Lesson 2 C BUILDING BLOCKS

Three important aspects of any programming language are data storage, data input/output, and the operators that manipulate the data. This lesson discusses these three building blocks.

Variables

A variable is a space in computer memory that can hold different values at different times. In the previous lesson, we saw how the variable **num** was used.

int num;

In the above line, a variable of type int is declared. The statement *num=2;* is an example of an assignment statement. The variable **num** is assigned the value of 2. All variables must be defined in C before they are used. This specifies their name and type and sets aside storage requirements. When defining a variable, the compiler sets aside an appropriate amount of memory to store that variable. In the example above, the compiler sets aside two bytes of memory (some machines) since this is the amount of space an integer takes up in memory. Integers in C is large hold numbers from -32,768 to 32,767. Please note that defining and declaring a variable are two different things, see the exercises below.

Variable types

This list gives you some typical values for the various types available in C. Compilers may offer different limits and sizes since there is a lot of latitude in what a compiler may offer. The values in this list are for Microsoft Visual C++ version 1.5 (16 bits) and Visual C++ version 2.0 (32 bits). Check your own 64 bit compiler.

```
Type Name         Bytes   Range
  ------------- 16 bit system ------------
char              1       -128 to 127
signed char       1       -128 to 127
unsigned char     1       0 to 255 short
2       -32,768 to 32,767 unsigned short
2       0 to 65,535 int               2
-32,768 to 32,767 unsigned int      2
0 to 65,535
long              4       -2,147,483,648 to 2,147,483,647
unsigned long     4       0 to 4,294,967,295 float
4       3.4E+/-38 (7 digits) double            8
1.7E+/-308 (15 digits) long double      10       1.2E+/-
4932 (19 digits)

  ------------- 32 bit system ------------
char              1       -128 to 127
signed char       1       -128 to 127
unsigned char     1       0 to 255 short
2       -32,768 to 32,767 unsigned short
2       0 to 65,535
int               4       -2,147,483,648 to 2,147,483,647
unsigned int      4       0 to 4,294,967,295
long              4       -2,147,483,648 to 2,147,483,647
unsigned long     4       0 to 4,294,967,295 float
4       3.4E+/-38 (7 digits) double            8
1.7E+/-308 (15 digits) long double      10       1.2E+/-
4932 (19 digits)
```

The learner will notice that the only difference in these two lists are in the sizes and ranges of the **int** type variables, both signed and unsigned. The ANSI-C standard states that an **int** type has the "natural size suggested by the architecture of the execution environment", so the ranges for the compiler listed above comply with the standard.

One other point about the above table is worth noting at this time. The **char** type is permitted to be either signed or unsigned at the discretion of the compiler writer. The writers of the Microsoft compiler choose to make **char** default to a **signed char**, as do most compiler writers, but you have a choice since most compilers provide a switch to select the default to **unsigned char**.

Some useful constants are available for you in determining the range limits of the standard types. For example, the names INT_MIN and INT_MAX are available in the file "limits.h" as constants, which can be used in your code. INT_MAX is the largest possible number that can be stored in an **int** type variable using the compiler that you are currently using. When you switch to a new compiler, which you will almost certainly do at some stage, INT_MAX will refer to the largest value that can be stored with that compiler. Even if you switch to a new operating system with 64 bits or even 128 bits, INT_MAX will still refer to the largest **int** available on your new system. The file "limits.h" contains such limits, all of which are available for your use simply by including the file in your program. This file is a text file and can be opened in an editor and studied, a highly recommended exercise for you at this time.

You may have wondered why no string type is available. C treats strings as arrays of characters.

Example var.c

```
#include<stdio.h>

void main(void)
{
 int event;
 char heat;
  float time;
  event =5;
  heat =`C`;
  time = 27.25;
   printf(" The winning time in heat %c",heat);
   printf(" of event %d was %f\n",event, time);
 }
```

The output reads : **The winning time in heat C of event 5 was 27.25.0000**

Note that it is possible to combine a variable definition with an assignment operator so that the variable is initialised at the same time that it is defined.

This program uses three variable types. To print out these variables, different format specifiers must be used. See the next topic Input/Output.

Input / Output

It is not only important to store data and make some calculations with it, we also want to be able to read in from a keyboard and output the data to a screen. The functions that we will examine here are **scanf()**, **printf()** and **getchar()**.

printf()

Format Specifiers

We have already seen how the printf function is used to print out integers, floats and characters. By changing the format specifier, we can output other common type variables.

```
      %d    decimal integer
      %i    decimal notation (new ANSI standard extension)
      %o    octal notation
      %x    hexadecimal notation
      %u    unsigned notation
      %c    character notation
      %s    string notation
      %f    floating point (decimal notation)
      %g    floating point (%f or %e, whichever is shorter)
l     prefix used with %d,%u,%x,%o to specify long integer
```

Field Width Specifiers

The printf() function gives the programmer considerable power to format the printed output. Suppose for example that we want the variable time as used above to print out only the significant digits ie two decimal places. This make more sense since 27.25 is all that is required. We can suppress the extra zeros by using `%.2f`. The number 2 following the decimal point specifies how many characters are printed after the decimal point, in this case 2. A digit before the decimal point controls the width of the space to contain the number when it is printed. A minus sign preceding the field-width specifier will put the output on the left side of the field instead of on the right.

Escape Sequences

Here is a list of the common escape sequence

```
\n Newline
\t Tab
\b Backspace
\r Carriage return
\f Formfeed
\' Single quote
\" Double quote
\\ Backslash
\xdd ASCII code in hexadecimal notation (d is a digit)
\ddd ASCII in octal notation (d is a digit)
```

Graphics Characters

Graphics characters are printed using their hex code in an escape sequence, such as `\xB0`.

scanf()

The following gives an example of how the scanf() function can be used

```
/* here is a program in which the user types in
their age in      years and then their age in days is
printed to the screen */
```

```
#include<stdio.h>
void main(void)
      {
       float years, days;
        printf("Type in your age in years: ");
        scanf("%f",&years);
        days = years * 365;
         printf(" you are %.1f days old\n",days);
       }
```

The scanf() function is not unlike the printf() function, the argument on the left is a string that contains format specifiers, on the right we have the variable name, &years. The format specifiers for scanf() are similar to those for printf().

The scanf() function can accept input to several variables at once. See the example line below.

```
scanf("%d %c %f ",&event,&heat,&time);
```

Address operator (&)

The C compiler requires the address of a variable as an argument to scanf(). The address of a variable is the first byte in memory where the variable resides. When we declare a variable *int num;* a place in memory is set aside at memory address 56677 (for example). If we then set num to be equal to 4 ie *num = 4*; then the following print statement *printf("num=%d address of num=%d\n",num,&num);*

Produces the following output:

num = 4 address of num = 56677

Modify the code above to write a short program that prints out the address in memory of the variable years.

getchar()

If we want to input data we can use the getchar(). This function accepts a character from an input stream and you must press return after the character in order for it to be ingested. Compare the getchar() function with another input function in C, the getc() function. Are they interchangeable ?

Operators

Operators are words or symbols that allow a program to act on variables. In C there are many different types of operators, we shall only look at the most useful ones.

Arithmetic Operators

```
+ addition
- subtraction
* multiplication
/ division
% remainder
```

We can use the above operators as Assignment Operators. For example, consider the following

cost = cost + vat, this is the same as *cost+=vat*. In a similar way we can use -=, *= etc.

C allows the use of an Increment(++) and Decrement(--) Operator.

Example *price = price + 1* is the same as *price++*.

Relational Operators

A program to ask questions about variables can use this type of operators. There are six relational operators in C.

```
<  less than
>  greater than
<= less than or equal to
>= greater than or equal to
== equal to
!= not equal to
```

Consider the following short program

```
#include<stdio.h>
void main(void)
   {
    int money;
    money = 23;
    printf("Is the amount of cash less than 21 ? %d \n",money < 24 );
    money = 56;
    printf("Is the amount of cash less than 21 ? %d \n",money < 24 );
   }
```

The output from this program is

```
Is the amount of cash less than 21 ? 1
Is the amount of cash less than 21 ? 0
```

Note, as seen from the program, although C has no "boolean" type as in other languages, it can use the integer 1 as true and the integer 0 as false.

Precedence of Operators

Precedence of operators is an important topic in C. We will only need a few rules for the moment but later the topic will be discussed in detail. When you have mixed arithmetic expressions, the multiplication and division operations are completed before the addition and subtraction operations when they are all at the same logical level.

Therefore, when evaluating *a * b + c / d*, the multiplication and division are done first, then the addition is performed. However in the expression a * (b + c / d), the addition follows the division, but precedes the multiplication because the operations are at two different logical levels as defined by the parentheses. We say that `*` and `/` have higher **precedence** than `+` and `-`. When we mix Arithmetic and Relational Operators, the Arithmetic Operators are evaluated before the Relational Operators i.e. they have higher precedence. Consider the following

```
void main(void)
    {
     printf("Answer is %d",2+1 < 4);
    }
```

The answer is 1. Why?

Try and guess what the answer will be from the following lines of code.

```
void main(void)
    {
     printf("Answer is %d", 1<=2 +4);
     }
```

Once again, the answer is 1. Why?

Definition and Executable Statements

Variable definitions coded before any executable statements in any program block. This is why variables must be defined at the beginning of a block in every C program. If you try to define a new variable after some executable statements, your compiler will issue an error. A program block is any unit of one or more statements surrounded by braces.

ANSI-C Modifications

Two new keywords have been added to C with the release of the ANSI-C standard. The two new keywords are **const** and **volatile.** They are used to tell the compiler that variables of these types will need special consideration.

To declare a constant, use the **const** keyword. This declares a value that cannot be changed by the program. If you try to modify an entity defined as **const**, the compiler will generate an error. Declaring an entity as **const** allows the optimizer to do a better job in making your program run a little faster. Since constants can never have a value assigned to them in the executable part of the program, they must always be initialized.

If the **volatile** keyword is used, it declares a value that may be changed by the program but some outside influence such as a clock update pulse incrementing the stored value may also change it. This prevents the optimizer from getting too ambitious and optimizing something that it thinks will never be changed.

Example of declaring a constant is:

```
const float Pi = 3.14;
```

Example of using a volatile statement is:

```
volatile int time ;
```

Programming Exercises

1. Write a program to read in 3 integers and print out the average of them to the screen.
2. Write a program to read in a temperature in Centigrade and output the corresponding temperature in Fahrenheit.

3. Given that the gradient of a straight line is found from the formulae (y2-y1)/(x2-x1) write a program that reads in two points (x1,y1), (x2,y2) and prints out the slope of the line joining the two points.
4. Modify the program in Q3 to first print out the point at which the line joining the points crosses the Y-axis. Then print out the equation of the line in the form "y = m x + c".
5. What are the two basic input functions in C? Use them in a program.
6. Write a program to output the square of a number that the user enters.
7. Type in the follow program and compile it. Comment on the output.

```
void main(void)
      {
       int age;
       age = 15;
       printf("Is age less than 21 ? %d \n", age < 21);
       age = 22;
        printf("Is age less than 21 ? %d \n", age < 21);
      }
```

8. Explain how would write a program to read in 3 numbers and calculate the standard deviation of the three numbers. Document it. Using your documentation, write the code and test with the following inputs: Comment on your answers.

   a) 50,99,101

   b) 48,50,52

   c) 2.1,3.45,5.573