int *arr, int n, int (*compare) (int , int ));
```

可以在使用前写作为参数的函数的原型. **函数指针的形参名不重要，只要签名对的上就行**.

调用参数为函数指针的函数, 记住函数指针本质就是指针, 所以直接把指针传递给函数即可, 就是函数名直接传递进去就行.

易错点：

1. 函数指针赋值时类型不一样也能编译成功. 但是，这完全用不了，所以函数指针赋值必须签名完全匹配.
2. 不用函数指针也可以在函数内调用其他函数, 这是显然的, 但是这个是**静态调用/直接调用**, 在编译期就已经确定，后续运行不会改变，是写流程. 而用函数指针是在运行期决定调用谁，意思就是反正我能给目标函数传递一个这样签名的函数指针，甭管传递谁，是写框架.

常见用法:

1. **回调函数：**把行为当参数传递进函数，外函数的行为不再拘泥于一种，而是用户决定用谁.
2. **策略：**实现算法时要求规则不同，改一个特别大的函数太麻烦.
3. **函数指针数组（驱动）：**首先函数指针是指针变量，可以存在存在指针数组，因此可以由函数指针组成的数组. **可代替 switch，常见与数据库，实现一个函数数组，根据编号执行操作.**

```
void enter(void), del(void), review(void), quit(void);
void (*options[]) (void) = {
   enter;
   del;
   review;
   quit;
};
```

函数指针数组举例, 先声明了 4 个同签名函数，然后注意下面. 首先 options 是数组每个元素都是指针，指针指向这样签名的函数.

3 一些其他类型

3.1 结构 (structure)

```
struct st{
   int mun;
   int score;
};

struct st a, b, c[4];
struct st{
   int num;
   int score;
} a, b, c[4];
a.num = 1;
a.score =90;
b=a;//相当于下面两句话
b.num = a.num;
b.score = a.score;
```

结构就是把不同类型的对象，按顺序打包成一个新的整体对象.struct 是关键词，st 是标签名，struct st 是结构类型名，num 和 score 是结构的成员 (member)，以后声明的每个 st 结构里都有两个 int. 所以最开始是定义了结构的类型 (模板)，还没声明变量分配内存，而后再用结构类型名声明变量.

结构类型定义在主函数外或内都可以定义，在外定义是全局都可使用，定义在函数内只能该函数使用. 声明结构变量时也可以跟在结构类型定义后立刻进行，同时注意，如果只用一个结构类型只声明一个结构，可以不写结构类型名，只在最后写上一个变量名.

从内存视角来看,首先结构是一个对象,所以会分配一块大小固定的连续内存,结构中的成员按顺序储存在其中. 但是，在不同的计算机上**中间会有不同的对齐填充 (padding)，相同种类个数成员若顺序不同可能影响大小.** 所以**总内存大小大于各成员类型内存大小之和.** 即 sizeof(struct st)>=4+4. 因此为了提高程序可移植性，想要知道某个结构的内存大小，最好用 sizeof 来求.

操作符. 用于访问结构中的成员（**优先级最高，同括号**），说成人是“的”，a 中的 num. **相同类型的结构可以直接相互赋值**, 即各结构内各元素分别相互赋值.C 语言中结构可以像数组一样初始化.

```
struct st{
   char name[10];
   int score;
} ;
struct st a = {"Tom", 90}, b={"Jerry", 90}, c[4];
struct st *p;
p = &a;
p = c;//等价p=&c[0];

&c[i].name[0] == c[i].name != &c[i].name;//c是结构数组，前两个是c[i]中字符数组
    name的首元素的指针，后一个是c[i]中字符数组name的指针.
(*(p+i)).name;
&(*(p+i)).score;
(*p).name;
&(*p).score;

p->score == &c[0]->score == c->score;
```

p 是指向结构的指针. **操作符->（优先级第一）**同样用于访问结构体中的变量，但是是通过结构指针直接访问结构中的元素. **总之，点左侧是结构变量，箭头左侧是结构指针，右侧都是结构中的元素.p-> 等价于 (*a).**(括号右边有个点)

排名第二，&，*，++，– 右结合 =, ?:

```
struct st {
   int a;
   float b;
   char str[10];
} x, y;
fun (x.a);
fun (&x.a);//
fun (x.b);
fun (&x.b);//
fun (x.str);
fun (x.str[2]);
fun (&x.str[2]);//

void fun (struct st aaa);
fun (x);//在函数内部用x名字就变成了aaa
```

```
16
17      void fun (struct st *bbb);
18      fun (&b);//在函数内部用就变成了bbb;
```

传递结构进入函数:

- **单独传递成员:** 和之前说过的传递对象每区别，可以值传递也可以指针传递.
- **值传递整个结构:** 首先声明一个结构类型 (最好全局声明)，然后函数参数声明这个结构（只用写出结构类型即可，因为全局都知道这个类型是什么），又因为结构是一个对象，调用时直接把想传递的结构传递进就行.（注意是值传递，just copy a struct）
- **指针传递整个结构:** 复习一下结构是对象，所以可以有指针，指针指向一个结构，类型就是这个结构类型，值为最小的地址. 那再说到结构的指针传递就没什么了，传递进取后可能改变这个结构中成员的值.

由最开始说的结构是把一些对象按顺序打包成一个新的对象，所以数组，二维数组，指针等等都可以是结构的成员. **因为结构也是对象，所以结构也可以作为结构的成员**. 访问成员方式如下，从外到内.

```
1       struct st {
2          char name[20];
3          int score;
4       }
5       struct emp {
6          struct st a;
7          float money;
8       } x;
9       x.a.score = 60;
10      strcpy (x.a.name, "Jupiter");//访问结构中的结构中的成员.
```

3.2 位域 (bit-field)

```
1       struct st {
2          unsigned a : 1;
3          unsigned   : 0;
4          unsigned b : 2;
5          unsigned   : 3;
6          unsigned c : 1;
7       }
```

位域是结构中的一种成员类型，不能在外面用的类型. 会在一个 unsigned int (4 bytes) **按照 bit 来分配内存**. 在内存中排列不一定紧贴着，比如，两个变量大小加起来超过 4bytes 了，第二个就会在新的 4bytes 中开始，前面剩下的补齐. 若类型不同 (short/long) 肯定分开存, 还有遇到分配 0bit 就是主动分开存.

因为位域是 bit 级别的，所以**不能对它取地址，不能组成数组，且不能跨平台，每个电脑都不一定一样.** 最后就是位域可以可以和其他对象混合声明在结构中.

3.3 联合 (union)

只有一块大小固定的连续内存，多个成员共享这一块. 所有成员起始地址相同，内存大小就是成员类型最大的类型长度，只是每个成员使用的长度不同. 同一时刻只能有效使用其中一个成员，因为他们公用内存.

声明方式，操作符，指针，传递均与结构相同.

```
union pw {
   int i;
   char ch[4];
} ;
unoin pw aaa;
aaa.i = 0x00004161;
printf ("%u %c", aaa.i, aaa.ch[2]);//输出: 16737 A
//正好验证小端规则.
```

3.4 枚举 (enumeration)

枚举定义了一组整型常量，非常类型与 #define 宏定义（几乎一样），并且引入了一个枚举类型，**该类型的值用某种整型表示!** 这一组整形常量中每个对象都叫做一个枚举常量，都是一个 int，而编译器也会为那个枚举变量选一个底层整形类型（通常是 int）. 因此 **enum 枚举是一个弱类型，就相当于不是一个新的类型，可以和 int 混用.**

```
enum color {
   RED,
   BLUE,
   YELLOW
} ;
enum color c;
```

声明枚举类型时，**枚举常量用逗号分开**. 如果不初始化每个枚举常量会默认按照 0,1,2,... 从小到大赋值. 也可以初始化一部分，没初始化的就仍按规则：第一个枚举常量如果没初始化就从 0 开始，如果一个枚举常量初始化了，那直到下一个初始化的枚举常量前之间的枚举常量都是从那个数开始一次 +1.

刚才说了枚举变量可以和 int 相互赋值，但是不同类型的枚举变量之间相互赋值会报错. 同时枚举常量有作用域，也不完全像、、#define 一样文本替换.

3.5 用户定义类 typedef

正常声明一个变量的格式，如果前面加上 typedef，那么就声明了一个类型名字（**不是创建新类型，只是给已有的变量起个别名**）

```
int a=10;
typedef int A;
A a=10;//等价 int a;

int *p;
typedef int *p;
```

```
    p p1=&a, p2=&b;//等价 int *p1, *p2;
    //int *name 中int和*是结合的, 所以用typedef定义了p为声明整型指针变量, 后面不用
        每个写*了.

    int arr[10];
    typedef arr[10];
    arr a, b;//a和b都是长度为10的整型数组.等价int a[10], b[10];

    char (*p)[4];//p是指向字符串的指针
    typedef char (*Pch)[4];//将Pch定义为指针类型, 指向大小为4的字符串
    Pch p;//p是指针, 指向大小为4的字符串

    int *p[4];//p是数组, 元素是指向int类型的指针
    typedef int *Pint[4];
    Pint p;//声明了一个数组, 每个元素都是指向int类型的指针

    double (*fp) (double , double );
    typedef int (*fp) (double , double );
    fp fun=pow;//等价double (*fp) (double, double )=pow;
    //函数不是对象, typedef不能作用于函数.
```

typedef 不是和 #define 一样纯文本替换的宏定义

```
    #define pint int*;
    pint a, b;//a是int *, b是int.因为宏是单纯文本带入;

    typedef int* pint;
    pint a, b;//a, b都是int *
```

3.6 链表

```
#include <stdlib>
struct st{
   int num;
   int score;
   struct st *next;//struct st *prev;指向上一个节点.
};

main ()
{
   struct st *p, *q, *r;//*p, *q, *r是链表的三个节点(node)
   p = malloc (sizeof(struct st));
   p->num = 1;
   p->score = 90;

   q = malloc (sizeof(struct st));
   q->num = 2;
   q->score = 80;
```

```
18
19    r = malloc (sizeof(struct st));
20    r->num = 3;
21    r->score = 70;
22
23    p->next = q;//p中的指针指向q
24    q->next = r;//q中的指针指向r
25    r->next = NULL;//r中的指向为NULL, 表示链接结束.
26
27    p->next->next;//通过p直接找到r,之前说过的嵌套
28 }
```

不知道数组大小的情况下，还非得储存一样大小的数据. 用链表.

看见第 5 行第一反应，我去这不递归吗，但是仔细想想，这个结构由三个部分组成，其中一个部分是他自己（???），似乎不可能，**所以其中一个成员是指向自己类型的指针，4 字节，可以分配内存**，这就合理了. 然后往结构里面放数据，都放满了后还剩一个指向自己类型的指针放什么，先不着急，注意到这个数据储存和数组不一样，**他的内存不是连续的，而是一块一块分开的.** 那怎么检索呢，这个时候就应该用那个指向自己类型的指针来指向下一个结构，每个结构中都有指向下一个结构中的指针，这样就让所有内存连在一起了，只能通过上一个找个下一个. 这就是**单向链表**，要想要双向链表可以在结构中加入一个成员指向上一个节点，想要循环链表可以让最后一个节点指向第一个节点.

上述 *p,*q,*r 称为链表中的节点 (node)

3.6.1 创建链表

可以写一个 create 函数用来创建链表，无参数，返回首节点的指针. 先声明三个结构指针都赋值为 NULL（不要出现野指针），然后开始循环，先分配一块内存，把指针赋给 pnew（当前最新的），随后输入该节点信息，同时把该节点德 next 指针赋为 NULL（还不知道指向德下个节点是谁）. **而后开始判断这个是头/尾/中间节点?** 先写一个 if 根据自己的标准判断是否为这个节点没意义表示结束, 上个节点是最后一个节点, 若是释放内存, pnew 赋为 NULL（一会有用）. 再写一个 if 判断是否为头节点（phead 是否为 NULL），(1) 若是则把当前 pnew 赋给 phead，这样后面 phead 就一直不是 NULL 了，(2) 若不是，则 pfore 是上一个节点的指针，把上一个节点中的 next 指针赋为 pnew，就是上一个节点指向当前节点. 最后把 pfore 赋值 pnew，现在这个节点的指针被赋给 pfore 在下一次循环中充当下一个节点的指针. 循环条件为 pnew 不是 NULL.

```
1  struct st *create (void)
2  {
3     struct st *phead, *pnew, *pfore;
4     phead=pnew=pfore=NULL;
5     do{
6        pnew=malloc (sizeof (struct st));
7       scanf ("%d %d", &pnew->num, &pnew->score);
8       pnew->next=NULL;
9       if (pnew->num==0){
10         free (pnew);
11         pnew=NULL;
```

```
        }
        if (phead==NULL) phead=pnew;
        else pfore->next=pnew;
        pfore=pnew;
    } while(pnew!=NULL);
}
```

3.6.2 展示链表

比别的操作简单多了，不返回值，参数为头节点地址. 构造 while 循环，之前说了最后一个节点的 next 为 NULL，所以循环条件设置为当前指针不为 NULL，循环过程就输出节点中的信息，然后给当前指针赋为这个指针指向节点中的 next，继续循环，如果是 NULL 就停了.

```
    void show (struct st *p)
{
    while (p!=NULL){
        printf ("%d %d", p->num, p->score);
        p=p->next;
    }
}
```

3.6.3 插入节点

分三种情况:

1. **插入到头节点：**新节点前面没有节点，后面有节点，思路：让新节点的 next 指向原头节点，头节点赋为新节点.
2. **插入到中间节点：**新节点前面有节点，后面也有节点. 思路：让新节点的 next 指向后节点，原节点的 next 指向新节点
3. **插入到尾节点：**新节点前面有节点，后面没有节点. 思路：让新节点的 next 为 NULL，前节点的 next 为新节点.

   函数返回值为新的头节点地址，参数为原链表头节点的指针和插入的新节点的指针. 这里在如何确定插入位置时用到了插入排序的思维. 这里把 p 叫当前节点（新节点后面的），pfore 叫前节点，pnew 叫新节点.

```
struct st *insert (struct st *p, struct st *pnew)
{
    struct st *phead, *pfore;
    phead=p;
    pfore=NULL;
    while (p!=NULL){
        if (pnew->num < p->num) break;
        else {
            pfore=p;
            p=p->next;
        }
    }
```

```
    if (pfore==NULL){
        pnew->next=p;//
        phead=pnew;
    }
    else {
        pnew->next=p;/*若不知后节点，则pnew->next=pfore->next，因为前节点的next原本就
            指向后节点*/
        pfore->next=pnew;
    }
    return phead;
}
```

3.6.4 删除节点

还是分三种情况:

1. **删除首节点：**把第二个节点的指针（头节点的 next）赋给头节点，释放原头节点内存.
2. **删除中间节点：**把前节点的 next 赋为后节点的指针/删除节点的 next，释放内存。
3. **删除尾节点：**把前节点的 next 赋为后节点的指针/删除节点的 next/NULL，释放内存.

传递参数是要删除节点的信息，和该链表的头节点地址. 函数返回新头节点指针，先把 pfore 赋值为 NULL，p 赋为头节点地址（存下原头节点），依据传入的删除节点的信息用循环一次次向后找找到. 找到后把删除节点地址就是 p，pfore 为前节点指针，然会删除.

```
struct st *delete (struct st *p, int num)
{
    struct st *pfore, *phead;
    pfore=NULL;
    phead=p;
    while (p!=NULL){
        if (p->num==num) break;
        pfore=p;
        p=p->next;
    }
    if (p==NULL) return phead;
    if (pfore==NULL){
        p=p->next;
        free (phead);
        phead=p;
    }
    else {
        pfore->next=p->next;
        free (phead);
    }
    return phead;
};
```

4 输入/输出 (input/output)

C 语言中不用关键字来执行输入输入, 而是通过标准库函数来执行.C 语言的标准输入输出也叫做 ANSI/ISO C I/O system.

4.1 读写字符

读取单个字符:

- **getchar**: 需要回车结束缓冲，回显到屏幕
- **getche**: 不需要回车，回显到屏幕
- **getch**: 不需要回车，不回显到屏幕

输出单个字符：putchar ()

读取字符串:

- **gets**: 遇到回车才停止，将回车读取后替换为 NULL，并且回车不会留在缓冲区.
- **scanf**: 遇到空白符就停止，后面补 NULL，但是空白符会留在缓冲区.

输出字符串:

- **puts**: 参数为字符串首元素地址，输出全部字符串，**结尾补上换行**.
- **scanf**: 格式修饰符为%s，参数为首元素地址.

4.2 printf

写的第一个 C 语言程序就有，但是好复杂的其实. 这是一个可变参数函数，参数中包含一个格式控制字符串，和可变参数，返回值为输出字符的数量.

- 输出字符：%c 输出单个字符，%s 输出字符串.
- 输出整数：%d 和%i 都是输出有符号长整型，%u 是输出无符号长整型.
- 输出浮点：%f 输出十进制浮点 (**是 double 类型，可变参数函数的参数都被提升了**).%e 和%E 以科学计数法输出浮点数，大小写只是为了区分输出的 E 的大小写 (普遍形式，x.dddddE+-yy).%G 和%g 根据%f 和%g 输出长短选择短的形式输出.
- 输出十六/八进制:%X 和%x 输出十六进制,%o 输出八进制.(前面不加 0x/0,除非写成%#x,%#o)
- 输出地址：%p 用来输出地址，输出的形式根据 CPU 决定，同时，也可以用%x%o 输出地址
- %n：平时不常见，简单来说就是，%n 需要对应的参数是一个整型指针，printf 运行到%n 时会将前面输出的字符个数赋值给参数指向的整类变量.

同时因为 printf 是格式化输出，所以有格式修饰符.

最小区域宽度修饰符:

% 和字母之间加上整数，即表示填充至至少多少位，默认为空格，若整数前加上 0 则用 0 填充. 如果本身区域宽度就比整数长，则按照原来输出.

精度修饰符:

在% 和字母之间加上. 整数，不同类型对应规则不同.

- %f：浮点为保留多少位小数.
- %g：科学计数法为保留多少位有效数字.
- %s：字符串为输出到的最大长度，后面不输出.
- %d：整数为输出的最小未熟，前面补 0.

对齐输出:

默认为右对齐，% 后整数前加上-负号强制改为左对齐.

其他数据类型:

有 **l 和 h** 作为修饰符可以使修饰 **d,i,o,u**. 具体看前. 有 **L** 可以修饰 **e,f,g**，表示 **long double**.

修饰符:

在 **g,G,f,E,e** 前加上 # 表示一定显示小数点 (默认保留 6 位). 在 **X,x** 前加 # 表示 16 进制输出前加上 **0X,0x**. 在 **o** 前面加上 # 表示在 8 进制输出前加上 **0**.

* 修饰符:

* 作为占位符，对应一个后面的参数，该参数表示该格式修饰符位置的数为多少

```
int main (void)
{
    double a=15.32;
    printf ("%*.*f\n", 10, 4, a);//输出00015.3200
    return 0;
}
```

5 程序结构 (program structure)

5.1 宏 (macro)

之前写程序每次开头 sharp punch 编译预处理指令 (preprocessor)，并不是不是 C 语言特有的. #define < 名字 > < 值 > 纯字符串替换没有分号，不是 C 的语句，如果有分号，分号也算文本替换的一部分。宏中如果有宏也会被替换，如果一个宏太长一行不够，可加反斜杠下一行接着（字符串也行）

虽然注释是 C 语言的语法，但是宏后可以写注释，注释不能成为宏的一部分

没有值的宏是合法的，用于条件编译，检查该宏是否被定义过

编译不是一步就完成，–sava-temps 可保留过程文件.i 文件是预处理后的文件

```
    __line__
    __FILE__
    __DATE__//记录运行版本

#define SQARE(x) ((x)*(x))
printf ("%d", SQARE(3)
#define MAX(a, b) (((a)>(b))?(a):(b))
int f (int a)
{
   return a+2;
}
int a=1, b=2, c=3;
printf ("%d", MAX (a+b, a+c));
printf ("%d", MAX (f(a+b), f(a+c)));//不优化
```

带参数的宏（像函数的宏），因此，要把所有东西都加上括号（整个值，所有参数）。

两个字符串中间没有任何符号会被连起来。宏中 # 加参数

switch case 编译时若不足 16 个 case 会被转成 if-else，但是都用 16 个了，还不如写一个数组，每个元素都是一个结构，每个结构成员是一个值和一个函数指针。

grep 寻找字符串

条件编译 #ifdef #ifndef #endif 不改变源代码，编译时选项加-D<MACRO> 定义一个宏简化版本 #if <macro> 宏非 0 才编译（字符?）#elif <macro> 相当于 else if #else #endif

快速注释大段代码选中 ctrl+/

#if 0

#end if

#error "string"告诉编译器到这编译出错，用于定义宏条件编译组合错误时终止编译。

5.2 变量

5.2.1 局部变量 (local variable)

声明在函数内的变量生存期/作用域, 从里向外遇到的第一个大括号, 可以渗透到内部的大括号。最近原则，若内外同名，总是寻找最近定义的。一个大括号内不能重复定义一个变量。局部变量没有确定的初始值,

函数声明可以在 main 函数内部声明 main 不是第一行代码，编译后前面还有初始的引导代码，只是调用了 main，所以 main 不能改名字。

C 语言编译过程:

1. 在编译时候加上–save-temps 保留中间代码
   gcc a.c –save-temps
2. 编译预处理，把所有的 # 都补充完整.c–>.i
3. 编译器编译成汇编代码.i–>.s
4. 汇编器汇编成目标代码.s–>.o
   其中有所有 C 语言代码的主体，全局变量布局，字面量（字符串、常量等），**未定位的**函数
5. 链接器将引导代码、库函数代码和.o 代码链接在一起成.exe(.out) 可执行文件
   静态链接是库函数代码在可执行文件内，动态链接是库函数代码不在可执行文件内（默认）
6. 上述文件可通过 gcc 分步加选项进行，比如加-c 只会编译到目标文件不链接.

一个.c 文件是一个**编译单元**，编译器每次只能处理一个编译单元，即**每次只能看见一个.c 文件**. 而后就是怎么将不同编译单元结合到一起，不同语言有不同的策略. 先说 C 语言中有 main 的文件中要调用函数必须有原型，没有原型就不能编译到.o 文件，而在其他.c 文件中的函数主体也只能编译到.o 文件. 这时候就想，这个函数写出来可能给很多程序用，并不止这一个，那每个都写一个原型太麻烦还容易错，所以就用**头文件**. 每个.c 函数文件都配套一个头文件，**把函数的原型都写进.h 头文件，在有 main 函数的文件开头 #include 这个头文件**，原型就存在于这个文件中了，在编译预处理时，#include 做了文本插入展开的工作（**原封不动的插入，文本复制**）. 最后不要忘了，标准库函数的 include 用的 < > 插入，**< > 表示到指定位置寻找头文件，即默认环境变量中，或者在编译时候选项指定.** 而自己写的头文件一般是用双引号""，**"" 代表先在.c 文件所在目录下寻找头文件，若没有再去指定位置寻找.**

8025，每行不超过 80 个字符，一个源文件在 100 行以内

5.3 变量存储类别

动态局部变量 (local autoatic variable)

定义在函数内或者函数头中并且不用 static 修饰的变量

全局变量 (global variable)

定义在所有函数外的变量.

如果 main 函数内还定义了同名变量，在 main 函数使用这个名字优先是局部变量. 如果非要用全局变量，需要在定义局部变量前取全局变量的指针，然后通过指针操作.

局部静态变量 (local static variable)

定义在函数体内并用 static 修饰的变量叫局部静态变量，反面就是之前讲的局部静态变量，本质需要用 auto 修饰，但是现在编译器默认为动态，所以直接声明即可. 局部静态变量的作用域是它所属的代码块，生命期是从 main 开始到结束. **局部静态变量的定义并赋值语句在 main 开始运行时就执行了，调用函数时该语句从不被执行，当该变量所处的函数调用结束时，该变量并不消失，若未赋值，初始值为 0.**

```
char *week (int i)
{
    char w[7][4]={

    };
    return w[i];
}
int main (void)
{
    char *p;
    p=week(0);
    puts (p);//错误，因为数组w在函数结束时已经不存在，返回得到的p只有地址回来了，但是这个地址上存的东西已经没了
    return 0;
}
//return 返回时，会先把值保存到CPU的值保存到CPU内部的一个全局变量中再带回给main
```

全局静态变量

在所有函数外部用 static 修饰声明的变量，作用域只在自己自己所在文件全局.

注意，之前说过函数头文件，假设在一个文件中声明了动态全局变量，想要在另一个文件中使用这个变量，需要 extern 一下.

6 文件操作

文件操作本质就四个：打开文件，读文件，写文件，关闭文件.

首先 **FILE** 是一个结构类型，定义时用的 typedef struct. 平时我们使用的是文件指针，FILE 已定义好，不需要管这个结构内是什么.

fopen 是打开文件函数，两个参数分别为字符指针，第一个是文件名，第二个是打开方式. 返回值就是指向该文件的指针，**若打开失败，则返回 NULL，打开后一定要判断是否打开成功**

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

```
#include <stdlib.h>
int main (void)
{
   FILE *fp;
   char c;
   fp = fopen ("a.txt", "r");
   if (fp==NULL) exit (1);
   while (!feof (fp)){
      c=fgetc (fp);
      putchar (c);
   }
   return 0;

}
```

打开方式：在 windows 上分为文本格式和二进制格式. **敲回车**是回车 (') 加换行 (). 但是正常写代码只用写, 而保存在文件中以为回车 + 换行, **读取文件时又会只读**，总的来说就是读的时一个变成两个，. 而在 Linux 中

1. "r" 表示只读文件,
2. "w" 表示只写，若文件不存在则创建新文件，若存在则覆盖写入.
3. "a" 表示追加，若文件不存在则创建新文件，若存在则在已有的内容后追加写入.
4. "r+" 要求打开文件已存在，可读可写.
5. "w+" 会重新创建新文件，可读可写.
6. "a+" 若文件存在不会删除原文件，可读可写.

用不包含 "b" 的打开方式，读取文件会把 0x0D 和 0x0A 整合成一个 0x0D

feof 是判断文件内容是否被读完的函数. **本质：**文件用 fopen() 打开后，有一个文件内部的指针指向现在读带哪个位置 (offset 偏移量). 若 offset 已到文件末尾 (EOF,end of file)，feof() 返回真 (1)，若没有文件末尾则返回假 (0). **注意：**当 offset 已经到末尾时 feof 还不返回 1，需要在读取一个字符向后便宜一位才会返回 1, **而读取到的数据是-1，对 char 类型是 255**.

改进:

```
    while ((c = fgetc(fp))!=EOF)
```

字符输入输出:

- **fgetc:** 参数为函数指针，读取当前 offset 后的一个字符，返回值为该字符.
- **fputc:** 参数第一个为字符变量, 第二个文件指针, 将第一个字符写入第二个文件当前 offset 后.

字符串输入输出:

- **fgets:** 有最大元素个数限制读取当前偏移量后的字符串 (**直到换行符，但是会读取回车，后面再追加一个 0**). 第一个参数为用来存储的字符串，第二个参数为读取最大数量，第三个参数为文件指针. 返回字符串首元素指针. **可能会读取失败，会返回 NULL**.
- **fputs:** 将字符串输出到文件当前偏移量后. 第一个参数为字符指针，第二个参数为文件指针. **输出到空字符前为止，不追加换行符.**

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
```

```
{
   FILE *fp1, *fp2;
   char s[100];
   fp1 = fopen ("a.txt", "r");
   fp2 = fopen ("b.txt", "w");
   if (fp1==NULL||fp2==NULL) exit (1);
   while (fgets (s, sizeof(s)-1, fp1)){
      fputs (s, fp2);
      puts (s);
   }
   fclose (fp1);
   fclose (fp2);
   return 0;
}
```

格式读写:

- fscanf
- fprintf

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
   FILE *fp1, *fp2;
   int sum=0, score, n=0;
   char name[10];
   fp1=fopen ("a.txt", "r");
   fp2=fopen ("b.txt", "w");
   if (fp1==NULL||fp2==NULL) exit (1);
   while (!feof (fp1)){
      fscanf (fp1, "%s%d", name, &score);
      sum+=score;
      n++;
   }
   fprintf (fp2, "sum=%d average=%d\n", sum, sum/n);
   fclose (fp1);
   fclose (fp2);
   return 0;
}
```

rewind() 用于将偏移量重置到文件开头首字节处.

7 库函数

7.1 stdio.h

1.abort

8 算法

进制转换:

边计算边输出:

```
void dectobin(unsigned int x)
{
    if (x<2){
        printf ("%d", x);
    }
    else {
        dectobin (x/6);//f(x>>1);
        dectobin (x%2);//f(x&1));
    }
}
```

结果存入字符串中

```
int f (unsigned int x, char *str);
int main (void)
{
    char a[100];
    int n;
    n=f (6, a);//n表示计算出来的二进制位数
    a[n]='\0';//a用来保存二进制字符串
    puts (a);
    return 0;
}
int f (unsigned int x, char *str)
{
    int n;
    if (x<2) {
        *str=x+'0';
        return 1;
    }
    else {
        n=f (x>>1, str);
        n+=f (x%2, str+n);
        return n;
    }
}
```

走迷宫（深度优先搜索）/扫雷