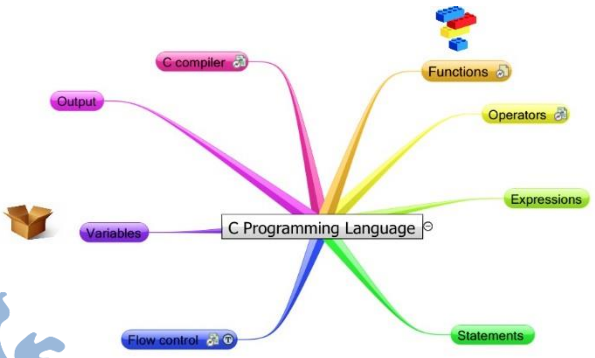
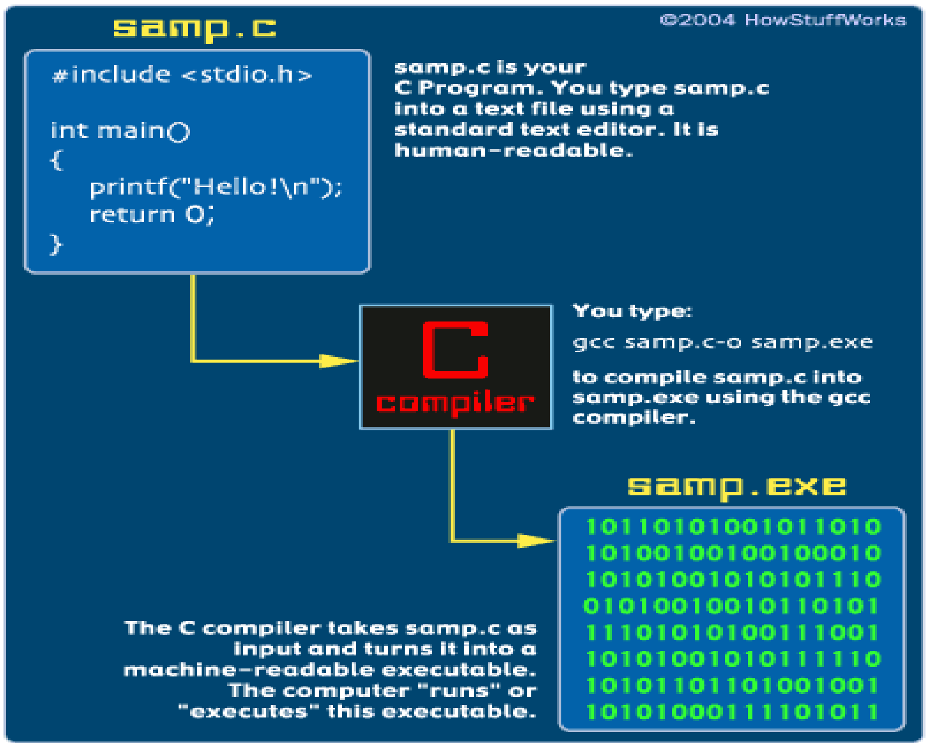
March
Towards
Excellence



C –PROGRAMMING HANDOUT

Detailed Lesson notes with practical exercises and applications in C-programming for Computer Engineers.


samp.c
©2004 HowStuffWorks
samp.c is your C Program. You type samp.c into a text file using a standard text editor. It is human-readable.
C compiler
You type:
gcc samp.c-o samp.exe
to compile samp.c into samp.exe using the gcc compiler.
samp.exe
The C compiler takes samp.c as input and turns it into a machine-readable executable. The computer "runs" or "executes" this executable.

```
#include <stdio.h>

int main()
{
   printf("Hello!\n");
   return 0;
}
```

```
1011010100101101 0
1010010010010001 0
1010100101010111 0
0101001001011010 1
1110101010011100 1
1010100101011111 0
1010110110100100 1
1010100011110101 1
```

By **NYAMBI BLAISE**
GBHS-MANKON
Software Engineer
PLET Computer Science
Tel: 679194380
masterb.ise@gmail.com

November 2019

Table of Contents

TOPIC 1

GETTING STARTED WITH C PROGRAMMING

C (and its object oriented version, C++) is one of the most widely used third generation programming languages. Its power and flexibility ensure it is still the leading choice for almost all areas of application, especially in the software development environment.

Many applications and many Operating systems are written in C or C++, including the compilers for other programming languages, Unix, DOS and Windows. It continues to adapt to new uses with new programming languages like Java, C#, Python, PHP, etc which lay the foundation on C language.

Objectives
At the end of this lesson, student should be able to:
- Describe the structure of a C-program
- write, compile and execute a simple C-program using the command "***printf***" and escape sequences, and containing comments

Contents

I. WHAT IS A C/C++ PROGRAMMING LANGUAGE?

The C/C++ language consists of two basic elements:

1) Syntax: This is a language structure (or grammar) that allows humans to combine these C commands into a program that actually does some-thing.
2) Semantics: This is a vocabulary of commands that humans can understand and that can be converted into machine language, fairly easily, using compiler

A **C program** is a text file containing a sequence of C commands put together according to the laws of C grammar. This text file is known as the **source file** carrying the extension **.C**

II.   C & C++: SIMILARITIES AND DIFFERENCES

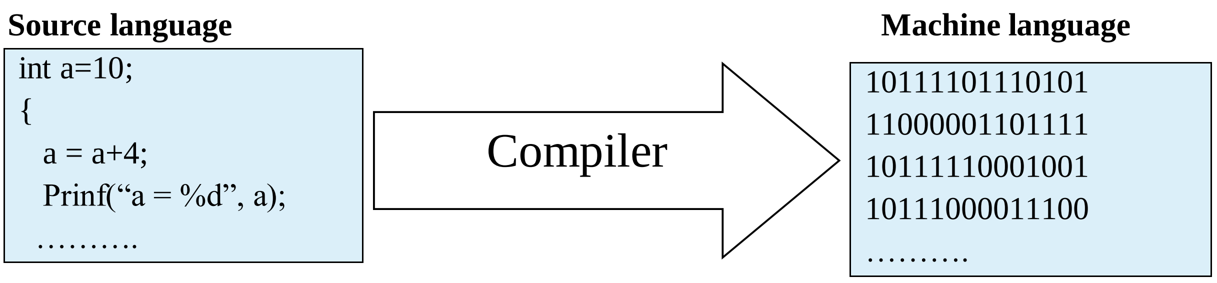
C++, as the name suggests is a superset of C. As a matter of fact, C++ compitler can run most of C code while C cannot run C++ code. Here are some basic differences between C++ & C...

1. C follows the procedural programming paradigm while C++ is a multi-paradigm language (procedural as well as object oriented)
2. In case of C, the data is not secured while the data is secured(hidden) in C++(due to specific OOP features like Data Hiding which are not present in C).
3. The standard input & output functions differ in the two languages (C uses ***scanf*** & ***printf*** while C++ uses ***cin***>> & ***cout***<< as their respective input & output functions)

III.   WHAT DO WE NEED?

To write a program, you need two specialized computer programs.

- an **editor** which is what you use to write your code as you build your *.C source file*.
- a **compiler (& linker)** converts your source file into a machine-executable **.EXE** file that carries out your real-world commands.



```
int a=10;
{
  a = a+4;
  Prinf("a = %d", a);
 ……….
```

```
10111101110101
11000001101111
10111110001001
10111000011100
……….
```

There are tools combining both compiler and editor in one called development **environment**. We will use ***DevC++ IDE*** (Integrated Development Environment), **version 5.5.3.** It can be downloaded freely in www.sourceforge.net

IV.   MY FIRST C-PROGRAM

[See Activity 1.1]

We are going in this paragraph to show different steps to handle a C-program from the edition to the execution

1) Launch the development environment (*Here Dev C++*)

To launch the Dev C++ environment, the process is :
Start button → All program → Bloodshed Dev-C++ → Dev C++

2) Open the text editor and type the program

To open a text editor in a Dev-C++ environment:
File → New → Source file

The program can be now typed into the editor using the c syntax. Our first program on what we are going to discover the functioning of a C-program is:

```
#include <stdio.h>
int main()
{
    /* my first program in C */
    printf("This is my first experience in C! ");
    return 0;
}
```

3) Save the program

To save the program: The file saved is called the **source file**

File → save as → in the dialog box type the name and choose the type** (here the type to choose is C source code) **→ save.

4) Compile and link the program

Compiling is the process of transforming the **source code** (the instructions in the text file) into the **object code** (instructions the computer's microprocessor can understand). The linking step is where the instructions are finally transformed into a program file. (Again, your compiler may do this step automatically.)

To compile the program: ***Execute → Compile***

After the compilation, if everything is ok, an executable code will be created with the same name and in a same folder and with the extension **.exe.** Else the compiler will produce an error message. In this case **debug** the program (check the code and correct the error(s)) and compile the source code again.

5) Execute the program

Finally, you run the program you have created. Yes, it's a legitimate program, like any other on your hard drive.

To execute the program: ***Execute → Run***

V. UNDERSTANDING OF THE PROGRAM STRUCTURE

Let's understand various parts of our above program

1) #include <stdio.h>

The first line of the program ***#include <stdio.h>*** is a preprocessor command which is used to include a library containing functions used in our program, It tells a C compiler to include **stdio.h** file before going to actual compilation.

2) Main()

The C program starting point is identified by **main().** This informs the computer where the program actually starts. Two empty parentheses follow the function name. Sometimes, items may be in these parentheses, which we will cover later.

3) { & }

All functions in C have their contents encased by curly braces. The two braces **{** and **}** signify the begin and the end segments of the program. In general, braces are used throughout C to enclose a block of statement to be treated as a unit.

4) Printf()

printf(...) is another function available in C which causes the message " This is my first experience in C!" to be displayed on the screen. For printf to work, the header file "stdio.h" must be included at the beginning of the program.

5) \n

An interesting part of the text string is \n. Together, the backslash **(\)** and **n** characters make up an **escape sequence**. This particular escape sequence (\n) tells the program to add a new line. Other example of escape sequence are: **\t** (*Moves the cursor to the next tab*), **\r**(*Moves the cursor to the beginning of the current line*), \\(*Inserts a backslash*), **"(** *Inserts a double quote*), \'(*Inserts a single quote*).

6) /*… */ or //

They are use to insert comments into a C program. Comments serve for internal documentation for program structure and functionalities. There are two types of comment: The **single line comment** (by using //) and by **multiple line command** (by using /*….*/)

7) Return 0;

return 0; terminates main() function and returns the value 0.

BEWARE

– The success of compilation does not assure in any case the good result of the execution. Errors can still happen during the execution. This type of error is called **runtime error** and will be treated further.
– C is **case sensitive** (**main ≠ Main**). All commands in C must be in lowercase.
– C has a **free-form line structure**. Multiple statements can be on the same line. White space is ignored. Statements can continue over many lines.
– In C program, the **semicolon** is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are two different statements:

```
printf("Hello, World! \n"); return 0;
```

EXERCISE 1

Exercise 1.1: Write a C program that print the following

```
  *
 ***
*****
```

Exercise 1.2: Write a C-program to print the following sentence:

The character \ name's "backslash"

Exercise 1.3: Using an appropriate escape sequence, write a C program that display a sample calendar month as the following: **[See Activity 1.2]**

```
-------------------------------------------------------------
 SunMon     Tue WedThu Fri  Sat
 ------------------------------------------------------------
            1    2    3    4    5    6
 7          8    9    10   11   12   13
 14         15   16   17   18   19   20
 21         22   23   24   25   26   27
  28        29   30   31
```

Exercise 1.4: Write a C program that prints on the screen the words "***Now is the time for all good men to come to the aid of their country***"

– all on one line;
– on three lines;
– on two lines inside a box composed of * characters.

TOPIC 2

C CONSTANTS AND VARIABLES

A C program consists of various tokens and a token is either a **keyword**, an **identifier**, **a constant**, a **string literal**, or a **symbol.**

Objectives: at the end of this topic, student should be able to

- Identify C language keywords
- Identify and define a variable or a constant
- Use formatted input/output (scanf and printf) and escape sequence

Contents

I. KEYWORD

[See Activity 2.1]

Keywords are reserved words which have standard, predefined meaning in C. They cannot be used as program-defined identifiers.

auto	**default**	**float**	**register**	**struct**	**volatile**
break	**do**	**for**	**return**	**switch**	**while**
case	**double**	**goto**	**short**	**typedef**	**_packed**
char	**else**	**if**	**signed**	**union**	**printf**
const	**enum**	**int**	**sizeof**	**unsigned**	**scanf**
continue	**extern**	**long**	**static**	**void**	**FILE**

II. IDENTIFIERS

Identifiers are the names given to various program elements such as constants, **variables**, **function names** and **arrays** etc. Let us study first the rules to define names or identifiers.

Identifiers are defined according to the following rules:

- It consists of letters and digits.
- First character must be an alphabet or underscore.
- Both upper and lower cases are allowed. Same text of different case is not equivalent, for example: TEXT is not same as text.
- Except the special character underscore (_), no other special symbols can be used.

For example, some valid identifiers are: ***nb1, cons, h456787***, ***tax_rate, _XI, FOR***…

For example, some invalid identifiers are shown below: **[See Activity 2.1]**

• **123**	First character to be alphabet.
• **"X."**	Not allowed.
• **order-no**	Hyphen not allowed.
• **error flag**	Blankspace allowed.
• **auto**	keyword

Note: Generally all keywords are in lower case although uppercase of same names can be used as identifiers.

III. DATA TYPE

In the C programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.
The types in C can be classified as follows:

S.N.	Types and Description
1	**Basic Types:** They are arithmetic types and consists of the two types: (a) integer types and (b) floatingpoint types.
2	**Enumerated types:** They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.
3	**The type void:** The type specifier *void* indicates that no value is available.
4	**Derived types:** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

III.1 Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges:

Type	Storage size	Value range
Char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
Int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
Short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
Long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions **sizeof(type)** yields the storage size of the object or type in bytes. Following is an example to get the size of **int** type on any machine:

```
#include <stdio.h>
int main()
{
printf("Storage size for int : %d \n", sizeof(int));
return 0;
}
```

III.2 Floating- Point Types

Following table gives you details about standard floating-point types with storage sizes and value ranges and their precision:

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. Following example will print storage space taken by a float type and its range values:

```
#include <stdio.h>
#include <float.h>
int main()
{
        printf("Storage size for float : %d \n", sizeof(float));
        printf("Minimum float positive value: %E\n", FLT_MIN );
        printf("Maximum float positive value: %E\n", FLT_MAX );
        printf("Precision value: %d\n", FLT_DIG );
        return 0;
}
```

IV. C VARIABLES

IV.1 Definition of a variable

A **variable** is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in previous chapter, there will be the following basic variable types:

Type	**Description**
Char	Typically a single octet(one byte). This is an integer type.
Int	The most natural size of integer for the machine.
Float	A single-precision floating point value.
Double	A double-precision floating point value.
Void	Represents the absence of type.

IV.2 Variable declaration **[See Activity 2.4]**

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows
The syntax for declaring variables is as follows:

data- type *variable_list;*

variable_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int  i, j, k;
char  c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** both declares and defines the variables **i, j** and **k**; which instructs the compiler to create variables named **i, j** and **k** of type **int**.

Variables can be **initialized** (*assigned an initial value*) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
data-type variable_name = value;
```

Some examples are:

```
 int d = 3, f = 5;  // declaration of d and f.
int d = 3, f = 5;  // definition and initializing d and f.
byte z = 22;  // definition and initializes z.
char x = 'x';  // the variable x has the value 'x'.
```

IV.3 Lvalues and Rvalues in C

There are two kinds of expressions in C:

1. **lvalue**: An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
2. **rvalue**: An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. **Numeric literals are rvalues** and so may not be assigned and cannot appear on the left-hand side.

Following is a valid statement: ***int g = 20;***	But following is not a valid statement and would generate compile-time error: ***10 = 20;***

V. C CONSTANTS AND LITERALS

[See Activity 2.3]

V.1 Defining Constants

There are two simple ways in C to define constants:

1. Using **#define** preprocessor.
2. Using **const** keyword.

a) The #define Preprocessor

Following is the form to use #define preprocessor to define a constant:

```
#define: identifier value
```

Following example explains it in detail:

```
#include<stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main()
{
    int area;
    area = LENGTH * WIDTH;
    printf("value of area : %d", area);    printf("%c",  NEWLINE);
    return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of area : 50
```

b) The const Keyword

You can use **const** prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Following example explains it in detail:

```
#include<stdio.h>
int main()
```

```
{
    const int  LENGTH =10;
    const int  WIDTH  =5;
    const char NEWLINE ='\n';
    int area;
    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);
    return0;
}
```

When the above code is executed, it produces the following result: **value of area : 50**

Note that it is a good programming practice to define constants in CAPITALS.

V.2 Character constants

[See Activity 1.3]

Character literals are enclosed in **single quotes, e.g., 'x'** and can be stored in a simple variable of **char** type. A character literal can be a plain character (**e.g., 'x'**), an escape sequence (**e.g., '\t'),** or a universal character **(e.g., '\u02C0').**

There are certain characters in C when they are preceded by a backslash they will have special meaning and they are used to represent like **newline** (**\n**) or **tab** (**\t**). Here, you have a list of some of such escape sequence codes:

Escape sequence	**Meaning**	**Escape sequence**	**Meaning**
\\	\ character	\n	Newline
\'	' character	\r	Carriage return
\"	" character	\t	Horizontal tab
\?	? character	\v	Vertical tab
\a	Alert or bell	\ooo	Octal number of one to three digits
\b	Backspace	\xhh . . .	Hexadecimal number of one or more digits
\f	Form feed		

Following is the example to show few escape sequence characters:

```
#include<stdio.h>
int main()
{
  printf("Hello\tWorld\n\n");
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello   World
```

VI. INPUT AND OUTPUT

VI.1 Formatted input — scanf

Let's consider the following statement: ***scanf("%f", &a);***

scanf is a function in C which allows the programmer to accept input from a keyboard. It obtains a value from the user

This **scanf** statement has two arguments

- **%f -** indicates data should be a floating number
- **&a -** location in memory to store variable

& is confusing in beginning – for now, just remember to include it with the variable name in ***scanf*** statements.

When executing the program the user responds to the **scanf** statement by typing in a number, then pressing the Enter (return) key

VI.2 Formatted output — printf

[See Activity 2.5]

printf("The new value of b is %f\n", b);

In the program statement above:

"The new value of b is %f\n" is the control string
b is the variable to be printed

scanf and ***printf*** use the same conversion characters as .

The arguments to **scanf** must be **pointers** (*addresses*), hence the need for the **&** character above. The following table show what format specifies should be used with what data types:

%c character %d decimal integer %x hexadecimal integer %o octal integer %ld long integer	%u unsigned %i integer %e or %f or %g floating point number %s string pointer to char %lf long double

NB: When reading integers using the "**I**" conversion character, the data entered may be preceded by **0** or **0x** to indicate that the data is in octal (base 8) or hexadecimal (base16).

VI.3 Characters' Input/Output

```
#include <stdio.h>
int main()
{
char me[20];
printf("What is your name?");
scanf("%s",&me);
printf("I arn glad to meet you, %s!\n",me,);
return(0);
}
```

```
int main()
{
int  c;
c = getchar(); /* read a character and assign to c */
putchar(c); /* print c on the screen */
return 0;
        }
```

getchar and **putchar** are used for the input and output of *single characters* respectively.

- **getchar()** returns an int which is either **EOF**(*indicating end-of-file, see later*) or the next character in the standard input stream
- **putchar(c)** puts the character c on the standard output stream.

EXERCICE 2

Exercise 2.1: Identify keywords and valid identifiers among the following:

hello   function   day-of-the-week   student_1   max_value   "what"   1_student   int   union

TOPIC 3

OPERATORS IN C

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C language is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Objectives:
This tutorial will explain the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Contents

I. ARITHMETIC OPERATORS

[See Activity 3.1] [See Activity 3.6]

Following table shows all the arithmetic operators supported by C language. Assume variable A holds 15 and variable B holds 20, then:

Operator	Description	Example
+	Adds two operands	A + B will give 35
-	Subtracts second operand from the first	A - B will give -5
*	Multiplies both operands	A * B will give 300
/	Divides numerator by de-numerator	B / A will give 1.5

%	Modulus Operator and remainder of after an integer division	B % A will give 5 A % B will give 15
++	Increments operator increases integer value by one	A++ will give 16
--	Decrements operator decreases integer value by one	A-- will give 14

Try the following example to understand all the arithmetic operators available in C programming language:

```
#include<stdio.h>
int main()
{
        int a =21; int  b=10; int c ;
        c = a + b; printf("Line 1 - Value of c is %d\n", c );
        c = a-b; printf("Line 2 - Value of c is %d\n", c );
        c = a * b; printf("Line 3 - Value of c is %d\n", c );
        c = a / b; printf("Line 4 - Value of c is %d\n", c );
        c = a % b; printf("Line 5 - Value of c is %d\n", c );
        c = a++; printf("Line 6 - Value of c is %d\n", c );
        c = a--; printf("Line 7 - Value of c is %d\n", c );
        return 0;
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

II. RELATIONAL OPERATORS

Following table shows all the relational operators supported by C language. Assume variable A holds 10 and variable B holds 20, then:

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

III. LOGICAL OPERATORS

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is nonzero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Try the following example to understand all the logical operators available in C programming language:

```
#include<stdio.h>
int main()
{
     int a =5;int b =20;int c ;
     if( a && b )
          printf("Line 1 - Condition is true\n");
     if( a || b )
          printf("Line 2 - Condition is true\n");
           /* lets change the value of  a and b */
      a =0;    b =10;
      if( a && b )
           printf("Line 3 - Condition is true\n");
```

```
        else
                printf("Line 3 - Condition is not true\n");
        if(!(a && b))
             printf("Line 4 - Condition is true\n");
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

IV. BITWISE OPERATORS

[See Activity 3.2]

Bitwise operator works on bits and performs bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

P	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

```
A  = 0011 1100
B  = 0000 1101
      -----------------
      A&B = 0000 1100   A|B = 0011 1101   A^B = 0011 0001 ~A  = 1100 0011
```

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	**Description**	**Example**
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101

^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60, which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Try the following example to understand all the bitwise operators available in C programming language:

```
#include<stdio.h>
 main()
{
      unsigned int a =60;      /* 60 = 0011 1100 */
      unsigned int b =13;       /* 13 = 0000 1101 */
      int c =0;
      c = a & b;                    /* 12 = 0000 1100 */
      printf("Line 1 - Value of c is %d\n", c );
      c = a | b;                    /* 61 = 0011 1101 */
      printf("Line 2 - Value of c is %d\n", c );
      c = a ^ b;                    /* 49 = 0011 0001 */
      printf("Line 3 - Value of c is %d\n", c );
      c =~a;                       /*-61 = 1100 0011 */
      printf("Line 4 - Value of c is %d\n", c );
      c = a <<2;                   /* 240 = 1111 0000 */
      printf("Line 5 - Value of c is %d\n", c );
      c = a >>2;                 /* 15 = 0000 1111 */
      printf("Line 6 - Value of c is %d\n", c );
       return 0;
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

V. ASSIGNMENT OPERATORS

[See Activity 3.3]

There are following assignment operators supported by C language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operar and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators ↦ sizeof & ternary

There are few other important operators including **sizeof** and **? :** supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.

? :	Conditional Expression	If Condition is true? Then value X : Otherwise value Y

VI. OPERATORS PRECEDENCE IN C

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 * 2**; here, *x is assigned 13, not 20 because operator * has higher precedence than* +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Try the following example to understand the operator precedence available in C programming language:

```
#include<stdio.h>
```

```
 main()
{
     int a =20;int b =10;int c =15;int   d =5;int e;
     e =(a + b)* c / d;// ( 30 * 15 ) / 5
     printf("Value of (a + b) * c / d is : %d\n",  e );
     e =((a + b)* c)/ d;// (30 * 15 ) / 5
     printf("Value of ((a + b) * c) / d is  : %d\n",  e );
     e =(a + b)*(c / d);// (30) * (15/5)
     printf("Value of (a + b) * (c / d) is  : %d\n",  e );
     e = a +(b * c)/ d;//  20 + (150/5)
     printf("Value of a + (b * c) / d is  : %d\n",  e );
     return0;
}
```

When you compile and execute the above program, it produces the following result:

```
Value of (a + b) * c / d is : 90

Value of ((a + b) * c) / d is  : 90

Value of (a + b) * (c / d) is  : 90

Value of a + (b * c) / d is  : 50
```

[See Activity 3.4]

EXERCISE 3

Exercise 3.1: Starting from the following declarations

```
int a=2,b=3,c;
float x=5.0,y;
```

Predict the outcome of the operations

```
y=a*b;
c=a*b;
y=a/b;
c=a/b;
y=a/b*x;
c=a/b*x;
y=a*x/b;
c=a*x/b;
```

Write the corresponding C code to check your predictions. You will use the printf command (<stdio.h>) to output the results on the screen. Note that you will have to use %i to print an integer and %f to print a float.

Exercise 3.2: Given a time in seconds (integer), print to the screen the corresponding time in hours, minutes and seconds. The output will be formatted like: "***XXXX seconds is equivalent to XX hours, XX minutes and XX seconds***".

Exercise 3.3: Write a C code that switches the values of two variables A and B and prints the result on the screen. How many variables do you need?

Exercise 3.4: Write a program that reads in an integer and prints out the given integer in decimal, octal and hexadecimal formats

Exercise 3.5: Write a program that reads in a temperature expressed in Celsius (Centigrade) and displays the equivalent temperature in degrees Fahrenheit.

Exercise 3.6: Create a new program that prompts a user for numbers and determines total revenue using the following formula: **Total Revenue = Price * Quantity**.

Exercise 3.7: Build a new program that prompts a user for data and determines a commission using the following formula: **Commission = Rate * (Sales Price – Cost)**.

Exercise 3.8: a) Given ***a = 5, b = 1, x = 10***, and ***y = 5,*** create a program that outputs the result of the formula ***f = (a   b)(x   y)*** using a single ***printf()*** function.
b) Create a program that uses the same formula above to output the result; this time, however, prompt the user for the values ***a, b, x,*** and ***y***. Use appropriate variable names and naming conventions.

Exercise 3.9: Create a program that prompts a user for her name. Store the user's name using the **scanf()** function and return a greeting back to the user using her name.

Exercise 3.10: Write a program to read in a three digit number and produce output like

```
3 hundreds
4 tens
7   units
```

For an input of 347. There are two ways of doing this. Can you think of both of them? Which do you think is the better?

TOPIC 4

CONDITIONS(Selection)

A program is usually not limited to a linear sequence of instruction. In real life, a program usually needs to change the sequence of execution according to some conditions. In C language, there are many control structure that are used to handle conditions and the resultant decisions. This lesson introduces if-else and switch construct.

Objectives: At the end of this lesson, student should be able to:

- Manipulate selection by the use of if - else statement
- Replace nested if statement by an appropriate switch – case statement
- Use ternary operation in selection

Contents_Toc400143325

Decision making structures require that the programmer specify one or more **conditions** to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
C programming language provides following types of decision making statements.

I. THE IF STATEMENT

An **if** statement consists of a Boolean expression followed by one or more statements

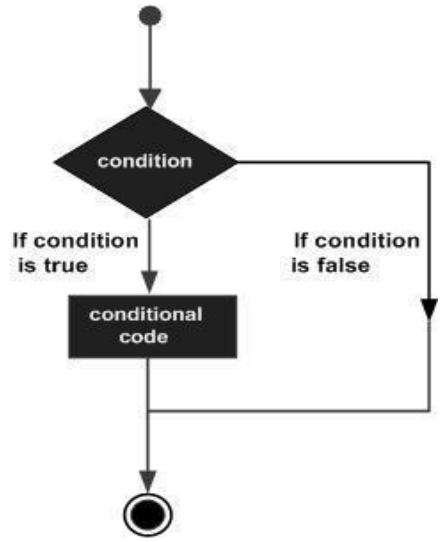
I.1 A simple if statement

Syntax: The syntax of an if statement in C programming language is:

```
if(boolean_expression)
{
   statements
} /* statement(s) will execute if the Boolean expression is true */
```

If the **Boolean** expression evaluates to **true,** then the block of code inside the **if** statement will be executed. If **Boolean** expression evaluates to **false,** then the first set of code after the end of the **if** statement (after the closing curly brace) will be executed.

Flow Diagram [See Activity 4.1]



C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

Example

```
#include<stdio.h>
int main ()
{
    int a =10; /* local variable definition */
    if( a <20) /* check the boolean condition using if statement */
    {
      printf("a is less than 20\n");/* if condition is true then print the following */
    }
    printf("value of a is : %d\n", a);
    return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20; value of a is : 10
```

I.2 The if-else statement

[See Activity 4.2] [See Activity 4.3]

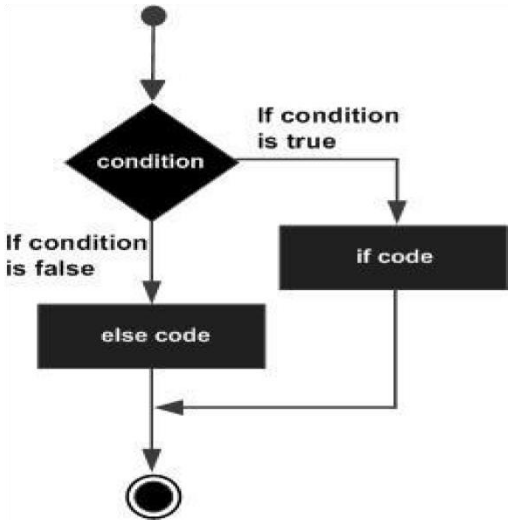
An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is **false**.

Syntax : The syntax of an **if...else** statement in C programming language is:

```
if(boolean_expression)
{
      statements_if_expression_true;
}/* statement(s) will execute if the boolean expression is true */
else
{
    statements_if_expression_false;
}/* statement(s) will execute if the boolean expression is false */
```

If the **boolean** expression evaluates to **true,** then the if block of code will be executed, otherwise **else** block of code will be executed.

Flow Diagram



Example

```
#include<stdio.h>
int main ()
{
       int a =100; /* local variable definition */
       if( a <20) /* check the boolean condition */
       {
          printf("a is less than 20\n");/* if condition is true then print the following */
       }
       else
```

```
        {
           printf("a is not less than 20\n");/* if condition is false then print the following */
        }
        printf("value of a is : %d\n", a);
        return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20; value of a is : 100
```

I.3 The if-else-if ladder **[See Activity 4.4]**

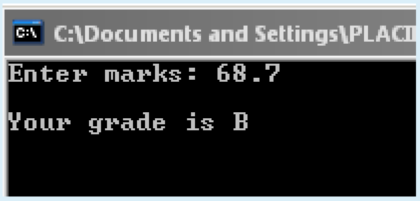
Let's consider a program displaying the student's grade based on the following table:

Marks	Grade
>=75	A
>=50 and < 75	B
>=25 and <50	C
<25	F

In this case multiple conditions are to be checked. Marks obtained by a student can only be one of the ranges. Therefore, ***if-else-if can*** be used to implement the following program.

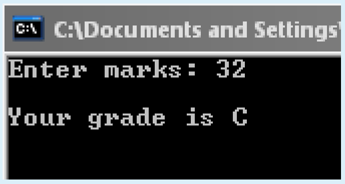
```
/* Distribution of grade */
#include<stdio.h>
main()
{
      float marks;
      printf("Enter marks: ");
      scanf("%f",&marks);
      if (marks >= 75)
            printf("\nYour grade is A");
      else if (marks >=50)
            printf("\nYour grade is B");
      else if (marks >=25)
            printf("\nYour grade is C");
      else if (marks < 25)
            printf("\nYour grade is F");
      return 0;
}
```

This is an example of execution screen



```
Enter marks: 68.7

Your grade is B
```



```
Enter marks: 32

Your grade is C
```

The general syntax of an **if-else-if** ladder is

```
if(boolean_expression 1)
{
  /* Executes when the boolean expression 1 is true */
}
elseif( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
elseif( boolean_expression 3)
{
  /* Executes when the boolean expression 3 is true */
}
else
{
   /* executes when the none of the above condition is true */
}
```

I.4 The nested if statement [See Activity 4.5]

It is always legal in C programming to **nest if-else** statements, which means you can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Syntax : The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2)
   {
     /* Executes when the boolean expression 2 is true */
   }
}
```

You can nest else **if...else** in the similar way as you have **nested if** statement.

Example

```
#include<stdio.h>
int main ()
{
int a =100;int b =200; /* local variable definition */
     if( a ==100) /* check the boolean condition */
     {
         if( b ==200) /* if condition is true then check the following */
         {
            printf("Value of a is 100 and b is 200\n");
         }
     }
     printf("Exact value of a is : %d\n", a );
     printf("Exact value of b is : %d\n", b );
     return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

II. CONDITIONAL SELECTION — SWITCH

[See Activity 4.6] [See Activity 4.7]

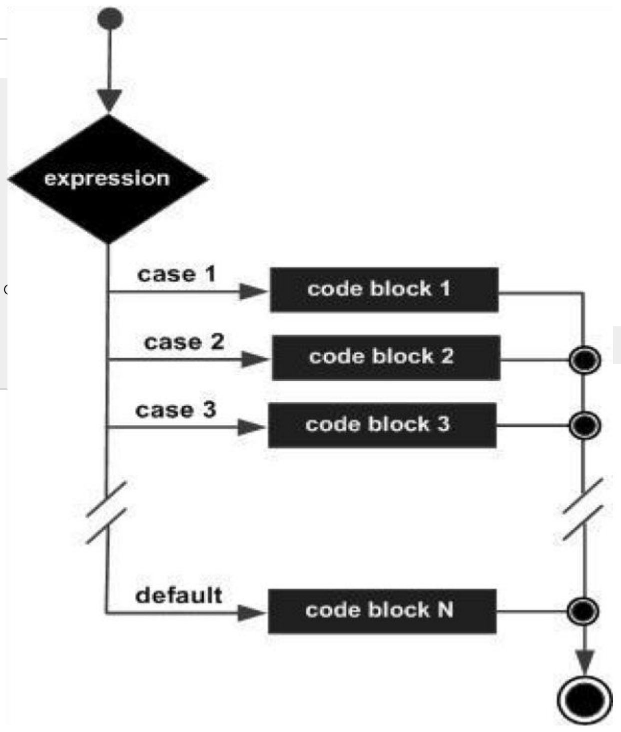
A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case. The syntax is

Syntax: The syntax for a switch statement in C programming language is as follows:

```
switch(expression){
    case constant-expression  :
       statement(s);
       break; /* optional */
    case constant-expression  :
       statement(s);
       break; /* optional */

    /* you can have any number
    default : /* Optional */
       statement(s);
}
```

Flow Diagram



Example

```
#include<stdio.h>
int main ()
{
  char grade ='B'; /* local variable definition */
  switch(grade)
  {

    case'A':
       printf("Excellent!\n");
       break;
    case'B':
    case'C':
       printf("Well done\n");
       break;
    case'D':
       printf("You passed\n");
       break;
    case'F':
       printf("Better try again\n");
       break;
    default:
      printf("Invalid grade\n");
  }
  printf("Your grade is  %c\n", grade );
  return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Well done
Your grade is B
```

III. THE ? : OPERATOR

[See Activity 4.8]

The conditional operator **?** can be used to replace **if...else** statements. It has the following general form:

```
Exp1? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is **true**, then Exp2 is evaluated and becomes the value of the entire **?** expression. If Exp1 is **false**, then Exp3 is evaluated and its value becomes the value of the expression.

EXERCISE 4

Exercise 4.1: Write a program which verifies whether a number entered is odd or even.

Exercise 4.2: Write a C-program to calculate the absolute value of a number

Exercise 4.3: Write a C code that returns the sign of a multiplication of A and B without doing the multiplication.

Exercise 4.4: Write a program which consists to say whether a student has passed a test or he has failed, being given an average. The student has passed if the average is greater than 12.00

Exercise 4.5: What gets printed?

```
#include <stdio.h>
int main()
{
    int i, j;
    i = j = 2;
    if (i == 1)
        if (j == 2)
    printf("%d\n", i = i + j);
    else
        printf("%d\n", i = i - j);
    printf("\n%d", i);
    return 0;
}
```

Exercise 4.6: A car increases it velocity from u ms^{-1} to v ms^{-1} within t seconds. Write a program to calculate the acceleration.

Exercise 4.7: Write a program that solve an equation of second degree (in the form $\boldsymbol{ax^2 + bx + c = 0}$). The program receive the three coefficients a, b and c. (*use the function sqrt() in the library math.h to calculate the square root of a number*)

a) Display either the real solution(s) or a message when the equation has no solution in the set $\mathbb{R}$.
b) When there is not a real solution, display the complex solutions in form of $\boldsymbol{x \pm yi}$

Exercise 4.8: A year is a leap year if it is divisible by 4 unless it is a century year (one that ends in 00) in which case it has to be divisible by 400. Write a program to read in a year and report whether it is a leap year or not.

Exercise 4.9: Rewrite the following fragment using switch:

```
if (ch == 'A')
   a_grade++;
else if (ch == 'B')
   b_grade++;
else if (ch == 'C')
   c_grade++;
else if (ch == 'D')
   d_grade++;
else
f_grade++;
```

Exercise 4.10: Consider the following code fragment:

```
int line = 0;
char ch;
while (cin.get(ch))
{
   if (ch == 'Q')
        break;
   if (ch != '\n')
        continue;
   line++;
}
```

Rewrite this code without using break or continue.

Exercise 4.11: Develop a simple calculator to accept two floating point numbers from the keyboard. Then display a menu to a user and let him/her select a mathematical operation to be performed on those two numbers. Then display the answer. A sample run of your program should be similar to the following:

```
Enter number 1: 20
Enter number 2: 12
       Mathematical Operation
-----------------------------------
1 - Add
2 - Subtract
3 - Multiply
4 - Divide
-----------------------------------
Enter your preference: 2

Answer : 8.00
```

Challenges

Exercise 4.12: Build a number guessing game that uses input validation (**isdigit()** function) to verify that the user has entered a digit and not a non-digit (letter). Store a random number between 1 and 10 into a variable each time the program is run. Prompt the user to guess a number between 1 and 10 and alert the user if he was correct or not.

Exercise 4.13: Build a Fortune Cookie program that uses either the Chinese Zodiac or astrological signs to generate a fortune, a prediction, or a horoscope based on the user's input. More specifically, the user may need to input her year of birth, month of birth, and day of birth depending on zodiac or astrological techniques used. With this information, generate a custom message or fortune. You can use the Internet to find more information on the Chinese Zodiac or astrology.

Exercise 4.14: Create a dice game that uses two six-sided dice. Each time the program runs, use random numbers to assign values to each die variable. Output a "player wins" message to the user if the sum of the two dice is 7 or 11. Otherwise output the sum of the two dice and thank te user for playing.

TOPIC 5

CONTROL STRUCTURES (Looping)

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The **first** statement in a function is executed first, followed by the **second**, and **so on**.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages

Objectives: At the end of this lesson, student should have mastered the following topics:

- ***Increment and decrement operator***
- ***For, while and do-while loops***
- ***Use of break and continue statements***
- ***Use of goto and label statements***

Contents_Toc400147248

I. FOR LOOP IN C

[See Activity 5.3]

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

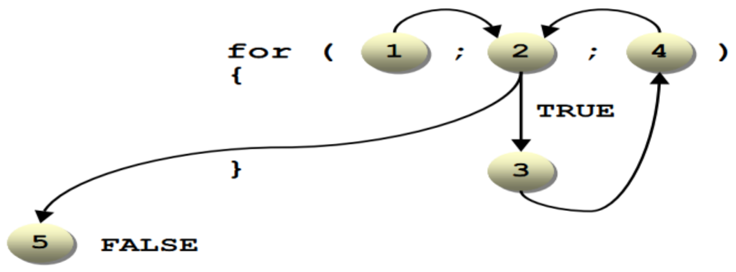
Syntax: The syntax of a “for loop” in C programming language is:

```
for( init; condition; increment )
{
   statement(s);
}
```

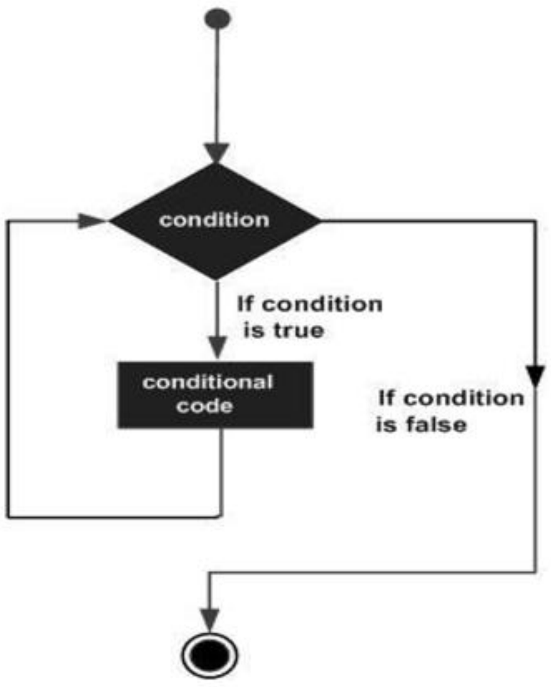
The operation for the loop is as follows

1) The initialization expression is evaluated.
2) The test expression is evaluated. If it is TRUE, body of the loop is executed. If it is FALSE, exit the for loop.
3) Assume test expression is TRUE. Execute the program statements making up the body of the loop.
4) Evaluate the increment expression and return to step 2.
5) When test expression is FALSE, exit loop and move on to next line of code.

The following diagram illustrates the operation of a for loop



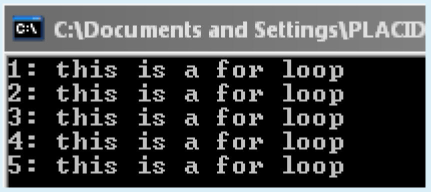
Flow Diagram



Consider the following example:

```
#include<stdio.h>
main()
{
        int counter;
        for(counter=1;counter<=5;counter++)
                printf("%d: this is a for loop\n", counter);
        return 0;
}
```

The program above will produce the following output.



```
1: this is a for loop
2: this is a for loop
3: this is a for loop
4: this is a for loop
5: this is a for loop
```

It is also possible to decrement the counter depending on the requirement, but one has to use suitable control expression and an initial value.

Exercise:

a) Write a program to display all the integers from 100 to 200.
b) A program consist to calculate the sum of all the even numbers up to 100
c) Modify the program above so that it computes the sum of all the odd number up to 100.

II. WHILE LOOP IN C

[See Activity 5.1]

A **while** loop statement in C programming language repeatedly executes a target statement as long as a given condition is **true**.

The while loop checks whether the test expression is true or not. If it is true, code/s inside the body of while loop is executed, is, code/s inside the braces { } are executed. Then again the test expression is checked whether test expression is true or not. This process continues until the test expression becomes false.

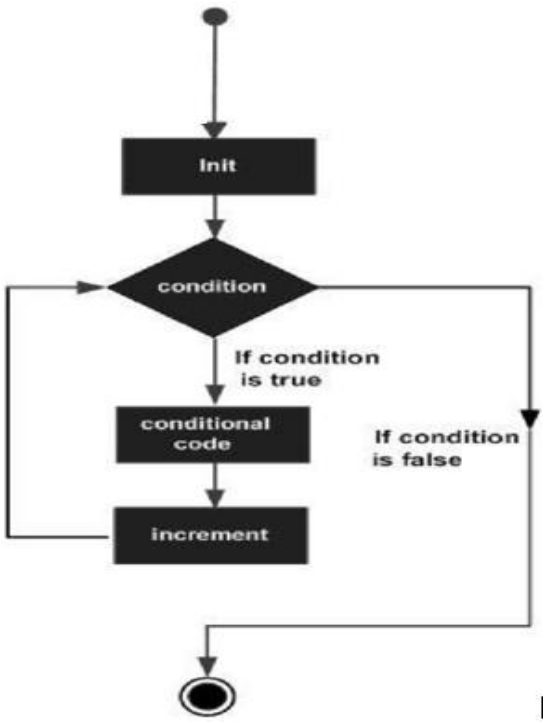
Syntax: The syntax of a while loop in C programming language is:

```
while(condition)
{
  statement(s);
}
```

Here, statement(s) may be a single statement or a block of statements. The **condition** may be any expression, and **true** is any **nonzero** value. The loop iterates while the condition is **true**.

When the condition becomes **false**, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is **false**, the loop body will be skipped and the first statement after the while loop will be executed.

Programmer is responsible for **initialization** and **incrementation**. At some point in the body of the loop, the control expression must be altered in order to allow the loop to finish. Otherwise: ***infinite loop***.

A for loop can be transformed into a while loop using the following rule:

for*(expr1; expr2; expr3)*
{
    statement
}

⇒

expr1;
while*(expr2)*
{
    statement
    expr3
} **[See Activity 5.2]**

The program above consisting to calculate the sum of all the even numbers up to 100 can be rewritten using the while loop as follow:

```
#include<stdio.h>
main()
{
     int i, sum;
     sum = 0;
     i = 0;
```

```
            while(i <= 100)
            {
                sum += i;
                i += 2;
            }
            printf("total : %d", sum);
            return 0;
        }
```

III. DO...WHILE LOOP IN C

[See Activity 5.4] [See Activity 5.8]

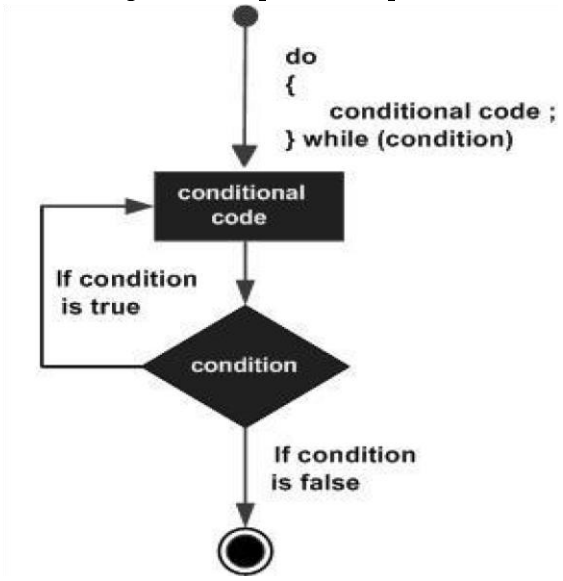
Unlike for and while loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a **do...while** loop is guaranteed to execute at least one time.

Syntax: The syntax of a do...while loop in C programming language is:

```
do
{
  statement(s);
}
while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.
If the condition is **true**, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes **false**.



Flow Diagram

Example

```
#include<stdio.h>
int main ()
{
    int a =10;
   do{
      printf("value of a: %d\n", a);
      a = a +1;
    }while( a <20);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

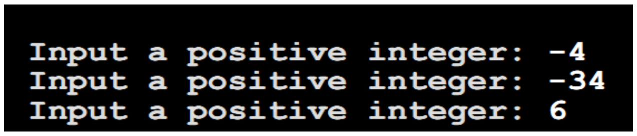
```
value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 15 value of a:
16 value of a: 17 value of a: 18 value of a: 19
```

A common use of the do while statement is input error checking. A simple form is shown here

```
do {
      printf("\n Input a positive integer: ");
     scanf("%d",&n);
} while (n<=0);
```

[See Activity 5.8]

The user will remain in this loop continually being prompted for and entering integers until a positive one is entered. A sample session using this loop looks like this



```
Input a positive integer: -4
Input a positive integer: -34
Input a positive integer: 6
```

IV. NESTED LOOPS IN C

[See Activity 5.5]

C programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

The syntax for a nested for loop statement in C is as follows:

```
for( init; condition; increment )
{
  for( init; condition; increment )
  {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested** while loop statement in C programming language is as follows:

```
while(condition)
{
  while(condition)
  {
    statement(s);
  {
  Statements
}
```

The syntax for a **nested do...while** loop statement in C programming language is as follows:

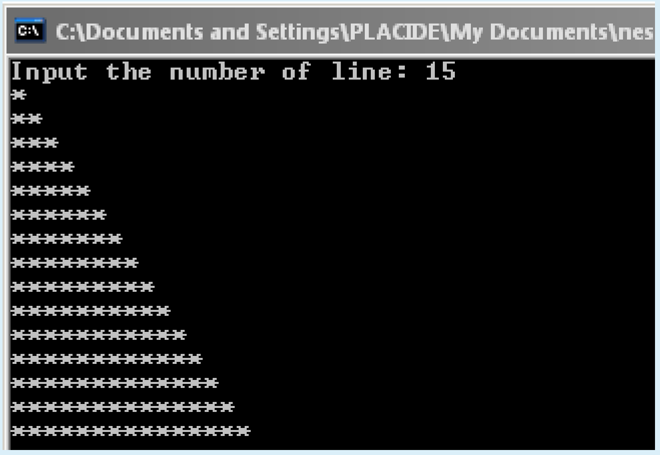
```
do{
  statement(s);
  do{
    statement(s);}
  while( condition );
}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a **for loop** can be inside a while loop or vice versa.

Example

```
#include<stdio.h>
main()
{
        int i,j,n;
        printf("Input the number of line: ");
        scanf("%d", &n);
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=i;j++)
            {
                printf("*");
            }
            printf("\n");
        }
        return 0:
}
```

The output of the program is as follow:



C:\Documents and Settings\PLACIDE\My Documents\nes

```
Input the number of line: 15
*
**
***
****
*****
******
*******
********
*********
**********
***********
************
*************
**************
***************
```

V. BREAK STATEMENT IN C

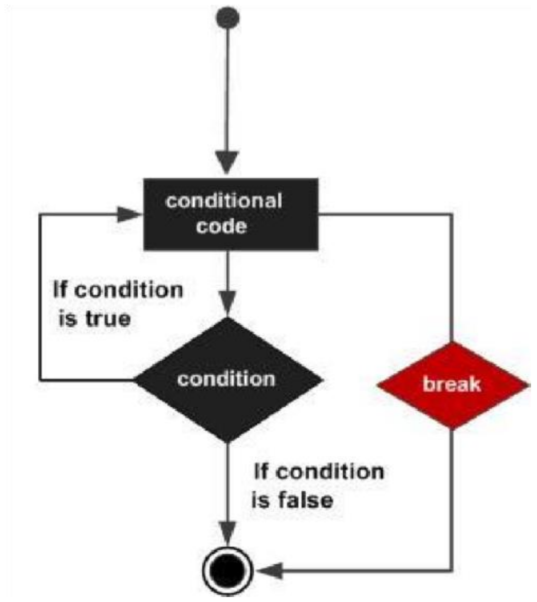
[See Activity 5.6]

The **break** statement in C programming language has the following two usages:

1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **switch** statement (covered in the next chapter).

If you are using **nested loops** (i.e., one loop inside another loop), the **break** statement will stop the execution of the innermost loop and start executing the next line of code after the block.

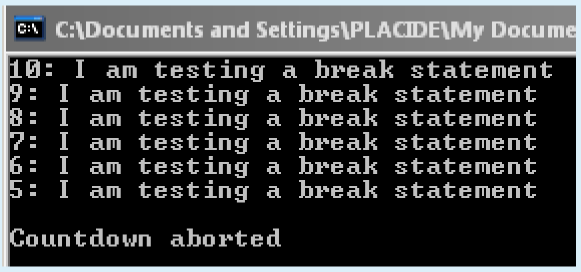
Flow Diagram



Example

```
#include<stdio.h>
main()
{
      int n;
      for(n=10;n>0;n--)
      {
         printf("%d: I am testing a break
statement\n",n);
         if(n==5)
         {
            printf("\nCountdown aborted");
            break;
         }
      }
      return 0;
}
```

This is the output of the program:



```
C:\Documents and Settings\PLACIDE\My Docume
10: I am testing a break statement
9: I am testing a break statement
8: I am testing a break statement
7: I am testing a break statement
6: I am testing a break statement
5: I am testing a break statement

Countdown aborted
```

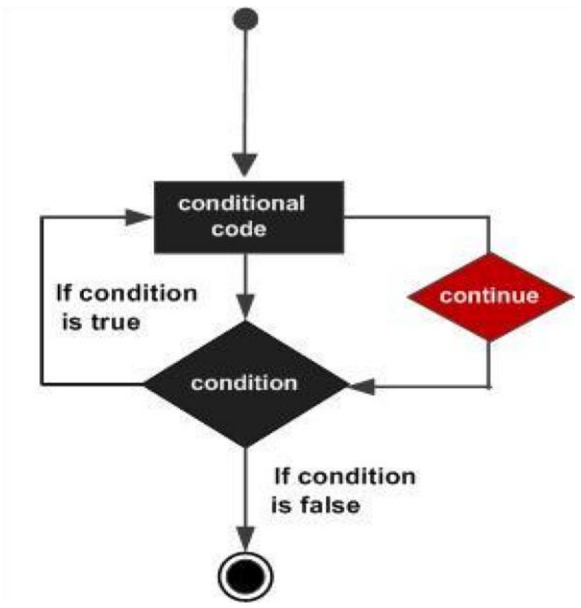
VI. CONTINUE STATEMENT IN C

[See Activity 5.6]

The **continue** statement in C programming language works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the "**for loop**", "**continue**" statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests.

Flow Diagram



Example

```
#include<stdio.h>
int main ()
{
      int a =10;
      do
      {
            if( a ==15)
            {
               a = a +1;
               continue; /* skip the iteration */
            }
            printf("value of a: %d\n", a);
            a++;
      }while( a <20);
      return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 16 value of
a: 17 value of a: 18 value of a: 19
```

VII. GOTO STATEMENT IN C

[See Activity 5.7]

A **goto** statement in C programming language provides an unconditional jump from the **goto** to a labeled statement in the same function.

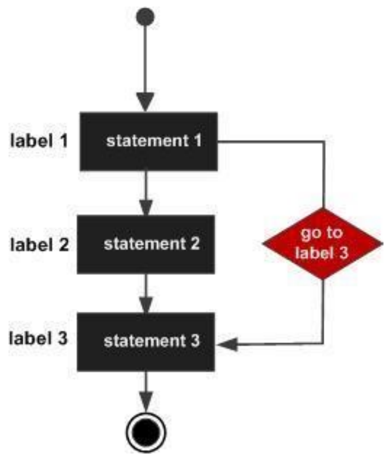
NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a **goto** can be rewritten so that it doesn't need the **goto**.

Syntax : The syntax for a goto statement in C is as follows:

```
goto label;
..
. label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram



Example

```
#include<stdio.h>
int main ()
{
    int a =10;
    LOOP:
    do /* do loop execution */
   {
        if( a ==15) /* skip the iteration */
       {
           a = a +1;
           goto LOOP;
        }
        printf("value of a: %d\n", a);       a++;
    }while( a <20);
    return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 16 value of
a: 17 value of a: 18 value of a: 19
```

VIII. THE INFINITE LOOP

[See Activity 5.9]

A loop becomes **infinite** loop if a condition never becomes **false**. The **for loop** is traditionally used for this purpose. Since **none** of the three expressions that form the **for loop** are required, you can make an endless loop by leaving the conditional expression empty.

```
#include<stdio.h>
int main ()
    {
      for(;;)
     {
         printf("This loop will run forever.\n");
    }
    return0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the **for(;;)** construct to signify an infinite loop.

NOTE: You can terminate an infinite loop by pressing **Ctrl + C** keys.

EXERCISE 5

Exercise 5.1: Write a program to compute the sum of all the integers from 1 to 100.

Exercise 5.2: Create a counting program that counts from 1 to 100 in increments of 5.

Exercise 5.3: Create a counting program that counts backward from 100 to 1in decrements of 10.

Exercise 5.4: Write the program to compute the sum of all integers between any given two numbers.

Exercise 5.5: Write the program to calculate the factorial of any given factorial number.

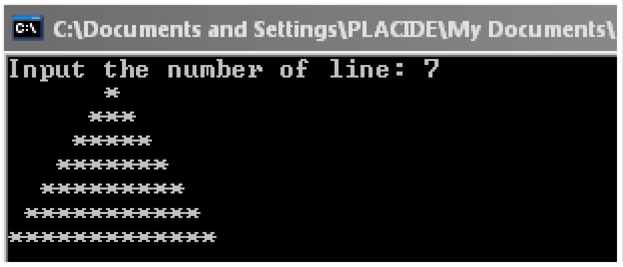
Exercise 5.6: Create a counting program that prompts the user for three inputs (shown next) that determine how and what to count. Store the user's answers in variables. Use the acquired data to build your counting program with a "for loop" and display the results to the user.

- Beginning number to start counting from
- Ending number to stop counting at
- Increment number

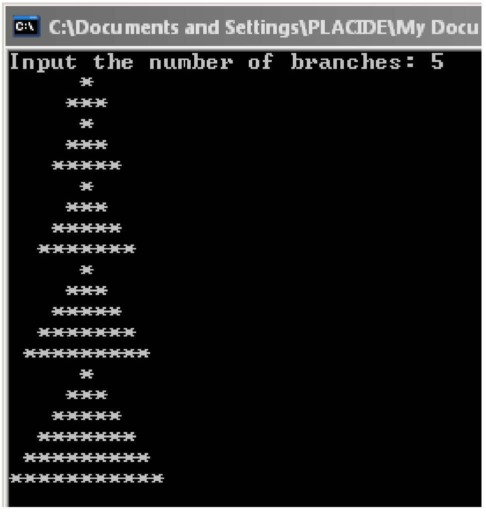
Exercise 5.7: Write a C code that permits the user to specify a number of lines or branches and then prints on

a) the shell a triangular pyramid consisting of stars (picture 1 below).

b) A picture of a Christmas tree consisting of star (figure 2 below).
Hints: all characters are printed separately and both the number of stars per line and the number of spaces per line must be calculated. The output will look like:



```
Input the number of line: 7
      *
     ***
    *****
   *******
  *********
 ***********
*************
```



```
Input the number of branches: 5
     *
    ***
     *
    ***
   *****
     *
    ***
   *****
  *******
     *
    ***
   *****
  *******
 *********
     *
    ***
   *****
  *******
 *********
***********
```

Exercise 5.8: Using a loop, ask the user to enter positive integer numbers. The program will output the number of values entered, the minimum value, the maximum value and the average of all numbers. The code will exit once a negative integer is entered.

Exercise 5.9: Write a program where the user supplies integer values between 1 and 9 and the program returns the sum, average and RMS of the values. The program will exit when 0 is entered. Values outside of the bounds will be discarded.

Exercise 5.10: Write a program that returns the number of years until a father will have an age double of its son's age.

Exercise 5.11: Write the C code that computes x^n using the three loop functions (for, while and do-while). We will consider n to be an integer.

Exercise 5.12: Write a program that tests if a positive integer is a prime number using a straightforward approach. You will assume that your number is smaller than 1000.

Exercise 5.13: Write a C-program to display the message "***God is love!***" 100000 times. The program should allow users to terminate the execution at any time by pressing any key before it display all the 100000 messages. The C function **kbhit()** can be used to check for a keystroke. If a key has been pressed, it returns the value "**1**" otherwise it returns "**0**". The **kbhit()** function is define in the header file **conio.h**

Exercise 5.14: a) Write a program to prompt the user for an integer and calculate the sum of all the integers up to and including the input value. Print out the result.

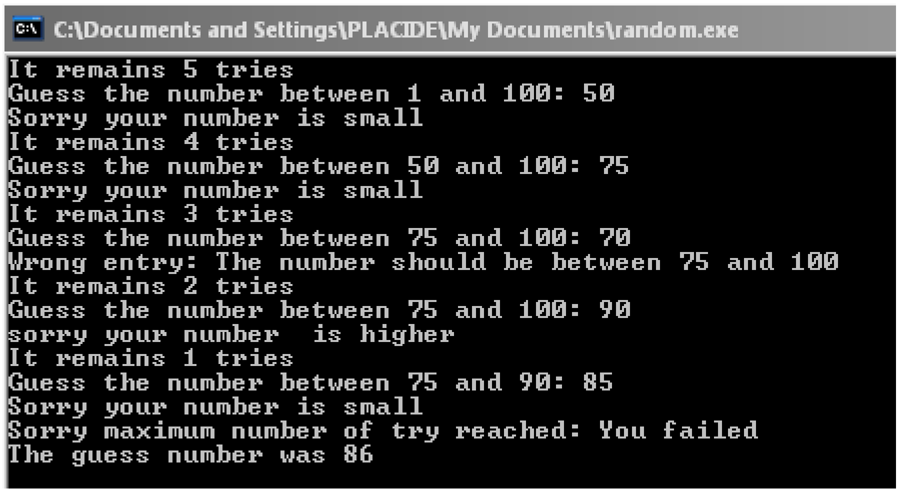
b) Modify the previous program to use floating point arithmetic to add up the reciprocals of all the integers up to and including the input value.

c) Further modify the previous program to print out a list of reciprocal sums for every integer up to and including the input value.

I.e. print out the sum of the reciprocals of the integers up to and including 1, up to and including 2, up to and including 3 etc., etc.

Exercise 5.15: A program consists to the following game: The machine guesses a number between a given intervals and ask to the student to find it. The max, the min and the number of tries should be defined as constant values (use ***#define***). For any number guess by the user, the machine should see whether the number is bigger or smaller than the searched number. The program stop when the user actually guess the right number, in which case the sentence "***Bravo you win in x tries***" (x to be given) is printed, or the maximum number of tries are reached, in which case the sentence "***Sorry exceeded number of tries, you failed***" is printed. The output will look like

(Use the rand() function in the library "stdlib.h" to generate a random number.)



```
It remains 5 tries
Guess the number between 1 and 100: 50
Sorry your number is small
It remains 4 tries
Guess the number between 50 and 100: 75
Sorry your number is small
It remains 3 tries
Guess the number between 75 and 100: 70
Wrong entry: The number should be between 75 and 100
It remains 2 tries
Guess the number between 75 and 100: 90
sorry your number  is higher
It remains 1 tries
Guess the number between 75 and 90: 85
Sorry your number is small
Sorry maximum number of try reached: You failed
The guess number was 86
```

TOPIC 6

MODULAR PROGRAMMING (Functions)

C and C++ come with a large library of useful functions (the standard ANSI C library plus several C++ classes), but real programming pleasure comes with writing your own functions. This chapter examines how to define functions, convey information to them, and retrieve information from them. After reviewing how functions work, this chapter concentrates on how to use functions this chapter touches on recursion to functions. If you've paid your C dues, you'll find much of this chapter familiar. But don't be lulled into a false sense of expertise. Meanwhile, let's attend to the fundamentals.

Objectives: Having read this section you should be able to:

- Write programs using correctly defined C functions
- Subdivide a big problems into many smaller problems and use function or procedure to facilitate the resolution
- pass the value of local variables into your C functions

Contents

I. INTRODUCTIVE TO FUNCTION (OR PROCEDURE)

[See Activity 6.1]

A **function in C** is a small "**sub-program**" that performs a particular task, and supports the concept of **modular programming** design techniques. In modular programming the various tasks that your overall program must accomplish are assigned to individual functions and the main program basically calls these functions in a certain order.

We have already been exposed to functions. The main body of a C program, identified by the keyword **main**, and enclosed by left and right braces is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system. We have also seen examples of **library functions** which can be used for I/O (***stdio.h***), like **printf** and **scanf**, mathematical tasks (math.h), and character/string handling (string.h).

There are many good reasons to program in a modular style:

- Don't have to ***repeat the same block of code*** many times in your code. Make that code block a function and call it when needed.
- ***Function portability***: useful functions can be used in a number of programs.
- Supports the ***top-down technique*** for devising a program algorithm. Make an outline and hierarchy of the steps needed to solve your problem and create a function for each step.
- ***Easy to debug***. Get one function working well then move on to the others.
- ***Easy to modify and expand***. Just add more functions to extend program capability
- For a ***large programming project***, you will code only a small fraction of the program.
- Make program ***self-documenting*** and readable

II. INTRODUCTION TO USER-DEFINED FUNCTIONS

[See Activity 6.2]

In order to define and use its own function, the programmer must do three things: ***Declare the function***, ***Define the function***, ***Use the function in the main code***. Let's study the following example.

/*Program to demonstrate the working of user defined function*/

```
#include<stdio.h>
void MyPrint()
{
printf("Printing from a function.\n");
}
int main()
{
        MyPrint();
        return 0;
}
```

```
#include <stdio.h>
int add(int a, int b);  //function prototype(declaration)
int main(){
     int num1,num2,sum;
     printf("Enters two number to add\n");
     scanf("%d %d",&num1,&num2);
     sum=add(num1,num2);         //function call
     printf("sum=%d",sum);
     return 0;
}
int add(int a, int b)            //function declarator
{
/* Start of function definition. */
     int add;
     add=a+b;
     return add;    //return statement of function
/* End of function definition. */
}
```

II.1 Function prototype

[See Activity 6.3]

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype tells the compiler about a function name and how to call the function. The actual **body of the function** can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function **add(),** following is the function declaration:

```
int add(int a, int b);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int add(int , int );
```

II.2 Function definition

[See Activity 6.4]

The function definition is the C code that imlements what the function does. Function definitions have the following syntax

```
return_type function_name (data_type variable_name_list)
{
        local declarations;
       function statements;
}
```

- where the **return_type** in the function header tells the type of the value returned by the function (default is int)
- where the **data type variable** name list tells what arguments the function needs when it is called (and what their types are)
- where **local declarations** in the function body are local constants and variables the function needs for its calculations.

```
float power(float r, int n)
{
      float prod;
      int i;
      prod=1;
      for (i=1;i<=n;i++)
      {
             prod *=r;
```

Example:

```
        }
        return prod;
    }
```

Some functions will not actually return a value or need any arguments. For these functions the keyword ***void*** is used. A function that does not return any value is also called a procedure. Here is an example:

```
void write_header(void)
{
        printf("**************************************************\n");
        printf("**          Welcome into this program?          **\n");
        printf("**          Developed by your name              **\n");
        printf("**          COMPUTER SCIENCE                    **\n");
        printf("**          IUGET – DOUALA – BONAMOUSSADI       **\n");
        printf("**************************************************\n");
}
```

The purpose of this function is just to display a menu on the screen, then it doesn't need any parameter and doesn't return any value to the main. Here:

– The **1st void** keyword indicates that no value will be returned.
– The **2nd void** keyword indicates that no arguments are needed for the function.
– This makes sense because all this function does is print out a header statement.

A function whose return type is different to void must contain the keyword return enabling to return a value to the calling program.

When a return is encountered the following events occur: execution of the function is terminated and control is passed back to the calling program, and the function call evaluates to the value of the return expression.

The data type of the return expression must match that of the declared *return_type* for the function.

```
float add_numbers (float n1, float n2)
{
        return n1 + n2; /*legal*/
        return 6; /*illegal, not the same data type*/
        return 6.0; /*legal*/
}
```

It is possible for a function to have multiple return statements. For example:

```
double absolute(double x)
{
        if (x>=0.0)
                return x;
        else
                return -x;
}
```

II.3 Calling functions

[See Activity 6.4]

To invoke a function, just type its name in your program and be sure to supply arguments (if necessary). A statement using our factorial program would look like

sum += power(x,i);

To invoke our write_header function, use this statement

write_header(void)

Some points to keep in mind when calling functions (your own or library's):

- The number of arguments in the function call must match the number of arguments in the function definition.
- The type of the arguments in the function call must match the type of the arguments in the function definition.
- The actual arguments in the function call are matched up in-order with the dummy arguments in the function definition.
- The actual arguments are passed by-value to the function. The dummy arguments in the function are initialized with the present values of the actual arguments. Any changes made to the dummy argument in the function will NOTaffect the actual argument in the main program.

III. RECURSION

[See Activity 6.5]

Recursion is the process in which a function repeatedly calls itself to perform calculations. Typical applications are games and sorting trees and lists. **Recursive algorithms** are not mandatory, usually an iterative approach can be found.

The following function calculates factorials recursively:

```
int factorial(int n)
{
        int result;
        if (n<=1)
                result=1;
        else
                result=n*factorial(n-1);
        return result;
}
```

IV. FUNCTION ARGUMENTS

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
While calling a function, there are **two ways** that arguments can be passed to a function:

IV.1 Function call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
/* function definition to swap the values */
void swap(int x,int y)
{
     int temp;
      temp = x;              /* save the value of x */
      x =        y;           /* put y into x */
      y = temp;                /* put x into y */
       return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example:

```
#include<stdio.h>
void swap(int x,int y);    /* function declaration */
int main ()
{
    int a =100;int b =200;       /* local variable definition */
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
    swap(a, b);        /* calling a function to swap the values */
    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
    return0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100

Before swap, value of b :200
After swap, value of a :100

After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

IV.2 Function call by reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the **value by reference**, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
void swap(int*x,int*y)     /* function definition to swap the values */
{
        int temp;
        temp =*x;       /* save the value at address x */
        *x =*y;/* put y into x */
        *y = temp;/* put x into y */
        return;
}
```

Let us call the function **swap()** by passing values by reference as in the following example:

```
#include<stdio.h>
void swap(int*x,int*y);      /* function declaration */
int main ()
{
        int a =100;int b =200;          /* local variable
definition */
        printf("Before swap, value of a : %d\n", a );
        printf("Before swap, value of b : %d\n", b );
        /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable a and
      * &b indicates pointer to b ie. address of variable b. */
        swap(&a,&b);
        printf("After swap, value of a : %d\n", a );
        printf("After swap, value of b : %d\n", b );
        return0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100
```

Which shows that there is no change in the values though they had been changed inside the function.

EXERCISE 6

Exercise 6.1: Write a function which take a real number and return its absolute value. Write a program which use the function

Exercise 6.2: Write two C functions that compute f(x)=2.3*x and g(x,y)=x*y.

Exercise 6.3: Write a function that returns 1 if an integer number is prime, 0 otherwise. Use that function to write a C program which display the list of prime number below a given integer n.

Exercise 6.4: Write a C function ***facto*** that computes the factorial of a number n (integer). The factorial function is a typical recursive problem. Rewrite the equation defining the factorial function in a recursive form. Write the recursive version of the code and call the function ***facto_rec***.

Exercise 6.5: Write the function called ***pow*** to calculate x^n for any real number *x* and integer *n*. Rewrite it in its recursive form and write the associate C function that you will call ***pow_rec.***

Exercise 6.6: The Taylor expansion of the exponential function is given by $\boldsymbol{e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}}$. Using the previous exercises (***function facto and pow***), write a C functions that calculates the exponential for integer values of x. Compare its results to the results of the C exponential function defined in <math.h>.

Exercise 6.7: Write a program that repeatedly asks the user to enter pairs of numbers until at least one of the pair is 0. For each pair, the program should use a function to calculate the harmonic mean of the numbers. The function should return the answer to main(), which should report the result. The harmonic mean of the numbers is the inverse of the average of the inverses and can be calculated as follows: harmonic mean of x and $y = 2.0 \times x \times y / (x + y)$

Exercise 6.8: Write a function which generate 50 integers between 1 and 100 randomly and do the follow. It calculates the factorial of each number divisible by 5 of and the inverse of each number divisible by 6. You must use two functions facto and inverse to calculate the factorial and the inverse of any number. Static variables should be used in those functions to count the number of time each of them has been called.

Exercise 6.9: Write a function that takes three arguments: the name of an int array, the array size, and an int value. Have the function set each element of the array to the int value.

Exercise 6.10: Write a program that asks the user to enter up to 10 golf scores, which are to be stored in an array. You should provide a means for the user to terminate input prior to entering 10 scores. The program should display all the scores on one line and report the average score. Handle input, display, and the average calculation with three separate array-processing functions.

Exercise 6.11: Write a function that takes three arguments: a pointer to the first element of a range in an array, a pointer to the element following the end of a range in an array, and an int value. Have the function set each element of the array to the int value.

Exercise 6.12: Write a function that takes a double array name and an array size as arguments and returns the largest value in that array. Note that this function shouldn't alter the contents of the array.

TOPIC 7

ARRAY, STRING & POINTER

Your fame as a programmer is now beginning to spread far and wide. The next person to come and see you is the chap in charge of the local cricket team. He would like to you write a program for him which allows the analysis of cricket results. What he wants is quite simple; given a list of cricket scores and their names he wants a list of them in ascending order. We will introduce in the following topic the notions of arrays, string and pointers.

Objectives: At the end of this lesson, student should have mastered the following sub-topics:

- ***Declaration and initialization an array, string and pointer***
- ***Character and string functions***
- ***Pointers and arrays***

Contents

I. ARRAY

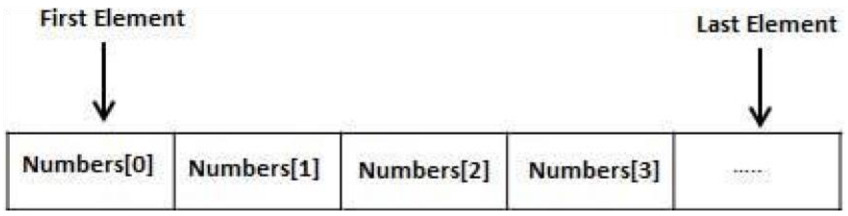
[See Activity 7.1]

I.1. Introduction to Array Variables

C programming language provides a data structure called **array**, which can store a fixed-size sequential collection of elements of the same type. An **array** is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as ***number0, number1, ..., and number99***, you declare one array variable such as **numbers** and use **numbers[0], numbers[1],** and ..., **numbers[99]** to represent individual variables. A specific element in an array is accessed by an **index**.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



I.2 Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a **single-dimensional array**. The **arraySize** must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10element array called balance of type double, use this statement:

```
double balance[10];
```

Now balance is a variable array which is sufficient to hold up-to 10 double numbers.

I.3 Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces **{ }** can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

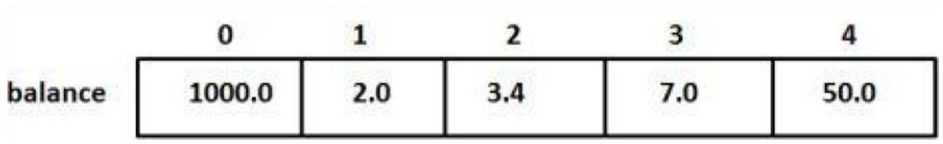
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th i.e. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:



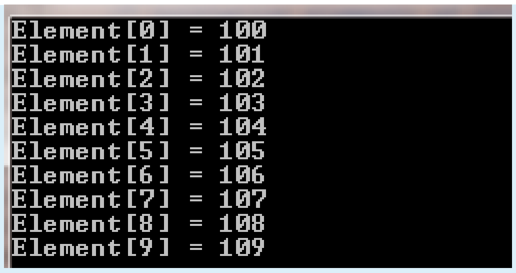
I.4 Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>
int main ()
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;
    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
   {
       n[i] = i + 100; /* set element at location i to i + 100 */
   }
  /* output each array element's value */
    for (j = 0; j < 10; j++ )
    {
     printf("Element[%d] = %d\n", j, n[j] );
   }
    return 0;
}
```



```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

II. TWO-DIMENSIONAL ARRAYS

[See Activity 7.2]

The simplest form of the **multidimensional** array is the **two-dimensional** array. A two dimensional array is, in essence, a list of one-dimensional arrays. To declare a two dimensional integer array of size x, y you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where type can be any valid C data type and **arrayName** will be a valid C identifier. A two dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimentional array **b**, which contains three rows and four columns can be shown as below:

	0th column	1st column	2nd column	3rd column	4th column
0th row	b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]
1st row	b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]
2nd row	b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

II.1 Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2
*/
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

II.2 Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>
 int main ()
{
  /* an array with 5 rows and 2 columns*/
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
  int i, j;
  /* output each array element's value */
   for ( i = 0; i < 5; i++ )
  {
    for ( j = 0; j < 2; j++ )
    {
      printf("a[%d][%d] = %d\n", i,j, a[i][j] );
    }
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Assignment

Write a C-program to search an element in a two dimensional array

III. STRING

[See Activity 7.3]

III.1 Introduction and declaration

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a **null-terminated** string contains the characters that comprise the string followed by a null.

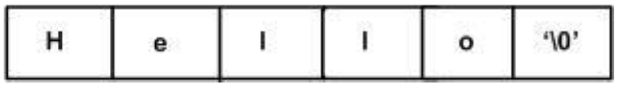
The following declaration and initialization create a string consisting of the word **"Hello"**. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word **"Hello"**.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above-defined string in C/C++:



Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above mentioned string:

```
#include <stdio.h>
 int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

III.2 Initializing Strings

Initializing a string can be done in three ways:

a. at declaration,
b. by reading in a value for the string, and
c. by using the **strcpy** function. **Direct initialization using the = operator is invalid**. The following code would produce an error:

```
char name[34];
name = "Erickson"; /* ILLEGAL */
```

To read in a value for a string use the **%s** format identifier: ***scanf("%s",name);***

Note that the address operator &is not needed for inputting a string variable (explained later). The end-of-string character will automatically be appended during the input process.

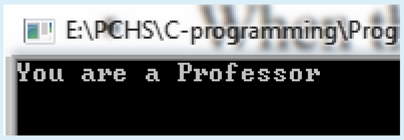
III.3 Copying Strings

The **strcpy** function is one of a set of built-in string handling functions available for the C programmer to use. To use these functions be sure to include the **string.h** header file at the beginning of your program. The syntax of strcpy is

strcpy(string1,string2);

When this function executes, **string2** is copied into string1at the beginning of string1. The previous contents of **string1** are overwritten. In the following code**, strcpy** is used for string initialization:

```
#include <string.h>
main ()
{
        char job[50];
        strcpy(job,"Professor");
        printf("You are a %s \n",job);
        return 0;
}
```



```
You are a Professor
```

III.4 String I/O Functions

[See Activity 7.4]

There are special functions designed specifically for string I/O. They are

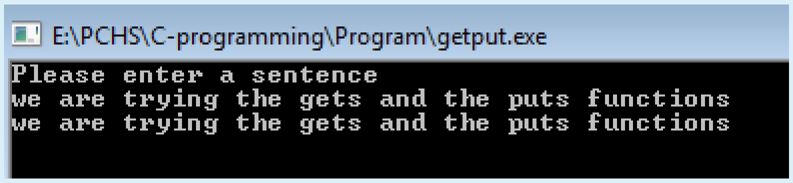
gets(string_name);

puts(string_name);

- The ***gets function*** reads in a string from the keyboard. When the user hits a carriage return the string is inputted.
- The ***function puts*** displays a string on the monitor. It does not print the end-of-string character, but does output a carriage return at the end of the string. Here is a sample program demonstrating the use of these functions:

```
#include<stdio.h>
main()
{
    char phrase[100];
    printf("Please enter a sentence\n");
    gets(phrase);
    puts(phrase);
    getchar();
}
```

A sample session would look like



```
Please enter a sentence
we are trying the gets and the puts functions
we are trying the gets and the puts functions
```

III.5 More String Functions

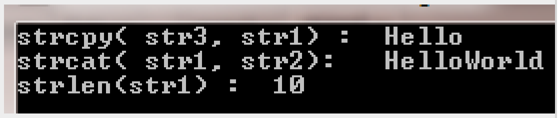
Included in the **string.h** library are several more string-related functions that are free for you to use. Here is a brief table of some of the more popular ones

Function	**Operation**	**Syntax**
Strcat	Append to a string	strcat(string1, string2);
Strchr	Finds first occurrence of a given character	**strchr(s1, ch);**

…		Returns a pointer to the first occurrence of character ch in string s1.
Strcmp	Compares two strings	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
Strcmpi	Compares two, strings, non-case sensitive	
Strcpy	Copies one string to another	**strcpy(s1, s2);** Copies string s2 into string s1.
Strlen	Finds length of a string	**strlen(s1);** Returns the length of string s1.
Strncat	Appends n characters of string	
strncmp	Compares n characters of two strings	
Strncpy	Copies n characters of one string to another	
…	…	…

Following example makes use of few of the above-mentioned functions:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int  len ;
    strcpy(str3, str1);     /* copy str1 into str3 */
    printf("strcpy( str3, str1) :  %s\n", str3 );
    strcat( str1, str2);  /* concatenates str1 and str2 */
     printf("strcat( str1, str2):   %s\n", str1 );
      len = strlen(str1); /* total lenghth of str1 after
concatenation */
      printf("strlen(str1) :  %d\n", len );
      return 0;
}
```



```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

Examples: Here are some examples of string functions in action:

char s1[]="big sky country";
char s2[]="blue moon";
char s3[]="then falls Caesar";

<u>**Function**</u> <u>**Result**</u>

strlen(s1)	15 /* e-o-s not counted */
strlen(s2)	9
strcmp(s1,s2)	negative number
strcmp(s3,s2)	positive number
strcat(s2,” tonight”)	blue moon tonight

III.6 Character I/O Functions

[See Activity 7.5]

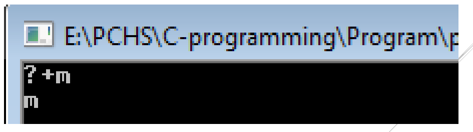
Analogous to the **gets** and **puts** functions, **getchar** and **putchar** functions specially designed for character I/O. The following program illustrates their use

<table>
<tr>
<td>

```
#include <stdio.h>
main()
{
    int n; char lett;
    putchar('?');
    n=45;
    putchar(n-2);
    lett=getchar();
    putchar(lett);
    putchar('\n');
    return 0;
}
```

</td>
<td>

A sample session using this code would look like:



E:\PCHS\C-programming\Program\p

```
?+m
m
```

</td>
</tr>
</table>

More Character Functions **[See Activity 7.6]**

As with strings, there is a library of functions designed to work with character variables. The file **ctype.h** defines additional routines for manipulating characters. Here is a partial list:

Function	**Operation**
isalnum	Tests for alphanumeric character
isalpha	Tests for alphabetic character
isascii	Tests for ASCII character
iscntrl	Tests for control character
isdigit	Tests for 0 to 9
isgraph	Tests for printable character
islower	Tests for lowercase character
isprint	Tests for printable character
ispunct	Tests for punctuation character
isspace	Tests for space character
isupper	Tests for uppercase character
isxdigit	Tests for hexadecimal
toascii	Converts character to ASCII code

tolower	Converts character to lowercase
toupper	Converts character to uppercase

Example: The following program give an overview on the use of some of those functions

```
#include<ctype.h>
#include<stdio.h>
main()
{
     char car;
     do
     {
         printf("\nEnter an alphabetic character: ");
         car=getchar();
         if(isdigit(car)) printf("\nwrong: %c is a numeric character",car);
         else if (ispunct(car)) printf("\nWrong: %c is a punctuation character",car);
         else if (isspace(car)) printf("\nWrong: %c is a space character",car);
         else if(!isalnum(car)) printf("\nWrong: %c is special character",car);
     }while(!isalpha(car));
     printf("\nWell done! %c is an alphabetic character", car);
     if(islower(car)) printf("\n%c in uppercase is %c.",car, toupper(car));
     if(isupper(car)) printf("\n%c in lowercase is %c.",car, tolower(car));
     printf("\n");
     return 0;
}
```

IV. POINTER

IV.1 Introduction to pointers

[See Activity 7.7]

Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

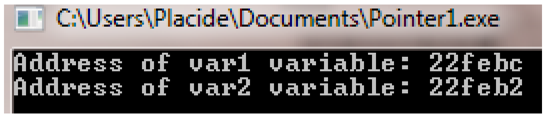
As you know, every variable is a memory location and every memory location has its address defined which can be accessed using **ampersand (&)** operator, which denotes an address in memory.

Consider the following example, which will print the address of the variables defined:

```
#include <stdio.h>
```

```
 int main ()
{
   int  var1;
      char var2[10];
       printf("Address of var1 variable: %x\n", &var1  );
       printf("Address of var2 variable: %x\n", &var2  );
       return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:



```
Address of var1 variable: 22febc
Address of var2 variable: 22feb2
```

IV.2 What Are Pointers?

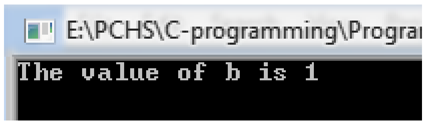
A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

`#include <stdio.h>`
`main()`
`{`
`    int a=1,b=78,*ip;`
`    ip=&a;`
`    b=*ip; /* equivalent to b=a */`
	
	`The value of b is 1`

printf("The value of b is %d\n",b);
return 0;
}	

Note that **b** ends up with the value of **a** but it is done indirectly; by using a pointer to **a**

IV.3 How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>
int main ()
{
     int  var = 20;   /* actual variable declaration */
     int  *ip;        /* pointer variable declaration */
     ip = &var;  /* store address of var in pointer variable*/
     printf("Address of var variable: %x\n", &var  );
     /* address stored in pointer variable */
     printf("Address stored in ip variable: %x\n", ip );
    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );
    return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of *ip variable: 20
```

IV.4 NULL Pointers in C

It is always a good practice to assign a **NULL** value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned **NULL** is called a **null** pointer.

The **NULL** pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>
 int main ()
{
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", &ptr  );
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr)      /* succeeds if p is not null */  if(!ptr)     /* succeeds if p is null */
```

IV.5 "Call-by-Reference" Arguments

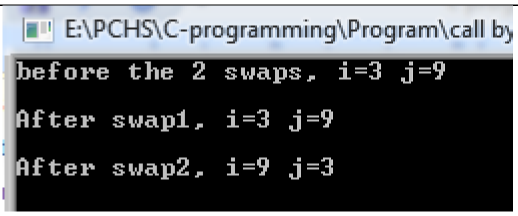
We learned earlier that if **a** variable in the main program is used as an actual argument in **a** function call, its value won't be changed no matter what is done to the corresponding dummy argument in the function.

What if we would like the function to change the main variable's contents?

- To do this we use pointers as dummy arguments in functions and indirect operations in the function body. (The actual arguments must then be addresses)
- Since the actual argument variable and the corresponding dummy pointer refer to the same memory location, changing the contents of the dummy pointer will- by necessity- change the contents of the actual argument variable.

The classic example of "call-by-reference" is a swap function designed to exchange the values of two variables in the main program. Here is a swapping program

```
#include <stdio.h>
void swap1(int p,int q);
void swap2(int *p,int *q);
main()
```

```
{
    int i=3,j=9;
    printf("before the 2 swaps, i=%d  j=%d\n",i,j);
    swap1(i,j);
    printf("\nAfter swap1, i=%d j=%d\n",i,j);
    swap2(&i,&j);
     printf("\nAfter swap2, i=%d j=%d\n",i,j);
     return 0;
}
void swap1(int p,int q)
{
    int temp;
    temp=p;
    p=q;
    q=temp;
}
void swap2(int *p,int *q)
{
    int temp;
    temp=*p;
    *p=*q;
    *q=temp;
}
```
	 E:\PCHS\C-programming\Program\call by
	```
	before the 2 swaps, i=3 j=9

	After swap1, i=3 j=9

	After swap2, i=9 j=3
	```

IV.6 Pointers and Arrays [See Activity 7.8]

Although this may seem strange at first, in C an array name is an address. In fact, it is the base address of all the consecutive memory locations that make up the entire array.

We have actually seen this fact before: when using **scanf** to input a character string variable called name the statement looked like

scanf("%s",name);   NOT   scanf("%s",&name);

Given this fact, we can use pointer arithmetic to access array elements.

Illustration

Given the following array declaration: ***int a[467];***

The following two statements do the exact same thing:

a[5]=56; ****(a+5)=56;***

Here is the layout in memory:

a	133268	a[0]
a + 1	133272	a[1]
a + 2	133276	a[2]
a + 3	133280	a[3]
a + 4	133284	a[4]
a + 5	133288	a[5]

Examples

Normal way	Other way	Another way

int a[100],i,*p,sum=0;
for(i=0; i<100; ++i)
sum +=a[i];
	int a[100],i,*p,sum=0;
	for(i=0; i<100; ++i)
	sum += *(a+i);
		int a[100],i,*p, sum=0;
		for(p=a; p<&a[100]; ++p)
		sum += *p;

EXERCISE 7

Exercise 7.1: Write a program which reads in 10 numbers and then prints them out in the reverse order to that which they were entered in.

Exercise 7.2: Write code that reads a list of numbers from the keyboard, store the numbers in an array, and then:

a) print out the array in reverse order
b) calculate the average of the numbers
c) print the index of the numbers that are below the average.

Exercise 7.3: Given a two-dimensional square array of integers, with size N * N, called **numTable**, write code to:

a) Initialize the array by command-line inputs
b) Calculate the statistics of the average, the minimum, and the maximum of the array
c) Calculate the sum on both diagonals.

Exercise 7.4: Consider an array X of length 25. Use the function **rand()** to initialized randomly the array. Write a program that returns the position and value of the minimum.

Exercise 7.5: Write code that creates a two-dimensional intarray of size M * N and set values for it. Write code that first calculates the sum of every column, and then the sum of every row. Specify you are printing row sums or column sums.

Exercise 7.6: Consider 2 three-dimensional vectors X and Y whose components are specified by the user and stored in 1D arrays. Write the C code that calculates if the two vectors are orthogonal.

Exercise 7.7: Write a program to remove all specified characters from a string.

Exercise 7.8: Given a polynomial of degree n defined as follow

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

write the associated C code assuming n<50. The user will be asked to supply the degree of the polynomial as well as the coefficients a_i. The coefficients will be stored in an array.

Exercise 7.9: Build a program that uses a single-dimension array to store 10 numbers input by a user. After inputting the numbers, the user should see a menu with two options to sort and print the 10 numbers in ascending or descending order.

Exercise 7.10: Write a code to insert a blank every fifth character in a string

Exercise 7.11: Create a program that performs the following functions:

- Uses character arrays to read a user's name from standard input.
- Tells the user how many characters are in his or her name.
- Displays the user's name in uppercase.

Exercise 7.12: Write a program that will output the characters of a string backwards. For example, given the string "computer", the program will produce "retupmoc".

Exercise 7.13: Consider X and Y, two sorted arrays of integers. Write a C code that concatenates the two tables into one table Z (sorted) which contains the elements of X and Y.

Exercise 7.14: Write a small C program that prints on the screen in uppercase a text supplied by the user. The string will not be more than 80 characters in size.

Exercise 7.15: Given a string as input, write a function that counts the numbers, the lower case, upper case and special characters.

Exercise 7.16: Using the integer representation of a character, write a function that replaces all the lower case characters in a string by upper case characters.

Exercise 7.17: Define two integer arrays, each 10 elements long, called array1 and array2. Using a loop, put some kind of nonsense data in each and add them term for term into another 10 element array named arrays. Finally, print all results in a table with an index number, for example

1- 2 + 10 = 12
2- 4 + 20 = 24
3- 6 + 30 = 36 etc.

Exercise 7.18: Build a program that performs the following operations:

- Declares three pointer variables called iPtr of type int, cPtr of type char, and fFloat of type float.
- Declares three new variables called iNumber of int type, fNumber of float type, and cCharacter of char type.
- Assigns the address of each non-pointer variable to the matching pointer variable.
- Prints the value of each non-pointer variable.
- Prints the value of each pointer variable.
- Prints the address of each non-pointer variable.
- Prints the address of each pointer variable.

Exercise 7.19: Create a program that allows a user to select one of the following four menu options:

- Enter New Integer Value
- Print Pointer Address
- Print Integer Address
- Print Integer Value

For this program you will need to create two variables: one integer data type and one pointer. Using indirection, assign any new integer value entered by the user through an appropriate pointer.

Exercise 7.20: Build a program that uses an array of strings to store the following names: *"Florida", "Oregon"; "California", "Georgia"* Using the preceding array of strings, write your own sort() function to display each state's name in alphabetical order using the strcmp() function.

Exercise 7.21: Define a character array and use strcpy to copy a string into it. Print the string out by using a loop with a pointer to print out one character at a time. Initialize the pointer to the first element and use the double plus sign to increment the pointer. Use a separate integer variable to count the characters to print. Modify the program to print out the string backwards by pointing to the end and using a decrementing pointer

TOPIC 8

COMPLEX DATA TYPES

In this tutorial you will learn about C Programming - Structures and Unions, initializing structure, assigning values to members, functions and structures, passing structure to functions, passing entire function to functions, arrays of structure, structure within a structure and union.

Structures are slightly different from the variable types you have been using till now. Structures are **data types** by themselves. When you define a structure or union, you are creating a custom data type.

Objectives: At the end of this lesson, student should have mastered the following topics:

- *Declaration and initialization of structures*
- *Array of structures*
- *Structures and functions*
- *files*

Contents

I. STRUCTURE

I.1. Structure definition **[See Activity 8.1]**

C arrays allow you to define type of variables that can hold several data items of the same kind but structure is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Unlike arrays, structure must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- **Title**
- **Author**
- **Subject**
- **Book ID**

To define a structure, you must use the **struct** statement. The **struct statement** defines a new data type, with more than one member for your program. The format of the **struct statement** is this:

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
      char  title[50];
      char  author[50];
      char  subject[100];
      int   book_id;
} book;
```

The above is a declaration of a data type called student. *It is not a variable declaration, but a type declaration*.

- The keyword **struct** declares a structure to hold the details of four data fields,
- **Book_record** is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure.
- ***title, author, nb,*** and ***price*** are different fields of the structure. These fields are called structure elements or members. Each member may belong to different type of data.

To actually declare a structure variable, the standard syntax is used:

struct book_record *book1, book2, book3;*

You can declare a structure type and variables simultaneously. Consider the following declaration:

Struct *book_record*
{

```
        char title[20];
        char author[15];
        int nb;
        float price;
    } book1, book2, book3;
```

I.2- Access to a member **[See Activity 8.2]**

To access the members of a structure, you use the “”.”” (scope resolution) operator. Shown below is an example of how you can accomplish initialization by assigning values using the scope resolution operator:

```
strcpy(book1.title, “Quick Mastery”);
strcpy(book1.author, “Bennett CHAH”);
book1.nb=145;
book1.price=3500.0;
```

I.3- Initializing a structure

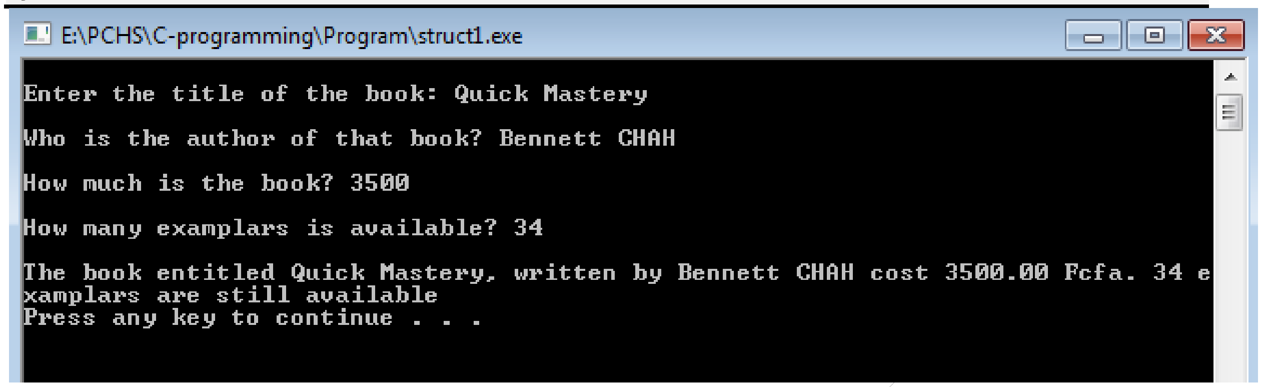
Structure members can be initialized when you declare a variable of your structure:

```
struct book_record book1={“Quick Mastery”,”Bennet CHAH”,145,3500.0};
```

The above declaration will create a *struct book_record* called book1 with a title “Quick Mastery”, the author “Bennett CHAH”, 145 exemplars at 3500 frs.

I.4- Example **[See Activity 8.3]**

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    struct book_record
    {
        char title[20];
        char author[15];
        int nb;
        float price;
    };
    struct book_record book1;
    printf("\nEnter the title of the book: ");
    gets(book1.title);
    printf("\nWho is the author of that book? ");
    gets(book1.author);
    printf("\nHow much is the book? ");
    scanf("%f",&book1.price);
    printf("\nHow many examplars is available? ");
    scanf("%d",&book1.nb);
    printf("\nThe book entitled %s, written by %s cost %.2f Fcfa. %d examplars are still
available\n", book1.title,book1.author,book1.price,book1.nb);
    system("pause");
}
```



```
Enter the title of the book: Quick Mastery

Who is the author of that book? Bennett CHAH

How much is the book? 3500

How many examplars is available? 34

The book entitled Quick Mastery, written by Bennett CHAH cost 3500.00 Fcfa. 34 e
xamplars are still available
Press any key to continue . . .
```

I.5- Comparison of structures

Two variables of the same structure type can be compared the same way as ordinary variables. If book1 and book2 belong to the same structure, then the following operations are valid:

- ***book1 = book2*** Assign book2 to book1.
- ***book1 = =book2*** Compare all members of book1 and book2 and return 1 if they are equal, 0 otherwise.
- ***book1 != book2*** Return 1 if all the members are not equal, 0 otherwise.

I.6- Functions and structures

Since structures are of custom data types, functions can return structures and also take them as arguments. Keep in mind that when you do this, you are making a copy of the structure and all it's members so it can be quite memory intensive.

To return a structure from a function, declare the function to be of the structure type you want to return. In our case the following code should produce the same output than the previous one:

```
#include<stdio.h>
#include<stdlib.h>
  struct book_record
    {
          char title[20];
          char author[15];
          int nb;
          float price;
    };
```

```
struct book_record book_func()
{
      struct book_record book;
      printf("\nEnter the title of the book: ");
      gets(book.title);
      printf("\nWho is the author of that book? ");
      gets(book.author);
      printf("\nHow much is the book? ");
      scanf("%f",&book.price);
      printf("\nHow many examplars is available? ");
      scanf("%d",&book.nb);
      return book;
}
main()
{
      struct book_record book1;
      book1=book_func();
      printf("\nThe book entitled %s, written by %s cost %.2f Fcfa. %d examplars are still
available\n",book1.title,book1.author,book1.price,book1.nb);
      system("pause");
}
```

I.6- Array of structures

Since structures are data types that are especially useful for creating collection items, why not make a collection of them using an array? Let us now modify our above example to use an array of structures rather than individual ones.

The following code is an example of use of array of structure

```
#include<stdio.h>
#include<stdlib.h>
  struct book_record
      {
             char title[20];
             char author[15];
             int nb;
             float price;
      };
struct book_record book_func()
{
      struct book_record book;
      printf("\nEnter the title of the book: ");
      gets(book.title);
      printf("\nWho is the author of that book? ");
      gets(book.author);
      printf("\nHow much is the book? ");
      scanf("%f",&book.price);
```

```
    printf("\nHow many examplars is available? ");
    scanf("%d",&book.nb);
    return book;
}
void printbook (struct book_record book, int n)
{
   int i;
   printf("\n BOOK\t\tTITLE\t\tPRICE\tNUMBER");
   for(i=0;i<n;i++)
    printf("\n %s\t\t%s\t\t%.2f\t%d\n",book.title,book.author,book.price,book.nb);
}
main()
{
    int i,n;
    printf("\nEnter the number of book to record: ");
    scanf("%d",&n);
    struct book_record library[100];
    for (i=0;i<n;i++)
    {
      library[i]=book_func();
    }
    printbook(library[i],n);
    system("pause");
}
```

II. ENUMERATION

enum data types are data items whose values may be any member of a symbolically declared set of values. A typical declaration would be.

***enum** days {Mon, Tues, Weds, Thurs, Fri, Sat, Sun};*

This declaration means that the values Mon...Sun may be assigned to a variable of type **enum** days. The actual values are 0...6 in this example and it is these values that must be associated with any input or output operations. For example the following program produced the following output: ***start = 2 end = 5***

```
#include <stdio.h>
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun };
main()
{
    enum days start, end;
    start = Wed;
    end = Sat;
    printf ("start = %d end = %d\n",start,end);
    getchar();
}
```

Each value of the enumerated type list is given an **int** value. If no extra information is provided by the programmer, the values start at zero and increase by one from left to right. The value of any constant in the enumeration list can be set by the programmer; subsequent enumeration constants will be given values starting from this value. There is no need for the values given to the enumeration constants to be unique.

The following (non-sensical) code fragment illustrates these three points:

enum *day { sun=1, mon, tues, weds, thur, fri=1, sat } d1;*

the values of the enumeration constants will be:

sun 1,mon 2,tues 3,weds 4,thurs 5,fri 1,sat 2

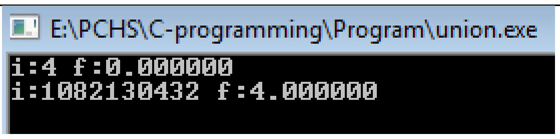
III. UNION

Unions and Structures are identical in all ways, except for one very important aspect. Only one element in the union may have a value set at any given time. Everything we have shown you for structures will work for unions, except for setting more than one of its members at a time. For example, the following code declares a union data type called intfloat and a union variable called proteus:

```
union intfloat
{
    floatf;
    inti;
};
union intfloat proteus;
```

- Once a union variable has been declared, the amount of memory reserved is just enough to be able to represent the largest member. (Unlike a structure where memory is reserved for all members).
- In the previous example, 4 bytes are set aside for the variable proteus since a float will take up 4 bytes and an int only 2 (on some machines).
- Data actually stored in a union's memory can be the data associated with any of its members. But only one member of a union can contain valid data at a given point in the program.
- It is the user's responsibility to keep track of which type of data has most recently been stored in the union variable.
- The following code illustrates the chameleon-like nature of the union variable Proteus defined earlier

#include <stdio.h>
main() {
union intfloat {	

float f;
int i;
} proteus;
proteus.i=4 /* Statement 1 */
printf("i:%d f:%f\n",proteus.i,proteus.f);
proteus.f=4.0; /* Statement 2 */
printf("i:%d f:%f\n",proteus.i,proteus.f);
}
	
	E:\PCHS\C-programming\Program\union.exe
	i:4 f:0.000000
	i:1082130432 f:4.000000

- After Statement 1, data stored in proteus is an integer the float member is full of junk.
- After Statement 2, the data stored in proteus is a float, and the integer value is meaningless

IV. TYPEDEF

It is possible to create new names for existing types with typedef . This is frequently used to give shorter or less complicated names for types, making programming safer and hopefully easier. The use of typedef is a simple macro-like facility for declaring new names for data types, including user-defined data types. Typical examples are shown below :-

```
typedef long BIGINT;
typedef double REAL;
typedef struct point
{
    double x;
    double y;
} POINT;
```

Given the above declarations, it would be possible to write the following declarations:

```
POINT a,b,c;
REAL a1,a2;
```

V. FILES

So far, all the output (formatted or not) in this course has been written out to what is called standard output (which is usually the monitor). Similarly all input has come from standard input (usually associated with the keyboard).

The C programmer can also read data directly from files and write directly to files. To work with files, the following steps must be taken:

I.1- Defining and opening the file **[See Activity 8.4]**

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore all files should be declared as type FILE before they are used. FILE is a defined data type. Constants such as FILE, EOF and NULL are defined in <stdio.h>. The following is the general format for declaring and opening a file:

FILE *fp;
fp = fopen("filename", "mode");

Mode can be one of the following:

- "r" open the file for reading only.
- "w" open the file for writing only.
- "a" open the file for appending(or adding)data to it.

The additional modes of operation are:

- "r+" the existing file is opened to the beginning for both reading and writing.
- "w+" same as w except both for reading and writing.
- "a+" same as a except both for reading and writing.

The following useful table lists the different actions and requirements of the different modes for opening a file:

	r	w	a	r+	w+	a+
file must exist before open	✓			✓		
old file contents discarded on open		✓			✓	
stream can be read	✓			✓	✓	✓
stream can be written		✓	✓	✓	✓	✓
stream can be written only at end			✓			✓

I.2- Closing a file

The **fclose** function in a sense does the opposite of what the **fopen** does: it tells the system that we no longer need access to the file. This allows the operating system to cleanup any resources or buffers associated with the file.
The syntax for file closing is simply

fclose(in_file);

I.3- Reading and writing on a file **[See Activity 8.5]**

The functions **fprintf** and **fscanf** are provided by C to perform the analogous operations for the printf and scanf functions but on a file.

Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

fscanf(fp,"%f %d",&x,&m);

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

fprintf(fp,"%s","This is just an example :)");

I.4- Additional File I/O Functions

Many of the specialized I/O functions for characters and strings that we have described in this course have analogs which can be used for file I/O. Here is a list of these functions

Function	**Result**
fgets	file string input
fputs	file string output
getc(fp)	file character input
putc(fp)	file character output

The Standard I/O Library provides similar routines for file I/O to those used for standard I/O.

The routine **getc(fp)** is similar to **getchar()** and **putc(c,fp)** is similar to **putchar(c)**.

Thus the statement **c = getc(fp);** reads the next character from the file referenced by fp and the statement **putc(c,fp);** writes the character c into file referenced by fp.

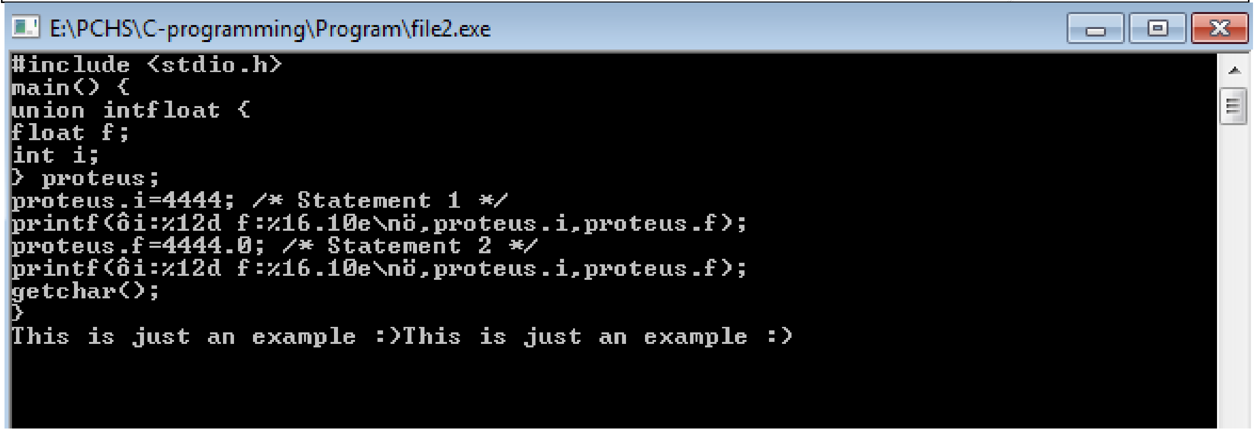
Another useful function for file I/O is **feof()** which tests for the end-of-file condition. **feof** takes one argument -- the FILE pointer -- and returns a nonzero integer value (TRUE) if an attempt has been made to read past the end of a file. It returns zero (FALSE) otherwise. A sample use:

if (feof(fp))
printf ("No more data \n");

I.4- Additional File I/O Functions

The following program read the file union.cpp which should be in the same folder than the source code, append it and display it on the screen.

```
#include <stdio.h>
  main( )
  {
    FILE *fp;
    char c;
    fp = fopen("union.cpp", "a");
    if (fp == NULL)
            printf("File doesn't exist\n");
    else {
    fprintf(fp,"%s","This is just an example :)"); /*append some text*/
    fclose(fp);
    fp = fopen("union.cpp", "r");
     do {
      c = getc(fp); /* get one character from the file */
        putchar(c); /* display it on the monitor*/
      } while (c != EOF); /* repeat until EOF (end of file) */
    }
   fclose(fp);
   getchar();
  }
```



```
#include <stdio.h>
main() {
union intfloat {
float f;
int i;
} proteus;
proteus.i=4444; /* Statement 1 */
printf(ôi:%12d f:%16.10e\nö,proteus.i,proteus.f);
proteus.f=4444.0; /* Statement 2 */
printf(ôi:%12d f:%16.10e\nö,proteus.i,proteus.f);
getchar();
}
This is just an example :)This is just an example :)
```

EXERCISE 8

Exercise 8.1: Let's consider the following structure declaration

```
Struct fraction
{
  Int num;
  Int dem;
};
```

Write functions enabling to do the following manipulations on fractions

a) Add two fractions
b) Multiply two fractions
c) Divide two fractions

Exercise 8.2: Create a structure called car with the following members:

- make
- model

• year
• miles

a) Create an instance of the car structure named myCar and assign data to each of the members. Print the contents of each member to standard output using the printf() function.
b) Using the car structure from challenge number one, create a structure array with three elements named myCars. Populate each structure in the array with your favorite car model information. Use a for loop to print each structure detail in the array.

Exercise 8.3: A complex number can be defined by the following structure

```
struct comp
{
      double real;
      double imag;
};
```

Use the following prototypes to write the corresponding function

a) ***void printcomp(struct comp a);*** to print a complex number in the form x+yi
b) ***int mod(struct comp a);*** To calculate the modulus of a given complex number a
c) ***int compare(struct comp a, struct comp b);*** to compare two complex numbers
d) ***struct comp add(struct comp a, struct comp b);*** to add two complex numbers
e) ***struct comp add(struct comp a, struct comp b);*** to multiply two complex numbers

Write a main program to use these functions. A menu should be displayed to enable the user choosing the operation he wants to perform

Exercise 8.4: Let's consider the following structure

```
Struct point
{
            Float x;
            Float y;
}
```

Write a function enable you to:

a) Calculate the distance between two points
b) Calculate the coordinate of vertor AB given 2 points A and B
c) Make a translation of a point by a vector

Exercise 8.5: Write a simple database program that will store students' information such as name, age, mark and grade. The following instructions should be followed:

- An array of structure should be used to store information of a maximum of 20 students
- The number of students to record and the information should be entered by the user

- The grade should be assigned by a function according to the following table

mark	**grade**
>= 80	A
<80 and >=60	B
<60 and >=40	C
<40	F

The write codes to perform the following

a) Calculate the mark average of all the student
b) Determine the first and the last student
c) Display the students who have passed knowing that a student has passed if he has at least 60/100
d) Given a student name display his details if the name exist in the database

Exercise 8.6: Let's consider the following code fragment. It show how a program might check if a file could be opened appropriately. The function **exit()** is a special function which terminates your program immediately.

```
fp = fopen (filename, “r”) ;
     if ( fp  ==  NULL)
     {
          printf(“Cannot open %s for reading \n”, filename );
          exit(1) ; /*Terminate program: Commit suicide !!*/
     }
```

a) Modify the code so that the a valid name of file should be entered before the execution of the program continue
b) Modify the above code fragment to allow the user 3 chances to enter a valid filename. If a valid file name is not entered after 3 chances, terminate the program.

Exercise 8.7: Write a program to count the number of lines and characters in a file.

Note: Each line of input from a file or keyboard will be terminated by the newline character '\n'. Thus by counting newlines we know how many lines there are in our input.

Exercise 8.8: Write a program to display file contents 20 lines at a time. The program pauses after displaying 20 lines until the user presses either Q to quit or Return to display the next 20 lines. The following algorithm can be used

```
read character from file
while not end of file and not finished do
begin
     display character
     if character is newline then
          linecount = linecount + 1;
```

```
        if linecount == 20 then
        begin
            linecount = 1 ;
            Prompt user and get reply;
        end
        read next character from file
    end
```

Exercise 8.9: Write a file copy program which copies the file "prog.c" to "prog.old"

Outline solution:

```
    Open files appropriately
    Check open succeeded
    Read characters from prog.c and
    Write characters to prog.old until all characters copied
      Close files
```

The step: "Read characters .... and write .." may be refined to:

```
    read character from prog.c
    while not end of file do
    begin
        write character to prog.old
        read next character from prog.c
    end
```

Exercise 8.10: The following simple structure declarations might be found in a graphics environment.

```
struct point
{
        double x;
        double y;
};
struct circle
{
        double rad;
        struct point cen;
};
```

With the declarations given above write simple C function to

a) Calculate the area of a circle
b) Determine whether a given point lay inside a circle

Exercise 8.11: Write a program to compare two files specified by the user, displaying a message indicating whether the files identical or different. This is the basis of a **compare** command provided by most operating systems. Here our file processing loop is as follows:

```
read character ca from file A;
read character cb from file B;
while ca == cb and not EOF file A and not EOF file B
begin
    read character ca from file A;
    read character cb from file B;
end
if ca == cb then
        printout(“Files identical”);
else
        printout(“Files differ”);
```

Exercise 8.12: By using the structure point defined in exercise 4, consider the following structure

```
struct line
{
    struct point start;
    struct point end;
};
```

Using the structure above, write a code to determine if two lines are

a) Parallel
b) Perpendicular

Exercise 8.13: Redo exercise 5 using a file to store the database. A menu should be prompted to the user to choose the operation to do

a) Display the detail of the database
b) Add a record to the database
c) Give the number of records of the database
d) Calculate the mark average of all the student
e) Etc,

BIBLIOGRAPHY

1- [BRIAN W. KERNIGHAN & DENNIS M. RITCHIE], C Programming Language, Second edition
2- [MICHAEL VINE], C Programming for absolute beginner, Second edition
3- [DAN GOOKIN], C for Dummies, Second edition
4- [MAUDE MANOUVRIER], Langage C: énoncé et corrigé des exercices, Université Paris Dauphine
5- [ROB MILES], Introduction to C programming, The University of Hull
6- Fundamental of C Programming, Department of Computer Science and engineering, University of Moratuwa