LESSON 5

ARITHMETIC DATA PROCESSING

The arithmetic data types are the fundamental data types of the C language. They are called "arithmetic" because operations such as addition and multiplication can be performed on them. Other data types such as arrays, pointers, and structures are derived from the fundamental types. They are presented in later lessons. This lesson discusses in detail the fundamental types, including their memory requirements. The number of bytes used to store a type in memory determines the range of values that can be assigned to a variable of that type. The lesson explains how the sizeof operator can be used to determine the storage requirement for any object, including a fundamental type. The lesson continues with a discussion of the mathematical library. Although functions such as sin( ) and cos( ) are not technically part of the C language, they usually are available in a mathematical library <math.h>. Examples are given to show how mathematical functions are used. After this the lesson discusses conversions and "type cast". In expressions with operands of different types certain implicit conversions occur. The rules for conversion are explained, including the cast operator that forces explicit conversion. The C language provides several fundamental types. Many of them you have already seen. For all of them the lesson discusses limitations on what can be stored in the memory allocated for the type.

FUNDAMENTAL DATA TYPES

char	short	int	long
unsigned char	unsigned short	unsigned	unsigned long
float	double		

The keywords short int, long int, and unsigned int may also be used. They are equivalent to short, long, and unsigned, respectively. Also, long float is equivalent to double. The fundamental types can be grouped into two separate categories:

integral types*:* char, short, int, long, unsigned char, unsigned short, unsigned, unsigned long

floating types*:* float, double

fundamental types = integral types + floating types

These collective names are a convenience. Later, for example, when the lesson discusses arrays, the lesson will explain that only integral expressions are allowed as subscripts, meaning only expressions involving integral types are allowed.

In C the data type int is the principal working type of the language. This type, alone with the other integral types such as short and long, is designed for working with the integer values that are represented on a computer. In addition to decimal integer constants, there are hexadecimal and octal constants. the lesson will not use them, but the reader has to know that integers that begin with a leading 0, such as 011 and 057, are octal numbers, not decimal. Hexadecimal integer constants begin with Ox or OX, as in 0x22 and 0X4.

Some examples of decimal integer constants are:

```
0
6
123
123456789123456789      /* too big for the computer memory? */
```

not decimal integer constants

```
5.0          /*  a floating constant */
0x66         /*  a hexadecimal integer, not decimal */
077          /*  an octal integer, not decimal */
1,890        /*  comma not allowed */
-444         /*  this is a constant expression */
```

A variable of type int cannot take on all integer values; some integers are too large to be stored in a computers memory. Moreover, the range of values that an int can hold is memory dependent. Typically, an int is stored in 4 bytes (which is 32 bits) large computer memory and in 2 bytes (which is 16 bits) in small computer memory. The newer computer now allow 64 bit memory allocation thus 8 byte integers. You should know how large the computer word is on the central processor unit (CPU) you are using.

On 32 bit memory computers:

the smallest int is about -2,147,483,648 or -2 billion
the largest int is about +2,147,483,647 or +2 billion

On 16 bit memory computers:

the smallest int is about -32.768 or -32 thousand
the largest int is about +32,767 or 32 thousand

When the a value larger than an int can hold is assigned to an int type variable, an overflow error occurs. This condition is called an integer overflow. Typically, when an integer overflow occurs, the program continues to run, but with logically incorrect results. For this reason you the programmer must strive at all times to keep the values of integer expressions within the proper range depending on the computers memory size.

In C, the data type int is considered the "natural" or "usual" type for working with integers. The other integral types: char, short, long, and the unsigned types, are intended for more specialized use. For example, the data type short might be used in situations where storage is of concern. The compiler may provide less storage for a short than for an int, although it is not required to do so. In a similar fashion the type long might be used in situations where large integer values are needed. The compiler may provide more storage for a long than for an int, although it is not required to do so. Typically, on both large and small computers a short is stored in 2 bytes and a long is stored in 4 bytes. Thus on most 32 bit memory computers the size of an int is the same as the size of a long, and on most 16 bit memory computers the size of an int is the same as the size of a short. A variable of type unsigned is stored in the same number of bytes as an int. However, as the name implies, the integer values stored have no sign. The sign bit is reserved for plus and minus. Typically, variables of type int and unsigned are stored in a machine word. The machine word depends on the CPU's memory allocation (16, 32 or 64 bits).

USING THE TYPE long

Since on most small machines an int can store values only up to approximately 32 thousand, the type long is often used. Constants of type long can be specified by appending the letter L or l to the number. For example, 22L is a constant of type long. On systems where an int is stored in less memory than a long, a constant too big to be stored in an int is automatically converted to a long. For example, on small machines the constant 33000 is automatically of type long.

When using printf( ) and scanf( ) with variables and expressions of type long, the conversion character d should be preceded by the l or L modifier. Here is are example:

```
long a, b, c;

printf("lnput three integers: "):

scanf("%Ld%Ld%Ld", &a, &b, &c);

printf("\nHere they are:\n");

printf("a= %Ld\nb = %Ld\nc= %Ld\n", a, b, c);
```

THE sizeof OPERATOR

The C language provides the unary operator sizeof to find the number of bytes needed to store an object. It has the same precedence and associativity as all the other unary operators. An expression of the form

```
sizeof(object)
```

returns an integer that represents the number of bytes needed to store the object in memory. An object can be a type such as int or float, or it can be just a variable such as a, or it can be an expression such as a+b, or it can be an array or structure type. The following program uses this operator. On a given computer it provides precise information about the storage requirements for the fundamental data types and the user defined data type.

```
main( )
{
  printf("The size of the fundamental data types is shown below.\n");
  printf(" char is %d bytes \n" , sizeof(char));
  printf(" short is %d bytes \n" , sizeof(short));
  printf(" int is %d bytes \n" , sizeof(int));
  printf(" unsigned is %d bytes \n" , sizeof(unsigned));
  printf(" long is %d bytes \n" , sizeof(long));
  printf(" float is %d bytes \n" , sizeof(float));
  printf(" double is %d bytes \n" , sizeof(double));
}/* end of  main function */
```

All that is guaranteed by the C language is that the

sizeof(char) = 1;

sizeof (short) <= sizeof(int) <= sizeof(long)

sizeof(int) <= sizeof(unsigned)

sizeof(float) <= sizeof(double)

The C language provides the two floating types float and double to deal with the real numbers such as 12.34 and 0.00123 and pie which is 3.1415926. Integers are represented as floating constants, but they must be written with a decimal point. For example, the constants 1.0 and 2.0 are both of type double, whereas the constant 3 is an int. All floating constants are of type double; there are no constants of type float.

In addition to the ordinary decimal notation for floating constants, there is an exponential notation (using the letter e) as in the example 1.234567e3. The number 1.234567e-3 calls for shifting the decimal point 3 places to the left to obtain the equivalent constant 0.001234567.

The exponential notation has precise rules and the lesson will show some examples. A floating constant such as 333.77777e-22 may not contain any embedded blanks or special characters. Each part of the constant is given a name:

333 is the integer part
77777 is the fractional part
e-22 is the exponential part.

A floating constant may contain an integer part, a decimal point, a fractional part, and an exponential part. A floating constant must contain either a decimal point or an exponential part or both. If a decimal point is present, either an integer part or fractional part or both must be present. If no decimal point is present, then there must be an integer part along with an exponential part. Some examples of floating constants are:

```
3.14159   /*  this is a floating constant expression */
314.159e-2
0.0
OeO  /* is equivalent to 0.0 */
1.0
1.  /* equivalent to 1.0, but harder to read user should put in the 0*/
```

but not

```
3.e14,159   /*The  comma is not allowed */
```

```
314159 /* The decimal point or exponential part is needed */
e0  /* The integer part or fractional part is needed */
```

The C compiler will provide more storage for a variable of type double than for one of type float. On most machines a float is stored in 4 bytes and a double is stored in 8 bytes. The effect of this is that a float stores about 6 decimal places of accuracy, and a double stores about 16 decimal places of accuracy. All floating constants are of type double.

The range of values that a floating variable can be assigned is computer dependent. Typically, positive values range from about 10^{-38} to 10^{+38} for both a float and a double. The two key points that the programmer must be aware of are that not all real numbers are represented, and that floating arithmetic operations, unlike the integer arithmetic operations, my not be computed exactly.

MATHEMATICAL FUNCTIONS

In C there are no built-in mathematical functions. Functions such as:

sqrt( )	pow( )	exp( )	log( )	sin( )	cos ( )	tan( )

usually occur in a special library called:

```
#include <math.h>
```

All of the above functions, except the power function pow( ), take a single argument of type double and return a value of type double. The power function takes two arguments of type double and returns a value of type double.

The next example will show how the function for the square root sqrt( ) can be applied. Recall the definition of the square root. If x is a nonnegative number, then the square root of x is a nonnegative number s having the property that: s times s yields x. For example, the square root of 4 is 2, because 2 times 2 is 4. The interactive program asks the user to input a number, and then prints out the number along with its square root.

```
/* Example program to compute square roots. */
#include <math. h>
main ( )
{
  double x;
  printf("\nThe square root of x is computed.\n\n" );
  for (; ; ) {                          /*A for loop forever.*/
     printf ("Input x  ==>");
     scant ("%Lf", &x);
```

```
    if ( x >= 0.0)
      printf ( "\n%10s%.16f\n%10s%.16f \n\n",  "x = ", x, "sqrt(x) = ",
               sqrt(x));
    else
      printf ("\nSorry, your number was negative. \n\n");
  }/* end for loop */
}/* end main */
```

The included file <math.h> contains the declarations of all the functions in the mathematical library. This file does not contain the sqrt( ) function itself. Now let us suppose that you execute this program. If you input the number 2 when prompted, the following appears on the screen:

The square root of x is computed.

```
Input x  ==>2


x = 2.0000000000000000

sqrt(x) = 1.4142135623730951

Input x  ==>
```

PROGRAM ANALYSIS

This declaration

```
double x;
```

tells the compiler that x is a variable of type double. If the function sqrt( ) were not obtained from a library supplied by the system, it would be declared in the program, if it is not declared, the compiler will assume that it is a function returning an int. The math.h include file provides the proper declaration, called the function prototype.

The infinite loop:

```
for (; ; ) {                    /*A for loop forever.*/
```

will loop until an interrupt is entered. What must be typed to enter an interrupt is system dependent. On some systems this is done by typing a control-c.

A prompt for the user is printed.

```
Input x  ==>
```

The function scanf( ) is used to convert the characters typed in by the user to a long float, or equivalently a double.

```
scant ("%Lf", &x);
```

The appropriate value is stored at the address of x. Notice that you typed 2 to illustrated the use of this program.

Equivalently, you could have typed 2.0 or 2e0 or 0.2e1 The function call scanf("%Lf", &x) would have converted each of these to the same double value in memory. In C code 2 and 2.0 are stored in memory different. The first is of type int, and the second is of type double. The input stream that is read by scanf( ) is not code, so the rules for code do not apply. When scanf( ) reads in a double, the number 2 is just as good as the number 2.0.

```
if ( x >= 0.0)


else
      printf ("\nSorry, your number was negative. \n\n");
```

Since the sqrt( ) function is not designed to work will negative numbers, a test is made to ensure that the value of x is positive. If the test fails, an explanation is printed "Sorry, your number was negative".

```
printf ( "\n%10s%.16f\n%10s%.16f \n\n",   "x = ", x, "sqrt(x) = ",
                sqrt(x));
```

The value for x and the computed value of sqrt(x) are printed. The format %.16f is used because on most systems a double is stored in 8 bytes, and this gives about 16 decimal places of accuracy. Here it would be just as reasonable to use %.16e or %.16g.

CONVERSIONS AND CASTS

An arithmetic expression such as x + y has both a value and a type. For example, if x and y are both variables of the same type, say int then x + y is also an int. However, if x and y are of different types, then x + y is called a mixed

expression. Suppose x is a short and y is an int. Then the value of x is converted to an int and the expression x + y has type int. The value of x as stored in memory is unchanged. It is only a temporary copy of x that is converted during the computation of the value of the expression. Now suppose that both x and y are of type short. Even though x + y is not a mixed expression, automatic conversions again take place; both x and y are promoted to int and the expression is of type int.

The rules are for automatic conversion in an arithmetic expression

First:

- Any char or short is promoted to int
- Any unsigned char or unsigned short is promoted to unsigned.

Second:

- If after the first step the expression is of mixed type, then according to the hierarchy of types int < unsigned < long < unsigned long < float < double

the operand of lower type is promoted to that of the higher type and the value of the expression has the higher type.

This process goes under various names:

- automatic conversion
- implicit conversion
- coercion
- promotion
- widening

In addition to implicit conversions, which can occur in a mixed expressions, there are explicit conversions called cast. The word cast is not a keyword in the C language. The cast is performed by enclosing in parentheses a type in front of the type variable to be cast to. An example is if i is an int, then

```
(double) i
```

will cast the value of i so that the expression has type double. The variable i itself remains unchanged. Casts can be applied to any type or expressions. Some examples are

```
(char) ('A' +1.0);
x = (float) ( (int) y + 1);
(double) (x = 77);
```

but not

```
(double) x = 77;       /*Which is equivalent to ((double) x) = 77 */
```

The cast "operator" (type) is a unary operator having the same precedence and "right to left" associativity as other unary operators. The expression

(float)1 + 3 is equivalent to ((float)1) + 3

because the cast operator (float) has higher precedence than +.