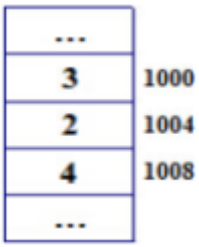
指针（在901数据结构读程序题中极为常见）

什么是指针

首先要解释下变量在内存中是如何存放的。

在计算机中，每个变量都会在内存中分配一个空间用来存放这个变量，而每种类型的变量所占的空间又是不一样的，例如 `int` 型的变量占用 4 个字节，而 `long long` 型的变量占用 8 个字节。我们可以把一个字节理解为一个“房间”，这样一个 `int` 型的变量（4 个字节）就需要占用 4 个连续的房间；而由于 `long long` 型的变量占用 8 个字节，因此一个 `long long` 型的变量就需要 8 个连续的房间来存放。那么，既然有房间，就肯定有“房间号”，且每个房间都会有一个房间号。对应在计算机中，每个字节（即房间）都会有一个地址（即房间号），这里的地址起的作用就是房间号的作用，即变量存放的位置，而计算机就是通过地址找到某个变量的。变量的地址一般指它占用的字节中第一个字节的地址。也就是说，**一个 int 型的变量的地址就是它占用的 4 个字节当中第一个字节的地址。**



根据上面的理解，我们可以认为，一个房间号“指向”一个房间，对应到计算机上就是一个地址“指向”一个变量，我们可以通过地址来找到变量。在 C 语言中，我们把地址用一个名词 “指针”来称呼。也就是说，**指针就是地址。**

那么，怎样获得变量的地址呢？很简单，就用前面讲到过的取地址运算符 `&`。只要在变量前面加上 `&`，就表示变量的地址。例如一个定义为 `int a` 的整型变量，`&a` 就表示它的地址。甚至我们可以把变量的地址输出来：

```
#include <stdio.h>
int main() {
    int a = 1;
    printf("%d, %d\n", &a, a);
    return 0;
}
```

输出结果：

```
2686748, 1
```

上面的输出结果在不同环境下会不同。

指针变量（很重要，好好理解）

指针变量用来存放指针（也就是地址），这个关系就跟 `int` 型变量用来存放 `int` 型常量相同。我们可以把地址当成是常量，然后专门定义了一种指针变量来存放它。但是指针变量的定义和普通变量有所区别，它在某种数据类型后加星号 `*` 来表示这是一个指针变量，例如下面这几个定义：

```
int* p;
double* p;
char* p;
```

注意，星号 `*` 的位置在数据类型之后或是变量名之前都是可以的，编译器不会对此进行区分。其中 C 程序员习惯于把星号放在变量名之前，也就是`int *p`的写法，而 C++程序员更习惯于把星号放在数据类型之后。本书采用把星号放在数据类型之后的写法。

另外，如果一次有好几个同种类型的指针变量要同时定义，星号只会结合于第一个变量名。也就是说，下面的定义中，只有 `p1` 是 `int*`型的，而`p2` 是 `int`型的：

```
int* p1, p2;
```

如果要让后面定义的变量也是指针变量，需要在后面的每个变量名之前都加上星号：

```
int* p1, *p2, *p3;
```

而为了美观起见，一般把第一个星号放在变量名 `p1` 前面：

```
int *p1, *p2, *p3;
```

正如刚才所说，指针变量存放的是地址，而`&`则是取地址运算符，因此给指针变量赋值的方式一般是把变量的地址取出来，然后赋给对应类型的指针变量：

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

上面的代码也可以写成：

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

而如果需要给多个指针变量初始化的话，方法也是一样：

```
int a, b;
int *p1 = &a, *p2 = &b;
```

需要注意的是，`int*`是指针变量的类型，而后面的 `p` 才是变量名，用来存储地址，因此地址`&a` 是赋值给 `p` 而不是`*p` 的。多个指针变量赋初值的时候由于写法上允许把星号放在所有变量名前面，因此容易搞晕，其实只要知道星号是类型的一部分就不会记错。

那么，对一个指针变量存放的地址，如何得到这个地址所指的元素呢？其实还是用星号`*`。假设我们定义了 `int* p = &a`，那么指针变量 `p` 就存放了 `a` 的地址。为了通过 `p` 来获得变量 `a`，我们把星号`*`视为一把开启房间的钥匙，将其加在 `p` 的前面，这样`*p` 就可以把房间打开，然后获得变量 `a` 的值。例如下面这个例子：

```c
#include <stdio.h>
int main() {
    int a;
    int* p = &a;
    a = 233;
    printf("%d\n", *p);
    return 0;
}
```

输出结果：

```
233
```

上面的代码中，我们首先定义了`int`型变量`a`，但是没有对其进行初始化。然后定义了指针变量`p`，并将`a`的地址赋值给`p`。这个时候，指针变量`p`存放了`a`的地址。之后`a`被赋值为`233`，也就是说，`a`所在地址的房间内的东西被改变了，但这并不影响它的地址。而在后面的输出中，我们使用星号`*`作为开启房间的钥匙，放在了`p`的前面，这样`*p`就获取到房间里的东西，即存储的数据。

由此我们也可以想到，既然`p`保存的是地址，`*p`是这个地址中存放的元素，那么如果我们直接对`*p`进行赋值，也可以起到改变那个保存的元素的功能，就像下面这个例子：

```c
#include <stdio.h>
int main() {
    int a;
    int* p = &a;
    *p = 233;
    printf("%d, %d\n", *p, a);
    return 0;
}
```

输出结果：

```
233, 233
```

上面的代码中，我们令指针变量`p`存放`a`的地址，然后直接对`*p`进行赋值，最后在输出时，`*p`和`a`都会输出那个值。

另外，指针变量也可以进行加减法，其中减法的结果就是两个地址偏移的距离。对一个`int*`型的指针变量`p`来说，`p + 1`是指`p`所指的`int`型变量的下一个`int`型变量地址。这个所谓的*下一个*是跨越了一整个`int`型、即4个字节，因此如果是`p + i`，则说明是跨越到当前`int`型变量之后的第`i`个`int`型变量。除此之外，指针变量支持自增和自减操作，因此`p++`等同于`p = p + 1`使用。指针变量的加减法一般用于数组中，关于这点，我们马上就会讲到。

对指针变量来说，我们把其存储的地址的类型称为基类型，例如定义为`int* p`的指针变量，`int`就是它的基类型。基类型必须和指针变量存储的地址类型相同，也就是说，上面定义的指针变量`p`不能够存放`double`型或`char`型数据的地址，而必须是`int`型数据的地址。

指针与数组

在之前对数组的讨论中我们提到，数组是由地址上连续的若干个相同类型的数据组合而成，对 `int` 型数组 `a` 来说，`a[0]`、`a[1]`、...、`a[n - 1]` 在地址上都是连续的。这样我们可以在元素前面加取地址运算符 `&` 来获取它的地址，例如 `a[0]` 的地址为 `&a[0]`，也即数组 `a` 的首地址为 `&a[0]`。

不过 C语言中规定，数组名称也作为数组的首地址使用，因此上面的例子中，有 `a == &a[0]` 成立。下面是一个例子：

```
#include <stdio.h>
int main() {
    int a[10] = {1};
    int* p = a;              //a本身就是一个地址
    printf("%d\n", *p);
    return 0;
}
```

输出结果：

```
1
```

代码中，`a` 作为数组 `a` 的首地址 `&a[0]` 而被赋值给指针变量 `p`，因此输出 `*p` 其实就是输出 `a[0]`。

前面还提到过，指针变量可以进行加减法，结合这个知识点，很容易可以推出 `a + i` 等同于 `&a[i]`，这是因为 `a + i` 就是指数组 `a` 的首地址偏移 `i` 个 `int` 型变量的位置。但是也应注意，`a + i` 其实只是地址，如果想要访问其中的元素 `a[i]` 的话，需要加上星号，使其变成 `*(a + i)` 后才和 `a[i]` 等价。由此，我们可以得到一种输入数组元素的新颖写法：`scanf("%d", a + i);`

我们知道，原先在 `a + i` 的位置填写的是 `&a[i]`，由于 `a + i` 和 `&a[i]` 等价，因此使用 `a + i` 来作为 `a[i]` 的地址是完全合适的。下面是读取一整个数组并输出的例子：

```
#include <stdio.h>
int main() {
    int a[10];
    for(int i = 0; i < 10; i++) {
        scanf("%d", a + i);
    }
    for(int i = 0; i < 10; i++) {
        printf("%d ", *(a + i));
    }
    return 0;
}
```

输入 10 个数组元素：

```
1 2 3 4 5 6 7 8 9 10
```

输出结果：

```
1 2 3 4 5 6 7 8 9 10
```

另外，由于指针变量可以使用自增操作，因此我们可以这样枚举数组中的元素：

```c
#include <stdio.h>
int main() {
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for(int* p = a; p < a + 10; p++) {    //地址增加了一个该类型的大小
        printf("%d ", *p);
    }
    return 0;
}
```

输出结果：

```
1 2 3 4 5 6 7 8 9 10
```

最后再提下指针的减法。我们来看下面这段代码：

```c
#include <stdio.h>
int main() {
    int a[10] = {1, 4, 9, 16, 25, 36, 49};
    int* p = a;
    int* q = &a[5];
    printf("q = %d\n", q);
    printf("p = %d\n", p);
    printf("q - p = %d\n", q - p);
    return 0;
}
```

输出结果：

```
q = 2686708
p = 2686688
q - p = 5                //指的是跨出了几个运算符
```

上面的代码中，`p` 和 `q` 的具体数值和运行环境有关，因此可能会得到跟上面不同的结果，但是两者之间一定是相差 `20` 的。但是我们会发现，**下面 `q - p` 的输出却是 `5` 而不是 `20`，这是怎么回事**？

前面说过，数组名 `a` 是直接作为数组 `a` 的首元素地址的，因此 `p` 和 `q` 其实分别是 `&a[0]` 与 `&a[5]`。这样 `q - p` 就是指两个地址之间的距离，而这个距离以 `int` 为单位。我们知道，`1` 个 `int` 占用 `4` 个字节，因此实际上两个指针之间的距离应该是 `20 / 4 = 5`，因此会输出 `5` 而不是 `20`。也许有些同学还是不太明白，那么我们可以说得更通俗一点：两个 `int` 型的指针相减，等价于在求两个指针之间相差了几个 `int`，由于 `&a[0]` 和 `&a[5]` 之间相差了 `5` 个 `int`，因此输出 `5`。这个解释对其他类型的指针同样适用。

使用指针变量作为函数参数

指针类型也可以作为函数参数的类型，这个时候视为把变量的地址传入函数。如果在函数中对这个地址中的元素进行改变的话，原先的数据就会确实地被改变。我们来看一个例子：

```c
#include <stdio.h>
void change(int* p) {
    *p = 233;
}
int main() {
    int a = 1;
    int* p = &a;
    change(p);
    printf("%d\n", a);
    return 0;
}
```

输出结果：

```
233
```

代码中我们把 `int*` 型的指针变量 `p` 赋值为 `a` 的地址，然后通过 `change` 函数把指针变量 `p` 作为参数传入。此时传入的其实是 `a` 的地址。在 `change` 函数中，使用 `*p` 修改地址中存放的数据，也就是改变了 `a` 本身。当最后输出 `a` 的时候，就已经是改变了的值。我们把这种传递方式称为地址传递。

我们来看一个经典例子：使用指针作为参数，交换两个数。

首先回顾如何交换两个数。一般来说，交换两个数需要借助中间变量，即先令中间变量 `temp` 存放其中一个数 `a`，然后再把另一个数 `b` 赋值给已被转移数据的 `a`，最后把存有 `a` 的中间变量 `temp` 赋值给 `b`。这样 `a` 和 `b` 就完成了交换。下面这段代码就实现了这个功能：

```c
#include <stdio.h>
int main() {
    int a = 1, b = 2;
    int temp = a;
    a = b;
    b = temp;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

输出结果：

```
a = 2, b = 1
```

那么，如果想要把交换功能写成函数的话，应该怎么做呢？首先需要指出，下面这种写法是做不到的：

```c
#include <stdio.h>
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
int main() {
    int a = 1, b = 2;
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

输出结果：

```
a = 1, b = 2
```

这是因为函数在接收参数的过程中是单向一次性的值传递。也就是说，我们在调用`swap(a, b)`的时候只是把`a`和`b`的值传进去了，这样相当于产生了一个副本，对这个副本的操作不会影响`main`函数中`a`、`b`的值。接下来我们介绍使用指针的方法。

我们知道，指针变量存放的是地址，那么使用指针变量作为参数的时候，传进来的也是地址。只有在获取地址的情况下对元素进行操作，才能真正地修改变量。为此，我们把上面的代码改写成下面这样：

```c
#include <stdio.h>
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int a = 1, b = 2;
    int *p1 = &a, *p2 = &b;
    swap(p1, p2);
    printf("a = %d, b = %d\n", *p1, *p2);
    return 0;
}
```

输出结果：

```
a = 2, b = 1
```

代码中，我们把`&a`（`a`的地址）和`&b`（`b`的地址）作为参数传入，使得`swap`函数中`int*`型指针变量`a`存放`&a`、指针变量`b`存放`&b`。这时候，`swap`函数中的`a`和`b`都是地址，而`*a`和`*b`就是地址中存放的数据，可以“看成”是`int`型变量，接下来就可以按前面正常的思路交换了。由于是直接对地址中存放的数据进行操作，因此交换操作会改变`main`函数中`a`与`b`的值，最终交换`a`与`b`。

下面指出两种初学者常犯的错误写法（学弟妹可以尝试改成错误的写法看看结果）：错误写法一：

```
void swap(int* a, int* b) {
    int* temp;
    *temp = *a;
    *a = *b;
    *b = *temp;
}
```

很多初学者会觉得，既然 `*temp`、`*a`、`*b` 都可以“看成”是 `int` 型变量，那完全就可以像普通变量那样进行交换操作。这个想法其实没有问题，出问题的地方在 `temp`。**我们在定义 `int*` 型的指针变量 `temp` 时，`temp` 没有被初始化，也就是说指针变量 `temp` 中存放的地址是随机的，那么如果该随机地址指向的是系统工作区间，那么就会出错（而且这样的概率特别大）。**

问题找到之后很容易想到解决办法：既然 `temp` 一开始没有被赋值而产生了随机的地址的话，那我们就可以给它赋初值，就不会有问题了。代码如下：

```
void swap(int* a, int* b) {
    int x;
    int* temp = &x;
    *temp = *a;
    *a = *b;
    *b = *temp;
}
```

错误写法二：

```
void swap(int* a, int* b) {
    int* temp = a; a = b;
    b = temp;
}
```

这种写法的思想在于直接把两个地址交换，认为地址交换之后元素就交换了。其实这种想法产生于很大的误区：`swap` 函数里交换完地址之后 `main` 函数里的 `a` 与 `b` 的地址被交换。前面说过，函数参数的传送方式是单向一次性的，`main` 函数传给 `swap` 函数的“地址”其实是一个“无符号整型”的数，其本身也跟普通变量一样只是“值传递”，`swap` 函数对地址本身进行修改并不能对 `main` 函数里的地址修改，能够使 `main` 函数里的数据发生变化的只能是 `swap` 函数中对地址指向的数据进行的修改。对地址本身进行修改其实跟之前对传入的普通变量进行交换的函数是一样的作用，都只是副本，没法对数据产生实质性的影响。总的来说就相当于我们把 `int*` 看作一个整体，传入的 a 和 b 都只是地址的副本**。

引用（引用在读程序题中即为重要）

引用的含义

引用是 C++中的一个强有力的语法，在编程时极为实用。我们知道，函数的参数是作为局部变量的，对局部变量的操作不会影响外部的变量，如果想要修改传入的参数的话只能用指针。那么，有没有办法可以不使用指针，也能达到修改传入参数的目的？一个很方便的方法是使用 C++中的“引用”。引用不产生副本，而是给原变量起了个别名。例如，假设我本名叫 `HDC`，某天大家给我起了个别名 `CG`，这样这两个名字说的都是同一个人（即这两个名字指向了同一个人）。引用就相当于给原来的变量又取了个别名，这样旧名字跟新名字其实都是指同一个东西，且对引用变量的操作就是对原变量的操作。

引用的使用方法很简单，只需要在函数的参数类型后面加个`&`就可以了（`&`加在`int`后面或者变量名前面都可以，考虑到引用是别名的意思，因此一般写在变量名前面），下面是一个例子（由于是 C++的语法，因此文件必须保存为`.cpp`后缀）。

```
#include <stdio.h>
void change(int &x) {
    x = 1;
}
int main() {
    int x = 10;
    change(x);
    printf("%d\n", x);
    return 0;
}
```

输出结果：

```
1
```

上面的代码中，我们在`change`函数的参数`int x`中加了`&`，在传入参数的时候对参数的修改就会对原变量进行修改。需要注意的是，不管是否使用引用，函数的参数名和实际传入的参数名可以不同。例如上面这个程序改成下面这样也是可以的：

```
#include <stdio.h>
void change(int &x) {
    x = 1;
}
int main() {
    int a = 10;
    change(a);
    printf("%d\n", a);
    return 0;
}
```

注意要把引用的`&`跟取地址运算符&区分开来，引用并不是取地址的意思。

指针的引用

在前面关于交换的错误写法二中，我们试图将传入的地址交换，以达到交换两个变量的效果，但是失败了，这是因为对指针变量本身的修改无法作用到原指针变量上。此处我们介绍了引用，就可以实现上面的效果了，实例如下：

```
#include <stdio.h>
void swap(int* &a, int* &b) {
    int* temp = a;
    a = b;
    b = temp;
}
int main() {
    int a = 1, b = 2;
    int *p1 = &a,
```

```
    *p2 = &b;
    swap(p1, p2);
    printf("a = %d, b = %d\n", *p1, *p2);
    return 0;
}
```

需要强调的是，由于引用是产生变量的别名，因此常量不可使用引用。于是上面的代码中不可以写成 `swap(&a, &b)`，而必须用指针变量 `p1` 和 `p2` 存放 `&a` 和 `&b`，然后把指针变量作为参数传入。

另外，如果学弟妹阅读代码时碰到了引用，却发现看晕了的时候，不妨先把引用去掉，看看其原意是什么，然后再加上引用，这样会容易理解很多。