Dr. Umayal Ramanathan College for Women, Karaikudi.
(Accredited with 'B+' Grade by NAAC)
Affiliated to Alagappa University
(Run by Alagappa Chettiar Educational Trust)

Study Material
2020-2021

Subject Name: Programming in C
Subject Code: 7BCE1C1

Name of the Staff: Ms. G. Divya
Department: Computer Science

Course Outcome:

Semester	Course Title	Course Code	Course Outcome	
I	Programming in C	7BCE1C1	CO1	To introduce the field of programming using C language. Usage of Arithmetic operator, Conditional operator, logical operator and relational operators and other C constructs.
			CO2	Compare and contrast of the decision making, branching and looping constructs data flow diagram with examples.
			CO3	Understand the concepts of one dimensional and two dimensional arrays.
			CO4	Implement different operations on User defined functions, structures and unions.
			CO5	Determine the address of variable as pointers and files.

B.Sc., COMPUTER SCIENCE
I YEAR – I SEMESTER
COURSE CODE: 7BCE1C1
CORE COURSE-I–PROGRAMMING IN C

Unit I

Overview of C: History of C – Importance of C – Basic Structure of C Programs –Programming Style – Character Set – C Tokens – Keywords and Identifiers – Constants, Variables and Data Types – Declaration of Variables – Defining Symbolic Constants –Declaring a variable as a constant – overflow and underflow of data – Operators and Expressions: Arithmetic, relational, logical, assignment operators – increment and decrement operators, conditional operators, bitwise operators, special operators – Arithmetic Expressions- Evaluation of Expressions – Precedence of Arithmetic Operators – Type Conversions in Expressions – Operator Precedence and Associativity – Mathematical functions.

Unit II

Managing I/O Operations: Reading and Writing a Character – Formatted Input, Output – Decision Making & Branching: if statement - if else statement - nesting of if else statements - else if ladder – switch statement – the ?: operator – goto statement – the while statement – do statement – the for statement – jumps in loops.

Unit III

Arrays: One-Dimensional Arrays – Declaration, Initialization – Two-Dimensional Arrays – Multi-dimensional Arrays – Dynamic Arrays – Initialization. Strings: Declaration, Initialization of string variables – reading and writing strings – string handling functions.

Unit IV

User-defined functions: need – multi-function programs – elements of user definedfunctions – definition – return values and their types – function calls, declaration, category –all types of arguments and return values – nesting of functions – recursion – passing arrays,strings to functions – scope visibility and life time of variables. Structures and Unions:Defining a structure – declaring a structure variable – accessing structure members –initialization – copying and comparing – operation on individual members – array of structures – arrays within structures – structures within structures – structures and functions –unions – size of structures – bit fields.

Unit V

Pointers: the address of a variable – declaring, initialization of pointer variables –accessing a variable through its pointer – chain of pointers – pointer increments and scale factors – pointers and character strings – pointers as function arguments – pointers and structures. Files: Defining, opening, closing a file – IO Operations on files – Error handling during IO operations – command line arguments.

Text Book:

1. Programming in ANSI C, E.Balagurusamy, 6th Edition, Tata McGraw Hill Publishing Company, 2012.

UNIT I: Chapters 1 (Except 1.3-1.7, 1.10-1.12), 2 (Except 2.9, 2.13), 3(Except 3.13)
UNIT II: Chapters 4 – 6
UNIT III: Chapters 7, 8 (Except 8.5, 8.6, 8.7, 8.9, 8.10)

UNIT IV: Chapters 9 (Except 9.20), 10
UNIT V: Chapters 11 (Except 11.8, 11.10, 11.12, 11.14, 11.15, 11.17), 12(Except 12.6)

Books for Reference:

1. Programming with C, Schaum’s Outline Series, Gottfried, Tata McGraw Hill, 2006
2. Programming with ANSI and Turbo C , Ashok N.Kamthane , Pearson Education, 2006
3. H. Schildt, C: The Complete Reference, 4th Edition, TMH Edition, 2000.
4. Kanetkar Y., Let us C, BPB Pub., New Delhi, 1999.

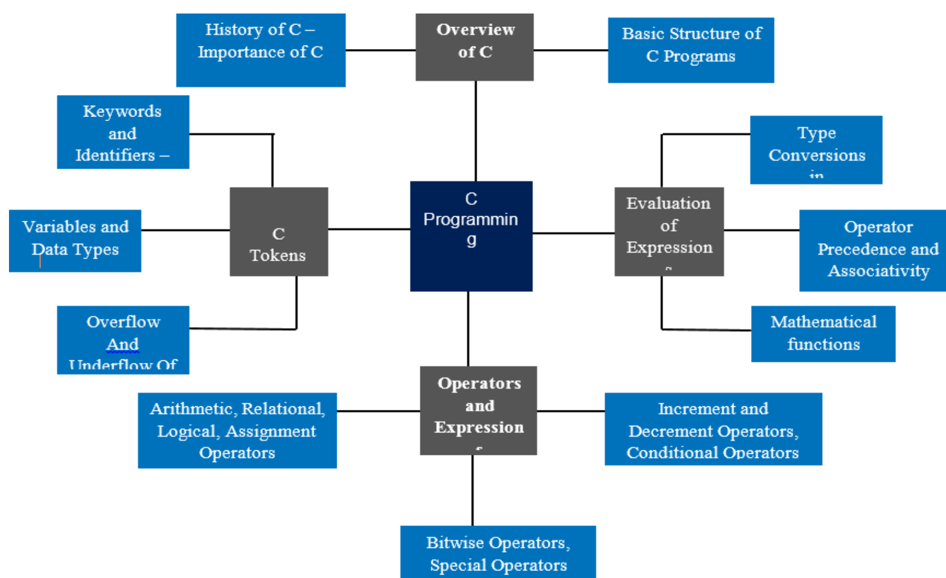
Programming in C

Unit I

Learning Objectives

1. To familiarize with fundamentals of C language
2. To understand the structure of C program
3. To develop algorithm/flowcharts
4. To know applications of C language
5. To identify the different types of operators

Mind Map



Summary

C is a general-purpose, high-level language. C language is mainly used for develop desktop based application. All other programming languages were derived directly or indirectly from C programming concepts. C is often called a middle-level computer language. C language is very often known as a

System Programming Language, because it is used for writing assemblers, compilers, editors and even operating systems. C language is used in wide variety of applications. Practical applications of C language are several, right form writing the operating systems like UNIX, Windows to creating antivirus programs. It was originally developed for the UNIX operating system.

Introduction to C:

Overview of C:

Computer programming (often shortened to **programming**) is a process that leads from an original formulation of a computing problem to executable computer programs. Programming involves activities such as analysis, developing understanding, generating algorithms, verification of requirements of algorithms including their correctness and resources consumption, and implementation (commonly referred to as coding of algorithms in a target programming language. Source code is written in one or more programming languages.

The purpose of programming is to find a sequence of instructions that will automate performing a specific task or solving a given problem. The process of programming thus often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

History of C:

C was written in 1972 by Dennis Ritchie at Bell Laboratories and it was used to implement UNIX on the Dec PDP-11, with 94% of its operating system written in C.

C belongs to the class of procedural languages (also known as prescriptive or imperative languages, as opposed to non-procedural/descriptive/declarative languages). Some other well-known procedural languages include COBOL1, Fortran2 and Pascal. Programs of procedural languages tell the computer how to solve a problem; programs of non-procedural languages, like Smalltalk, specify what is to be solved. Many languages are a mix. For example, Prolog3 is one, though it is considered as largely non-procedural.

C is generally treated as a high-level programming language. However, strictly speaking, it is part high-level and part low-level. Hence, it is actually a low-level language among the high-level ones.

Importance of C:

C is compact. Its core is small, but it is supplemented by a large set of *systems calls* and *libraries*, such as stdio.h, math.h, etc. The ANSI standard for C requires that certain standard libraries be provided in every ANSI C implementation.

C is expressive, portable and efficient. Coupled with its compactness, it is an excellent language for systems programming, time-critical functions, and applications involving close interaction with the operating system or hardware.

On the down side, C has a complicated syntax, which makes it appear confusing at first glance. As it accords the programmer a great deal of flexibility, maturity and discipline are demanded of the programmer. These are real obstacles to beginners. In short, C is baffling to look at, easy to mess up, and tricky to learn.

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

Basic Structure of C:

The structure of c is:

```
/*comments*/
Header files;
main( )
{
Block of statements;
}
```

C Hello World Example

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
#include <stdio.h>

int main()
{
  /* my first program in C */
  printf("Hello, World! \n");

  return 0;
}
```

Let us look various parts of the above program:

1. The first line of the program *#include <stdio.h>* is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
2. The next line *int main()* is the main function where program execution begins.
3. The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
4. The next line *printf(...)* is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
5. The next line **return 0;** terminates main()function and returns the value 0.

Compile & Execute C Program:

Lets look at how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.
2. Save the file as *hello.c*
3. Open a command prompt and go to the directory where you saved the file.
4. Type *gcc hello.c* and press enter to compile your code.
5. If there are no errors in your code the command prompt will take you to the next line and would generate *a.out* executable file.
6. Now, type *a.out* to execute your program.
7. You will be able to see *"Hello World"* printed on the screen

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

Programming Style:

Header file:

A header file is a file with extension **.h** which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

You request the use of a header file in your program by including it, with the C preprocessing directive **#include** like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be very much error-prone and it is not a good idea to copy the content of header file in the source files, specially if we have multiple source file comprising our program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in header files and include that header file wherever it is required.

Include Syntax

Both user and system header files are included using the preprocessing directive **#include**. It has following two forms:

```
#include <file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named file in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive. For example, if you have a header file header.h as follows:

```
char *test (void);
```

and a main program called *program.c* that uses the header file, like this:

```
int x;
#include "header.h"

int main (void)
{
  puts (test ());
}
```

the compiler will see the same token stream as it would if program.c read

```
int x;
char *test (void);

int main (void)
{
  puts (test ());}
```

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and will result an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef HEADER_FILE
```

#define HEADER_FILE

the entire header file file

#endif

This construct is commonly known as a wrapper **#ifndef**. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Basic Syntax:

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens:

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

The individual tokens are:

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

Semicolons ;

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;
```

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminates with the characters */ as shown below:

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. C is a **case sensitive** programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers:

```
mohd       zara abc move_name a_123
myname50   _temp   j     a23b9      retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constant or variable or any other identifier names.

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

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:

```
int age;
```

There must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement:

```
fruit = apples + oranges;  // get the total fruit
```

No whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

Data Types & Variables:

DataTypes:

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) floating-point 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.

The array types and structure types are referred to collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see basic types in the following section, whereas, other types will be covered in the upcoming chapters.

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

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

```
Storage size for int : 4
```

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 theBINARY 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;
}
```

Output:

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

S.N.	Types and Description
1	**Function returns as void** There are various functions in C which do not return value or you can say they return void. A function with no return value has the return type as void. For example **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, **int rand(void);**
3	**Pointers to void**

	A pointer of type void * represents the address of an object, but not its type. For example a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

The void type may not be understood to you at this point, so let us proceed and we will cover these concepts in the upcoming chapters.

Variables:

A variable definition means to tell the compiler where and how much to create the storage for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool or any user-defined object, etc., and **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:

```
type variable_name = value;
```

Some examples are:

```
extern 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'.
```

Constant:

Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*. There are also enumeration constants as well.

The **constants** are treated just like regular variables except that their values cannot be modified after their definition.

Integer literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various type of Integer literals:

```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Floating-point literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character constants

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

Following is the example to show few escape sequence characters:

```
#include <stdio.h>

int main()
{
  printf("Hello\tWorld\n\n");

  return 0;
}
```

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

```
Hello World
```

String literals

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \
```

dear"

"hello, " "d" "ear"

Defining Constants

There are two simple ways in C to define constants:

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

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);

  return 0;
}
```

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

```
value of area : 50
```

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);

   return 0;
}
```

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

```
value of area : 50
```

Storage class:

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. These specifiers precede the type that they modify. There are the following storage classes, which can be used in a C Program

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
   int  mount;
   auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main()
{
  while(count--)
  {
    func();
  }
  return 0;
}
/* function definition */
void func( void )
{
  static int i = 5; /* local static variable */
  i++;

  printf("i is %d and count is %d\n", i, count);
}
```

You may not understand this example at this time because I have used *function* and *global variables*, which I have not explained so far. So for now let us proceed even if you do not understand it completely. When the above code is compiled and executed, it produces the following result:

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main()
{
  count = 5;
  write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void)
{
  printf("count is %d\n", count);
}
```

Here, *extern* keyword is being used to declare *count* in the second file where as it has its definition in the first file, main.c. Now, compile these two files as follows:

```
$gcc main.c support.c
```

This will produce **a.out** executable program, when this program is executed, it produces the following result:

Operators:

Simple answer can be given using expression *4 + 5 is equal to 9*. Here 4 and 5 are called operands and + is called operator. C language supports following type of operators.

- Arithmetic Operators
- Logical (or Relational) Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Lets have a look on all operators one by one.

Arithmetic Operators:

There are following arithmetic operators supported by C language:

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by denumerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Logical (or Relational) Operators:

There are following logical operators supported by C language

Assume variable A holds 10 and variable B holds 20 then:

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

!=	Checks if the value of two operands is 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.
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands is non zero then 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 false.

Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation.

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

There are following Bitwise operators supported by C language

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 eather 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 efect 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

Assignment Operators:

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 assigne 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 operand from the left operand 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

<table>
<tr><td>/=</td><td>Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand</td><td>C /= A is equivalent to C = C / A</td></tr>
<tr><td>%=</td><td>Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand</td><td>C %= A is equivalent to C = C % A</td></tr>
<tr><td><<=</td><td>Left shift AND assignment operator</td><td>C <<= 2 is same as C = C << 2</td></tr>
<tr><td>>>=</td><td>Right shift AND assignment operator</td><td>C >>= 2 is same as C = C >> 2</td></tr>
<tr><td>&=</td><td>Bitwise AND assignment operator</td><td>C &= 2 is same as C = C & 2</td></tr>
<tr><td>^=</td><td>bitwise exclusive OR and assignment operator</td><td>C ^= 2 is same as C = C ^ 2</td></tr>
<tr><td>|=</td><td>bitwise inclusive OR and assignment operator</td><td>C |= 2 is same as C = C | 2</td></tr>
</table>

Short Notes on L-VALUE and R-VALUE:

x = 1; takes the value on the right (e.g. 1) and puts it in the memory referenced by x. Here x and 1 are known as L-VALUES and R-VALUES respectively L-values can be on either side of the assignment operator where as R-values only appear on the right.

So x is an L-value because it can appear on the left as we've just seen, or on the right like this: y = x; However, constants like 1 are R-values because 1 could appear on the right, but 1 = x; is invalid.

Misc Operators

There are few other operators supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is interger, will return 4.
&	Returns the address of an variable.	&a; will give actaul 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

Operators Categories:

All the operators we have discussed above can be categorised into following categories:

- Postfix operators, which follow a single operand.

- Unary prefix operators, which precede a single operand.
- BINARY operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators, which assign a value to a variable.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.

Precedence of C Operators:

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 precedenace than + so it first get 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 precedenace operators will be evaluated first.

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

Questions

1. What are different data types in C?

2. What do you mean by Variable?

3. What are the applications of C programming?

4. Write the basic structure of C PROGRAM?

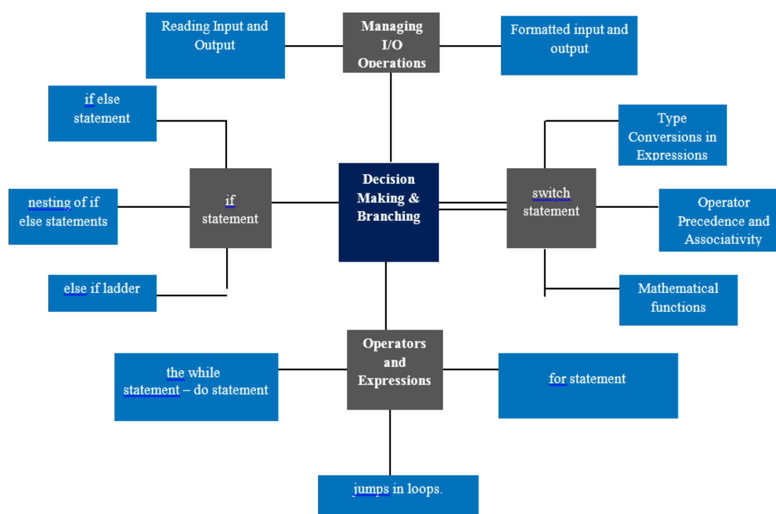
5. Write a C program print the text “Hello World”.

Unit II

Learning Objective

1. To know how to manage I/O statements
2. To know the looping concepts
3. To develop program in decision making statements
4. To know jump in loops

Mind Map



Summary

- Input/output functions are used to accept values into variables and printing them after the processing
- By using getchar() function reading a single character representing the following manner variable-name=getchar().
- In putchar() function it is used to display a character by character.
- It receives a single character to a standard output devices is called monitor.
- In C library function scanf() it is one of the input function. Just like the getchar() function. The scanf() function reads information from the terminal stores with the

particular variable. The formatted data output function can be used to show the output any combination of numerical values such as character and string.

Managing I/O Operations:

Output Statement:

The function printf() is used for formatted output to standard output based on a format specification. The format specification string, along with the data to be output, are the parameters to the printf() function.

Syntax:

printf (format, data1, data2, ...);

In this syntax format is the format specification string. This string contains, for each variable to be output, a specification beginning with the symbol % followed by a character called the conversion character.

Example:

printf ("%c", data1);

The character specified after % is called a conversion character because it allows one data type to be converted to another type and printed.

See the following table conversion character and their meanings.

Conversion Character	Meaning
d	The data is converted to decimal (integer)
c	The data is taken as a character.
s	The data is a string and character from the string , are printed until a NULL, character is reached.
f	The data is output as float or double with a default Precision 6.
Symbols	**Meaning**
\n	For new line (linefeed return)
\t	For tab space (equivalent of 8 spaces)

Example

printf("%c\n",data1);

The format specification string may also have text.

Example

printf (“Character is:”%c\n”, data1);

The text "Character is:" is printed out along with the value of data1.

Example with program

```
#include<stdio.h>
#include<conio.h>
main()
{
char alphabh="a";
int number1= 55;
float number2=22.34;
printf(“char= %c\n”,alphabh);
printf(“int= %d\n”,number1);
printf(“float= %f\n”,number2);
getch();
clrscr();
retrun 0;
}
```

Output :

char =a
int= 55
flaot=22.340000

Input Statement:

The function scanf() is used for formatted input from standard input and provides many of the conversion facilities of the function printf().

Syntax

scanf (format, num1, num2,……);
The function scnaf() reads and converts characters from the standards input depending on the format specification string and stores the input in memory locations represented by the other arguments (num1, num2,….).

For Example:

scanf(“ %c %d”,&Name, &Roll No);

Note: the data names are listed as &Name and &Roll No instead of Name and Roll No respectively. This is how data names are specified in a scnaf() function. In case of string type data names, the data name is not preceded by the character &.

Example with program

Write a function to accept and display the element number and the weight of a proton. The element number is an integer and weight is fractional.

Solve here:

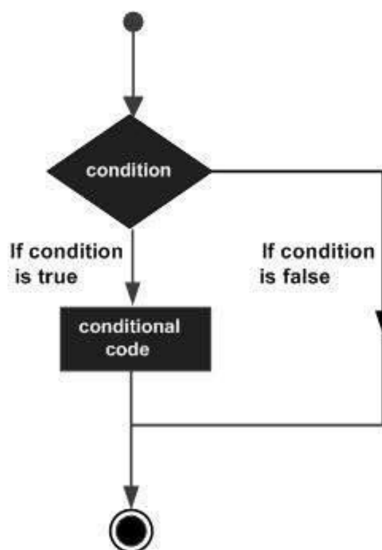
```
#include<stdio.h>
#include<conio.h>
main()
{
Int e_num;
Float e_wt;
printf (“Enter the Element No. and Weight of a Proton\n”);
scanf (“%d %f”,&e_num, &e_wt);
printf (“The Element No.is:”,e_num);
printf (“The Weight of a Proton is: %f\n”, e_wt);
getch();
return 0;
}
```

Control Statement:

Decision making and branching statements:

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.

Following is the general form of a typical decision making structure found in most of the programming languages:



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.

C programming language provides following types of decision making statements. Click the following links to check their detail.

Statement	Description
if statement	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if**or **else if** statement(s).
switch statement	A **switch** statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one **switch** statement inside another **switch**statement(s).

The ? : Operator:

We have covered **conditional operator ? :** in previous chapter which 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.

If statement:

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

Syntax:

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

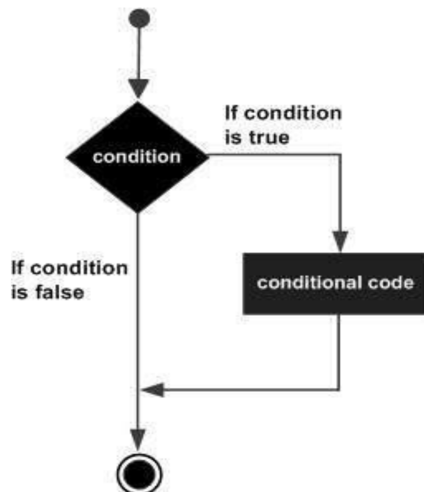
```
if(boolean_expression)
{
```

```
   /* 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.

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.

Flow Diagram:



Example:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;

   /* check the boolean condition using if statement */
   if( a < 20 )
   {
      /* if condition is true then print the following */
      printf("a is less than 20\n" );
   }
   printf("value of a is : %d\n", a);

   return 0;
}
```

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

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

If…Else statement:

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

Syntax:

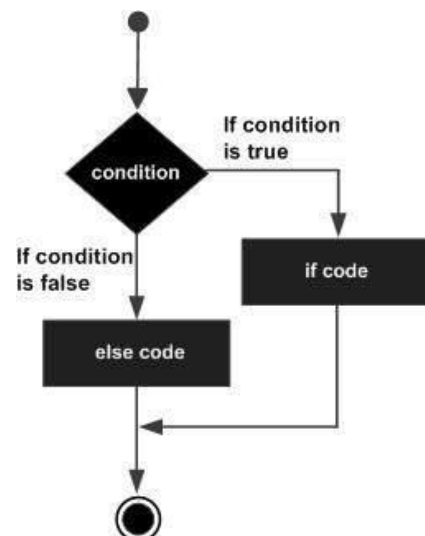
The syntax of an **if...else** statement in C programming language is:

```
if(boolean_expression)
{
  /* statement(s) will execute if the boolean expression is true */
}
else
{
 /* 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.

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.

Flow Diagram:



Example:

```
#include <stdio.h>

int main ()
{
  /* local variable definition */
```

```
   int a = 100;

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

   return 0;
}
```

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

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

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind:

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

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

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
   /* Executes when the boolean expression 3 is true */
}
else
```

```
{
  /* executes when the none of the above condition is true */
}
```

Example:

```
#include <stdio.h>

int main ()
{
  /* local variable definition */
  int a = 100;

  /* check the boolean condition */
  if( a == 10 )
  {
     /* if condition is true then print the following */
     printf("Value of a is 10\n" );
  }
  else if( a == 20 )
  {
     /* if else if condition is true */
     printf("Value of a is 20\n" );
  }
  else if( a == 30 )
  {
     /* if else if condition is true  */
     printf("Value of a is 30\n" );
  }
  else
  {
     /* if none of the conditions is true */
     printf("None of the values is matching\n" );
  }
  printf("Exact value of a is: %d\n", a );

  return 0;
}
```

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

```
None of the values is matching
Exact value of a is: 100
```

Nested If Statement:

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 ()
{
  /* local variable definition */
  int a = 100;
  int b = 200;

  /* check the boolean condition */
  if( a == 100 )
  {
     /* if condition is true then check the following */
     if( b == 200 )
     {
       /* if condition is true then print 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 );

  return 0;
}
```

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
```

Switch statements:

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**.

Syntax:

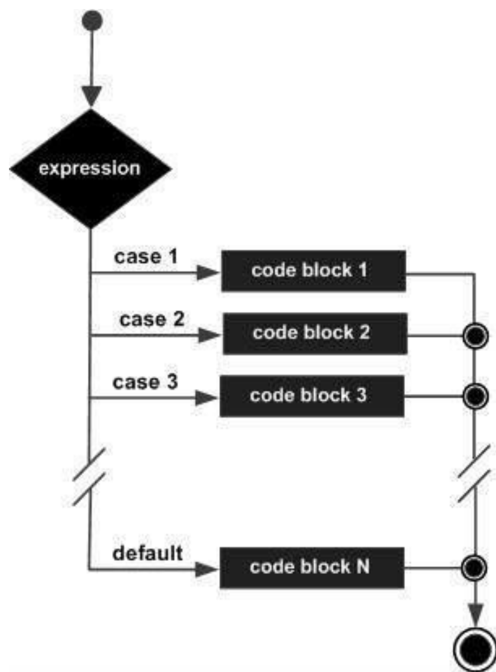
The syntax for a **switch** statement in C programming language is as follows:

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

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

The following rules apply to a **switch** statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram:



Example:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   char grade = 'B';

   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 );

   return 0;
}
```

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

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

Nested Switch satements:

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax:

The syntax for a **nested switch** statement is as follows:

```
switch(ch1) {
   case 'A':
      printf("This A is part of outer switch" );
      switch(ch2) {
         case 'A':
            printf("This A is part of inner switch" );
            break;
         case 'B': /* case code */
      }
      break;
   case 'B': /* case code */
}
```

Example:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   switch(a) {
      case 100:
         printf("This is part of outer switch\n", a );
         switch(b) {
            case 200:
               printf("This is part of inner switch\n", a );
         }
   }
   printf("Exact value of a is : %d\n", a );
   printf("Exact value of b is : %d\n", b );

   return 0;
}
```

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

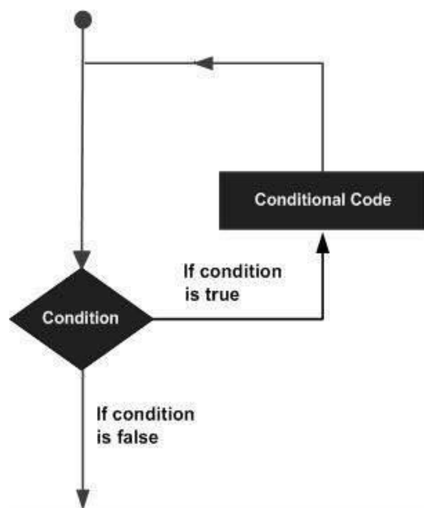
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200

Looping statements:

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.

Programming languages provide various control structures that allow for more complicated execution paths.

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:



C programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

While loop:

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

Syntax:

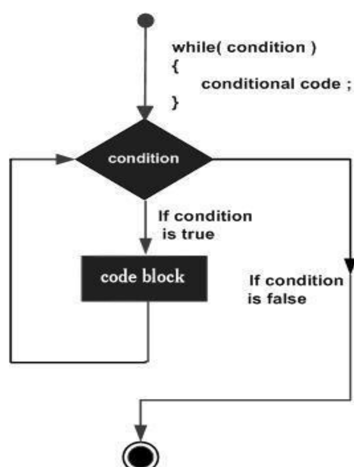
The syntax of a **while** loop in C programming language is:

```
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.

Example:

```
#include <stdio.h>

int main ()
{
  /* local variable definition */
  int a = 10;

  /* while loop execution */
  while( a < 20 )
  {
    printf("value of a: %d\n", a);
    a++;
  }

  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
```

Do..while loop:

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:

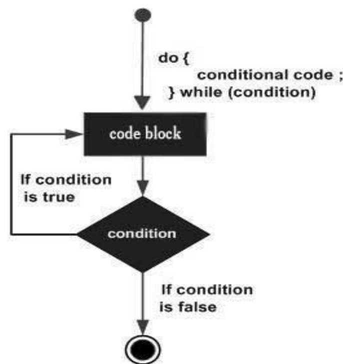
```
do
{
  statement(s);
```

```
}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 ()
{
  /* local variable definition */
  int a = 10;
  /* do loop execution */
  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
```

Nested loop:

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);
  }
  statement(s);
}
```

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:

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int i, j;

   for(i=2; i<100; i++) {
      for(j=2; j <= (i/j); j++)
        if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime\n", i);
   }

   return 0;
}
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

for loop:

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:

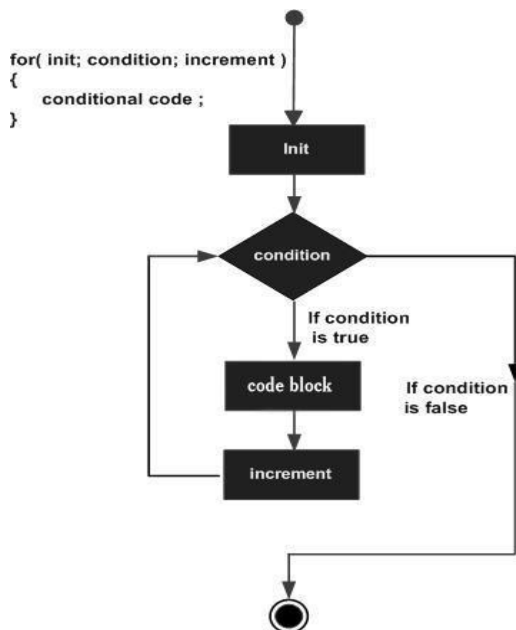
The syntax of a **for** loop in C programming language is:

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

Here is the flow of control in a for loop:

1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example:

```
#include <stdio.h>
int main ()
{
  /* for loop execution */
  for( int a = 10; a < 20; a = a + 1 )
  {
    printf("value of a: %d\n", a);
```

```
  }
  return 0;
}
```

Output:

```
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
```

Jump in loops:

The Infinite Loop:

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");
  }
  return 0;
}
```

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: Infinite loop can be terminated by pressing Ctrl + C keys.

Control Statement	Description
break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

Questions

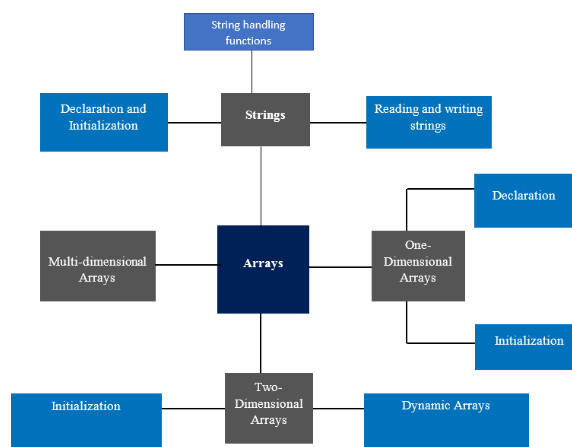
1. Write the syntax of using getchar() and putchar() function.
2. What is the purpose of output statement?
3. Write the general form of switch statement.
4. Write a program to generate fibonnacci series within limit.
5.Explain in detail about formatted input and output functions.

Unit III

Learning Objective

1. Students able to understand different types of Arrays
2. Develop own programs using Arrays
3. Define and Describe Two Dimensional Arrays
4. Know the concepts of Multidimensional array and Strings

Mind Map



Summary

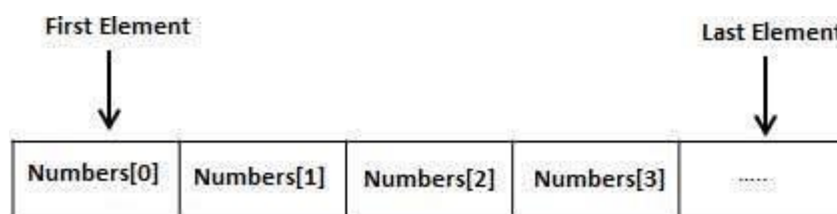
- An array is a group of elements (data items) that have a common characteristics (Ex: numbeic data, character data etc.,) and share a common name.
- Arrays whose elements are specified by a single subscript are called one dimensional or single dimensional array.
- The Subscript used to declare an array is sometimes called a dimension and declaration for it sometimes called the array referred to as dimensioning.
- An individual element in an array can be referred to by means of the subscript.
- Arrays whose elements are specified by two subscripts are referred as two-dimensional array or double-dimensional arrays.

Arrays:

C programming language provides a data structure called **the 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.



One Dimensional Array:

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 10-element array called **balance** of type double, use this statement:

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

Now *balance* is avariable array which is sufficient to hold upto 10 double numbers.

Initializing Arrays

Initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 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 [ ].

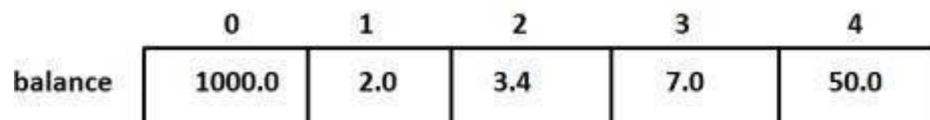
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

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

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:



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;
}
```

Output:

```
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
```

Two-dimensional Array:

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

Two-Dimensional Arrays:

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-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

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.

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};
```

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:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
```

a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Multi Dimensional Array

The basic syntax or, the declaration of multi-dimensional array in C Programming is

Data_Type Arr_Name[Tables][Row_Size][Column_Size]

- Data_type: It will decide the type of elements it will accept. For example, If we want to store integer values then we declare the Data Type as int, If we want to store Float values then we declare the Data Type as float etc
- Arr_Name: This is the name you want to give it to Multi Dimensional array in C.
- Tables: It will decide the number of tables one can accept. Two Dimensional is always a single table with rows and columns. In contrast, Multi Dimensional array in C is more than 1 table with rows and columns.
- Row_Size: Number of Row elements it can store. For example, Row_Size =10, the array will have 10 rows.
- Column_Size: Number of Column elements it can store. For example, Column_Size = 8, it will have 8 Columns.

Calculate the maximum number of elements in a Three Dimensional using: [Tables] * [Row_Size] * [Column_Size]
For Example,
int Employees[2][4][3];

Initialization

Initializing Multi Dimensional Array as follows:

```
int Employees[2][4][3] = { { {10, 20, 30}, {15, 25, 35}, {22, 44, 66}, {33, 55, 77} },

                         { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }

                        };
```

Dynamic arrays:

A dynamic array is quite similar to a regular array, but its size is modifiable during program runtime. Dynamic Array elements occupy a contiguous block of memory.

Once an array has been created, its size cannot be changed. However, a dynamic array is different. A dynamic array can expand its size even after it has been filled.

During the creation of an array, it is allocated a predetermined amount of memory. This is not the case with a dynamic array as it grows its memory size by a certain factor when there is a need.

Features in C that enables to implement a own dynamic array are:

The malloc( ) function:

The malloc() function only alocate memory cells for one variable:

malloc(nBytes) will allocates a block of memory cells of at least nBytes bytes that us suitably aligned for any usage.

The calloc( )function:

To allocate memory cells for N consecutive variable (= array), you must use this function:

calloc(nElems, nBytes)

Example:

calloc(10, 8)

// Allocate space for 10 elements of size 8 (= array of 10 double)

The calloc() function allocates space for an array of nElems elements of size nBytes.

The free( ) function:

The allocated space will alos be initialized to zeros.

Strings:

String:

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++:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

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;
}
```

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

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

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 ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
```

```
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):  %s\n", str1 );

   /* total lenghth of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

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

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

You can find a complete list of c string related functions in C Standard Library.

Questions

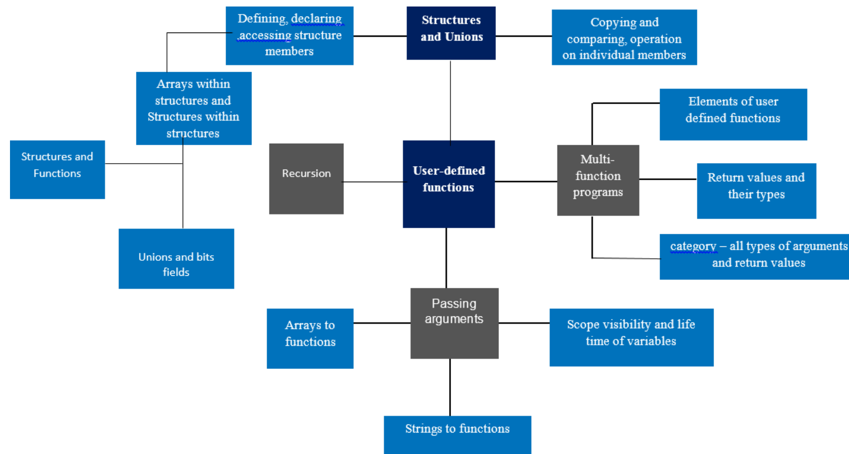
1. Define an array.
2. Explain single dimensional arrays with examples.
3. Explain two dimensional arrays with examples.
4. Write about the rules to be followed for array initialization.
5. Write a program to display array of names.
6. Write the syntax of initializing one-dimensional array at run-time?
7. Discuss String handling functions.

Unit IV

Learning Objective

1. Discuss for user-defined functions
2. Discuss about multi-function program
3. Define and function and function declaration
4. Discuss about elements of user-defined function
5. Discuss return values and their types

Mind Map



Summary

- A function is defined as a self-contained program which is written for the purpose of accomplishing some task.
- A User-Defined Function, or UDF, is a function provided by the user of a program. C functions can be classified into two categories, namely, library functions and user-defined functions.
- The function definition is an independent program module that is specially written to implement the requirements of the function.
- The return statement is used to return the control from the calling function to the next statement of the called portion of the program.
- A function can be accessed (i.e., called) by specifying its name, followed by a list of arguments enclosed in parenthesis and separated by commas.

Functions:

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure, etc.

Defining a Function:

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{
  body of the function
}
```

A function definition in C programming language consists of a *function header* and a*function body*. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
  /* local variable declaration */
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Function Declarations:

A function **declaration** 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 max(), following is the function declaration:

```
int max(int num1, int num2);
```

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

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main ()
{
  /* local variable definition */
  int a = 100;
  int b = 200;
  int ret;

  /* calling a function to get max value */
  ret = max(a, b);

  printf( "Max value is : %d\n", ret );
```

```
   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

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:

Call Type	Description
Call by value	This method 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.
Call by reference	This method 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 argument.

By default, C uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Scope, Visibility and Lifetime of Variables:

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable can not be accessed. There are three places where variables can be declared in C programming language:

1. Inside a function or a block which is called **local** variables,
2. Outside of all functions which is called **global** variables.
3. In the definition of function parameters which is called **formal** parameters.

Let us explain what are **local** and **global** variables and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables. Here all the variables a, b and c are local to main() function.

```
#include <stdio.h>
int main ()
{
 /* local variable declaration */
 int a, b;
 int c;

 /* actual initialization */
 a = 10;
 b = 20;
 c = a + b;
 printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
 return 0;
}
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <stdio.h>
/* global variable declaration */
int g;
int main ()
{
  /* local variable declaration */
  int a, b;
  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;
  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
  return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. Following is an example:

```
#include <stdio.h>
/* global variable declaration */
int g = 20;
int main ()
{
  /* local variable declaration */
  int g = 10;
  printf ("value of g = %d\n", g);
  return 0;
}
```

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

```
value of g = 10
```

Formal Parameters

Function parameters, formal parameters, are treated as local variables with-in that function and they will take preference over the global variables. Following is an example:

```
#include <stdio.h>
 /* global variable declaration */
int a = 20;
 int main ()
{
```

```
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;
  printf ("value of a in main() = %d\n", a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n", c);
 return 0;
}
/* function to add two integers */
int sum(int a, int b)
{
   printf ("value of a in sum() = %d\n", a);
   printf ("value of b in sum() = %d\n", b);
   return a + b;
}
```

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

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows:

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0

pointer	NULL

It is a good programming practice to initialize variables properly otherwise, your program may produce unexpected results because uninitialized variables will take some garbage value already available at its memory location.

Structures and Unions:

Structure:

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.

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

Defining a Structure

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:

```
struct [structure tag]
{
  member definition;
  member definition;
  ...
  member definition;
} [one or more structure variables];
```

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;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

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

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

int main( )
{
  struct Books Book1;        /* Declare Book1 of type Book */
  struct Books Book2;        /* Declare Book2 of type Book */

  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;

  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;

  /* print Book1 info */
  printf( "Book 1 title : %s\n", Book1.title);
  printf( "Book 1 author : %s\n", Book1.author);
  printf( "Book 1 subject : %s\n", Book1.subject);
  printf( "Book 1 book_id : %d\n", Book1.book_id);

  /* print Book2 info */
  printf( "Book 2 title : %s\n", Book2.title);
  printf( "Book 2 author : %s\n", Book2.author);
  printf( "Book 2 subject : %s\n", Book2.subject);
  printf( "Book 2 book_id : %d\n", Book2.book_id);

  return 0;
}
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

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

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

/* function declaration */
void printBook( struct Books book );
int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printBook( Book1 );

   /* Print Book2 info */
   printBook( Book2 );
```

```
   return 0;
}
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

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

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

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

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

```
};

/* function declaration */
void printBook( struct Books *book );
int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info by passing address of Book1 */
   printBook( &Book1 );

   /* print Book2 info by passing address of Book2 */
   printBook( &Book2 );

   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

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

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Union:

A **union** is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

Defining a Union

To define a union, you must use the **union** statement in very similar was as you did while defining structure. The union statement defines a new data type, with more than one member for your program. The format of the union statement is as follows:

```
union [union tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

The **union 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 union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data which has the three members i, f, and str:

```
union Data
{
   int i;
   float f;
   char str[20];
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

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

union Data
{
   int i;
   float f;
   char str[20];
};
```

```
int main( )
{
   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

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

```
Memory size occupied by data : 20
```

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type. Following is the example to explain usage of union:

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

union Data
{
   int i;
   float f;
   char str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

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

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that values of **i** and **f** members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value if **str** member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

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

union Data
{
   int i;
   float f;
   char str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

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

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

Size of Structures

In C language, sizeof() operator is used to calculate the size of structure, variables, pointers or data types, data types could be pre-defined or user-defined.

Example:

```
#include <stdio.h>

struct student {  // Declaring a structure named "student"
   int rollno;
   char name[16];
```

```
  int marks;
};

int main() {
  struct student s;  // Declaring a structure type data named "s"
  int size = sizeof(s);
  printf("Size of Structure : %d", size);

  return 0;
}
```

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in. E.g. 9 bit integers.

C allows us do this in a structure definition by putting :bit length after the variable. For example:

```
struct packed_struct {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4 bit type and a 9 bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

Questions

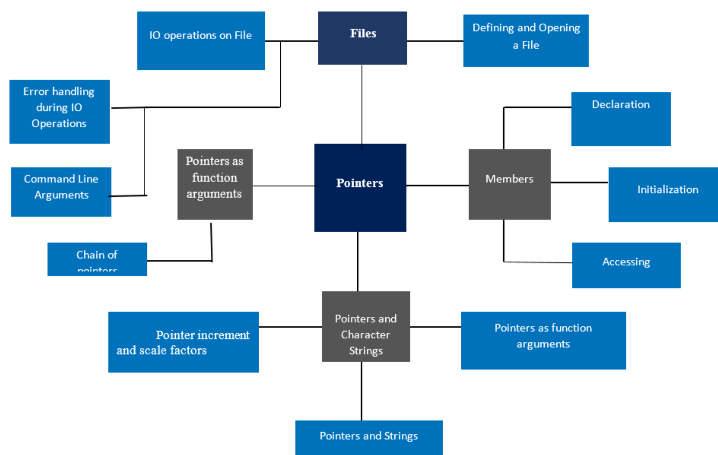
1. What is the need for functions?
2. What is a function?
3. Explain the general form of defining a function.
4. What are formal arguments?
5. What are actual arguments?
6. How the structures members are accessed.
7. What is the difference between structure and union.
8. Define bit field.

Unit V

Learning Objective

1. To differentiate the variables and Pointers variables
2. To Know how to define and initialize Pointer Variables
3. Understanding Accessing Variable through its Pointers
4. To identify the pointes as function arguments and string
5. To understand the concepts of files
6. To know how to handle errors and understand command line arguments

Mind Map



Summary

- A pointer is a variable, which represents the memory location (not the value) of a data items, such as a variable or an array element.
- The ampersand operator (&) gives the address of a variable.
- The three values that can be used to initialize the pointer are zero, null and address.
- The pointer that has not been initialized is referred to as the dangling pointer.
- Only an address of a variable can be stored in a pointer variable.
- Do not store the address of a variable of one type into a pointer variable of another type.
- The value of a variable cannot be assigned to a pointer variable.
- A pointer variable contains garbage until it is initialized.

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.

Declaring, Initializing and accessing pointer members:

There are few important operations, which we will do with the help of pointers very frequently. (a) 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:

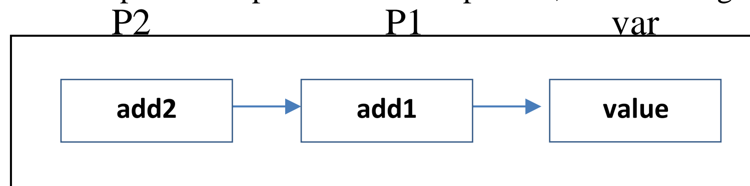
```
#include <stdio.h>

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

Chain of Pointers

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers.



Here, the pointer variable p2 contains the add of the pointer variable P_1, which points to the location that contains the desired value. This is known as multiple indirection's.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.

Ex:– int□□□p2;

This declaration tells the compiler that p2 is a pointer to a pointer of int type.

Ex:–

```
#include
main()
{
Int x, □p1, □□p2;
x=100;
p1=&x;
p2=&p1;
printf("%d", **p2);
}
```

This code will display the value 100. Here, P1 is declared as a pointer to an integer & p2 as a declared as a pointer to an integer & p2 as a pointer to a pointer to an integer.

Pointer increments and scale factors

Pointers can be incremented like.

p1=p1+1;

p1=p2+2; &so on.

The expression like p1++;

Will cause the pointer p1 to point to the next value of its type. For ex. If p1 is an integer pointer with an initial value, say 2800, then after with an initial value, the value of p1 will be 2902, & not 2801.

i.e., when we increment a pointer, its value is incremented by the length of the data type that it points to . This length is called the scale factor.

The no of bytes used to store various data types depends on the system & can be found by making use of the size of operator.

Character 1 byte
Integers 2 bytes
Floats 4 bytes
Long integers 4 bytes
Double 8 bytes

File I/o:

A file represents a sequence of bytes, does not matter if it is a text file or binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through important calls for the file management.

Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

```
FILE *fopen( const char * filename, const char * mode );
```

Here, **filename** is string literal, which you will use to name your file and access **mode** can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.

w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is:

```
int fclose( FILE *fp );
```

The **fclose( )** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.

Writing a File

Following is the simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example:

Make sure you have **/tmp** directory available, if its not then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("/tmp/test.txt", "w+");
  fprintf(fp, "This is testing for fprintf...\n");
```

```
   fputs("This is testing for fputs...\n", fp);
   fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

Reading a File

Following is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns **EOF**. The following functions allow you to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>

main()
{
   FILE *fp;
   char buff[255];

   fp = fopen("/tmp/test.txt", "r");
   fscanf(fp, "%s", buff);
   printf("1 : %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("2: %s\n", buff );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", buff );
   fclose(fp);
}
```

Output

```
1 : This
2: is testing for fprintf...
```

3: This is testing for fputs...

First **fscanf()** method read just **This** because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.

Binary I/O Functions

There are following two functions, which can be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
          size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
          size_t number_of_elements, FILE *a_file);
```

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

Error Handling in C

C language does not provide any direct support for error handling. However a few methods and variables defined in error.h header file can be used to point out error using the return statement in a function. In C language, a function returns -1 or NULL value in case of any error and a global variable errno is set with the error code. So the return value can be used to check error while programming.

```
#include  <stdio.h>
#include <errno.h>
#include <string.h>
int main ()
{
   FILE *fp;
    /* If a file, which does not exists, is opened,
       we will get an error
   */
   fp = fopen("IWillReturnError.txt", "r");
    printf("Value of errno: %d\n ", errno);
   printf("The error message is : %s\n", strerror(errno));
   perror("Message from perror");
    return 0;
}
```

Command-Line Arguments

Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system.

C programming language gives the programmer the provision to add parameters or arguments inside the main function to reduce the length of the code. These arguments are called command line arguments in C.

Components of command line arguments

Generally, 2 parameters are passed into the main function:

1. Number of command line arguments
2. The list of command line arguments

The basic syntax is:

```
int main( int argc, char *argv[] )
{
----// body of the main function
}
```

Another way to implement command line arguments is:

```
int main( int argc, char **argv[] )
{
-----// body of the main function
}
```

i) **argc:** It refers to "argument count". It is the first parameter that we use to store the number of command line arguments. It is important to note that the value of argc should be greater than or equal to 0.

ii) **agrv:** It refers to "argument vector". It is basically an array of character pointer which we use to list all the command line arguments.

Example:

```
#include<stdio.h>
int main(int argc, char** argv)
{
printf("Command line arguments!\n\n");
int i;
printf("The number of arguments are: %d\n",argc);
printf("The arguments are:");
for ( i = 0; i < argc; i++)
{
printf("%s\n", argv[i]);
}
return 0;
}
```

Questions

1. How to find the address of a variable?
2. What is chain of Pointers?
3. How will C errors are handled?
4. Write the different modes of file access.
5. Write the general form of file opening and closing.

6. What is purpose of argv( ) and argc( )?