**Arrays**

Arrays provide a useful way to access a block of adajcent memory cells using

- a single name
- a numeric index

An array of 100 integers is declared as follows:

```
int nums[100];
```

This declaration

- reserves 4 x 100 *bytes* of continuous memory
- the first word of the array is accessed using index 0
  - `nums[0]  = x;`
- the last word in the array is accessed using index 99
  - `nums[99] = y;`

The real power of the array is the capability to use a *variable* to access individual cells.

```
ndx = 0;
while (ndx < 100)
{
   nums[ndx] = ndx;
   ndx = ndx + 1;
}
```

**When to use an array**

Beginning programmers sometimes encounter an irresistable urge to *use arrays where they are not needed. Please resist this urge!*

Reasons *to use* an *array*

- The algorithm being used requires repeated access to a whole collection of values (e.g. sorting).
- Economies of scale in input/output exceed the performance penalty of filling a large array.
- Character arrays are commonly used to store words or sentences.

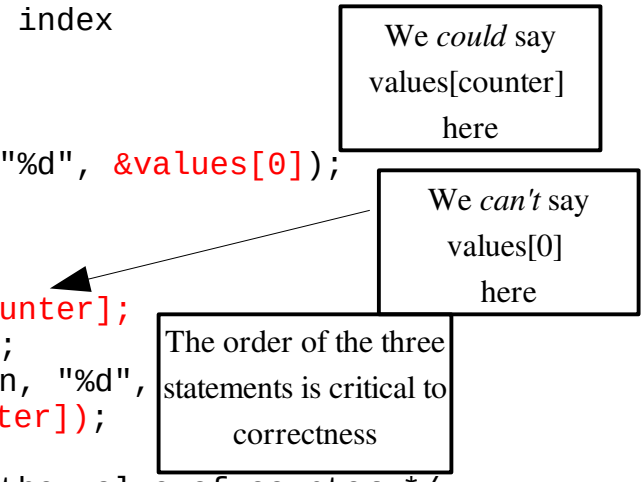
None of the algorithms for counting, summing, searching and recurrences require an array and no array should be used in problems of these types.

**A problem that does benefit from the use of an array.**

Read in a collection of values from the standard input and print those that are less than or equal to the average.

- Read values into an array computing the average
- Process the array printing values less than or equal to the average

```
/* p14.c */
#include <stdio.h>

int values[100];
int main()
{
   int counter;    // the number of values read in
   int howmany;    // howmany values were read
   int sum;        // sum of the values
   int average;    // average of the values
   int ndx;        // array index

   sum = 0;
   counter = 0;
   howmany = fscanf(stdin, "%d", &values[0]);

   while (howmany == 1)
   {
      sum = sum + values[counter];
      counter = counter + 1;
      howmany = fscanf(stdin, "%d",
                &values[counter]);
   }
/* When the read loop ends the value of counter */
/* is the number of elements in the array and  */
/* so (counter - 1) is the largest valid index */
   ndx = 0;
   average = sum / counter;
   while (ndx < counter)
   {
      if (values[ndx] <= average)
         fprintf(stdout, "%d \n",  values[ndx]);
      ndx = ndx + 1;
   }
}
```



Running the program produces the expected output:

```
==> p14
1 2 3 4 5 6 7 8 9 10
1
2
3
4
5
```

**A problem with no pleasant solution**

What if the number of input values is 110?

- The program as written will over write unallocated memory causing either a program fault or incorrect output!
- A better approach is something like

```
while (howmany == 1)
{
   sum = sum + values[counter];
   counter = counter + 1;
   if (counter == 100)
   {
       fprintf(stderr, “Too many values \n”);
       return(-1)
   }
   howmany = fscanf(stdin, "%d", &values[counter]);
}
```

But obviously what we would really like is a solution that works for *all possible* numbers of input values --- *which is an important reason to avoid using an array if you don't have to!!!*

**Using expressions as array indexes**

It is perfectly legal (and useful) to use expressions as indexes into an array.

For example we can assign values to *table[4], table[5],* and *table[6]* in the following ways.

```
int table[10];
int k;

k = 5;
table[k – 1] = 11;
table[k]     = 12;
table[k + 1] = 13;
```

Swapping adajacent values in a array is also a useful thing to do in sorting operations (and in the current codelab assignment);

```
int table[10];
int k;
if (table[k] > table[k + 1])
{
   table[k] = table[k + 1];
   table[k + 1] = table[k];
}
```

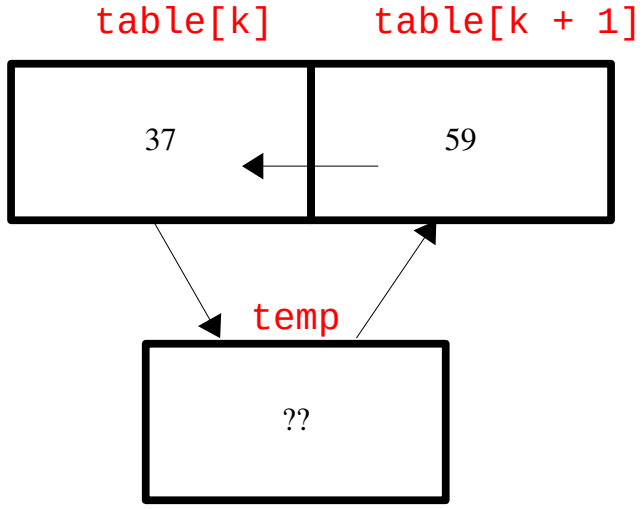
This approach *won't work!* By playing human computer we can see that the value of *table[k]*is destroyed and lost forever and both *table[k]* and *table[k + 1]* end up with the original value of *table[k + 1].*

*Exercise:* If we interchange the order of the assignments will it fix the problem. Will the results be different but still broken?

**A correct solution to the swapping problem**

To avoid destroying the contents of one of the elements we copy it to a temporary variable;

```
int table[10];
int k;
int temp;

if (table[k] > table[k + 1])
{
     temp = table[k];
     table[k] = table[k + 1];
     table[k + 1] = temp
}
```



**Designing a solution to the swapping problem**

When confronting problems such as this one it is very useful to

- start by solving a few examples by hand.

because

- Until you fully understand the problem and how to solve it manually you can't hope to be able to tell a computer how to do it!

- The examples you have solved by hand make useful test cases

From this example you might infer that all you need to do is move the first (or largest) value to the end of the array.

23 11 12 21 3 5

But this one shows you that all the values can move.

4 19 3 4 25 1 7

**Identifying and fixing errors**

Regardless of your approach either your initial design or your initial implementation or both are likely to be WRONG.

The ability to discover what when wrong is just as important as the ability to design and implement the solution.

There are three rational approaches and all three can be useful:

- Play human computer and manually *walk through* your code noting all changes of state. The primary disadvantage of this approach is that *if you misunderstand the semantics of the language, your walk through will yield incorrect results.*

- Instrument your program with diagnostic prints directed to the standard error.

```
int table[10];
int k;
int temp;

if (table[k] > table[k + 1])
{
     fprintf(stderr, “swapping elements %d and %d \n”,
                      k, k + 1);
     temp = table[k];
     table[k] = table[k + 1];
     table[k + 1] = temp
     fprintf(stderr, “new value of table[%d] is %d \n”,
                      k, table[k]);
     fprintf(stderr, “new value of table[%d] is %d \n”,
                      k + 1, table[k + 1]);
}
```

  The main disadvantage of this technique is the possiblity for *data overload.* So you want to start with short test cases.

- Use *gdb (my favorite)*

**Initializing scalars and arrays.**

Both scalar values and arrays may be initialized when they are declared:

```
int counter = 0;
int sum = 0;
```

Beware of

```
int counter, sum = 0;
```

Although C supports it, I recommend *against* declaring multiple variables in a single declaration.

When initializing an array,

- the initializers must be enclosed in {}
- you may provide fewer than the size of the array
- you may not provide more than the size of the array

```
int table[5] = {7, 9, 8};
```

**Using an array of counters**

Suppose an input file contains a collection of single digit non-negative integers and I want to count how many 0's , 1's, 2's  etc that there are in the collection.  Your initial instinct may be to create a maximally ugly collection of  *ifs*

```
if (val == 0)
     counters[0]= counters[0] + 1;
else if (val == 1)
     counters[1] = counters[1] + 1;
else if (val == 2)
```

The proper approach is surprisingly simple and should reside *forever* in your mental toolchest.

```
#include <stdio.h>

int counters[10];
int main()
{

   int howmany;    // howmany values were read
   int sum;        // sum of the values
   int average;    // average of the values
   int ndx;        // array index

   howmany = fscanf(stdin, "%d", &ndx);

   while (howmany == 1)
   {
      counters[ndx] = counters[ndx] + 1;
      howmany = fscanf(stdin, "%d", &ndx);
   }
```

Why did we not initialize the *counters[]* array.

The C language promises us that any variable declared *outside of* the body of all functions will be initialized to 0!

**Using a single dimension array to represent 2 - D data**

Suppose the integer variables *numrows* and *numcols* represent the number of rows and columns in the image and that they have been correctly set using information in the *.ppm* header.

Suppose the image is being constructed in the following array:

```
unsigned char image[300 * 200];
int numrows = 200;
int numcols = 300;
int ndx;
```

If we wish to access the pixel at location *(row, col)* within the image the value of *ndx* should be computed as:

```
ndx = row * numcols + col;
```

and the grayscale level of the pixel is:

```
image[ndx] = image[row * numcols + col];
```

For example, if the value of *numcols* is 10, then there are 10 pixels per image row. To reach the pixel whose (row, column) address is *(3, 5)* it is necessary to pass over three complete rows (row 0, row 1, and row 2) and 5 pixels in row three (pixels 0, 1, 2, 3, and 4).

Thus, the offset of the pixel at *(3, 5)* is *3 * 10 + 5* as shown above.

**Mapping the offset within an image to (*row, col)***

If we know the value of *ndx,* the offset within an image but wish to compute the value of *row* and *col,* we can divide both sides of the equation by *numcols* and see that

```
          _ row remainder col
numcols | ndx
```

In integer arithemetic in C this becomes

```
row = ndx / numcols;
col = ndx – row * numcols = ndx % numcols;
```

In C the operator % performs the *mod* function.

`a % b` is the remainder when a is divided by b.

For example `17 % 5 is 2.`

Thus one way to build a grayscale picture is:

```
ndx = 0;
while (ndx < numrows * numcols)
{
     row = ndx / numcols;
     col = ndx % numcols;
     image[ndx] = compute_graylevel(row, col);
     ndx += 1;
}
```

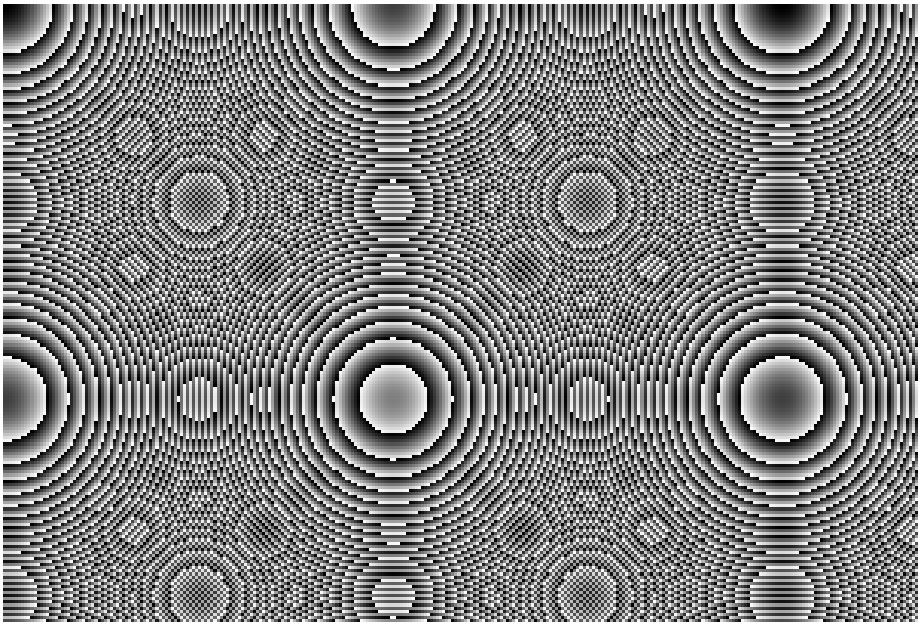
**Grayscale image creation**

This program shows how images with different geometric patterns may be created:

```
/* p13.c */

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

unsigned char image[300 * 200];

unsigned char pix_value(
int r,    /* this pixels row */
int c)    /* this pixels col */
{
    unsigned char pix;

    pix = 28 + (r + c) % 200;
    return(pix);
 }
```

More interesting patterns can be generated by using non-linear functions

```
unsigned char pix_value(
int r,
int c)
{
    unsigned char pix;

    pix = (r * r + c * c) % 255;

    return(pix);
}
```



**The main program**

```
int main(

int argc,
char **argv)
{
   int ndx = 0;
   int row;
   int col;


   while (ndx < (300 * 200))
   {
      row = ndx / 300;
      col = ndx % 300;

      image[ndx] = pix_value(row, col);

      ndx = ndx + 1;
   }
   fprintf(stdout, "P5\n");
   fprintf(stdout, "%d %d 255\n", 300, 200);
   fwrite(image, 1, 300 * 200, stdout);
   return(0);
}
```

**The mapping in color images**

In color images each pixel is represent by three bytes (R, G, B). Given a (row, col) location the offset or index of the *red* component is thus

```
ndx = 3 * row * numcols + 3 * col;
```

The index of the *green* component is at

```
ndx = 3 * row * numcols + 3 * col + 1;
```

and the blue one is at:

```
ndx = 3 * row * numcols + 3 * col + 2;
```

**The reverse mapping**

Suppose *ndx* contains the offset of the *red* component of some pixel. Then *ndx* is divisible by 3 and.

```
row = ndx / (3 * numcols);
col = ndx % (3 * numcols);
```

**Character arrays**

- The C language has no character string data type.
- Character strings such as words and sentences are stored in character *arrays*.
- A byte of binary 0's indicates the end of the string.
- The %s format code is used to read/write strings with fscanf/fprintf

```
#include <stdio.h>

char word[6] = {'H', 'e', 'l', 'l', 'o', 0};
int  nums[4]  = {-1, -2, 3};

int main()
{
   fprintf(stdout, "%s\n", word);
}
```

```
class/101/examples ==> p13
Hello
class/101/examples ==>
```

Warning, leaving off the trailing 0 is fatal.

**A useful shorthand:**

The notation above is clearly tedious... so a handy substitute is provided.
Strings may be implicitly created using the double quotes operator (as we have seen in format strings).

```
char word[] = “Hello”;
```

**Reading individual characters with *fscanf()***

The %c format code may be used to read or write integer values without performing the code conversion done by *%d.*

One might ask: How does the %c code treat whitespace characters such as space, tab, return, and newline?

The easiest and most reliable way to find out is with a simple program:

```
/* p14.c */

#include <stdio.h>

int main()
{
   char c;
   int  howmany;

   howmany = fscanf(stdin, "%c", &c);
   while (howmany == 1)
   {
      fprintf(stdout, "%02x %c \n", c, c);
      howmany = fscanf(stdin, "%c", &c);
   }
   return(0);
}
```

We create an input file consisting of a-space-b-tab-c-return-d-newline-e-newline

```
hexdump < p14.in

      0 - 61 20 62 09 63 0d 64 0a 65 0a
          a     b     c     d     e
```

Running the program gives us output that looks a little strange: Why did we get *d* instead of 0d. Where did the blank lines come from.

The answer is that when we print a control character using the %c code *it performs its control function.* So when the *newline* following the *letter* d was printed we get a new line injected after the 0a and then another newline from the format string itself.

```
acad/cs101/examples/notes ==> p14 < p14.in
61 a
20
62 b
09
63 c
 d
64 d
0a

65 e
0a
```

We can fix this by simply not printing the control characters.

```
/* p15.c */

#include <stdio.h>

int main()
{
   char c;
   int  howmany;

   howmany = fscanf(stdin, "%c", &c);
   while (howmany == 1)
   {
      if (c >= ' ')
         fprintf(stdout, "%02x %c \n", c, c);
      else
         fprintf(stdout, "%02x    \n", c);

      howmany = fscanf(stdin, "%c", &c);
   }
   return(0);
}
```

```
acad/cs101/examples/notes ==> p15 < p14.in
61 a
20
62 b
09
63 c
0d
64 d
0a
65 e
0a
```

**Reading with the %s format.**

The whitespace characters are treated differently when using the %s format. Each time a whitespace character appears it is interpreted as a delimiter ending the current string.

- The string is read into the location specifed
- *fscanf()* will automagically append the required byte of 0 used to indicate the end of memory resident strings
- *fscanf()* assumes that the the program has provided sufficient space to hold the entire string. If this is not true, *memory will be overwritten and the error may or may not be detected by the operating system.*

```
/* p16.c */

#include <stdio.h>

int main()
{
   char c[100];
   int  howmany;

   howmany = fscanf(stdin, "%s", &c[0]);
   while (howmany == 1)
   {
      fprintf(stdout, "%s \n", &c[0]);
      howmany = fscanf(stdin, "%s", &c[0]);
   }
   return(0);
}

acad/cs101/examples/notes ==> p16 < p14.in
a
b
c
d
e
```

**Multicharacter strings**

We can see that this behavior persists even when multicharacter strings are used:

```
 0 - 61 62 63 20 64 65 66 09 67 68 69 0d 6a 6b 6c 6d
      a  b  c     d  e  f     g  h  i     j  k  l  m
10 - 6e 6f 0a 70 0a
      n  o     p
```

```
acad/cs101/examples/notes ==> p16 < p16.in
abc
def
ghi
jklmno
p
```

We can also infer that *fscanf()* will return the number of *complete items read.* This will always be 1 until end of file

**Nested loops**

We have seen many examples of *if* statements embedded in *while* loops.

```
while (howmany == 1)
{
   if (c >= ' ')
      fprintf(stdout, "%02x %c \n", c, c);
   else
      fprintf(stdout, "%02x    \n", c);

   howmany = fscanf(stdin, "%c", &c);
}
```

It is also perfectly legal to embed *while* statements within *if* statements and even *while* statements within other *while* statements. The latter case is referred to as a nested loop.

- There are cases in which nested loops *provide a convenience*
- There are other cases in which nested loops *are the only way to solve a problem*

The ability to identify both situations is an important skill to have in programming.

**Nested loops as a convenience**

In constructing a *.ppm* image of a flag it may be convenient to use a nested loop, but as we have seen it is not necessary.

```
unsigned char red, green, blue;
int row, col;
int width, height;

row = 0;
while (row < height)
{
   col = 0;
   while (col < width)
   {
       red   = make_red_pix(row, col);
       green = make_green_pix(row, col);
       blue  = make_blue_pix(row, col);

       image[3 * row * width + 3 * col]     = red;
       image[3 * row * width + 3 * col + 1] = green;
       image[3 * row * width + 3 * col + 2] = blue;

       col = col + 1;
    }
    row = row + 1;
}
```

**Nested loops as a necessity**

Write a program that reads in a C string using the %s format and then reads individual characters using the %c format code until end-of-file is encountered. It should print out the number of the individual characters that appear in the string.

Sample input:
abcdefghijklmnop aqxqbcztud
Sample output:
4

Design approach

What data items do we need:

- a character array to hold the initial string
- a character variable to hold the individual character presently under scrutiny
- a integer index used to traverse the character array
- a flag used to remember if the current value has been seen yet
- a counter used to count up the number of values in the initial string

Given these data items, walk through the execution of the program you haven't built yet building in your mind a picture of how it must necessarily work. Make notes along the way in some form of recipe for what it is you want to do.

**The complete program**

```
#include <stdio.h>
char string[100];

int main()
{
   char value;
   int  howmany;
   int  index;
   int  counter = 0;
   int  found;

   fscanf(stdin, "%s", &string[0]);
   howmany = fscanf(stdin, "%c", &value);

   while (howmany == 1)
   {
      index = 0;
      found = 0;
      while (string[index] != 0)
      {
         if (value == string[index])
            found = 1;
         index = index + 1;
      }
      if (found)
         counter = counter + 1;
      howmany = fscanf(stdin, "%c", &value);
   }
   fprintf(stdout, "%d\n", counter);
}

p18
abcdefghij aabqqqce
5
```

**A more challenging example..**

In this example we do the *insertion* sort. Integer values a read one at a time in an outer loop and space is made to insert them in the correct spot in the inner loop by sliding existing values that are larger one slot to the right. The basic idea is quite simple and there are a number of workable strategies.

Nevertheless there are many ways to screw it up if one simply sits down and tries to *wing it* without designing in, as my implementation(s), p19, p19a, p19b, p19c show. In p19d I finally get it right (I think).

Attempt 1

```
/* p19.c */

#include <stdio.h>

int tab[100];

int main()
{
   int ndx;
   int counter;
   int howmany;
   int val;

   counter = 0;
   howmany = fscanf(stdin, "%d", &val);

   while (howmany > 0)
   {
      ndx = counter - 1;
      while ((ndx > 0) && (tab[ndx] > val))
      {
         tab[ndx + 1] = tab[ndx];
         fprintf(stderr, "tab[%d] now has %d \n", ndx + 1,
                                 tab[ndx + 1]);
      }
      tab[ndx - 1] = val;
      fprintf(stderr, "inserted %d in tab[%d] \n", val, ndx - 1);
   }
}
```

Input

5 4 3

Output:

```
inserted 5 in tab[-2]
inserted 5 in tab[-2]
inserted 5 in tab[-2]
inserted 5 in tab[-2]
inserted 5 in tab[-2]
and on and on and on........
```

Attempt 2: Fix the infinite loop.

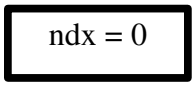
```
/* p19.c */

#include <stdio.h>

int tab[100];

int main()
{
   int ndx;
   int counter;
   int howmany;
   int val;

   counter = 0;
   howmany = fscanf(stdin, "%d", &val);

   while (howmany > 0)
   {
      ndx = counter - 1;
      while ((ndx > 0) && (tab[ndx] > val))
      {
         tab[ndx + 1] = tab[ndx];
         fprintf(stderr, "tab[%d] now has %d \n", ndx + 1,
                         tab[ndx + 1]);

      }
      tab[ndx - 1] = val;
      fprintf(stderr, "inserted %d in tab[%d] \n", val, ndx - 1);
      howmany = fscanf(stdin, "%d", &val);
   }
```

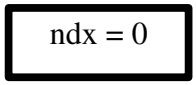
ndx = -1

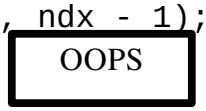
ndx-1 = -2

```
acad/cs101/examples/notes ==> p19a
5 4 3

inserted 5 in tab[-2] <---- trying to insert in slot -2 is bad.
inserted 4 in tab[-2]
```

```
/* p19.c */

#include <stdio.h>

int tab[100];

int main()
{
   int ndx;
   int counter;
   int howmany;
   int val;

   counter = 0;
   howmany = fscanf(stdin, "%d", &val);

   while (howmany > 0)
   {
      ndx = counter;
      while ((ndx > 0) && (tab[ndx] > val))
      {
         tab[ndx + 1] = tab[ndx];
         fprintf(stderr, "tab[%d] now has %d \n", ndx + 1,
                                                  tab[ndx + 1]);
         ndx = ndx - 1;
      }
      tab[ndx] = val;
      fprintf(stderr, "inserted %d in tab[%d] \n", val, ndx - 1);
      howmany = fscanf(stdin, "%d", &val);
   }
}
```

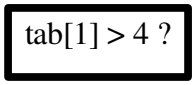


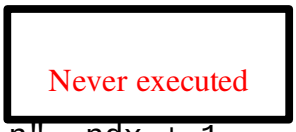




```
acad/cs101/examples/notes ==> p19b
5 4 3 2
inserted 5 in tab[-1]  Still looks like trouble
inserted 4 in tab[-1]
inserted 3 in tab[-1]
inserted 2 in tab[-1]
```

```
/* p19c.c */

#include <stdio.h>

int tab[100];

int main()
{
   int ndx;
   int counter;
   int howmany;
   int val;

   counter = 0;
   howmany = fscanf(stdin, "%d", &val);

   while (howmany > 0)
   {
      ndx = counter;
      while ((ndx > 0) && (tab[ndx] > val))
      {
         tab[ndx + 1] = tab[ndx];
         fprintf(stderr, "tab[%d] now has %d \n", ndx + 1,
                                    tab[ndx + 1]);
         ndx = ndx - 1;
      }
      tab[ndx] = val;
      counter = counter + 1;
      fprintf(stderr, "inserted %d in tab[%d] \n", val, ndx);
      howmany = fscanf(stdin, "%d", &val);
   }
}
```




```
acad/cs101/examples/notes ==> p19c
5 4 3 2 1
inserted 5 in tab[0]
inserted 4 in tab[1]
inserted 3 in tab[2]
inserted 2 in tab[3]
inserted 1 in tab[4]
```

```
/* p19.c */

#include <stdio.h>

int tab[100];

int main()
{
   int ndx;
   int counter;
   int howmany;
   int val;

   counter = 0;
   howmany = fscanf(stdin, "%d", &val);

   while (howmany > 0)
   {
      ndx = counter;
      while ((ndx > 0) && (tab[ndx - 1] > val))
      {
         tab[ndx] = tab[ndx - 1];
         fprintf(stderr, "tab[%d] now has %d \n", ndx, tab[ndx]);
         ndx = ndx - 1;
      }
      tab[ndx] = val;
      counter = counter + 1;
      fprintf(stderr, "inserted %d in tab[%d] \n", val, ndx);
      howmany = fscanf(stdin, "%d", &val);
   }
}
5 4 3 2 1
inserted 5 in tab[0]
tab[1] now has 5
inserted 4 in tab[0]
tab[2] now has 5
tab[1] now has 4
inserted 3 in tab[0]
tab[3] now has 5
tab[2] now has 4
tab[1] now has 3
inserted 2 in tab[0]
tab[4] now has 5
tab[3] now has 4
tab[2] now has 3
tab[1] now has 2
inserted 1 in tab[0]
```

So maybe we don't even need the *ndx > 0* test???

```
/* p19e.c */

#include <stdio.h>

int tab[100];

int main()
{
   int ndx;
   int counter;
   int howmany;
   int val;

   counter = 0;
   howmany = fscanf(stdin, "%d", &val);

   while (howmany > 0)
   {
      ndx = counter;
      while (tab[ndx - 1] > val)
      {
         tab[ndx] = tab[ndx - 1];
         fprintf(stderr, "tab[%d] now has %d \n", ndx, tab[ndx]);
         ndx = ndx - 1;
      }
      tab[ndx] = val;
      counter = counter + 1;
      fprintf(stderr, "inserted %d in tab[%d] \n", val, ndx);
      howmany = fscanf(stdin, "%d", &val);
   }
}
```

```
acad/cs101/examples/notes ==> p19e
1 2 3 4 5
inserted 1 in tab[0]
inserted 2 in tab[1]
inserted 3 in tab[2]
inserted 4 in tab[3]
inserted 5 in tab[4]

acad/cs101/examples/notes ==> p19e
5 4 3 2 1
inserted 5 in tab[0]
tab[1] now has 5
inserted 4 in tab[0]
tab[2] now has 5
tab[1] now has 4
inserted 3 in tab[0]
tab[3] now has 5
tab[2] now has 4
tab[1] now has 3
inserted 2 in tab[0]
tab[4] now has 5
tab[3] now has 4
tab[2] now has 3
tab[1] now has 2
inserted 1 in tab[0]

acad/cs101/examples/notes ==> p19e
1 1 1 1 1
inserted 1 in tab[0]
inserted 1 in tab[1]
inserted 1 in tab[2]
inserted 1 in tab[3]
inserted 1 in tab[4]
```

Note that the output is correct for all three tests but the *program is broken.*

Moral: *Depending upon your compiler and system there may be errors that are undetectable with any tests but may cause your program to instantly fail on another compiler or system!!!*

**A better approach to design and implementation:**

```
int tab[100];

main()
{
   int howmany;
   int val;           /* value to be inserted          */
   int count = 0;     /* number of values in the array */

   howmany = fscanf(stdin, “%d”, &val);
   while (howmany == 1)
   {

    /* insert val in proper slot in tab here */

       fprintf(stderr, “Current val is %d \n”, val);
       howmany = fscanf(stdin, “%d”, &val);
   }
}
```

**How to insert into the array:**

Assumptions

Name of the array is *tab*

Number of values presently is in *count*

Values already in the array are sorted in increasing order

*The basic step*

Compare last element in the array with *val*

If *val* is >=, add it to the end of the array and increment *count*

Otherwise move the current last element one position to the right.

*The iteration*

The variable *ndx* will point to the current "hole" in the array.

Set *ndx* to *count*

*While more stuff to do*

*tab[ndx] = tab[ndx -1]; // file hole with value to its left.*

*ndx = ndx – 1; // move the hole to the left*

*Store val in the hole (tab[ndx])*

*How to terminate the iteration*

Can't enter the loop with *ndx = 0,* because that would force a reference to *tab[-1];*

Want to exit loop when *val >= the array slot to the left of the hol*

Putting it all together:

```
ndx = counter;
while ((ndx > 0) && (tab[ndx - 1] > val))
{
   tab[ndx] = tab[ndx - 1];
   fprintf(stderr, "tab[%d] now has %d \n", ndx, tab[ndx]);
   ndx = ndx - 1;
}
tab[ndx] = val;
counter = counter + 1;
```