数据结构

知识点笔记

一、指针

1. 基本定义和读写认知

指针是一个寄存器，每个寄存器都有唯一的地址编码，该寄存器又分为数据寄存器和地址寄存器，指针所指的数据内容存放在数据寄存器，而“那根指针”当前状态的地址就存放在地址寄存器。

也因此，指针前面所定义的数据类型是定义的数据寄存器中数据类型；而地址寄存器根据系统的不同，永远是固定不变的宽度。

【写入数据】

```
*a = 5; //将 5 写入到 a 的数据寄存器中
*a = *a + 1; //将 a 数据寄存器中的值加 1 再存到 a 的数据寄存器中
```

【移动指针】

```
a = a + 1; //把 a 的地址寄存器中的地址值加 1 个数据宽度
```

【读出数据】

```
x = *a; //将 a 的数据寄存器中的数据读出赋给变量 x
y = a; //将 a 的地址寄存器中的地址值存入变量 y
x = *(a+1); //将当前 a 的地址寄存器的地址值加 1 个数据宽度后的那个数据寄存器中的数据存入变量 x
```

【区分指针符号和乘法运算符】

```
a*b;//a 乘以指针 b 的地址寄存器        2*b;//指针 b 的地址寄存器乘以 2
a+*b;//a 的值加上指针的数据寄存器      a*(*b);//a 的值乘以指针 b 的数据寄存器
```

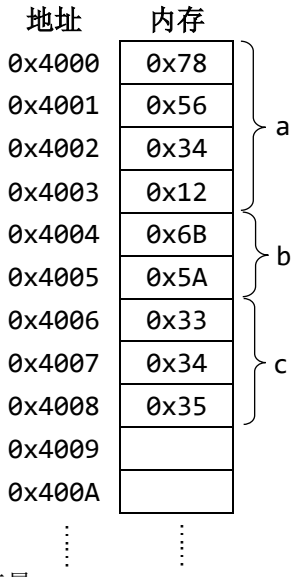
注意：以上的例子均是在已定义好指针的前提下。

2. 计算机存储机制

将如下几个变量存放到内存中，其分配方式如右图所示。

```
int a = 0x12345678;
short b = 0x5A6B;
char c[] = {0x33, 0x34, 0x35};
```

说明：对于 a 来说，数据低位被存在内存的低地址处，这种存储方式叫小端模式，反之则是大端模式；对于 b 来说也是小端存储模式；对于 c 来说，由于是数组，存储方式是连续分配的。



3. 指针与指针变量的区别

（1）概念

①**指针：**一个变量的地址称为该变量的指针。通过变量的指针能够找到该变量。

②**指针变量：**专门用于存储其他变量地址的变量。指针变量 pnum 的值就是变量 num 的指针（地址），指针与指针变量的区别，就是变量值与变量的区别。

③为表示指针变量和它指向的变量之间的关系，用指针运算符“*”表示。

（2）代码理解

```
int num = 10;
&num                  //num 的地址或者叫 num 的指针
int* pnum = &num;     //pnum 是指针变量，这里存储 num 整形变量的指针（地址）
*pnum = 20;           //通过 pnum 访问指向的指针来访问对应变量的内存空间，使用*pnum 就相当于取到 pnum 指向的变量值（该操作也称为解引用）
```

（3）注意事项

①指针变量的值是可以改变的，所以必须注意其当前值，否则容易出错。

②指向数组的指针变量，可以指向数组以后的内存单元，虽然没有实际意义。

（4）指针的宽度（**注：** 16 位系统的 *x*=2；32 位系统的 *x*=4；64 位系统的 *x*=8）

数据类型		指向该数据类型的指针	
(unsigned) char	1 字节	(unsigned) char*	*x* 字节
(unsigned) short	2 字节	(unsigned) short*	*x* 字节
(unsigned) int	4 字节	(unsigned) int*	*x* 字节
(unsigned) long	4 字节	(unsigned) long*	*x* 字节
float	4 字节	float*	*x* 字节
double	8 字节	double*	*x* 字节

4. 指针的操作

若已定义：

```
int a;      //定义一个 int 型的数据
int* p;     //定义一个指向 int 型数据的指针
```

则对指针 p 有如下操作方式：

操作方式	举例	解释
取地址	p=&a;	将数据 a 的首地址赋值给 p
取内容	*p;	取出指针指向的数据单元
加	p++;	使指针向下移动 1 个数据宽度
	p=p+5;	使指针向下移动 5 个数据宽度
减	p--;	使指针向上移动 1 个数据宽度
	p=p-5;	使指针向上移动 5 个数据宽度

5. 数组与指针的联系（易混淆知识点）

（1）数组的概念

数组是一些相同数据类型的变量组成的集合，数组的定义等效于申请内存、定义指针和初始化。一维数组名通常被视为指针常量，这是因为数组名代表数组首个元素的地址，并且这个地址在大多数情况下是不可修改的。然而，需要注意的是，数组名并不等同于一个指向数组的常量指针。在 C 语言中，数组名具有特定的类型和行为，它与指向数组的指针（例如 type (*pointer)[size]）有所不同。

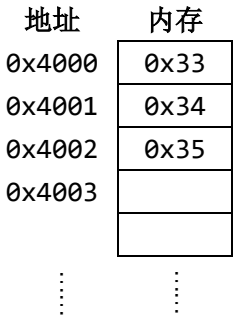
例 1： `char c[] = {0x33, 0x34, 0x35};`

等效于：申请内存

定义 `char* c = 0x4000;` //此处地址是举例

初始化数组数据

（**说明：例 1** 如右图所示）



例 2：利用下标引用数组数据也等效于指针取内容。

```
c[0]; //等效于*c;
c[1]; //等效于*(c+1);
c[2]; //等效于*(c+2);
```

（2）数组与指针的关系是什么？

数组并非指针，在初学 C 语言时，我们会觉得“数组和指针是相同的”，实际上，这是一种错误的说法，并不完全正确。下面先来看一段代码：

```
int mango[10];          //文件 1
extern int* mango;      //文件 2
```

上面程序演示了一个错误，文件 1 定义了数组 mango，文件 2 想使用它，声明它为指针。但实际上它们的类型并不匹配，相当于把整数和浮点数混为一谈。

但是为什么人们会认为指针和数组始终是应该可以互换的呢?答案是对数组的引用总是可以写成对指针的引用，而且确实存在一种指针和数组的定义完全相同的上下文环境。不幸的是，这只是数组的一种极为普通的用法，并非所有情况下都是如此。

（3）数组与指针的区别在哪？

它们访问过程不一样。如果是数组，数组名就代表了数组第一个元素的地址，如果需要访问数组中的元素，编译器可以直接进行操作，因为有地址（其他元素加上偏移量即可），并不需要增加指令首先取得具体的地址。相反，对于指针，必须首先在运行时取得它的当前值，然后才能对它进行解除引用操作。

①对数组下标的引用

```
char a[8] = "abcdefgh";
char c = a[1];
```

②对指针的引用

```
char* p;
c = *p;
```

注意：如果声明 extern char* p，它将告诉编译器 p 是一个指针，它指向的对象是一个字符。为了取得这个字符，必须得到地址 p 的内容，把它作为字符的地址并从这个地址中取得这个字符。

提示：出现在赋值符号左边的符号有时被称为左值，出现在赋值符号右边的值有时被称为右值。编译器会为每个变量分配一个地址（左值），产生一个符号表。这个地址在编译时可知，而且该变量在运行时一直保存于这个地址。相反，存储于变量中的值（它的右值）只有在运行时才可知。如果需要用到变量中存储的值，编译器就发出指令从指定地址读入变量值并将它存于寄存器中。

数组与指针的其它区别如下：

指　针	数　组
保存数据的地址	保存数据
间接访问数据，首先取得指针的内容，把它作为地址，然后从这个地址提取数据。若指针有一个下标[i]，就把指针的内容加上 i 作为地址，从中提取数据	直接访问数据，a[i]只是简单地以 a+i 为地址取得数据
通常用于动态数据结构	通常用于存储固定数据类型相同的元素
相关的函数为 malloc()、free()	隐式分配和删除
通常指向匿名数据	自身即为变量名

（4）数组与指针什么时候可以等同？

所有**作为函数参数**的数组名总是可以通过编译器转换为指针。在其他所有情况下，数组的声明就是数组，指针的声明就是指针，两者不能混淆。但在使用数组（在语句或表达式中引用）时，数组总是可以写成指针的形式，两者可以互换。

（5）数组名

数组名要区分以下几种情况：

①`sizeof(arr)`：表示的是整个数组的大小。

②`&arr`：表示整个数组，取出的是整个数组的大小。

③其他情况数组名都表示数组首元素地址。

6. 数组指针与指针数组

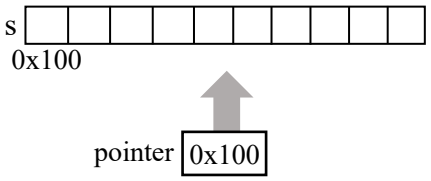
数组指针和指针数组的区别主要体现在它们的数据结构和用途上，以下是两者的具体区别：

①数据结构不同。数组指针是一个指针，它指向一个数组的首地址；指针数组是一个数组，其中的每个元素都是指针类型。

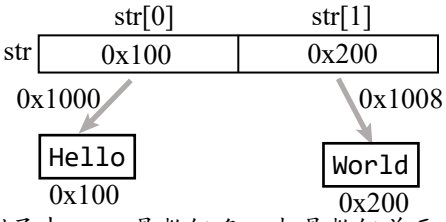
②用途不同。数组指针主要用于访问和操作它指向的那个数组的所有元素，通常用于处理多维数组；指针数组的每个元素可以指向不同的内存地址，因此它更灵活，可以用于存储不同类型或不同位置的数据。

③声明方式不同。数组指针的声明方式通常是 `type (*pointer)[size]`；指针数组的声明方式通常是 `type* pointerName[count]`。下面通过图示来更好地理解：

【数组指针】例：
```
char (*pointer)[10] = NULL;
char s[10] = "Hello";
pointer = &s;
```



【指针数组】例：`char* str[2] = {"Hello", "World"};`



注意：在上面这个指针数组的例子中，str 是数组名，也是数组首元素地址，即 str 是 0x100 的地址 0x1000，str+1 就是 0x200 的地址 0x1008。**（前提条件是系统为 64 位）**

7. 数组参数与指针参数

（1）数组参数

一维数组传参： int arr[10]（形参部分数组大小可以不写）、int* arr

二维数组传参： int arr[3][5]、int arr[][5]（行可以省略，列不能）、int (*arr)[5]

（注意：二维数组不能直接传数组名，二维数组的首地址是第一行元素的首地址。）

（2）指针参数

一级指针传参：int* p

二级指针传参：int** p

8. 野指针

我们对没有指向的指针变量叫做野指针。野指针会造成非法访问内存，从而导致程序崩溃或出现意想不到的错误，这是很危险的，所以在定义指针变量时，如果暂时没有明确的指向，对其一般初始化为 NULL（空指针）。

例如：`int* parray = NULL;`

9. 指针运算

（1）指针±整数（px±*n*）

将指针从当前位置向前(+)或回退(-)*n* 个数据单位（*n* 个数据类型的大小）。

例如：当一个 int*的指针+1，跳过 4 个字节；当一个 char*的指针+1，跳过 1 个字节。

（注意：这里的加减数值其实是偏移量，并非真正的加减运算）

示例代码：通过下列代码来理解。

```
int a[] = {52,47,95,123,53,66};
int main(){
    int* px = &a[3];
    int* py = &a[1];
    int* m = px+1;
    printf("%p\n", px);
    printf("%p\n",m);
    return 0;
}
```

输出的结果是：

```
    00007FF73615D0B4
    00007FF73615D0B8
```

说明：

①px+py 没有意义（两个指针变量或指针相加没有意义，加完的那个数据不知道指向何处）

②指针变量定义时要么指向明确变量的地址，要么初始化为空。如果不初始化，p 有可能指向任何地方，这种指针叫野指针，通过野指针去改变指向的不确定的地址的行为是很危险的。此外，动态分配内存，释放之后，没有清空，这种也是野指针。

（2）指针－指针（px−py）

指针－指针的绝对值指的是两个指针之间数据元素的个数，而不是指针的地址之差。（**前提：**两个指针必须指向同一空间）

例如：&arr[9] － &arr[0]

示例代码：下面通过一个例子来实现 my_strlen 函数。

```
int my_strlen(char str[]){
    char* p = str;
    int count = 0;
    while (*p != '\0'){
        p++;
        count++;
    }
    return count;
}
int main(){
    char arr[] = "abcdef";
    printf("%d\n",my_strlen(arr));
    return 0;
}
```

（3）指针的关系运算

指针与指针之间比较大小就是指针的关系运算。但是要遵循一个标准：允许指针指向数组元素和指针指向数组最后一个元素后面的位置进行比较，不允许指针指向数组元素与指针指向数组第一个位置的前面进行比较。

例如：定义 int* p1，数组 int arr[5]，p1>&arr[5]。

10. 二级指针

二级指针就是存放指针地址的指针变量（存放“地址的地址”）。就像有三个抽屉，第一个抽屉的钥匙放在第二个抽屉，第二个抽屉的钥匙放在第三个抽屉。

例如：

```
int a = 10;
int* p1 = &a;
int** p2 = &p1;
```

11. 指针函数与函数指针

（1）指针函数：返回值是指针的函数就是指针函数。

下面通过一个例子来详细说明：

```
int func(){
  int m = 8;
  return m;
} //这个函数中的返回值类型与定义函数时的数据类型必须对应！
```

顾名思义，我们由上面这个法则可知，返回值的类型为指针的函数就是指针函数，如下：

```
int* func2(){
  int a = 10;
  int* pa_ = &a; //将变量 a 的地址给指针 pa_
  return pa_;    //返回指针 pa_，定义函数时的数据类型也为对应的指针类型
}
```

我们现在再写一个函数 func3，代码如下：

```
int* func3(){
    int b = 22;
    int* pb_ = &b; //将变量 b 的地址给指针 pb_
    return pb_;    //返回指针 pb_，定义函数时的数据类型也为对应的指针类型
}
```

然后，我们来调用 func2 和 func3 这两个函数，看看会发现什么问题。

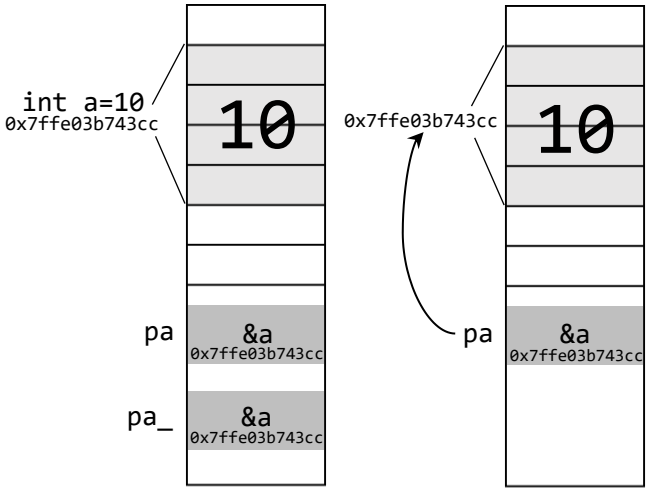
```
int main(){
  int* pa = func2();
  printf("pa:%p\n",pa);
  printf("*pa:%d\n",*pa);

  int* pb = func3();
  printf("pb:%p\n",pb);
  printf("*pb:%d\n",*pb);

  printf("pa:%p\n",pa);
  printf("*pa:%d\n",*pa);
  return 0;
}
```

运行结果如下：

```
pa:0x7ffe03b743cc
*pa:10
pb:0x7ffe03b743cc
*pb:22
pa:0x7ffe03b743cc
*pa:22
```

正如右上方运行结果所示，变量 a 的值没有发生改变，但是当第二次输出*pa 时，却变成了 22，下面通过一个图来解释这个现象。（右图为虚拟内存，一小格代表 1Byte=8bit）

```
int main(){
    int* pa=func2();
    int* pb=func3();
}


int* func2(){
    int a = 10;
    int* pa_ = &a;
    return pa_;
}
```



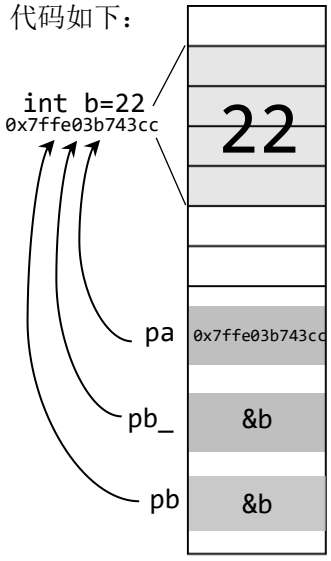
（图 1）

（图 2）

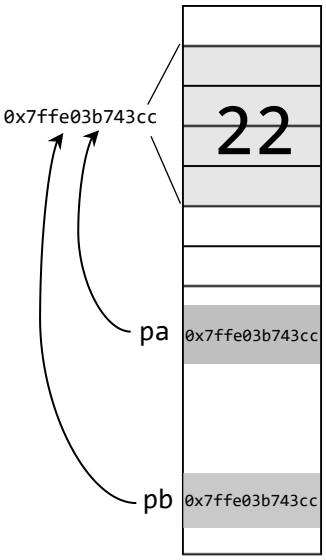
说明：上图为虚拟内存运行这段代码时的存储情况（如图 1），当函数 func2 运行完后，函数中的变量会自动销毁，这包括变量 a 和指针 pa_（如图 2）。因为在函数运行结束后，函数中定义的所有变量（栈区变量）都会被自动释放。

现在继续调用 func3 函数，代码如下：

```
int* func3(){
    int b = 22;
    int* pb_ = &b;
    return pb_;
}
```



（图 3）



（图 4）

说明：函数 func3 运行（如图 3），当函数 func3 运行结束时，变量 b 和指针 pb_自动销毁（如图 4），虽然函数中定义的变量会销毁，但在原内存中保存的数据仍然存在，而指针 pa 又一直指向原来的那个地址，但那个地址所对应的内存中存放的数已经变成 22，故*pa=22。

根据上述例子，我们将两个函数中所定义的指针 pa_和指针 pb_称为**悬挂指针**。言归正传，上述这种情况我们必须要规避才能使输出的结果正确。此时需要引入一个概念——堆区变量，堆区变量一般由程序员手动分配和释放，释放的方式是使用 free()或.out，若程序员不释放，程序结束时可能由 OS 回收。注意它与数据结构中的堆是两码事。只要使用堆区变量就能够达到上述示例代码输出结果正确的目的，现在我们就利用 malloc 函数进行动态内存分配来创建堆区变量，具体过程如下：

```
int* func4(){
    int* pa_ = (int*)malloc(sizeof(int));
    *pa_ = 10;
    return pa_;
}
int* func5(){
    int* pb_ = (int*)malloc(sizeof(int));
    *pb_ = 25;
    return pb_;
}
int main(){
    int* pa = func4();
    printf("pa:%p\n",pa);
    printf("*pa:%d\n",*pa);

    int* pb = func5();
    printf("pb:%p\n",pb);
    printf("*pb:%d\n",*pb);

    printf("pa:%p\n",pa);
    printf("*pa:%d\n",*pa);
    free(pa);
    free(pb);
    return 0;
}
```

运行结果如下：

```
pa:0x55fd80a322a0
*pa:10
pb:0x7ffd8232e9dc
*pb:25
pa:0x55fd80a322a0
*pa:10
```

（2）函数指针：指向函数的指针

定义函数指针的方法：`int (*pa)(int)=NULL;`

说明：因为()的优先级大于*，所以上述代码中的主体是指针（根据第一个括号），代表这是一个指向函数的指针，第二个括号中的 int 代表该函数的参数是 int 变量，而指向的这个函数的返回值是 int 类型。

下面通过一段代码来帮助理解：

```
int func(int a){
    int b = 10;
    int c = a*b;
    return c;
}
int main(){
    int (*pa)(int)=NULL; //定义一个指向函数的指针
    pa = func;   //让指针与函数对接
    int result1 = func(10);
    int result2 = pa(10);
    printf("func(10):%d\n",result1);
    printf("pa(10):%d\n",result2);
    return 0;
}
```

（3）总结

项目	主体（是指针还是函数）	特点
指针函数	函数	返回类型为指针
函数指针	指针	指向一个函数

（4）悬挂指针与野指针的区别

①**悬挂指针：**悬挂指针是指向已经释放（例如通过调用 delete（C++语法）或 free（C 语法））或已经超出作用域的内存的指针。当我们试图通过这样的指针访问或操作内存时，就可能导致未定义行为，因为那块内存可能已经被操作系统重新分配给其他程序进行使用了。

示例代码：

```
int* ptr = (int*)malloc(sizeof(int));
free(ptr);   //现在 ptr 是一个悬挂指针
*ptr = 10;   //未定义行为
```

②**野指针：**野指针是一个未初始化的指针，也就是说，它的值是未知的，可能指向任意内存地址。如果我们试图通过下面代码这样的指针访问或操作内存，同样可能导致未定义行为。

示例代码：

```
int* ptr;   //ptr 是一个野指针
*ptr = 10;  //未定义行为
```

12. 函数指针数组

函数指针数组就是**存放函数指针类型元素的数组**。也就是说，数组中的每一个元素都能存放一个函数的入口地址，函数名就这样就能通过数组元素来调用函数。它能够实现计算器的应用。函数指针数组的定义方式例如：`int(*pfarr[])(int, int) = {0,Add,Sub,Mul,Div}`（前提条件是需要先定义好存入数组的相关函数）

另一个概念是**指向函数指针数组的指针**，定义比如：`int (*(*parr)[4])(int int)=&parr`

下面是计算器的应用代码：

```
int Add(int x, int y){
    return x + y;
```

```
}
int Sub(int x, int y){
    return x - y;
}
int Mul(int x, int y){
    return x * y;
}
int Div(int x, int y){
    return x / y;
}
void menu(){
    printf("* 1.Add  2.Sub *\n");
    printf("* 3.Mul  4.Div *\n");
    printf("***  0.exit  ***\n");
}
int main(){
    menu();
    int input = 0;
    printf("请选择：");
    scanf("%d",&input);
    int ret = 0;
    int(*pfarr[])(int, int) = { 0,Add,Sub,Mul,Div };
    do{
        if (input == 0){
            printf("退出\n");
            break;
        }
        else if (input >= 1 && input <= 4){
            int x = 0;
            int y = 0;
            printf("请输入两个操作数：");
            scanf("%d %d",&x,&y);
            ret = pfarr[input](x,y);
            printf("结果是%d\n",ret);
            break;
        }else{
            printf("选择错误！");
        }} while(input);
    return 0;
}
```

13. 回调函数（Callback Function）

（1）定义

回调函数就是一个通过函数指针调用的函数（对某段代码的引用，它被作为参数传递给另一段代码，并在某个时刻被调用）。

如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是**回调函数**。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。一般来说，当某个函数内部缺少一段逻辑，需要动态补充时，我们就要用到回调函数。

下面通过一段代码来理解：

```
int foo(void){
    return 1;
}
int bar(int (*param)(void)){
```

```
    return param();
}
int main(void){
    bar(foo);
}
```

说明：在上面这段代码中，foo 函数被作为参数传递给了 bar 函数，bar 函数在内部通过函数指针 param 间接调用了 foo 函数，因此 foo 函数在整个代码逻辑的行为中符合回调函数的定义。

（2）qsort()经典回调函数

qsort()是一个**库函数**，基于**快速排序**算法实现的一个排序的函数。优点是**任意类型的数据都能排序**。qsort()函数的形参定义：void qsort(void* base(**起始地址**),size_t num(**元素个数**),size_t width（**一个元素的字节长度**）,int (*cmp)(const void* e1,const void* e2)（**自定义比较函数**））

（3）qsort()函数应用

①模拟计算器

```
void print(int* str, int sz){
    int i = 0;
    for (i = 0; i < sz; i++){
        printf("%d ", str[i]);
    }
    printf("\n");
}
void swap(char* e1, char* e2, int width){
    int temp = 0;
    int i = 0;
    for (i = 0; i < width; i++){
        temp = *e1;
        *e1 = *e2;
        *e2 = temp;
        e1++;
        e2++;
}}
int cmp(const void* e1, const void* e2){
    return (*(int*)e1 - *(int*)e2);//e1>e2-> >0;e1=e2-> 0;e1<e2-> <0
}
int bubble_sort(void* base, int num, int width, int(*cmp)(const int* e1, const
int* e2)){
    int i = 0;
    int j = 0;
    for (i = 0; i < num - 1; i++){
        for (j = 0; j < num - 1 - i; j++){
            if (cmp((char*)base + j * width, (char*)base + (j + 1) * width)>0){
                swap((char*)base+j*width, (char*)base+(j + 1)*width, width);
}}}}
int main(){
    int arr[] = { 9,8,7,3,5,4,2,1,6,0 };
    int sz = sizeof(arr) / sizeof(arr[0]);
    bubble_sort(arr, sz, sizeof(arr[0]), cmp);
    print(arr, sz);
    return 0;
}
```

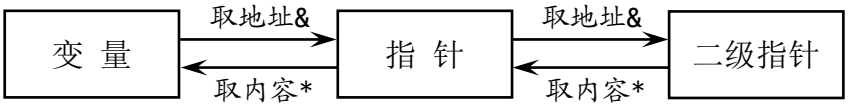
②排序结构体

```
struct Student{
    char name[20];
    int age;
    double score;
};
int cmp_name(const void* e1, const void* e2){
    return strcmp(((struct Student*)e1)->name, ((struct Student*)e2)->name);
}
int main(){
    struct Student arr[] = {{"zhang",17,80.6},{"wang",20,85.2},{"li",19,92.0}};
    int sz = sizeof (arr) / sizeof (arr[0]);
    qsort(arr,sz,sizeof (arr[0]),cmp_name);
    return 0;
}
```

14. 指针使用中易出错的情况

在对指针取内容之前，一定要确保指针指在了合法的位置，否则将会导致程序出现不可预知的错误；其次，指针必须指向一个和定义的类型相同的变量。

如下图所示，跨级赋值将会导致编译器报错或警告，此时需要取地址或取内容的操作，使其变为同级指针。（箭头代表向某种数据类型赋值）



15. 指针的应用

（1）传递参数

使用指针传递大容量的参数，主函数和子函数使用的是同一套数据，避免了参数传递过程中的数据复制，提高了运行效率，减少了内存占用，但由于主函数与子函数是共用数据，所以原数据可能改变，为了避免在调用函数的过程中对原数据误改，可在函数定义参数中的指针变量前面加上const（只读）。

指针传递与值传递参数有所不同，值传递仅仅为单向传递，当函数调用完后不会对主函数中原参数产生影响，但由于是拷贝参数到函数，所以加大内存占用从而导致运行效率低。

主函数与子函数共同数据并不一定都是弊端，也能够利用主函数和子函数使用同一套数据的特性，实现数据的返回，也可实现多返回值函数的设计，下面通过一段代码来设计多返回值函数：

```
void FindMaxAndCount(int* max, int* count, const int* array, int length){
    *max = array[0];
    *count = 1;
    for(int i = 1; i < length; i++){
        if(array[i] > *max){
            *max = array[i];
            *count = 1;
        }else if(array[i]==*max){
            (*count)++;
}}}
int main(){
    int a[] = {25, 149, 32, 16, 52, 33};
    int Max, Count;
    FindMaxAndCount(&Max, &Count, a, 6);//把变量 Max 和 Count 的地址传递给了指针变量
    printf("最大值为%d\n 最大值的个数为%d\n",Max, Count);
    return 0;
}
```

说明：在上述这段代码中，对于 Max 这个整型变量来说，在内存中开辟了 4 个字节的空间，而指针变量 max 在内存中开辟了 8 个字节的空间（64 位系统），用于存入整形变量 Max 的地址，当函数调用后，运算结果通过地址又返回到主函数中，实现了 Max 值的改变。Count 也同理。

（2）传递返回值

将模块内的公有部分返回，让主函数持有模块的“句柄”，便于程序对指定对象的操作。下面通过一段代码来详细说明：

```
int Time[] = {23,59,55};  //这里必须设成全局变量
int* GetTime(){
    return Time;
}
int main(){
    int* pt;
    pt = GetTime();
    printf("pt[0] = %d\n",pt[0]);
    return 0;
}
```

注意：如果把 Time[]设成函数局部变量，则不能输出正确的值，因为在函数调用完后，局部变量自动销毁，这时所返回的值会出问题。

（3）直接访问物理地址下的数据

①访问硬件指定内存下的数据，如设备 ID 号等。

②将复杂格式的数据转换为字节，方便通信与存储。

二、结构体（重点）

1. 使用结构体的原因

因为在实际问题中，一组数据往往有很多种不同的数据类型。例如，登记学生的信息，可能需要用到 char 型的姓名，int 型或 char 型的学号，int 型的年龄，char 型的性别，float 型的成绩。又例如，对于记录一本书，需要 char 型的书名，char 型的作者名，float 型的价格。在这些情况下，使用简单的基本数据类型甚至是数组都是很困难的。而结构体（类似 Pascal 中的“记录”），则可以有效的解决这个问题。

结构体本质上还是一种数据类型，但它可以包括若干个“成员”，每个成员的类型可以相同也可以不同，也可以是基本数据类型或者又是一个构造类型。

结构体的优点：结构体不仅可记录不同类型的数据，而且使得数据结构是“高内聚，低耦合”的，更利于程序的阅读理解和移植，而且结构体的存储方式可以提高 CPU 对内存的访问速度。

2. 如何声明结构体类型

（1）基本的结构体概念

以学生的个人信息为例，示例代码（结构体代码段）如下：

```
#include <stdio.h>
struct Student{      //注意：struct 后面的是结构体的数据类型，不是名称!!
    int id;
    char* name;
    int age;
    float score;
};
```

当用上述这样的结构体代码，无论是调用函数还是在结构体代码段之外使用 Student 这个代码时，都要在前面加上 struct，即 struct Student。如果想在结构体代码段之外不加 struct，那么可在上述这段结构体代码的后面加上 typedef struct Student Student1;此时定义结构体变量就是 Student1 stu，注意一下，这时的 Student1 是新定义的数据类型，而 stu 是变量名称。不过我们一般写代码不用以上这种方法，直接用以下这段代码即可，示例代码如下：

```
typedef struct Student{      //注意：这里的 Student 可不要，但建议加上
    int id;
    char* name;
    int age;
    float score;
}Student;        //这里的 Student 是结构体的数据类型别名
```

以上这段代码在定义结构体变量时，可直接使用 `Student stu;`

现若想用上面没改进前的那段代码，让 stu1 和 stu2 这两个学生信息的变量使用上述结构体，则需要把 struct 后面的 Student 去掉，然后在结构体的尾部加上这两个学生的变量名称（注意：stu1 和 stu2 不是数据类型！），以后就可以使用这两个变量，这叫匿名结构体，一般很少用，代码示例如下：

```
struct {
    int id;
    char* name;
    int age;
}stu1, stu2; //这两个是变量名，而不是数据类型
stu1.id=66;         //使用匿名结构体
stu2.name="小明";  //使用匿名结构体
```

（2）结构体的嵌套

设学生的生日信息单独构成一个结构体，然后再把这个结构体嵌套进上述学生个人信息的结构体中，示例代码如下：

```
typedef struct Birthday{
    int year;
    int month;
    int day;
}Birthday;

typedef struct Student{
    int id;
    char* name;
    int age;
    float score;
    Birthday birthday;    //将学生生日信息结构体嵌入学生个人信息结构体中
}Student;
```

在主函数访问时，可以用例如 `stu.birthday.month;`的方法去访问具体的某一项。

3. 结构体变量

（1）在主函数中定义结构体变量，并给结构体变量赋值

```
Student stu1 = {52,"王某",26,150,1998,6,2};
```

（2）输出信息

当结构体变量很多时，在主函数中用 printf 一个个编写太麻烦，代码的重复率太高，这里建议用一个专门输出结构体变量信息的函数，这样就能减少代码的重复率，相当于是一个通用公式，在主函数中直接调用就可以输出对应信息了，但使用前必须要先定义好结构体变量，具体代码如下：

```
void printStudentInfo(Student stu){   //此处的 stu 是结构体变量名的统一格式，前面的
                                        Student 是前面自己定义的数据类型
    printf("学号:%d\t 姓名:%s\t 年龄:%d\t 成绩:%.1f\t 生日:%d-%d-%d\n",
           stu.id, stu.name, stu.age, stu.score, stu.birthday.year,
           stu.birthday.month, stu.birthday.day);
}
```

然后在主函数中调用这个输出变量的函数就可以了，具体代码如下：

```
int main(){
    printStudentInfo(stu1);  //使用这个函数的前提是自己要先定义好变量
    return 0;
}
```

注意：上述输出函数的调用属于**值传递**，本质是数据拷贝，运行效率相对较低，负担较重，因此推荐使用结构体指针来减小运行负担，该知识点下方即将讲解。

4. 结构体指针

继续以上述学生个人信息这个例子展开，在调用输出函数时，使用结构体指针会减小很多运行负担，其本质是利用地址进行输出，提高代码效率。具体代码如下：

```
void printStudentInfo(Student* pStu){  //先对这个函数的变量定义成指针变量
    printf("学号:%d\t 姓名:%s\t 年龄:%d\t 成绩:%.1f\t 生日:%d-%d-%d\n",
           stu->id,stu->name,stu->age,stu->score,stu->birthday.year,stu->birth
day.month,stu->birthday.day);          //此时各数据要用指针操作，故要用“箭头”表示
}
int main(){
    Student* pStu = &stu1; //定义指针变量 pStu，用来存放变量 stu1（数据类型为 Student）的指针（地址）
    printStudentInfo(pStu);  //使用这个函数的前提是自己要先定义好变量
    pStu = &stu2;   //将 stu2 的地址赋给 pStu（这里只是举例，只有定义好了 stu2 才能用）
    printStudentInfo(pStu);  //此时输出的结果为 stu2 的信息
    return 0;
}
```

5. 结构体数组

继续以上述学生个人信息的例子来展开，利用结构体数组的方式来实现数据输出（**前提条件：**结构体已定义好），具体代码如下：

```
void printStudentInfo(Student* pStu, int len){    //* pStu 也可写成 pStu[]
    for(int i = 0; i < len; i++){
        printf("学号:%d\t 姓名:%s\t 年龄:%d\t 成绩:%.1f\t 生日:%d-%d-%d\n",
    (pStu+i)->id,(pStu+i)->name,(pStu+i)->age,(pStu+i)->score,(pStu+i)->birthd
ay.year,(pStu+i)->birthday.month,(pStu+i)->birthday.day);
    }   //(pStu+i)也可写成 pStu[i]，当用 pStu[i]时，后面的“箭头”要改成“点”
}
int main(){
        Student students[]={    //以下是 3 个学生的信息
            {52,"王某",26,150,1998,6,2},
            {53,"李某",22,133,1998,3,8},
            {54,"陈某"21,128.5,1998,1,9}
        };
    printStudentInfo(students, 2);    //输出前 2 个学生的信息
    printStudentInfo(students, sizeof(students)/sizeof(students[0]));
    //用计算式巧妙地输出所有学生的信息
    return 0;
}
```

6. 位域（位段）　（注意：位域在网络协议的开发中应用得比较多）

位域是指信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用一位二进位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为“位域”或“位段”。所请“位域”是把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

下面通过几段代码来更深刻地认识位域。

【代码 1】

```
typedef struct Test{
    char a : 1;
    char b : 2;
    char c : 1;
}Test;

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

左边这段代码中，给这几个变量定义了位域，所定义的数字代表比特位，那么如左边代码所示，a、b、c 共占 4 个比特位，此时计算出的结构体容量应为 1 字节（8 个比特位），注意，最小容量不能小于变量数据类型本身的大小；当不定义位域时，计算出的结果是 3 字节（24 个比特位）。

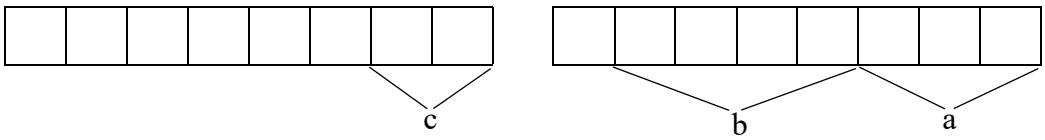
通过以上结果，也就是说，只要定义的位域大小是小于等于 1 字节（8 位），计算出的结果都是 1 字节。

【代码 2】

```
typedef struct Test{
    char a : 3;
    char b : 4;
    char c : 2;
}Test;
int main(){
    printf("%d\n", sizeof(Test));
    return 0;
}
```

左边这段代码中， a、b、c 共占 9 个比特位，此时计算出的结构体容量应为 2 字节（16 个比特位）。

注意：在代码 2 中，需要开辟 2 个字节的空间，但是在存放数据时应是下图这样。



上图这样的存储方式说明了由于 c 所需要的存储空间是 2 个比特位，那么就算第 1 个字节没有存完，也不能再使用了，这时需要从第 2 个字节开始存。因此，位域不能跨字节存储，这是位域的第一个限制条件。

【代码 3】

```
typedef struct Test{
    char a : 1;  //开辟 1 个字节空间（字节对齐，再加 3 个字节）
    int  b : 1;  //开辟 4 个字节空间
}Test;
int main(){
    printf("%d\n", sizeof(Test));
    return 0;
}
```

在上面这段代码 3 中，需要开辟的空间是 8 个字节（64 位），原因是位域不能跨数据类型存储，由字节对齐额外开辟的那 3 个字节空间无法用来存储另一个变量的数据。这是位域的第二个限制条件。示意图如下：

上面就是 char 类型变量 a 开辟的 4 字节空间，只能用来存放 a 变量（1 个比特位），而 int 类型

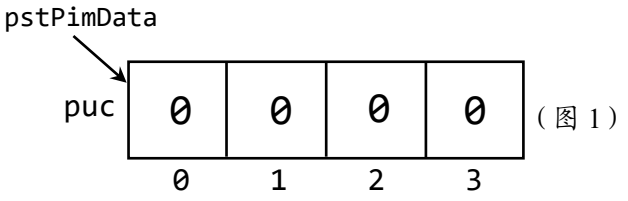
的变量 b 就要存到 int 所开辟的那 4 个字节空间中了。

综上所述，位域有**两个**重要的**限制条件**：①位域不能跨字节存储；②位域不能跨数据类型存储。

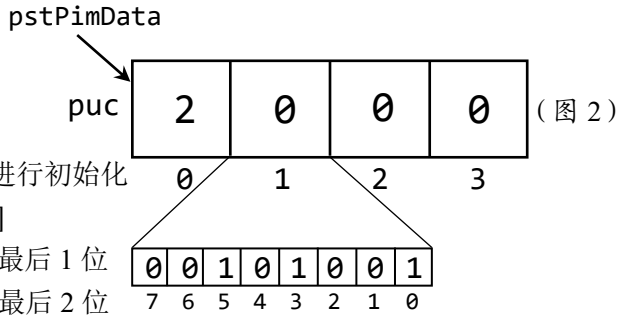
下面通过一道华为的笔试题来更深刻得学习位域的知识：

题目：求出下面代码的输出结果。

```
unsigned char puc[4];
struct tagPIM{
    unsigned char ucPim1;
    unsigned char ucData0 : 1;
    unsigned char ucData1 : 2;
    unsigned char ucData2 : 3;
}*pstPimData;

pstPimData = (struct tagPIM*)puc;
memset(puc,0,4);    //用 memset 函数对数组进行初始化
pstPimData->ucPim1 = 2;  //将 2 存入 puc[0]
pstPimData->ucData0 = 3; //0000 0011 保留最后 1 位
pstPimData->ucData1 = 4; //0000 0100 保留最后 2 位
pstPimData->ucData2 = 5; //0000 0101 保留最后 3 位
printf("%02x %02x %02x %02x\n",puc[0], puc[1], puc[2], puc[3]);
return 0;
```



（图 1）



（图 2）

解析：此结构体总共需要开辟 2 个字节（16 个比特位），为 ucPim1 开辟 1 个字节，为位域再开辟 1 个字节，将 pstPimData 结构体指针指向此数组，并将数组初始化（如图 1），将 2 存入 puc[0]中，因为 ucData0、ucData1、ucData2 的位域分别是 1、2、3，所以存入 puc[1]时，保留 ucData0 所赋值的最后 1 位，保留 ucData1 所赋值的最后 2 位，保留 ucData2 所赋值的最后 3 位，其余没填满的位用 0 补充（如图 2），注意在存入 puc[1]时从低位依次向高位存，因此，puc[0]中的值是 2，puc[1]中的值是 41（十进制），puc[2]和 puc[3]的值都为 0，题目要求输出的结果为两位十六进制数，故最后的输出结果是 02 29 00 00。

三、将 txt 文件中的数据读(写)入指定变量中

1. 读文件

示例代码如下：

```
#include <stdio.h>
int main(){
  FILE* fp;
  char ch;
  fp = fopen("D:\wenjian\file.txt","r"); //若 txt 文件在.c 目录下则不用写出具体路径
  if (fp == NULL){
    printf("无法打开文件\n"); //也可写成 perror("fopen");
    return 1;
  }
  while ((ch = fgetc(fp))!=EOF){ //用 fgetc 函数逐个字符地读取文件内容，直到文件结尾(EOF)
    printf("%c",ch);       //输出每个字符
  }
  fclose(fp);      //关闭文件
  return 0;
}
```

2. 写文件

示例代码如下：

```
#include <stdio.h>
int main(){
  //将以下内容写入到 txt 文件中去
  int num = 1;
  char name[] = "张某";
  int age = 25;
  float height = 182.5;

  FILE* f;
  f = fopen("D:\wenjian\file.txt","w"); //若 txt 文件在.c 目录下则不用写出具体路径
  if(f == NULL){
    printf("无法打开文件\n"); //也可写成 perror("fopen");
    return 1;
  }else{
    fprintf(f,"%d, %s, %d, %.2f",num,name,age,height);//写入数据
    fclose(f);
    puts("写入成功");
  }
  return 0;
}
```

注意：当想输出**汉字**或**非单英文字符**时，须定义成数组或指针，值必须用双引号，输出代码用%s;
但若不定义成数组或指针，就只能输出单个英文字母，值必须用单引号，输出代码用%c。

四、形参与实参

1. 形参

形式参数是指函数名后括号中的变量，形参只有在函数调用过程中才实例化（分配内存单元），函数调用之前，形参还未创建（不占内存），形参实例化之后其实相当于实参的一份临时拷贝，函数调用结束，形参生命周期结束，被销毁（内存释放）。因此形式参数只在函数中有效。

形参必须指定类型，只能是简单变量或数组，不能是常量或表达式。

2. 实参

实参是指在调用有参函数时，函数名后面括号中的实际参数，是我们真实传给函数的参数，实参可以是：常量、变量、表达式、函数等。无论实参是何种类型的量，在进行函数调用时，它们都必须有确定的值，以便把这些值传送给形参；若实参是数组名，则传递的是数组首地址。

3. 两者的关系

形参与实参类型一致，个数相同，顺序相同。若形参与实参的类型不一致，自动按形参类型转换——函数调用转换。

实参对形参的数据传送是值传递，也是单向传递，当被调函数的形参发生变化时，并不改变主调函数实参的值。形参和实参占据的是不同的存储单元。

五、线性表

1. 定义

由 $n(n\geq 0)$个数据特性相同（同类数据元素组成）的元素构成的有限序列，称为**线性表**。因此，线性表的特性有同一性、有限性、有序性。当 $n=0$ 时称之为**空表**，而把线性表中元素的个数 $n(n\geq 0)$ 定义为线性表的长度，对于非空线性表或线性结构，具有如下特点：

①存在唯一的一个被称作“第一个”的数据元素。

②存在唯一的一个被称作“最后一个”的数据元素。

③除第一个元素之外，结构中的每个数据元素均只有一个前驱。

④除最后一个元素之外，结构中的每个数据元素均只有一个后继。

2. 线性表的顺序表示

（1）**顺序表的定义：**用一组地址连续的存储单元依次存储线性表的数据元素，这种表示被称为线性表的顺序存储结构或顺序映像。通常，称这种存储结构的线性表为顺序表。

（2）**顺序表的特点：**逻辑上相邻的数据元素，其物理位置也是相邻的。

（3）**顺序表的性质：**①是一种随机存取的存储结构；②存储密度高；③扩展容量不方便；④插入、删除数据元素不方便

（4）**顺序表的实现：**静态分配和动态分配

【静态分配】使用静态数组来实现，大小一旦确定就无法改变。

静态分配
- 结构体
  - data 数据
  - length 表长(length <= MaxSize)
- 操作

【动态分配】使用动态数组来实现（要用指针），容量大小可以改变。

动态分配
- 结构体
  - data 数据
  - length 表长(length <= MaxSize)
  - L->MaxSize 最大表长(可以用malloc函数扩展)
- 操作

注意：

①用 malloc()分配空间，申请一片连续的存储空间，还可扩展空间，核心代码如下：

```
L->data=(Elemtype*)malloc((L->maxsize+len)*sizeof(Elemtype));
```

②用 free()回收空间，分配了新空间后，需移动元素到新空间，并释放旧空间；或者当不再需要顺序表时，务必调用 DestroyList 函数释放其空间（系统会自动释放指针变量 L，但不会自动释放 L 所指向的存储空间，可能造成内存泄漏），代码段如下：

```
void DestroyList(SqList* &L){
    free(L);
}
```

③当为指针变量时，用->表示；当为普通变量时，用.表示。

（5）**顺序表的示例代码：**

【预编译内容】

```
//定义顺序表最大长度和动态数组初始默认的最大容量
#define MaxSize 10    //静态顺序表的最大长度（举例定为 10）
#define InitSize 10   //动态顺序表的初始最大长度（举例定为 10）
//需要包含的头文件
#include <stdio.h>
#include <stdlib.h>
```

```
//C 语言自定义 bool 变量（宏定义）
#define bool char
#define true 1
#define false 0
//自定义数据元素的数据类型
typedef int Elemtype;  //实质是给基本数据类型取别名，格式形如：typedef 原名 新名;
```

【结构体】

```
/* 顺序表数据元素结构体（静态分配）*/
typedef struct SqList{
    Elemtype data[MaxSize];  //用静态的“数组”存放数据元素
    int length;              //顺序表的当前长度
}SqList;                     //顺序表的类型定义

/* 顺序表数据元素结构体（动态分配）*/
typedef struct SeqList{
    Elemtype* data;     //指示动态分配数组的指针
    int length;         //顺序表的当前长度
    int maxsize;        //顺序表的最大容量
}SeqList;               //顺序表的类型定义
```

【基本操作：初始化】

```
/* 初始化一个静态顺序表 */
bool InitsqList(SqList* L){
    for(int i = 0; i < MaxSize; i++){
        L->data[i] = 0;  //将所有数据元素设为默认初始值
    }                    //本步骤其实可以省略，之所以赋初值是因为内存中会有遗留的“脏
                           数据”，但是常规操作下，用户无法访问大于当前表长的数据元素，
                           所以可以在需要时再赋值
    L->length = 0;       //顺序表初始长度设为 0
    return true;
}

/* 初始化一个动态顺序表 */
bool InitseqList(SeqList* L){
    L->data = (Elemtype*)malloc(sizeof(Elemtype) * InitSize);
    L->length = 0;      //表长和默认最大长度初始化
    L->maxsize = InitSize;
    return true;
}
```

【基本操作：插入】

```
/* 顺序表的元素按位插入（静态分配） */
bool SqListInsert(SqList* L, int i, Elemtype e){
    //[1]判断插入操作是否合法
    if(i < 1 || i > L->length + 1){
        printf("The position of the element to be inserted is invalid\n");
        return false;
    }
    if(L->length >= MaxSize){
        printf("This sequence table is full!\n");
        return false;
    }
```

```
    //[2]将第 i 位以及第 i 位后面的所有元素都后移一位
    for(int j = L->length;j >= i;j--){
        L->data[j] = L->data[j-1];
    }
    L->data[i -1] = e;  //[3]将新元素插入到正确位置(注意区分数组下标和元素位序的排列)
    L->length++;        //[4]表长加一
    return true;        //[5]返回 true，插入成功
}
```

/* 顺序表的元素按位插入（动态分配） */ （注意：动态的代码与静态同原理!）

```
bool SeqListInsert(SeqList* L, int i, Elemtype e){
    //[1]判断插入操作是否合法
    if(i < 1 || i > L->length + 1){
        printf("The position of the element to be inserted is invalid\n");
        return false;
    }
    if(L->length >= L->maxsize){
        printf("This sequence table is full!\n");
        return false;
    }
    //[2]将第 i 位以及第 i 位后面的所有元素都后移一位
    for(int j = L->length; j >= i; j--){
        L->data[j] = L->data[j-1];
    }
    L->data[i -1] = e;  //[3]将新元素插入到正确位置(注意区分数组下标和元素位序的排列)
    L->length++;        //[4]表长加一
    return true;        //[5]返回 true，插入成功
}
```

【基本操作：删除】

/* 顺序表的元素按位删除（静态分配） */

```
//将顺序表 L 的第 i 位元素删除，并把删除的元素的值返回给 e
bool SqListElemDelete(SqList* L, int i, Elemtype* e){
    //[1]判断删除操作是否合法
    if(i < 1 || i > L->length + 1){
        printf("The position of the element to be deleted is invalid\n");
        return false;
    }
    if(L->length <= 0){
        printf("This sequence table is empty!\n");
        return false;
    }
    *e = L->data[i-1];                  //[2]将要删除的元素值赋给 e
    for(int j = i; j < L->length; j++){
          L->data[j-1] = L->data[j];  //[3]将第 i 位后面的元素都往前移 1 位
    }
    L->length--;  //[4]表长减 1
    return true;  //[5]返回 true，删除成功
}
```

/* 顺序表的元素按位删除（动态分配） */

```
bool SeqListElemDelete(SeqList* L, int i, Elemtype* e){
    //[1]判断删除操作是否合法
    if(i < 1 || i > L->length + 1){
        printf("The position of the element to be deleted is invalid\n");
        return false;
```

```
    }
    if(L->length <= 0){
        printf("This sequence table is empty!\n");
        return false;
    }
    *e = L->data[i-1];                  //[2]将要删除的元素值赋给 e
    for(int j = i; j < L->length; j++){
          L->data[j-1] = L->data[j];  //[3]将第 i 位后面的元素都往前移 1 位
    }
    L->length--;  //[4]表长减 1
    return true;  //[5]返回 true，删除成功
}
```

【基本操作：查找】

```
/* 静态顺序表按值查找元素，并返回其位序 */
int SqListLocElem(SqList L, Elemtype e){
    for(int i = 0;i < L.length; i++){
        if(L.data[i] == e){
            return i+1;
        }
    }
    return 0;
}

/* 动态顺序表按值查找元素，并返回其位序 */
int SeqListLocElem(SeqList L, Elemtype e){
    for(int i = 0;i < L.length; i++){
        if(L.data[i] == e){
            return i+1;
        }
    }
    return 0;
}
```

【基本操作：动态顺序表的扩展容量】

```
bool IncreaseSize(SeqList* L, int len){
    Elemtype* p = L->data; //[1]生成指向扩展前那个动态顺序表存储空间的指针，将原本的
                                数据值存入数据寄存器中
    L->data = (Elemtype*)malloc(sizeof(Elemtype)*(L->maxsize + len)); //[2]扩展
    for(int i = 0; i < L->length; i++){
      L->data[i] = p[i];  //[3]将之前存入指针数据寄存器中的数据值赋给新的空间，这里要
                            注意的是，指针本身可被作为数组来使用
    }
    L->maxsize += len;    //[4]L->maxsize = L->maxsize + len 的简写方式
    free(p);              //[5]释放原来的(旧)存储空间
    return true;          //[6]返回 true，扩展空间成功
}
```

【输出】

```
/* 输出静态顺序表 */
bool SqListPrint(SqList L){
    if(L.length == 0){   //判空
        printf("This consequence table is empty!\n");
        return false;
   }
```

```
   printf("SqList:\n");
   for(int i = 0; i < L.length; i++){
      printf("%d--->",L.data[i]);
   }
   printf("end\n");
   return true;
}

/* 输出动态顺序表 */
bool SeqListPrint(SeqList L){
//以下代码跟静态顺序表可完全同理
   if(L.length == 0){   //判空
        printf("This consequence table is empty!\n");
        return false;
   }
   printf("SeqList:\n");
   for(int i = 0; i < L.length; i++){
      printf("%d--->",L.data[i]);
   }
   printf("end\n");
   return true;
}

【函数调用】
int main(){
    SqList L1;    //定义结构体变量（生成静态顺序表 L1）
    SeqList L2;   //定义结构体变量（生成动态顺序表 L2）

    InitSqList(&L1);   //初始化静态顺序表 L1
    InitSeqList(&L2);  //初始化动态顺序表 L2

    SqListInsert(&L1,1,8);  //在静态顺序表 L1 中的第 1 个位置插入 8
    SqListInsert(&L1,2,4);  //在静态顺序表 L1 中的第 2 个位置插入 4
    SqListInsert(&L1,3,5);  //在静态顺序表 L1 中的第 3 个位置插入 5

    SeqListInsert(&L2,1,4); //在动态顺序表 L2 中的第 1 个位置插入 4
    SeqListInsert(&L2,2,6); //在动态顺序表 L2 中的第 2 个位置插入 6
    SeqListInsert(&L2,3,7); //在动态顺序表 L2 中的第 3 个位置插入 7

    int e;
    SeqListElemDelete(&L2, 2, &e);  //删除动态顺序表 L2 中的第 2 个元素，将其值赋给 e

    //输入要查找的元素，在静态顺序表 L1 中利用函数输出要找的元素在第几位
    int value;
    printf("请输入要查找的元素：");
    scanf("%d", &value);
    printf("你要找的元素%d 所在的位置是第%d 位\n", value, SqListLocElem(L1, value));

    SqListPrint(L1);    //输出静态顺序表 L1
    SeqListPrint(L2);   //输出动态顺序表 L2

    IncreaseSize(&L2, 20);   //扩展存储容量（在原来的基础上加 20 的空间）
    printf("L2.maxsize:%d\n",L2.maxsize); //查看新的存储容量

    return 0;
}
```

3. 线性表的链式表示

（1）线性表链式存储结构的特点：

用一组任意的存储单元存储线性表的数据元素（**注：**这组存储单元可以是连续的，也可以是不连续的）。

（2）线性表链式存储结构的一些概念：

①为了表示每个数据元素 a_i 与其直接后继数据元素 a_{i+1} 之间的逻辑关系，对数据元素 a_i 来说，除了存储其本身的信息之外，还需存储一个指示其直接后继的存储位置信息（直接后继的地址），这两部分信息所组成的数据元素 a_i 的存储映像称为**结点**。

②**结点**包括两个域：数据域（存储该结点数据信息）和指针域（存储直接后继的地址）

③n 个结点链接成一个链表，这个链表就是线性表的链式存储结构。

I. **单链表：**当链表的每个结点中只包含一个指针域时，称该链表为**单链表**或**线性链表**。

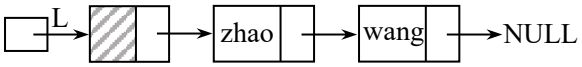
II. **双向链表：**当链表的每个结点中包含两个指针域，称该链表为**双向链表**，双向链表的两个指针域，一个指向直接后继，一个指向直接前驱。

III. **循环链表：**当链表的最后一个结点的指针域指向头结点，整个链表形成一个环，此时称该链表为**循环链表**。循环链表又分单向循环链表和双向循环链表。

④若**头结点**的指针域为 NULL 时，该链表为**空表**。

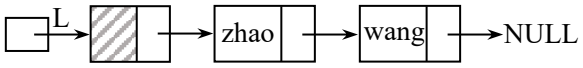
⑤链表不一定必须有头结点。

⑥下图以单链表为例，对**首元结点**、**头结点**、**头指针** 3 个容易混淆的概念加以说明：

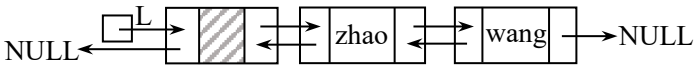


上图是一个单链表的逻辑状态，**首元结点**是指该表中存储第一个数据元素“zhao”的结点；**头结点**是在首元结点之前附设的一个结点，其指针域指向首元结点，头结点的数据域可以不存储任何信息，也可存储与数据元素类型相同的其它附加信息，比如当数据元素为 int 类型时，头结点的数据域中可存放该线性表的长度；**头指针**是指向链表中第一个结点的指针，若链表设有头结点，则头指针所指结点为线性表的头结点，若链表不设头结点，那头指针所指结点为该线性表的首元结点。

（3）链表的图示：



（图 1：非循环单链表）



（图 2：非循环双链表）

（4）单(向)链表的示例代码：

【预编译内容】

```
//需要包含的头文件
#include <stdio.h>
#include <stdlib.h>
//C 语言自定义 bool 变量（宏定义）
#define bool char
#define true 1
#define false 0
//自定义数据元素的数据类型
typedef int Elemtype;  //实质是给基本数据类型取别名，格式形如：typedef 原名 新名;
```

【结构体】

```
typedef struct LNode{
    Elemtype data;      //结点的数据域，其类型为自定义类型 Elemtype（通用类型标识符）
    struct LNode* next; //结点的指针域，其类型为指向结点的指针类型 LNode*
}LNode,*LinkList;
```

说明：

①上述结构体代码段中，为提高程序的可读性，在此对同一结构体指针类型起了两个名称，分别为 LinkList 与 LNode*（视为数据类型），两者本质上是等价的。通常习惯上用 LinkList 定义单链表，强调定义的是某个单链表的头指针；用 LNode*定义指向单链表中任意结点的指针变量。例如：若定义 LinkList L，则 L 为单链表的头指针；若定义 LNode* p，则 p 为**指向**单链表中某个结点的指针，用*p 代表**该结点**。当然也可以使用定义 LinkList p，这种定义形式完全等价于 LNode* p。

②单链表是由表头指针唯一确定的，因此单链表可以用头指针的名字来命名。若头指针名为 L，则简称该链表为表 L。

③注意区分**指针变量**和**结点变量**两个不同的概念，若定义 LinkList p 或 LNode* p，则 p 为指向某结点的指针变量，存放该结点的地址；而*p 为对应的结点变量，表示该结点的名称。

【初始化】（**注意：如果不需要头结点，可以省略此步**）

```
//方法 1：单链表的初始化（创建头结点）
bool InitLinkList(LinkList L){
    L->data = 0;     //头结点的数据域用来存表长
    L->next = NULL;
}
int main(){
    LinkList L;       //L 是头指针，也是该单链表的表名
    L = (LNode*)malloc(sizeof(LNode)); //产生头结点，并使 L 头指针指向此头结点
    InitLinkList(L);
    return 0;
}
```

```
//方法 2：单链表的初始化（创建头结点）
LNode* InitLinkList(){
    LNode* L = (LNode*)malloc(sizeof(LNode));  //产生头结点
    L->data = 0;   //头结点的数据域用来存表长
    L->next = NULL;
    return L;
}
int main(){
    LinkList L;          //L 是头指针，也是该单链表的表名
    L = InitLinkList(); //使 L 头指针指向头结点
    return 0;
}
```

注意：此时需要注意，由上述这段代码，初始化创建了头结点，并且这个带头结点的单链表被命名为 L，那么在后续操作中可以直接使用 L 这个链表名称在各函数中，但是如果要用到不带头结点的操作，就需要注意更改各函数中所使用的链表名称，以免混淆造成程序运行错误。

【判断单链表是否为空表】

下面的代码是针对**带头结点**的单链表：

```
bool Empty(LinkList L) {
      if(L->next==NULL)
          return true;
      return false;
```

```
//main 函数中测试 bool c = Empty(L); printf("%d",c);返回 1
}
```

【基本操作：求单链表的表长】

（一）计算带头结点的单链表（注意：表长是从首元结点开始计算的！不包括头结点！）

```
int ListLength(LinkList L){
    LinkList p;   //用于指向除头结点以外的结点
    int count=0;
    p=L->next;
        while(p){ //遍历整个单链表
          count++;
          p=p->next;
        }
    return count;
 //测试：printf("%d",ListLength(L));
}
```

（二）计算不带头结点的单链表

```
int ListLength(LinkList Head) {  //此时的表名已不再是有头结点的那一个了
    int count = 0;
    LinkList p;
    p = Head;    //指针 p 指向头指针 Head 所指的位置
    while (p) {  //遍历整个单链表
        count++;
        p = p->next;
    }
    return count;
    //测试：printf("%d",ListLength(L));
}
```

【基本操作：插入】

（一）单链表的头插法

第Ⅰ种情况：该单链表**带头结点**

（特别注意：这里所输入的插入个数是不将头结点计入的，而是从首元结点开始计数！）

```
bool InsertLinkList_H(LinkList L) {
    LNode* NewNode;
    int e,n;
    printf("输入要插入的个数：");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        NewNode = (LNode*)malloc(sizeof(LNode));
        printf("输入要插入的值：");
        scanf("%d", &e);
        NewNode->data = e;
        NewNode->next = L->next;
        L->next = NewNode;
        L->data++;    //每增加一个结点，头结点的数据域中的表长信息加 1
}}
```

第Ⅱ种情况：该单链表**不带头结点（重难点）**

注意事项：

①要使用这个函数就不用初始化来创建头结点。

②在写这个代码时，容易出现结果可能不如预期的情况。这是因为在 C 语言中，函数参数默认是通过值传递的，意味着当你将 Head 作为参数传递给 InsertLinkList_H_NoHeadNode 函数时，它实际上接收的是 Head 指针的一个副本。因此，函数内部对 Head 指针的任何修改（例如：将其指向新

的结点）不会反映到原始的 Head 指针上。要解决这个问题有两个思路，一个是使用二级指针，也就是通过指针的指针（即 LinkList*或 LNode**）传递 Head 指针的地址，这样函数就可以直接修改原始 Head 指针的值了；另一个是返回新的头指针。下面对这两种解决思想进行代码说明。

③一旦需要对不带头结点的链表进行操作时，都需要用到下面两种思路来解决。

思路一：使用二级指针

```
bool InsertLinkList_H_NoHeadNode(LinkList* HeadPtr) { //定义头指针 Head 的指针
    int e,n;
    LNode* NewNode;
    printf("输入要插入的个数：");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        NewNode = (LNode*)malloc(sizeof(LNode));  //创建新结点
        if (NewNode == NULL){
            printf("malloc failed\n");
            return;
        }
        printf("输入待插入结点数据域中的数据：");
        scanf("%d", &e);
        NewNode->data = e;
        NewNode->next = *HeadPtr;
        *HeadPtr = NewNode;
    }
}
int main() {   //调用
    LinkList Head = NULL;
    InsertLinkList_H_NoHeadNode(&Head);  //注意这里传递头指针 Head 的地址
    PrintLinkList(Head);
    return 0;
}
```

思路二：返回新的头指针

```
//将此函数的返回值类型定义为 LinkList 结构体指针类型
LinkList InsertLinkList_H_NoHeadNode(LinkList Head) {
    int e, n;
    LNode* NewNode;
    printf("输入要插入的个数：");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        NewNode = (LNode*)malloc(sizeof(LNode));  //创建新结点
        if (NewNode == NULL) {
            printf("malloc failed\n");
            return;
        }
        printf("输入待插入结点数据域中的数据：");
        scanf("%d", &e);
        NewNode->data = e;
        NewNode->next = Head;
        Head = NewNode;
    }
    return Head;  //此处一定要返回 Head 头指针
}
int main() {
    LinkList Head = NULL;
    //把 Head 指针的返回值（旧头指针的地址）赋给新头指针 NewHead
    LinkList NewHead = InsertLinkList_H_NoHeadNode(Head);
```

```
    PrintLinkList(NewHead);  //以后在调用函数时，要传入新头指针 NewHead 的值进去
    return 0;
}
```

（二）单链表的尾插法

第Ⅰ种情况：该单链表**带头结点**

```
bool InsertLinkList_R(LinkList L) {
    int n;
    LinkList Tail=L;
    printf("你要插入的元素个数为：");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        LNode* new = (LNode*)malloc(sizeof(LNode));
        printf("请输入第%d 个元素所要插入的元素值：",i+1);
        scanf("%d", &new->data);
        new->next = NULL;
        Tail->next = new;
        Tail = new;
        L->data++;
    }
    return true;
}
```

第Ⅱ种情况：该单链表**不带头结点（重难点）**

略（跟单链表不带头结点的头插法同理）

（三）单链表的按元素位置插入法

（特别注意：此方法是针对带头结点的单链表，插入元素的位置是不考虑头结点来计数的）

```
bool ListInsert(LinkList L) {
    LinkList p, q;
    int i, e;     //定义需要插入元素的实际位置 i、需要插入的数据 e
    p = L;        //指向头结点
    //先新建并初始化好需要插入的结点，用指针 q 指向它
    q = (LNode*)malloc(sizeof(LNode));   //新建待插入结点，并用 q 指针指向它
    printf("你插入的数据是：");
    scanf("%d", &e);
    q->data = e;       //将待插入结点的数据赋给其数据域中
    q->next = NULL;
    printf("你想插入的位置是：");
    scanf("%d", &i);

    for (int j = 0; p && j < i - 1; j++) {
        //循环条件：1≤i≤n+1 且 j<i-1，其中 p 代表 1≤i≤n+1，n 为链表中结点的总个数
        if (i > ListLength(L) + 1){
            printf("你想插入的位置不合法\n");
            return false;  //如果插入的位置大于链表的尾结点后面的一个位置,则返回错误
        }
        p = p->next;
    }
    //将待插入结点的前后结点与之形成链式关系
    q->next = p->next;      //接后继结点
    p->next = q;            //接前驱结点
    L->data++;              //头结点的数据域中所存放的表长信息加 1
    return true;
}
```

【基本操作：查找】

（一）单链表的按位查找法（带头结点）

```
LNode* GetElem_Location(LinkList L, int n){
     //[1]判断要查找的元素位置 n 的合法性
     if (n == 0) {
        printf("The LinkList's element you are looking for does not exist!");
        return L;
    }
    if (n<1 || n>L->data) {
        printf("The LinkList's element you are looking for does not exist!");
        return NULL;
    }
    //[2]查找数据元素
    LNode* p = L;
    for (int i = 0;i < n;i++) {
        p = p->next;
    }
    return p;
}
int main(){
    printf("第 6 个元素是%d\n",GetElem_Location(L,6)->data);
    return 0;
}
```

由于上面这段代码的设计较为机械，灵活性不大，因此可对上述代码进行优化，优化后如下：

```
bool GetElem_Location(LinkList L) {
    int n;
    printf("你要查找的元素位置是：");
    scanf("%d", &n);
    //[1]判断要查找的元素位置 n 的合法性
    if (n == 0) {
        printf("The LinkList's element you are looking for does not exist!");
        return false;
    }
    if (n<1 || n>L->data) {
        printf("The LinkList's element you are looking for does not exist!");
        return false;
    }
    //[2]查找数据元素
    LNode* p = L;
    for (int i = 0;i < n;i++) {
        p = p->next;
    }
    printf("你要找的第%d 位元素是%d\n", n, p->data);
    return true;
}
```

（二）单链表的按值查找法（带头结点）

```
bool GetElem_Value(LinkList L) {
    Elemtype e;
    printf("你找的元素是");
    scanf("%d",&e);
    if (!L->next) {  //如果 L->next 为空
        printf("This LinkList is empty!\n");
        return false;
    }
    LNode* p = L->next;
    int count = 1;
```

```
    while (p && p->data != e) {
        p = p->next;
        count++;
    }
    if (p) {
        printf("你所找的元素%d 所在的位置是第%d 位\n", e, count);
        return true;
    }else{
        printf("The LinkList's element you are looking for does not exist!\n");
           return false;
    }
}
```

【基本操作：结点按位删除】（**注意：**该算法只针对带头结点的情况，此操作不能删除头结点！）

```
bool DeleteElem_Location(LinkList L) {
    LinkList p = L;
    int curNum = 1;
    int n;
    printf("请输入你要删除元素的位置：");
    scanf("%d", &n);
    if (n<1 || n>L->data) {
        printf("你输入的位置不合法\n");
        return false;
    }
    if (L->next == NULL) {
        printf("你输入的位置不合法\n");
        return false;
    }
    while (p->next) {
        if (curNum == n) {
            LinkList temp = p->next;
            p->next = p->next->next;
            free(temp);
            L->data--;
            return true;
        }
        p = p->next;
        curNum++;
    }
}
```

【输出】

（一）输出带头结点的单链表

```
bool PrintLinkList(LinkList L){
    LNode* p;
    p = L;        //指针 p 指向头结点（将头结点的地址赋给指针变量 p）
    while(p->next){
        p = p->next; //指针 p 向后移一位（指针 p 指向的结点的指针域指向下一个结点的地址赋给指针变量 p）
        printf("%d-->",p->data);
    }
    printf("NULL\n");
    return true;
}
```

（二）输出不带头结点的单链表

```
bool PrintLinkList(LinkList Head) {
    LNode* m;
```

```
    m = Head;
    while (m) {   //遍历整个单链表
        printf("%d--->", m->data);
        m = m->next; //指针 m 向后移一位（指针 m 指向的结点的指针域指向下一个结点的地址赋给指针变量 m）
    }
    printf("NULL\n");
    return true;
}
```

【销毁】

```
bool DestroyLinkList(LinkList L) {
    LinkList p = L;
    while (L->data) {
        LinkList temp = p->next;
        p->next = p->next->next;
        free(temp);
        L->data--;
    }
    free(L);
    return true;
}
```

【函数调用】

```
int main(){
    InsetLinkList_H(L);   //对单链表L调用带头结点的头插法函数
    ListInsert(L);        //对单链表L调用按位置插入法函数
    InsertLinkList_H_NoHeadNode(L);   //对单链表L调用不带头结点的头插法函数
    PrintLinkList(L);     //对单链表L调用结果输出函数
    printf("该单链表的表长为%d\n", ListLength(L));//输出该单链表表长（调用求表长函数）
    printf("%d\n", L->data); //输出该单链表表长（输出头结点数据域存放的表长信息）
    GetElem_Value(L);         //按值查找
    GetElem_Location(L);      //按位查找
    DeleteElem_Location(L);   //按位删除（删除头结点之外的结点）
    DestroyLinkList(L);       //销毁单链表
    return 0;
}
```

（5）双(向)链表的示例代码：

【预编译内容】

```
//宏定义
#define true 1
#define false 0
#define bool char
//自定义类型
typedef int ElemType;
```

【结构体】

```
typedef struct DNode {
    struct DNode* prior;
    ElemType data;
    struct DNode* next;
}DNode,*DLinkList;
```

【初始化(创建头结点)】

```
DNode* InitDLinkList() {
```

```
    DNode* Head = (DNode*)malloc(sizeof(DNode));
    assert(Head);
    Head->prior = NULL;
    Head->data = 0;
    Head->next = NULL;
    return Head;
}
```

【基本操作：双链表的头插法】

```
bool HeadInsert(DLinkList Double) {
    int n;
    printf("请输入你要头插的个数为");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        DNode* New= (DNode*)malloc(sizeof(DNode));
        assert(New);
        printf("请输入第%d 个元素的值为", i + 1);
        scanf("%d", &New->data);
        New->next = Double->next;
        New->prior = Double;
        if (Double->next) {
            Double->next->prior = New;
        }
        Double->next = New;
        Double->data++;
    }
    return true;
}
```

【基本操作：双链表的尾插法】

```
bool TailInsert(DLinkList Double) {
    int n;
    printf("请输入你要尾插的个数为");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        DNode* New = (DNode*)malloc(sizeof(DNode));
        assert(New);
        printf("请输入第%d 个元素的值为", i + 1);
        scanf("%d", &New->data);
        DLinkList t;
        t = Double;
        New->next = NULL;
        while (t->next) {
            t = t->next;
        }
        New->prior = t;
        t->next = New;
        Double->data++;
    }
    return true;
}
```

【基本操作：按值删除结点】

```
bool Delete_Value(DLinkList Double) {
        int e;DLinkList m=Double->next;
        printf("输入要删除的值为");
        scanf("%d", &e);
```

```
        while (m) {
            if (m->data == e) {
                if (m->next == NULL) {
                    m->prior->next = m->next;
                }
                else {
                    m->prior->next = m->next;
                    m->next->prior = m->prior;
                }
                free(m);
                Double->data--;
                return true;
            }
                m = m->next;
        }
        printf("未找到你想要删除的结点\n");
        return false;
}
```

【基本操作：按位删除结点】

```
bool Delete_Location(DLinkList Double) {
    int n;
    printf("输入你要删除的元素位序是");
    scanf("%d", &n);
    if (n <= 0 || n > Double->data) {
        printf("你输入的位序不合法\n");
        return false;
    }
    else {
        DLinkList m = Double;
        for (int i = 0;i < n;i++) {
            m=m->next;
        }
        if (m->next == NULL) {
            m->prior->next = m->next;
        }
        else {
            m->prior->next = m->next;
            m->next->prior = m->prior;
        }
        free(m);
        Double->data--;
        return true;
    }
}
```

【基本操作：按位查找(返回该位的值)】

```
bool Search_Location(DLinkList Double) {
    int n;
    printf("输入你要查找的元素位序");
    scanf("%d", &n);
    if (n > Double->data || n <= 0) {
        printf("你输入的元素位序无效\n");
        return false;
    }
    else {
        DLinkList m = Double;
        for (int i = 0;i < n;i++) {
```

```
            m = m->next;
        }
        printf("第%d 位元素的值为%d\n", n, m->data);
        return true;
    }
}
```

【基本操作：按值查找(返回该值的位序)】

```
bool Search_Value(DLinkList Double) {
    int e;
    printf("请输入你要找的元素值为");
    scanf("%d", &e);
    DLinkList m = Double;
    int count = 0;
    while (m) {
        if (m->data == e) {
            printf("该元素值的位序是%d\n", count);
            return true;
        }
        else {
            m = m->next;
            count++;
        }
    }
    printf("没有你要找的元素\n");
    return false;
}
```

【输出整个双链表】

```
bool Print(DNode* Double) {
    DNode* m= Double->next;
    while (m) {
        if (m->next == NULL) {
            printf("%d--->", m->data);
            m = m->next;
        }
        else {
            printf("%d<===>", m->data);
            m = m->next;
        }
    }
    printf("NULL\n");
    return true;
}
```

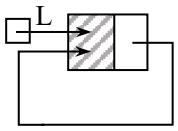
【函数调用】

```
int main() {
    DLinkList D = InitDLinkList();
    HeadInsert(D);
    TailInsert(D);
    Delete_Value(D);
    Search_Location(D);
    Search_Value(D);
    Delete_Location(D);
    Print(D);
    return 0;
}
```

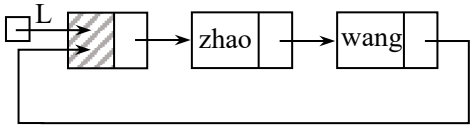
（6）**循环链表的示例代码：**

（**注意：**仅演示部分操作，其它操作自行举一反三即可!）

〔Ⅰ〕**单(向)循环链表**



（图 1：只有头结点的循环单链表）



（图 2：带头结点的非空循环单链表）

【预编译内容】

```
//宏定义
#define true 1
#define false 0
#define bool char
//自定义类型
typedef int ElemType;
```

【结构体】

```
typedef struct CLNode {
    ElemType data;
    struct CLNode* next;
}CLNode,*CLinkList;
```

【初始化(创建头结点)】

```
CLNode* InitCLinkList() {
    CLNode* Head = (CLNode*)malloc(sizeof(CLNode));
    Head->data = 0;
    Head->next = Head;
    return Head;
}
```

【基本操作：循环单链表的头插法】

```
bool HeadInsert(CLinkList L) {
    int n;
    printf("请输入要头插元素的个数为");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        CLNode* New = (CLNode*)malloc(sizeof(CLNode));
        printf("输入第%d 个元素的给定值为",i+1);
        scanf("%d", &New->data);
        New->next = L->next;
        L->next = New;
        L->data++;
    }
    return true;
}
```

【基本操作：循环单链表的尾插法】

```
bool TailInsert(CLinkList L) {
    int n;
    printf("请输入要尾插元素的个数为");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        CLNode* New = (CLNode*)malloc(sizeof(CLNode));
        printf("输入第%d 个元素的给定值为", i + 1);
        scanf("%d", &New->data);
```

```
        New->next = L;
        CLNode* t = L;
        while (t->next != L) {
            t = t->next;
        }
        t->next = New;
        L->data++;
    }
    return true;
}
```

【基本操作：按值删除结点】

```
bool Delete_Value(CLinkList L) {
    int e;
    printf("输入你要删除的值为");
    scanf("%d", &e);
    CLNode* p = L;
    CLNode* m = L->next;
    while (m != L) {
        if (m->data == e) {
            p->next = m->next;
            free(m);
            L->data--;
            return true;
        }
        m = m->next;
        p = p->next;
    }
    printf("未找到你想要删除的结点\n");
    return false;
}
```

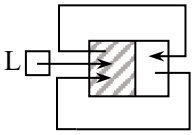
【输出整个循环单链表】

```
bool Print(CLinkList L) {
    if (L->data == 0) {
        printf("该循环单链表中无元素\n");
        return false;
    }
    CLNode* p = L->next;
    while (p != L) {
        printf("%d--->", p->data);
        p = p->next;
    }
    printf("loop\n");
    return true;
}
```

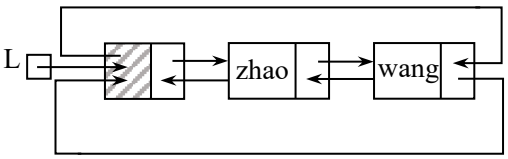
【函数调用】

```
int main() {
    CLinkList L= InitCLinkList();
    HeadInsert(L);
    TailInsert(L);
    Delete_Value(L);
    Print(L);
    printf("该循环单链表的表长为%d", L->data);
    return 0;
}
```

〔Ⅱ〕双(向)循环链表



（图 1：只有头结点的循环双链表）



（图 2：带头结点的非空循环双链表）

【预编译内容】

```
//宏定义
#define true 1
#define false 0
#define bool char
//自定义类型
typedef int ElemType;
```

【结构体】

```
typedef struct CDNode {
    ElemType data;
    struct CDNode* prior;
    struct CDNode* next;
}CDNode,*CDLinkList;
```

【初始化(创建头结点)】

```
CDNode* InitCDLinkList() {
    CDNode* Head = (CDNode*)malloc(sizeof(CDNode));
    assert(Head);
    Head->data = 0;
    Head->prior = Head;
    Head->next = Head;
    return Head;
}
```

【基本操作：循环双链表的头插法】

```
bool HeadInsert(CDLinkList D) {
    int n;
    printf("输入要头插结点的个数为");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        CDNode* New = (CDNode*)malloc(sizeof(CDNode));
        assert(New);
        printf("输入第%d 个结点的元素值为",i+1);
        scanf("%d", &New->data);
        if (D->next != D) {
            New->next = D->next;
            D->next->prior = New;
            New->prior = D;
            D->next = New;
        }
        else {
            New->next = D;
            New->prior = D;
            D->next = New;
            D->prior = New;
        }
        D->data++;
    }
```

```
    return true;
}
```

【基本操作：循环双链表的尾插法】

```
bool TailInsert(CDLinkList D) {
    int n;
    printf("输入要尾插结点的个数为");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        CDNode* New = (CDNode*)malloc(sizeof(CDNode));
        assert(New);
        printf("输入第%d 个结点的元素值为", i + 1);
        scanf("%d", &New->data);
        CDNode* t = D;
        while (t->next != D) {
            t = t->next;
        }
        New->next = D;
        D->prior = New;
        t->next = New;
        New->prior = t;
        D->data++;
    }
    return true;
}
```

【基本操作：按值删除结点】

```
bool Delete_Value(CDLinkList D) {
    int e;
    printf("输入你要删除的结点元素值为");
    scanf("%d", &e);
    CDNode* p = D->next;
    while (p->next != D) {
        if (p->data == e) {
            p->prior->next = p->next;
            p->next->prior = p->prior;
            free(p);
            D->data--;
            return true;
        }
        p = p->next;
    }
    printf("未找到你想要删除的结点\n");
    return false;
}
```

【输出整个循环双链表】

```
bool Print(CDLinkList D) {
    CDNode* m = D->next;
    while (m != D) {
        printf("%d<===>", m->data);
        m = m->next;
    }
    printf("loop\n");
    return true;
}
```

【函数调用】
```
int main() {
    CDLinkList D = InitCDLinkList();
    HeadInsert(D);
    TailInsert(D);
    Delete_Value(D);
    Print(D);
    return 0;
}
```

六、栈（栈是限定仅在表尾进行插入或删除操作的线性表。特点是后进先出。）

1. 顺序栈

（1）定义

顺序栈是指利用顺序存储结构实现的栈，即利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针 top 指示栈顶元素在顺序栈中的位置。

（2）代码实现

【预编译内容】
```
//宏定义
#define MaxSize 100
#define bool char
#define true 1
#define false 0
//自定义数据元素的数据类型
typedef int ElemType;
```

【结构体】
```
typedef struct Stack {
    ElemType data[MaxSize];
    int top;
    int cursize;
}Stack;
```

【基本操作：初始化】
```
bool InitStack(Stack* S) {
    S->top = -1;
    S->cursize = 0;
    return true;
}
```

【判空】
```
bool empty(Stack S) {
    if (S.top == -1) {
        printf("此顺序栈为空\n");
        return true;
    }
    else {
        return false;
}}
```

【基本操作：入(压)栈】
```
bool Push(Stack* S) {
    if (S->top >= MaxSize - 1) {
```

```
        printf("此栈已满，无法压栈");
        return false;
    }
    int n;
    printf("请输入要压栈的元素个数：");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        int e;
        printf("请输入第%d 个要压入的元素值：", i + 1);
        scanf("%d", &e);
        S->data[++S->top] = e;  //先让 S->top 加 1，再把 e 赋值给 S->data[top 的值加 1]
        S->cursize++;
    }
    return true;
}
```

【基本操作：出栈】

```
bool Pop(Stack* S) {
    if (empty(*S)) {   //该处 if 语句条件中的函数参数数据类型要与本 Pop 函数的一致，与
                         empty 函数本来的形参数据类型无关
        printf("由于栈空，无元素可出栈\n");
        return false;
    }
    else {
        ElemType e;
        int n;
        printf("输入要出栈元素的个数为：");
        scanf("%d", &n);
        for (int i = 0;i < n;i++) {
            e = S->data[S->top--];
            S->cursize--;
        }
        return true;
    }
}
```

【计算顺序栈当前长度】

```
bool Stacksize(Stack S) {
    if (S.cursize < 0)
        return false;
    printf("该栈当前长度为%d\n", S.cursize);
    return true;
}
```

【输出整个顺序栈】

```
bool PrintStack(Stack S) {
    for (int i = 0;i < S.cursize;i++) {
        printf("%d\n", S.data[S.top]);
        S.top = S.top--;
    }
    return true;
}
```

【读栈顶元素】

```
bool GetTop(Stack S) {
    if (empty(S)) {
        printf("由于栈空，无元素可出栈\n");
```

```
        return false;
    }
    else {
        printf("栈顶元素为%d", S.data[S.top]);
    }
        return true;
}
```

【函数调用】

```
int main() {
    Stack S;
    InitStack(&S);
    empty(S);
    Push(&S);
    PrintStack(S);
    Pop(&S);
    Stacksize(S);
    GetTop(S);
    return 0;
}
```

2. 链栈

（1）定义

链栈是指采用链式存储结构实现的栈。通常链栈用单链表来表示，链栈的结点结构与单链表的结构相同。

（2）代码实现

【预编译内容】

```
//宏定义
#define bool char
#define true 1
#define false 0
//自定义数据类型
typedef int ElemType;
```

【结构体】

```
typedef struct SNode {
    ElemType data;
    struct SNode* next;
}SNode,*LinkStack;
```

【基本操作：入(压)栈】

（**注意**：该代码的函数形参用到了二级指针，原理与不带头结点的单链表头插法相同!）

```
bool Push(LinkStack* ST) {
    int n;
    printf("输入你要压入链栈的元素个数：");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
        SNode* p = (SNode*)malloc(sizeof(SNode));
        printf("输入第%d 个要压入的元素值：", i+1);
        scanf("%d", &p->data);
        p->next = *ST;
        *ST = p;
    }
    return true;
```

```
}
```

【基本操作：出栈】

```
bool Pop(LinkStack *ST) {
    if (*ST == NULL) {
        printf("该链栈为空，出栈不可用\n");
        return false;
    }
    else {
        int n;
        printf("要出栈元素的个数是");
        scanf("%d", &n);
        for (int i = 0;i < n;i++) {
            LinkStack p = *ST;
            int e = p->data;
            *ST = (*ST)->next;
            free(p);
        }
    }
    return true;
}
```

【计算链栈的长度】

```
bool LinkStackLength(LinkStack S) {
    int n=1;
    SNode* m = S;
    while (m->next) {
        m = m->next;
        n++;
    }
    printf("该链栈的长度为%d\n", n);
    return true;
}
```

【输出整个链栈】

```
bool PrintLinkStack(LinkStack S) {
    LinkStack p = S;    //让指针 p 指向栈顶指针所指的位置
    while (p) {
        printf("%d\n", p->data);
        p = p->next;
    }
    return true;
}
```

【读栈顶元素】

```
bool GetElem_Top(LinkStack S) {
    if (S == NULL) {
        printf("链栈为空，无法读取\n");
        return false;
    }
    else {
        printf("栈顶元素为%d\n", S->data);
        return true;
    }
}
```

【函数调用】

```
int main() {
    LinkStack S=NULL;
    Push(&S);
    Pop(&S);
    LinkStackLength(S);
    PrintLinkStack(S);
    GetElem_Top(S);
    return 0;
}
```

七、队列

1. 循环队列

（1）定义

将向量空间想象为一个首尾相接的圆环，并称这种向量为循环向量。存储在其中的队列称为循环队列。简单来讲，循环队列就把顺序队列尾相连，把存储队列元素的表从逻辑上看成一个环。

（2）引入目的

为了充分利用向量空间，克服“假溢出”的现象。（**注意：**“假溢出”是指数组越界而导致程序的非法操作错误，但队列的实际可用空间并未占满）

（3）引入两个指示标识（可认为起到了指针的作用）

一个指向队头 front，一个指向队尾 rear。在循环队列中，当队列为空时，有 front=rear；而当所有队列空间全占满时，也有 front=rear。所以，为了区别这两种情况，规定循环队列最多只能有 MaxSize−1 个队列元素（也就是空一个位置），当循环队列中只剩下一个空存储单元时，队列就已经满了。因此，队列判空的条件是 front=rear，而队列判满的条件是 front=(rear+1)%MaxSize。

因此，这个空位置主要就是为了用来区分队空与队满情况的。

（4）代码实现

【预编译内容】

```
//宏定义
#define MaxSize 10    //给顺序存储结构的循环队列定义最大容量
#define bool char
#define true 1
#define false 0
//自定义数据类型
typedef int Elemtype;
```

【结构体】

```
typedef struct Queue {
    Elemtype data[MaxSize];
    int front;
    int rear;
}Queue;
```

【初始化】

```
bool InitQueue(Queue* Q) {
    Q->front = Q->rear = 0;
    return true;
}
```

【判断循环队列为满】

```
bool full(Queue* Q) {
    if ((Q->rear + 1) % MaxSize == Q->front) {
        printf("循环队列已满\n");
        return true;
    }
    return false;
}
```

【判断循环队列为空】

```
bool Empty(Queue* Q) {
    if (Q->front == Q->rear) {
        printf("循环队列为空\n");
        return true;
    }
    return false;
}
```

【计算循环队列的长度】

```
int Queue_Length(Queue* Q) {
    int Length = (Q->rear - Q->front + MaxSize) % MaxSize;
    return Length;
}
```

【基本操作：入队】

```
bool Queue_en(Queue* Q) {
    if (full(Q)) {
        return false;
    }
    else {
        int n;
        printf("请输入要入队的个数为");
        scanf("%d", &n);
        if (Queue_Length(Q) + n >= MaxSize) {
            printf("没有足够的空间存你想入队的个数\n");
            return false;
        }
        else {
            for (int i = 0;i < n;i++) {
                printf("输入第%d 个入队的元素值：", i + 1);
                Q->rear=(Q->rear + 1) % MaxSize;
                scanf("%d", &Q->data[Q->rear]);
            }
            return true;
        }}}
```

【基本操作：出队】

```
bool Queue_de(Queue* Q) {
    if (Empty(Q)) {
        printf("此循环队列为空，出队无效");
        return false;
    }
    else {
        int n;
        printf("请输入要出队的个数为");
        scanf("%d", &n);
```

```
        for (int i = 0;i < n;i++) {
            Q->front = (Q->front + 1) % MaxSize;
            int m = Q->data[Q->front];
        }
        return true;
    }}
```

【输出整个循环队列】

```
    bool Print_Queue(Queue Q) {
        int Length = (Q.rear - Q.front + MaxSize) % MaxSize;
        for (int i = 0;i < Length;i++) {
            printf("%d\n", Q.data[Q.rear]);
            Q.rear = (Q.rear - 1) % MaxSize;
        }
        printf("NULL\n");
        return true;
    }
```

【读取循环队列的队头元素】

```
    bool GetElem_front(Queue Q) {
        if (Q.front == Q.rear) {
            printf("该循环队列为空，无法读取队头元素\n");
            return false;
        }
        else {
            printf("该循环队列的队头元素是%d\n", Q.data[(Q.front + 1) % MaxSize]);
            return true;
        }}
```

【读取循环队列的队尾元素】

```
    bool GetElem_rear(Queue Q) {
        if (Q.front == Q.rear) {
            printf("该循环队列为空，无法读取队尾元素\n");
            return false;
        }
        else {
            printf("该循环队列的队尾元素是%d\n", Q.data[Q.rear]);
            return true;
        }}
```

【函数调用】

```
    int main() {
        Queue q;
        InitQueue(&q);
        Queue_en(&q);
        Print_Queue(q);
        Queue_de(&q);
        Print_Queue(q);
        printf("该循环队列的长度为%d\n", Queue_Length(&q));
        GetElem_front(q);
        GetElem_rear(q);
        return 0;
    }
```

2. 链队列

（1）定义

链队列是指采用链式存储结构实现的队列。通常链队列用单链表来表示。一个链队列需要两个分别指示队头和队尾的指针（分别称为头指针和尾指针）才能唯一确定。

（2）代码实现

（注：下面的示例代码是针对添加了头结点的链队列）

【预编译内容】

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
//宏定义
#define bool char
#define true 1
#define false 0
//自定义数据类型
typedef int Elemtype;
```

【结构体】

```
typedef struct LinkQueueNode {
    Elemtype data;
    struct LinkQueueNode* next;
}LinkQueue_Node;

typedef struct {
    LinkQueue_Node* front;
    LinkQueue_Node* rear;
}LinkQueue;
```

【初始化】

```
LinkQueue* InitLinkQueue() {
    LinkQueue_Node* L = (LinkQueue_Node*)malloc(sizeof(LinkQueue_Node));
    assert(L);    //若条件不成立则终止程序
    L->data = 0;
    L->next = NULL;

    LinkQueue* Pointer = (LinkQueue*)malloc(sizeof(LinkQueue));
    assert(Pointer);
    Pointer->front = L;
    Pointer->rear = L;
    return Pointer;
}
```

【求链队列长度】

```
int LinkQueueLength(LinkQueue* P) {
    LinkQueue_Node* p;
    int count = 0;
    p = P->front->next;
    while (p) {
        count++;
        p = p->next;
    }
    return count;
}
```

【基本操作：入队】

```
bool LinkQueue_en(LinkQueue* P) {
    int n;
```

```
    LinkQueue_Node* m = P->rear;
    printf("请输入入队元素的个数为");
    scanf("%d", &n);
    for (int i = 0;i < n;i++) {
       LinkQueue_Node* NewNode=(LinkQueue_Node*)malloc(sizeof(LinkQueue_Node));
       assert(NewNode);
       printf("输入第%d 个元素的值：", i + 1);
       scanf("%d", &NewNode->data);
       NewNode->next = NULL;
       m->next = NewNode;
       m = NewNode;
       P->rear = m;
       P->front->data++;
    }
    return true;
}
```

【基本操作：出队】

```
bool LinkQueue_de(LinkQueue* P) {
    if (P->front->data == 0) {
        printf("该链队列为空，出队无效\n");
        return false;
    }
    else {
        int n;
        printf("请输入要出队的元素个数");
        scanf("%d", &n);
        if (P->front->data - n < 0) {
            printf("你想要出队的个数超限，操作无效!\n");
            return false;
        }
        else {
            for (int i = 0;i < n;i++) {
                LinkQueue_Node* Dele = P->front->next;
                int e = Dele->data;   //用变量 e 来保存队头元素的值
                P->front->next = Dele->next;
                free(Dele);
                P->front->data--;
                if (P->front->data == 0) {
                    P->rear = P->front;
                }
            }
            return true;
    }}}
```

【输出整个链队列】

```
bool PrintLinkQueue(LinkQueue* P) {
    LinkQueue_Node* p = P->front->next;
    while (p) {
        printf("%d--->", p->data);
        p = p->next;
    }
    printf("NULL\n");
    return true;
}
```

【读队头元素】

```
bool GetElem_front(LinkQueue* P) {
    if (P->front->next) {
        printf("队头元素是%d\n", P->front->next->data);
        return true;
    }
    else return false;
}
```

【读队尾元素】

```
bool GetElem_rear(LinkQueue* P) {
    if (P->front->next) {
        printf("队尾元素是%d\n", P->rear->data);
        return true;
    }else return false;
}
```

【函数调用】

```
int main() {
    LinkQueue* P= InitLinkQueue();    //用一个指针变量 P 来存初始化的指针
    //因为主函数定义了指针变量，所以下面传参时直接传入指针变量 P
    LinkQueue_en(P);          //入队
    PrintLinkQueue(P);        //输出入队后的链队列
    printf("该链队列的长度是%d\n",LinkQueueLength(P));    //求链队列长度

    LinkQueue_de(P);          //出队
    PrintLinkQueue(P);        //输出出队后的链队列
    printf("该链队列的长度是%d\n",LinkQueueLength(P));    //求链队列长度

    GetElem_front(P);         //读队头元素
    GetElem_rear(P);          //读队尾元素

    return 0;
}
```

八、串

串有两种基本存储结构，分别为顺序存储和链式存储。但考虑到存储效率和算法的方便性，**因此串多采用顺序存储结构**。

1. 定义

串或字符串是由零个或多个字符组成的有限序列，一般记为 s="$a_1a_2\cdots a_n$"　($n\geqslant 0$)

说明：

①其中，s 是串的名称，用双引号标识的字符序列是串的值；$a_i(1\leqslant i\leqslant n)$可以是字母、数字或其它字符；串中字符的数目 n 称为串的长度。零个字符的串称为空串，其长度为 0。

②串中任意个连续的字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。通常称字符在序列中的序号为该字符在串中的位置。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

③只有当两个串的长度相等，并且各个对应位置的字符都相等时，两个串才相等。

④在各种应用中，空格常常是串的字符集合中的一个元素，因而可以出现在其它字符中间。由一个或多个空格组成的串“ ”称为空格串（**注：**此处不是空串，空串的符号为∅），其长度为串中空格字符的个数。

2. 串的存储结构

（1）串的顺序存储（顺序串）

〔Ⅰ〕串的静态(定长)顺序存储结构

按照预定义的大小，为每个定义的串变量分配一个固定长度的存储区，定义内容如下：

```
#define MAXLEN 255        //串的最大长度
typedef struct{
    char ch[MAXLEN+1];    //存储串的一维数组
    int length;           //串的当前长度
}SString;
```

这种定义方式的静态的，在编译时刻就确定了串空间的大小。而多数情况下，串的操作时以串的整体形式参与的，串变量之间的长度相差较大，在操作中串值长度的变化也较大，这样为串变量设定固定大小的空间不尽合理。

下面来介绍此种存储结构的一系列代码实现：

【宏定义】

```
#define bool char
#define true 1
#define false 0
#define MAXLEN 255   //定义串的最大长度
```

【结构体】

```
typedef struct {
    char ch[MAXLEN + 1];
    char length;
}SString;
```

【初始化】

```
bool InitString(SString* str){
    str->length = 0;
    return true;
}
```

【判空】

```
int StringEmpty(SString str) {
    return str.length==0;   //当字符串为空时，返回 0
}
```

【求赋值内容的长度】

```
int CharLen(char* src) {
    int len = 0;
    while (*src != '\0') {
        len++;
        src++;
    }
    return len;
}
```

【清空】

```
bool StringClear(SString* str) {
    str->ch[0] = '\0';
    str->length = 0;
    return true;
}
```

【基本操作：赋值】

```
bool StrAssign(SString* str, char* src) {
    if (!src) {
        printf("无赋值内容\n");
        return false;
    }
    int i = 0;
    if (CharLen(src) <= MAXLEN) {
        for (;src[i] != '\0';i++) {
            str->ch[i] = src[i];
            str->length++;
        }
        str->ch[i] = '\0';
        return true;
    }
    else {
        printf("赋值超出规定范围\n");
        return false;
    }
}
```

【基本操作：获取子串】

说明：返回串 s 的第 pos 个字符起长度为 len 的子串。该算法在设计时未考虑新变量 s 是否为空，若非空，则要把 s 给清空并将 s 的串长清零。

```
bool SubString(SString* s,SString src,int pos,int len) {
    if (pos<=0 || len<0 || pos + len - 1 > src.length){
        printf("无法获取子串\n");
        return false;
    }
    if (pos + len - 1 <= src.length) {
        for (int i = pos;i < pos + len;i++) {
            s->ch[i - pos] = src.ch[i-1];
            s->length++;
        }
        s->ch[len] = '\0';
        return true;
    }
}
```

【基本操作：复制】

说明：该算法在设计时未考虑新变量 str 是否为空，若非空，则要把 str 给清空并将 str 的串长清零。

```
bool StrCopy(SString* str, SString* s) {
    if (StringEmpty(*s)) {
        printf("所复制的字符串为空，复制无效\n");
        return false;
    }
    int i = 0;
    for (;i < s->length;i++) {
        str->ch[i] = s->ch[i];
    }
    str->ch[i] = '\0';
    str->length = s->length;
    return true;
}
```

【基本操作：字符串的串连接】

说明：该算法在设计时未考虑新变量 new 是否为空，若非空，则要把 new 清空并将 new 串长清零。

```
bool Concat(SString* new, SString s1, SString s2) {
    if (StringEmpty(s1) || StringEmpty(s2)) {
        printf("有空串，无须连接\n");
        return false;
    }
    else if (s1.length + s2.length <= MAXLEN) {
        StrCopy(new, &s1);
        for (int i = new->length;i < s1.length + s2.length;i++) {
            new->ch[i] = s2.ch[i - s1.length];
            new->length++;
        }
        new->ch[new->length] = '\0';
        return true;
    }
    else {
        printf("连接后的字符串超过规定容量，连接失败\n");
        return false;
    }
}
```

【基本操作：字符串的比较】(注意：作比较的是 ASCII码)

说明：s1>s2 返回>0 的数，s1<s2 返回<0 的数，s1=s2 返回 0

```
int StrCompare(SString s1, SString s2) {
    for (int i = 0;i < s1.length && i < s2.length;i++) {
        if (s1.ch[i] != s2.ch[i])
            return s1.ch[i] - s2.ch[i];
    }
    return s1.length - s2.length; //当 s1 和 s2 其中一个是空串或 s1 与 s2 都是空串时执行
}
```

下面这一段代码在主函数中调用即可：

```
if (StrCompare(str6, str7) > 0) {
    printf("str6 比 str7 大\n");
}
else{
    if (StrCompare(str6, str7) == 0) {
        printf("str6 与 str7 相等\n");
    }
    else {
        printf("str6 比 str7 小\n");
    }
}
```

【基本操作：字符串的字符删除】

```
bool StrDelete(SString* str, int pos, int len) {
    int str_len=str->length;
    if (pos > 0&&len<=MAXLEN) {
        for (int i = pos;i < str_len;i++) {
            str->ch[i - 1] = str->ch[i - 1 + len];
        }
        str->length = str->length - len;
        str->ch[str->length] = '\0';
        return true;
    }
}
```

【基本操作：字符串的插入】

```
bool StrInsert(SString* S, int pos,SString T) {
    int t_len;
    if (S->length + T.length <= MAXLEN) {
        t_len = T.length;
    }
    else {
        t_len = MAXLEN - S->length;
    }
        for (int i = S->length;i >= pos;i--) {
            S->ch[i + t_len-1] = S->ch[i-1];
        }
        S->ch[S->length + t_len] = '\0';
        for (int j = 0;j < t_len;j++) {
            S->ch[pos + j - 1] = T.ch[j];
            S->length++;
        }
    return true;
}
```

【输出整个字符串】

```
bool PrintString(SString str) {
    if (StringEmpty(str)) {
        printf("该字符串为空，输出无效\n");
        return false;
    }
    printf("%s\n 该字符串的长度为%d\n\n", str.ch, str.length);
    return true;
}
```

【函数调用】

```
int main() {
    SString str1, str2, str3,str4, str5;     //声明结构体变量
    InitString(&str1);
    InitString(&str2);
    InitString(&str3);
    InitString(&str4);
    InitString(&str5);

    StrAssign(&str1, "A2mfyd82.1");   //给结构体变量 str1 赋值
    StrAssign(&str2, "0jjhd2k");      //给结构体变量 str2 赋值
    StrAssign(&str3, "HUAWEI pro");   //给结构体变量 str3 赋值
    Concat(&str4, str1, str2);        //将 str1 和 str2 连接起来放入 str4
    SubString(&str5, str1, 2, 3);     //将 str1 中第 2 位开始长度为 3 的子串放入 str5
    StrInsert(&str1, 2, str3);        //将 str3 插入到 str1 的第 2 位
    StrDelete(&str2, 2, 3);           //将 str2 从第 2 位开始删除长度为 3 的子串

    PrintString(str1);
    PrintString(str2);
    PrintString(str3);
    PrintString(str4);
    PrintString(str5);
    return 0;
}
```

〔Ⅱ〕串的动态(堆式)顺序存储结构

上面提到的定长顺序存储不太适用于某些实际情况，因此介绍动态的顺序存储结构，它可以根

据实际需要,在程序执行过程中动态地分配和释放字符数组空间。在 C 语言中,存在一个称之为“堆”的自由存储区，可以为每个新产生的串动态分配一块实际串长所需的存储空间，若分配成功，则返回一个指向起始地址的指针，作为串的基址，同时为了以后处理方便，约定串长也作为存储结构的一部分。这种字符串的存储方式也称为串的堆式顺序存储结构，定义内容如下：

```
typedef struct{
    char* ch;             //若是非空串，则按串长分配存储区，否则 ch 为 NULL
    int length;           //串的当前长度
}HString;
```

下面来介绍此种存储结构的一系列代码实现：

【宏定义】

```
#define bool char
#define true 1
#define false 0
```

【结构体】

```
typedef struct {
    char* ch;
    char length;
}HString;
```

【初始化】

```
void InitString(HString* S) {
    S->ch = NULL;
    S->length = 0;
}
```

【判空】

```
char StrEmpty(HString* S) {
    return S->length == 0;
}
```

【清空】

```
bool StrClear(HString* S) {
    if (S->ch) {
        S->length = 0;
        free(S->ch);
    }
    S->ch = NULL;
    return true;
}
```

【求赋值内容的长度】

```
int CharLen(char* t) {
    int len = 0;
    while (*t != '\0') {
        len++;
        t++;
    }
    return len;
}
```

【基本操作：赋值】

```
bool StrAssign(HString* S, char* t) {
    int t_len = CharLen(t);
    if (S->ch) {
```

```
        free(S->ch);
    }
    S->ch = (char*)malloc(sizeof(char) * t_len);
    assert(S->ch);
    for (int i = 0;i < t_len;i++) {
        S->ch[i] = t[i];
    }
    S->length = t_len;
    return true;
}
```

【基本操作：获取子串】

```
bool SubString(HString* T, HString S, int pos, int len) {
    if (pos <= 0 || len < 0 || pos + len-1>S.length) {
        printf("无法获取子串\n");
        return false;
    }
    else {
        if (T->ch) {
            T->length = 0;
            free(T->ch);
        }
        T->ch = (char*)malloc(sizeof(char) * len);
        assert(T->ch);
        for (int i = pos;i <= pos - 1 + len;i++) {
            T->ch[i - pos] = S.ch[i - 1];
            T->length++;
        }
        return true;
    }
}
```

【基本操作：复制】

```
bool StrCopy(HString* S, HString* T) {
    if (S->ch) {
        free(S->ch);
    }
    S->ch = (char*)malloc(sizeof(char) * T->length);
    assert(S->ch);
    for (int i = 0;i < T->length;i++) {
        S->ch[i] = T->ch[i];
    }
    S->length = T->length;
    return true;
}
```

【基本操作：字符串的串连接】

```
bool StrConcat(HString* New,HString S1, HString S2) {
    if (New->ch) {
        free(New->ch);
    }
    New->ch = (char*)malloc(sizeof(char) * (S1.length + S2.length));
    assert(New->ch);
    for (int i = 0;i < S1.length;i++) {
        New->ch[i] = S1.ch[i];
    }
    for (int j = S1.length;j < S1.length + S2.length;j++) {
        New->ch[j] = S2.ch[j - S1.length];
```

```
    }
    New->length = S1.length + S2.length;
    return true;
}
```

【基本操作：字符串的比较操作】

```
int StrCompare(HString S1, HString S2) {
    for (int i = 0;i < S1.length && i < S2.length;i++) {
        if (S1.ch[i] != S2.ch[i])
            return S1.ch[i] - S2.ch[i];
    }
    return S1.length - S2.length; //当 s1 和 s2 其中一个是空串或 s1 与 s2 都是空串时执行
}
```

调用本函数时用如下代码：

```
if (StrCompare(S1, S2) > 0) {
    printf("前者比后者大\n");
}
else {
    if (StrCompare(S1, S2) == 0) {
        printf("前者与后者相等\n");
    }
    else {
        printf("前者比后者小\n");
    }
}
```

【基本操作：字符串的字符删除】

```
bool StrDelete(HString* S, int pos, int len) {
    if (pos <= 0 || pos > S->length || len<0 || len>S->length - pos + 1) {
        printf("删除的位置不合法\n");
        return false;
    }
    for (int i = pos;i < S->length-1;i++) {
        S->ch[i-1]=S->ch[i + len-1];
    }
    S->length -= len;
    return true;
}
```

【基本操作：字符串的插入】

```
bool StrInser(HString* S1, int pos,HString S2) {
    if (S2.length == 0) {
        printf("无须进行插入操作\n");
        return false;
    }else{
        if (pos <= 0 || pos > S2.length) {
            printf("插入位置不合法\n");
            return false;
        }
        //下面进行重分配空间
        char* t = (char*)realloc(S1->ch, sizeof(char) * (S1->length+S2.length));
        assert(t);
        S1->ch = t;
        for (int i = S1->length;i >= pos;i--) {
            S1->ch[i-1+S2.length]=S1->ch[i - 1];
        }
```

```
        for (int j = pos;j < pos + S2.length;j++) {
            S1->ch[j - 1] = S2.ch[j - pos];
        }
        S1->length +=S2.length;
        return true;
    }
}
```

【输出整个字符串】

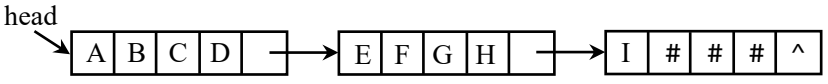
```
bool PrintString(HString S) {
    for (int i = 0;i < S.length;i++) {
        printf("%c", S.ch[i]);
    }
    printf("\n 该字符串的长度为%d\n", S.length);
    return true;
}
```

【函数调用】

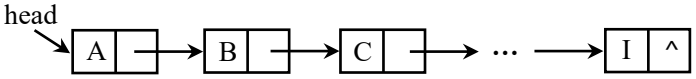
```
int main() {
    HString S1;
    HString S2;
    HString New;
    InitString(&S1);
    InitString(&S2);
    InitString(&New);

    StrAssign(&S1, "5478jj879");
    StrAssign(&S2, "ffdj");

    StrConcat(&New, S1, S2);
    PrintString(New);

    SubString(&New,S1, 1, 1);
    PrintString(New);

    StrInser(&S1, 2, S2);
    StrDelete(&S1, 3, 7);
    PrintString(S1);
    return 0;
}
```

（2）串的链式存储（链串）

因为顺序串的插入和删除操作不方便，需要移动大量的字符，所以可采用单链表方式存储串。由于串结构的特殊性——结构中的每个数据元素是一个字符，则在用链表存储串值时，存在一个“结点大小”的问题，即每个结点可存放一个字符，也可存放多个字符。



（图 1：结点大小为 4 的链表）



（图 2：结点大小为 1 的链表）

说明：在上述两个图中我们可以看到，当结点大小大于 1 时，由于串长不一定是结点大小的整倍数，则链表中的最后一个结点不一定全被串值占满，此时通常补上“#”或其它的非串值字符（通常“#”不属于串的字符集，是一个特殊的符号）。

为了便于进行串的操作，当以链表存储串值时，除头指针外，还可附设一个尾指针指示链表中的最后一个结点，并给出当前串的长度。称如此定义的串存储结构为块链结构。由于与单链表的操作类似，这里不详细提及。

3. 串的模式匹配算法（该部分是重难点内容，需要多复习！）

（1）BF 算法

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

（注：该算法设计只能返回第一次模式串匹配上的首元素位置。）

```
void BF(char text[], char pattern[]) {
    int text_len = strlen(text);
    int pattern_len = strlen(pattern);
    if (text_len == 0 || pattern_len == 0) {
        printf("无从查找\n");
        return;
    }
    int i = 0;
    int j = 0;
    while (i < text_len && j < pattern_len) {
        if (text[i] == pattern[j]) {
            i++;j++;
        }
        else {
            i = i - j + 1;
            j = 0;
        }
    }
    if (j == pattern_len) {
        printf("Found at %d\n", i - j);
    }
    else {
        printf("未找到\n");
    }
}
int main() {
    char text[] = "ABCDFAXSD";
    char pattern[] = "AX";
    BF(text, pattern);
    return 0;
}
```

（2）KMP 算法

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

【求 next 数组】

```
void prefix_table(char pattern[], int prefix[], int n) {
    prefix[0] = 0;
    int len = 0;
```

```
    int i = 1;
    while (i < n) {
        if (pattern[i] == pattern[len]) {
            len++;
            prefix[i] = len;
            i++;
        }
        else {
            if (len > 0) {
                len = prefix[len - 1];
            }
            else {
                prefix[i] = len;
                i++;
            }}}}
```

调用该代码如下：

```
int main(){
    char pattern[] = "AABACAAB";
    int n=strlen(pattern);
    int* prefix=(int*)malloc(sizeof(int)*n);
    prefix_table(pattern,prefix,n);
    for(int i=0;i<n;i++){
        printf("%d\t",prefix[i]);
    }
    free(prefix);
    return 0;
}
```

【给 next 数组移位】

```
void move_prefix_table(int prefix[], int n) {
    for (int i = n - 1;i > 0;i--) {
        prefix[i] = prefix[i - 1];
    }
    prefix[0] = -1;
}
```

【KMP 算法找模式串的位置】（注：这是优化后的算法，可找多个位置的模式串）

```
void KMP(char text[], char pattern[]) {
    int n = strlen(pattern);
    int m = strlen(text);
    int* prefix = (int*)malloc(sizeof(int) * n);
    assert(prefix);
    prefix_table(pattern, prefix, n);
    move_prefix_table(prefix, n);
    int i = 0;
    int j = 0;
    while (i < m) {
        if (j == n - 1 && text[i] == pattern[j]) {
            printf("Found at %d\n", i - j);
            j = prefix[j];
            if (j == -1) {
                j = 0;
            }
        }
        if (text[i] == pattern[j]){
            if (j == n - 1) {
                i++;
            }
            else {
                i++;j++;
```

```
            }
        }
        else {
            j = prefix[j];
            if (j == -1) {
                i++;j++;
            }}}
    free(prefix);
}
```

下面是优化之前的 while 循环，只能返回第一次模式串匹配上的首元素位置：

```
while (i < m) {
        if (j == n - 1 && text[i] == pattern[j]) {
            printf("Found at %d\n", i - j);
            j = prefix[j];
        }
        if (text[i] == pattern[j]){
                i++;j++;
        }
        else {
            j = prefix[j];
            if (j == -1) {
                i++;j++;
            }}}
```

【调用 KMP 算法】

```
int main() {
    char pattern[] = "AABACAAB";
    char text[] = "ACDAABACAABBASCX";
    KMP(text, pattern);    //找出模式串在文本串中第一次匹配成功的首元素下标
    return 0;
}
```

九、数组

1. 数组的类型定义

数组是由类型相同的数据元素构成的有序集合，每个元素称为数组元素，每个元素受 $n(n\geqslant 1)$个线性关系的约束，每个元素在 n 个线性关系中的序号 $i_1,i_2,\ldots,i_n$ 称为该元素的下标，可以通过下标访问该数据元素。数据可以看成线性表的推广，其特点是结构中的元素本身可以是具有某种结构的数据，但属于同一数据类型。

一维数组可看成一个线性表，二维数组可看成数据元素是线性表的线性表。二维数组就是由多个一维数组组成的数组。（**注意：**数组一旦被定义，它的维数和维界就不再改变。因此，除了结构的初始化和销毁之外，数组只有存取元素和修改元素值的操作。）

（1）二维数组的类型定义格式如下：

数据类型　数组名[数字 1][数字 2];

说明：

①数字 1 代表总共有多少个一维数组。

②数字 2 代表所组成的二维数组中每个一维数组中元素的个数。

（2）二维数组的遍历输出代码如下：

```
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            printf("%d",array[i][j]);
        }
```

```
        printf("\n");
    }
```

（3）二维数组赋值

```
int arr[3][4]={
    {11,12,13,14},
    {21,22,23,24},
    {31,32,33,34}
};
```

说明：二维数组可以不给定组的个数，但必须给定每组中元素的个数，形如：int arr[][4]。对于不给定组数的这种情况，在赋值时可直接用一个花括号，里面的元素会被自动分配成合适的组数，形如：int arr[][4]={11,12,13,14,15,16,17,18,19,20,21,22};这个步骤被称为二维数组的初始化，在没有初始化的情况下，是不能直接定义成 int arr[][4];的。

（4）二维数组例题

【例 1】设计一个使 5 行 5 列矩阵转置的函数并调用，要求可以自己输入矩阵。

```
void transposition(int arr[5][5]) {
    for (int i = 0;i < 5;i++) {
        for (int j = 0;j < i;j++) {
            int t = arr[i][j];
            arr[i][j] = arr[j][i];
            arr[j][i] = t;
        }}}
int main() {
    int a[5][5];
    for (int i = 0;i < 5;i++) {
        for (int j = 0;j < 5;j++) {
            scanf("%d", &a[i][j]);
        }}
    transposition(a);
    return 0;
}
```

（5）二维数组的表示方法

对于下面这个二维数组，我们对其进行输出。

```
int arr[3][2]={{11,12},{21,22},{31,32}};
int* p1=arr[0];
int* p2=arr[1];
int* p3=arr[2];
printf("%d",arr[0][0]); //还可写成(*(arr+0))[0]或 p1[0]或*(p1+0)或*(*(arr+0)+0)
printf("%d",arr[0][1]); //还可写成(*(arr+0))[1]或 p1[1]或*(p1+1)或*(*(arr+0)+1)
```

（6）二维数组的动态分配

下面用不同班级的学生为例，来说明此分配方式。**（本质：创建能存放指针类型的数组）**

```
int m,n;
printf("请输入班级数：");
scanf("%d",&m);
double** d=(double**)malloc(sizeof(double*)*m);
printf("请输入每个班的学生人数：");
scanf("%d",&n);
for(int i=0;i<m;i++){
    d[i]=(double*)malloc(sizeof(double)*n);
```

```
}
```

2. 数组的顺序存储

由于对数组一般不进行插入或删除操作，也就是说，一旦建立了数组，则结构中的数据元素个数和元素之间的关系一般就不再发生变动，因此，采用顺序存储结构表示数组比较合适。

由于存储单元是一维的结构，而数组可能是多维的结构，则用一组连续存储单元存放数组的数据元素就有次序约定问题，也就是以下的两种存储方式。

对二维数组可有两种存储方式：①以列序为主序的存储方式；②以行序为主序的存储方式

（**注意：**在扩展 Basic、Pascal、Java 和 C 语言中，用的都是以行序为主序的存储结构；而在 FORTRAN 语言中，用的是以列序为主序的存储结构。）

下面以行序为主序的存储结构为例，来总结计算存储位置的公式，设每个数据元素占 L 个存储单元，则二维数组 $A[0\ldots m\text{-}1, 0\ldots n\text{-}1]$（下标从 0 开始，共有 m 行 n 列，**注：**有些题目的下标不一定是从 0 开始的，因此做题时要根据实际情况来分析）中任一元素 a_{ij} 的存储位置可由下式确定：

$$\mathrm{LOC}(i, j) = \mathrm{LOC}(0, 0) + (n \times i + j) \times L$$

其中的 $\mathrm{LOC}(i,j)$ 是 a_{ij} 的存储位置；$\mathrm{LOC}(0,0)$ 是 a_{00} 的存储位置，即二维数组 A 的起始存储位置，也称为基地址或基址。

3. 特殊矩阵的压缩存储

该部分内容自行学习教材即可。

4. 静态数组与动态数组

【静态数组】

在程序运行过程中，数组的大小是不能改变的，这种数组称为**静态数组**。静态数组的缺点是：对于事先无法准确估计数据量的情况，无法做既满足处理需要，又不浪费内存空间。

【动态数组】

所谓的动态数组是指，在程序运行过程中，根据实际需要指定数组的大小。

在 C 语言中，可利用内存的申请和释放库函数，以及指向数组的指针变量可当数组名使用的特点，来实现动态数组。

编码规范：不可以返回局部变量的地址！

下面通过一个例子来更好地认识动态数组，用动态数组计算 n（由用户从键盘输入的）名学生的平均成绩示例：

```
int main(int argc, char* argv[]){
    int stucount = 0;
    float* pscore = NULL;
    int i = 0;
    float sum = 0, ave = 0;
    printf("请输入学生的人数：");
    scanf("%d", &stucount);
    pscore = (float*)malloc(sizeof(float)*stucount);  //动态申请内存
    if(pscore == NULL)        //如果指针为空，直接返回
        return 0;
    printf("请输入%d 个学生的成绩\n", stucount);
    for (i = 0; i < stucount; i++){
        scanf("%f", pscore + i);    //也可写成 scanf("%f", &pscore[i]);
    }
```

```
    for (i = 0; i < stucount; i++){
        sum += *(pscore + i);       //也可写成 sum += pscore[i];
    }
    ave = sum / stucount;
    printf("ave score is %.1f\n", ave);
    free(pscore);
    pscore = NULL;
    return 0;
}
```

上述这段代码用到了动态申请内存，下面是动态申请内存使用的注意点：

①malloc 动态申请内存，不一定 100%成功，对于申请的返回地址需要判断是不是 NULL，如果是 NULL 说明申请失败，则进行返回或者异常处理。

②动态申请的内存是内存模型中的堆区申请的，生命周期由程序员控制，需要调用 free()才能够释放。

③free()之后，还需要对之前使用的指针变量置 NULL，避免野指针。

十、广义表（列表）

仅着重提及广义表的两个重要运算，由于该部分其它内容较简单，此处略，自行学习教材即可。

取表头操作 GetHead()：取出的表头为非空广义表的第一个元素，它可以是一个单原子，也可以是一个子表。

取表尾 GetTail()：取出的表尾为除去表头之外由其余元素**构成的表**，即表尾一定是一个广义表。

【例题】

（1）GetHead((p,h,w))=p　　GetTail((p,k,p,h))=(k,p,h)

（2）GetHead((a,(b,c,d)))=a　　GetTail((a,(b,c,d)))=((b,c,d))

（3）GetHead(((a,b),(c,d)))=(a,b)　　GetTail(((a,b),(c,d)))=((c,d))

（4）GetHead((e))=e　　GetTail((e))=()

（5）GetHead(GetTail(((a,b),(c,d))))=(c,d)　　GetTail(GetHead(((a,b),(c,d))))=(b)

（6）GetHead(GetTail(GetHead(((a,b),(c,d)))))=b　　GetTail(GetHead(GetTail(((a,b),(c,d)))))=(d)

十一、二叉树

1. 二叉树的存储结构

（1）二叉树的顺序存储结构

顺序存储结构使用一组地址连续的存储单元来存储数据元素。对于完全二叉树，只要从根起按层序存储即可，将结点元素按自上而下、自左向右的规则依次存入到一维数组中；对于一般二叉树，可能会存在有一维数组单元空出来的情况，导致空间的浪费。因此，顺序存储结构仅适用于完全二叉树或满二叉树（**注：**满二叉树属于完全二叉树）。

示例代码如下：

```
#define MAXSIZE 100
char tree[MAXSIZE];
//按先序遍历的规则来输入结点元素从而构建一棵二叉树（要用递归思想）
void CreateTree(int index) {
    char ch;
    scanf("%c", &ch);
    if (ch == '#') return;
    else {
        tree[index] = ch;
        CreateTree(index * 2 + 1);
```

```
        CreateTree(index * 2 + 2);
    }
}

int main() {
    CreateTree(0);      //此处的 0 是第 1 个结点的数组下标
    for (int i = 0;i < 12;i++) {
        printf("%c ", tree[i]);   //输出将构建的二叉树结点元素所存入的一维数组
    }
    return 0;
}
```

（2）二叉树的链式存储结构

设计不同的结点结构可构成不同形式的链式存储结构。

表示二叉树链表中的结点至少包含 3 个域：数据域、左指针域、右指针域（如下图 1）；有时为了便于找到结点的双亲，还可在结点结构中增加一个指向其双亲结点的指针域（如下图 2），利用这两种结点结构所得二叉树的存储结构分别称为二叉链表和三叉链表。

lchild	data	rchild

（图 1：含两个指针域的结点结构）

lchild	data	parent	rchild

（图 2：含三个指针域的结点结构）

下面先介绍第一种比较原始的构建树的方法：

```
typedef int ElemType;
typedef struct node {
    ElemType data;
    struct node* left;
    struct node* right;
}TNode;
int main() {
    TNode n1, n2, n3, n4, n5;
    n1.data = 8;
    n2.data = 2;
    n3.data = 5;
    n4.data = 9;
    n5.data = 1;
    n2.left = &n1;   n2.right = &n4;      //以 n2 为树的根结点，遍历时传参就传入 n2
    n1.left = &n5;   n1.right = &n3;
    n3.left = NULL;  n3.right = NULL;
    n4.left = NULL;  n4.right = NULL;
    n5.left = NULL;  n5.right = NULL;
    return 0;
}
```

上面这种方法需要写入大量的代码，操作不灵活，接下来介绍一种新的构建二叉树的方法——按遍历规则来构建二叉树，下列以先序遍历为例来构建二叉树，示例代码如下：

```
typedef char ElemType;
typedef struct TNode {
    ElemType data;
    struct TNode* left;
    struct TNode* right;
}TNode;
void createtree_pre(TNode** tree) {
    char e;
    printf("输入当前结点的值(按先序遍历来输):");
    scanf("%c", &e);
```

```
    getchar();          //输入字符后直接按回车可能遗留缓冲区中的换行符，影响后
                          续的 scanf 调用。getchar()用于消耗掉换行符
    if (e == '#') {
        *tree = NULL;
        return;
    }
    else {
        TNode* new = (TNode*)malloc(sizeof(TNode));
        assert(new);
        new->data = e;
        *tree = new;
        createtree_pre(&((*tree)->left));
        createtree_pre(&((*tree)->right));
    }
}

int main() {
    TNode* tree = NULL;
    createtree_pre(&tree);
    return 0;
}
```

2. 遍历二叉树

说明：先序遍历顺序为根左右；**中序遍历**顺序为左根右；**后序遍历**顺序为左右根；**层次遍历**的顺序是按每层从左至右依次访问。

（1）遍历的递归算法

【先序遍历】

```
void preorder(TNode* node) {
    if (node) {
        printf("%d ", node->data);
        preorder(node->left);
        preorder(node->right);
    }
}
```

【中序遍历】

```
void inorder(TNode* node) {
    if (node) {
        inorder(node->left);
        printf("%d ", node->data);
        inorder(node->right);
    }
}
```

【后序遍历】

```
void postorder(TNode* node) {
    if (node) {
        postorder(node->left);
        postorder(node->right);
        printf("%d ", node->data);
    }
}
```

【层次遍历】

【递归输出二叉树】

```
void PrintTree(TNode* tree,int k) {
    if (tree == NULL||k<1) {
        return;
```

```
    }
    if(k==1) {
        printf("%c ", tree->data);
    }
    PrintTree(tree->left, k - 1);
    PrintTree(tree->right, k - 1);
}
```

【层次遍历(递归)】

```
void LevelTraversal(TNode* tree) {
    if (tree == NULL) {
        return;
    }
    int h = TreeHeight(tree);        //此处要用到已编写好的求树高函数
    for (int i = 1;i <= h;i++) {     //每层依次遍历
        PrintTree(tree, i);          //每一层都调用递归输出二叉树的函数
    }
}
```

（2）遍历的非递归算法（**重点!**）

【先序遍历】

核心思路：我们面对的主要问题是在先序遍历完某个结点的整个左子树后，如何找到该结点的右子树的根指针？解决的办法是在访问完该结点后，将该结点的指针保存到栈中，以便后续通过它找到该结点的右子树。

示例代码如下：

【预编译内容】

```
//宏定义
#define bool char
#define true 1
#define false 0
//自定义数据类型
typedef char ElemType;
```

【二叉树结点结构体】

```
typedef struct TNode {
    ElemType data;
    struct TNode* left;
    struct TNode* right;
}TNode;
```

【临时栈的结构体】

```
typedef struct LinkStack {
    TNode* data;     //临时栈中的数据域用来存放树结点的指针
    struct LinkStack* next;
}LinkStack;
```

【用先序遍历的法则创建二叉树(不含头结点)】

```
void createtree_pre(TNode** tree) {
    char e;
    printf("输入当前结点的值(按先序遍历来输):");
    scanf("%c", &e);
    getchar();
    if (e == '#') {
        *tree = NULL;
        return;
    }
```

```
    else {
        TNode* new = (TNode*)malloc(sizeof(TNode));
        assert(new);
        new->data = e;
        *tree = new;
        createtree_pre(&((*tree)->left));
        createtree_pre(&((*tree)->right));
}}
```

【初始化临时栈】（**注：**这步操作等价于单链表中的创建头结点）

```
LinkStack* InitLinkStack() {
    LinkStack* S = (LinkStack*)malloc(sizeof(LinkStack));
    S->data = NULL;
    S->next = NULL;
    return S;
}
```

【临时栈判空】

```
bool isEmpty(LinkStack* S) {
    if (S->next == NULL)
        return true;
    return false;
}
```

【入栈】

```
void Push(LinkStack* S, TNode* p) {
    LinkStack* new = (LinkStack*)malloc(sizeof(LinkStack));
    new->data = p;
    new->next = S->next;
    S->next = new;
}
```

【获取栈顶元素（本质是获取出栈元素的指针）并让栈顶元素出栈】

```
LinkStack* Pop(LinkStack* S) {
    if (isEmpty(S)) {
        return NULL;
    }
    else {
        LinkStack* m = S->next;
        S->next = m->next;   //实现出栈
        return m;            //返回出栈元素的指针
}}
```

【先序遍历输出（非递归）】

```
void no_cycle_preorder(TNode* tree) {
    LinkStack* S = InitLinkStack();
    TNode* p = tree;
    while (p || !isEmpty(S)) {
        if (p) {
            printf("%c ", p->data);
            Push(S, p);
            p = p->left;
        }
        else {
            p=Pop(S)->data;
            p = p->right;
}}}
```

【函数调用】

```
int main() {
    TNode* tree = NULL;
    createtree_pre(&tree);
    no_cycle_preorder(tree);
    return 0;
}
```

【中序遍历】

核心思路：与前序遍历的思路类似，访问结点的操作发生在该结点的左子树遍历完毕并准备遍历右子树时，因此程序的逻辑顺序为：当遍历到某结点时，先将该结点的指针压入栈，等到它的左子树全部遍历完后，再从栈中弹出并访问该结点。

示例代码：只需要将 `printf("%c ", p->data);`移到 `p=Pop(S)->data;`后面即可。

【后序遍历】

核心思路：从根结点开始寻找最左边的结点，寻找的过程中将访问到的结点标记为 1 并依次入栈，当指针为空时，回退到上一结点（栈顶元素），并且栈顶元素出栈，检测标记，若标记是 1，说明没访问过该结点的右子树，那么将标记变为 2 又再次将该结点地址压入栈中，指针向该结点右子树方向移动，当右子树方向的指针为空时，指针重新指向该子树的根结点处，然后栈顶元素出栈，并检测标记是否为 1，如果不是 1 就输出该结点的值。

示例代码：

【预编译内容】

```
//宏定义
#define bool char
#define true 1
#define false 0
//自定义数据类型
typedef char ElemType;
```

【二叉树结点结构体】

```
typedef struct TNode {
    ElemType data;
    struct TNode* left;
    struct TNode* right;
    int flag;       //增加一个标识结点访问次数的变量
}TNode;
```

【临时栈的结构体】

```
typedef struct LinkStack {
    TNode* data;    //临时栈中的数据域用来存放树结点的指针
    struct LinkStack* next;
}LinkStack;
```

【用先序遍历的法则创建二叉树(不含头结点)】

```
void createtree_pre(TNode** tree) {
    char e;
    printf("输入当前结点的值(按先序遍历来输):");
    scanf("%c", &e);
    getchar();
    if (e == '#') {
        *tree = NULL;
        return;
    }
    else {
```

```
        TNode* new = (TNode*)malloc(sizeof(TNode));
        assert(new);
        new->data = e;
        new->flag = 0;
        *tree = new;
        createtree_pre(&((*tree)->left));
        createtree_pre(&((*tree)->right));
}}
```

【初始化临时栈】

```
LinkStack* InitLinkStack() {
    LinkStack* S = (LinkStack*)malloc(sizeof(LinkStack));
    S->data = NULL;
    S->next = NULL;
    return S;
}
```

【临时栈判空】

```
bool isEmpty(LinkStack* S) {
    if (S->next == NULL)
        return true;
    return false;
}
```

【入栈】

```
void Push(LinkStack* S, TNode* p) {
    LinkStack* new = (LinkStack*)malloc(sizeof(LinkStack));
    new->data = p;
    new->next = S->next;
    S->next = new;
}
```

【出栈】

```
bool Pop(LinkStack* S) {
    if (isEmpty(S)) {
        return false;
    }
    else {
        LinkStack* m = S->next;
        S->next = m->next;
        free(m);
        return true;
    }
}
```

【获得栈顶元素】

```
LinkStack* GetTop(LinkStack* S) {
    if (isEmpty(S)) {
        return NULL;
    }
    else {
        LinkStack* m = S->next;
        return m;
    }
}
```

【后序遍历输出(非递归)】

```
void no_cycle_postorder(TNode* tree) {
    LinkStack* S = InitLinkStack();
```

```
    TNode* p = tree;
    while (p || !isEmpty(S)) {
        if (p) {
            p->flag = 1;
            Push(S, p);
            p = p->left;
        }
        else {
            p = GetTop(S)->data;
            Pop(S);
            if (p->flag == 1) {
                p->flag = 2;
                Push(S, p);
                p = p->right;
            }
            else {
                printf("%c ", p->data);
                p = NULL;
}}}}
```

【函数调用】

```
int main() {
    TNode* tree = NULL;
    createtree_pre(&tree);
    no_cycle_postorder(tree);
    return 0;
}
```

【层次遍历】

示例代码：

【预编译内容】

```
//宏定义
#define bool char
#define true 1
#define false 0
//自定义数据类型
typedef char ElemType;
```

【二叉树结点结构体】

```
typedef struct TNode {
    ElemType data;
    struct TNode* left;
    struct TNode* right;
}TNode;
```

【临时队列(实质是循环双链表)的结构体】

```
typedef struct QueueNode {
    TNode* data;   //临时队列中的数据域用来存放树结点的指针
    struct QueueNode* prior;
    struct QueueNode* next;
}QueueNode;
```

【用先序遍历的法则创建二叉树】

与前面相同，这里略

【临时队列初始化】

```
QueueNode* InitQueue() {
    QueueNode* Q = (QueueNode*)malloc(sizeof(QueueNode));
    Q->data = NULL;
    Q->prior = Q;
    Q->next = Q;
    return Q;
}
```

【临时队列判空】

```
bool IsEmpty(QueueNode* Q) {
    if (Q->next == Q)
        return true;
    return false;
}
```

【入队(尾插)】

```
void enQueue(QueueNode* Q,TNode* p) {
    QueueNode* new = (QueueNode*)malloc(sizeof(QueueNode));
    assert(new);
    new->data = p;
    new->next = Q;
    new->prior = Q->prior;
    Q->prior->next = new;
    Q->prior = new;
}
```

【获取队首元素指针并出队】

```
QueueNode* exitQueue(QueueNode* Q) {
    if (IsEmpty(Q)) {
        return NULL;
    }
    else {
        QueueNode* t = Q->next;      //获取队首元素的指针
        Q->next->next->prior = Q;
        Q->next = Q->next->next;
        return t;                    //将队首元素指针返回
    }
}
```

【层次遍历输出(非递归)】

```
void LevelTraversal(TNode* tree, QueueNode* Q) {
    enQueue(Q, tree);
    while (!IsEmpty(Q)) {
        QueueNode* m = exitQueue(Q);
        printf("%c ", m->data->data);
        if (m->data->left) {
            enQueue(Q, m->data->left);
        }
        if (m->data->right) {
            enQueue(Q, m->data->right);
}}}
```

【函数调用】

```
int main() {
    TNode* tree = NULL;
    QueueNode* Q = InitQueue();
    createtree_pre(&tree);
```

```
    LevelTraversal(tree, Q);
    return 0;
}
```

（3）二叉树遍历算法的应用

〔I〕**求二叉树的高度**

```
int TreeHeight(TNode* tree) {
    if (!tree) {
        return 0;
    }
    else {
        int leftheight = TreeHeight(tree->left);
        int rightheight= TreeHeight(tree->right);
        int height=leftheight>rightheight?leftheight:rightheight;  //三目运算符
        return height + 1;
    }
}
int main() {
    TNode* tree = NULL;
    createtree_pre(&tree);    //必须先写好建树函数，该处省略
    printf("该二叉树的高度是%d\n", TreeHeight(tree));
    return 0;
}
```

〔II〕**求二叉树的结点个数**

```
int Count_Node(TNode* tree) {
    if (tree == NULL)
        return 0;
    return Count_Node(tree->left) + Count_Node(tree->right) + 1;
}
```

//每遍历到一个结点就递归访问它的左右子树，到指针为空为止，并返回，直到返回到最上面的根结点为止，得到最终的结点个数（加上整棵树的根结点本身）。

3. 线索二叉树

提问：如何保存在动态遍历中的前驱和后继信息呢？

我们知道，以二叉链表存储时，只能找到结点的左孩子、右孩子信息，而不能获取该结点在任意遍历序列中的前驱和后继；这种信息只能在遍历的动态过程中才能得到。

解决：为了解决上述问题，引入了**线索化**的概念，也就是在遍历过程中，使二叉链表中结点的**空链域**存放其前驱或后继信息的过程。指向该线性序列中的前驱或后继的**指针**称为**“线索”**。

（1）构造线索二叉树

（**图示方法：**先把二叉树需要的遍历序列写出，然后再逐个结点对照空链域有几个，若没有空链域就说明此结点不需要操作；有 1 个空链域就判断是左空还是右空，左空就让空链域指向该结点的遍历前驱，反之就指向遍历后继；有 2 个空链域就分别指向前驱和后继。指向线条用带指向箭头的虚线表示。）

【中序线索二叉树的构建】

注意：构建线索二叉树是在原有二叉树的基础上进行构建的，因此你必须事先构建好一棵未线索化的二叉树，不要忘记给每个树结点的 ltag 和 rtag 标记为 0。下列示例代码省略构建未线索化的二叉树（不带头结点，要用二级指针）的步骤。

示例代码：

```
/*构建中序线索二叉树*/
//第一部分（易错点：因为指针 pre 是从指向 NULL 开始创建的，故要用二级指针）
void inorderThTree(TNode* p, TNode** pre) {
```

```
    if (p == NULL) {
        return;
    }
    else {
        inorderThTree(p->left, pre);
        if (p->left == NULL) {
            p->left = *pre;
            p->ltag = 1;
        }
        if (*pre && !(*pre)->right) {
            (*pre)->right = p;
            (*pre)->rtag = 1;
        }
        *pre = p;
        inorderThTree(p->right,pre);
    }
}
//第二部分
void CreateInThread(TNode* tree) {
    TNode* pre = NULL;
    if (tree) {
        inorderThTree(tree, &pre);
        pre->right = NULL;
        pre->rtag = 1;
    }
}
```

（2）遍历线索二叉树

示例代码：

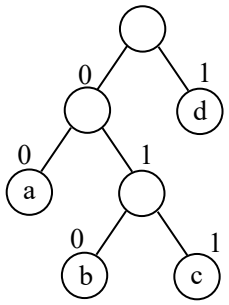
```
/*访问中序线索二叉树的结点*/
//求中序序列第 1 个结点
TNode* FirstNode(TNode* tree) {
    TNode* p = tree;
    while (p->ltag == 0)
        p = p->left;
    return p;
}
//求中序序列中，某个结点的后继结点
TNode* NextNode(TNode* tree) {     //传入的参数是所求的那个结点的地址
    TNode* p = tree;
    if (p->rtag == 0)
        return FirstNode(p->right);
    return p->right;
}
```

4. 哈(霍)夫曼树

哈夫曼树在通信、编码和数据压缩等技术领域有很广泛的应用。以数据压缩为例，为了使压缩后的数据文件尽可能短，可采用不定长编码。但如果设计得不合理，在解码时可能会出错（例如设定 A 为 0，B 为 01，C 为 010，将 001010010 这段编码进行解码就会产生歧义导致出错，无法得到准确的结果），因此，若要设计长短不等的编码，必须满足一个条件：任何一个字符的编码都不是另一个字符的编码的前缀。

综上所述，为确保对数据文件进行有效的压缩和对压缩文件进行正确的解码，可以利用哈夫曼树来设计二进制编码。

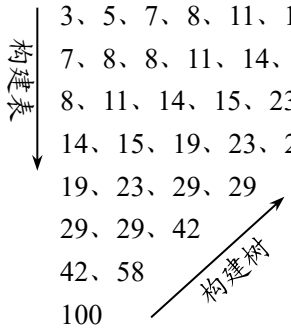
右图是一棵已构建好的哈夫曼树，这时我们便可以求哈夫曼编码，**编码的原则**是整棵树的树根结点不编号，树根结点的左孩子编为 0，树根结点的右孩子编为 1，以此类推，将整棵树编完，整棵树的叶子结点为数据串中的所有字符种类；此外，每种字符的编码为从树根结点走到对应叶子结点所经过的数码，例如：a 的编码为 00，b 的编码为 010，c 的编码为 011，d 的编码为 1。

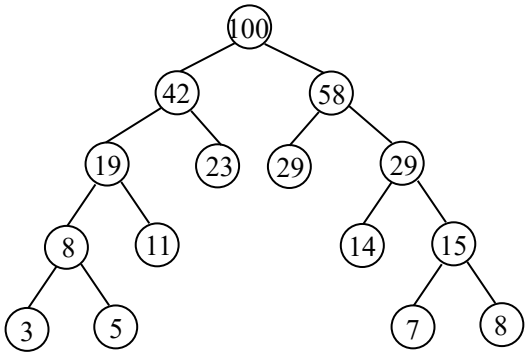


【例题】

有一组数据（权值）分别是 5、29、7、8、14、23、3、11，将其构造一棵哈夫曼树。

解析：先将这组数据从小到大排序得 3、5、7、8、11、14、23、29，然后将首两个数据相加得 8，再与首两个之外的数据重新排序得 7、8、8、11、14、23、29，重复这一过程得 8、11、14、15、23、29，继续重复得 14、15、19、23、29，继续重复得 19、23、29、29，继续重复得 29、29、42，继续重复得 42、58，最后将剩下的两个数据相加得 100，具体图示如下：





将写好的数据从最下层往上观察并构建哈夫曼树**（如右上图）**。刚才有相加过程的数据分别是：42+58=100; 29+29=58; 19+23=42; 14+15=29; 8+11=19; 7+8=15; 3+5=8。我们可以发现这颗哈夫曼树所有的叶子结点的权值构成了题干中的那组数据（5、29、7、8、14、23、3、11），计算其 *WPL* 为(3+5+7+8)×4+(11+14)×3+(23+29)×2=271。

除了构建哈夫曼树，还需对此树的存储结构创建表**（如右图）**，原则是：将已知数据填入表格（可不按权值大小顺序），其余的接着后面填，按数据相加过程从上至下观察。

要注意的是，在此题中，叶子结点中的值不是数据串的字符种类，而是权值，在做题时一定要根据题意来分析。

结点编号	weight	parent	lchild	rchild
1	3	**9**	**0**	**0**
2	5	**9**	**0**	**0**
3	7	**10**	**0**	**0**
4	8	**10**	**0**	**0**
5	11	**11**	**0**	**0**
6	14	**12**	**0**	**0**
7	23	**13**	**0**	**0**
8	29	**14**	**0**	**0**
9	8	**11**	**1**	**2**
10	15	**12**	**3**	**4**
11	19	**13**	**9**	**5**
12	29	**14**	**6**	**10**
13	42	**15**	**11**	**7**
14	58	**15**	**8**	**12**
15	100	**0**	**13**	**14**

（1）基本概念

①树的路径长度（*PL*）：从树根到所有叶子结点的路径长度之和。

②权：赋予某个实体的一个量，是对实体的某个或某些属性的数值化描述。

③树的带权路径长度（*WPL*）：树中所有叶子结点的带权路径长度之和。

④哈夫曼树（最优二叉树）：假设有 *n* 个权值（每个叶子结点代表一个数据串中的字符，且每个叶子结点都有一个权值，权值一般用每个字符在数据串中出现的次数表示），可构造多种含 *n* 个叶子结点的二叉树方案，其中 *WPL* **最小**的二叉树称为哈夫曼树（最优二叉树）。

（2）特点

①在哈夫曼树中，权值越大的结点离树根结点越近。（出现次数越多的字符编的码越短）

②哈夫曼树（最优二叉树）不一定是完全二叉树。

③哈夫曼树中没有度为 1 的结点，因此一棵有 *n* 个叶子结点的哈夫曼树共有 2*n*−1 个结点。

④哈夫曼树不唯一，但 *WPL* 必然相同且最优。

（3）代码实现（**说明：**代码构建哈夫曼树的本质其实是一个创建哈夫曼树存储结构表的过程。）

案例一：已知权值，没给定字符种类，仅存储哈夫曼树的结点信息即可，不编码，代码如下：

【结构体】

```
typedef struct{
    int weight;                    //单个结点的权值
    int parent,lchild,rchild;      //单个结点的双亲结点、其左孩子、其右孩子的下标
}HTNode;
```

weight	parent	lchild	rchild

（结点的存储结构设计）

```
typedef struct{
    HTNode* data; //哈夫曼树中的结点单元(需计算整棵树的结点总数2n-1，n代表叶子结点数)
    int length;   //当前哈夫曼树中结点的个数
}HFTree;
```

【初始化】

```
HFTree* InitHFTree(int* info, int leafcount) {
    HFTree* T = (HFTree*)malloc(sizeof(HFTree));
    T->data = (HTNode*)malloc(sizeof(HTNode) * (leafcount * 2 - 1));
    T->length = leafcount;
    for (int i = 0;i < leafcount;i++) {
        T->data[i].weight = info[i];
        T->data[i].parent = 0;     //标识没有双亲结点
        T->data[i].lchild = -1;    //标识没有左孩子
        T->data[i].rchild = -1;    //标识没有左孩子
    }
    return T;
}
```

【求数据中最小的两个】

```
int* Select(HFTree* T) {
    int Min = INT_MAX;              //给Min赋值为整型最大值(库备)
    int Second_Min = INT_MAX;       //给Second_Min赋值为整型最大值(库备)
    int MinIndex, Second_MinIndex;
    for (int i = 0; i < T->length; i++) {
        if (T->data[i].parent == 0) {
            if (T->data[i].weight < Min) {
                Min = T->data[i].weight;
                MinIndex = i;
            }
        }
    }
    for (int i = 0; i < T->length; i++) {
        if (T->data[i].parent == 0 && i != MinIndex) {
            if (T->data[i].weight < Second_Min) {
                Second_Min = T->data[i].weight;
                Second_MinIndex = i;
            }
        }
    }
    int* res = (int*)malloc(sizeof(int) * 2); //由于有两个返回值，故需开辟两个空间
    res[0] = MinIndex;
    res[1] = Second_MinIndex;
    return res;    //返回两个最小权值的下标
}
```

【构建哈夫曼树】

```
void CreateHuffmanTree(HFTree* T) {
    int length = T->length * 2 - 1;
    for (int i = T->length; i < length; i++) {
        int* res = Select(T);
        int MinIndex = res[0];
        int Second_MinIndex = res[1];
        T->data[i].weight = T->data[MinIndex].weight + T->data[Second_MinIndex].weight;
        T->data[i].lchild = MinIndex;
        T->data[i].rchild = Second_MinIndex;
        T->data[i].parent = 0;
        T->data[MinIndex].parent = i;
        T->data[Second_MinIndex].parent = i;
        T->length++;
    }
}
```

【先序遍历哈夫曼树】

```
void preorder(HFTree* T, int index) {
    if (index != -1) {
        printf("%d ", T->data[index].weight);
        preorder(T, T->data[index].lchild);
        preorder(T, T->data[index].rchild);
    }
}
```

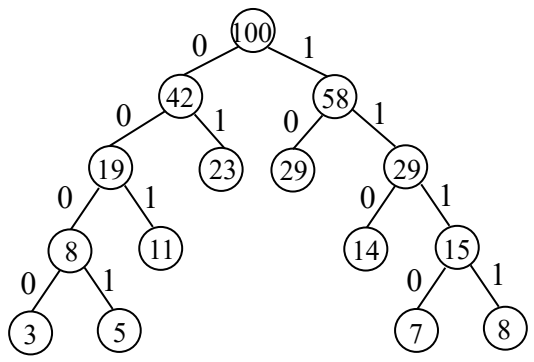
【函数调用】

```
int main() {
    int info[] = { 5,29,7,8,14,23,3,11 };
    HFTree* T = InitHFTree(info, 8);  //函数的第2个参数要传入数组中实际的元素个数
    CreateHuffmanTree(T);
    preorder(T, T->length - 1);
    free(T);
    return 0;
}
```

案例二：给定字符种类及权值，或是给定数据串（根据数据串中字符种类出现的频率来定权值），并给这些字符编码。

基本思路：

在构建完哈夫曼树后，我们要对每种字符进行编码，建议利用存储结构创建表（如下图）来理解比较直观，由于每种字符最终都在哈夫曼树的叶子结点上，因此用逆向求解的方法是最佳的，顺着叶子结点向根结点去“走”，下表中的结点编号 **1~8** 都在叶子结点上。



结点编号	weight	parent	lchild	rchild
1	3	**9**	**0**	**0**
2	5	**9**	**0**	**0**
3	7	**10**	**0**	**0**
4	8	**10**	**0**	**0**
5	11	**11**	**0**	**0**
6	14	**12**	**0**	**0**
7	23	**13**	**0**	**0**
8	29	**14**	**0**	**0**
9	8	**11**	**1**	**2**
10	15	**12**	**3**	**4**
11	19	**13**	**9**	**5**
12	29	**14**	**6**	**10**
13	42	**15**	**11**	**7**
14	58	**15**	**8**	**12**
15	100	**0**	**13**	**14**

以编号 **2**（weight=5）为例，它的双亲结点是编号 **9**（weight=8），且编号 **2**（weight=5）是编号 **9**（weight=8）的右子树，故编码为 1；

编号 **9**（weight=8）的双亲结点是编号 **11**（weight=19），且编号 **9**（weight=8）是编号 **11**（weight=19）的左子树，故编码为 0；

编号 **11**（weight=19）的双亲结点是编号 **13**（weight=42），且编号 **11**（weight=19）是编号 **13**（weight=42）的左子树，故编码为 0；

编号 **13**（weight=42）的双亲结点是编号 **15**（weight=100），且编号 **13**（weight=42）是编号 **15**（weight=100）的左子树，故编码为 0。

将编码 1000 倒置为 0001，最终我们得到编号 **2** 的编码为 0001。其它叶子结点同理。

代码如下：

注意：其它代码（包括结构体定义、哈夫曼树初始化、哈夫曼树构建等等）与案例一中的完全一致，故省略，以下为新增功能代码。

【哈夫曼编码创建】

```
char** CrtHuffmanCode(HFTree* T, int n) { //传入参数为构建好的哈夫曼树和叶结点个数
    char* cd = (char*)malloc(sizeof(char) * n);       //暂存数组
    char** code = (char**)malloc(sizeof(char*) * n); //最终保存数组
    for (int i = 0;i < n;i++) {
        int start = n - 1;
        cd[start] = '\0';
        int child = i;
        int parent = T->data[child].parent;
        while (parent != 0) {
            if (T->data[parent].lchild == child) {
                start--;
                cd[start] = '0';
            } else {
                start--;
                cd[start] = '1';
            }
            child = parent;
            parent = T->data[child].parent;
        }
        code[i] = (char*)malloc(sizeof(char) * (n - start));
        strcpy(code[i], &cd[start]);  //将求得的编码拷贝到最终保存数组当前行中(库备函数)
    }
    free(cd);      //释放掉暂存数组的临时空间
    cd = NULL;
    return code;   //将最终保存数组返回
}
```

【打印哈夫曼编码】

```
void PrintCode(char** savecode, char* s, int n) {
    for (int i = 0; i < n; i++) {
        printf("%c: %s\n", s[i], savecode[i]);
}}
```

【函数调用】

```
int main() {
    char s[] = { 'A', 'B', 'C', 'D','E', 'F', 'G', 'H' };
    int info[] = { 5,29,7,8,14,23,3,11 }; //对应字符的出现频率(权值)
    HFTree* T = InitHFTree(info, 8); //函数的第 2 个参数要传入数组中实际的元素个数
    CreateHuffmanTree(T);            //构建哈夫曼树
    char** code = CrtHuffmanCode(T, 8);  //使用前提是已经构建好哈夫曼树了
    PrintCode(code, s, 8);
```

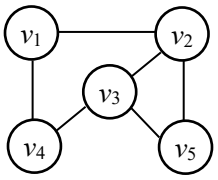
```
    free(T);
    return 0;
}
```

十二、图

1. 图的存储结构

（1）邻接矩阵（**注意：**一般用于表示稠密图）

〔I〕**无向图**



邻接矩阵 A.arcs[i][j]=

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	1	0
v_2	1	0	1	0	1
v_3	0	1	0	1	1
v_4	1	0	1	0	0
v_5	0	1	1	0	0

[1]基本概念：

①对角线上是每一个顶点与自身之间的关系，没有到自身的边，所以对角线上为 0。

②无向图的邻接矩阵是对称的。

③两个顶点之间如果有边的话，那么两个顶点互为邻接关系，值为 1。

④顶点 i 的度 = 第 i 行(列)中 1 的个数。

⑤完全图（完全图指任意两个顶点间都有边，分无向与有向，有向图是双向效果）的邻接矩阵的对角元素为 0，其余为 1。

⑥无向图和无向网的区别在于无向图没有权值，而无向网有权值。（**注：**带权值的图称为网）

[2]代码实现：

【预编译内容】

```
//宏定义
#define bool char
#define true 1
#define false 0
#define MaxInt 99999          //表示无穷大
#define MVNum 100             //最大顶点数
//自定义数据类型
typedef char VerTexType;      //假设顶点的数据类型为字符型
typedef int ArcType;          //假设边的权值类型为整型
```

【结构体】

```
typedef struct{
    VerTexType vexs[MVNum];         //顶点表
    ArcType arcs[MVNum][MVNum];    //邻接矩阵
    int vexnum, arcnum;             //图的当前顶点数和边数
}AMGraph;
```

【定位】(确定特定边所依附的两个顶点在G中的位置)

```
int LocateVex(AMGraph G, VerTexType v) {
    for (int i = 0; i < G.vexnum; i++) {
        if (G.vexs[i] == v) {
            return i;
    }}
    return -1;
}
```

【创建无向网】

（**注意：**创建无向图也是同理，只需要在初始化邻接矩阵时，修改 G->arcs[i][j] = MaxInt 为 G->arcs[i][j] = 0; 定义变量 w 为 1 并赋值给相应的边即可）

```
bool CreateUDN(AMGraph* G) {
    printf("输入总顶点数和总边数：");
    scanf("%d%d", &G->vexnum, &G->arcnum);
    for (int i = 0; i < G->vexnum; i++) {
        printf("输入第%d 个顶点的名称：", i + 1);
        scanf(" %c", &G->vexs[i]);
    }

    for (int i = 0; i < G->vexnum; i++) {
        for (int j = 0; j < G->vexnum; j++) {
            G->arcs[i][j] = MaxInt;         //初始化邻接矩阵
        }
    }

    for (int k = 0; k < G->arcnum; k++) {
        int w; VerTexType v1, v2;
        printf("输入第%d 条边所依附的顶点及权值：", k + 1);
        scanf(" %c %c %d", &v1, &v2, &w);
        getchar();

        int i = LocateVex(*G, v1);
        int j = LocateVex(*G, v2);
        if (i == -1 || j == -1) {
            printf("输入的顶点有误\n");
            return false;
        }

        G->arcs[i][j] = w;
        G->arcs[j][i] = G->arcs[i][j];     //对称边的权值设置
    }
    return true;
}
```

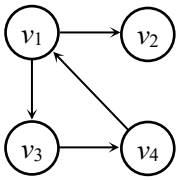
【输出邻接矩阵】

```
void PrintGraph(AMGraph G) {
    printf("\t");
    for (int i = 0;i < G.vexnum;i++)
        printf("%c\t", G.vexs[i]);
    printf("\n");

    for (int i = 0; i < G.vexnum; i++) {
        printf("%c\t", G.vexs[i]);
        for (int j = 0; j < G.vexnum; j++)
            printf("%d\t", G.arcs[i][j]);
        printf("\n");
}}
```

【函数调用】

```
int main() {
    AMGraph G;
    CreateUDN(&G);
    PrintGraph(G);
    return 0;
}
```

〔II〕**有向图**



邻接矩阵 B.arcs[i][j]=

	v_1	v_2	v_3	v_4
v_1	0	1	1	0
v_2	0	0	0	0
v_3	0	0	0	1
v_4	1	0	0	0

[1]基本概念：

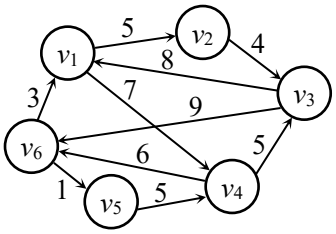
①在有向图的邻接矩阵中，**第 i 行含义：**以结点 v_i 为尾的弧(即出度边)，顶点的出度=第 i 行元素之和；**第 i 列含义：**以结点 v_i 为头的弧(即入度边)，顶点的入度=第 i 列元素之和。顶点的度=第 i 行元素之和+第 i 列元素之和。

②有向图的邻接矩阵**可能**是不对称的。

③有向图和有向网的区别在于有向图没有权值，而有向网有权值。(**注：**带权值的图称为网)

[2]补充知识：

网（实质是有权图，分**有向网**和**无向网**）的邻接矩阵表示法如下：



邻接矩阵 C.arcs[i][j]=

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	∞	5	∞	7	∞	∞
v_2	∞	∞	4	∞	∞	∞
v_3	8	∞	∞	∞	∞	9
v_4	∞	∞	5	∞	∞	6
v_5	∞	∞	∞	5	∞	∞
v_6	3	∞	∞	∞	1	∞

C.arcs[i][j]= $\begin{cases} w_{ij} & \langle v_i, v_j\rangle \text{或} (v_i, v_j) \in E \\ \infty & \text{无边(弧)} \end{cases}$

[3]代码实现：

略（有向网跟无向网的创建原理完全一致，有向图跟无向图的创建原理也完全一致，把对称边的权值设置这一步 `G->arcs[j][i] = G->arcs[i][j]`删除即可）

〔III〕**邻接矩阵表示法的优缺点**

[1]优点：

①方便检查任意一对顶点间是否存在边。

②方便找任一顶点的所有“邻接点”（有边直接相连的顶点）。

③方便计算任一顶点的“度”（从该点发出的边数为“出度”，指向该点的边数为“入度”）。

（**注意：无向图**中对应行（或列）非 0 元素的个数为该顶点的“度”；**有向图**中对应行非 0 元素的个数是“出度”，对应列非 0 元素的个数是“入度”）

[2]缺点：

①不便于增加和删除顶点。

②邻接矩阵的空间复杂度为 $O(n^2)$，跟其有的边的条数无关，只与其顶点数有关，无论边少还是边多，空间复杂度都为 $O(n^2)$，由于存稀疏图（点很多而边很少）有大量无效元素，故导致空间浪费。

③浪费时间（统计稀疏图中一共有多少条边，因为必须遍历所有元素）。

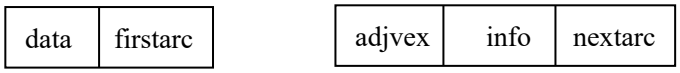
（2）邻接表（**注意：**一般用于表示稀疏图）

邻接表是图的一种链式存储结构。在邻接表中，对图中每一个顶点 v_i 建立一个单链表，把与 v_i 相邻接的顶点放在这个链表中。

邻接表由两部分组成：表头结点表和边表。

表头结点表：所有表头结点以顺序结构的形式存储，以便可以随机访问任一顶点的边链表，表头结点包括**数据域**(data)和**链域**(firstarc)两部分。其中，**数据域**用于存储顶点 v_i 的名称或其它有关信息；而**链域**用于指向链表中第一个结点（与顶点 v_i 邻接的第一个邻接点）。

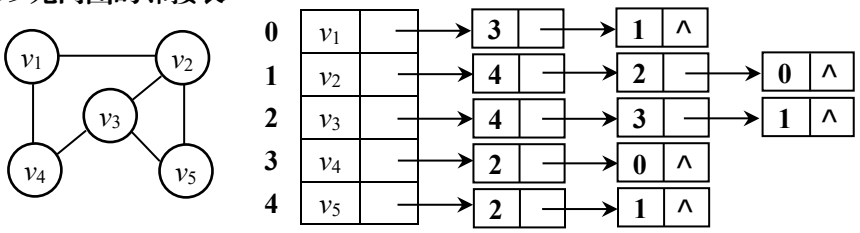
边表：由表示图中顶点间关系的 n 个边链表组成。边链表中边结点包括**邻接点域**(adjvex)、**数据域**(info)和**链域**(nextarc)三部分。其中，**邻接点域**指示与顶点 v_i 邻接的点在图中的位置；**数据域**存储和边相关的信息，如权值等；**链域**指示与顶点 v_i 邻接的下一条边的结点。



（表头结点）　　（边结点）

值得注意的是，一个图的邻接矩阵表示是唯一的，但其邻接表表示不唯一，这是因为邻接表表示中，各边表结点的链接次序取决于建立邻接表的算法，以及边的输入次序。

〔I〕**无向图的邻接表**



（**注意：**无向图通常不使用逆邻接表，因为无向图中的边是没有方向的。逆邻接表主要应用于有向图中，用来表示指向每个顶点的边。）

[1]基本概念：

①以上图为例，邻接表中的箭头所指的边结点中的数字表示与 v_i 相连接的顶点编号（邻接点域）。（例如：与 v_1 相连的是 v_4 和 v_2，那箭头就指向编号 **3** 和编号 **1**，如图中第一行所示）

②在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点个数。

[2]代码实现：

〔II〕**有向图的邻接表和逆邻接表**

在有向图中，边是有方向的，从一个顶点指向另一个顶点，所以需要逆邻接表来存储所有指向某个特定顶点的边，这在某些算法中非常有用，例如拓扑排序或计算入度等。

[1]基本概念：

[2]代码实现：

（3）十字链表

（4）邻接多重表

2. 图的遍历

（1）深度优先搜索（DFS）

[1]过程

[2]代码

（2）广度优先搜索（BFS）

[1]过程

[2]代码

3. 图的应用

下面介绍 4 种常用的应用算法：最小生成树、最短路径、拓扑排序、关键路径。

（1）最小生成树

〔I〕**普里姆** Prim **算法**（**方法：**找点）

[1]过程

先随便选定一个点，再找到该选定点的所有边，选取权值最小的一条边（包括边两端关联的点），将其提取出来；然后再找到与刚才选取的那条边两端的点相连的其它所有边，选取权值最小的一条边（包括边两端关联的点），将其提取出来与一开始所提取的合并，重复此步骤，最终得到最小生成树。特别注意，**在提取的过程中不能出现构成环的情况**，哪条边的加入构成了环就要舍去那条边。

[2]代码

〔II〕**克鲁斯卡尔** Kruskal **算法**（**方法：**找边）

[1]过程

先把权值最小的边提取出来（包括边两端关联的点），若有多条重复权值的边，要看这几条边是否构成环，若不构成环则都保留，若构成环，则哪条边的加入构成了环就要舍去那条边。重复以上步骤，最终直到所有顶点都被连在一起且没有环路，则此时最小生成树构成。

[2]代码

（2）最短路径

〔I〕**迪杰斯特拉** Dijkstra **算法**（从某个源点到其余各顶点的最短路径）

[1]过程

[2]代码

〔II〕**弗洛伊德** Floyd **算法**（求任意一对顶点之间的最短路径）

[1]过程

[2]代码

（3）拓扑排序

（4）关键路径

十三、查找

1. 线性表的查找

（1）顺序查找

（2）折半查找（二分查找）

（3）分块查找（索引顺序查找）

2. 树表的查找

（1）二叉排序树（又称二叉搜索树或二叉查找树）

[1]定义

（2）平衡二叉树（AVL 树）

[1]定义

[2]平衡调整方法

[3]代码实现

（3）B− 树

[1]定义

[2]B- 树的创建(插入)

[3]B- 树的删除

[4]B- 树的查找

（4）B+ 树

3. 散列表的查找（**注：**该部分考试内容主要涉及定性知识）

（1）基本概念

如果能在元素的存储位置和其关键字之间建立某种直接关系，那么在进行查找时，就无须作比较或只需作很少的比较，按照这种关系直接由关键字找到相应的记录，这就是**散列查找法**的思想。**散列查找法**又叫**杂凑法**或**散列法**。

下面给出散列法中常用的几个术语：

①**散列函数**和**散列地址**：在记录的存储位置 p 和其关键字 key 之间建立一个确定的对应关系 H，使 $p=H(key)$，称这个对应关系 H 为散列函数，p 为散列地址。

②**散列表**：一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常散列表的存储空间是一个一维数组，散列地址是数组的下标。

③**冲突**和**同义词**：对不同的关键字可能得到同一散列地址，即 $key_1 \neq key_2$，而 $H(key_1)=H(key_2)$，这种现象称为**冲突**。具有相同函数值的关键字对该散列函数来说称作**同义词**，key_1 与 key_2 互为同义词。

注意：通常，散列函数是一个多对一的映射，所以冲突是不可避免的（要将非常多个可能的标识符映射到有限的地址上，难免产生冲突），只能通过选择一个“好”的散列函数使得在一定程度上减少冲突。而一旦发生冲突，就必须采取相应措施及时予以解决。

综上所述，散列查找法主要研究两方面的问题：①如何构造散列函数；②如何处理冲突

（2）散列函数的构造法

[1]根据具体问题选用不同的散列函数，通常要考虑的因素如下：

①散列表的长度

②关键字的长度

③关键字的分布情况

④计算散列函数所需的时间（执行时间）

⑤记录的查找频率

[2]构造一个“好”的散列函数应遵循的两条原则如下：

①函数计算要简单，每一关键字只能有一个散列地址与之对应。

②函数的值域需在表长的范围内，计算出的散列地址的分布应均匀，尽可能减少冲突。

[3]构造散列函数的常用方法

①数字分析法（略，见教材）

②平方取中法（略，见教材）

③折叠法（略，见教材）

④直接定址法

⑤除留余数法

（3）处理冲突的方法

处理冲突的方法与散列表本身的组织形式有关。按组织形式的不同，处理冲突的方法通常分两大类：**开放地址法**和**链地址法**。下面以创建散列表为例，来说明处理冲突的方法。

[1]开放地址法

[2]链地址法（拉链法）

十四、排序

由于待排序记录的数量不同，使得排序过程中数据所占用的存储设备会有所不同。根据在排序过程中记录所占用的存储设备，可将排序方法分为两大类：一类是**内部排序**，指的是待排序记录全部存放在计算机内存中进行排序的过程；另一类是**外部排序**，指的是待排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。

（一）内部排序

1. 插入排序

基本思想：每一趟将一个待排序的记录按其关键字的大小插入已经排好序的一组记录的适当位置，直到所有待排序记录全部插入为止。

（1）直接插入排序

（2）折半插入排序

（3）希尔排序

2. 交换排序

基本思想：两两比较待排序记录的关键字，一旦发现两个记录不满足次序要求时则进行交换，直到整个序列全部满足要求为止。

（1）冒泡排序

（2）快速排序

3. 选择排序

基本思想：每一趟从待排序的记录中选出关键字最小的记录，按顺序将其放在已排好序的记录序列的最后，直到全部排完为止。

（1）简单选择排序（直接选择排序）

（2）树形选择排序（锦标赛排序）

（3）堆排序

4. 归并排序（将两个或两个以上的有序表合并成一个有序表的过程称为归并排序。）

基本思想：假设初始序列含有 n 个记录，则可将其看成 n 个有序的子序列，每个子序列的长度为 1，然后两两归并，得到$[n/2]$个长度为 2 或 1 的有序子序列；再两两归并，如此重复，直至得到一个长度为 n 的有序序列为止。

注：2-路归并排序是最为常见的归并排序法，是指将两个有序表合并成一个有序表的过程。

5. 分配排序

前述的各类排序法都建立在关键字比较的基础上，而分配排序不需要比较关键字的大小，它是根据关键字中各位的值，通过对待排序记录进行若干趟“分配”与“收集”来实现排序的，是一种借助于多关键字排序的思想对单关键字进行排序的方法。而基数排序是典型的分配排序法。

（二）外部排序