第八章　指针及其应用

指针是 C++语言中广泛使用的一种数据类型，运用指针编程是 C++语言最主要风格之一。利用指针变量可以表示各种数据结构，能很方便地使用数组和字符串，并能像汇编语言一样处理内存地址，从而编出精炼而高效的程序，指针极大地丰富了 C++语言的功能。学习指针是学习 C++语言最重要的一环，能否正确理解和使用指针是我们是否掌握 C++语言的一个标志。同时，指针也是 C++语言中最为困难的一部分，在学习中除了要正确理解基本概念，还必须要多编程、多上机调试，只要做到这些，指针也是不难掌握的。

第一节　指针变量

一、指针变量的定义、赋值

在使用指针之前要先定义指针，对指针变量的类型说明，一般形式为：

类型说明符　*变量名；

其中，*表示这是一个指针变量，变量名即为定义的指针变量名，类型说明符表示该指针变量所指向的变量的数据类型。先通过例子看看指针与普通的变量有什么不同。

1、普通变量定义

```
int  a=3;
```

定义了变量 a，是 int 型的，值为 3。内存中有一块内存空间是放 a 的值，对 a 的存取操作就是直接到这个内存空间存取。内存空间的位置叫地址，存放 3 的地址可以用取地址操作符“&”对 a 运算得到：&a。

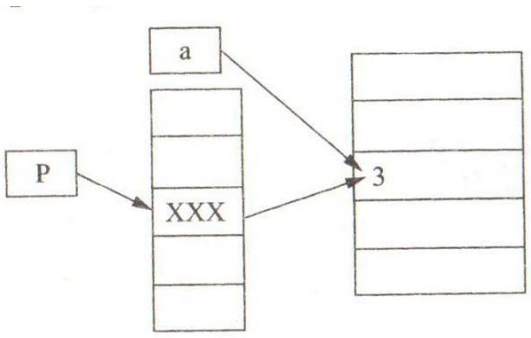
2、指针变量定义

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

定义了一个指针变量 p，p 指向一个内存空间，里面存放的是一个内存地址。现在赋值为 NULL（其实就是 0，表示特殊的空地址）。

3、给指针变量 p 赋值

```
P=&a;
```



即把 a 变量的内存空间地址（比如：XXX）给了 p。显然，直接对 p 存取，操作的是地址。通过这个地址间接地操作，才是整数 3。P 的间接操作要使用指针操作符“*”，即*p 的值才是 3。设有指向整型变量的指针变量 p，如要把整型变量 a 的地址赋予 p 可以有以下两种方式：

①指针变量初始化的方法

```
int a; int *p=&a;
```

②赋值语句的方法

```
int a; int *p; p=&a;
```

不允许把一个数赋予指针变量，故如下的赋值是错误的：int *p; p=1000;。被赋值的指针变量前不能再加“*”说明符，故如下的赋值也是错误的：*p=&a;。

指针的几个相关操作说明表

说　明	样 例
指针定义： 　　类型说明符　*指针变量名；	int a=10; int *p;
取地址运算符： 　　&	p=&a;
间接运算符： 　　*	*p=20;
指针变量直接存取的是内存地址	cout<<p; 结果可能是：0x4097ce
间接存取的才是储存类型的值	cout<<*p; 结果是：20

指针变量同普通变量一样，使用之前不仅要定义说明，而且必须被赋值具体的值，未经赋值的指针变量不能使用。如定义了 int a; int *p=&a;，则*p 表示 p 指向的整型变量，而 p 中存放的是变量 a 占用单元的起始地址，所以*p 实际上访问了变量 a，也就是说*p 与 a 等价。下面举一个简单的指针使用的例子：

【例 1】 输入两个不同的数，通过指针对两个数进行相加和相乘，并输出。

```
#include<cstdio>
#include<iostream>
using namespace std;
int main()
{
    int a,b,s,t,*pa,*pb;
    pa=&a;pb=&b;
    a=10;b=20;
    s=*pa+*pb;
    t=*pa**pb;
    printf("a=%d,b=%d\n",*pa,*pb);
    printf("s=%d,t=%d\n",s,t);
    return 0;
}
```

输出：

```
a=10,b=20
s=30,t=200
```

二、指针的引用与运算

一般的，我们可以这样看指针(int *p)与普通变量(int a)的对应关系：

p	----	&a
*p	----	a
*p=3	----	a=3

下面介绍指针的一些运算。

1、指针变量的初始化

指针的几个初始化操作说明表

	方 法	说 明
1	int *p=NULL;	NULL 是特殊的地址 0，叫零指针
2	int a; int *p=&a;	p 初始化为 a 的地址
3	int *p=new(int);	申请一个空间给 p，*p 内容不确定

要强调的是，对于定义的局部指针变量，其内容（地址）是随机的，直接对它操作可能会破坏程序或系统内存的值，引发不可预测的错误。所有编程中指针变量要保证先初始化或赋值，给予正确的地址再使用。

2、指针变量的+、-运算

指针变量的内容是内存地址，它有两个常用的运算：加、减，这两个运算一般都是配合数组操作的。

【例 2】输入 N 个整数，使用指针变量访问输出。

```
#include<cstdio>
using namespace std;
int a[101],n;
int main()
{
    scanf("%d",&n);
    for (int i=1; i<=n; i++)
    scanf("%d",&a[i]);
    int p=&a[1];          //定义指针变量 int p,初始化为数组开始元素的地址,即 a[1];
    for (int i=1; i<=n; i++)
    {
        printf("%d ",*p);
        p++;              //p 指向下一个数，详见说明
    }
    return 0;
}
```

输入：4
2 1 6 0

输出：2 1 6 0

【说明】

“p++”的意思是“广义的加 1”，不是 p 的值（地址）加 1，而是根据类型 int 增加 sizeof（int），即刚好“跳过”一个整数的空间，达到下一个整数。

类似的：

①、“p--”就是向前“跳过”一个整数的空间，达到前一个整数。

②、（p+3）就是指向后面第 3 个整数的地址。

3、无类型指针

有时候，一个指针根据不同的情况，指向的内容是不同类型的值，我们可以先不明确定

义它的类型，只是定义一个无类型的指针，以后根据需要再用强制类型转换的方法明确它的类型。

【例 3】无类型指针运用举例。

```
#include<iostream>
using namespace std;
int a=10;
double b=3.5;
void *p;
int main()
{
    p=&a;                          //p 的地址赋值
    cout<<*(int*)p<<endl;          //必须明确 p 指向的空间的数据类型，详见说明
    p=&b;
    cout<<*(double*)p<<endl;
    return 0;
}
```

输出：10
3.5

【说明】

必须明确 p 指向的空间的数据类型，类型不一样的不仅空间大小不相同，储存的格式也不同。如果把 cout<<*(double*)p<<endl;改成 cout<<*(long long*)p<<endl;输出的结果将是：4615063718147915776。

4、多重指针

既然指针是指向其他类型的，指针本身也是一种类型。

C++允许递归地指针指向指针的指针——多重指针。

【例 4】双重指针运用举例。

```
#include<cstdio>
using namespace std;
int a=10;
int *p;
int **pp;                                  //定义双重指针
int main()
{
    p=&a;                                  //将 p 指向 a
    pp=&p;                                 //将 pp 指向 p
    printf("%d=%d=%d\n",a,*p,**pp);  //**pp 通过 2 次间接访问了 a 的变量的值 10
    return 0;
}
```

输出：

10=10=10

【说明】

多重指针除了可以多次“间接”访问数据，OI 上主要的应用是动态的多维数组，这个强大的功能将在后面专门介绍。

第二节　指针与数组

一、指针与数组的关系

指向数组的指针变量称为数组指针变量。一个数组是一块连续的内存单元组成的，数组名就是这块连续内存单元的首地址。一个数组元素的首地址就是指它所占有的几个内存单元的首地址。一个指针变量即可以指向一个数组，也可以指向一个数组元素，可把数组名或第一个元素的地址赋予它。如要使指针变量指向第 i 号元素，可以把 i 元素的首地址赋予它，或把数组名加 i 赋予它。

设有数组 a，指向 a 的指针变量为 pa，则有以下关系：pa、a、&a[0]均指向同一单元，是数组 a 的首地址，也是 0 号元素 a[0]的首地址。pa+1、a+1、&a[1]均指向 1 号元素 a[1]。类推可知 pa+i、a+i、&a[i]指向 i 号元素 a[i]。pa 是变量，而 a，&a[i]是常量，在编程时应予以注意。

二、指向数组的指针

数组指针变量说明的一般形式为：

类型说明符　*指针变量名

其中类型说明符表示所指数组的类型，从一般形式可以看出，指向数组的指针变量和指向普通变量的指针变量的说明是相同的。

引入指针变量后，就可以用两种方法访问数组元素了，

例如定义了 int a[5];int *pa=a;

第一种方法为下标法，即用 pa[i]形式访问 a 的数组元素。

第二种方法为指针法，即采用*(pa+i)形式，用间接访问的方法来访问数组元素。

【例 5】scanf 使用数组名，用数组名或指针访问数组。

```
#include<cstdio>
using namespace std;
int main()
{
    int a[5],i,*pa=a;                  //定义整型数组和指针，*pa=a 可以在下一行 pa=a;
    for (i=0;i<5;i++)
      scanf("%d",a+i);                  //可写成 pa+i 和 &a[i]
    for (i=0;i<5;i++)
      printf("a[%d]=%d\n",i, *(a+i)); //指针访问数组,可写成*(pa+i)或 pa[i]或 a[i]
    return 0;
}
```

输入：1 2 3 4 5

输出：a[0]=1
a[1]=2
a[2]=3
a[3]=4
a[4]=5

【说明】

①、直接拿 a 当指针用，a 指向数组的开始元素，a+i 是指向数组的第 i 个元素的指针。

②、指针变量 pa 是变量，可以变的。但数组 a 是静态的变量名，不可变，只能当做常量指针使用。例如：p=p+2;是合法的，a=a+2;是非法的。

③、最早在使用标准输入 scanf 时就使用了指针技术，读入一个变量时要加取地址运算符’&’传递给 scanf 一个指针。对于数组，可以直接用数组名当指针。

三、指针也可以看成数组名

指针可以动态申请空间，如果一次申请多个变量空间，系统给的地址是连续的，就可以当成数组使用，这就是传说中的动态数组的一种。

【例 6】动态数组，计算前缀和数组。b 是数组 a 的前缀和的数组定义：
b[i]=a[1]+a[2] +…+a[i]，即 b[i]是 a 的 i 个元素的和。

```
#include<cstdio>
using namespace std;
int n;
int *a;                          //定义指针变量 a，后面直接当数组名使用
int main()
{
    scanf("%d",&n);
    a=new int[n+1];              //向操作系统申请了连续的 n+1 个 int 型的空间
    for (int i=1; i<=n; i++)
        scanf("%d",&a[i]);
    for (int i=2; i<=n; i++)
        a[i]+=a[i-1];
    for (int i=1; i<=n; i++)
        printf("%d ",a[i]);
    return 0;
}
```

输入：5
1 2 3 4 5
输出：1 3 6 10 15

【说明】

动态数组的优点：在 OI 中，对于大数据可能超空间的情况是比较纠结的事，用小数组只的部分分，大数组可能爆空间（得 0 分）。使用“动态数组”，可以在确保小数据没问题的前提下，尽量满足大数据的需求。

第三节　指针与字符串

一、字符串的表示形式

在C++中，我们可以用两种方式访问字符串。

（1）用字符数组存放一个字符串，然后输出该字符串。

```
int main()
{
   char str[]="I love China!";
   printf("%s\n", str);
}
```

（2）用字符指针指向一个字符串。可以不定义字符数组，而定义一个字符指针。用字符指针指向字符串中的字符。

```
int main()
{
   char *str="I love China!";
   printf("%s\n", str);
}
```

在这里，我们没有定义字符数组，而是在程序中定义了一个字符指针变量str，用字符串常量"I love China!"，对它进行初始化。C++对字符串常量是按字符数组处理的，在内存中开辟了一个字符数组用来才存放该字符串常量。对字符指针变量初始化，实际上是把字符串第1个元素的地址（即存放字符串的字符数组的首元素地址）赋给str。有人认为str是一个字符串变量，以为在定义时把"I love China!"这几个字符赋给该字符串变量，这是不对的。

实际上，char *str="I love China!";

等价于：char *str;

　　str="I love China!";

可以看到，str被定义为一个指针变量，指向字符型数据，**请注意**它只是指向了一个字符变量或其他字符类型数据，不能同时指向多个字符数据，更不是把"I love China!"这些字符存放到str中（指针变量只能存放地址）。只是把"I love China!"的第一个字符的地址赋给指针变量str。

在输出时，要用：printf(“%s\n”, str);

其中“%s”是输出字符串时所用的格式符，在输出项中给出字符指针变量名，则系统先输出它所指向的一个字符数据，然后自动是str加1，使之指向下一个字符，然后再输出一个字符……如此知道遇到字符串结束标志“\0”为止。

注意：可以通过字符数组名或者字符指针变量输出一个字符串。而对一个数值型数组，是不能企图用数组名输出它的全部元素的。

例如：

```
int i[10];
……
printf（”%d\n”, i);
```

这样是不行的，只能逐个输出。显然 %s可以对一个字符串进行整体的输入和输出。

二、字符串指针作函数参数

将一个字符串从一个函数传递到另外一个函数，可以用地址传递的方法，即用字符数组名作参数或用指向字符的指针变量做参数。在被调用的函数中可以改变字符串内容，在主调函数中可以得到改变了的字符串。

【例 8】输入一个长度最大为 100 的字符串，以字符数组的方式储存，再将字符串倒序储存，输出倒序储存后的字符串。(这里以字符指针为函数参数)

```
#include<cstdio>
#include<cstring>
using namespace std;
void swapp(char &a,char &b) //交换两个字符的位置
{
    char t;
    t=a;
    a=b;
    b=t;
}
void work(char* str)
{
    int len=strlen(str);    //strlen(str)这个函数返回的是 str 的长度，
                            //需包含头文件 cstring
                            //这个函数的原型是"size_t strlen(const char* str)"
    for (int i=0; i<=len/2; ++i)
      swapp(str[i],str[len-i-1]);
}
int main()
{
    char s[110];
    char *str = s;
    gets(s);
    work(str);
    printf("%s",s);
    return 0;
}
```

输入：!anihC evol I
输出：I love China!

第四节　指针与函数

一、指针作为函数参数

指针可以作为函数的参数。在函数章节中，我们把数字作为参数传入函数中，实际上就是利用了传递指针（即传递数组的首地址）的方法。通过首地址，我们可以访问数组中的任何一个元素。

对于指向其他类型变量的指针，我们可以用同样的方式处理。

例如，我们编写如下一个函数，用于将两个整型变量的值交换。

```
void swap(int *x,int *y)
{
    int t=*x;
    *x=*y;
    *y=t;
}
```

这时，我们在其他函数中可以使用这个函数：

```
int a=5,b=3;
swap(&a,&b);
printf(“a=%d,b=%d”,a,b);
```

输出：a=3,b=5

在这个过程中，我们先将 a 和 b 的地址传给函数，然后在函数中通过地址得到变量 a 和 b 的值，并且对它们进行修改。当退出函数时，a 和 b 的值就已经交换了。

这里有一点值得我们注意。看如下这个过程：

```
void swap(int x,int y)
{
    int t=x;
    x=y;
    y=t;
}
```

我们调用了 swap(a,b);然而这个函数没有起作用，没有将 a 和 b 的值互换。

为什么呢？因为这里在传入变量 a 和 b 的时候，是将 a 的值赋值给函数中的形参 x，将 b 赋值给形参 y。这里接下来的操作就完全与 a 和 b 无关了，函数将变量 x 和 y 的值互换，然后退出函数。这里没有像上面例子那样传入指针，所以无法对传进来的变量进行修改。

将指针传入函数与将变量传入函数的区别在于：前者是通过指针来使用或修改传入的变量；而后者是将传入的变量的值赋给新的变量，函数对新的变量进行操作。

同理，对 scanf()函数而言，读取变量的时候我们要在变量之前加&运算符，即将指针传入函数。这是由于 scanf()函数通过指针将读取的值返回给引用的变量，没有&，就无法进行正常的读取操作。

【例 9】编写一个函数，将三个整型变量排序，并将三者中的最小值赋给第一个变量，次小值赋给第二个变量，最大值赋给第三个变量。

```
#include<cstdio>
using namespace std;
void swap(int *x,int *y)
```

```
{
    int t=*x;
    *x=*y;
    *y=t;
}
void sort(int *x,int *y,int *z)
{
    if (*x>*y) swap(x,y);
    if (*x>*z) swap(x,z);
    if (*y>*z) swap(y,z);
}
int main()
{
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);
    sort(&a,&b,&c);
    printf("%d %d %d",a,b,c);
    return 0;
}
```

输入：2 3 1
输出：1 2 3

二、函数返回指针

一个函数可以返回整数值、字符值、实型值等，也可以返回指针联系的数据（即地址）。

返回指针值的函数的一般定义形式为：

类型名 * 函数名（参数列表）；

例如：

int *a（int a,int b）

a 是函数名，调用它后得到一个指向整型数据的指针（地址）。x 和 y 是函数 a 的形参，为整型。

注意：在*a 的两侧没有括号；在 a 的两侧分别为*运算符和（）运算符，由于（）的优先级高于*，因此 a 先于（）结合。在函数前面有一个*，表示此函数是返回指针类型的函数。最前面的 int 表示返回的指针指向整型变量。对初学 C++语言的人来说，这种定义形式可能不太习惯，容易弄错，用时要十分小心。

【例 10】编写一个函数，用于在一个包含 N 个整数的数组中找到第一个质数，若有则返回函数的地址；否则返回 NULL(空指针)。

```
#include<cmath>
#include<cstdio>
using namespace std;
int n,a[10001];
bool isprime(int n)                    //判断 n 是不是素数
{
    if (n<2) return false;
    if (n==2) return true;
```

```
    for (int i=2; i<=sqrt(n); ++i)
      if (n%i==0)
        return false;
    return true;
}
int* find()
{
    for (int i=1; i<=n; ++i)
      if (isprime(a[i]))
        return &a[i];                    //这句也可以写成：return a+i;
    return NULL;                         //如果找不到则返回 NULL(空指针)
}
int main()
{
    scanf("%d",&n);
    for (int i=1; i<=n; ++i)
      scanf("%d",&a[i]);
    int *p=find();
    if (p!=NULL)
      printf("%d\n%d\n",p,*p);           //输出这个素数的地址和它本身
    else
      printf("can't find!");
    return 0;
}
```

输入：

```
7
1 6 9 2 3 4 5
```

输出：

```
(可能是)4214864
        2
```

三、函数指针和函数指针数组

一个指针变量通过指向不同的整数变量的地址，就可以对其他的变量操作。

程序中不仅数据是存放在内存空间中，代码也同样存放在内存空间里。具体讲，C++的函数也保存在内存中，函数的入口地址也同样可以用指针访问。

另一方面，有些函数在编写时对要调用的辅助函数尚未确定，在执行时才能根据情况为其传递辅助函数的地址。比如 sort 函数的调用：“sort(a,a+n,cmp);”其中的比较函数 cmp 是我们根据需要转给 sort 的(也可能是 cmp1，cmp2 等)，其实就是传递了函数指针。

下面我们来看一个具体例子。

【例 11】使用函数指针调用函数示例。

```
#include<iostream>
using namespace std;
int t(int a)
```

```
{
    return a;
}
int main()
{
    cout<<t<<endl;              //显示函数地址
    int (*p)(int a);            //定义函数指针变量 p
    p=t;                        //将函数 t 的地址赋给函数指针 p
    cout<<p(5)<<','<<(*p)(10)<<endl;
          //输出 p(5)是 C++的写法,(*p)(10)是兼容 C,这是使用 p 来调用函数的两种方法
    return 0;
}
```

输出：

```
1
5,10
```

函数指针的基本操作有 3 个：

⑴声明函数指针。

声明要指定函数的返回类型以及函数的参数列表，和函数原型差不多。

例如，函数的原型是：int test(int);

相应的指针声明就是：int (*fp)(int);

要注意的是，不能写成：int *fp(int);

这样计算机编译程序会处理成声明一个 fp(int)的函数，返回类型是 int* 。

⑵获取函数的地址。

获取函数的地址很简单，只要使用函数名即可，例如，fp=test;

这表明函数名和数组名一样，可以看作是指针。

⑶使用函数指针来调用函数。

类似普通变量指针，可以用（*fp）来间接调用指向的函数。但 C++也允许像使用函数名一样使用 fp，虽然有争议，但 C++确实是支持了。

函数指针还有另一种结合 typedef 的声明方式，如**例 X** 所示。

【例 12】使用 typedef 定义函数指针示例。

```
#include<iostream>
using namespace std;
int sum(int a,int b)
{
    return a+b;
}
typedef int (*LP)(int,int);  //定义了一个函数指针变量类型 LP
int main()
{
    LP p=sum;     //定义了一个 LP 类型的函数指针 LP，并赋值为 sum
    cout<<p(2,5);//使用 p 来调用函数，实参为 2 和 5，调用 sum 函数，输出返回值 7
    return 0;
```

```
}
```

输出：7

在软件开发编程中，函数指针的一个广泛应用是菜单功能函数的调用。通常选择菜单的某个选项都会调用相应的功能函数，并且有些软件的菜单会根据情况发生变化（上下文敏感）。如果使用 swith/case 或 if 语句处理起来会比较复杂、冗长，不利于代码的维护，可以考虑使用函数指针数组方便灵活地实现。

【例 13】使用函数指针数组，模拟菜单功能实现方法示例。

```
#include<iostream>
using namespace std;
void t1() { cout<<"test1"; }
void t2() { cout<<"test2"; }
void t3() { cout<<"test3"; }
void t4() { cout<<"test4"; }
void t5() { cout<<"test5"; }
typedef void(*LP)();          //定义了一个函数指针变量类型 LP
int main()
{
    LP a[]={t1,t2,t3,t4,t5};  //定义了一个 LP 类型的函数指针数组 a，并初始化
    int x;
    cin>>x;
    a[x]();                   //使用 a[x]()来调用选择的函数
    return 0;
}
```

输入：2

输出：test3

第五节　结构体指针

一、结构体指针的定义与使用

当一个指针变量用来指向一个结构体变量时，称之为**结构体指针变量**。

结构体指针变量的值是所指向的结构体变量的起始地址。通过结构体指针即可访问该结构体变量，这与数组指针和函数指针的情况是相同的。

结构体指针变量定义的一般形式：

结构体名 *结构体指针变量名

当然也可以在定义结构体的同时定义这个结构体指针变量。

例如：（定义一个结构体（类型为自己定义的 student）指针变量 p）

```
struct student
{
    char name[20];
    char sex;
    float score;
} *p;
```

也可写成

```
struct student
{
    char name[20];
    char sex;
    float score;
};
student *p;
```

与前面讨论的各类指针变量相同，结构体指针变量也必须要赋值后才能使用。赋值是把结构体变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。

例如：如果 p 是被定义为 student 类型的结构体指针变量，boy 是被定义为 student 类型的结构体变量，则：p=&boy 是正确的，而 p=&student 是错误的。

引用结构体指针变量指向的结构体变量的成员的方法如下：

①、**指针名->成员名**

②、**(*指针名).成员名**

例如：

(*p).score 与 p->score 是等价的。

【例 14】结构体指针运用举例。

```
#include<cstdio>
using namespace std;
struct student
{
    char name[20];
    char sex;
    int score;
} s[3]={{"xiaoming",'f',356},
        {"xiaoliang",'f',350},
        {"xiaohong",'m',0}};
int main()
```

```
{
        student *p;
        printf("Name        Sex    Score\n");
        for (p=s; p<s+3; p++)
        printf("%-9s%3c%7d\n",p->name,p->sex,p->score);
        return 0;
}
```

输出:

```
Name        Sex    Score
xiaoming    f      356
xiaoliang   f      350
xiaohong    m      0
```

【说明】

这里 p++起到移动指针的作用，随着 p 的变化，输出数组不同元素内容。

二、自引用结构

在一个结构体内部包含一个类型为该结构体本身的成员是否合法呢？

```
struct stu
{
    char name[20];
    int age,score;
    stu p;
};
```

这种类型的自引用是非法的，因为成员 p 是另一个完整的结构，其内部还将包含它自己的成员 p。这第 2 个成员又是一个完整的结构，它还将包含自己的成员 p……这样重复下去就永无止境了。这有点像永远不会终止的递归程序。但下面这个程序是合法的：

```
struct stu
{
    char name[20];
    int age,score;
    stu *p;
};
```

这个声明和前面那个声明的区别在于 p 现在是一个**指针**而不是结构体。编译器在结构体的长度确定之前就已经知道指针的长度，所以这种类型的自引用是合法的。

当一个结构体中有一个或是多个成员是指针，它们所指向的类型就是本结构体类型时，通常这种结构体称为“引用自身的结构体”，即“**自引用结构**”。这种自引用结构是实现其他一些结构的基础。

自引用结构在动态数据结构中有重要作用，甚至可以说，自引用结构是 C/C++语言实现动态数据结构的基石。包括动态的链表、堆、栈、树，无不是自引用结构的具体实现。

例如，下面的定义就可以在实际操作中建立起一个链表。

```
struct node
{
    int x,y;
    node *next;
} point;
```

在下一节将对链表结构进行深入的研究。

第六节　链表结构

【存储方式的分类】：顺序存储结构和链式存储结构；

【顺序存储结构】：在（子）程序的说明部分就必须加以说明，以便分配固定大小的存储单元，直到（子）程序结束，才释放空间。因此，这种存储方式又称为静态存储。所定义的变量相应的称为静态变量。它的优缺点如下：

1、优点：可以通过一个简单的公式随机存取表中的任一元素，逻辑关系上相邻的两个元素在物理位置上也是相邻的，且很容易找到前趋与后继元素；

2、缺点：在线性表的长度不确定时，必须分配最大存储空间，使存储空间得不到充分利用，浪费了宝贵的存储资源；线性表的容量一经定义就难以扩充；在插入和删除线性表的元素时，需要移动大量的元素，浪费了时间；

【链式存储结构】：在程序的执行过程中，通过两个命令向计算机随时申请存储空间或随时释放存储空间，以达到动态管理、使用计算机的存储空间，保证存储资源的充分利用。这样的存储方式称为动态存储。所定义的变量称为动态变量。它的优点如下：

可以用一组任意的存储单元（这些存储单元可以是连续的，也可以不连续的）存储线性表的数据元素，这样就可以充分利用存储器的零碎空间；

【概念】：为了表示任意存储单元之间的逻辑关系，对于每个数据元素来说，除了要存储它本身的信息（数据域、data）外，还要存储它的直接后继元素的存储位置（指针域、link或next）。我们把这两部分信息合在一起称为一个“结点node”。

1、N个结点链接在一起就构成了一个链表。N=0时，称为空链表。

2、为了按照逻辑顺序对链表中的元素进行各种操作，我们需要定义一个变量用来存储整个链表的第一个结点的物理位置，这个变量称为“头指针、H或head”。也可以把头指针定义成一个结点，称为“头结点”，头结点的数据域可以不存储任何信息，也可以存储线性表的长度等附加信息，头结点的指针域（头指针）存储指向第一个结点的指针，若线性表为空表，则头结点的指针域为空（NIL）。由于最后一个元素没有后继，所以线性表中最后一个结点的指针域为空（NIL）。

3、 由于此链表中的每个结点都只包含一个指针域，故称为“线性链表或单链表”。

（一）单链表的定义

1．类型和变量的说明

```
struct Node
{
   int data;
   Node *next;
};
Node *p;
```

2．申请存储单元　　//动态申请、空间大小由指针变量的基类型决定

```
p=new Node;
```

3．指针变量的赋值

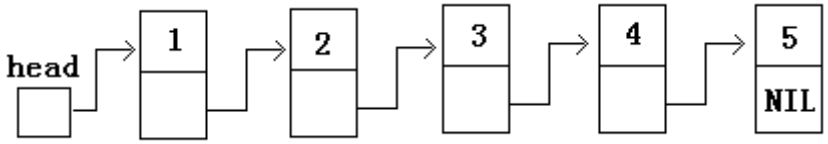
```
指针变量名=NULL;
```

//初始化，暂时不指向任何存储单元

如何表示和操作指针变量？不同于简单变量（如 A=0;），c++规定用“指针变量名->”的形式引用指针变量（如 P->data=0;）。

（二）单链表的结构、建立、输出

由于单链表的每个结点都有一个数据域和一个指针域，所以，每个结点都可以定义成一个记录。比如，有如下一个单链表，如何定义这种数据结构呢？



下面给出建立并输出单链表的程序，大家可以把它改成过程用在以后的程序当中。

```
#include<iostream>
using namespace std;
struct Node
{
    int data;
    Node *next;
};
Node *head,*p,*r;        //r 指向链表的当前最后一个结点，可以称为尾指针
int x;
int main()
{
   cin>>x;
   head=new Node;        //申请头结点
   r=head;
   while(x!=-1)          //读入的数非-1
   {
      p=new Node;        //否则，申请一个新结点
      p->data=x;
      p->next=NULL;
      r->next=p;         //把新结点链接到前面的链表中，实际上 r 是 p 的直接前趋
      r=p;               //尾指针后移一个
      cin>>x;
    }
   p=head->next;         //头指针没有数据，只要从第一个结点开始就可以了}
   while(p->next!=NULL)
    {
      cout<<p->data<<" ";
      p=p->next;
    }
   cout<<p->data<<endl; //最后一个结点的数据单独输出，也可以改用 do-while 循环
   system("pause");
}
```

（三）单链表的操作

1. 查找“数据域满足一定条件的结点”

```
p=head->next;
while((p->data!=x)&&(p->next!=NULL))p=p->next; //找到第一个就结束
if(p->data==x)
    找到了处理;
else
    输出不存在;
```

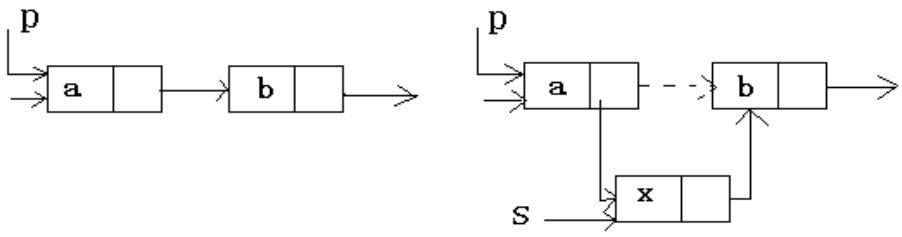
如果想找到所有满足条件的结点，则修改如下：

```
p=head->next;
while(p->next!=NULL)        //一个一个判断
{
    if(p->data==x)           //找到一个处理一个;
    p=p->next;
}
```

2. 取出单链表的第 i 个结点的数据域

```
void get(Node *head,int i)
{
    Node *p;int j;
    p=head->next;
    j=1;
    while((p!=NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    }
    if((p!=NULL)&&(j==i))
      cout<<p->data;
  else
      cout<<"i not exsit!";
}
```

3. 插入一个结点在单链表中去

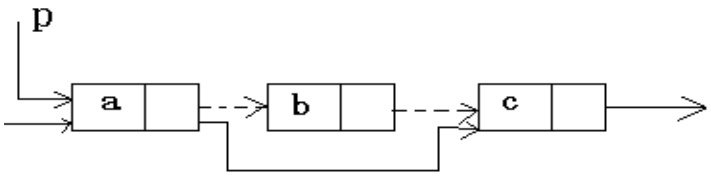


插入结点前和后的链表变化

```
void insert(Node *head,int i,int x) //插入 X 到第 i 个元素之前
```

```
{
    Node *p,*s;int j;
    p=head;
    j=0;
    while((p!=NULL)&&(j<i-1))          //寻找第 i-1 个结点,插在它的后面
    {
        p=p->next;
        j=j+1;
    }
    if(p==NULL)
      cout<<"no this position!";
    else
    {                                  //插入
        s=new Node;
        s->data=x;
        s->next=p->next;
        p->next=s;
    }
}
```

4. 删除单链表中的第 i 个结点（如下图的“b”结点）



删除一个结点前和后链表的变化

```
void delete(Node *head,int i)          //删除第 i 个元素
{
    Node *p,*s;int j;
    p=head;
    j=0;
    while((p->next!=NULL)&&(j<i-1))
    {
        p=p->next;
        j=j+1;
    }                                  //p 指向第 i-1 个结点
if (p->next==NULL)
  cout<<"no this position!";
else
{                              //删除 p 的后继结点，假设为 s
    s=p->next;
    p->next=p->next->next; //或 p->next=s->next
```

```
        free(s);
    }
}
```

5. 求单链表的实际长度

```
int len(Node *head)
{
    int n=0;
    p=head
    while(p!=NULL)
    {
        n=n+1;
        p=p->next
    }
    return n;
}
```

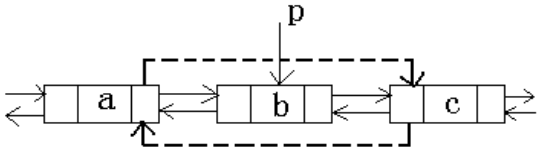
（四）双向链表

每个结点有两个指针域和若干数据域，其中一个指针域指向它的前趋结点，一个指向它的后继结点。它的优点是访问、插入、删除更方便，速度也快了。但“是以空间换时间”。

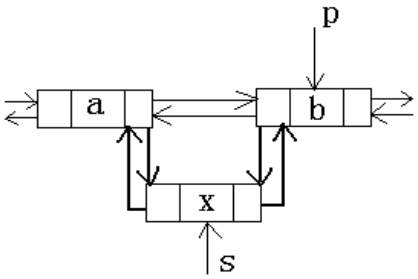
【数据结构的定义】

```
struct node
{
   int data;
   node *pre,*next;    //pre 指向前趋，next 指向后继
}
node *head,*p,*q,*r;
```

下面给出双向链表的插入和删除过程。



删除P结点前后的指针变化



在P结点之前插入S结点前后的指针变化

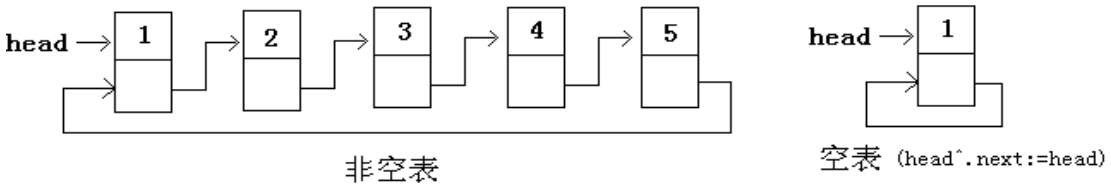
```
void insert(node *head,int i,int x) //在双向链表的第 i 个结点之前插入 X
{
    node *s,*p;
    int j;
    s=new node;
```

```
    s->data=x;
    p=head;
    j=0;
    while((p->next!=NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    }  //p 指向第 i 个结点
    if(p==NULL)
      cout<<"no this position!";
    else
    {                          //将结点 S 插入到结点 P 之前
        s->pre=p->pre;      //将 S 的前趋指向 P 的前趋
        p->pre=s;           //将 S 作为 P 的新前趋
        s->next=p;          //将 S 的后继指向 P
        p->pre->next=s;     //将 P 的本来前趋结点的后继指向 S
    }
}

void delete(node *head,int i)   //删除双向链表的第 i 个结点
{
    int j;
    node *p;
    P=head;
    j=0;
    while((p->next!=NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    }                                  //p 指向第 i 个结点
    if(p==NULL)
      cout<<"no this position!";
    else
    {                                  //将结点 P 删除
        p->pre->next=p->next;          //P 的前趋结点的后继赋值为 P 的后继
        p->next->pre=p->pre;           //P 的后继结点的前趋赋值为 P 的前趋
    }
}
```

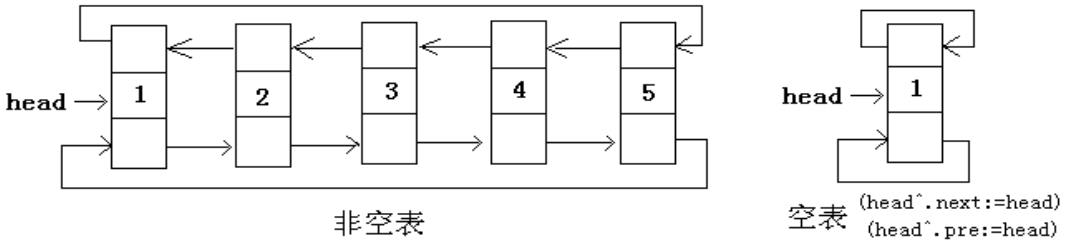
（五）循环链表

单向循环链表：最后一个结点的指针指向头结点。如下图：



单向循环链表

双向循环链表：最后一个结点的指针指向头结点，且头结点的前趋指向最后一个结点。如下图：



双向循环链表

（六）循环链表的应用举例

约瑟夫环问题

【问题描述】

有 M 个人，其编号分别为 1－M。这 M 个人按顺序排成一个圈（如图）。现在给定一个数 N，从第一个人开始依次报数，数到 N 的人出列，然后又从下一个人开始又从 1 开始依次报数，数到 N 的人又出列．．．如此循环，直到最后一个人出列为止。

【输入格式】

输入只有一行，包括 2 个整数 M，N。之间用一个空格分开(0 < n <= m <= 100)。

【输出格式】

输出只有一行，包括 M 个整数

【样列输入】

8 5

【样列输出】

5 2 8 7 1 4 6 3

【参考程序】

```
#include <iostream>
using namespace std;
struct node
{
       long d;
       node *next;
};
long n,m;
node *head,*p,*r;
int main()
```

```
{
    long i,j,k,l;
    cin>>n>>m;
    head=new node;
    head->d=1; head->next=NULL; r=head;
    for (i=2;i<=n;i++)
    {
        p=new node;
        p->d=i;
        p->next=NULL;
        r->next=p;
        r=p;
    }
    r->next=head; r=head;
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=m-2;j++) r=r->next;
        cout<<r->next->d<<" ";
        r->next=r->next->next;
        r=r->next;
    }
}
```

【上机练习】

1、利用指针，编写用于交换两个整型变量值的函数。

样例输入：5 6

样例输出：6 5

2、利用指针，编写主程序，将输入字符串反序输出。

样例输入：ABCDEFGHIJK

样例输出：KJIHGFEDCBA

3、编写一个用于在字符串中查找某字符的函数。

查找成功，函数返回该字符第一次出现的地址（指针）；查找失败，返回 NULL。

编写主函数测试该函数。在主函数中输入原字符串和要查找的字符。如果找到，输出字符在原字符串中的序号；如果找不到，输出”no”。

输入格式：

输入包括两行，第一行为原字符串，第二行为要查找的字符。

输出格式：

输出包括一行，找到输出字符在原字符串中的序号（从 1 开始），找不到输出”no”。

样例输入 1：

ABCDEFGHIJKLMN

D

样例输出 1：

4

样例输入 2：

ABCDEFG

S

样例输出 2：

no

4、约瑟夫问题（使用链表）【3.2 数据结构之指针和链表 1748】

约瑟夫问题：有 n 只猴子，按顺时针方向围成一圈选大王（编号从 1 到 n），从第 1 号开始报数，一直数到 m，数到 m 的猴子退出圈外，剩下的猴子再接着从 1 开始报数。就这样，直到圈内只剩下一只猴子时，这个猴子就是猴王，编程求输入 n，m 后，输出最后猴王的编号。

输入格式：

每行是用空格分开的两个整数，第一个是 n, 第二个是 m（ 0 < m,n <=300)。最后一行是：

0 0

输出格式：

对于每行输入数据（最后一行除外)，输出数据也是一行，即最后猴王的编号。

样例输入：

6 2

12 4

8 3

0 0

样例输出：

5

1

7

5、删除数组中的元素（链表）【3.2 数据结构之指针和链表 6378】

给定 N 个整数，将这些整数中与 M 相等的删除。

假定给出的整数序列为：1,3,3,0,-3,5,6,8,3,10,22,-1,3,5,11,20,100,3,9,3

应该将其放在一个链表中，链表长度为 20

要删除的数是 3，删除以后，链表中只剩 14 个元素：1 0 -3 5 6 8 10 22 -1 5 11 20 100 9

要求：必须使用链表，不允许使用数组,也不允许不删除元素直接输出

程序中必须有链表的相关操作：建立链表，删除元素，输出删除后链表中元素，释放链表。

输入格式：

输入包含 3 行：

第一行是一个整数 n(1 <= n <= 200000)，代表数组中元素的个数。

第二行包含 n 个整数，代表数组中的 n 个元素。每个整数之间用空格分隔；每个整数的取值在 32 位有符号整数范围以内。

第三行是一个整数 k，代表待删除元素的值（k 的取值也在 32 位有符号整数范围内)。

输出格式：

输出只有 1 行：

将数组内所有待删除元素删除以后，输出数组内的剩余元素的值，每个整数之间用空格分隔。

样例输入：

20

1 3 3 0 -3 5 6 8 3 10 22 -1 3 5 11 20 100 3 9 3

3

样例输出：

1 0 -3 5 6 8 10 22 -1 5 11 20 100 9

6、统计学生信息（使用动态链表完成）【3.2 数据结构之指针和链表 6379】

利用链表记录输入的学生信息（学号、姓名、性别、年龄、得分、地址）。其中,学号长度不超过 20, 姓名长度不超过 40, 性别长度为 1, 地址长度不超过 40

输入格式：

包括若干行，每一行都是一个学生的信息，如：

00630018 zhouyan m 20 10.0 28#460

输入的最后以"end"结束

输出格式：

将输入的内容倒序输出。每行一条记录，按照

学号 姓名 性别 年龄 得分 地址

的格式输出

样例输入：

00630018 zhouyan m 20 10 28#4600

0063001 zhouyn f 21 100 28#460000

0063008 zhoyan f 20 1000 28#460000

0063018 zhouan m 21 10000 28#4600000

00613018 zhuyan m 20 100 28#4600

00160018 zouyan f 21 100 28#4600

01030018 houyan m 20 10 28#4600

0630018 zuyan m 21 100 28#4600

10630018 zouan m 20 10 28#46000

end

样例输出：

10630018 zouan m 20 10 28#46000

0630018 zuyan m 21 100 28#4600

01030018 houyan m 20 10 28#4600

00160018 zouyan f 21 100 28#4600

00613018 zhuyan m 20 100 28#4600

0063018 zhouan m 21 10000 28#4600000

0063008 zhoyan f 20 1000 28#460000

0063001 zhouyn f 21 100 28#460000

00630018 zhouyan m 20 10 28#4600