LESSON 1

FUNDAMENTALS OF C

The purpose of this lesson is to explain the fundamental elements of the C programming language. C like other languages has all alphabet and rules for putting together words and punctuation to make correct or legal programs. These rules are the syntax of the language. The program that checks the legality of C code is called the compiler. The C code is called the source code and is in ASCII.

Compilers are very precise in their requirements. The C compilers will not provide a translation of a syntactically incorrect program, no matter how trivial the program error. The programmer must learn to code precise C code syntax when writing code. The programmer should also write understandable code. A key part of this is to have well commented code and meaningful identifier names. This lesson will explain these important concepts, including a discussion of comment style. The Natural and Applied Science Computer Science uses the style in Reference 1.

A C program is constructed as a sequence of characters. Among the characters that can be used in a program are:

1. lowercase letters abc...z
2. uppercase letters ABC...Z
3. digits 0123456789
4. other characters +-*/=(){}[]"!#$%&l""\.:?
5. white space characters such as blank newline, and tab.

These characters are used to produce the source language elements of C. The basic language elements are identifiers, keywords, constants, operators, and other separators. Let us look at a simple program and pick out some of these elements.

```
/* Read in -two integers and print their sum.*/
main ( )
 {
   int a, b, sum;
   printf ("Input two integers: \n")f
   scant ("%d%d", &a, &b);
   sum=a+ b;
   printf ("\nThe sum of %d and %d is %d\n",a, b, sum);
 }
```

PROGRAM ANALYSIS

/* Read in two integers and print their sum. */

Comments are delimited by /* and */ and are treated as white space by the compiler. They must be in pairs and enclose the intended information.

```
main ( )
 {
   int a, b, sum;
```

The function name main is an identifier and a keyword in C. The left and right characters ( ) and right bracket { are separators. The keyword int is a keyword telling the compiler that a, b and sum are identifiers, and the comma, and semicolon ; character are separators.

```
printf ("Input two integers: \n")f
scant ("%d%d", &a, &b);
```

The function names printf( ) and scanf( ) are keyword identifiers and the parentheses following them tell the compiler that they are functions. These functions are in the standard library called stdio.h. A programmer does not normally redefine these identifiers by using their names.

"Input two integers:”

A series of characters enclosed in double quotes “ is a string constant.

&a,&b

The character & is an address operator which will be explained latter.

```
sum = a + b;
```

The characters = and + are operators. White space here will be ignored, so the expression could have written

```
sum=a+b;
```

Good programming style is to put one blank space on each side of binary operators to allow readability of the source code by the programmer and user.

The keyword int specifies that the identifiers or variable names in the list following the keyword contain only integer values. A list is a sequence of identifiers separated by commas and end with the semicolon. Characters such

as = and + are operators. The semicolon character ; is punctuation used to terminate declarations and statements. White space is a bland and is used by the C compiler to determine how to separate elements of the language. Programmers to provide legible code also use white space. To the C compiler the ASCII text source code is a single stream of characters. To the programmer or user reading the code, it is two-dimensional.

An identifier is a sequence of letters, digits, and the special character. The special character _ is called an underscore. A letter or underscore in C must be the first character of an identifier. In all implementations of C the upper and lowercase letters are treated as distinct. They are case sensitive meaning a and A are not the same. It is good programming practice to choose identifiers that have mnemonic significance to your application so they contribute to the readability and documentation of the program.

Some examples of identifiers are:

```
total_tax_rate
totaltaxrate
_total
```

do not use the special character # in an identifier

```
a#pound  /* special character # not allowed */
5_dollars  /* must not start with a digit */
-minus_  /* do not mistake dash or minus - for the underscore _*/
```

Identifiers are created to give unique names to various variables names or objects in your program. Some words are reserved as special to the C language; these are called keywords. Although not technically part of the C language, the identifiers scanf( ) and printf( ) are already known to the C compiler as input/output functions in the standard library stdio.h. The identifier main is also special in that C programs always begin execution at the function called main( ).

Good programming style requires the programmer to choose names that are meaningful. To write a program to compute taxes, you might have identifiers such as tax rate, price, and total tax. The C compiler does not know that tax rate is one identifier so you must use tax_rate to signify one name or identifer.

The statement:

```
total_tax = price * tax_rate;
```

would have an obvious meaning to you and the C compiler. The underscore is used to create a single identifier from a string of words separated by white spaces. Meaningful identifier and avoiding reader confusion go hand in hand to

constitute the guidelines for a good C programming style. Keywords are explicitly reserved identifiers that have a strict meaning for the C compiler. They should not be redefined or used in other contexts.

KEYWORDS

auto	default	extern	int	sizeof	unsigned
break	do	float	long	static	void
case	double	for	register	struct	while
char	else	goto	return	switch	
continue	enum	if	short	typedef	
const	far	near	volatile	union	

C has only a small number of keywords and a characteristic of C is that you can do a lot with few special symbols and keywords.

C has many special characters that have significant meaning based on the type of identifier they are used with.

Examples of the arithmetic operators are:

+	-	*	/	%

They are the arithmetic operations of addition, subtraction, multiplication, division, and modulus, respectively. In a C program these will often separate identifiers called operands. For example: operand operator operand.

```
a – b        /*the expression a minus b */
a_b          /*a three character identifier with the underscore*/
```

Some symbols have meanings that depend on C context. An example of this is the % symbol. In the statement:

```
printf("%d", a);
```

the percent % symbol is a format control character. In an arithmetic expression:

```
4 + 5 % 3
```

the % symbol is the modulus operator returning the value 2.

Special characters and the white space separate language elements, and the special characters are used in many different contexts. The context will determine which use is intended.

For the expressions:

a + b	++a	a += b

all use + as a character. The single + means add a to b, the ++ is a single operator, an + = is another single operator. Since C has the meaning for a symbol dependent on context allows C to have a small symbol set and a terse language. Operators have rules of precedence and associativity that determine precisely how expressions are evaluated by the compiler. You as a programmer must learn this syntax. Since expressions inside parentheses are evaluated first, parentheses can be used to clarify or change the order in which operations are performed. This is called precedence. Consider the expression:

3 + 2 * 5

In C the operator * has a higher precedence than + , and the multiplication is performed first, and then the addition. The value of the expression is 30. An equivalent expression with parentheses is:

3 + (2 * 5)

To change the order of precedence use parentheses, since expressions inside parentheses are evaluated first by C. The expression:

(3+2)*5

is 25.

Now consider the expression:

1 + 2 – 3 + 4 – 5

Because the operators + and have the same precedence, the associativity rule "left to right" is applied in C for evaluation. The "left to right" rule means that the operations are performed from left to right. Thus:

(((1 + 2) –3) + 4) - 5

is an equivalent expression. The following table gives the rules of precedence and associativity for some of the operators of C. In addition to the operators already discussed, the following table includes operators that will be discussed in this lesson.

OPERATOR TABLE

Precedence	Associativity	Type	Operator	Name
15	Left to Right	Primary	( )	Parenthesis
			[ ]	Subscript
			->	Arrow
			.	Member (dot)
14	Right to Left	Unary	!	Logical Not
			~	Bitwise Not
			++	Increment
			--	Decrement
			-	Negative
			( )	Type Cast
			*	Indirection
			&	Address of
			sizeof	size of
13	Left to Right	Arithmetic	*	Multiplication
			/	Division
			%	Modulus
12	Left to Right	Arithmetic	+	Addition
			-	Subtraction

OPERATOR TABLE Continued

Precedence	Associativity	Type	Operator	Name
11	Left to Right	Bitwise	<<	Left Shift
			>>	Right Shift
10	Left to Right	Relational	>	Greater Than
			>=	Greater Than or Equal
			<	Less Than
			<=	Less Than or Equal
9	Left to Right	Relational	==	Equal tp
			!=	Not Equal
8	Left to Right	Bitwise	&	Bitwise AND
7	Left to Right	Bitwise	^	Exclusive OR
6	Left to Right	Bitwise	|	Bitwise OR
5	Left to Right	Logical	&&	Logical AND
4	Left to Right	Logical	||	Logical OR
3	Right to Left	Conditional	? :	Then, Else
2	Right to Left	Assignment	=	Assignment
			+=,-=,*= /=,%=, etc.	Combination Assignment
1	Left to Right	Sequence	,	Comma

THE C OPERATORS

All the operators with a given precedence number such as level 13 (* and / and %), have equal precedence with respect to each other, but have higher precedence than all the operators with precedence values below them. The associativity rule for all the operators with a given precedence appears in the second column of the table. Whenever we introduce new operators, we will give their rules of precedence and associativity. These rules are essential information for every programmer.

In addition to the binary minus, which represents subtraction, there is a unary minus, and both these operators are represented by a minus sign. The precedence of a unary minus is higher than * and / and %, and its associativity is "right to left". In the expression

```
-a*b-c
```

the first minus sign is unary and the second is binary. Using the rules of precedence, the equivalent association expression with parentheses is:

```
((-a) * b) - c
```

INCREMENT AND DECREMENT

The increment operator ++ and decrement operator -- are unary operators with the same precedence as the unary minus, and they all associate from right to left. Both + + and -- can be applied to variables, but not to constants or ordinary expressions. They can also occur in either prefix or postfix position, with possibly different effects occurring. Some examples are:

```
++a
a++
-count
count-
777++  /*A constant cannot be incremented */
++(a * b -1)  /*An expressions cannot be incremented */
```

Each of the expressions ++a and a++ has a value, and moreover, each causes the stored value of a in memory to be incremented by 1. Each of the expressions takes as its value the value of a, but with ++a the stored value of a is incremented before the value of a is computed, whereas with a++ the stored value of a is incremented after the value of a is computed. The following code illustrates the situation.

```
int a, b, c = 0;  /*The c integer is assigned a value 0 */
a = ++c;
```

```
b = c++;
printf("%d %d %d\n", a, b,c);
/* a = 1, b = 1, c = 2 */
```

In a similar fashion --a causes the stored value of a in memory to be decremented by 1 first, and this new value is the value of the expression. In contrast, the expression a-- takes as its value the current value of a, and it is only after the value of the expression has been computed that the stored value of a is decremented by 1.

Be aware that ++ and -- are unlike other operators in that they cause the value of a variable in memory to be changed. For example, the operator + does not cause a variables value to be incremented. An expression such as a + b has value, but the stored values in memory of the variables a and b are left unchanged. These ideas are expressed by saying that ++ and -- have what is called a side effect. These operators yield a value and also change the value of the variable stored in memory. In some cases we can use ++ in either prefix or postfix position with both uses producing equivalent results. For example, each of the two statements

++a; and a++; is equivalent to a = a + 1;

In a similar fashion each of the two statements

--a; and a--; is equivalent to a = a - 1;

In simple situations one can consider ++ and -- as operators that provide concise notation for the incrementing and decrementing of a variable. In other situations careful attention must be paid as to whether prefix or postfix position is desired.

ASSIGNMENT

To change the value of a variable **a** use the assignment statements such as:

```
a=b+c;
```

C treats equal sign = as an operator. Its precedence is lower than all the operators discussed so far and its associativity is right to left. To understand the assignment operator = as an operator, first consider + for comparison. The binary operator + takes two operands, as in the expression a + b. The value of the expression is just the sum of the values of a plus b. In comparison, a simple assignment expression is of the two forms:

```
variable = rightside
variable = rightside;
```

where the rightside is itself an expression. Notice that a semicolon placed at the end of the second expression makes it an assignment statement. The assignment operator = has the two operands, the variable and the rightside. The value of rightside is assigned to the variable and that value becomes the value of the assignment expression. To illustrate this, consider:

```
int a, b, c;
b = 2;
c = 3;
a = b + c;
```

The first two assignment statements assign the values 2 and 3 to b and c, respectively. Then the value of b + c is computed and assigned to a. By using assignment expressions, this could be condensed to one statement:

```
int a, b, c;
a=(b=2)+(c=3);
```

The assignment expression b = 2 assigns the value 2 to the variable b, and the assignment expression itself takes on this value. The assignment expression c = 3 assigns the value 3 to the variable c, and the assignment expression itself takes on this value. Finally, the values of the two assignment expressions are added and the resulting value 5 is assigned to a. Although the above example is simple, there are many situations where assignment occurs naturally as part of an expression. When we get to the if statement another precautionary example will be given. A frequently occurring situation is multiple assignment. Consider the statement:

```
a=b=c=0;
```

Since the operator = associates from right to left, an equivalent statement is

```
a = (b = (c = 0));
```

First c is assigned the value 0 and the expression c = 0 has value 0. Then b is assigned the value 0 and the expression b = (c = 0) has value 0. Finally, a is assigned the value 0 and the expression a = (b = (c = 0)) has value 0.

There are other assignment operators such as plus equal += and minus equal -=. An expression such as

```
a = a + 2
```

will add 2 to the old value of a and assign the result to a, and the expression on the righthand side will have that value. The expression

a += 2

in C produces the same result as the previous one. The following table contains all the assignment operators.

ASSIGNMENT OPERATORS

=	+=	-=	*=	/=	%=	>>=	<<=	&=	^=	|=

These operators have the same precedence, and they have Right to Left associativity. The semantics is specified by

variable operator = expression

and is equivalent to

variable = variable operator expression

There is one exception and that is if the variable is an expression, the variable is evaluated only once. An assignment expression such as

b *= 2 + a is equivalent to b = b* (2 + a)

rather than

b = b * 2 + a

COMMENTS

Comments are programmer defined strings of symbols placed between the two pairs of delimiters /* and */. Two examples are:

```
/*This is a comment */
/****************This is a fancy comment******************/
```

The following illustrates some styles that can be used to give meaning to comments or if you want to enclose a prolog or a sentence.

```
/***
***This is a comment that
***contains information about the
***program style that you are to use
***when you code your program.
***/
```

```
/*********************************************************************************************
*                                                                                            *
* You can put comments in a box                                                              *
*but do not put another pair of comment delimiters in the box                                *
*the C compiler will not like it.                                                           *
*                                                                                           *
*********************************************************************************************/
```

Comments are not part of the executable program, but are used by the programmer as a documentation aid. The documentation is used to explain how the program works and how it is to be used by another person. Comments should contain information documenting the variables, the logic and flow of control in the program. Comments should be written simultaneously with the C program statements. Frequently beginner programmers leave the insertion of comments as a last step in documentation. There are two problems with this. The first is that once the program is running or executing, the tendency is either to omit or abbreviate the comments. The second is that ideally the comments should serve as an ongoing dialogue with the programmer, indicating program structure and contributing to program clarity and correctness. The comments cannot serve this purpose if they are inserted after the coding is finished.

Out of Place

As we have seen in some simple introductory programs, C manipulates various kinds of values, some of them constants and some of them variables. Integers such as 5 and floating numbers such as 3.14159 are examples of constants. Also, there are character constants such as 'a', 'b', and 'c', and there are string constants such as "a string is a freighted knot". Note carefully that character and string constants are different. For example, 'a' and "a" are not the same, one is a character and the other is a string. A good way to remember a character is a single a so use the single' make. A string can have multiple characters so use " mark.