Fall 2006 Handout 3

Common errors

The chief obstacle to correct diagnosis in painful conditions is the fact that the symptom is often felt at a distance from its source.

—*Textbook of Orthopaedic Medicine,* quoted by Philip Roth in *The Anatomy Lesson*

(1) The error message may have the wrong line number: look at the previous line(s).

(2) Make sure that the following pairs of characters are balanced:

```
{  }                    "  "
(  )                    '  '
[  ]                    <  >    only in #include directives
/*  */
```

(3) Does every comment end with a `*/`? Has a comment without a terminating `*/` swallowed a section of your program? Comments do not nest.

```
/* good comment */
/ * bad comment */
/* bad comment * /
```

(4) Did you accidentally type parentheses instead of `[`square brackets`]` around an array subscript?

(5) `if`, `for`, `#define`, etc., must all be lowercase.

(6) Did you confuse the letters and digits `l`, `1`, `I`, `o`, `0`? Did you use the wrong case (upper vs. lower)? Did you confuse `/` with `\`? Did you accidentally type `$` instead of `#` in a `#define` or `#include`? Did you accidentally type invisible control characters into your `.c` file?

(7) Put no white space in tokens such as `++ += == &&` etc.

(8) There must be parenthesis around the logical expression in an `if`, `while`, and `do-while`, and the expression in a `switch`.

(9) Did you forget the comma between function arguments? Is there a semicolon at the end of every statement? Did you accidentally write a semicolon immediately after the `}` at the end of a function? Did you accidentally write a semicolon after the `)` at the end of a `for` or `while` line:

```
for (i = 0; i < 10; ++i); {       /* wrong */
    blah blah blah;
}
```

(10) Did you accidentally say `if (a = b)` instead of `if (a == b)`?

(11) Did you try to `printf` a `long` with `%d` instead of `%ld`? Did you try to `scanf` a `double` with `%f` instead of `%lf`? Make sure that the `%` formats agree with the data types of the expressions you're outputting or inputting. Did you forget the `&`'s in a `scanf`?

(12) Read your program from a printout instead of the screen. Did you accidentally chop off the top or bottom of the program? *Can you get it to compile by removing or commenting out certain sections?* Show it to someone else. Take a break.

(13) Examine the **.i** file created by running only the C preprocessor, not the full compiler, with the uppercase **-E** option:

```
1$ gcc -E prog.c > prog.i
2$ more prog.i
```

(14) Does the C++ compiler give better error messages?

Subscripting is actually a binary operator: K&R p. 201, §A7.3.1; King pp. 140−141

a[i] is an expression, just like **i+j** and **i-j**:

$$\boxed{\boxed{\texttt{i}} + \boxed{\texttt{j}}}$$

$$\boxed{\boxed{\texttt{a}} \; [\; \boxed{\texttt{i}} \;]}$$

The value of the expression **a[i]** is the value of the **i**'th element of the array **a**. Of all the expressions we have seen so far, only an array can be the left operand of **[]**:

```
int a[10];
int i = 0;

a[i]                /* good */
i[a]                /* bad */
```

When evaluating the expression **c+a[b]**, the computer first delves into the array to get the value of the expression **a[b]**, and then performs the addition. In other words it executes the **[]** operator before the **+** operator, because the **[]** has higher precedence than the unary ***** operator in the table on K&R p. 53; King p. 595.

$$\boxed{\boxed{\texttt{c}} + \boxed{\boxed{\texttt{a}} \; [\; \boxed{\texttt{b}} \;]}}$$

Function calling is actually an operator: K&R p. 201, §A7.3.2; King pp. 161−162

f() is an expression, just like **i++** and **i--**:

$$\boxed{\boxed{\texttt{i}} \, \texttt{++}}$$

$$\boxed{\boxed{\texttt{f}} \, \texttt{()}}$$

f(x) is an expression, just like **i+j** and **a[i]**:

$$\boxed{\boxed{\texttt{i}} + \boxed{\texttt{j}}}$$

$$\boxed{\boxed{\texttt{a}}\ \texttt{[}\ \boxed{\texttt{i}}\ \texttt{]}}$$

$$\boxed{\boxed{\texttt{f}}\ \texttt{(}\ \boxed{\texttt{x}}\ \texttt{)}}$$

The value of the expression **`f(x)`** is the value computed by (or *returned by*) the function **`f`** when you give it **`x`** as an argument. Not every function returns a value; those that do not are said to return **`void`**. If **`f`** were a **`void`** function, you could not use the value of the expression **`f(x)`** as part of a larger expression. Of all the expressions we have seen so far, only a function can be the left operand of **`()`**.

```
#include <stdio.h>
#include <math.h>       /* needed for sqrt */

double x = 2.0;

sqrt(x)     /* good */
x(sqrt)     /* bad */
```

When evaluating the expression **`y+f(x)`**, the computer first calls the function **`f`** with the argument **`x`** to get the value of the expression **`f(x)`**, and then performs the addition. In other words it executes the **`()`** operator before the **`+`** operator, because the **`()`** has higher precedence than the binary **`+`** operator in the table on K&R p. 53; King p. 595.

$$\boxed{\boxed{\texttt{y}}\ \texttt{+}\ \boxed{\boxed{\texttt{f}}\ \texttt{(}\ \boxed{\texttt{x}}\ \texttt{)}}}$$

Each argument of a function is a subexpression, and it is impossible to predict the order in which these subexpressions will be evaluated. The following **`printf`** may print different values on different machines (K&R p. 53; King p. 595):

```
int i = 10;
printf("%d %d\n", ++i, i);
```

`11 11`	*Some machines evaluate the function arguments from left to right.*

`11 10`	*Other machines evaluate the function arguments from right to left.*

How to recognize the three kinds of parentheses in an expression

(1) A pair of parentheses that encloses the name of a data type is a cast (K&R p. 45); King pp. 127−128, listed on line 2 of the table on K&R p. 53; King p. 595:

```
(int)i
(unsigned long)i
```

(2) Otherwise, a pair of parentheses that has an expression in front of it is the function call operator **`()`**, listed on line 1 of the table on K&R p. 53; King p. 595:

(3) Otherwise, the pair of parentheses is the kind that overrides the precedence and associativity of operators. This kind is not listed in the table on K&R p. 53; King p. 595.

`f(x)`	`f` *is an expression.*
`a * (b + c)`	`a+` *is not an expression.*
`(*p)(x)`	`(*p)` *is an expression.*
`*p(x)`	`p` *is an expression.*

In the expression **`(*p)(x)`**, the first pair of parentheses is the overriding kind, and the second pair is the function call operator. Handout 2, p. 30, lines 14 and 21 each have all three kinds of parentheses.

Create a pointer variable: K&R pp. 93−95; King pp. 205−207

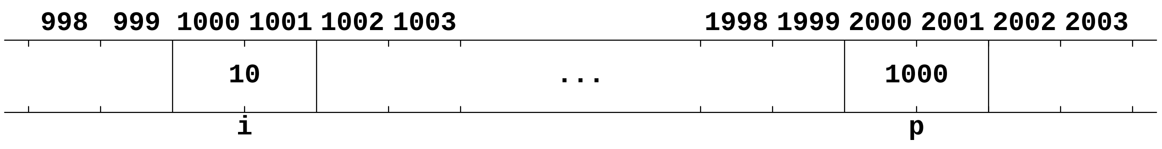
Every variable has two numbers: its *value* and its *address*. The value of a variable may change as the program runs, but the address never will. If a variable occupies two or more bytes of memory, these bytes will always be consecutive. The address of a variable is the address of the byte within it that has the lowest address.

In the following example the value of **`i`** is **`10`**. Assume that the address of **`i`** is **`1000`** and that **`sizeof i`** is **`2`**.

A *pointer* is a variable whose value is the address of another variable. For example, the value of **`p`** is the address of **`i`**. We say that **`p`** *is a pointer to* **`i`**, or that **`p`** *points to* **`i`**, when the value of **`p`** is the address of **`i`**.

Why does the **`*p`** in line 12 fetch an **`int`** from memory, as opposed to a **`double`** or an individual **`char`**? It's because of the declaration in line 6.

The following example assumes that **`sizeof p`** is equal to **`sizeof i`**, but this may not be true on all machines.



```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int i = 10;         /* Create a variable of type "int". */
 6     int *p = &i;        /* The value of p is now the address of i, namely 1000. */
 7
 8     printf("%d\n", i);  /* Print i in decimal. */
 9     printf("%p\n", &i); /* Print &i in your platform’s conventional base. */
10
11     printf("%p\n", p);
12     printf("%d\n", *p);
13 }
```

You can split line 5 into

```
14     int i;
15     i = 10;
```

—but why would you want to?

You can split line 6 into

```
16     int *p;
17     p = &i;
```

—but why would you want to?

```
10          line 8
1000        line 9
1000        line 11
10          line 12
```

You can’t **`printf("%d\n", *p)`** until you have first assigned a value to **`p`** and also assigned a value to the variable whose address you put in **`p`**. These two assignments can be made in either order as long as both are performed before the **`printf`**.

```
#include <stdio.h>

main()
{
    int i;
    int *p;

    printf("%d\n", *i);    /* won’t compile: i is an int, not a pointer */
    printf("%d\n", *p);    /* might blow up: p has not yet received a value */

    p = &i;                /* okay, even though i has not yet received a value */
    printf("%p\n", p);     /* okay, even though i has not yet received a value */

    i = 10;
    printf("%d\n", *p);    /* okay */
```

Change the value of a variable without mentioning its name

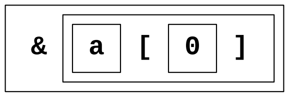
Here are two ways to add the values of the variables **`i`** and **`j`** and put the sum in **`k`**.

```
#include <stdio.h>

main()
{
    int i = 10;
    int j = 20;
    int k = i + j;

    printf("%d\n", k);
}
```

```
#include <stdio.h>

main()
{
    int i;
    int j;
    int k;

    int *p = &i;
    int *q = &j;
    int *r = &k;

    *p = 10;                  /* Put 10 into i. */
    *q = 20;                  /* Put 20 into j. */
    *r = *p + *q;             /* Put i + j into k. */
    printf("%d\n", *r);       /* Print k. */
27 }
```

Use a pointer to loop through an array: K&R pp. 97−100; King pp. 224−226

Here is another way to sum up the numbers in the array in Handout 2, p. 10. The address of the first array element is **`&a[0]`**.



The address of the second element is **`&a[1]`**. The address of the twelfth element is **`&a[11]`**. If there was a thirteenth array element, its address would be **`&a[12]`**; but there isn't. Therefore **`&a[12]`** is the address of the first byte after the end of the array (top of K&R p. 103; King p. 225).

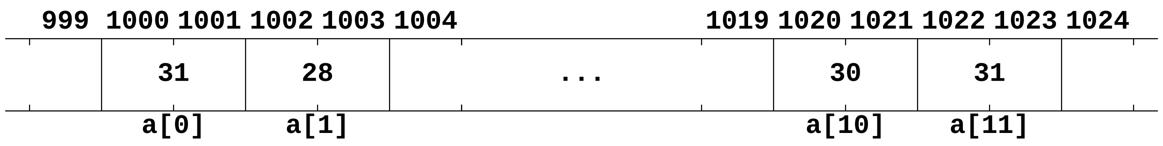
We could also have written **`p <= &a[11]`** instead of **`p < &a[12]`** in line 24 below, just as we could have written **`i <= 11`** instead of **`i < 12`** in Handout 2, p. 10, line 24. But in a program to loop through the twelve months of the year, it's clearer to write the magic number **`12`** explicitly.

On a machine where **`sizeof(int)`** is 2, the expression **`p=p+1`** actually adds 2 to the value of **`p`**. On a machine where **`sizeof(int)`** is 4, the expression **`p=p+1`** will add 4 to the value of **`p`**. In both cases, this leaves **`p`** pointing to the next **`int`** in memory.

In Handout 2, p. 10, line 24, you can change **`i = i + 1`** to **`++i`**. In line 24 below, you can change **`p = p + 1`** to **`++p`**.

In Handout 2, p. 10, line 25, you can change **`sum = sum + a[i]`** to **`sum += a[i]`**. In line 25 below, you can change **`sum = sum + *p`** to **`sum += *p`**.

The following diagram assumes that **`&a[0]`** is **`1000`** and that **`sizeof(int)`** is **`2`**.



—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/array2.c`

```
 1 /* Print the number of days in a non-leap year. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int a[12] = {
 7         31,     /* January */
 8         28,     /* February */
 9         31,     /* March */
10         30,     /* April */
11         31,     /* May */
12         30,     /* June */
13         31,     /* July */
14         31,     /* August */
15         30,     /* September */
16         31,     /* October */
17         30,     /* November */
18         31,     /* December */
19     };
```

```

    int *p;          /* point to each array element */
    int sum = 0;

    for (p = &a[0]; p < &a[12]; p = p + 1) {
        sum = sum + *p;
    }

    printf("%d\n", sum);
}
```

What not to use a pointer for

There is no need to create an array and a pointer just to iterate 10 times. Simply create a conventional **int** variable and have it count from 1 to 10.

```
/* Print the word "hello" 10 times. */
#include <stdio.h>

main()
{
    int a[10];
    int *p;

    for (p = &a[0]; p < &a[10]; ++p) {
        printf("hello\n");
    }
}
```

Leading * vs. trailing [0]

If **p** points to an **int**, the two expressions ***p** and **p[0]** have the same value—the value of the **int** that **p** points to. They are equally efficient in space and speed.

```
    int i = 10;
    int *p = &i;

    printf("%d\n", i);
    printf("%d\n", *p);
    printf("%d\n",  p[0]);
```

```
10
10
10
```

This is as far as we can go in the above example—there is only one **int** to point to (the variable **i**). But in the next example with a pointer, **p** will point to an **int** that has other **int**'s as its left and right neighbors.

Access adjacent int's in memory: K&R p. 99; King pp. 230−231

```
/* Given an array of 10 int's, this program prints a list of 8 int's.  Each of
the 8 int's is the average of 3 consecutive int's in the array.  For example,
the first int printed is the average of the first 3 int's in the array. */
#include <stdio.h>

```

```
 6 main()
 7 {
 8     int a[10] = {
 9         261, 256, 251, 276, 271, 266, 291, 286, 281, 306
10     };
11     int i;
12
13     for (i = 1; i <= 8; ++i) {
14         printf("%d\n", (a[i-1] + a[i] + a[i+1]) / 3);
15     }
16 }
```

```
256            a nice linear progression, once the static has been removed
261
266
271
276
281
286
291
```

In the following example, the values of the expressions **`p[1]`**, **`p[2]`**, **`p[3]`**, etc., are the values of the adjacent **`int`**'s to the right of **`p[0]`** (i.e., towards higher memory addresses), and **`p[-1]`**, **`p[-2]`**, **`p[-3]`**, etc., are the values to the left. Thus any region of memory can be accessed by a notation that makes the region look like an array of **`int`**'s extending in both directions. Simply put the address of the start of the region into **`p`** and then use **`p[0]`**, **`p[1]`**, **`p[2]`**, **`p[-1]`**, etc.

We write

```
p[-1] + p[0] + p[1]
```

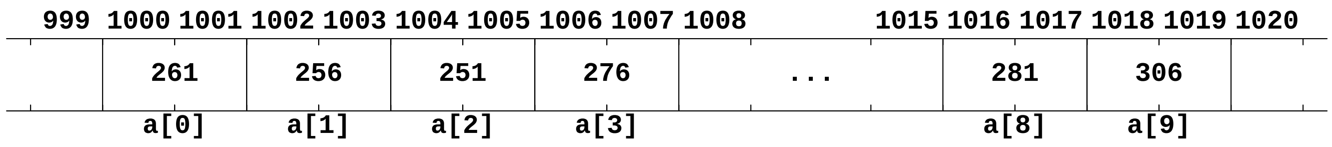
in line 14 below, rather than

```
p[-1] + *p + p[1]
```

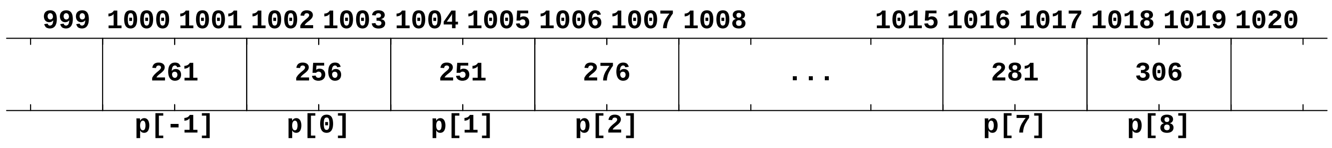
for stylistic consistency: we want **`p[-1]`**, **`p[0]`**, and **`p[1]`** to look like three elements of an array called **`p`**. Although there is no an array with this name, we create the illusion that one exists and that it is located exactly where we want it: with its zero element **`p[0]`** being the second of the three **`int`**'s we want to sum up. This illusory array is portable: we center it on a different **`int`** during each iteration.

```
 1 /* The same program, with a pointer p instead of an integer index i. */
 2
 3
 4 #include <stdio.h>
 5
 6 main()
 7 {
 8     int a[10] = {
 9         261, 256, 251, 276, 271, 266, 291, 286, 281, 306
10     };
11     int *p;
12
13     for (p = &a[1]; p <= &a[8]; ++p) {
14         printf("%d\n", (p[-1] + p[0] + p[1]) / 3);
15     }
16 }
```

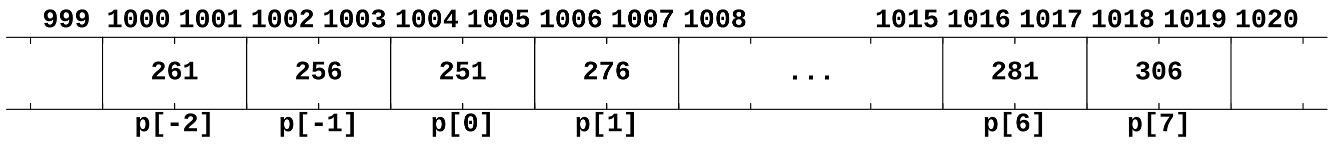
Here is the array **a**:



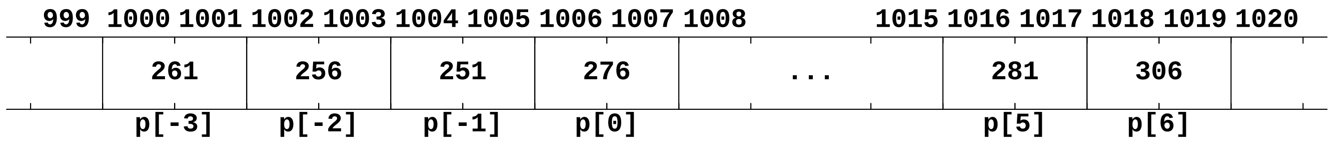
The first time we execute line 14, we can access this region of memory as if it were an array of **int**'s named **p** starting at address 1002:



The second time we execute line 14, we can access this region of memory as if it were an array of **int**'s named **p** starting at address 1004:



The third time we execute line 14, we can access this region of memory as if it were an array of **int**'s named **p** starting at address 1006:

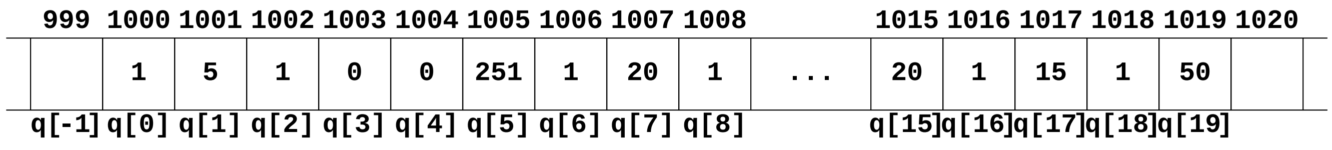


Access any region of memory as if it were an array of whatever data type you want

In binary, the number 261 is **00000001 00000101**, i.e., a byte containing 1 followed by a byte containing 5. If we said

```
char *q = (char *)&a[0]; /* Put the number 1000 into q. */
```

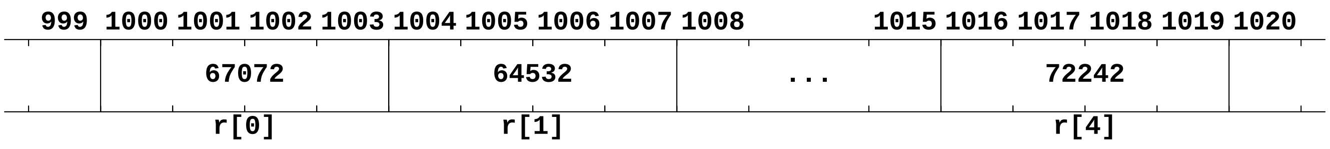
we could then access this region of memory as if it were an array of **char**'s named **q** starting at address 1000:



In binary, the number 261 is **00000001 00000101** and the number 256 is **00000001 00000000**. Together, they make up the four-byte number **00000001 00000101 00000001 00000000**, i.e., 67,072. If we said

```
long *r = (long *)&a[0]; /* Put the number 1000 into r. */
```

we could then access this region of memory as if it were an array of **long**'s named **r** starting at address 1000:



★ A notation that you should never use

There’s another way to write **`p[1]`**, although it’s less concise. If **`p`** is the address of an **`int`**, then the expression **`p+1`** is the address of the adjacent **`int`**. (We used this when we wrote **`p=p+1`** or **`++p`**.) Therefore **`*(p+1)`** is the value of the adjacent **`int`**. Therefore the expressions **`p[1]`** and **`*(p+1)`** have the same value, but you should write **`p[1]`** because it’s simpler.

▼ Homework 3.1: Simplify the following expressions

Change the unary **`*`** operator and the binary **`+`** operator to the **`[]`** operator. Remove the parentheses.

```
int *p;
int i;

*(p+1)                                  *(p+i)
*(p-1)                                  *(p-i)
*(p+10)                                 *(p+i-1)
*(p-10)                                 *(p+0)
```

Keep the **`*`**.

▲

An array name is a pointer

In most languages you get an error message if you omit the subscript from an array:

```
int a[10];
int i = 0;

a[i]    /* legal */
a       /* illegal in most languages, but legal in C */
```

In C, however, the name of an array by itself is a legal expression whose value is the address of the first element of the array. For example, the expressions **`a`** and **`&a[0]`** below have the same value.

You can apply the unary **`&`** operator to a non-array variable such as **`i`** or to an array element such as **`a[0]`**or **`a[i]`**. But **`&a`** is illegal: to get the address of the first element of the array, simply write **`a`**.

```
#include <stdio.h>

main()
{
    int a[10];
    int i = 10;

    printf("%p\n", &i);          /* Print the address of i. */
    printf("%p\n", &a[0]);       /* Print the address of a[0]. */
    printf("%p\n",  a);          /* Print the address of a[0]. */
}
```

A simpler notation: K&R p. 99; King pp. 230−231

The expression **`a`** is really a “pointer to **`int`**”: its value is the address of an **`int`** (namely, the first **`int`** in the array). Moreover, as with any other “pointer to **`int`**”, **`a+1`** is the address of the next **`int`**, and **`a+2`** is the address of the **`int`** after that. For example,

```
    for (p = &a[0]; p < &a[10]; ++p) {          /* before */
    for (p = a; p < a + 10; ++p) {              /* after */

    for (p = &a[10 - 1]; p >= &a[0]; --p) {     /* before */
    for (p = a + 10 - 1; p >= a; --p) {         /* after */
```

▼ **Homework 3.2: Rewrite a bubble sort using pointers**

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/bubble.c`

```
 1 /* Bubble sort an array of 10 ints into ascending order.  The for loop in lines
 2 29-38 will move the array elements part of the way into the correct order.
 3
 4 If some moves were made, it means that we should execute this for loop again to
 5 see if additional moves will be made.  In this case, flag is set to 1 to make
 6 the do-while loop execute the for loop again.
 7
 8 If no moves were made, it means that the elements are already in order.  In this
 9 case, flag remains 0 and the do-while loop terminates. */
10
11 #include <stdio.h>
12
13 main()
14 {
15     int a[10];
16     int i;      /* index into the array */
17     int flag;   /* set to 1 to ensure one more trip */
18     int temp;   /* temporary storage for exchanging values */
19
20     /* Initialize the array with the numbers to be sorted. */
21     printf("Type %d numbers.  Press RETURN after each one.\n", 10);
22     for (i = 0; i < 10; ++i) {
23         scanf("%d", &a[i]);
24     }
25
26     /* Bubble sort the array into ascending order. */
27     do {
28         flag = 0;
29         for (i = 0; i < 9; ++i) {
30             if (a[i] > a[i+1]) {
31                 temp = a[i];    /* swap a[i] and a[i+1] */
32                 a[i] = a[i+1];
33                 a[i+1] = temp;
34
35                 flag = 1;
36             }
37             printf("debug: i == %d\n", i);
38         }
39     } while (flag == 1);
40
41     /* Output the array. */
42     for (i = 0; i < 10; ++i) {
43         printf("%11d\n", a[i]);
44     }
45 }
```

Change the program to refer to the array elements by a pointer rather than by subscripting. Follow these nine steps.

(1) Download the above program and make sure it works.

(2) Remove the variable **`i`** and its declaration, and replace it with a "pointer to **`int`**" called **`p`**. The induction variable of the three **`for`** loops will now be **`p`** instead of **`i`**. You get no credit if the variable **`i`** appears anywhere, even in the comments.

(3) The starting and ending points of the three **`for`** loops will now be memory addresses instead of numbers. Change **`0`** to **`&a[0]`**; change **`10`** to **`&a[10]`**; and change **`9`** to **`&a[9]`**;

(4) Change **`&a[i]`** to **`p`** in the **`scanf`**: the variable **`p`** now holds the address at which **`scanf`** is to deposit each input number.

(5) Change **`a[i]`** to **`*p`** or to **`p[0]`** everywhere. **`*p`** and **`p[0]`** have the same value. But it's stylistically better to use **`p[0]`** in statements in which you also mention its neighbors **`p[1]`** or **`p[-1]`**, etc; use **`*p`** elsewhere.

(6) Change **`a[i+1]`** to **`p[1]`** everywhere.

(7) In the debugging statement, change **`i`** to **`p - &a[0]`**. **`p`** is the address of one of the **`int`**'s in the array, and **`&a[0]`** is the address of the first **`int`** in the array. When you subtract the addresses of two **`int`**'s, the result is automatically divided by the number of bytes in an **`int`**, yielding the number of **`int`**'s from the start of the array to the **`int`** pointed to by **`p`**. See K&R p. 103; King pp. 221−223.

This division is the counterpart of the automatic multiplication that takes place when you add a number to a pointer: **`p=p+1`**. The number **`1`** is multiplied by the number of bytes in an **`int`** before it is added to **`p`**. See K&R pp. 98−99; King pp. 222−223.

(8) Use the simpler notation shown above wherever possible. For any integer **`n`**, change **`&a[n]`** to **`a+n`**. When **`n`** is zero, change **`&a[0]`** to **`a`** instead of to **`a+0`**. You get no credit if you apply the **`[]`** operator and the **`&`** operator to the same expression (e.g., **`&a[0]`**). You get no credit if you use the ★ "notation you should never use" (e.g., **`*(a+n)`**) on p. 11.

(9) Also change **`bubble.c`** to use a **`#define`** so that the number **`10`** appears exactly once in the file. Change **`10`** to **`N`** (in the comments too) and change **`9`** to **`N - 1`**. You get no credit if the number **`9`** appears anywhere, even in the comments. You get no credit if the number **`10`** appears anywhere, even in the comments, except in exactly one **`#define`**.

(10) Make sure the comments are up-to-date. You get no credit if they mention the variable **`i`** or the numbers **`9`** or **`10`**.

Verify that the program still works after this surgery. The debugging statement will tell you how many times the middle **`for`** loop iterates. Make sure it still iterates nine times during each iteration of the **`do-while`** loop. Then remove the debugging statement before you make the printout to hand in.

▲

□