Use of scanf

We have now discussed how to print out formatted information to the screen, but this isn't nearly as useful unless we can read in information from the user. (This is one way we can make a program more useful!)

When you read in information from the keyboard, you MUST read it into a variable. In particular, you must read it into the address of the variable. (This is where the variable is stored in memory.)

In order to do this, you must specify the type of information being read in, and the name of the variable you want the information read into. For example, the following line would read in an integer into the variable number (assuming it is already declared):

```
scanf("%d", &number);
```

The & in front of number means "address of." This simply says to store whatever the user enters into the memory address where number is stored. *Leaving out the & will cause your program to work incorrectly!*

You may also read in multiple pieces of information at once. The following line reads in the month and day of birth into the integer variables month and day, (assuming they are already declared.)

```
scanf("%d%d", &month, &day);
```

When the user enters this information, they must separate what they enter with a space.

Let's use the added feature of being able to read in information from the user to edit our second program:

```
/* Arup Guha
   My Second C Program, edited
   9/2/03
   Computes the number of feet user ran. */

#include <stdio.h>

#define YARDS_IN_MILE  1760
#define FEET_IN_YARD   3

int main(void) {

  int feet_in_mile, num_miles;
  feet_in_mile = YARDS_IN_MILE*FEET_IN_YARD;
  printf("How many miles did you run?\n");
  scanf("%d", &num_miles);
  printf("You ran %d feet.\n",
                feet_in_mile*num_miles);

  return 0;
}
```

This program runs and works as before computing the variable feet_in_mile. The picture after this second line of code looks like:

feet_in_mile num_miles

5280	

Thus, num_miles is currently uninitialized.

Now, the user will be prompted with a message to enter the number of miles they ran.

The scanf statement will WAIT until the user enters some information and hits enter. After this is done, the value entered by the user will be stored in the variable num_miles.

Note: If the user doesn't enter the proper type of information, then the behavior of the scanf may not be as intended. A good exercise would be to try running a program that is expecting an integer input and actually enter a floating point number or a string and see what happens.

Let's say that the user entered 3, now our picture looks like:

```
feet_in_mile    num_miles
```

5280	**3**

In the final printf statement, an integer is printed out. Note that even though we don't have a variable to store the total number of feet the user ran, we can STILL display that information to the screen!

In particular, anytime we print a quantity, we simply have to place the proper arithmetic expression that evaluates to that quantity in the printf list in the appropriate place.

The final output for this run of the program will be:

```
You ran 15840 feet.
```

Programming Style

Programming style refers to how you lay out your code. Although we haven't seen too many C constructs, it is instructive to list a few of the common elements of programming style:

1) Include ample white space in your code so it is easy to read.

2) Indent all code in between corresponding curly braces {}

3) Use a header comment that specifies the author's name, the date, and a brief description of the program.

4) Comment each major block of code with a quick description of the function of the block of code.

5) Give your variables meaningful names so that it is easy to determine what information they are intended to store.

6) Use constants (#define) when appropriate and have them as all capital letters.

7) Be consistent with your style.

Common Programming Errors

1) Using the statements we have discussed so far, probably the most common error is:

```
Unterminated string or character constant
```

This means that you have forgotten a close quote to match an open quote.

2) Another very common error for a beginning programmer is forgetting the semicolon at the end of a statement. This seemingly easy error produces a surprisingly varied set of compiler error messages. The reason for this is that the compiler thinks that the current statement is continuing.

3) The proper character code for a double is %lf. But, printf will work for a double with the character code %f. This is tricky because scanf will not always work properly for reading in a double if the character code %f is used instead of %lf.

4) Forgetting the & when reading in a value into a variable. The compiler doesn't catch this error, and instead a run time error is often produced.

Types of Errors

1) Compile-Time Error: This is when your program doesn't use the proper syntax and does not compile.

2) Run-Time Error: Your program compiles, but when it runs, in certain cases it terminates abruptly. An example of this would be a program that attempted to divide by zero while running.

3) Logical Error: Your program compiles and executes without abnormal termination, but does not always produce the desired results.

Notes about White Space and Compilation

You may have noticed that the compiler sometime ignores white space and that it is important other times. In order to understand when white space makes a difference in the meaning of a program, we have to understand how the compiler breaks a program into tokens. A token is a single element in a program. Here are examples of tokens:

variable names
keywords
curly braces
string literal
operators
&

Each of these are naturally separated out when read in. White space is needed between tokens when no separator (like a comma), is between them.

For example, when defining variables, it is okay to omit spaces:

```
int num,val;
```

The comma is a delimiter that signifies the end of a token. This means that a comma can not be part of a variable name.

Since the & is a token on its own, we can choose to separate it with white space or not. Both of the following are valid:

```
scanf("%d",&num);
scanf("%d" , & num);
```

Since operators are tokens, no white space is necessary when other tokens are delimited by these. Thus, both of the following are valid:

```
val=num+1;
val = num + 1;
```

Finally, as you might imagine, you can not insert white space within a variable name. Thus, the following would not be valid:

```
v a l = num + 1;
```

Rules for Identifiers (variable names)

1) Must be comprised of letters, digits and/or underscores.

2) A letter or underscore must be the first character.

3) In most implementations, variable names ARE case sensitive.

4) Must not be a keyword. A keyword is a word reserved in the C language with special meaning. Here are some examples of key words: main, char, int, float, return, and switch.

5) Although this isn't a rule, good programming practice is to choose variable names that specify the function of the variable.

Use of Constants

Constants (using #define) should be used instead of typing out the literal value for a couple reasons:

1) A symbolic name (such as WATER_DENSITY) has more meaning than its value (which in the case above is 1.)

2) If for any reason you need to change the value of the constant, you only need to change it in one place.

So why not just use a variable?

1) To prevent you from changing the value of it during the execution of the program.

2) To distinguish between what values may change in a program and which ones do not.

String and Character Constants

A string constant is denoted text inside of double quotes. For now, we have yet to use String variables. String variables are nothing but the concatenation of several char characters.

A character constant is a single character inside of a single quote. This is different than a character variable, which must have a valid identifier name.

Increment and Decrement Operators

The assignment statements of the form:

```
x = x + 1;
x = x -  1;
```

are so common that they have the following shorthand notations:

```
x++; or ++x;
x--; or   --x;
```

When these statements are on a line by themselves, then the first two are equivalent as are the last two. However, in other situations, there is a slight difference between x++ and x--. These will be discussed at a later time.

Also note, that even though in mathematics, x = x+1 is a false statement, in C, it's perfectly logical. It takes the value of the variable x and adds 1 to it. When we discuss loops, you will see statements of this form often.

Other Shorthand Assignments

We can also take an assignment statement of the form:

<var> = <var> <op> <expr>

and shorten it to:

<var> <op>= <expr>

Here are a few examples:

x = x + 10;	**becomes**	**x += 10;**
x = x - 15;	**becomes**	**x -= 15;**
x = x*(3+y);	**becomes**	**x *= (3+y);**
x = x/(p*q);	**becomes**	**x /= (p*q);**

You are not required to use these shorthand notations in your code, but you should be able to trace through code that uses it.

Note about the Assignment Operator

We have already discussed how the assignment operator works, however, there are two pieces of information we didn't discuss about the assignment operator:

An assignment statement evaluates to the value of the right-hand side of the assignment.

The associativity of the assignment operator goes right to left, not left to right.

These two facts make the following statement possible:

int x = 3;
int y = 7;
int z = x = (y+2);

The last statement is actually grouped as follows:

int z = (x = (y+2));

First we evaluate the expression y+2, which is 9.
Then, this is the value that x will get assigned to.

When x gets assigned to 9, the whole expression (x = (y+2)) also evaluates to 9.

Finally, z gets assigned to this value as well.

Also, keep in mind that the assignment operator is NOT like an equal sign in mathematics. While in math it is okay to say x+2 = 9, in C, this does not make a valid assignment statement.