**KARPAGAM ACADEMY OF HIGHER EDUCATION**
(Deemed to be University Established under Section 3 of UGC Act 1956)
Pollachi Main Road, Eacharani Post, Coimbatore-641 021

**DEPARTMENT OF COMPUTER APPLICATIONS**

**Semester – I**

**18CAU101 PROGRAMMING FUNDAMENTALS USING C / C++ 4H – 4C**

---

**Instruction Hours / week: L: 4 T: 0 P: 0** **Marks:** Int.: **40** Ext.: **60** Total: **100**

**Course Objective**: This course provides student with a comprehensive study of the C and C++ programming language. Classroom lectures stress the strength of C and C++, which provide programmers with the means of writing efficient, maintainable and portable code.

**Learning Outcomes**: By the end of this course, students should understand the concept of a program (i.e., a computer following a series of instructions). Understand the concept of a loop – that is, a series of statements which is written once but executed repeatedly- and how to use it in a programming language. Be able to break a large problem into smaller parts, writing each part as a module or function. Understand the concept of a program in a high-level language being translated by a compiler into machine language program and then executed.

**Unit-I**

**Introduction to C and C++:** History of C and C++, Overview of Procedural Programming and Object-Orientation Programming, Using main() function, Compiling and Executing Simple Programs in C++.
**Data Types, Variables, Constants, Operators and Basic I/O:** Declaring, Defining and Initializing Variables, Scope of Variables, Using Named Constants, Keywords, Data Types, Casting of Data Types, Operators (Arithmetic, Logical and Bitwise), Using Comments in programs, Character I/O (getc, getchar, putc, putchar etc), Formatted and Console I/O (printf(), scanf(), cin, cout), Using Basic Header Files (stdio.h, iostream.h, conio.h etc).
**Expressions, Conditional Statements and Iterative Statements:** Simple Expressions in C++ (including Unary Operator Expressions, Binary Operator Expressions), Understanding Operators Precedence in Expressions, Conditional Statements (if construct, switch-case construct), Understanding syntax and utility of Iterative Statements (while, do-while, and for loops), Use of break and continue in Loops, Using Nested Statements (Conditional as well as Iterative)

**Unit-II**

**Functions and Arrays:** Utility of functions, Call by Value, Call by Reference, Functions returning value, Void functions, Inline Functions, Return data type of functions, Functions parameters, Differentiating between Declaration and Definition of Functions, Command Line Arguments/Parameters in Functions, Functions with variable number of Arguments.
Creating and Using One Dimensional Arrays ( Declaring and Defining an Array, Initializing an Array, accessing individual elements in an Array, Manipulating array elements using loops), Use Various types of arrays (integer, float and character arrays / Strings) Two-dimensional Arrays (Declaring, Defining and Initializing Two Dimensional Array, Working with Rows and Columns), Introduction to Multi-dimensional arrays.

**Unit-III**

**Derived Data Types (Structures and Unions):** Understanding utility of structures and unions, Declaring, initializing and using simple structures and unions, Manipulating individual members of structures and unions, Array of

Structures, Individual data members as structures, Passing and returning structures from functions, Structure with union as members, Union with structures as members.

**Pointers and References in C++:** Understanding a Pointer Variable, Simple use of Pointers (Declaring and Dereferencing Pointers to simple variables), Pointers to Pointers, Pointers to structures, Problems with Pointers, Passing pointers as function arguments, Returning a pointer from a function, using arrays as pointers, Passing arrays to functions. Pointers vs. References, Declaring and initializing references, using references as function arguments and function return values

**Unit-IV**

**Memory Allocation in C++:** Differentiating between static and dynamic memory allocation, use of malloc, calloc and free functions, use of new and delete operators, storage of variables in static and dynamic memory allocation.

**File I/O, Preprocessor Directives:** Opening and closing a file (use of fstream header file, ifstream, ofstream and fstream classes), Reading and writing Text Files, Using put(), get(), read() and write() functions, Random access in files, Understanding the Preprocessor Directives (#include, #define, #error, #if, #else, #elif, #endif, #ifdef, #ifndef and #undef), Macros.

**Unit-V**

**Using Classes in C++:** Principles of Object-Oriented Programming, Defining & Using Classes, Class Constructors, Constructor Overloading, Function overloading in classes, Class Variables &Functions, Objects as parameters, Specifying the Protected and Private access, Copy Constructors, Overview of Template classes and their use.

**Overview of Function Overloading and Operator Overloading:** Need of Overloading functions and operators, Overloading functions by number and type of arguments, Looking at an operator as a function call, Overloading Operators (including assignment operators, unary operators)

**Inheritance, Polymorphism and Exception Handling: Introduction** to Inheritance (Multi-Level Inheritance, Multiple Inheritance), Polymorphism (Virtual Functions, Pure Virtual Functions), Basics Exceptional Handling (using catch and throw, multiple catch statements), Catching all exceptions, Restricting exceptions, Rethrowing exceptions.

**Suggested Readings**


1. Balaguruswamy,E.,(2012). *Object Oriented Programming with C++.* Tata McGraw-Hill Education.
2. Bjarne Stroustroup, (2014). *Programming -- Principles and Practice using C++.* (2nd ed.). Addison-Wesley.
3. Bjarne Stroustrup, (2013). *The C++ Programming Language*, (4th ed.). Addison-Wesley.
4. Harry, H. Chaudhary,(2014). *Head First C++ Programming*: *The Definitive Beginner's Guide.* CreateSpace Independent Publishing Platform.
5. Herbtz Schildt, (2012). *C++: The Complete Reference.* (5th ed.). McGraw-Hill Osborne Media
6. Paul Deitel, Harvey Deitel, (2011).*C++ How to Program.* (8th ed.). Prentice Hall.
7. Stanley B. Lippman, JoseeLajoie, Barbara E. Moo,(2012). *C++ Primer.* (5th ed.) Addison-Wesley.


**Websites**

1. http://www.cs.cf.ac.uk/Dave/C/CE.html
2. http://www2.its.strath.ac.uk/courses/c/
3. http://www.iu.hio.no/~mark/CTutorial/CTutorial.html
4. http://www.cplusplus.com/doc/tutorial/
5. www.cplusplus.com/
6. www.cppreference.com/

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University Established under Section 3 of UGC Act 1956)**
**Pollachi Main Road, Eacharani Post, Coimbatore-641 021**

**DEPARTMENT OF COMPUTER APPLICATIONS**

**Semester – I**

**PROGRAMMING FUNDAMENTALS USING C/C++(18CAU101)**

**LESSON PLAN**

**UNIT-1**

| S.NO | DURATION | TOPICS | SUPPORTED MATERIALS |
|---|---|---|---|
| 1. | 1 | INTRODUCTION TO C/C++:HISTORY OF C/C++,OVERVIEW OF PROCEDURE ORIENTED PROGRAMMING & OBJECT ORIENTED PROGRAMMING | W1, T1:5-13 |
| 2. | 1 | USING MAIN ()FUNCTION,COMPILING,AND EXECUTING SIMPLE PROGRAM IN C++. | W1, T1:54-60 |
| 3. | 1 | DECLARING,DEFINING AND INITIALIZAING VARIABLE,SCOPE OF VARIABLE | T1:33-45 |
| 4. | 1 | USING NAMED CONSTANT,KEYWORDS,DATA TYPE,CASTING OF DATA TYPES,OPERATOR. | T1:25-30 |
| 5. | 1 | COMMENTS IN PROGRAM,CHARACTER P/O,USING HEAD FILE. | T1:30-40 R1:45-60 |
| 6. | 1 | SIMPLE EXPRESSION IN C++ | T1:45-46,101 |
| 7. | 1 | OPERATOR PRECEDENCE IN EXPRESSION,CONDITION STATEMENT. | T1:50-70,101 |
| 8. | 1 | UNDERSTANDING THE SYNTAX,UTILITY OF INERATION STATEMENTS,USE OF BREAK,CONTINUE IN LOOP,NESTED STATEMENTS | T1:80-95,101. R1:75,80 |
| 9. | 1 | RECAPITULATION OF IMPORTANT QUESTIONS | |
| | | TOTAL HOURS:9 HOURS | |

**TEXTBOOK :**E:BALAGURUSAMY(2008).OBJECT ORIENTED PROGRAMMING WITH C++,TATA MC GRAW HILL EDUCATION.

**REFERENCE BOOKS:**BJARNEE STROUSTRUP,(2013).THE C++ PROGRAMMING LANGUAGE ,(2ND ED.).ADDITION – WELSEY.

**WEBSITE:**WWW.geeksforgreeks.org/

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University Established under Section 3 of UGC Act 1956)**
**Pollachi Main Road, Eacharani Post, Coimbatore-641 021**

**DEPARTMENT OF COMPUTER APPLICATIONS**

**Semester – I**

**PROGRAMMING FUNDAMENTALS USING C/C++(18CAU101)**

**LESSON PLAN**

**UNIT-II**

| S.NO | DURATION | TOPICS | SUPPORTED MATERIALS |
|---|---|---|---|
| 1. | 1 | FUNCTIONS &ARRAY:UTILITY OF FUNCTION,CALL BY VALUE,CALL BY REFERENCE,FUNCTION RETURNING VALUE,VOID FUNCTION,INLINE FUNCTION,RETURN DATA TYPE OF FUNCTION | T1:77-84 |
| 2. | 1 | FUNCTION PARAMETER,DIFFERENTIATING DECLARATION & FUNCTION DEFINITION | T1:84-95, |
| 3. | 1 | COMMAND LINE ARGUMENT,PARAMETER IN FUNCTION WITH VARIABLE NUMBER OF ARGUMENT | W1,T1:95-100 |
| 4. | 1 | CREATING & USING 1DIMENSIONAL ARRAY,DECLARING & DEFINING AN ARRAY,INITIALIZING AN ARRAY | T1:119-125 |
| 5. | 1 | ACCESSING INDIVIDUAL ELEMENTS IN AN ARRAY,MANIPULATING ARRAY,ELEMENTS USING LOOP | T1:125-130 |
| 6. | 1 | USE OF VARIOUS TYPES OF ARRAY | T1:130,131 |
| 7. | 1 | 2 DIMENSIONAL ARRAY | T1:131,132 |
| 8. | 1 | INTRODUCTION TO MULTI DIMENSIONAL ARRAY | T1:132,133 |
| 9. | 1 | RECAPITULATION OF IMPORTANT QUESTIONS | |
| | | TOTAL HOURS:9HOURS | |

**TEXTBOOK :**E:BALAGURUSAMY(2008).OBJECT ORIENTED PROGRAMMING WITH C++,TATA MC GRAW HILL EDUCATION.

**REFERENCE BOOKS:**BJARNEE STROUSTRUP,(2013).THE C++ PROGRAMMING LANGUAGE ,(4TH ED.).ADDITION – WELSEY.

**WEBSITE:**WWW.cplusplus.com/

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University Established under Section 3 of UGC Act 1956)**
**Pollachi Main Road, Eacharani Post, Coimbatore-641 021**

**DEPARTMENT OF COMPUTER APPLICATIONS**

**Semester – I**

**PROGRAMMING FUNDAMENTALS USINGC/C++(18CAU101)**

**LESSON PLAN**

**UNIT-III**

| S.NO | DURATION | TOPICS | SUPPORTED MATERIALS |
|---|---|---|---|
| 1. | 1 | Understanding utility of structures and unions ,Declaring,initializing and using simple structures and unions | T1:140-145 |
| 2. | 1 | Manipulating individual members of structures and unions | T1:145,146 |
| 3. | 1 | Array of Structures, Individual data members as structures | T1:146,W1 |
| 4. | 1 | Passing and returning structures from functions, Structure with union as members, Union with structures as members | T1:148-150,W1 |
| 5. | 1 | Understanding a Pointer Variable, Simple use of Pointers | T1:251-254,W1 |
| 6. | 1 | Pointers to Pointers, Pointers to structures, Problems with Pointers | T1:273-275,W1 |
| 7. | 1 | Passing pointers as function arguments, Returning a pointer from a function, using arrays as pointers, Passing arrays to functions | T1:275-280,W1 |
| 8. | 1 | Pointers vs. References, Declaring and initializing references, using references as function arguments and function return values | T1:280-286,R1:315,325 |
| 9. | 1 | RECAPITULATION OF IMPORTANT QUESTIONS | |
| | | TOTAL HOURS: 9 HOURS | |

**TEXTBOOK :**E:BALAGURUSAMY(2008).OBJECT ORIENTED PROGRAMMING WITH C++,TATA MC GRAW HILL EDUCATION.

**REFERENCE BOOKS:**BJARNEE STROUSTRUP,(2013).THE C++ PROGRAMMING LANGUAGE ,(4TH ED.).ADDITION – WELSEY.

**WEBSITE:**WWW.cplusplus.com/

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University Established under Section 3 of UGC Act 1956)**
**Pollachi Main Road, Eacharani Post, Coimbatore-641 021**

**DEPARTMENT OF COMPUTER APPLICATIONS**

**Semester – I**

**PROGRAMMING FUNDAMENTALS USING C/C++(18CAU101)**

**LESSON PLAN**

**UNIT-IV**

| S.NO | DURATION | TOPICS | SUPPORTED MATERIALS |
|---|---|---|---|
| 1. | 1 | Differentiating between static and dynamic memory allocation | T1:291-293, R1:330-333 |
| 2. | 1 | use of malloc, calloc and free functions ,Use of new operator,delete operator | T1:295-298, R1:334 |
| 3. | 1 | storage of variables in static and dynamic memory allocation | T1:300-310, W1 |
| 4. | 1 | Opening and closing a file ,reading and writing text files | T1:325-330 |
| 5. | 1 | Using put(), get(), read() and write() functions ,random access in file | T1:333-345 |
| 6. | 1 | Understanding the Preprocessor Directives,Macros | T1:400-410 |
| 7. | 1 | RECAPITULATION OF IMPORTANT QUESTIONS | |
| | | TOTAL HOURS: 7HOURS | |

**TEXTBOOK :**E:BALAGURUSAMY(2008).OBJECT ORIENTED PROGRAMMING WITH C++,TATA MC GRAW HILL EDUCATION.

**REFERENCE BOOKS:**BJARNEE STROUSTRUP,(2013).THE C++ PROGRAMMING LANGUAGE ,(4TH ED.).ADDITION – WELSEY.

**WEBSITE:**WWW.cplusplus.com/

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**(Deemed to be University Established under Section 3 of UGC Act 1956)**
**Pollachi Main Road, Eacharani Post, Coimbatore-641 021**

**DEPARTMENT OF COMPUTER APPLICATIONS**

**Semester – I**

**PROGRAMMING FUNDAMENTALS USING C/C++(18CAU101)**

**LESSON PLAN**

**UNIT-V**

| S.NO | DURATION | TOPICS | SUPPORTED MATERIALS |
|---|---|---|---|
| 1. | 1 | Principles of Object-Oriented Programming, Defining & Using Classes, Class Constructors, Constructor Overloading | T1:359-362 |
| 2. | 1 | Function overloading in classes, Class Variables &Functions, Objects as parameters | T1:362-365, W1 |
| 3. | 1 | Specifying the Protected and Private access, Copy Constructors ,overview of template classes and their uses | T1:366-370,W1 |
| 4. | 1 | Need of Overloading functions and operators | T1:371-373,W1 |
| 5. | 1 | Overloading functions by number and type of arguments, | T1:373-375,W1 |
| 6. | 1 | Looking at an operator as a function call, Overloading Operators | T1:378-380,W1 |
| 7. | 1 | Introduction to Inheritance | R1:202,W1 |
| 8. | 1 | Polymorphism | R1:210,W1 |
| 9. | 1 | Basics Exceptional Handling | T1:381,W1 |
| 10. | 1 | Catching all exceptions, Restricting exceptions, Rethrowing exceptions | T1:386-398,W1 |
| 11. | 1 | RECAPITULATION OF IMPORTANT QUESTIONS | |
| 12. | 1 | DISCUSSION OF ESE QUESTION PAPER | |
| 13. | 1 | DISCUSSION OF ESE QUESTION PAPER | |
| 14. | 1 | DISCUSSION OF ESE QUESTION PAPER | |
| | | TOTAL HOURS:14 HOURS | |

**TEXT BOOK :**E:BALAGURUSAMY(2008).OBJECT ORIENTED PROGRAMMING WITH C++,TATA MC GRAW HILL EDUCATION.

**REFERENCE BOOKS**:BJARNEE STROUSTRUP,(2013).THE C++ PROGRAMMING LANGUAGE ,(4TH ED.).ADDITION – WELSEY.

**WEBSITE:**WWW.cplusplus.com/

# UNIT I
# Introduction to C and C++:

## History of C and C++

C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972.

In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard.

The UNIX operating system, the C compiler, and essentially all UNIX application programs have been written in C. C has now become a widely used professional language for various reasons:

- Easy to learn
- Structured language
- It produces efficient programs
- It can handle low-level activities
- It can be compiled on a variety of computer platformsC was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around the early 1970s.
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- The UNIX OS was totally written in C.
- Today C is the most widely used and popular System Programming Language.
- Most of the state-of-the-art software have been implemented using C.
- Today's most popular Linux OS and RDBMS MySQL have been written in C.

- The C++ programming language has a history going back to 1979, by Bjarne Stroustrup
- Language included classes, basic inheritance, inlining, default function arguments, and strong type checking in addition to all the features of the C language.
- In 1983, the name of the language was changed from C with Classes to C++. The ++ operator in the C language is an operator for incrementing a variable, which gives some insight into how Stroustrup regarded the language.

- Many new features were added around this time, the most notable of which are virtual functions, function overloading, references with the & symbol, the const keyword, and single-line comments using two forward slashes (which is a feature taken from the language BCPL).
- In 1985, Stroustrup's reference to the language entitled *The C++ Programming Language* was published

## Overview of Procedural Programming and Object-Orientation Programming

### Procedural Programming

**Procedural programming** uses a list of instructions to tell the computer what to do step-by-step. Procedural programming relies on - you guessed it - procedures, also known as routines or subroutines. A procedure contains a series of computational steps to be carried out. Procedural programming is also referred to as imperative programming. Procedural programming languages are also known as top-down languages.

Procedural programming is intuitive in the sense that it is very similar to how you would expect a program to work. If you want a computer to do something, you should provide step-by-step instructions on how to do it. It is, therefore, no surprise that most of the early programming languages are all procedural. Examples of procedural languages include Fortran, COBOL and C, which have been around since the 1960s and 70s.

### Object-Oriented Programming

**Object-oriented programming**, or **OOP**, is an approach to problem-solving where all computations are carried out using objects. An **object** is a component of a program that knows how to perform certain actions and how to interact with other elements of the program. Objects are the basic units of object-oriented programming. A simple example of an object would be a person. Logically, you would expect a person to have a name. This would be considered a property of the person. You would also expect a person to be able to do something, such as walking. This would be considered a method of the person.

A method in object-oriented programming is like a procedure in procedural programming. The key difference here is that the method is part of an object. In object-oriented programming, you organize your code by creating objects, and then you can give those objects properties and you can make them do certain things.

A key aspect of object-oriented programming is the use of classes. A class is a blueprint of an object. You can think of a class as a concept and the object as the embodiment of that concept. So, let's say you want to use a person in your program. You want to be able to describe the person and have the person do something. A class called 'person' would provide a blueprint for what a person looks like and what a person can do. Examples of object-oriented languages include C#, Java, Perl and Python.

**Using main() function**

```
#include <stdio.h>
int main()
{
    /* my first program in C */
    printf("Hello, World! \n");
    return 0;
}
```

Let us take a look at the 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 the 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 the main() function and returns the value 0.

**Compiling and Executing Simple Programs in C++**

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 have 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 see the output *"Hello World"* printed on the screen.

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

Make sure the gcc compiler is in your path and that you are running it in the directory containing the source file hello.c.

**Data Types, Variables, Constants, Operators and Basic I/O:**

**Declaring, Defining and Initializing Variables**

- The data name which is used to store the data value is called ‘Variable’
- Variables are symbolic references to the addresses, where data values are
- stored
- Variables assume data values at the time of execution
- Variables may take different values at the different times of execution
- Variables are formed using the following rules:
- They must begin with a letter or underscore
- They must contain alphabets, digits or underscore
- First 31 characters are significant
- Cannot use keywords
- Cannot contain white spaces
- Variables are handled by the compiler at the time of compilation
- Variable name conveys to the compiler, the type of data it holds

**Example:**

Valid Variable Name
_name
Reg_No
MARK_1
mark_1
country
Quantity
results_UG
Rate_per

**Scope of Variables**

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes 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 within 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 the 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);
}
```

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, however, it points the variable name at a storage location that has been previously defined.

**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* is being used to declare *count* in the second file, whereas it has its definition in the first file, main.c.

## Using Named Constants

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

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 enumeration constants as well.

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 types 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 decimal form, you must include the decimal point, the exponent, or both; and while representing 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' 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 that represent special meaning when preceded by a backslash, for example, newline (\n) or tab (\t). Here, you have a list 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 a 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.

```
"hello, dear"
"hello, \
```

```
dear"
"hello, " "d" "ear"
```

## Defining Constants

There are two simple ways in C to define constants:

- Using **#define** preprocessor
- Using **const** keyword

### The #define Preprocessor

Given below is the form to use #define preprocessor to define a constant:

```
#define identifier value
```

The 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;
```

The 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
```

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

## Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables 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 | | | |

## Data Types

Data types in C 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 are further classified into: (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 assign 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 collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see the basic types in the following section, whereas other types will be covered in the upcoming chapters.

## Integer Types

The following table provides the details of standard integer types with their 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. Given below 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

The following table provides the details of standard floating-point types with storage sizes and value ranges and their precision:

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

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the 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;
}
```

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

```
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 any 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 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.

**Casting of Data Types**

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer, then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the **cast operator** as follows:

```
(type_name) expression
```

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation:

```
#include <stdio.h>

main()
{
    int sum = 17, count = 5;
    double mean;
    mean = (double) sum / count; printf("Value of
    mean : %f\n", mean );
}
```

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

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

## Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character with an integer:

```
#include <stdio.h>
main()
{
    int     i = 17;
    char c = 'c'; /* ascii value is 99 */
    int sum;
    sum = i + c;
    printf("Value of sum : %d\n", sum );
}
```

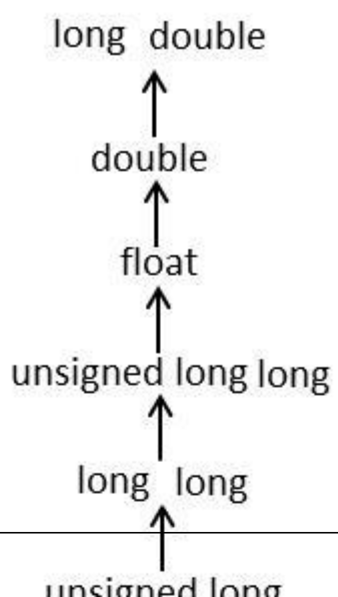
When the above code is compiled and executed, it produces the following result:

```
Value of sum : 116
```

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

## Usual Arithmetic Conversion

The **usual arithmetic conversions** are implicitly performed to cast their values to a common type. The compiler first performs *integer promotion*; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy:



The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take the following example to understand the concept:

```
#include <stdio.h>
main()
{
    int     i = 17;
    char c = 'c'; /* ascii value is 99 */
    float sum;
    sum = i + c;
    printf("Value of sum : %f\n", sum );
}
```

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

```
Value of sum : 116.000000
```

Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

**Operators**

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

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

We will, in this chapter, look into the way each operator works.

**Arithmetic Operators**

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20, then:

| Operator | Description | Example |
|---|---|---|

| | | |
|---|---|---|
| + | Adds two operands. | A + B = 30 |
| - | Subtracts second operand from the first. | A - B = -10 |
| * | Multiplies both operands. | A * B = 200 |
| / | Divides numerator by de-numerator. | B / A = 2 |
| % | Modulus Operator and remainder of after an integer division. | B % A = 0 |
| ++ | Increment operator increases the integer value by one. | A++ = 11 |

-- Decrement operator decreases the integer A-- = 9 value by one.

**Example**

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

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

}

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

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

**Relational Operators**

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

| Operator | Description | Example |
|---|---|---|
| == | Checks if the values of two operands are equal or not. If yes, then the condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not. If the values are not equal, then the 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 the 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 the 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 the condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than | (A <= B) is true. |

| | | |
|---|---|---|
| | or equal to the value of right operand. If yes, then the condition becomes true. | |

**Example**

Try the following example to understand all the relational operators available in C:

```
#include <stdio.h>
main()
{
    int a = 21;
    int b = 10;
    int c ;

    if( a == b )
    {
        printf("Line 1 - a is equal to b\n" );
    }
    else
    {
        printf("Line 1 - a is not equal to b\n" );
    }
    if ( a < b )
    {
        printf("Line 2 - a is less than b\n" );
    }
    else
    {
        printf("Line 2 - a is not less than b\n" );
    }
    if ( a > b )
    {
        printf("Line 3 - a is greater than b\n" );
    }
    else
```

```
    {
        printf("Line 3 - a is not greater than b\n" );
    }
    /* Lets change value of a and b */
    a = 5;
    b = 20;
    if ( a <= b )
    {
        printf("Line 4 - a is either less than or equal to            b\n" );
    }
    if ( b >= a )
    {
        printf("Line 5 - b is either greater than          or equal to b\n" );
    }
}
```

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

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to          b
Line 5 - b is either greater than          or equal to b
```

## Logical Operators

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

| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |

| | | |
|---|---|---|
| \|\| | Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true. | (A \|\| B) is true. |
| ! | Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false. | !(A && B) is true. |

**Example**

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

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

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

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

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

## Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation. The truth table for &, |, and ^ is as follows:

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

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

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then:

| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12, i.e., 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) = 61, i.e., 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49, i.e., 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = -61, i.e., 1100 0011 in 2's complement form. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 = 240, i.e., 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 = 15, i.e., 0000 1111 |

**Example**

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

```
#include <stdio.h>

main()
{

    unsigned int a    = 60;      /* 60 = 0011 1100 */
    unsigned int b    = 13;      /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;              /* 12 = 0000 1100 */
    printf("Line 1    - Value   of c is %d\n", c );

    c = a | b;              /* 61 = 0011 1101 */
    printf("Line 2    - Value   of c is %d\n", c );

    c = a ^ b;              /* 49 = 0011 0001 */
    printf("Line 3    - Value   of c is %d\n", c );

    c = ~a;                 /*-61 = 1100 0011 */
    printf("Line 4    - Value   of c is %d\n", c );

    c = a << 2;             /* 240 = 1111 0000 */
    printf("Line 5    - Value   of c is %d\n", c );

    c = a >> 2;           /* 15 = 0000 1111 */
    printf("Line 6    - Value of c is %d\n", c );
}
```

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

Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15

## Assignment Operators

The following tables lists the assignment operators supported by the C language:

| Operator | Description | Example |
|---|---|---|
| = | Simple assignment operator. Assigns values from right side operands to left side operand. | C = A + B will assign the value of A + B to C |
| += | Add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand. | C -= A is equivalent to C = C - A |
| *= | Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand. | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand. | C %= A is equivalent to C = C % A |

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

**Example**

Try the following example to understand all the assignment operators available in C:

```
#include <stdio.h>
main()
{
    int a = 21;
    int c ;

    c =  a;
    printf("Line 1 - =       Operator Example, Value of c = %d\n", c );
    c +=  a;
    printf("Line 2 - += Operator Example, Value of c = %d\n", c );
    c -=  a;
    printf("Line 3 - -= Operator Example, Value of c = %d\n", c );
    c *=  a;
    printf("Line 4 - *= Operator Example, Value of c = %d\n", c );
```

```
    c /=      a;
    printf("Line 5 - /= Operator Example, Value of c = %d\n", c );
    c    = 200;
    c %=  a;
    printf("Line 6 - %= Operator Example, Value of c = %d\n", c );

    c <<=    2;
    printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );

    c >>=    2;
    printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );

    c &=    2;
    printf("Line 9 - &= Operator Example, Value of c = %d\n", c );

    c ^=     2;
    printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );

    c |=      2;
    printf("Line 11 - |= Operator Example, Value of c = %d\n", c );

 }
```

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

```
Line 1 - = Operator Example, Value of c = 21 Line 2 - +=
Operator Example, Value of c = 42 Line 3 - -= Operator
Example, Value of c = 21 Line 4 - *= Operator Example,
Value of c = 441 Line 5 - /= Operator Example, Value of
c = 21 Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44 Line 8 -
>>= Operator Example, Value of c = 11
```

Line 9 - &= Operator Example, Value of c = 2

Line 10 - ^= Operator Example, Value of c = 0

Line 11 - |= Operator Example, Value of c = 2

**Misc Operators ↦ sizeof & ternary**

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

| Operator | Description | Example |
|---|---|---|
| sizeof() | Returns the size of a variable. | sizeof(a), where a is integer, will return 4. |
| & | Returns the address of a variable. | &a; returns the actual address of the variable. |
| * | Pointer to a variable. | *a; |
| ? : | Conditional Expression. | If Condition is true ? then value X : otherwise value Y |

**Example**

Try following example to understand all the miscellaneous operators available in C:

```
#include <stdio.h>
main()
{
    int a = 4;
    short b;
    double c;
    int* ptr;
   /* example of sizeof operator */
```

```
printf("Line 1 - Size of variable a = %d\n", sizeof(a) ); printf("Line 2 -
Size of variable b = %d\n", sizeof(b) ); printf("Line 3 - Size of variable
c= %d\n", sizeof(c) );

/* example of & and * operators */
ptr = &a;        /* 'ptr' now contains the address of 'a'*/
printf("value of a is          %d\n", a);
printf("*ptr is %d.\n", *ptr);

/* example of ternary operator */
a = 10;
b = (a == 1) ? 20: 30;
printf( "Value of b is %d\n", b );
b = (a == 10) ? 20: 30;
printf( "Value of b is %d\n", b );
}
```

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

```
value of a is        4
*ptr is 4.
Value of b is 30
Value of b is 20
```

**Using Comments in programs**

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminate 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.

**Character I/O**

**The getchar() and putchar() Functions**

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example:

```
#include <stdio.h>
int main( )
{
    int c;
    printf( "Enter a value :");
    c = getchar( );
    printf( "\nYou entered: ");
    putchar( c );
    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows:

```
$./a.out
Enter a value : this is test
You entered: t
```

## The gets() and puts() Functions

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char *s)** function writes the string ‘s’ and ‘a’ trailing newline to **stdout**.

```
#include <stdio.h>
int main( )
{
    char str[100];
    printf( "Enter a value :");
    gets( str );
    printf( "\nYou entered: ");
```

```
    puts( str );
    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows:

```
$./a.out
Enter a value : this is test
You entered: This is test
```

## Formatted and Console I/O

### The scanf() and printf() Functions

The **int scanf(const char *format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character, or float, respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better:

```
#include <stdio.h>
int main( )
{
    char str[100];
    int i;
    printf( "Enter a value :");
    scanf("%s %d", str, &i);
    printf( "\nYou entered: %s %d ", str, i);
    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows:

```
$./a.out
Enter a value : seven 7
```

You entered: seven 7

Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

Using Basic Header Files

| | |
|---|---|
| **<assert.h>** | Conditionally compiled macro that compares its argument to zero |
| **<complex.h>** (since C99) | Complex number arithmetic |
| **<ctype.h>** | Functions to determine the type contained in character data |
| **<errno.h>** | Macros reporting error conditions |
| **<fenv.h>** (since C99) | Floating-point environment |
| **<float.h>** | Limits of float types |
| **<inttypes.h>** (since C99) | Format conversion of integer types |
| **<iso646.h>** (since C95) | Alternative operator spellings |
| **<limits.h>** | Sizes of basic types |
| **<locale.h>** | Localization utilities |
| **<math.h>** | Common mathematics functions |
| **<setjmp.h>** | Nonlocal jumps |
| **<signal.h>** | Signal handling |
| **<stdalign.h>** (since C11) | alignas and alignof convenience macros |
| **<stdarg.h>** | Variable arguments |
| **<stdatomic.h>** (since C11) | Atomic types |
| **<stdbool.h>** (since C99) | Boolean type |
| **<stddef.h>** | Common macro definitions |
| **<stdint.h>** (since C99) | Fixed-width integer types |
| **<stdio.h>** | Input/output |
| **<stdlib.h>** | General utilities: memory management, program utilities, string conversions, random numbers |
| **<stdnoreturn.h>** (since C11) | noreturn convenience macros |
| **<string.h>** | String handling |
| **<tgmath.h>** (since C99) | Type-generic math (macros wrapping math.h and complex.h) |
| **<threads.h>** (since C11) | Thread library |
| **<time.h>** | Time/date utilities |
| **<uchar.h>** (since C11) | UTF-16 and UTF-32 character utilities |

**<wchar.h>** (since C95) Extended multibyte and wide character utilities
**<wctype.h>** (since C95) Functions to determine the type contained in wide character data

**Expressions, Conditional Statements and Iterative Statements**

In programming, an expression is any legal combination of symbols that represents a value. Each programming language and application has its own rules for what is legal and illegal. For example, in the C language *x+5* is an expression, as is the character string *"MONKEYS."*

Every expression consists of at least one *operand* and can have one or more *operators*. Operands are values, whereas operators are symbols that represent particular actions. In the expression

x + 5

x and 5 are operands, and + is an operator.

Expressions are used in programming languages, database systems, and spreadsheet applications. For example, in database systems, you use expressions to specify which information you want to see. These types of expressions are called *queries*.

Expressions are often classified by the type of value that they represent. For example:

- **Boolean expressions :** Evaluate to either TRUE or FALSE
- **integer expressions:** Evaluate to whole numbers, like 3 or 100
- **Floating-point expressions:** Evaluate to real numbers, like 3.141 or -0.005
- **String expressions:** Evaluate to character strings

**Operators Precedence**

| **Category** | **Operator** | **Associativity** |
|---|---|---|
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)* & sizeof | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |

| | | |
|---|---|---|
| | | |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %=>>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

## DECISION MAKING AND BRANCHING STATEMENTS

***Need for Decision Making/Control Statements***

- C program is a set of statements which are normally **executed sequentially** in the order in which they appear.This happens when no options or no repetitions of certain calculations are necessary.
- Due to certain conditions, order of execution of statements may be changed based on certain conditions, or repeat a group of statements. Such a condition, the decision making statements is used for branching and/or looping of the statements.
- As these statements control the flow of execution of the program, they are known as ***“Control Statements”***

***Decision Making and Branching Statements***

C language possesses decision making capabilities and supports the following statements known as control or decision making statements.

- if statement
- switch statement
- Conditional operator
- goto statement

***Decision Making with if Statement***

- The ***if*** Statement is a powerful decision making statement and is used to control the flow of execution. It is basically two way branching statement.
- ***Syntax***

  ***if (test expression)***
- The test expression may be the relational expression or the logical expression or the condition. The result of the expression always may be true (non-zero value) or false (zero).
- It allows the compiler to evaluate the expression first and then depending upon the result of the expression, it transfers the control to a particular statement.
- At the time of control transfer, the control chooses the two paths namely true block (if part) and false block (else part).

*The if statement is implemented in different forms depending on the complexity of conditions to be tested:*

- ***Simple if statement***
- ***if…..else statement***
- ***Nested if…..else statement***
- ***Else if ladder***

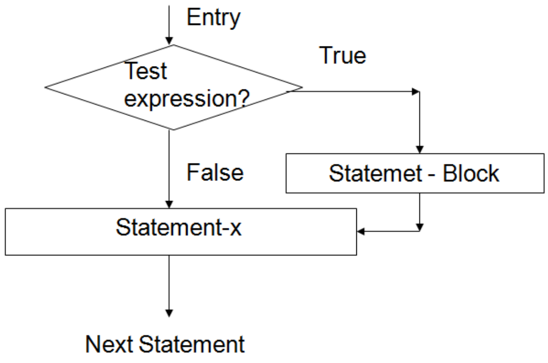
***Simple if Statement***

- The general form of a simple if statement is:

```
if(test-expression)
{
    statement-block;
}
Statement-x;
```

- The statement block may be the single statement or compound statements.
- If the test expression is true, the statement block is executed and then statement x also executed. Otherwise thestatement block will be skipped and will jump to statement x.

- **Flowchart**




- Sample Program:

```
void main()
{
int a=1;
if (a==1)
        {
printf("Hi..");
        }
printf("Hello");
}
```

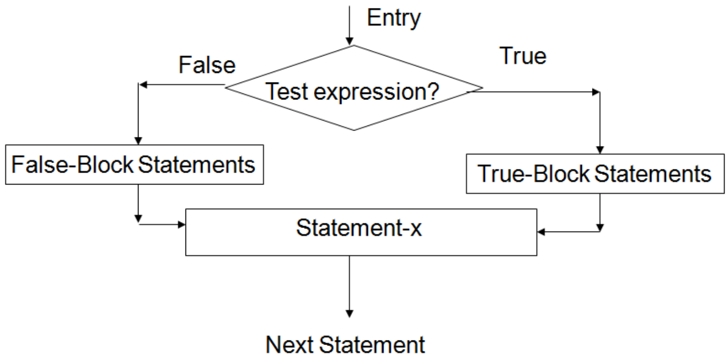
**Output: Hi.. Hello**

***if else statement***

- The if….else statement is an extension of the simple if statement.
- The general form is:

***if(test-expression)***
***{***
***True-block statement(s);***
***}***
***else***
***{***
***False-block statement(s);***
***}***
***Statement-x;***

- if the test expression/condition is true, then true block (immediately following the ***if*** statements) is executed, otherwise the false block is executed.
- In this category, either true block or false block will be executed, not both.
- **Flowchart**




- ***Sample Program***

```
void main()
{
    int a;
    printf("Enter the Number\n");
    scanf("%d",&a);
        if((a%2)==0)
printf("%d is Even",a);
        else
        printf("%d is Odd",a);
    getch();
}
```

***Output:***

***Enter the Number: 7***
***7 is Odd***
***Enter the Number: 14***
***14 is Even***

***Nested if Statement***

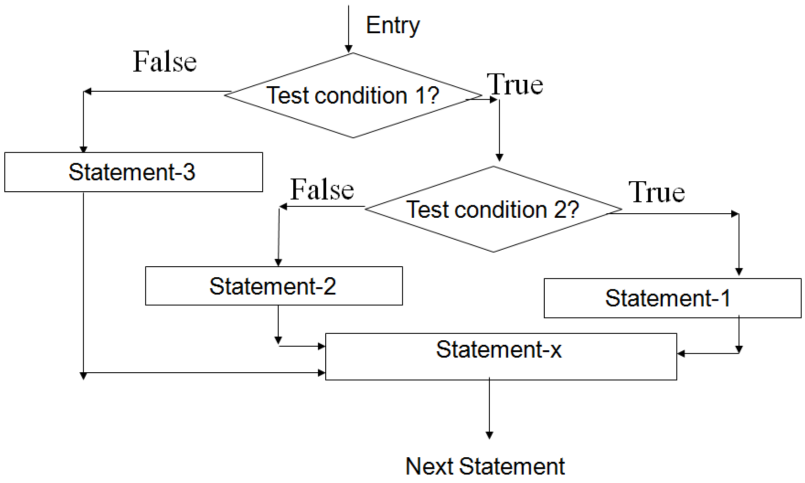
- C facilitates to write the ***if*** statement within either the body of another or outer ***if*** block or the body of the ***else*** block. This if structure is called ***nested if***. Nested if means if statements within if statements.
- The logical execution of nested if is

```
if(test condition 1)
{
    if(test condition 2)
    {
            statement-1;
    }
    else
    {
        statement-2;
    }
}
else
```

```
{
        statement-3;
}
Statement-x;
```

- In the above code segment, if the condition-1 is true, then the condition-2 is checked otherwise the statement-3 is executed. If the condition-2 is true, the statement-1 is run otherwise the statement-2 is executed.

- Flowchart




- ***Sample Program***

```
void main()
{
int a=0,b=1,c=2;
        if (a>b)
            {
if(a>c)
printf("\n a is Big");
            }
else
            {
if(c>b)
printf("\n c is Big");
else
        printf("\n b is Big);
            }
  }
```

**Output: c is Big**

***Dangling else problem:***

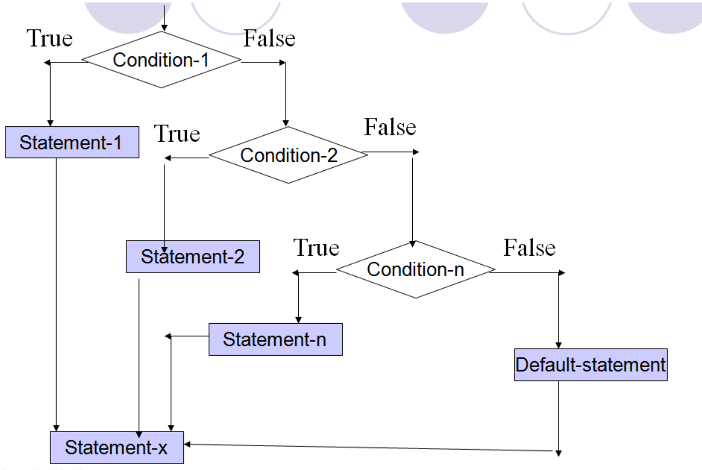
- The occurrence of unpaired or unmatched else in the program is called dangling else problem. This problem mostly occurs in nested if statements.
- Solution of Dangling else problem:
  - Dangled else is paired with recent ***if***.
  - Dangled else may be omitted if it is unnecessary.

***Else if Ladder***

- C provides the way of putting ***if***s together for multiple decision makings. A multiple decision is a chain of ***if***s in which the statement associated with each ***else*** is an ***if.***
- The structure of multiple ***else if*** statements is known as ***else if ladder.***
- The general form is

```
if (test condition 1)
        {
                statement –1 ;
         }
else if (test condition 2)
        {
                statement –2 ;
        }
        else if (test condition 3)
                {
                        statement – 3 ;
                 }
               …..
                else if (test condition n)
                    {
                        statement – n ;
                }
                else
                        {
                                default-statement;
                }
                        Statement – x ;
```

- If the condition-1 is false, the control transfers to next if condition i.e. condition-2 to be checked. If all the conditions are failed, then default statement is executed.
- Flowchart




- ***Sample Program***

```
void main()
 {
int age;
printf("Enter the Age:");
scanf("%d",&age);
if (age <15)
printf(" Childhood \n");
else  if( age>=15 && age<35)
printf("Youth \n");
else  if( age> 35 && age<50)
printf("Middle Age \n");
else
printf(" Oldage \n");
  }
```

***Output:***

***Enter the Age:25***

***Youth***

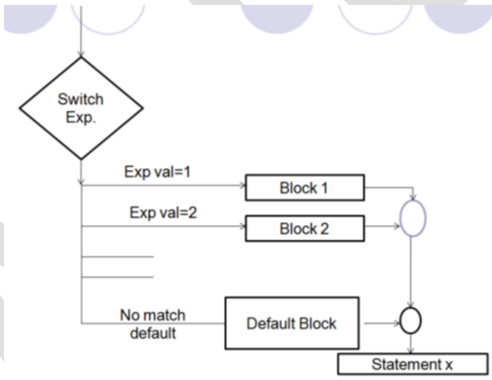
**SWITCH STATEMENT**

- C provides the powerful multiway decision statement known as ***'switch'***
- The ***switch*** statement checks the value of given variable or expression against a list of case values and when a match is found, a block of statements associated with that case is executed.
- The general form of ***switch*** statement is

```
switch ( expression )
{
        case Label/value-1 :
                        block-1;
                        break;
        case Label/value-2 :
                        block-2;
                        break;
        default :
```

```
                    default-block;
                    break;
    }
    Statement-x;
```

- The expression is an integer expression or characters.
- value-1, value-2 …are constants or constant expression and they are otherwise called as case labels.
- block-1, block-2 …..are statement list and may contain zero or more statements.
- The ***break*** statement at the end of the each block signals end of the particular case and causes an exit from the ***switch*** statement, transferring the control to the ***statement x.***
- The ***default*** is an optional case. When present it will be executed if the value of the expression does not match with any one of the cases values.
- If the ***default*** case does not present, the control automatically transfers to ***statement x*** when none of the cases match.
- The flowchart of the switch statement




***Example program for switch:***

```
void main()
 {
  int a;
   printf ("Enter the choice:");
  scanf("%d",&a);
  switch(a)
   {
     case 1:
           printf(" \n  Post Graduate");
            break;
     case 2:
           printf(" \n Under Graduate");
            break;
     default:
           printf("\n Diploma");
            break;
   }
   printf("\n  Gandhigram Rural Institute – Deemed University");
```

```
}
```

**Output:** Enter the choice 1
Post Graduate
Gandhigram Rural Institute – Deemed University

***//Program to check whether given character is vowel or not***

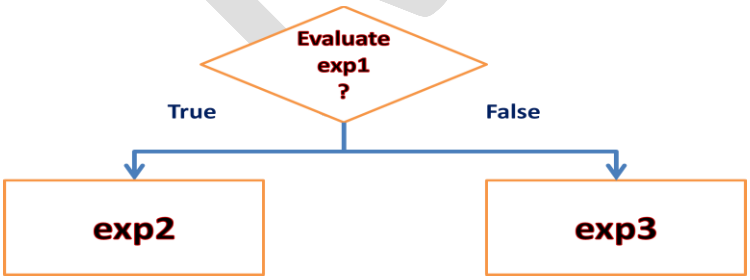
```
void main()
{
  char ch;
  printf ("Enter the character");
  scanf("%c",&ch);
  switch(ch)
  {
    case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
          printf(" \n  The given character is Vowel");
          break;
    default:
          printf("\n The given character is consonent" );
          break;
  }
  getch();   }
```

**Output:** Enter the character a
The given character is Vowel

## CONDITIONAL OPERATOR

- The ternary operators are ? and :
- These operators are called as ternary operators as they operate on three operands
- General Format: **exp1 ? exp2 : exp3**

where exp1 exp2 and exp3 are expressions




- The nested conditional operators are also allowed in C.

***Sample program:***

```
void main()
{
        int a=9,b=4,c=8,big;
        clrscr();
    big=((a>b)&&(a>c))?a:((b>c)?b:c);
        printf("%d",big);
}
```

Output: 9

## GOTO STATEMENT

- The statement in C which facilitates to branch/ transfer the control ***unconditionally*** from one point to another point in the program is called ***'goto'*** statement.
- The syntax of ***goto*** statement:

```
Forward Jump
statements;
goto lable1;
        Statements;
lable1 :
        statements;
```

```
Backward Jump
statements;
lable1 :
        statements;
goto lable1;
        Statements;
```

- The ***goto*** requires a label in order to identify the place where the branch is to be made.
- A label is any valid variable name and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred.

***Sample program***

```
void main()
{
        double a,b;
        read:
                printf("\n Enter the Number:");
                scanf("%lf",&a);
        if(a<0)
                goto read;
        b=sqrt(a);
        printf("\n Square Root of %lf is:%lf",a,b);
        getch();
}
```

**Output:**

Enter the Number:-1
Enter the Number:4
Square Root of 4 is 2.000000

***Infinite loop***

- Unconditional branching statement leads to repeat the some actions indefinitely and it puts in the permanent loop. Such a loop is called '***infinite loop'.***

```
void main()
{
        double a,b;
        read:
                printf("\n Enter the Number:");
                scanf("%lf",&a);
        if(a<0)
                goto read;
        b=sqrt(a);
        printf("\n Square Root of %lf is:%lf",a,b);
        goto read;
        getch();
}
```

- Due to the unconditional ***goto*** statement at the end, the control always transferred back to the input statement. This program is never executed and puts in permanent loop.

***********

## DECISION MAKING AND LOOPING STATEMENTS

- During looping a set of statements are executed until some conditions for termination of the loop is encountered.
- A program loop therefore consists of two segments one known as body of the loop and other is the control statement.
- The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.
- In looping process in general would include the following four steps

1. Setting and initialization of a counter
2. Application of the statements in the loop
3. Test for a specified conditions for the execution of the loop
4. Incrementing the counter

- The test may be either to determine whether the loop has repeated the specified number of times or to determine whether the particular condition has been met.
- Type of Looping Statements are
  - while statement
  - do while statement
  - for statement

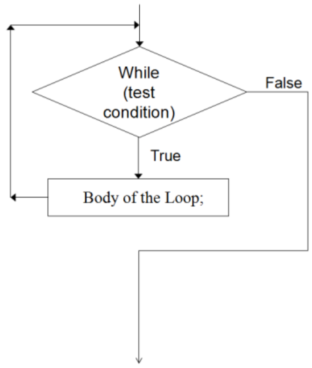
***while Statement:***

- The simplest of all looping structure in C is the while statement.

- The general format of the while statement is:

  ***while (test condition)***
  ***{***
  ***body of the loop;***
  ***}***

- the given test condition is evaluated and if the condition is true then the body of the loop is executed.
- After the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again.
- This process continues until the test condition fails.
- The flowchart of while statement




***Sample Program:***

```
#include<stdio.h>
void main()
{
        int num=0, rev_num=0;
        printf("Enter the number to be reversed:");
        scanf("%d", &num);
        while(num != 0)             // While statement with condition
        {
                rev_num = num % 10;         // get the last digit
                printf("%d", rev_num);       // print the digit
                num = num / 10;
        }
        getch();
}
```

**Output:**

Enter the number to be reversed: **123**

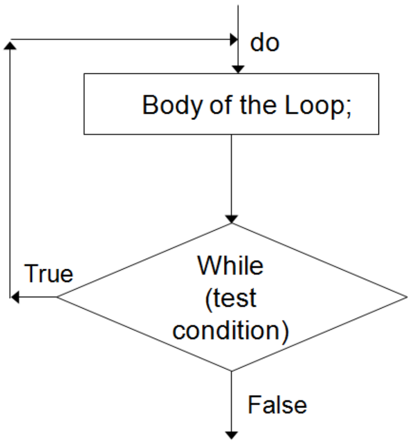
**321**

***do… while statement***

- The do while loop tests at the bottom of the loop after executing the body of the loop.
- Since the body of the loop is executed first and then the loop condition is checked, this statement can be assured that the body of the loop is executed at least once.
- The general format of the do..while statement is:

```
do
{
        body of the loop;
}
while (test condition);
```

- The flowchart of do….while statement




***Sample Program:***

```
#include<stdio.h>
void main()
{
        int num=0, rev_num=0;
        printf("Enter the number to be reversed:");
        scanf("%d", &num);
        do
        {
                rev_num = num % 10;          // get the last digit
                printf("%d", rev_num);       // print the digit
                num = num / 10;
        } while(num != 0);      // do…while statement with condition

        getch();
}
```

**Output:**

Enter the number to be reversed: **123**
**321**

***for loop statement***

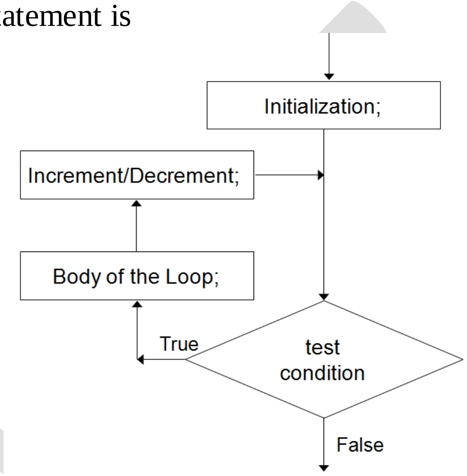
- The for loop is most commonly and popularly used loop in C. the for loop allows us to specify three things about the loop in a single line. They are,
    - Initializing the value for the loop

- Condition in the loop counter to determine whether the loop should continue or not
- Incrementing or decrementing the value of loop counter each time the program segment has been executed.
- 

- The general form of the for loop is:

```
for(initialization; test condition; increment)
{
body of the loop;
}
```

- The flowchart of for statement is




***Sample Program:***

```
#include<stdio.h>
void main()
{
        int n, i,f=1;;
        printf("Enter the number:");
        scanf("%d", &n);
        for(i=1;i<=n;i++)
        {
                f=f*i;
       }
        printf("Factorial :%d", f);
        getch();
}
```

**Output:**

Enter the number: **5**
Factorial: **120**

**Additional Features of for loops:**

- More than one variable can be initialized at a time in for loop.
  - ***for(n=1,p=1;i<=n;i++)***
- More than one decrement/ increment can given in increment section in for loop

- ***for(n=1,m=50;n<=m;n=n+1,m=m-1)***
- Test condition may have any compound relation and testing need not be limited only to the loop control variable.
  - ***for(i=1;i<20 && sum<100;i++)***
- The expressions can be permitted in the assignment statements of initialization and increment section.
  - ***for(x=(m+n)/2;x>0;x=x/2)***
- One or more sections in for loop can be omitted.
  - ***for(;i<100;i++)***
  - ***for(;;i++)***
  - ***for(;i<100;)***
  - ***for(i=1;;)***
  - ***for(;;)***
- null statements can be given in the for loop by using **;**. This type of loop is used for time delay. So, they are otherwise called "time delay loops"
  - ***for(j=1000;j>0;j--)***
    ***;***

***Nesting of for loops***

- for loop within for loop is called nested for loop statements.
- Nest for loops statements are used for comparison of variables in one array and operations on tables, matrix and tables of table.
- The general form of nested for loop is

```
for(i=0; i<10; i++)          → outer for loop
  {
    for( j=0; j<10; j++)     → inner for loop
      {
      }
  }
```

***Sample code:***

```
for(row=1;row<=3;row++)
        {
                for(col=1;col<=3;col++)
                 {
                        y=row*col;
                        printf("%d",y);
                 }
          printf("\n")
        }
```

***Jumps in Loops***

- Jumps are used for skipping the loop iteration or exit/leave from the loops
- C provides two statements for jumping in loops. They are namely
  - Break – Exit/leave from the loop/branching statement.
  - Continue – Skip the current iteration in the loop

***Break Statement:***

- C language permits a jump from one statement to another within a loop as well as to jump out of the loop.
- Sometimes while executing a loop it becomes desirable to skip a part of the loop or quit the loop as soon as certain condition occurs.
- For example consider searching a particular number in a set of 100 numbers. As soon as the search number is found it is desirable to terminate the loop.
- The break statement allows us to accomplish this task. A break statement provides an early exit from for, while, do and switch constructs. A break causes the innermost enclosing loop or switch to be exited immediately

***Sample Program:***

```
 void main()
{
    int num=0, loop=0; float sum=0;
    printf("Enter the marks, -1 to end\n");
    while(1)
    {
        scanf("%d", &num0);
        if(num == -1)
                break;
        sum+=num;    loop++;
    }
    printf("The average marks is: %d", sum/loop);
}
```

***Continue Statement***

- During loop operations it may be necessary to skip a part of the body of the loop under certain conditions.
- Like the break statement C supports similar statement called continue statement.
- The continue statement causes the loop to be continued with the next iteration after skipping any statement in between.

**Sample Program:**

```
#include < stdio.h >
void main()
{
        int loop, num, sum=0;
        for (loop = 0; loop < 5; loop++)                // for loop
        {
                printf("Enter the integer");            //Message to the user
                scanf("%d", &num);                      //read and store the number
                if(num < 0)  //check whether the number is less than zero
                {
                        printf("You have entered a negative number");
                        continue;   // starts with the beginning of the loop
                }                       // end of for loop
```

```
        sum+=num;       // add and store sum to num
    }
    printf("The sum of positive numbers entered = %d",sum);
}
```

## POSSIBLE QUESTIONS

### PART B

### (Each Question Carries 2 marks)

1. Write difference between algorithm and flowchart.
2. Explain the importance of C language.
3. What is format specifier?
4. What are local and global variable?
5. Define keyword, constant and variable.
6. Why do we use header files?
7. Define relational operator.
8. What is an interpreter?
9. What is the purpose of adding comments in a program?
10. What is the syntax of switch statement?

### PART C

### (Each Question Carries 8 marks)

1. Discuss the structure of a C program. Explain with example.
2. What are the various I/O functions in C?
3. What do you mean by data types? Give examples of data types available in C language.
4. Explain the various control statements used in c language.
5. Write a program to find whether a number is Armstrong or not.
6. What is the difference between pre and post increment operator? Explain with the help of an example.
7. What is the difference between break and continue statements? Explain with the help of an example.
8. Write the advantage and disadvantage of for loop over while.
9. Write difference between while and do while with example.
10. What do you mean by ternary operator? Explain with example.

**Karpagam Acader**
**(Deemed University Establishe**
**Coimbat**
**DEPARTMENT OF CC**

**PROGRAMMING FUNDAME**

U

| S.No | Question | Choice1 |
|---|---|---|
| 1 | The decomposition of a problem into a number of entities called___________ | objects |
| 2 | OOPS follows______________ approach in program design | bottom-up |
| 3 | Objects take up ______________in the memory | space |
| 4 | _________________is a collection of objects of similar type | Objects |
| 5 | We can create ____________of objects belonging to that class | 1 |
| 6 | The wrapping up of data & function into a single unit is known as _______________ | Polymorphism |
| 7 | __________________refers to the act of representing essential features without including the background details or explanations | encapsulation |
| 8 | Attributes are sometimes called______________ | data members |
| 9 | The functions operate on the datas are called______________ | methods |
| 10 | ______________is the process by which objects of one class acquire the properties of objects of another class | polymorphism |
| 11 | __________________means the ability to take more than one form | polymorphism |
| 12 | The process of making an operator to exhibit different behaviors in different instances is known as ________________ | function overloading |
| 13 | Single function name can be used to handle different types of tasks is known as ___________ | function overloading |
| 14 | _______________means that the code associated with a given procedure call is not known until the time of the call at run-time. | late binding |
| 15 | Objects can be___________ | created |
| 16 | ______________helps the programmer to build secure programs | Dynamic binding |
| 17 | _________________techniques for communication between objects makes the interface descriptions with external systems much simpler | message passing |

| | | |
|---|---|---|
| 18 | Variables are declared in__________________ | only in main() |
| 19 | How many sections in C++? | 2 |
| 20 | _____________________refers to permit initialization of the variables at run time | Dynamic initialization |
| 21 | ______________________provides an alias for a previously defined variable | static variable |
| 22 | Reference variable must be initialized at the time of ______________ | declaration |
| 23 | The ____________________is an exit-controlled loop | while |
| 24 | The _________________is an entry-entrolled loop | while |
| 25 | _____________________is an entry-controlled one | while |
| 26 | Error checking does not occur during compilation if we are using________________ | functions |
| 27 | _____________________is a function that is expanded in line when it is invoked | macros |
| 28 | _________________refers to the use of same thing for different purposes | overloading |
| 29 | __________________are extensively used for handling class objects | overloaded functions |
| 30 | _____________________is used to reduce the number of functions to be defined | default arguments |
| 31 | Control structures are said to be________________ | programs |
| 32 | _________________________is a decision making statement | for |
| 33 | The bool type data occupies ____________byte in memory | two |
| 34 | if-else-if ladder sometimes called_________________ | if-else-if nested |
| 35 | How many statements are used to perform an unconditional transfer? | 2 |
| 36 | The label must start with____________ | character |
| 37 | _________________statement is frequently used to terminate the loop in the switch case() | jump |
| 38 | _______________statement does not require any condition | for |
| 39 | _____________statement is used to transfer the control t pass on t the beginning of the block/loop | break |
| 40 | _________________statement is a multiway branch statement | for |
| 41 | Every case statement in switch case statement terminates with | ; |

| | | |
|---|---|---|
| 42 | How many types of loop control structure exist in c++? | 1 |
| 43 | The expression are separated by ____________in the for loop | : |
| 44 | Test is performed at the ____________of the for loop. | top |
| 45 | Condition is checked at the ____________of the loop in the do-while statement. | beginning |
| 46 | Every expression always return____________ | 0 or 1 |
| 47 | Which of the following loop statement uses 2 keyword? | do-while loop |
| 48 | The meaning of if(1) is________________ | always false |
| 49 | The for loop comprises of ______________actions | 2 |
| 50 | _____________statement present at the bottom of the switch case statements | default |
| 51 | __________________is an assignment statement that is used to set the loop control variables | Increment |
| 52 | Which of the following control expressions are valid for an of statement ? | an integer expression |
| 53 | Which of the following cannot be passed to a function? | reference variables |
| 54 | Function should return a _________. | value |
| 55 | _______________function is useful when calling function is small | Built-in |
| 56 | Inline function needs more_____________ | variables |
| 57 | Multiple function with the same name is known as ________________ | function overloading |
| 58 | The ____________ function creates a new set of variables and copies the values of arguments into them. | calling function |
| 59 | Function contained within a class is called a _____________ | built-in |
| 60 | In c++,Declarations can appear________________in the body of the function | Only at the top |

**ny of Higher Education**
**d Under Section 3 of UGC Act 1956)**
**tore – 641 021**
**)MPUTER APPLICATIONS**

**:NTALS USING C/C++(18CAU101)**

NIT --1

| Choice2 | Choice3 | Choice4 | | | Ans |
|---|---|---|---|---|---|
| classes | methods | messages | | | **objects** |
| top-down | middle | top | | | **bottom-up** |
| address | memory | bytes | | | **space** |
| methods | classes | messages | | | **classes** |
| 2 | 10 | any number | | | **any number** |
| encapsulation | functions | data members | | | **encapsulation** |
| inheritance | Dynamic binding | Abstraction | | | **Abstraction** |
| methods | messages | functions | | | **data members** |
| data members | messages | classes | | | **methods** |
| encapsulation | data binding | Inheritance | | | **Inheritance** |
| encapsulation | data binding | information hiding | | | **polymorphism** |
| operator overloading | method overloading | message overloading | | | **operator overloading** |
| operator overloading | polymorphism | encapsulation | | | **operator overloading** |
| Dynamic binding | Static binding | Quick binding | | | **Dynamic binding** |
| created & destroyed | permanent | temporary | | | **created & destroyed** |
| Data hiding | Data building | message passing | | | **Data hiding** |
| Data binding | Encapsulation | Data passing | | | **message passing** |

| | | | | | |
|---|---|---|---|---|---|
| anywhere in the scope | before the main() only | only at the beginning | | | **anywhere in the scope** |
| 4 | 1 | 5 | | | **4** |
| Dynamic binding | Data binding | Dynamic message | | | **Dynamic initialization** |
| Dynamic variable | reference variable | address of an variable | | | **reference variable** |
| assigning | initialization | running | | | **declaration** |
| do-while | for | switch | | | **do-while** |
| do-while | for | switch | | | **for** |
| do-while | for | switch | | | **while** |
| macros | pre-defined functions | operators | | | **macros** |
| inline function | predefined function | preprocessor macros | | | **inline function** |
| Dynamic binding | message loading | overriding | | | **overloading** |
| methods | objects | messages | | | **overloaded functions** |
| methods | objects | classes | | | **default arguments** |
| structured programs | statements | case statements | | | **structured programs** |
| jump | break | if | | | **if** |
| one | three | four | | | **one** |
| nested-if-else-if | if-else-if-staircase | if-else-if | | | **if-else-if-staircase** |
| 3 | 4 | 5 | | | **4** |
| symbols | number | alphanumeric | | | **character** |
| goto | continue | break | | | **break** |
| if | goto | while | | | **goto** |
| jump | goto | continue | | | **continue** |
| switch | if | while | | | **switch** |
| : | , | >> | | | **:** |

| | | | | | |
|---|---|---|---|---|---|
| 3 | 2 | 4 | | | **3** |
| ; | , | ++ | | | **;** |
| middle | end | program terminates | | | **top** |
| end | middle | program terminates | | | **end** |
| 1 or 2 | -1 or 0 | 3 or 4 | | | **0 or 1** |
| for loop | if loop | while loop | | | **do-while loop** |
| always true | true or false | negative | | | **always true** |
| 3 | 1 | 4 | | | **3** |
| case | label | caption | | | **default** |
| declaring | Initialization | decrement | | | **Initialization** |
| a Boolean expression | either A or B | Neither A nor B | | | **a Boolean expression** |
| arrays | class objects | header files | | | **header files** |
| character | value and character | symbols | | | **value** |
| Inline | user-defined | undefined | | | **Inline** |
| functions | memoryspace | control structures | | | **memoryspace** |
| function polymorphism | overloading and polymorphism | operator overloading | | | **overloading and polymorphism** |
| called function | built in | function declaration | | | **called function** |
| member function | user-defined function | calling function | | | **member function** |
| middle | bottom | anywhere | | | **anywhere** |

## USER-DEFINED FUNCTIONS

**Why Functions in C?**

- Functions are used when certain type of calculations are repeated at many points in a program.
  Ex.: nCr = n! /r! (n-r)!
- As functions are reusable, saves time and space.
- Reduces the complexity of
  - writing
  - debugging
  - testing
  - maintaining
- User-Defined functions help in dividing the large programs into meaningful and independent modules (i.e) subprograms
- These subprograms are called functions

## ADVANTAGES OF FUNCTIONS

- Facilitates top-down approach
  - High level logic of overall problem is defined first
  - Functions are defined at the lower-level
- Easy to debug
- One function may be used many a times by the same program or by other programs (Reusability)

## TYPES OF C FUNCTIONS

i. **Library Functions ( Built-in)** eg. scanf(), printf(), sqrt() etc.,
(These functions are not written by users)
ii. **User-Defined Functions** eg. main()
(Written by users)

## COMPONENTS OF THE USER-DEFINED FUNCTIONS

i. **Function Declaration (or) Function Prototype:** The declaration of a function
ii. **Function Definition:** Independent Module written
iii. **Function Call:** Invoking the defined function. The program that calls the function is calling function

## FUNCTION DEFINITION (or) FUNCTION IMPLEMENTATTION

- Independent Module written for solving/ performing the problems/ operations

**General Format:**

```
function_type function_name (parameter_list) // semicolon is not used here
{
  local variable declaration;
   executable statements;
```

```
  …
 return statement;
}
```

**The Elements of Function Definition:**

- Function name |
- Function type | Function Header
- List of parameters |

- Local variables declaration |
- Function statements and | Function Body
- A return statement |

***Function header*** contains Function Type, Function Name and Arguments

- **Function type:** It specifies the type of value returned by the function. If return type is not specified, C assumes default data type int.
- **Function name:** It specifies the name of the module written. While naming the function, the rules of identifiers are to be followed.
- **Parameter list/ Arguments:** It serves as the input to the function. Parameters are also known as arguments. Parameters are to be separated by comma. It represents the actual values which is passed by calling function and known as formal *parameters*. They are used to send values to the calling program

**Example:**

```
float quadratic_root (int a, int b, int c)    // VALID FUNCTION DEFINITION
float quadratic_root (int a, b, c)            // INVALID FUNCTION DEFINITION
```

***Function Body*** Contains the local declaration and statements necessary for performing the required task.

- **Local Declaration** of variables needed by the function
- **Function statements** that perform the task of the function
- **A return statement** that returns the value computed by the function

**RETURN VALUES AND THEIR TYPES**

```
return(expression);
or
return;
or
if (error)
return;
```

- If the data type of return value does not match with the function type, it is suitably modified using C implicit typecasting

**FUNCTION DECLARATION (or) FUNCTION PROTOTYPE**

- User Defined Functions are declared in global or local.
- Parts of function declaration are

- Function Type
- Function Name
- Parameter List
- Termination Semicolon

- **GENERAL FORMAT**

  *function_type function_name (parameter_list);*

- **eg. int mul (int m, int n); // Function Prototype**

  **int mul (int, int ); // Other Valid Declarations**

  **mul (int m, int n);**

  **mul (int, int );**

**TYPES OF FUNCTION PROTOTYPE**

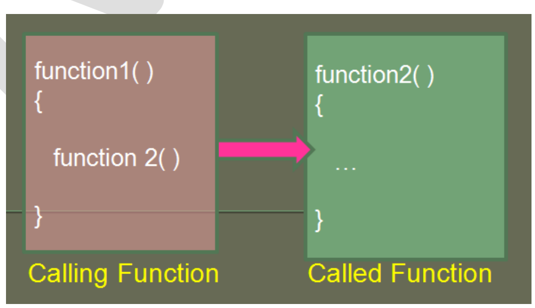
- Global Prototype
  - Prototype declared above all functions in the program
  - Available for all the functions in the program
- Local Prototype
  - Prototype declared within a function
  - Used by the function containing it.

**CATEGORIES OF FUNCTIONS**

Based on the presence and absence of variables and the return value, they are categorized into FIVE as:

1. Function with no arguments and no return value
2. Function with arguments and no return value
3. Function with no arguments and one return value
4. Function with arguments and return values
5. Function that return multiple values

**FUNCTIONS WITH NO ARGUEMNTS AND NO RETURN VALUE**




- Control of execution is transferred between these two functions
- Calling function does not pass any data to the called function
- Called function does not return any value to the calling function

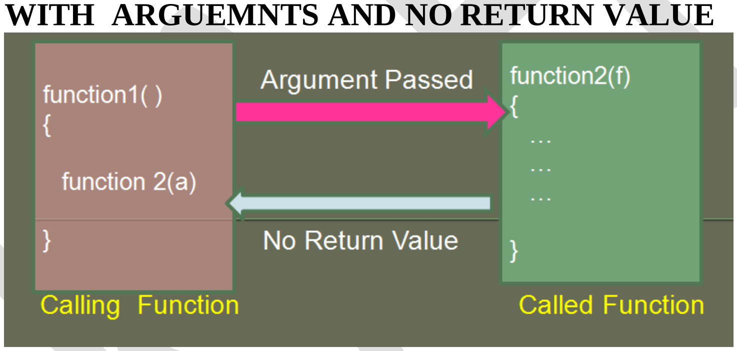
- No data transfer between the calling function and the called function
- No input from function1( ) to function2( )
- No return value from function2( ) to function1( )

**Example**

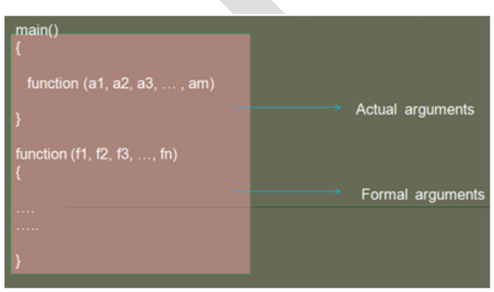
```
void sum() /*  Function Declaration*/
 void main()
 {
 Sum()  /* Function Call*/
 }
 /*Function Definition*/
 void sum()
 {
         int a=7,b=1;
         printf("Sum=%d", (a+b));
 }
```

**Output: Sum = 8**

**FUNCTIONS WITH ARGUEMNTS AND NO RETURN VALUE**




- Control of execution is transferred between these two functions
- Calling function passes argument(s) to called function
- Data is transferred from calling function to called function
- No return value from called function to calling function



- If actual arguments are more than formal arguments, the extra arguments are discarded (m > n)
- If actual arguments are less than formal arguments, the corresponding arguments are initialized with garbage values (m < n)
- When a function call is made, only a copy of the values of actual arguments is passed to the called function

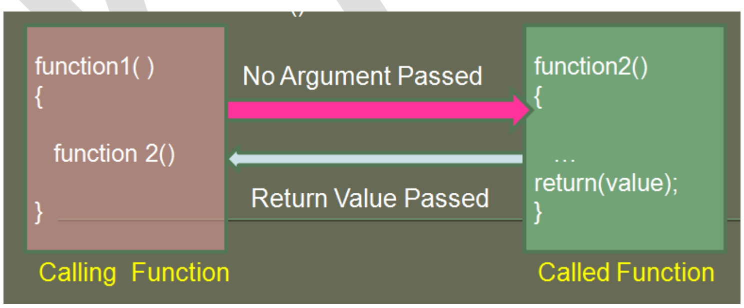
**Example**

```
void sum(int, int) /*  Function Declaration*/
 void main()
 {
         int a=7,b=1;
 Sum(a,b)  /* Function Call*/
 }
 /*Function Definition*/
 void sum(int m, int n)
 {
         printf("Sum=%d", (m+n));
 }
```

**Output: Sum = 8**

**FUNCTIONS WITH NO ARGUEMNTS AND RETURN A VALUE**

- Control of execution is transferred between these two functions
- Called function does not receive argument(s) from calling function
- Called function returns a value to the calling function
- No data transfer from calling function to called function
- A value is returned from called function to calling function




**Example**

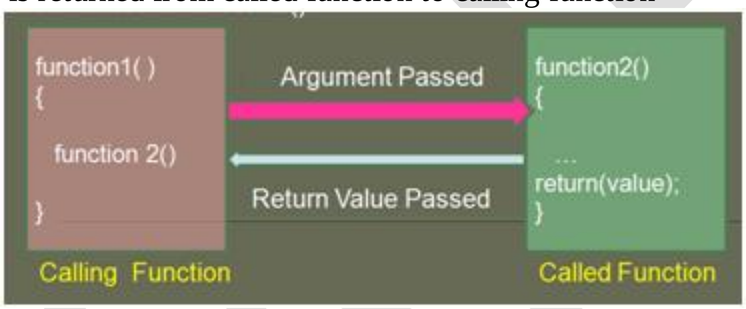
```
int sum() /*  Function Declaration*/
 void main()
 {
         int c;
 c = Sum()  /* Function Call*/
```

```
        printf("Sum=%d", c);
    }
    /*Function Definition*/
    int sum()
    {
        int a=7,b=1;
        return(a+b);
    }
```

**Output: Sum = 8**

**FUNCTIONS WITH NO ARGUEMNTS AND RETURN A VALUE**

- Control of execution is transferred between these two functions
- Called function receives argument(s) from calling function
- Called function returns a value to the calling function
- Data transfer are between from calling function to called function
- A value is returned from called function to calling function




**Example**

```
   int sum(int,int) /*  Function Declaration*/
    void main()
    {
        int a,b,c;
    c = Sum(a,b)  /* Function Call*/
        printf("Sum=%d", c);
    }
    /*Function Definition*/
    int sum(int m, int n)
    {
        return(m+n);
    }
```

**Output: Sum = 8**

**FUNCTIONS THAT RETURNS MULTIPLE VALUES**

- If we want to get more than one return value from a function, then additional arguments should be declared through which data can be received from the called function
- The list of arguments used to send out data from the called function is called output parameters.

- The output parameters perform the task of returning data to called function using
  - address operator (&)
  - indirection operator (*)

**Example:**

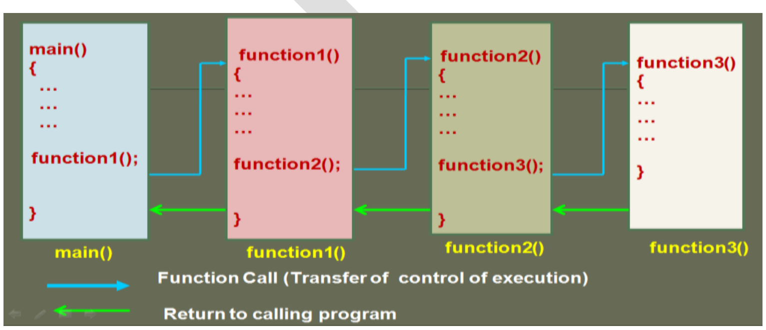
```
void add_sub(int x, int y, int *s, int *d);  // Function Prototype with arguments
main()
{
  int x = 20, y = 10, s, d;
  add_sub(x, y, &s, &d);          // Function call with actual arguments and output  parameters
  printf("/n Sum = %d\n  Difference =%d\n", sum, diff);
}

void add_sub (int a, int b, int *sum,  int *diff)  // Function with arguments & output  parameters
{
  *sum = a + b;
  *diff   = a – b;
}
```

**RULES FOR PASS BY POINTERS**

- The data type of actual and formal arguments are to be the same
- The actual parameters (given in the function call) must be the addresses of variables that are local to the calling function
- The formal arguments in the function header (in function definition), must be prefixed with indirection operator *
- In the function declaration (ie. function prototype) the arguments must be prefixed by *
- To access the value of actual arguments in the calling function, the corresponding formal arguments are prefixed with *

**NESTING OF FUNCTIONS**

- C permits the functions within the function.
- C provides main() function calling function1(), function1() calling function2(), function2() calling function3() and so on.




**Example:**

**int sum(int,int) /* Function Declaration*/**

```
void arith() /*  Function Declaration*/
 void main()
 {
         arith();
 }
 /*Function Definition*/
 void arith()
 {
         int a=7,b=1;
         printf("Sum=%d",sum(a,b));
 }
 int sum(int m, int n)
 {
         return(m+n);
 }
```

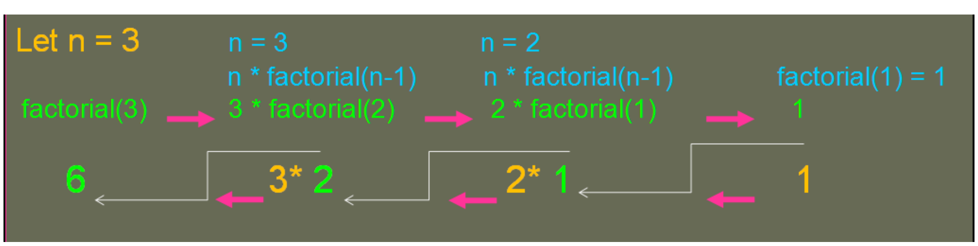
**Output: Sum = 8**

**RECURSION**

- Recursion is a special type of chained function call, in which a function calls itself

**Example:**

```
int factorial(int);
void main()
{
      int a;
      scanf("%d",&a);
      printf("Factorial =%d",factorial(a));
      getch();
 }
int  factorial(int n)                 // Definition of factorial() function
   {
      int fact;
      if (n == 1)
        return(1);
       else
         fact = n * factorial(n-1); // Function factorial() calls itself
      return(fact);
    }
```



**PASSING ARRAYS TO FUNCTIONS – RULES**

1. In function call statement, array name alone should be passed
   e.g. printf("%f\n", largest(value,4);
2. In function definition, the formal parameters must be an array type; The array size need not be specified
   e.g. float largest(float a[ ], int n)
3. The function prototype must show that the argument is an array
   e.g. float largest(float a[ ], int n);

**Example:**

```
main()   // Using One-Dimensional Array
{
  float largest(float a[ ], int n);              // Function Declaration - Local Prototype
  float value[4] = {2.5, 1.6, 7.5, 5.9};
  printf("%f\n", largest(value,4);           // Function call
                                             // In actual argument array name alone is given
                                             // Array dimension is not given
 }
float largest(float a[ ], int n)    // Function definition with formal arguments
                                    // Pair of brackets describe array
{
   int i;
   float max;
   for(i=0; i <= n; i++)
     if (max < a[i])
        max = a[i];
   return(max);
}
```

**PASS BY VALUE Vs. PASS BY REFERENCE ( Call by Value Vs. Call by Reference)**

- The technique of passing data to a function is called Parameter passing.
- The types are: Pass by Value & Pass by Reference

| PASS BY VALUE | PASS BY REFERENCE |
|---|---|
| **Known as Call by Value** | **Known as Call by Pointer or Pass by Address** |
| **The values of actual parameters are copied to formal parameters** | **Memory address of the variables are passed to the calling function** |
| **Called function works on copied values of calling function** | **Called function directly works on the data of the calling function** |
| **Original data of the calling function can not be changed accidentally** | **Original data of the calling function is changed by the called function** |
| **Example Program:**<br>**int sum(int,int)** | **Example Program:**<br>void swap(int *,int *); |

```
void main()
{
        int a,b,c;
c = Sum(a,b)
printf("Sum=%d", c);
}
        int sum(int m, int n)
{
        return(m+n);
}
```

**Output: Sum = 8**

```
void main()
 {
int x=200,y=100;
printf("Before swapping: x=%d, y=%d",x,y);
 swap(&x,&y)
 printf("After swapping: x=%d  y=%d",x,y);
  }
 swap(int *a,int *b)
  {
       int t;
        t=*a; *a=*b;*b=t;
    }
```

**OUTPUT**
**Before Swapping: x=200 , y=100**
**After Swapping : x=100 , y=200**

**STORAGE CLASSES :**

- The SCOPE, VISIBILITY and the LONGEVITY of the variables in C differ with respect to the storage class of the variable
- **Scope:** Determines the region of the program within which the variable is actually available for use
- **Visibility:** The accessibility of the variable from the memory
- **Longevity:** The period during which, the variable can retain its value during the execution of the program
- The types of storage classes are:
    - **Automatic (auto)**
    - **External (extern)**
    - **Static (static)**
    - **Register (register)**

| Storage Class | Declaration Point | Visibility | Lifetime |
|---|---|---|---|
| None | Before all functions in the file | Entire file and other files where variable is declared as extern | Entire Program |
| extern | Before all functions in the file extern and the file where originally declared as global. (can not be initialized within a function) | Entire file and Other files where variable is declared | Global |
| none or auto | Inside a function (or a block) | Only in that function or block | Until end of the function or block |
| register | Inside a function or block | Only in that function or block | Until end of the function or block |
| static (external) | Before all functions in a file | Only in that file | Global |

| | | | |
|---|---|---|---|
| static (internal) | Inside a function | Only in that function | |

## Arrays

**Introduction:**

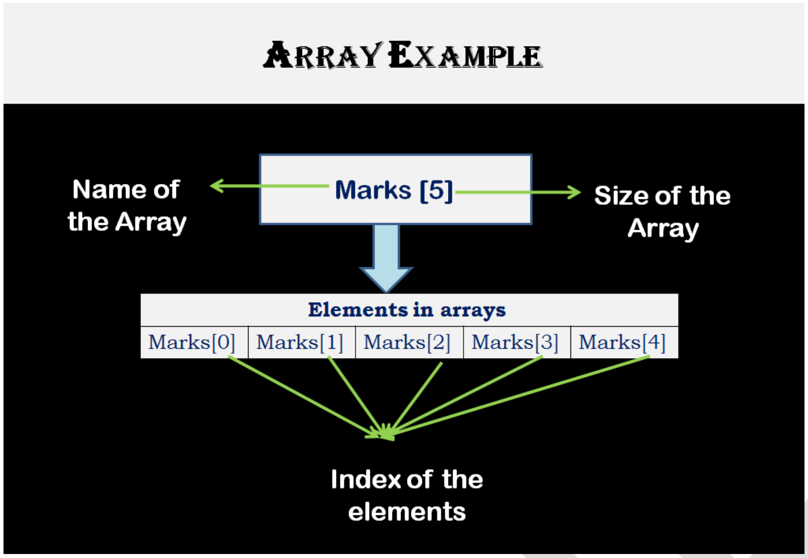
- Need for the Arrays:
  - A variable can store a only one value at a time.
  - A variable handles limited amounts of data.
- Advantages of Arrays
  - Used to handle the large Volumes of data in terms of reading, processing and printing.
  - Facilitate efficient storing, accessing and manipulation of data items.
  - Use a single name to represent a collection of items

**Definition & Facts**

- Array is a fixed-size sequenced collection of elements of the same data type and shares the common data name.
- Array is a derived data type. Because it builds on primary data type.
- Array is a one of the Data Structure in C, because it provides a convenient structure to represent data.
- Array is allocated by continuous memory locations.

**Examples**

- List of Temperature recorded every hour in a day or a month or a year.
- List of employees
- List of Products and their costs
- Marks of a class of students
- List of Customers and their telephone numbers
- Table of daily rainfall data




**Types of Arrays:**

- One Dimensional Arrays
- Two Dimensional Arrays
- Multi Dimensional Arrays

**ONE DIMENSIONAL ARRAY**

- A list of items can be given one variable name using only one subscript and such a variable is called a *'Single subscripted variable or a one-dimensional array'*
- The subscripts of an array can be *integer constants*, *integer variables.*
- *C performs no bounds checking and ensure that the array indices are within the declared limit.*
- Like other variables, arrays must be declared before they are used so that the compiler can allocate space for them in memory.
- The general form of array declaration is

  ***type variable_name[Size];***

  – The *type* specifies that type of element that will be contained in the array such as *int, float or char*
  – The *size* indicates the maximum number of elements that can be stored inside the array.

**One Dimensional Array Declaration**

- Valid Declarations:
  - ***float a[10]*** – ***a*** contains 10 real elements.
    - Any subscript 0 to 10 are valid.
  - ***int b[10]*** - it contains maximum 10 elements.
  - ***char s[10]*** - array of characters.

- size of character array represents the maximum number of characters to be stored in array.

- Any reference to the arrays outside the declared limits would not cause error. It might result in unpredictable results.
- The size should be ***numeric or symbolic constants.***

**One Dimensional Array – Initialization**

- Arrays can be initialized in two ways
  - At Compile Time
  - At Run Time

***Compile Time Initialization***

General Form

***type array_name[size] = { list of Values};***

The values in the list are separated by commas.

- ***Valid Initialization***
  - ***int number[3] ={0,0,0};***
  - ***float mark[5] ={0.0,1.5,2.5};***
  - ***int Counter[ ] = { 1,2,3,4};***
  - ***char name[ ]={ 'J','u','s','t','\0'};***
  - ***char name[ ] = "Just";***
- If the numbers of assigned elements are less than the size of the array, the remaining(uninitialized values) takes **zero or NULL** values
  - int a[5] = { 10,20} – a[0] =10, a[1]=20
    Remaining elements a[2],a[3] & a[4] assign as 0.
  - char city[5]={'B'} - city[0] takes 'B'
    city[1] takes '\o' or NULL and remaining elements take garbage values
- If the more elements are initialized than declared size, then the compiler returns an error.
  - int number[3] ={1,2,3,4,5}; // illegal initialization

**Runtime Initialization**

- An array can be *explicitly initialized at runtime.*
- The values of the elements in arrays is assigned/read by ***scanf()*** and it also use the ***for*** loop/ any loop structure.

```
//initialization – I
 for(i=0;i<100;i++)
  {
    if i<50
      sum[i] = 0.0;
    else
      sum[i] = 1.0;
  }
```

```
//initialization – II
   int a[3];
   scanf("%d%d%d", &a[0], &a[1], &a[2]);
```

```
//initialization – III
   int a[3],i;
   for(i=0; i<3;i++)
   scanf("%d", &a[i]);
```

*Example Program*

**// Average of 'n' numbers**

```
void main()
{
  int a[5], i, s=0;
  float avg;
  clrscr();
  printf("Enter the Array elements\n"):
    for(i=0;i<5;i++)
        scanf("%d",&a[i]);
    for(i=0;i<5;i++)
         s=s+a[i];
     avg=s/5;
   printf("Average of 5 Numbers: %f", avg);
   getch();
}
```

**Output**

```
Enter the Array elements:
 1
 2
 3
 4
 5
Average of 5 Numbers: 3
```

***TWO DIMENSIONAL ARRAY***

- *A table of items can be given one variable name using two subscript and such a variable is called as "Two Dimensional Array".*
- *Declaration*

  ***type array_name [row_size] [column_size];***

  - *The type represents the data type of the array*
  - *Row Size & Column Size represents the total numbers of elements to be stored in an array.*
- *In two dimensional array, the first index selects the row and the second selects the column within that row.*

- *Storage of Two Dimensional array in memory:*

| Col. 0 | Col. 1 | Col. 2 | |
|---|---|---|---|
| 310 | 20 | 24 | Row 0 |
| [0][0] | [0] [1] | [0][2] | |
| 45 | 76 | 89 | Row 1 |
| [1][0] | [1][1] | [1][2] | |
| 67 | 90 | 78 | Row 2 |
| [2][0] | [2][1] | [2][2] | |

***Two Dimensional Array – Initialization***

- *Two dimensional arrays may be initialized at compile time and runtime.*
- *At compile time, list or table of values assign to elements of two dimensional array.*
    - ***int table[2][3]= {0,0,0,1,1,1};***

      *The first three elements initialize to first row, the rest of the variable assign to second row.*
    - ***int table[2][3]={{0,0,0},{1,1,1}};***

      *The first inner brace's values initialize to first row, the second inner brace's values variable assign to second row.*
    - ***int table[ ][3]={{0,0,0},{1,1,1}};***

      *The array is completely initialized with all values, explicitly, the first dimension of the array need not to specify .*
    - ***int table[2][3]={{0,0},{1,1,1}};***

      *Uninitialized elements takes **Zero or Null Values**.*

- *At runtime, the elements assign by values dynamically using scanf().*

```
int a[2][3],r,c;
   for(r=0; r<2;r++)   // for Row
     for(c=0;c<3;c++) //for Column
           scanf("%d", &a[r][c]);
```

***Example Program***

**// Average of elements of the Matrix**

```
void main()
{
  int a[3][3], r,c s=0;
  float avg;
  clrscr();
  printf("Enter the Matrix\n"):
   for(r=0;r<3;r++)
        for(c=0;c<3;c++)
            scanf("%d",&a[r][c]);
    for(r=0;r<3;r++)
        for(c=0;c<3;c++)
            s=s+a[r][c]
   avg=s/5;
  printf("Average of 5 Numbers: %f", avg);
  getch();
}
```

**Output**

```
Enter the Matrix:
 1  2  3
 4  5  6
 7  8  9
Average of 5 Numbers: 5
```

***MULTIDIMENSIONAL ARRAYS***

- *C allows arrays of three or more dimensions. These arrays are called "Multidimensional Arrays"*
- *The general Form:*

  ***type array_name [$s_1$] [$s_2$] [$s_3$] [$s_4$] …. [$s_m$] ;***
- *ANSI C does not specify any limit of an array dimension.*

***Example***

***int survey [3] [5] [12];***

- *survey is a three dimensional array declared to contain 180 integer values.*
- *The first subscript represents cities.*
- *The second and third subscript represents the rainfall and month respectively.*

***Example Program***

```
void main()
{
  int a[2][2][2], r,c,z,s=0;
  float avg;
  clrscr();
  printf("Enter the elements\n"):
  for(z=0;z<2;z++)
    for(r=0;r<2;r++)
            for(c=0;c<2;c++)
                    scanf("%d",&a[z][r][c]);
    for(z=0;z<2;z++)
       for(r=0;r<2;r++)
            for(c=0;c<2;c++)
                    s=s+a[z][r][c]
    avg=s/5;
   printf("Average : %f", avg);
   getch();
}
```

**Output**

```
Enter the elements:
   1  2
   3  4

    5  6
    7  8
Average : 3.250000
```

*STATIC ARRAYS*

- The process of allocating memory at compile time is known as **static memory allocation.**
- The arrays that receive static memory allocation are called **static arrays.**

*DYNAMIC ARRAYS*

- In C, it is possible to allocate memory to arrays at runtime. This feature is known as **Dynamic Memory Allocation**.
- The arrays created at run time are called **"Dynamic Arrays".**
- Dynamic arrays are created using **pointer variables** and memory management functions **malloc, calloc and realloc.** These functions are included in the header file **<stdlib.h>.**
- The dynamic arrays are used in creating and manipulating data structure like linked list, stack, queue.

***Applications of Arrays***

*They include:*

– Using printers for accessing arrays
– Passing arrays as function parameters
– Arrays as members of structures
– Using structure type data as array elements
– Arrays as dynamic data structures
– Manipulating character arrays and strings

# CHARACTER ARRAYS

- String is a sequence of characters treated as a single data type.
- C does not have a data type as string. Hence, string is declared as a character array
- The size of the character array determines the number of characters in a string.
- String is a variable-length structure stored in a fixed-length array
- The size of the string is to be smaller than the size of the array. So, the last element of the sting should be shown explicitly.
- When compiler assigns characters to a string, it automatically assigns ‘\0’ (null character) at the end of the string, to terminate it.
- The size of the character array should be : the maximum size of string + 1

Syntax :

**char string_name[size];**

- e.g. char name[30];

- String variables can be initialized during
  - ✓ Compile-time (Static) or
  - ✓ Run-time (Dynamic)

***Initializing String Variables during Compile Time***

| | |
|---|---|
| char name[6] = “BENNY”; | B \| E \| N \| N \| Y \| \0 |
| char name[6] = {‘B’,’E’,’N’,’N’,’Y’}; | B \| E \| N \| N \| Y \| \0 |
| char name[ ] = {‘T’,’O’,’M’}; /* without specifying the size */ | T \| O \| M \| \0 |
| char name[10] = {‘T’,’O’,’M’}; | T \| O \| M \| \0 \| \0 \| \0 \| \0 \| \0 \| \0 \| \0 |
| char name[6] = {‘F’,’E’,’N’,’N’,’E’,’R’}; | Compiler Error: Too many Initializers |
| char name[7] = {‘G’,’R’,’I’,’”’,’D’,’U’}; | G \| R \| I \| “ \| D \| U \| \0 |
| char name[7] = “BEN”NY”; | Compiler Error: Declaration Syntax Error<br>Non-terminated string or character constant |

- While initializing during declaration, the string length should not exceed the maximum elements size
- Initialization cannot be separated from declaration, as
  - ***char name[6];***
  - ***name[6] = "BENNY";***
- Array name cannot be used, as used as the left operand of an assignment operator
- ***char name[6] = "BENNY";***
  - ***name1[6] = name[6];***
- Null character serves as the 'end-of-string' marker

***READING STRINGS FROM TERMINALS***

***Using scanf() Function:***

- scanf( ) function with %s format specification is used to read the string of characters
- Ampersand(&) is not required before character array variable name. Because the character array is also a pointer.
- scanf( ) function terminates when it encounters a white spaces (blank, tab, carriage return, form feed and new line)
- Example:

```
char name[10];
scanf("%s", name);
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **G** | **R** | **I** | **\0** | **?** | **?** | **?** | **?** | **?** | **?** |

If the input is GRI DU then GRI is assigned to name

- Using scanf( ), variable cannot read a string with white space.
- To achieve this, two character arrays may be used one to store GRI and the other to store DU
- The field width w can be specified in scanf( ) using %ws
  - The entire input string is stored in the character array, if the width w is equal or greater than the number of characters typed in
  - If the width is less than the number of characters in the input string, the excess characters get truncated and are not read in

- To read more than one word, edit set conversion code %[..] is used.
  Example:

```
char line [80];
scanf("%[^\n]", line);
```

printf("%s", line);

scanf("%[^\n]", line); reads sequence of strings including whitespace until the new line character entered.

**Using getchar() function**

- getchar() is used for reading the single character from the terminal.
- Reading input character is terminated when a new line character is read.
- The null character is appended at the end of the read input string

- Syntax

  **Variable_name=getchar();**
- **Example:** char a;
  a=getchar();

**Program:**

```
#include<stdio.h>
void main()
{
  char a;
  a=getchar();
   printf(" Character using getchar()
is %c", a);
}
output;:
J
Character using getchar() is J
```

**Using gets() function**

- gets( ) reads string of characters including whitespace into a character array
- It reads string until new line character is read and then null character is appended at the end of the read input string
- Syntax: **gets(variable_name);**
- **Example Program**

```
#include<stdio.h>
void main()
{
char str[80], str1[80];
gets(str);
printf("%s", str);
printf("\n%s", gets(str1));
puts(str1);
}
```

**WRITING STRINGS ON TERMINALS ( using %s)**

- %s is used to display the elements in the character array up to null terminator. (Without Formatting)
  Example: name[10] = {'G','R','I',' ',,'D','U'};
  printf("%s", name);
- Formatted Output : %w.p
  w is the field width and p is the precision
- Example 1: printf("%10.4s", name);
  (Out of field width10, first four characters are displayed)

  Example 2: printf("%-10.4s", name);
  /* string to be printed is left-justified */
- Variable Field Format : printf("%*.*s", w,p,name);
  where w is the field width and p is the precision.

**WRITING STRINGS ON TERMINALS: putchar() and puts()**

- putchar( ) displays the value of the character constant stored in character variable.
  Ex: char ch ='a';
  putchar(ch);

// displays a String data can also be displayed using putchar()

Ex: char name[6]= "gandhi";
for (i=0; i<5; i++)
putchar(name[i]);

- puts() directly displays a string stored in a character array
  Ex: char name[6];
  gets(name);
  puts(name);

**STRING MANIPULATION FUNCTIONS**

| | | |
|---|---|---|
| strcat( ) | strcat(string1, string2); | To concatenate two strings<br>Appends string2 to string1 |
| strcmp( ) | strcmp(string1, string2); | To compare two strings |
| strcpy( ) | strcpy(string1, string2); | Copies string2 to string1 |
| strlen( ) | strlen(string); | Finds the length of string |
| strncpy( ) | strncpy(string1, string2, n); | Copies first n characters of string2 to string1 |
| strncmp( ) | strncmp(s1, s2, n); | Compares leftmost n characters of s1 and s2<br>0 if s1 == s2 ; Negative , s1 < s2;<br>Positive, s1 > s2 |

| | | |
|---|---|---|
| strncat( ) | strncat(s1, s2, n); | Concatenates leftmost n characters of s2 to the end of s1 (i.e. appends) |
| strstr( ) | strstr(s1, s2); | Searches s1 to check if s2 is in s1 |
| strchr( ) | strchr(s1, 'a'); | To locate the first occurrence of 'a' in s1. |
| strrchr( ) | strrchr(s1, 'a'); | To locate the last occurrence of 'a' in s1. |

## POSSIBLE QUESTIONS

## PART B

### (Each Question Carries 2 marks)

1. Write a program in C language to convert the given string in uppercase
2. What do you mean by formal arguments and the actual arguments?
3. What do you mean by functions?
4. What are parameter passing?
5. Define array and how we can access elements of an array?
6. What is the difference between character and string?
7. Write difference between character and string?
8. What is the purpose of the return statement?
9. What is array overflow?
10. Explain function prototyping.

## PART C

### (Each Question Carries 8 marks)

1. What are user define function its advantages and write its different types.
2. What are String functions and write some string function?
3. Write a program to find sum of two matrices.
4. Write a program in c to reverse a given string.
5. Explain with example the concept of passing array to function.
6. What is recursion? Write a program to find Fibonacci series till a given number using recursion.
7. Write a program to find factorial of a number using recursion.
8. Write a program to find the sum of the rows, column, and diagonal elements of a matrix.
9. Write a program to multiply any two matrixes.
10. Explain with example the relationship of one dimensional array and pointers.

*********

| S.No | Question | Choice1 |
|---|---|---|
| 1 | C++ supports all the features of ____________ as defined in C | structures |
| 2 | A structure can have both variable and functions as _________ | objects |
| 3 | The class __________ describes the type and scope of its members | Functions |
| 4 | The class ___________ describes how the class function are implemented | Function definition |
| 5 | The keywords private and public are known as __________ labels | Static |
| 6 | The class members that have been declared as _________ can be accessed only from within the class | Private |
| 7 | The class members that have been declared as _________ can be accessed from outside the class also | Private |
| 8 | The variables declared inside the class are called as __________ | Function variables |
| 9 | The functions which are declared inside the class are known as ______ | Member function |
| 10 | The class variables are known as _________ | Functions |
| 11 | The symbol ______ is called the scope resolution operator | >> |
| 12 | A member function can call another member function directly without using the __________ operator | Assignment |
| 13 | A ______ member variable is initialized to zero when the first object of its class is created | Dynamic |
| 14 | __________ Variables are normally used to maintain values common to the entire class. | Private |
| 15 | When a copy of the entire object is passed to the function it is called as __________ | Pass by reference |
| 16 | When the address of the object is transferred to the function it is called as __________ | pass by reference |
| 17 | A _________ function can be invoked like a normal function without the help of any object | Void |
| 18 | The _________ member variables must be defined outside the class. | Static |
| 19 | A friend function, although not a member function, has full access right to the ______ members of the class | Static |
| 20 | ___________ enables an object to initialize itself when it is created | Destructor |
| 21 | _________ destroys the objects when they are no longer required | Destructor |
| 22 | The ___________ is special because its name is the same as the class name. | Destructor |

| | | |
|---|---|---|
| 23 | A constructor that accepts no parameters is called the __________ constructor | Copy |
| 24 | Constructors are invoked automatically when the ________ are created | Data |
| 25 | Constructors cannot be _________ | Inherited |
| 26 | Constructors cannot be _________ | Destroyed |
| 27 | Constructors make _________ calls to the operators new and delete when memory allocation is required | Explicit |
| 28 | The constructors that can take arguments are called _________ constructors | Copy |
| 29 | The constructor function can also be defined as ________ function | Friend |
| 30 | When a constructor can accept a reference to its own class as a parameter, in such cases it is called as _________ constructors | Multiple |
| 31 | When more than one constructor function is defined in a class, then the constructor is said to be _________ | Multiple |
| 32 | C++ complier has a _________ constructor, which creates objects, even though it was not defined in the class. | Explicit |
| 33 | A _________ constructor is used to declare and initialize an object from another object | Default |
| 34 | The process of initializing through a copy constructor is known as ________ initialization | Overloaded |
| 35 | A ______ constructor takes a reference to an object of the same class as itself as an argument | Delete |
| 36 | Allocation of memory to objects at the time of their construction is known as ________ construction | Static |
| 37 | We can create and use constant objects using ______ keyword before object declaration. | Static |
| 38 | A destructor is preceded by ______ symbol | Dot |
| 39 | _________ is used to allocate memory in the constructor | Delete |
| 40 | _________ is used to free the memory | new |
| 41 | Which is a valid method for accessing the first element of the array item? | item(1) |
| 42 | Which of the following statements is valid array declaration? | int number (5); |
| 43 | An object is an _________ unit | group |
| 44 | Public keyword is terminated by a ________ | Semicolon |
| 45 | Private keyword is terminated by a _________ | semicolon |
| 46 | The memory for static data is allocated only ________ | twice |

| | | |
|---|---|---|
| 47 | Static member functions can be invoked using _________ name | class |
| 48 | When a class is declared inside a function they are called as _________ classes. | global |
| 49 | ___________ can be virtual | destructors |
| 50 | The __________ doesn't have any argument | constructor |
| 51 | The __________ also allocates required memory . | constructor |
| 52 | Any constructor or destructor created by the complier will be ________ | private |
| 53 | The class can have only ______ destructor | two |
| 54 | __________ cannot be overloaded | destructor |
| 55 | __________ releases memory space occupied by the objects | constructor |
| 56 | Constructors and destructors are automatically inkoved by __________ | operating system |
| 57 | Constructors is executed when _________ | object is destroyed |
| 58 | The destructor is executed when ___________ | object goes out of scope |
| 59 | The members of a class are by default _________ | protected |
| 60 | The _________ is executed at the end of the function when objects are of no used or goes out of scope | destructor |

| Choice2 | Choice3 | Choice4 | | | Ans |
|---|---|---|---|---|---|
| union | objects | classes | | | **structures** |
| classes | members | arguments | | | **members** |
| declaration | objects | variables | | | **declaration** |
| declaration | arguments | paramenter | | | **Function definition** |
| dynamic | visibility | const | | | **visibility** |
| public | static | protected | | | **Private** |
| Public | static | protected | | | **Public** |
| data members | member function | declarations | | | **data members** |
| member variables | data variables | constants | | | **Member function** |
| members | objects | structures | | | **objects** |
| :: | << | ::* | | | **::** |
| equal | dot | greater than | | | **dot** |
| constant | static | protected | | | **static** |
| protected | Public | static | | | **static** |
| pass by function | pass by pointer | pass by value | | | **pass by value** |
| pass by function | pass by pointer | pass by value | | | **pass by reference** |
| friend | inline | built in | | | **friend** |
| private | public | protected | | | **Static** |
| private | public | protected | | | **private** |
| constructor | overloading | overriding | | | **constructor** |
| constructor | overloading | overriding | | | **Destructor** |
| static | constructor | dynamic | | | **constructor** |

| | | | | | |
|---|---|---|---|---|---|
| default | multiple | single | | | **default** |
| classes | objects | union | | | **objects** |
| destroyed | encaptulated | abstraction | | | **Inherited** |
| virtual | static | dynamic | | | **virtual** |
| implicit | function | header | | | **implicit** |
| multiple | parameterized | levels | | | **parameterized** |
| inline | default | numeric | | | **inline** |
| copy | default | implicit | | | **copy** |
| copy | default | overloaded | | | **overloaded** |
| default | implicit | user defined | | | **implicit** |
| copy | multiple | parameterized | | | **copy** |
| multiple | copy | single | | | **copy** |
| new | copy | update | | | **copy** |
| copy | dynamic | user defined | | | **dynamic** |
| new | const | sample | | | **const** |
| asterisk | colon | tilde | | | **tilde** |
| binding | free | new | | | **new** |
| delete | clrscr() | update | | | **delete** |
| item[1] | item[0] | item(0) | | | **item[0]** |
| float avg[5]; | double [5] marks; | counter int[5]; | | | **float avg[5];** |
| individual | three | multiple | | | **individual** |
| comma | dot | colon | | | **colon** |
| comma | dot | colon | | | **colon** |
| thrice | once | sigma | | | **once** |

| | | | | | |
|---|---|---|---|---|---|
| object | data | function | | | **class** |
| invalid | local | private | | | **local** |
| constructors | static | dynamic | | | **destructors** |
| copy constructor | destructor | level of degrees | | | **destructor** |
| destructor | dynamic | functions | | | **constructor** |
| public | protected | global | | | **public** |
| many | one | four | | | **one** |
| constructor | friend | private | | | **destructor** |
| destructor | malloc | calloc | | | **destructor** |
| main() | complier | object | | | **complier** |
| object is declared | new keyword is invoked | derefernced memory | | | **object is declared** |
| when object is not used | when object contains nothing | null value detected | | | **object goes out of scope** |
| private | public | new | | | **private** |
| constructor | inheritance | class | | | **destructor** |

## Structure and Union

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly, **structure** is another user-defined data type available in C that allows 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. The format of the struct statement is as follows:

```
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 the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program:

```
#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;
    struct Books Book2;
/* Declare
Book1 of type
Book */ /*
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:

```
Book1 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 the same way as you pass any other variable or pointer.

```
#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;
    struct Books Book2;
/* Declare
Book1 of type
Book */ /*
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 the same way as you define pointer to any other variable:

```
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 rewrite the above example using structure pointer.

```
#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;
    struct Books Book2;          /* Declare Book1 of type Book */ /*
                                 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
```

**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 include:

- 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 to 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, while others would store the next field in the next word.

A **union** is a special data type available in C that allows 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 multiple-purpose.

**Defining a Union**

To define a union, you must use the **union** statement in the same way as you did while defining a 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 having 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. It means a single variable, i.e., 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 the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union −

Live Demo

```
#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 the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

Live Demo

```
#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 the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **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 unions −

Live Demo

```
#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
```

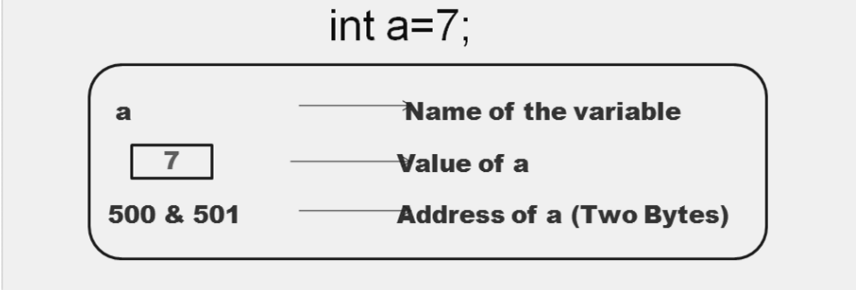
Here, all the members are getting printed very well because one member is being used at a time.

## POINTERS

***Pointer variable:*** The variable that points/holds the address of the another variable is called ***the pointer variable***.

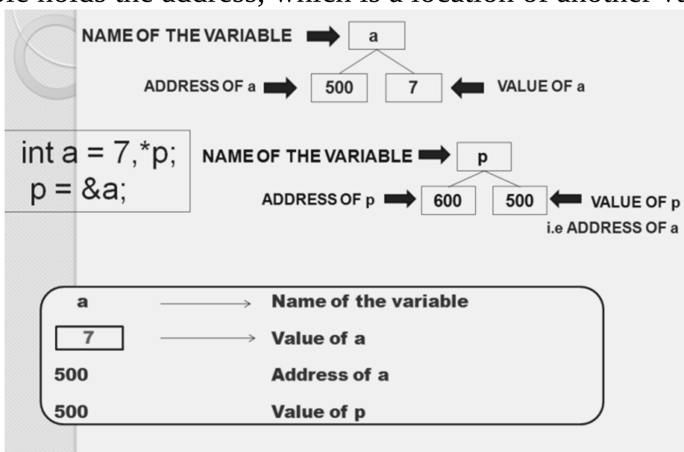
***Representation of Variable:***

- When a variable is declared, the system allocates an appropriate memory location to hold the value of the variable.
- The number of bytes allocated to each variable is decided by its data type.



***Representation of a Pointer Variable:***

- Memory address of the variable is assigned to some other variable that can be stored in memory.
- A pointer variable holds the address, which is a location of another variable.




***Components of Pointers:***

- **Pointer Constants:** Memory address allotted to a variable (Address of i) (It is machine-dependent; It cannot be changed)
- **Pointer Value:** The memory address of the variable, stored in the pointer variable, is known as Pointer Value. It is done using Address Operator (&). [ eg. p = &a; ]
- **Pointer Variable:** The variable that contains a pointer value is called a Pointer Variable. ( p is the pointer variable)

***Accessing the Address of the variable***

- The address of the variable is assigned to a pointer variable, using the address operator (&).
- While reading the inputs through scanf(), the address operator is used.
- The operator & that immediately precedes a variable, returns the address of the variable associated with it.
- It can be used with a simple variable or an array element.

***Example :***

```
void main()
{
  int a=75;
 char b='J';
printf("%d  is stored at address %u",a,&a);
printf("%c  is stored at address %u",b,&b);
}
```

***Declaration of Pointer Variable:***

- Pointer variable should be declared with a data_type
- The data_type of the pointer variable and the data_type of variable (pointed by the pointer variable) must be the same.
- Syntax of pointer variable declaration is:

  **data_type *pt_name;**

- The asterisk(*) indicates that the variable is a pointer variable

***Illustration***

int *ip ; *// Pointer to an integer Variable*
float *fp;
char *cp;

**Pointer Declaration Style**

Style 1: *int* p;*
Style 2: *int *p;*
Style 3: *int * p;*

**Initialization of Pointer Variables:**

- The process of assigning the address of a variable to a pointer variable is known as initialization.
- All uninitialized pointers will have some unknown values (garbage) that will be an invalid memory addresses. So, uninitialized pointers will point to some values that are wrong.
- Hence, uninitialized pointers will produce erroneous results in the programs.

**Example:**

```
int  a;
int  *p;      /*Declaration*/
p= &a;        /*Initialization*/

float a;
int  *p;      /*Declaration of integer pointer */
p= &a;  /*Wrong Initialization; data_type mismatch*/
```

- *The above pointer variable initialization is wrong , because the address of* ***float variable is assigned*** *to an* ***integer pointer***

int x, *p=&x; **/*Valid */**
int *p=NULL; **/*Valid; Initialized with null Value*/**
int *p=0; **/*Valid; initialized with zero */**
int *p=&x, x; **/* Invalid; Pointer variable is assigned with the address of the variable x which is declared later */**

***Pointer Flexibility***

- **Same pointer can point to different data variables, (one after the other, simultaneously ; not at the same time)**

*int x,y,z,*p;*
*p=&x; // p contains the address of x*
*p=&y; // p contains the address of x*
*p=&z; // p contains the address of x*

- **Different pointers can point to the same data variable.**

*int x,*p1,*p2,*p3;*
*p1=&x; // p1 contains the address of x*
*p2=&x; // p2 contains the address of x*
*p3=&x; // p3 contains the address of x*

***Accessing a variable through its pointer***

- A pointer variable is assigned with the address of a variable using the operator *
- The asterisk operator is also called as indirection operator or dereferencing operator

```
p=&a;  // Address of a assigned to p
n=*p;  //  Value stored in the value of p is assigned to n
```
= n=*&a; = n=a;

***Example:***

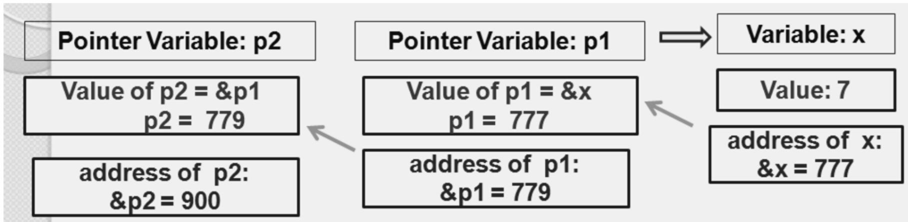
```
void main()
{
  int x,y.*p; // Declaration of variables & Pointer Variable p
   x=10;     //  Initialization of x
   p=&x;     // Assigns address of x to p
   y=*p;     // Assigns the value stored in value of p
   printf("\n 1. x Value :%d",x);
   printf("\n 2. x Value :%d,  Address: %u",x, &x);
   printf("\n 3. x Value :%d, Address: %u",*&x, &x);
   printf("\n 4. x Value :%d, Address: %u",*p, p);
   printf("\n 5. x Value :%d, Address: %d",x, &p);
   printf("\n 6. y value :%d, Address: %d",y, &y);
   *p=15;
   printf("\n 7. X value :%d",x);
}
```

**OUTPUT**

```
1. X value : 10
2. X value : 10
Address:777
3. Value : 10    Address:
777
4. Value : 10    Address:
777
5. Value : 777   Address:
770
6. Y value : 10
```

***Chain of Pointers***

- The mechanism of a pointer to point another pointer makes the **chain of pointers**.




***Example:***

```
main()
{
  int x,*p1,**p2;
   x=7;
   p1=&x;
   p2=&p1;
   printf("\n 1. *p1=%d",*p1);
   printf("\n 2. **p2=%d",**p2);
   printf("\n 3. p1=%u,&p1=%u",p1,&p1);
   printf("\n 4. p2=%u",p2);
}
```

**OUTPUT**
1. *p1= 7
2. **p2=7
3. p1= 777, &p1=779
4. p2= 779

***Pointer Expressions***

- *Pointer variables p1 and p2 can be used in expressions.*
- *They can be employed in arithmetic and relational expressions like other variables.*

```
y = *p1 - *p2;
sum=sum+ *p1;
z = 5+ *p2/ *p1;    // 5+(*p2/*p1)
```

- *Pointer variables can be used with shorthand operator.*

```
*p1++;
  -*p2;
sum+=  *p1;
```

- *Pointer variables can be compared using relational operators*

```
  *p1>*p2
*p1==*p2
*p1!=*p2
```

***Pointer increments and Scale Factor***

- *Pointer variable can incremented.*

```
p1=p1+1;
p1=p2+2;
p1++;
```

*If p1 is an integer pointer with an initial value,777 then after the operation p1=p1+1, the value of p1 is 779, not 778.*

- *When a pointer variable increments, its value is increased by the length of the data type that it points to. This length called the Scale Factor.*

- *The number of bytes used to store various data types depends on the system and can be found by making use of the sizeof operator.*

***Rules of Pointer Operations***

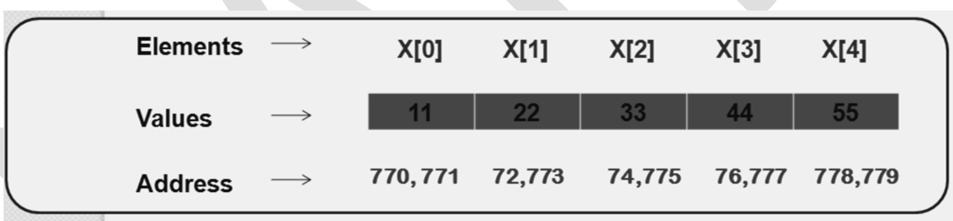
- *A pointer variable can be assigned with the address of the another variable.*
- *A pointer variable can be assigned with the values of another pointer variable.*
- *A pointer variable can be initialized with NULL or Zero value.*
- *A pointer variable can be pre-fixed or post-fixed with increment or decrement.*
- *An integer value may be added or subtracted from a pointer variable.*
- *When two pointers point to the same array, they can be compared using relational operator.*
- *A pointer variable cannot be multiplied by a constant.*
- *Two pointer variables cannot be added.*
- *An arbitrary value cannot be assigned as an address to a variable*

*(i.e &x=10; is illegal)*

***Pointers and Arrays***

- *When an array is declared, the compiler allocates the base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.*
- *The base address is the location of the first element of array. The compiler also defines the array name as a constant pointer to the first element.*

***int x[5] = {11,22,33,44,55}***




- *Let p is an integer pointer and let p points to the array x*

***p = x; implies that p = &x[0];***

- *Every value (item) of x can be accessed by incrementing the pointer variable.*

***p = &x[0] (ie 777)***
***p+1 = &x[1] (ie 779 = 777+(1 × 2), where 1 is the index & 2 is the Scale Factor***
***p+2 = &x[2] (ie 781 = 777+ (2 × 2)***
***p+3 = &x[3] (ie 783 = 777+ (3 × 2)***
***p+4 = &x[4] (ie 785 = 777+ (4 × 2)***

- *The address of an element is calculated using its index and the scale factor of the data type*
***Address of an element = base address + (index × scale factor of data type)***
- *The pointer accesses the values of an array faster than array indexing.*

**** (p+2) = x[2]***
****(p+2) gives the value of x[2]***

***Example:***

```
main()
{
  int sum,i,*p;
  int x[5]={5,9,6,3,7};
  i=0;
  p=x;
  printf("ELEMENT     VALUE     ADDRESS\n");
 while(i<5)
  {
   printf("x[%d] \t     %d \t     %u\n",i,*p, p);
    sum=sum+p;
     i++;
    p++;
          }
   printf("\n Sum     = %d\n",sum);
   printf("\n &x[0]    = %u\n",&x[0]);
   printf("\n p        = %u\n",p);
  }
```

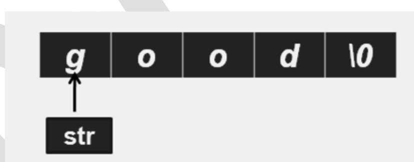
```
OUTPUT
ELEMENT     VALUE     ADDRESS
  x[0]        5         770
  x[1]        9         772
  x[2]        6         774
  x[3]        3         776
  x[4]        7         778
  Sum    =  55
  &x[0]  = 770
  p      = 780
```

***Pointers and Character Strings***

- *C supports the method to create strings using pointer variables of type char*

  ***char str[5]="good";***
  ***char  *str="good";***

- *The compiler automatically inserts the null character'\0' at the end of the string. The string pointer stores its address in the pointer variable str.*




***Example Program: Palindrome Checking***

***Pointers and Functions***

***<u>Pointers as Function Arguments</u>***

- *When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using a pointer to pass the address of a variable is known as 'call by reference'.*

- *The function which is called by 'reference' can change the value of the variable used in the call. Call by reference provides a mechanism by which the function can change the values stored by the calling function.*

***Example***

```
main()
{
  int x=20;
  change(&x); // Call by reference or address
  printf("%d\n",x);
}
change(int *p)
{
   *p=*p+10
}
```

**Output :** 30

**Functions Returning Pointers**

- The function can return a single value by its name, whereas using pointers, the function can return multiple values.
- The address returned must be the address of a variable in the calling function
- *Without parentheses, fptr is declared as a function returning a pointer to type*

**type *fptr();**

***Example:***

```
int *big(int *, int *);            // Prototype
main()
{
   int a=10,b=20,*p;
   p=big(&a,&b);             // Function call
   printf("The Larger value is %d",*p);
}
int *big(int *x,int *y)
{
   if( *x > *y)
     return(x);                // Address of a
   else
     return(y);               //Address of b
}
```

**Pointers To Functions**

- *A function has a type and address location in the memory. So, the pointers can be declared to a function which can be used as an argument in another function.*

***type (*fptr)();***

*This tells the compiler that fptr is a pointer to a function which returns type value. The parentheses around *fptr are necessary.*

***Example:***

***double mul(int,int);***
***double (*p1)();***

- If p1 is declared as a pointer to the function mul, then the address of mul is assigned to p1, which is equivalent to p1 = mul
- We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.
- p1 is a pointer to a function and mul is a function. p1 is made to point to the function mul p1 is used for instead of mul as

(*p1)(x,y) is equivalent to mul(x,y)

**Program:**

```
#include <stdio.h>
#include <conio.h>
  double mul(int,int);
  double (*p1)();
void main()
  {
     int a,b;
     double c;
       p1=mul;
     printf("Enter a & b Values:\n\n");
     scanf("%d%d",&a,&b);
     c=(*p1)(a,b);
     printf("C value= %lf",c);
     getch();
  }
  double mul(int a, int b)
  {
  return(a*b);
  }
```

**OUTPUT**

Enter a & b Values: 6

7
C value = 42.0000

**Pointers: Compatibility and Casting**

- All pointer variables store memory address.
- A pointer always has a type associated with it. A pointer of one type can not be assigned to a pointer of another type, although both of them have memory addresses as their values. This is known as ***incompatibility*** of pointers.
- But, if the data types of values pointer by pointers differ, then those pointers are known as incompatible pointers.
- The assignment operator can not be used with pointers of different types.
- The **cast** operator can be used for explicit **type casting,** between incompatible pointer types.

```
int x;
char *p;
p=(char *) &x;
```

***Generic Pointer***

The void pointer is a ***generic pointer*** that can represent any pointer type. All pointer types can be assigned to a void pointer and void pointer can be assigned to any pointer without casting.

```
void *vp;
```

**Troubles with Pointers**

- Pointers give us more flexibility and power. It could become a nightmare when they are not used correctly.
- **Major Problem:**
  1. The wrong use of pointers is that the complier may not detect the error in most cases. So, the program is produced unexpected results.
  2. The output may not give us any clue regarding the use of bad pointers
- Assigning values to uninitialized pointers
```
int *p,m=100;
*p=m;          /*Error*/
```
- Assigning value to a pointer variable
```
int *p,m=100;
p=m;          /*Error*/
```
- Not dereferencing a pointer when required
```
int *p,m=100;
p=&m;
printf("%d",p);      /*Error*/
```
- Assigning the address of an uninitialized variables
```
int *p,m;
p=&m;          /*Error*/
```
- Comparing pointers that point to different objects

```
char name1[20],name2[30];
char *p1=name1;
char *p2=name2;
if(p1>p2) ……   /*Error*/
```

## POSSIBLE QUESTIONS

## PART B

**(Each Question Carries 2 marks)**

1. What is pointer?
2. What are advantages of pointers?
3. Define structure with syntax.
4. Write a C program using pointers to multiply two integers.
5. What are self-referential structures?
6. Differentiate between structure and union.
7. Define text & binary files.
8. What do you mean by data file?
9. What is purpose of library function feof()?
10. What do you mean by enumerated data types?

## PART C

**(Each Question Carries 8 marks)**

1. Differentiate between pass by Value and pass by reference with the help of example.
2. What is a pointer to an array and an array of pointers?
3. Write a program to swap two variables by using call by reference.
4. What is the difference between static and dynamic memory allocation?
5. What do you mean by structure? How does a structure differ from an array?
6. Write a program in C language to create a data file.
7. Explain with examples the various file handling functions.
8. Write a program to copy contents of input,txt file to output.txt file.

9. What different function to handle errors in files?
10. What are command line arguments?

*******

**Karpagam Academy (**
**(Deemed University Established U**
**PROGRAMMING FUNDAMENT**
UNIT

| S.No | Question | Choice1 | Choice2 |
|---|---|---|---|
| 1 | The ________ function takes no operator. | Operator +() | Operator –() |
| 2 | In overloading of binary operators, the ________ operand is used to invoke the operator function. | Right-hand | Arithmetic |
| 3 | ________ functions may be used in place of member functions for overloading a binary operator | Inline | Member |
| 4 | The operator that cannot be overloaded is ________ | Single of | + |
| 5 | The friend functions cannot be used to overload the ________ operator. | :: | ?: |
| 6 | ________ is called compile time polymorphism. | Operator overloading | Function overloading |
| 7 | ________ feature can be used to add two user-defined operator data types. | Function | Overloading |
| 8 | ________ operator cannot be overloaded. | = | + |
| 9 | Operator overloading is done with the help of a special function called ________ function. | Conversion | Operator |
| 10 | ________ functions must either be member functions or friend functions. | Operator | User-defined |
| 11 | The overloading operator must have atleast ________ operand that is of user-defined data type. | Two | Three |
| 12 | ________ operator function should be a class member. | Arithmetic | Relational |
| 13 | The casting operator must not have any ________ | Arguments | Member |
| 14 | The casting operator function must not specify a ________ type. | User-defined type | Return |
| 15 | The operator that cannot be overloaded is ________. | Casting | Binary |
| 16 | The friend function cannot be used to overload ________ operator. | + | - |
| 17 | ________ operator cannot be overloaded by friend function. | [] | * |
| 18 | The operator that cannot be overloaded by friend function is ________ | . | :: |
| 19 | Operator overloading is called ________ | Function Overloading | Compile time polymorphism |
| 20 | Overloading feature can add two ________ data types. | In-built | Enumerated |

| | | | |
|---|---|---|---|
| 21 | The mechanism of deriving a new class from an old one is called ________ | Operator overloading | Inheritance |
| 22 | ________ provides the concept of reusability. | Overloading | Message passing |
| 23 | C++ can be reused using ________ | Inheritance | Encapsulation |
| 24 | Inheritance provides the concept of ________. | Derived class | Subclass |
| 25 | The ________ class inherits some or all of the properties of base class. | Abstract class | Father class |
| 26 | A derived class with only one base class is called ________ inheritance. | Single | Multi-level |
| 27 | The derived class inherits some or all of the properties of ________ class. | Member | Base |
| 28 | A derived class can have only one ________ class. | Derived | Base |
| 29 | ________ class inherits some or all of the properties of base class. | Base | Virtual base |
| 30 | A class that inherits properties from more than one class is known as ________ inheritance. | Multiple | Multilevel |
| 31 | The class that can be derived from another derived class is known as ________ inheritance. | Hierarchical | Single |
| 32 | When the properties of one class are inherited by more than one class, it is called ________ inheritance. | Single | Hybrid |
| 33 | A ________ can inherit properties from more than one class. | Class | Member class |
| 34 | A class can be derived from another ________ class. | Member | Common base |
| 35 | ________ of one class can be inherited by more than class. | Functions | Properties |
| 36 | A private member of a class cannot be inherited either in public mode or in ________ mode. | Private | Protected |
| 37 | A protected member inherited in public mode becomes ________ | Highly protected | Private |
| 38 | A protected member inherited in private mode becomes ________ | Visibility | Private |
| 39 | A ________ member of a class cannot be inherited in public mode. | Public | Protected |
| 40 | A member inherited in public mode becomes ________ in the derived class. | Private | Class |
| 41 | A protected member inherited in ________ mode becomes private in the derived class. | Protected | Visibility |

| | | | |
|---|---|---|---|
| 42 | A public member inherited in ________ mode becomes public. | Private | Public |
| 43 | A public member inherited in private mode becomes ________. | Private | Public |
| 44 | The ________ functions of a friend class can directly access the private and protected data. | Inline | Friend |
| 45 | A ________ member inherited in public mode becomes public in the derived class. | Protected | Private |
| 46 | A public member inherited in ________ mode become private in the derived class. | Visibility | Private |
| 47 | The private and protected class can directly access the ________ functions of a friend class. | Virtual | Inline |
| 48 | The member functions of a ________ class can directly access only the protected and public data. | Indirect Base | Ancestor |
| 49 | The member functions of a ________ class can access the private data. | Base | Derived |
| 50 | ________ inheritance may lead to duplication of inherited members from a 'grand parent' base class. | Multiple | Multipath |
| 51 | Duplication of inherited members of ________ inheritance can be avoided by making the common base class, a virtual base class. | Single | Multi-level |
| 52 | In ________ inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. | Hybrid | Multipath |
| 53 | In multi-level inheritance, the ________ are executed in the order of inheritance. | Derivations | Constructors |
| 54 | A class that contains objects of other classes is known as ________. | Indirect base class | Nesting |
| 55 | The ________ section of constructor function is used to assign initial values to its data members. | Initialization | Declaration |
| 56 | The grand parent class is sometimes referred to as ________ class. | Ancestor | Virtual base |
| 57 | ________ may arise in single inheritance application | Ambiguity | Visibility |
| 58 | A ________ that contains objects of other classes is known as containership | Function | Friend |
| 59 | A member declared as ________ cannot be accessed by the function outside the class. | Private | Protected |
| 60 | The public member of a class can be accessed by its own objects using the ________ operator. | Scope resolution | Relational |

| Choice3 | Choice4 | Choice5 | choice6 | Ans |
|---|---|---|---|---|
| Friend | Conversion | | | **Operator –()** |
| Left-hand | Multiplication | | | **Left-hand** |
| Conversion | Friend | | | **Friend** |
| - | = | | | **Single of** |
| . | = | | | **::** |
| Overloading unary operator | Overloading binary operator | | | **Operator overloading** |
| Arrays | Pointers | | | **Overloading** |
| ?: | – | | | **?:** |
| User-defined | In-built. | | | **Operator** |
| Static Member | Overloading | | | **Operator** |
| One | Four | | | **One** |
| Casting | Overloading | | | **Casting** |
| Return type | Operator | | | **Arguments** |
| Member | In-built | | | **Return** |
| Unary | Scope resolution | | | **Scope resolution** |
| ( ) | :: | | | **( )** |
| . | ?: | | | **[]** |
| -> | Single of | | | **::** |
| Casting operator function | Temporary object | | | **Compile time polymorphism** |
| User-defined | Static | | | **User-defined** |

| | | | | |
|---|---|---|---|---|
| Polymorphism | Access mechanism | | | **Inheritance** |
| Data abstraction | Inheritance | | | **Inheritance** |
| Polymorphism | Overloading | | | **Inheritance** |
| Virtual base class | Reusability | | | **Reusability** |
| Derived class | Child class | | | **Derived class** |
| Multiple | Hierarchical | | | **Single** |
| Father | Ancestor | | | **Base** |
| Child | Member | | | **Base** |
| Subclass | Derived | | | **Derived** |
| Single | Hybrid | | | **Multiple** |
| Multi-level | Hybrid | | | **Multi-level** |
| Multiple | Hierarchical | | | **Hierarchical** |
| Inheritance | Base class | | | **Class** |
| Derived | Indirect base class | | | **Derived** |
| Friend | Subclass | | | **Properties** |
| Visibility | Nesting | | | **Private** |
| Public | Protected | | | **Protected** |
| Protected | Public | | | **Private** |
| Private | Access | | | **Private** |
| Public | Protected | | | **Protected** |
| Private | Public | | | **Private** |

| | | | | |
|---|---|---|---|---|
| Visibility | Protected | | | **Public** |
| Protected | Visibility | | | **Private** |
| Virtual | Static members | | | **Friend** |
| Static | Public | | | **Public** |
| Protected | Public | | | **Private** |
| Member | Static member | | | **Member** |
| Base | Derived | | | **Derived** |
| Ancestor | Indirect base | | | **Base** |
| Hybrid | Single | | | **Multipath** |
| Multipath | Hierarchical | | | **Multipath** |
| Hierarchical | Multiple | | | **Multiple** |
| Destructors | Containership | | | **Constructors** |
| Subclass | Inheritance | | | **Nesting** |
| Argument | Assignment | | | **Assignment** |
| Indirect base | Direct base | | | **Indirect base** |
| Nesting | Derivation | | | **Ambiguity** |
| Class | Subclass | | | **Class** |
| Public | Visibility | | | **Protected** |
| Arithmetic | Dot | | | **Dot** |

**Memory Allocation in C++:**

The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** header file.

**Sr.No. Function & Description**

1 **void *calloc(int num, int size);**

This function allocates an array of **num** elements each of which size in bytes will be **size**.

2 **void free(void *address);**

This function releases a block of memory block specified by address.

3 **void *malloc(int num);**

This function allocates an array of **num** bytes and leave them uninitialized.

4 **void *realloc(void *address, int newsize);**

This function re-allocates memory extending it upto **newsize**.

**Allocating Memory Dynamically**

While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows −

```
char name[100];
```

But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory as shown in the below example −

Live Demo

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

int main() {

   char name[100];
   char *description;

   strcpy(name, "Zara Ali");
```

```
   /* allocate memory dynamically */
   description = malloc( 200 * sizeof(char) );

   if( description == NULL ) {
      fprintf(stderr, "Error - unable to allocate required memory\n");
   } else {
      strcpy( description, "Zara ali a DPS student in class 10th");
   }

   printf("Name = %s\n", name );
   printf("Description: %s\n", description );
}
```

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

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

Same program can be written using **calloc();** only thing is you need to replace malloc with calloc as follows −

```
calloc(200, sizeof(char));
```

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

**Resizing and Releasing Memory**

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**. Let us check the above program once again and make use of realloc() and free() functions −

Live Demo

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

int main() {
```

```
  char name[100];
  char *description;

  strcpy(name, "Zara Ali");

  /* allocate memory dynamically */
  description = malloc( 30 * sizeof(char) );

  if( description == NULL ) {
    fprintf(stderr, "Error - unable to allocate required memory\n");
  } else {
    strcpy( description, "Zara ali a DPS student.");
  }

  /* suppose you want to store bigger description */
  description = realloc( description, 100 * sizeof(char) );

  if( description == NULL ) {
    fprintf(stderr, "Error - unable to allocate required memory\n");
  } else {
    strcat( description, "She is in class 10th");
  }

  printf("Name = %s\n", name );
  printf("Description: %s\n", description );

  /* release memory using free() function */
  free(description);
}
```

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

```
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```

## FILE MANAGEMENT

### Introduction

- Many real problems involve large volumes of data and in such situations, the console-oriented I/O operations pose two major problems
    - **It becomes cumbersome and time consuming to handle large volumes of data through terminals.**

  - **The entire data is lost when either the program is terminated or the computer is turned off.**
- Therefore, another method is needed to store data on the disk and read it whenever necessary without destroying the data. Such a method is employed in the concept of files to store data.
- The file is placed on the disk where a group of related data is stored.

**Basic File Operations**

- C supports a number of functions that have the ability to perform basic file operations:
  - Naming a file
  - Opening a file
  - Reading data from a file
  - Writing data from a file
  - Closing a file
- C performs file operations in two ways:
  - Low Level I/O : Uses UNIX System calls
  - High Level I/O : Uses functions in C's Standard Library

**High level I/O Functions**

| FUNCTION NAME | OPERATIONS |
|---|---|
| fopen() | Creates a new file for use (or) Opens an existing file for use |
| fclose() | Closes a file which is already opened for use |
| getc() | Reads a character from a file |
| putc() | Writes a character to a file |
| fprintf() | Writes a set of data values to a file |
| fscanf() | Reads a set of values from a file |
| getw() | Reads an integer from a file |
| putw() | Writes an integer to a file |
| fseek() | Sets the position to a desired point in the file |
| ftell() | Gives the current position in the file<br>(in terms of bytes from the start) |
| rewind() | Sets the position to the beginning of the file |

**DEFINING AND OPENING A FILE**

- If data is stored in a file in the secondary memory, certain details / information about the file are specified to the operating system

- They are:
  - File Name – a string of characters that make- up a valid file name for the OS.
  - Pointer to Data structure (ie to the file)( eg.: fp1, fp2, etc.,)
  - Mode of file operation (eg. "r", "w", "a", "r+", "w+", "a+")
- The general format for declaring and opening a file

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

  where fp is a pointer to the FILE data type,
  filename denotes the file which will be opened,
  mode represents the purpose of the opening file.
- The second statement opens the file, named as *filename* and assigns a pointer to the FILE type pointer fp. This pointer which contains all the information about the file is subsequently used as a communication link between the system and program
- The modes of the file

| | |
|---|---|
| **r** | **Opens the file for reading only** |
| **w** | **Opens the file for writing only** |
| **a** | **Opens the file for appending (or adding ) data to it** |

- When trying to open a file, one of the following things may happen
  1. When mode is 'writing', a file is created if the file does not exist. The contents of the file are deleted, of the file already exists.
  2. When the purpose is 'appending', the file is opened with current contents safe. A file with the specified name is created if the file does no t exist.
  3. If the purpose is 'reading' and if it exists, then the file is opened with current contents safe otherwise an error occurs.

  **FILE *p1,*p2;**
  **p1=fopen("data","r");**
  **P2=fopen("results","w");**
- The additional modes are included in recent compliers

| | |
|---|---|
| **r+** | **The existing file is opened to the beginning for both reading and writing** |
| **w+** | **Same as w except both for reading and writing** |
| **a+** | **Same as a except both for reading and writing** |

**Closing a file**

- A file must be closed after completion of all operations.
- This ensures that all information associated with the file is flushed out from the buffer and a links to the file are removed.
- In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files.
- The general format to close a file

  **fclose(file_pointer)**

**Example:**

```
FILE *p1,*p2;
p1=fopen(“INPUT”,”w”);
p2=fopen(“OUTPUT”,”r”);
………
….
fclose(p1);
fclose(p2);
………..
```

***getc*** **and** ***putc*** **functions**

- The simplest I/O functions are getc() and putc(). These functiosn handle one character at a time.
- Assume that the file is opened with mode w and file pointer fp1. the below statement writes the character contained in the character variable c to the file associated with FILE pointer fp1.

  **putc(c,fp1);**
- getc is used to read a character from a file.

  **c=getc(fp1);**

**Example:**

```
#include<stdio.h>
void main()
{
  FILE *f1;
  char c;
  printf(“Data Input \n\n”);
  f1=fopen(“INPUT”,”w”);
  while((c=getchar()) != EOF)
       putc(c,f1);
   fclose(f1);
   printf(“Data Output \n\n”);
       f1=fopen(“INPUT”,”r”);
        while((c=getc(f1)) != EOF)
               printf(“%c”,c);
         fclose(f1);
}
```

```
OUTPUT:
 Data Input
   Hi world^Z
 Data Output
   Hi world
```

***getw()*** **and** ***putw()*** **functions**

- The getw() and putw() are integer-oriented functions.
- They are used to read and write integer values.
- The general format is:

  **putw(integer,fp);**
  **getw(fp);**
- The illustrations of getw() and putw()

**putw(number,f1);**
**(number=getw(f1)) != EOF)**

**Example Program**

```
#include  <stdio.h>
#include <conio.h>
void main()
  {
     FILE  *f1, *f2, *f3;
     int   number, i;
     printf("Contents of DATA file\n\n");
     f1 = fopen("DATA", "w"); /*Create DATA file*/
     for(i = 1; i <= 30; i++)
     {
       scanf("%d", &number);
       if (number == -1) break;
       putw(number,f1);
     }
     fclose(f1);
     f1 = fopen("DATA", "r");
     f2 = fopen("ODD", "w");
     f3 = fopen("EVEN", "w");
/* Read from DATA file */
     while((number = getw(f1)) != EOF)
     {
        if(number %2 == 0)
 putw(number, f3); /*Write to EVEN file */
        else
  putw(number, f2); /*Write to ODD file  */
     }
fclose(f1);
     fclose(f2);
     fclose(f3);
  f2 = fopen("ODD","r");
     f3 = fopen("EVEN", "r");
     printf("\n\nContents of ODD file\n\n");
     while((number = getw(f2)) != EOF)
       printf("%4d", number);
     printf("\n\nContents of EVEN file\n\n");
     while((number = getw(f3)) != EOF)
       printf("%4d", number);
     fclose(f2);
```

```
        fclose(f3);
        getch();
      }
```

**OUTPUT:**

**Contents of DATA File:**
**23 42 77 7 28 88 98 ^Z**
**Contents of ODD File :**
**23 77 7**
**Contents of Even File:**
**42 28 88 98**

***fprintf* and *fscanf* functions**

- The functions fprintf() and fscanf() perform the I/O operations that are identical to the printf and scanf, that they work on files
- The general format
  **fprintf(fp,"Control String", list);**
  **fscanf(fp,"Control String",list);**
- The illustrations of fprintf() and fscanf():
  **fprintf(f1,"%d %s %d", rno, name, mark1);**
  **fscanf(f1,"%d %s %d", &rno, name, &mark1);**

**Example:**

```
#include  <stdio.h>
#include <conio.h>
void main()
  {
     FILE  *fp;
     int    number, quantity, i;
     float  price, value;
     char   item[10], filename[10];
     printf("Input file name\n");
     scanf("%s", filename);
     fp = fopen(filename, "w");
     printf("Input inventory data\n\n");
     printf("Item name  Number   Price   Quantity\n");
     for(i = 1; i <= 3; i++)
     {
         fscanf(stdin,"%s %d %f %d",item, &number, &price, &quantity);
          fprintf(fp,"%s %d %f %d",item, number, price, quantity);
     }
 fclose(fp);
     fprintf(stdout,"\n\n");
     fp = fopen(filename,"r");
printf("Item name  Number   Price   Quantity    Value\n");
```

```
    for(i = 1; i <= 3; i++)
    {
        fscanf(fp,"%s %d %f %d",item,&number,&price,&quantity);
        value = price * quantity;
        fprintf(stdout,"%-8s %7d %8.2f %8d %11.2f\n",item, number, price, quantity, value);
    }
    fclose(fp);
    getch();
  }
```

**OUTPUT:**

```
Input File Name :    Inventory
Input inventory data
Item Name Number Price Quantity
A  111  17.50 115
B 125 36.00 75
 Item Name      Number    Price   Quantity      Value
      A              111        17.50     115         2012.50
      B              125        36.00      75        2700.00
```

**ERROR HANDLING DURING I/O OPERATIONS**

- It is possible that an error may occur during I/O operations on a file. Typical error situations include:
    - Trying to read beyond the EOF
    - Device overflow
    - Trying to use a file that has not been opened
    - Trying to perform an operation on a file, when the file is opened for another type of operation
    - Opening a file with an invalid filename
    - Attempting to write to write protected file
    - The feof(f) function can be used to test for an end of file condition.
  - ✓ **feof(f)**
    - determines if end-of-file is reached;
    - Returns an integer value.
    - If end of file is reached, it returns a non-zero; else returns 0
- The ferror() function reports the status of the file indicated.
- It also takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point during processing.

**Error Handling : Sample Program**

```
#include <stdio.h>
#include <conio.h>
void main()
   {
     char  *filename;
```

```
    FILE  *fp1;
    int   i, number;
   fp1 = fopen("TEST", "w");
    for(i = 10; i <= 50; i += 10)
      putw(i, fp1);
    fclose(fp1);
    printf("\nInput filename\n");
    open_file:  scanf("%s", filename);
 if((fp1== fopen(filename,"r")) == NULL)
    { printf("Cannot open the file.\n");
      printf("Type another file name\n\n");
      goto open_file;
    }
else
    for(i = 1; i <= 20; i++)
    {  number = getw(fp1);
      if(feof(fp1))
      { printf("\nOut of data.\n");
        break;
      }
      else {   printf("%d\n", number);
    }

    fclose(fp1);
      getch();
  }
```

**Output**
**Input file name**
**Tes**
**Cannot open the file**
**Type another file name**
**TEST**
**10**
**20**
**30**
**40**
**50**

**RANDOM ACCESS TO FILES**

- A part of a file can be accessed directly, (unlike sequential access) using **fseek(), ftell() and rewind()** functions.

**ftell()**

- ftell() takes a file pointer and returns a number of type **long**, that corresponds to the current position. This function is useful in saving the current position of a file which can be used later in the program.

**n=ftell(fp);**

n gives the relative offset (in bytes) of the current position

**rewind()**

- Rewind takes a file pointer and resets the position to the start of the file.

**rewind(fp);**
**n = ftell(fp);**

- The above statements would assign **0** to **n** because the file position has been set to the start of the file by **rewind()**.
- It helps us in reading a file more than once, without having to close and open the file.
- When a file is opened for reading or writing, **rewind()** is done implicitly.

**fseek()**

- **fseek** function is used to move the file position to a desired location within the file.
  ***fseek(fp,offest,position);***
  ***fp*** is a pointer to the file concerned
  ***offset*** is number or variable of type long
  ***position*** is an integer
- The *offset* specifies the number of positions to be moved from the location specified by position.
- The position can take one of the following three values

| **Value** | **Meaning** |
|---|---|
| **0** | **Beginning of file** |
| **1** | **Current Position** |
| **2** | **End of file** |

- When the operations is successful, fseek returns a zero.
- If file pointer is moved beyond the file boundaries, an error occurs and fseek returns -1. It is to check whether an error has occurred or not, before proceeding further.

| STATEMENT | MEANING |
|---|---|
| fseek(fp,0L,0) | Go to the beginning |
| fseek(fp,0L,1) | Stay at the current position |
| fseek(fp,0L,2) | Go to the end of the file, past the last character of the file |
| fseek(fp,m,0) | Move to $(m+1)^{th}$ byte in the file |
| fseek(fp,m,1) | Go forward by m bytes |
| fseek(fp,-m,1) | Go backward by m bytes from the current position |
| fseek(fp,-m,2) | Go backward by m bytes from the end |

**POSSIBLE QUESTIONS**

**PART B**

**(Each Question Carries 2 marks)**

1. What is memory allocation?
2. What is the purpose of new operators?
3. What is the purpose of delete operator?
4. Define: Preprocessor directives
5. How do you read a file?
6. How do you write a file?
7. Define: Macros

8. What is the purpose of #define statement?
9. Differentiate static and dynamic memory allocation?
10. What is the purpose of malloc function?

**PART C**

**(Each Question Carries 8 marks)**

1. Differentiate between static and dynamic memory allocation.
2. Explain about the use of Fstream header file with example.
3. Explain about the use of malloc and calloc function with example.
4. Explain the hierarchy of File stream classes.
5. Explain about the use of New and Delete operators in memory.
6. Write a C++ program to Read and Write Text files with fstream header.
7. Discuss about storage of variables in static memory allocation.
8. Explain about Random Access in Files.
9. Discuss about storage of variables in dynamic memory allocation.
10. Explain any five preprocessor directives with example.

**Karpagam Academy o**

**(Deemed University Established Un**

**PROGRAMMING FUNDAMENTA**

UNIT -

| S.No | Question | Choice1 | Choice2 |
|---|---|---|---|
| 1 | The stream is an _________ between I/O devices and the user. | Translater | Destination |
| 2 | If the data is received from the input devices in sequence then it is called________. | Source stream | Object stream |
| 3 | When the data is passed to the output devices it is called_____ | Source stream | Object stream |
| 4 | The C++ have a number of stream classes that are used to work with _________ operations. | Console I/O | Console and file |
| 5 | The data accepted with default setting by I/O function of the language is known as | Formatted | Unformatted |
| 6 | _________ is used as the input stream to read data. | Cout | Printf |
| 7 | cin and cout are ________ for input and output of data. | user defined stream | system defined stream |
| 8 | The data obtained or represented with some manipulators are called ______. | formatted data | unformatted data |
| 9 | The output formats can be controlled with manipulators having the header file as | iostream.h | conio.h |
| 10 | The _____ and ______ are derived classes from ios based class. | istream and ostream | source and destination stream |
| 11 | The manipulator << endl is equivalent to____ | ‘\t’ | ’\r’ |
| 12 | A virtual function must be defined in _____ | Friend | enemy |
| 13 | The virtual function must be defined in _______ | Public | Private |
| 14 | The function in base class is declared as virtual using the keyword _______ | Virtual | Class |
| 15 | Precision() is an __________ format function | Manipulator | Istream |
| 16 | Width of the output field is set using the ______ | width() | iomanip.h |
| 17 | ______ is used to achieve run time polymorphism | operator overloading | function overloading |
| 18 | Pointers are used to access ________ | Object | Virtual function |
| 19 | The member functions can be refered by using the ________ and _______ | dot operator and object | address operator and virtual functions |
| 20 | The paranthesis are necessary because the dot operator has higher precedence than the __________ | dot operator | this |
| 21 | ______ is used to represent an object that involves a member function. | friend | this |

| | | | |
|---|---|---|---|
| 22 | The this pointer acts as an _____ argument to all the member function | implicit | explicit |
| 23 | When two or more objects are compare inside a member function the result in return is an _________ | virtual function | derived class |
| 24 | Pointers are used as the objects of _______ | user defined | derived class |
| 25 | Virtual functions are from the concept of ______ | objects | polymorphism |
| 26 | The virtual functions are accessed with the help of a pointer declared as ____ to the base class | Class | object |
| 27 | _____ is achieved when a virtual function is accessed through a pointer to the base class. | run time polymorphism | inheritance |
| 28 | we cannot have virtual constructors but _____ are allowed | translators | virtual function |
| 29 | The virtual functions cannot be _________ | class | object |
| 30 | Virtual functions must be _____ of some class. | class | pointer |
| 31 | A ____ is a function declared in a base class that has no definition relative to the base class | Virtual function | pure virtual function |
| 32 | A ____ equated to zero is called a pure virtual function. | virtual function | pure virtual function |
| 33 | Stream and stream classes are used to implement its I/O operations with the ______ | the console and disk files | cin and cout |
| 34 | The interface supplied by an I/O system which is independent of actual device is called _____ | stream | class |
| 35 | A _____ is a sequence of bytes. | Stream | class |
| 36 | The _____ streams automatically open when the program begins its execution | user defined | predefined |
| 37 | The class that is defined to various streams to deal with both the console and disk files is called ________ | stream class | derived class |
| 38 | ____ provide an interface to physical devices through buffers. | stream buffer | iostream |
| 39 | The _____ are called as overloaded operators | >> and << | + and – |
| 40 | The >> operator is overloaded in the _______ | istream | ostream |
| 41 | The ____ functions are used to handle the single character I/O operation. | get() and put() | clrscr() and getch() |
| 42 | ____ functions are used to display text more efficiently by using the line oriented i/o functions. | getline() and write() | cin and cout |
| 43 | The getline() reads character input to the ______ line | datatype | function |
| 44 | _____ is used to clear the flags specified. | width() | precision() |
| 45 | _____ is used to specify the required field size for displaying an output value | width() | self |

| | | | |
|---|---|---|---|
| 46 | By default the floating numbers are printed with ______ after the decimal point. | 5 digits | 6 |
| 47 | ____ returns the setting in effect until it is reset | width | precision() |
| 48 | In fill() the unused positions of the field are filled with _____ by default. | null character | white spaces |
| 49 | set(f) is the member function of _____ | istream | ioclass |
| 50 | setf() can be with the falg _____ as a single argument to achieve a formatted output | ios :: showpoint | ios :: left |
| 51 | In flags there ______ it fields | 3 | 4 |
| 52 | The flag formatted for the octal base is ________ | ios::doc | ios::hex |
| 53 | The flag is formatted with ______ arguments. | 1 | 2 |
| 54 | The bit field is formatted with ______ arguments. | 1 | 2 |
| 55 | _____ flush all streams after insertion | ios::stdio | ios::shoebase |
| 56 | _____ is used as base indicator on output. | ios::stdio | ios::shoebase |
| 57 | _____ manipulator is equalent to fill() | setw() | setprecision() |
| 58 | ______ returns the previous format state. | ios member function | manipulator |
| 59 | The bitfield used for fixed point notation is ______ | ios::floatfield | ios::adjustfield |
| 60 | The flag used to format the decimal base is _____ | ios::oct | ios::fixes |

**t Higher Education**
**der Section 3 of UGC Act 1956)**
**ALS USING C/C++(18CAU101)**
-4

| Choice3 | Choice4 | choice5 | choice6 | Ans |
|---|---|---|---|---|
| Intermediator | source | | | **Intermediator** |
| Destination stream | Input stream | | | **Source stream** |
| Destination stream | Input stream | | | **Destination stream** |
| formatted console | unformatted | | | **Console and file** |
| Argumented | paramerized | | | **Unformatted** |
| Cin | Scanf | | | **Cin** |
| Pre defined stream | macro | | | **system defined stream** |
| extracted data | derived data | | | **formatted data** |
| stdlib.h | iomanip.h | | | **iomanip.h** |
| iostream and source stream | stdio | | | **istream and ostream** |
| '\n' | '\b' | | | **'\n'** |
| member | class | | | **Friend** |
| Protected | global | | | **Public** |
| Pointer | Structure | | | **Virtual** |
| ios | user defined | | | **ios** |
| Manipulator | hight | | | **width()** |
| inline function | virtual function | | | **virtual function** |
| Class members | defintion | | | **Class members** |
| class and object | scope resolution | | | **dot operator and object** |
| class | indirection operator | | | **indirection operator** |
| class | virtual | | | **this** |

| | | | | |
|---|---|---|---|---|
| formal | actual. | | | **implicit** |
| invoking objects | inline function | | | **invoking objects** |
| virtual function | object. | | | **derived class** |
| inheritance | encaptulation | | | **polymorphism** |
| pointer | stream | | | **pointer** |
| class | static class | | | **run time polymorphism** |
| virtual destructor | static members | | | **virtual destructor** |
| constructors | static members | | | **static members** |
| stream | member | | | **member** |
| stream | class | | | **pure virtual function** |
| stream | class. | | | **virtual function** |
| manipulators | getch() | | | **the console and disk files** |
| object | structure | | | **stream** |
| object | union | | | **Stream** |
| input | output | | | **predefined** |
| object | retrived class | | | **stream class** |
| ostream | istream | | | **stream buffer** |
| * and && | – and . | | | **>> and <<** |
| iostream | fstream | | | **istream** |
| cin and cout | getc() | | | **get() and put()** |
| get() and put() | getchar() | | | **getline() and write()** |
| variable | constants | | | **variable** |
| setf() | unsetf() | | | **unsetf()** |
| fill() | free() | | | **width()** |

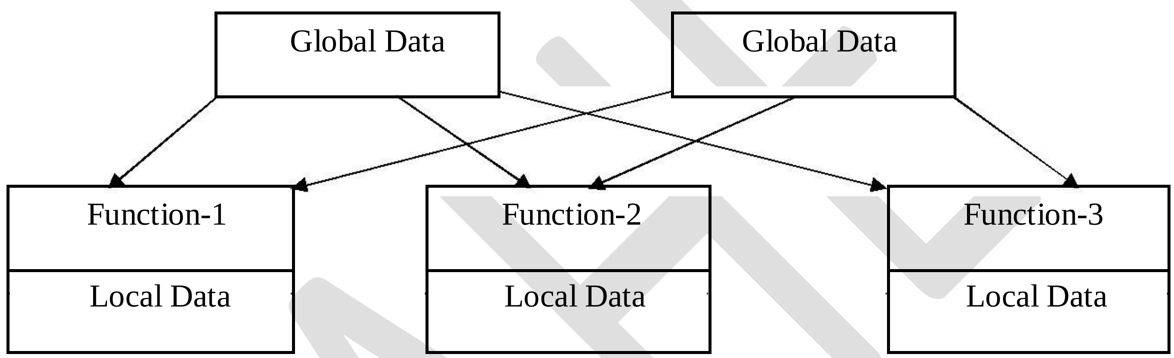
| | | | | |
|---|---|---|---|---|
| 7 | 8 | | | **6** |
| setf() | fill() | | | **precision()** |
| zeros | both a and b | | | **white spaces** |
| ios class | both b and c | | | **ios class** |
| ios::floatfield | ios::basefield | | | **ios :: showpoint** |
| 5 | 8 | | | **3** |
| ios::fixwd | ios::oct | | | **ios::oct** |
| 3 | 4 | | | **1** |
| 3 | 4 | | | **2** |
| ios::showpoint | ios:: unitbuf | | | **ios:: unitbuf** |
| ios::showpoint | d ios:: unitbuf | | | **ios::shoebase** |
| setfill() | endif | | | **setfill()** |
| class | a and b | | | **ios member function** |
| ios::basefield | ios::field | | | **ios::floatfield** |
| ios::left | ios::dec. | | | **ios::dec.** |

**Using Classes in C++:**

**Principles of Object-Oriented Programming**

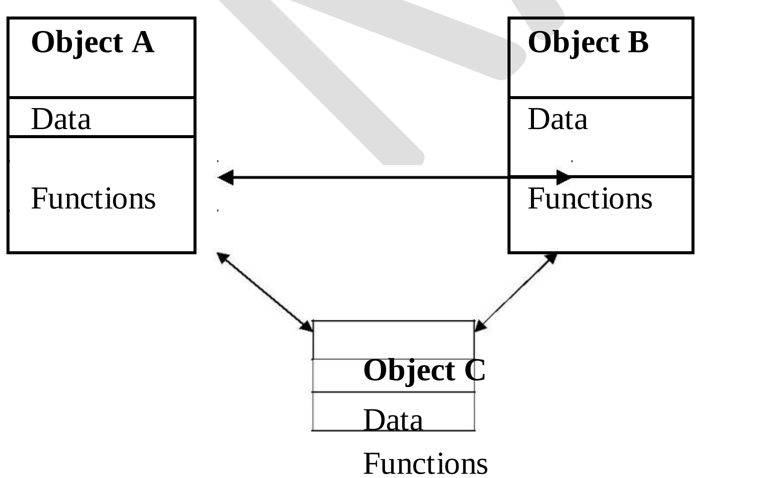
**Procedure/ structure oriented Programming**

- Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure-oriented programming (POP).
- In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks.
- The primary focus is on functions.




**Object Oriented Programming**

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.



## Basic Concepts of Object-Oriented Programming

### Objects

Objects are the basic runtime entities in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

### Class

Object contains data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

### Data Encapsulation

The wrapping up of data and functions into a single unit is known as encapsulation.

The data is not accessible to the outside world, only those function which are wrapped in the can access it.

These functions provide the interface between the object's data and the program.

This insulation of the data from direct access by the program is called **data hiding** or **information hiding**.

### Data Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations.

Since classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT)**.

### Inheritance

**Inheritance** is the process by which objects of one class acquire the properties of objects of another class.

In OOP, the concept of inheritance provides the idea of reusability. This means we can add additional features to an existing class without modifying it.

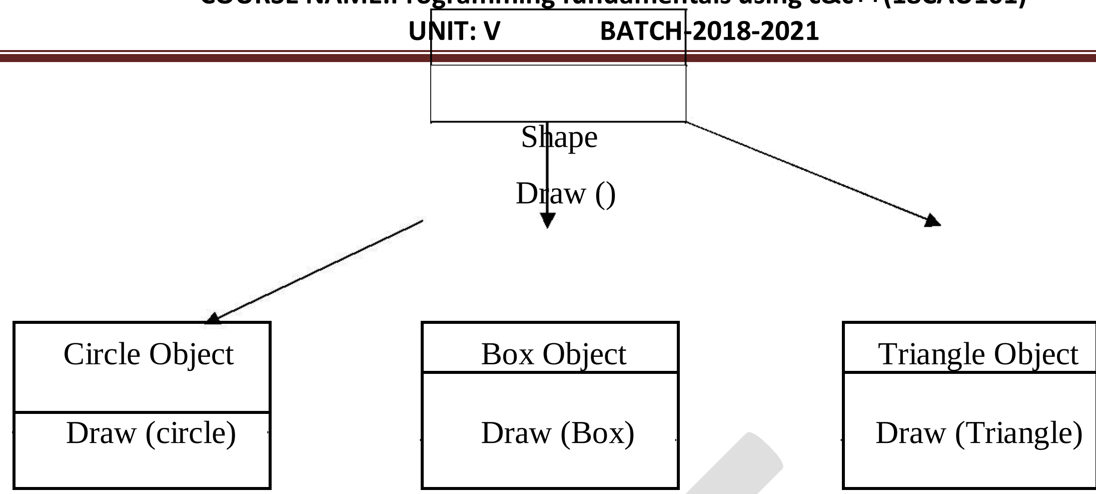
### Polymorphism

**Polymorphism,** a Greek term means to ability to take more than one form.

An operation may exhibits different behaviors in different instances. The behavior depends upon the type of data used in the operation.

For example consider the operation of addition for two numbers; the operation will generate a sum. If the operands are string then the operation would produce a third string by concatenation.

The process of making an operator to exhibit different behavior in different instances is known operator overloading.




## Output and Input Statement in C++

**An Output statement** is used to print the output on computer screen. cout is an output statement. cout<<**"**Srinix College of Engineering**"**; prints **Srinix College of Engineering** on computer screen. cout<<**"**x**"**; print **x** on computer screen.

cout<<x; prints **value of x** on computer screen. cout<<"**\n**"; takes the cursor to a newline.

cout<< **endl**; takes the cursor to a newline. We can use **endl** (a manipulator) instead of **\n**. << (two "less than" signs) is called insertion operator.

**An Input statement** is used to take input from the keyboard. cin is an input statement. cin>>x; takes the value of x from keyboard.

cin>>x>>y; takes value of x and y from the keyboard.

**Program 1.1 WAP to accept an integer from the keyboard and print the number when it is multiplied by 2.**

**Solution:**

```
#include
<iostream.h>      void
main ()
{
 int x;
 cout << "Please enter an integer
 value: "; cin >>x;
```

```
 cout <<endl<< "Value you entered is "
 <<x; cout << " and its double is "
 <<x*2 << ".\n";
}
```

**Output:**

Please enter an integer value:

**Class Constructors**

**Constructor**

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

**C++ Default Constructor**

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
 {
  public:
      Employee()
      {
         cout<<"Default Constructor Invoked"<<endl;
      }
};
int main(void)
{
   Employee e1; //creating an object of Employee
   Employee e2;
   return 0;
}
```

Output:

Default Constructor Invoked
Default Constructor Invoked

**C++ Parameterized Constructor**

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
  public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
      float salary;
      Employee(int i, string n, float s)
       {
          id = i;
          name = n;
          salary = s;
       }
      void display()
       {
          cout<<id<<"  "<<name<<"  "<<salary<<endl;
       }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Nakul", 59000);
   e1.display();
   e2.display();
   return 0;
}
```

Output:

101 Sonoo 890000
102 Nakul 59000

**C++ Destructor**

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

**Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.**

### C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
            cout<<"Constructor Invoked"<<endl;
        }
        ~Employee()
        {
            cout<<"Destructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}
```

Output:

Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked

### Constructor Overloading

Constructor can be overloaded in a similar way as function overloading

.

Overloaded constructors have the same name (name of the class) but different number of arguments.

Depending upon the number and type of arguments passed, specific constructor is called.

Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

```
#include <iostream>
using namespace std;

class Area
{
   private:
     int length;
     int breadth;

   public:
     // Constructor with no arguments
     Area(): length(5), breadth(2) { }

     // Constructor with two arguments
     Area(int l, int b): length(l), breadth(b){ }

     void GetLength()
     {
        cout << "Enter length and breadth respectively: ";
        cin >> length >> breadth;
     }

     int AreaCalculation() {  return length * breadth;  }

     void DisplayArea(int temp)
     {
        cout << "Area: " << temp << endl;
     }
};

int main()
{
```

```
    Area A1, A2(2, 1);
    int temp;

    cout << "Default Area when no argument is passed." << endl;
    temp = A1.AreaCalculation();
    A1.DisplayArea(temp);

    cout << "Area when (2,1) is passed as argument." << endl;
    temp = A2.AreaCalculation();
    A2.DisplayArea(temp);

    return 0;
}
```

For object *A1*, no argument is passed while creating the object.

Thus, the constructor with no argument is invoked which initialises *length* to 5 and *breadth* to 2. Hence, area of the object *A1* will be 10.

For object *A2*, 2 and 1 are passed as arguments while creating the object.

Thus, the constructor with two arguments is invoked which initialises *length* to *l* (2 in this case) and *breadth* to *b* (1 in this case). Hence, area of the object *A2* will be 2.

**Output**

```
Default Area when no argument is passed.
Area: 10
Area when (2,1) is passed as argument.
Area: 2
```

**Default Copy Constructor**

An object can be initialized with another object of same type. This is same as copying the contents of a class to another class.

In the above program, if you want to initialise an object *A3* so that it contains same values as *A2*, this can be performed as:

```
....
int main()
{
  Area A1, A2(2, 1);

  // Copies the content of A2 to A3
  Area A3(A2);
    OR,
  Area A3 = A2;
}
```

You might think, you need to create a new constructor to perform this task. But, no additional constructor is needed. This is because the copy constructor is already built into all classes by default.

Function refers to a segment that groups code to perform a specific task.

In C++ programming, two functions can have same name if number and/or type of arguments passed are different.

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

```
include <iostream>
```

```
using namespace std;

void display(int);

void display(float);

void display(int, float);

int main() {

    int a = 5;

    float b = 5.5;

    display(a);

    display(b);

    display(a, b);


    return 0;

}

void display(int var) {

    cout << "Integer number: " << var << endl;

}

void display(float var) {

    cout << "Float number: " << var << endl;

}

void display(int var1, float var2) {

    cout << "Integer number: " << var1;

    cout << " and float number:" << var2;
```

}

**Output**

Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5

Here, the display() function is called three times with different type or number of arguments.

The return type of all these functions are same but it's not necessary.

**Templates Classes**

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
```

Edit & Run

```
cout << k << endl;
cout << n << endl;
return 0;
}
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;

```

Edit & Run

```
template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

### Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
   T values [2];
  public:
   mypair (T first, T second)
   {
     values[0]=first; values[1]=second;
   }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

100 Edit & Run



Notice the syntax of the definition of member function getmax:

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

## Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
```

8 Edit & Run
J

```
mycontainer<int> myint (7);
mycontainer<char> mychar ('j');
cout << myint.increase() << endl;
cout << mychar.uppercase() << endl;
return 0;
}
```

This is the syntax used in the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an empty template<> parameter list. This is to explicitly declare it as a template specialization.

But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

## Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
// sequence template
#include <iostream>
```

```
100
3.1416
```

Edit & Run

```
using namespace std;

template <class T, int N>
class mysequence {
    T memblock [N];
  public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
  memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
  return memblock[x];
}

int main () {
  mysequence <int,5> myints;
  mysequence <double,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3,3.1416);
  cout << myints.getmember(0) << '\n';
  cout << myfloats.getmember(3) << '\n';
  return 0;
}
```

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

```
template <class T=char, int N=10> class mysequence {..};
```

We could create objects using the default template parameters by declaring:

```
mysequence<> myseq;
```

Which would be equivalent to:

```
mysequence<char,10> myseq;
```

### Templates and multiple-file projects

From the point of view of the compiler, templates are not normal functions or classes. They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation with specific template arguments is required. At that moment, when an instantiation is required, the compiler generates a function specifically for those arguments from the template.

When projects grow it is usual to split the code of a program in different source code files. In these cases, the interface and implementation are generally separated. Taking a library of functions as example, the interface generally consists of declarations of the prototypes of all the functions that can be called. These are generally declared in a "header file" with a .h extension, and the implementation (the definition of these functions) is in an independent file with c++ code.

Because templates are compiled when required, this forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration. That means that we cannot separate the interface in a separate header file, and that we must include both interface and implementation in any file that uses the templates.

Since no code is generated until a template is instantiated when required, compilers are prepared to allow the inclusion more than once of the same template file with both declarations and definitions in a project without generating linkage errors.

**Overview of Function Overloading and Operator Overloading**

**C++ Overloading (Function and Operator)**

If we create two or more members having same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- **Function overloading**
- **Operators overloading**

**C++ Function Overloading**

Having two or more function with same name but different in parameters, is known as function overloading in C++.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for same action.

## C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
   Cal C;
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Output:

30
55

## C++ Operators Overloading

Operator overloading is used to overload or redefine most of the operators available in C++. It is used to perform operation on user define data type.

The advantage of Operators overloading is to perform different operations on the same operand.

## C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

```
#include <iostream>
using namespace std;
class Test
{
   private:
      int num;
   public:
       Test(): num(8){}
       void operator ++()
       {
          num = num+2;
       }
       void Print() {
           cout<<"The Count is: "<<num;
       }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

The Count is: 10

## C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.
C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
  };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

Output:

Salary: 60000
Bonus: 5000

In the above example, Employee is the base class and Programmer is the derived class.

**C++ Single Level Inheritance Example: Inheriting Methods**

Let's see another example of inheritance in C++ which inherits methods only.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:

Eating...
Barking...

**C++ Multi Level Inheritance Example**

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
```

```
    {
       public:
     void bark(){
    cout<<"Barking..."<<endl;
     }
    };
    class BabyDog: public Dog
    {
       public:
     void weep() {
    cout<<"Weeping...";
     }
    };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

Eating...
Barking?
Weeping?

**C++ Polymorphism**

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

**There are two types of polymorphism in C++:**

- **Compile time polymorphism:** It is achieved by function overloading and operator overloading which is also known as static binding or early binding.
- **Runtime polymorphism:** It is achieved by method overriding which is also known as dynamic binding or late binding.

---

**C++ Runtime Polymorphism Example**

Let's see a simple example of runtime polymorphism in C++.

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {
       cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

Eating bread...

**C++ virtual function**

C++ virtual function is a member function in base class that you redefine in a derived class. It is declare using the virtual keyword.

It is used to tell the compiler to perform **dynamic linkage or late binding** on the function.

---

**Late binding or Dynamic linkage**

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

---

## C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
using namespace std;
class A
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

Output:

Derived Class is invoked

## C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

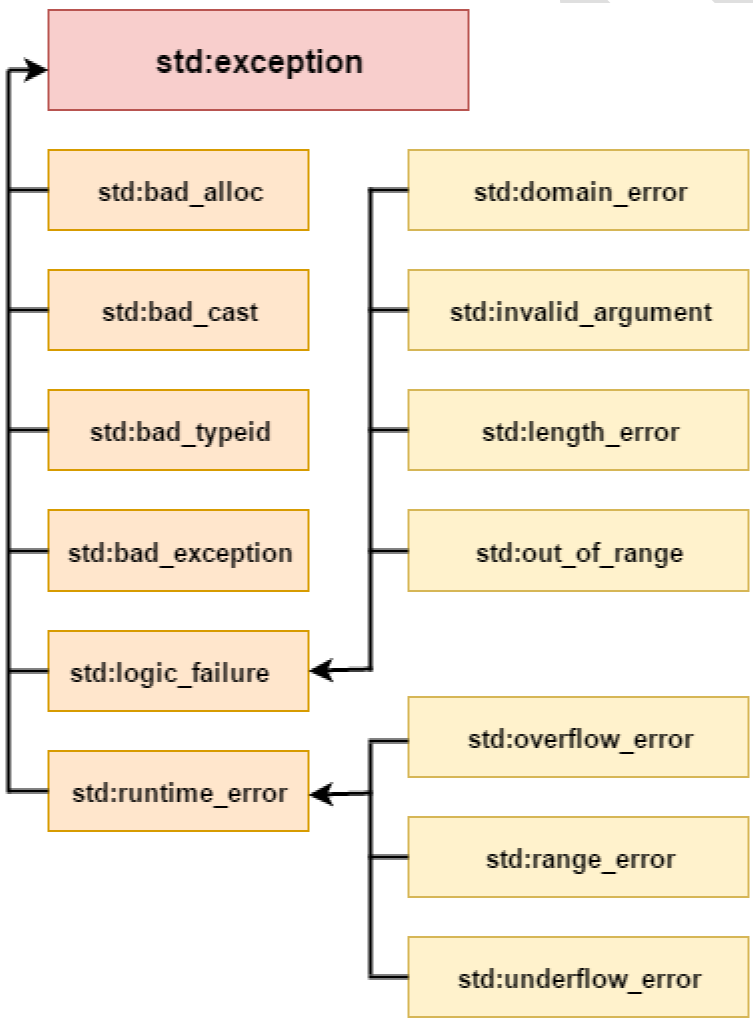
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

### Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

### C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

| Exception | Description |
|---|---|
| std::exception | It is an exception and parent class of all standard C++ exceptions. |
| std::logic_failure | It is an exception that can be detected by reading a code. |
| std::runtime_error | It is an exception that cannot be detected by reading a code. |
| std::bad_exception | It is used to handle the unexpected exceptions in a c++ program. |
| std::bad_cast | This exception is generally be thrown by **dynamic_cast.** |
| std::bad_typeid | This exception is generally be thrown by **typeid.** |

**C++ try/catch**

In C++ programming, exception handling is performed using try/catch statement. The C++ **try block** is used to place the code that may occur exception. The **catch block** is used to handle the exception.

---

**C++ example without try/catch**

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  return (x/y);
}
int main () {
  int i = 50;
  int j = 0;
  float k = 0;
    k = division(i, j);
    cout << k << endl;
  return 0;
}
```

Output:

Floating point exception (core dumped)

## POSSIBLE QUESTIONS

**PART B**

**(Each Question Carries 2 marks)**

1. Define : Class
2. What is function overloading?
3. What is an object give example?
4. Define : Constructor
5. What is the purpose of this keyword?
6. Define : Virtual function
7. What is exception handling?
8. Define : Polymorphism
9. What is an inheritance?
10. Define : Operator overloading

**PART C**

**(Each Question Carries 8 marks)**

1. Discuss about object oriented concepts with real time examples.
2. Describe about Single and Multilevel inheritance with example.
3. Describe class constructor in detail.
4. Explain about overloading functions by number and type of arguments.
5. Discuss about Constructor OverLoading in detail with suitable example.
6. Describe virtual function .Explain with examples.
7. Describe the usage copy constructor with example.
8. Explain about the importance of operator overloading.
9. Discuss about overview of Template classes and explain their usage.
10. Write a C++ program to throw multiple exceptions and define multiple catches.

**Karpagam A**

**(Deemed University Estal**

PROGRAMMING FUN

| S.No | Question | Choice1 |
|---|---|---|
| 1 | A _______ is a collection of related data stored in a particular area on a disk. | Field |
| 2 | File streams act as an ________ between programs and files. | interface |
| 3 | Ifstram, Ofstream, Fstream are derived form __________. | iostream |
| 4 | Classes designed to manage the ________ files are declared in fstream. | random |
| 5 | _________ is to set the file buffer to read and write. | filebuf |
| 6 | ________ inherits get(), getline(), read(), seekg(), and tellg() from istream. | conio |
| 7 | Put(), seekp(), tellp(), and write() functions are inherited by ofstream from _______ | ostream |
| 8 | ______ inherits all functions from istream and ostream through iostream | file stream |
| 9 | The file mode parameter for opening a binary file is ________. | ios::ate |
| 10 | _______ is the file mode parameter for go to end of file on opening. | ios::ate |
| 11 | The file mode paramenter for writing only onto the file is ________. | ios::in |
| 12 | Opening a file in ios::out mode also opens it in the ________ mode by default | ios::trunc |
| 13 | Both _______ and _______ take us to the end of the file | ios::ate, ios::create |
| 14 | The parameter _______ can be used only with the files capable of output. | ios::ate |
| 15 | The parameter ios::app can be used only with the files capable of ______. | input |
| 16 | The eof ( ) stands for _____. | end of file |
| 17 | Command line arguments are used with ________ function | main() |
| 18 | The close() function _________. | closes the file |
| 19 | The write() function writes ___________. | single character |
| 20 | The __________ function shifts the associated files input (get) file pointer | seekg() |
| 21 | The __________ function shifts the associated files output (put) file pointer. | seekg() |
| 22 | The object of fstream class provides ________operation | both read and rite |

| | | |
|---|---|---|
| 23 | To add data at the end of file, the file must be opened in _______ mode. | read() |
| 24 | When a file is opened read or write mode a file pointer is set at _____of the file. | beginning |
| 25 | Whie performing file operations this file must be included in _______ | fstream |
| 26 | The constructor of this class requires ________file name and mode for opening | ofstream |
| 27 | Templates are suitable for _______ data type. | any |
| 28 | Templates can be declared using the keyword _______ | class |
| 29 | Templates is also called as _______ class. | generic |
| 30 | Function Templates can accept only _______ parameters. | one |
| 31 | Select the correct Template definition _______ . | template <class T> |
| 32 | Function Templates are normally defined _______ . | in main function |
| 33 | The statement catches the exception _______ . | catch |
| 34 | In a multiple catch statement the number of throw statements are . | same as catch statement |
| 35 | The exception is generated in _________block. | try |
| 36 | The exception handling one of the function is implicitly invoked. | abort |
| 37 | The exception handling mechanism is basically built upon ______ keyword | try |
| 38 | The point at which the throw is executed is called _________. | try |
| 39 | A template function may be overloaded by _______ function | template |
| 40 | ________function returns true when an input or output operation has failed | eof() |
| 41 | .In ________ inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. | Hybrid |
| 42 | .In multi-level inheritance, the ________ are executed in the order of inheritance. | Derivations |
| 43 | .A class that contains objects of other classes is known as ________. | Indirect base class |
| 44 | .The ________ section of constructor function is used to assign initial values to its data members. | Initialization |
| 45 | .The grand parent class is sometimes referred to as ________ class. | Ancestor |
| 46 | .________ may arise in single inheritance application. | Ambiguity |
| 47 | .A ________ that contains objects of other classes is known as containership. | Function |

| | | |
|---|---|---|
| 48 | .A member declared as ________ cannot be accessed by the function outside the class. | Private |
| 49 | .The public member of a class can be accessed by its own objects using the ________ operator. | Scope resolution |
| 50 | .The stream is an _________ between I/O devices and the user. | Trans later |
| 51 | . If the data is received from the input devices in sequence then it is called________. | Source stream |
| 52 | . When the data is passed to the output devices it is called_____ | Source stream |
| 53 | . The C++ have a number of stream classes that are used to work with _________ operations. | Console I/O |
| 54 | . The data accepted with default setting by I/O function of the language is known as----- | Formatted |
| 55 | _________ is used as the input stream to read data. | Cout |
| 56 | . cin and cout are ________ for input and output of data. | user defined stream |
| 57 | .The data obtained or represented with some manipulators are called ______. | formatted data |
| 58 | . The output formats can be controlled with manipulators having the header file as | iostream.h |
| 59 | . The _____ and ______ are derived classes from ios based class. | istream and ostream |
| 60 | The manipulator << endl is equivalent to____ | '\t' |
| | | |

| Choice2 | Choice3 | Choice4 | choice5 | choice6 | Ans |
|---|---|---|---|---|---|
| File | Row | Vector | | | **File** |
| converter | translator | operator | | | **interface** |
| ostream | streambuff | fstreambase | | | **fstreambase** |
| sequential | disk | tape | | | **disk** |
| filestream | thread | package | | | **filebuf** |
| ifstream | fstream | iostream | | | **ifstream** |
| fstream | ifstream | istream | | | **ostream** |
| ofstream | fstream | ifstream | | | **fstream** |
| ios::hex | ios::dec | ios::binary | | | **ios::binary** |
| ios::app | ios::del | ios::end | | | **ios::ate** |
| ios::app | ios::ate | ios::out | | | **ios::out** |
| ios::create | ios::create | ios::ate | | | **ios::trunc** |
| ios::trunk, ios::ate | ios::app, ios::ate | ios::app, ios::out | | | **ios::app, ios::ate** |
| ios::app | ios::in | ios::create | | | **ios::in** |
| input and output | append | output | | | **output** |
| error opening file | error of file | destination | | | **end of file** |
| member function | with all function | void | | | **main()** |
| closes all files opened | closes only read mode file | termination | | | **closes the file** |
| object | string | multicharacter | | | **single character** |
| seekp() | tellg() | tellp(). | | | **seekg()** |
| seekp() | tellg() | tellp(). | | | **tellg()** |
| read only | write only | manipulate | | | **both read and rite** |

| | | | | | |
|---|---|---|---|---|---|
| write() | append() | both write and update . | | | **append()** |
| end | middle | last | | | **beginning** |
| iostream | constream | all the above | | | **fstream** |
| ifstream | fstream | all the above | | | **fstream** |
| basic | derived | all the above | | | **basic** |
| template | try | object | | | **template** |
| container | virtual | base | | | **generic** |
| any | two | many | | | **any** |
| class <template T> | template <T> | template class <T>. | | | **template <class T>** |
| globally | in a class | anywhere | | | **in a class** |
| try | template | throw. | | | **catch** |
| twice than catch | only one | final value | | | **only one** |
| catch | finally | throw. | | | **try** |
| exit | assert | terminate | | | **abort** |
| catch | throw | all the above | | | **all the above** |
| catch | throw point | throw | | | **throw point** |
| ordinary | both (template)and (ordinary). | special | | | **both (a)and (b).** |
| fail() | bad() | good() | | | **fail()** |
| Multipath | Hierarchical | Multiple | | | **Multiple** |
| Constructors | Destructors | Containership | | | **Constructors** |
| Nesting | Subclass | Inheritance | | | **Nesting** |
| Declaration | Argument | Assignment | | | **Assignment** |
| Virtual base | Indirect base | Direct base | | | **Indirect base** |
| Visibility | Nesting | Derivation | | | **Ambiguity** |
| Friend | Class | Subclass | | | **Class** |

| | | | | | |
|---|---|---|---|---|---|
| Protected | Public | Visibility | | | **Protected** |
| Relational | Arithmetic | Dot | | | **Dot** |
| Destination | Intermediator | source | | | **Intermediator** |
| Object stream | Destination stream | Input stream. | | | **Source stream** |
| Object stream | Destination stream | Input stream. | | | **Destination stream** |
| Console and file | formatted console | unformatted | | | **Console and file** |
| Unformatted | Argumented | paramerized | | | **Unformatted** |
| Printf | Cin | Scanf. | | | **Cin** |
| system defined stream | Pre defined stream | macro | | | **system defined stream** |
| unformatted data | extracted data | source | | | **formatted data** |
| conio.h | stdlib.h | iomanip.h | | | **iomanip.h** |
| source and destination stream | iostream and source stream | fstream | | | **istream and ostream** |
| ’\r’ | ’\n’ | ’\b’ | | | **’\n’** |
| | | | | | |

Reg.No. .........................

[18CAU101]

**Karpagam Academy of Higher Education**

**(Established Under Section 3 of UGC Act 1956)**

**Coimbatore -641021**

**BCA Degree Examination**

(For the candidates admitted from 2018 onwards)

First Semester

**First Internal Exam**

**PROGRAMMING FUNDAMENTALS USING C/C++**

**Duration: 2 hrs** **Maximum Marks: 50**

**Date & Session:** **Class : I BCA**

**Part – A (20x1=20 Marks)**

**(Answer all the questions)**

1. Which of the following is not a valid variable name declaration?

a) int __a3; b) **int __3a;** c) int __A3; d) None of the mentioned

2. Which of the following is not a valid C variable name?

a) int number; b) float rate; c) int variable_count; d) **int $main;**

3. Which of the following is a User-defined data type?

a) typedef int Boolean; b) **typedef enum {Mon, Tue, Wed, Thu, Fri} Workdays;** c) struct {char name[10], int age}; d) all of the mentioned

4. The format identifier '%d' is also used for _____ data type?

a) char b) **int** c) float d) double

5. Which of the following is not an arithmetic operation?

a) a *= 10; b) a /= 10; c) **a != 10;** d) a %= 10;

6. The precedence of arithmetic operators is (from highest to lowest)

a) %, *, /, +, – b) %, +, /, *, – c) **+, -, %, ***, / d) %, +, -, *, /

7. The concept of object is belongs to which category?

a) POP b)**OOP** c)ALGOL d) POPS

8. Which of the following is an invalid assignment operator?

a) a %= 10; b) **a /= 10;** c) a |= 10; d) None of the mentioned

9. ___is a fixed meaning and these meaning cannot be changed

a) identifier b) function c) **keywords** d) arrays

10. Which one is the correct trigraph character

a) ??+ b)??/ c)??~ d)**??-**

11. The range of char datatype is

a) 127 to -127 b)128 to -128 c)**-128 to 127** d) none of the above

12. which storage class is used to mention global variable declaration

a) auto b) **static** c) register d) extern

13. One of the operand is real and another one is integer, the expression is called____

a) mixed mode arithmetic b)**mixed data type** c)mixed mode logic d) mixed mode testing

14. which operator has lowest priority in operator precedence

a) + b)< c), d){

15. which one is the correct rule for switch statement?

a) it must be an integer b) case labels end with semicolon c) at most one default label d) **all of the above**

16. printf and scanf are belongs to which function ?

a) user defined b) **library** c) none d) both a and b

17. Which one is correct regarding the rules of identifier?

a) Cannot be a keyword b) must not contain white spaces c) must consist of letters, digits and underscore d) **all the above**

18. String constant enclosed by_______________

a) **single quote** b) double quote c) parenthesis d) brackets

19. Which one is logical operator?

a) NAND b) XOR c) **NOT** d)XNOR

20. Which one is the compile time operator?

a) **Sizeof** b) Comma c) Plus d) Minus

**Part – B (3x2=6 Marks)**

**(Answer all the questions)**

21. What is variable? How do you declare a variable?

C variable is a named location in a memory where a program can manipulate the data. This location is used to hold the value of the variable.

The value of the C variable may get change in the program.

C variable might be belonging to any of the data type like int, float, char etc.

| **Type** | **Syntax** |
|---|---|
| Variable declaration | data_type variable_name;<br>Example: int x, y, z; char flat, ch; |
| Variable initialization | data_type variable_name = value;<br>Example: int x = 50, y = 30; char flag = ‘x’, ch=’l’; |

22. Define Object.

In the class-based object-oriented programmingparadigm, object refers to a particular instance of a class, where the object can be a combination of variables, functions, and data structures.

23. List types of operators in c.

Types of C operators:

Arithmetic operators.

Assignment operators.

Relational operators.

Logical operators.

Bit wise operators.

Conditional operators (ternary operators)

Increment/decrement operators.

Special operators.

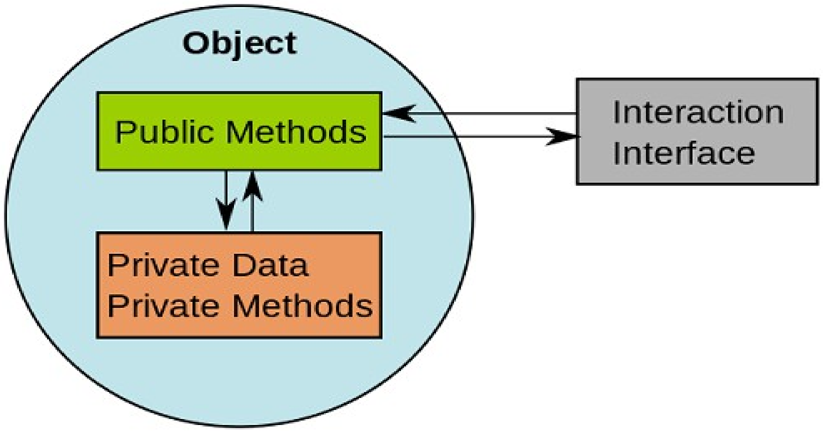
**Part – C (3x8=24 Marks)**

**(Answer all the questions)**

24. (a). Differentiate Procedural Oriented Programming (POP) and Object Oriented Programming (OOP) in detail.

Difference Between OOP and POP

Both are programming processes whereas OOP stands for "Object Oriented Programming" and POP stands for "Procedure Oriented Programming". Both are programming languages that use high-level programming to solve a problem but using different approaches. These approaches in technical terms are known as programming paradigms. A programmer can take different approaches to write a program because there's no direct approach to solve a particular problem. This is where programming languages come to the picture. A program makes it easy to resolve the problem using just the right approach or you can say 'paradigm'. Object-oriented programming and procedure-oriented programming are two such paradigms.




**What is Object Oriented Programming (OOP)?**

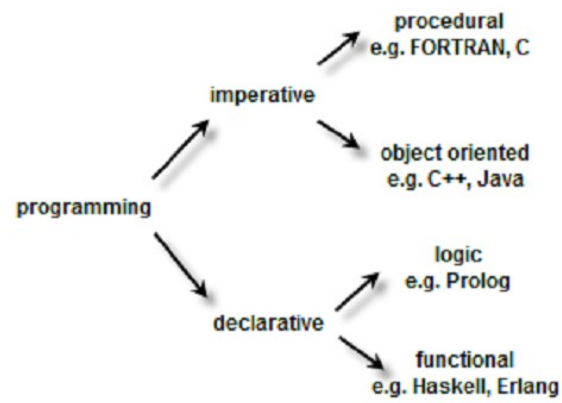
OOP is a high-level programming language where a program is divided into small chunks called objects using the object-oriented model, hence the name. This paradigm is based on objects and classes.

**Object** – An object is basically a self-contained entity that accumulates both data and procedures to manipulate the data. Objects are merely instances of classes.

**Class** – A class, in simple terms, is a blueprint of an object which defines all the common properties of one or more objects that are associated with it. A class can be used to define multiple objects within a program.

The OOP paradigm mainly eyes on the data rather than the algorithm to create modules by dividing a program into data and functions that are bundled within the objects. The modules cannot be modified when a new object is added restricting any non-member function access to the data. Methods are the only way to assess the data.

Objects can communicate with each other through same member functions. This process is known as message passing. This anonymity among the objects is what makes the program secure. A programmer can create a new object from the already existing objects by taking most of its features thus making the program easy to implement and modify.




**What is Procedure Oriented Programming (POP)?**

POP follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do. The program is divided into small parts called functions and then it follows a series of computational steps to be carried out in order.

It follows a top-down approach to actually solve a problem, hence the name. Procedures correspond to functions and each function has its own purpose. Dividing the program into functions is the key to procedural programming. So a number of different functions are written in order to accomplish the tasks.

Initially, all the computer programs are procedural or let's say, in the initial stage. So you need to feed the computer with a set of instructions on how to move from one code to another thereby accomplishing the task. As most of the functions share global data, they move independently around the system from function to function, thus making the program vulnerable. These basic flaws gave rise to the concept of object-oriented programming which is more secure.

## Difference between OOP and POP

### Definition

OOP stands for Object-oriented programming and is a programming approach that focuses on data rather than the algorithm, whereas POP, short for Procedure-oriented programming, focuses on procedural abstractions.

### Programs

In OOP, the program is divided into small chunks called objects which are instances of classes, whereas in POP, the main program is divided into small parts based on the functions.

### Accessing Mode

Three accessing modes are used in OOP to access attributes or functions – 'Private', 'Public', and 'Protected'. In POP, on the other hand, no such accessing mode is required to access attributes or functions of a particular program.

### Focus

The main focus is on the data associated with the program in case of OOP while POP relies on functions or algorithms of the program.

**Execution**

In OOP, various functions can work simultaneously while POP follows a systematic step-by-step approach to execute methods and functions.

**Data Control**

In OOP, the data and functions of an object act like a single entity so accessibility is limited to the member functions of the same class. In POP, on the other hand, data can move freely because each function contains different data.

**Security**

OOP is more secure than POP, thanks to the data hiding feature which limits the access of data to the member function of the same class, while there is no such way of data hiding in POP, thus making it less secure.

**Ease of Modification**

New data objects can be created easily from existing objects making object-oriented programs easy to modify, while there's no simple process to add data in POP, at least not without revising the whole program.

**Process**

OOP follows a bottom-up approach for designing a program, while POP takes a top-down approach to design a program.

**Examples**

Commonly used OOP languages are C++, Java, VB.NET, etc. Pascal and Fortran are used by POP.

| **OOP** |
|---|
| OOP takes a bottom-up approach in designing a program. |
| Program is divided into objects depending on the problem. |
| Each object controls its own data. |
| Focuses on security of the data irrespective of the algorithm. |
| The main priority is data rather than functions in a program. |
| The functions of the objects are linked via message passing. |
| Data hiding is possible in OOP. |
| Inheritance is allowed in OOP. |
| Operator overloading is allowed. |
| C++, Java. |

A program is nothing but a set of step-by-step instructions that only a computer can understand so that it can come up with a solution. There are different approaches to do that, which in technical term, are referred to as programming paradigms.

OOP and POP are such high-level programming paradigms that use different approaches to create a program to solve a particular problem in the less time possible.

The idea is to solve complicated tasks using programming with less code. While an object-oriented program depends mainly upon data rather than the algorithm, a procedure-oriented program follows a step-by-step approach to solve a problem.

OOP, of course, has a little edge over POP on many fronts such as data security, ease of use, accessibility, operator overloading, and more.

(b). what is an operator? List and explain various types of Operators in C.

## Operators in C / C++

Operators are the foundation of any programming language. Thus the functionality of C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that helps us to perform specific mathematical and logical computations on operands. In other words we can say that an operator operates the operands.
For example, consider the below statement:

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

Here, '+' is the operator known as *addition operator* and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'. C/C++ has many built-in operator types and they can be classified as:

**Arithmetic Operators**: These are the operators used to perform arithmetic/mathematical operations on operands. Examples: (+, -, *, /, %,++,–).
Arithmetic operator are of two types:

**Unary Operators**: Operators that operates or works with a single operand are unary operators.
For example: (++ , –)

**Binary Operators**: Operators that operates or works with two operands are binary operators.For example: (+ , – , * , /)

To learn Arithmetic Operators in details visit this link.

**Relational Operators**: Relational operators are used for comparison of the values of two operands. For example: checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not etc. Some of the relational operators are (==, > , = , <= ). To learn about each of these operators in details go to thislink.

**Logical Operators**: Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a boolean value either true or false. To learn about different logical operators in details please visit this link.

**Bitwise Operators**: The Bitwise operators is used to perform bit-level operations on the operands. The operators are first converted to bit-level and then calculation is performed on the operands. The mathematical operations such as addition , subtraction , multiplication etc. can be performed at bit-level for faster processing. To learn bitwise operators in details, visit this link.

**Assignment Operators**: Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of variable on the left side otherwise the compiler will raise an error.
Different types of assignment operators are shown below:

**"="**: This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.
For example:

```
a = 10;
b = 20;
ch = 'y';
```

**"+="**:This operator is combination of '+' and '=' operators. This operator first adds the current value of the variable on left to the value on right and then assigns the result to the variable on the left.
Example:

```
(a += b) can be written as (a = a + b)
```

If initially value stored in a is 5. Then (a += 6) = 11.

**"-="**:This operator is combination of '-' and '=' operators. This operator first subtracts the current value of the variable on left from the value on right and then assigns the result to the variable on the left.
Example:

```
(a -= b) can be written as (a = a - b)
```

If initially value stored in a is 8. Then (a -= 6) = 2.

**"*="**:This operator is combination of '*' and '=' operators. This operator first multiplies the current value of the variable on left to the value on right and then assigns the result to the variable on the left.
Example:

```
(a *= b) can be written as (a = a * b)
```

If initially value stored in a is 5. Then (a *= 6) = 30.

**"/="**:This operator is combination of '/' and '=' operators. This operator first divides the current value of the variable on left by the value on right and then assigns the result to the variable on the left.
Example:

```
(a /= b) can be written as (a = a / b)
```

If initially value stored in a is 6. Then (a /= 2) = 3.

**Other Operators**: Apart from the above operators there are some other operators available in C or C++ used to perform some specific task. Some of them are discussed here:

**sizeof operator**: sizeof is a much used in the C/C++ programming language. It is a compile time unary operator which can be used to compute the size of its operand. The result of sizeof is of unsigned integral type which is usually denoted by size_t. Basically, sizeof operator is used to compute the size of the variable. To learn about sizeof operator in details you may visit this link.

**Comma Operator**: The comma operator (represented by the token ,) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator. Comma acts as both operator and separator. To learn about comma in details visit this link.

**Conditional Operator**: Conditional operator is of the form *Expression1 ? Expression2 : Expression3* . Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3. We may replace the use of if..else statements by conditional operators. To learn about conditional operators in details, visit this link.

### Operator precedence chart

The below table describes the precedence order and associativity of operators in C / C++ . Precedence of operator decreases from top to bottom.

| OPERATOR | DESCRIPTION | ASSOCIATIVITY |
|---|---|---|
| () | Parentheses (function call) | left-to-right |
| [] | Brackets (array subscript) | |
| . | Member selection via object name | |
| -> | Member selection via pointer | |
| ++/– | Postfix increment/decrement | |
| ++/– | Prefix increment/decrement | right-to-left |
| +/- | Unary plus/minus | |
| !~ | Logical negation/bitwise complement | |
| (type) | Cast (convert value to temporary value of type) | |
| * | Dereference | |
| & | Address (of operand) | |
| sizeof | Determine size in bytes on this implementation | |
| *,/,% | Multiplication/division/modulus | left-to-right |

| | | |
|---|---|---|
| +/- | Addition/subtraction | left-to-right |
| <> | Bitwise shift left, Bitwise shift right | left-to-right |
| < , <= | Relational less than/less than or equal to | left-to-right |
| > , >= | Relational greater than/greater than or equal to | left-to-right |
| == , != | Relational is equal to/is not equal to | left-to-right |
| & | Bitwise AND | left-to-right |
| ^ | Bitwise exclusive OR | left-to-right |
| \| | Bitwise inclusive OR | left-to-right |
| && | Logical AND | left-to-right |
| \|\| | Logical OR | left-to-right |
| ?: | Ternary conditional | right-to-left |
| = | Assignment | right-to-left |
| += , -= | Addition/subtraction assignment | |
| *= , /= | Multiplication/division assignment | |
| %= , &= | Modulus/bitwise AND assignment | |
| ^= , \|= | Bitwise exclusive/inclusive OR assignment | |
| <>= | Bitwise shift left/right assignment | |
| , | expression separator | left-to-right |

## 25. (a). Explain the Conditional statements with examples.

Conditional statements are used to execute statement or group of statements based on some conditon.

C supports following conditional statements.

if statement

if else statement

if else if ladder

nested if

go to statement

**a.) if statement :**

Syntax :

If(Conditon) {

C statements;

}

If the condition is true then C statements are executed other wise next statement will be executed.

Example :

File1.c

```
#include<stdio.h>
int main(){
int a=10;
if(a%2==0){
printf(“ a is even no :”);
}
printf(“ statement after if “);
return 0;}
```

**Output :**

a is even no:

statement after if

File2.c

```
#include<stdio.h>
int main(){
int a=10;
if(a%2==0){
printf(“a is even no :”);
}
Printf(“statement after if”);
```

**Output:**

statement after if

**b.) if else statement**

Syntax :-

```
if(condition){
Statements to be executed when condition is true;
}
else {
Statements to be executed when condition is false;
}
```

```
#include<stdio.h>
int main(){
int a=10;
if(a%2==0) {
printf(“ a is even “);
```

```
}
else {
printf("a is odd ");
}
Return 0;
}
```

**Output :**

**a is even.**

**c.) if else if ladder**

Syntax :

```
If(condition1){
Statements to be executed when  condition1 is true;
}
else if(condition2){
Statements to be executed when  condition2 is true;
}
else if(condition3){
Statements to be executed when  condition3 is true;
}
else if(….){
… .. .
…
}
else {
Statements to be executed when no condition is true;
}
```

Example :

```
#include<stdio.h>
int main(){
int a;
printf("n Enter the no of day :" );
scanf("%d",&a);
if(a==1){
printf(" Monday");
}
else if(a==2){
```

```
printf("Tuesday");
}
else if(a==3){
printf("Wednesdat");
}
else if(a==4){
printf("Thursday");
}
else if(a==5){
printf("Friday");
}
else if(a==6){
printf("Saturday");
}
else if(a==7){
printf("Sunday !! ");
}
else {
printf(" Enter the valid day between 1-7");
}
return 0;
}
```

**d.) Nested if statements**

if statement within if statements.

Example:

To find max of three no (a,b,c)

Input:

5 10 20

**Output:**

**20 is max**

**e.) go to statement**

Syntax:

So far we have seen conditional statements which are executed when certain condition is true or false.

go to statement is used to branch unconditinally from one point to another point. go to requires a label to identify where the control to be transferred.

Example:

(b). Explain the decision making statements with examples.

### Decision making in C

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- `if` statement
- `switch` statement
- conditional operator statement (`? :` operator)
- `goto` statement

## Decision making with `if` statement

The `if` statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple `if` statement
2. `if....else` statement
3. Nested `if....else` statement
4. Using `else if` statement

### Simple `if` statement

The general form of a simple `if` statement is,

```
if(expression)
{
    statement inside;
}
    statement outside;
```

If the *expression* returns true, then the **statement-inside** will be executed, otherwise **statement-inside** is skipped and only the **statement-outside** is executed.

Example:

```
#include <stdio.h>

void main( )
{
    int x, y;
```

```
        x = 15;
        y = 13;
        if (x > y )
        {
                printf("x is greater than y");
        }
}
```

x is greater than y

## `if...else` statement

The general form of a simple `if...else` statement is,

```
if(expression)
{
        statement block1;
}
else
{
        statement block2;
}
```

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

Example:

```
#include <stdio.h>


void main( )
{
        int x, y;
        x = 15;
        y = 18;
        if (x > y )
        {
                printf("x is greater than y");
        }
        else
        {
                printf("y is greater than x");
        }
}
```

y is greater than x

### Nested `if....else` statement

The general form of a nested `if...else` statement is,

```
if( expression )
{
        if( expression1 )
        {
                statement block1;
        }
        else
        {
                statement block2;
        }
}
else
{
        statement block3;
}
```

if *expression* is false then **statement-block3** will be executed, otherwise the execution continues and enters inside the first `if` to perform the check for the next `if` block, where if *expression 1* is true the **statement-block1** is executed otherwise **statement-block2** is executed.

## Example:

```
#include <stdio.h>

void main( )
{
        int a, b, c;
        printf("Enter 3 numbers...");
        scanf("%d%d%d",&a, &b, &c);
        if(a > b)
        {
                if(a > c)
                {
                        printf("a is the greatest");
                }
                else
                {
                        printf("c is the greatest");
```

```
                }
        }
        else
        {
                if(b > c)
                {
                        printf("b is the greatest");
                }
                else
                {
                        printf("c is the greatest");
                }
        }
}
```

### `else if` ladder

The general form of else-if ladder is,

```
if(expression1)
{
        statement block1;
}
else if(expression2)
{
        statement block2;
}
else if(expression3 )
{
        statement block3;
}
else
        default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a **true** condition is found, the statement associated with it is executed.

Example :

```
#include <stdio.h>

void main( )
{
```

```
    int a;

    printf("Enter a number...");

    scanf("%d", &a);

    if(a%5 == 0 && a%8 == 0)

    {

        printf("Divisible by both 5 and 8");

    }

    else if(a%8 == 0)

    {

        printf("Divisible by 8");

    }

    else if(a%5 == 0)

    {

        printf("Divisible by 5");

    }

    else

    {

        printf("Divisible by none");

    }

}
```

(a). Explain the concepts of functions in detail.

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 is such that 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, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as 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 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

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value 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(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a 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, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

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

Live Demo

```
#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;
}
```

We have kept max() along with main() and compiled the source code. While running the 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.

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 in which arguments can be passed to a function −

| Sr.No. | Call Type & Description |
|---|---|
| 1 | **Call by value**<br>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. |
| 2 | **Call by reference**<br>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. |

(b). Differentiate while and do-while statements in detail with example

| BASIS FOR COMPARISON | WHILE | DO-WHILE |
|---|---|---|
| General Form | while ( condition) {<br>statements; //body of loop<br>} | do{<br>.<br>statements; // body of loop.<br>.<br>} while( Condition ); |
| Controlling Condition | In 'while' loop the controlling condition appears at the start of the loop. | In 'do-while' loop the controlling condition appears at the end of the loop. |
| Iterations | The iterations do not occur if, the condition at the first iteration, appears false. | The iteration occurs at least once even if the condition is false at the first iteration. |

Iteration statements allow the set of instructions to execute repeatedly till the condition doesn’t turn out false. The Iteration statements in C++ and Java are, for loop, while loop and do while loop. These statements are commonly called loops. Here, the main difference between a while loop and do while loop is that while loop check condition before iteration of the loop, whereas do-while loop, checks the condition after the execution of the statements inside the loop.

In this article, we are going to discuss the differences between “while” loop and “do-while” loop.

Content: while Vs do-while Loop



Comparison Chart


| BASIS FOR COMPARISON | WHILE | DO-WHILE |
|---|---|---|
| General Form | while ( condition) {<br>statements; //body of loop<br>} | do{<br>.<br>statements; // body of loop.<br>.<br>} while( Condition ); |
| Controlling Condition | In 'while' loop the controlling condition appears at the start of the loop. | In 'do-while' loop the controlling condition appears at the end of the loop. |

| BASIS FOR COMPARISON | WHILE | DO-WHILE |
|---|---|---|
| Iterations | The iterations do not occur if, the condition at the first iteration, appears false. | The iteration occurs at least once even if the condition is false at the first iteration. |

Definition of while Loop

The while loop is the most fundamental loop available in C++ and Java.  The working of a while loop is similar in both C++ and Java.The general form of while loop is:

1.**while** ( condition) {

2.statements; //body of loop

3.}

The while loop first verifies the condition, and if the condition is true then, it iterates the loop till the condition turns out false. The condition in while loop can be any boolean expression. When expression returns any non-zero value, then the condition is “true”, and if an expression returns a zero value, the condition becomes “false”. If the condition becomes true, then loop iterates itself, and if the condition becomes false, then the control passes to the next line of the code immediately followed by the loop.

The statements or the body of the loop can either be an empty statement or a single statement or a block of statements.

Definition of do-while Loop

As in while loop, if the controlling condition becomes false in the first iteration only, then the body of the while loop is not executed at all. But the do-while loop is somewhat different from while loop. The do-while loop executes the body of the loop at least once even if the condition is false at the first attempt.

The general form of do-while is as follows.

1.do{

2..

3.statements // body of loop.

4..

5.} **while**( Condition );

In a do-while loop, the body of loop occurs before the controlling condition, and the conditional statement is at the bottom of the loop. As in while loop, here also, the body of the loop can be empty as both C++ and Java allow null statements or, there can be only a single statement or, a block of statements. The condition here is also a boolean expression, which is true for all non-zero value.

In a do-while loop, the control first reaches to the statement in the body of a do-while loop. The statements in the body get executed first and then the control reaches to the condition part of the loop. The condition is verified and, if it is true, the loop is iterated again, and if the condition is false, then the control resumes to the next line immediate after the loop.

**Reg.No. .........................**

**[18CAU101]**

**Karpagam Academy of Higher Education**
**(Established Under Section 3 of UGC Act 1956)**
**Coimbatore -641021**
**BCA Degree Examination**
(For the candidates admitted from 2018 onwards)
First Semester
**Third Internal Exam**
**PROGRAMMING FUNDAMENTALS USING C/C++**

**Duration: 2 hrs** **Maximum Marks: 50Marks**
**Date & Session:** **Class : I BCA**

**Part – A (20x1=20 Marks)**
**(Answer all the questions)**

1.The exception is generated in _________block.
**a) try** b)catch c) finally d)throw.

2. Which of the following is a two-dimensional array?
a) array anarray[20][20]; **b) int anarray[20][20];** c) int array[20, 20];
d)char array[20];

3. The ____ functions are used to handle the single character I/O operation.
a) get() and put() b)clrscr() and getch() c) cin and cout
**d) None**

4. The overloading operator must have atleast ________ operand that is of user-defined data type.
a) Two b) Three **c) One** d) Four

5. The eof ( ) stands for _____.
**a) end of file** b) error opening file c) error of file d) closing a file

6. _______ enables an object to initialize itself when it is created
a) Destructor **b) constructor** c) overloading
d)member

7.What function should be used to free the memory allocated by calloc() ?
a) dealloc(); **b) malloc(variable_name, 0)** c) free();
d)memalloc(variable_name, 0)

8. Constructors cannot be _________
**a) Inherited** b) destroyed c) both a & b
d)created

9. The functions which are declared inside the class are known as ______
**a) Member function** b)member variables c) data variables
d)function variable

10. Duplication of inherited members of ________ inheritance can be avoided by making the common base class, a virtual base class.

a) Single b) Multi-level c) Multipath **d)Hierarchical**

11.The -------------- is invoked whenever an object of its associated class is created.

**a) Default constructor** b) destructor c) constructor d) parameterized

12. The class variables are known as ________

a) Functions **b) members** c) objects d) none of the above

13. ________ inheritance may lead to duplication of inherited members from a 'grand parent' base class.

**a) Multiple** b) Multipath c) Hybrid d) Single

14. Multiple functions with the same name is known as ________________

**a) Function overloading** b) function polymorphism
c) both a & b d)operator overloading

15. ________ operator function should be a class member.

a) Arithmetic b) Relational c) Casting **d)Overloading**

16. What are mandatory parts in function declaration?

**a) return type, function name** b) return type, function name, parameters
c) both a and b d) none of the mentioned

17. Which of the following correctly declares an array?

**a) int array[10];** b) int array; c) array{10}; d) array array[10];

18. The ____ functions are used to handle the single character I/O operation.

a) get() and put() b) clrscr() and getch() c) cin and cout **d)Printf()and scanf()**

19. What function should be used to free the memory allocated by calloc() ?

a) dealloc(); **b) malloc(variable_name, 0)**
c) free(); d) memalloc(variable_name, 0)

20. A _____ is a sequence of bytes.

**a) Stream** b) class c) object d) function

**Part – B (3*2=6 Marks)**
**(Answer all the questions)**

**21. Give the syntax for opening and closing a file.**

**Opening a file** is performed using the library function in the "**stdio.h**" header file: fopen().

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode")
```

For Example:

```
fopen("E:\\cprogram\\newprogram.txt","w");

open("E:\\cprogram\\oldprogram.bin","rb");
```

**Closing a file** is performed using library function fclose().

```
fclose(fptr); //fptr is the file pointer associated with file to be closed.
```

**22. What is a constructor?**

A **constructor** is a special type of member function that initialises an object automatically when it is created. Compiler identifies a given member function is a **constructor** by its name and the return type. **Constructor** has the same name as that of the class and it does not have any return type.

**23. What's the difference between public, private and protected?**

A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of **public** variables without any member.
A**private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access **private** members.

**Part – C (3*8=24 Marks)**
**(Answer all the questions)**

**24.a) With proper example explain new and delete operators in c++.**

**new operator**

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- **Syntax to use new operator**: To allocate memory of any data type, the syntax is:
- pointer-variable = **new** data-type;

  Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.
  Example:

```
// Pointer initialized with NULL

// Then request memory for the variable
```

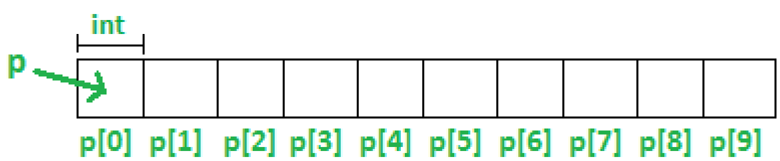
```
int *p = NULL;

p = new int;

 // Combine declaration of pointer

// and their assignment

int *p = new int;
```

- **Initialize memory:** We can also initialize the memory using new operator:
- pointer-variable = **new** data-type(value);
- **Example:**
- int *p = new int(25);
- float *q = new float(75.25);
- **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type *data-type*.
- pointer-variable = **new** data-type[size];

  where size(a variable) specifies the number of elements in an array.

```
Example:

        int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.




**Normal Array Declaration vs Using new**

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

**What if enough memory is not available during runtime?**

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc and new operator returns a pointer. Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new int;

if (!p)

{

  cout << "Memory allocation failed\n";

}
```

**delete operator**

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

**Syntax:**

```
// Release memory pointed by pointer-variable

delete pointer-variable;
```

Here, pointer-variable is the pointer that points to the data object created by *new*.

Examples:

```
  delete p;

  delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
// Release block of memory

// pointed by pointer-variable

delete[] pointer-variable;

Example:

   // It will free the entire array

   // pointed by p.

   delete[] p;
```

```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;
  int main ()
{
    // Pointer initialization to null
    int* p = NULL;
      // Request memory for the variable
    // using new operator
    p = new int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }
      // Request block of memory
    // using new operator
    float *r = new float(75.25);
      cout << "Value of r: " << *r << endl;
      // Request block of memory of size n
    int n = 5;
```

```
    int *q = new int[n];
      if (!q)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i+1;

        cout << "Value store in block of memory: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
    }
      // freed the allocated memory
    delete p;
    delete r;
      // freed the block of allocated memory
    delete[] q;
      return 0;
}
```

Output:

```
Value of p: 29

Value of r: 75.25

Value store in block of memory: 1 2 3 4 5
```

**b).Explain reading and writing text files.**

**Read/Write Class Objects from/to File in C++**

Given a file “Input.txt” in which every line has values same as instance variables of a class. Read the values into the class’s object and do necessary operations.

Theory :

The data transfer is usually done using '>>'

and <<' operators. But if you have

a class with 4 data members and want

to write all 4 data members from its

object directly to a file or vice-versa,

we can do that using following syntax :

**To write object's data members in a file :**

```
// Here file_obj is an object of ofstream
file_obj.write((char *) & class_obj, sizeof(class_obj));
```

**To read file's data members into an object :**

```
// Here file_obj is an object of ifstream
file_obj.read((char *) & class_obj, sizeof(class_obj));
```

Examples:

```
Input :
Input.txt :
Micheal 19 1806
Kemp 24 2114
Terry 21 2400
Operation : Print the name of the highest
            rated programmer.

Output :
Terry
```

```
// C++ program to demonstrate read/write of class
// objects in C++.
#include <iostream>
#include <fstream>
using namespace std;

// Class to define the properties
class Contestant {
public:
    // Instance variables
    string Name;
    int Age, Ratings;

    // Function declaration of input() to input info
    int input();

    // Function declaration of output_highest_rated() to
    // extract info from file Data Base
    int output_highest_rated();
};

// Function definition of input() to input info
int Contestant::input()
{
    // Object to write in file
    ofstream file_obj;
      // Opening file in append mode
    file_obj.open("Input.txt", ios::app);
      // Object of class contestant to input data in file
    Contestant obj;
      // Feeding appropriate data in variables
    string str = "Micheal";
    int age = 18, ratings = 2500;
      // Assigning data into object
    obj.Name = str;
    obj.Age = age;
```

```
    obj.Ratings = ratings;
      // Writing the object's data in file
    file_obj.write((char*)&obj, sizeof(obj));
      // Feeding appropriate data in variables
    str = "Terry";
    age = 21;
    ratings = 3200;
      // Assigning data into object
    obj.Name = str;
    obj.Age = age;
    obj.Ratings = ratings;
      // Writing the object's data in file
    file_obj.write((char*)&obj, sizeof(obj));
      return 0;
}
// Function definition of output_highest_rated() to
// extract info from file Data Base
int Contestant::output_highest_rated()
{
    // Object to read from file
    ifstream file_obj;
      // Opening file in input mode
    file_obj.open("Input.txt", ios::in);
      // Object of class contestant to input data in file
    Contestant obj;
      // Reading from file into object "obj"
    file_obj.read((char*)&obj, sizeof(obj));
      // max to store maximum ratings
    int max = 0;
      // Highest_rated stores the name of highest rated contestant
    string Highest_rated;
      // Checking till we have the feed
    while (!file_obj.eof()) {
        // Assigning max ratings
        if (obj.Ratings > max) {
            max = obj.Ratings;
            Highest_rated = obj.Name;
        }
          // Checking further
        file_obj.read((char*)&obj, sizeof(obj));
    }
      // Output is the highest rated contestant
    cout << Highest_rated;
    return 0;
}

// Driver code
int main()
{
```

```
    // Creating object of the class
    Contestant object;

    // Inputting the data
    object.input();

    // Extracting the max rated contestant
    object.output_highest_rated();

    return 0;
}
```

**25.a) Explain class constructors with suitable example program**

**Constructors in C++**

**What is constructor?**

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.

**How constructors are different from a normal member function?**

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

**Types of Constructors**

1. **Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
```

```
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}
```

Output:

```
a: 10

b: 20
```

**Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
```

```
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call

Example e(0, 50);          // Implicit call
```

**b) List the different types of inheritance. Explain multiple with suitable program.**

**Inheritance in C++**

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

**Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.

**Super Class:**The class whose properties are inherited by sub class is called Base Class or Super class.

**The article is divided into following subtopics:**

1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

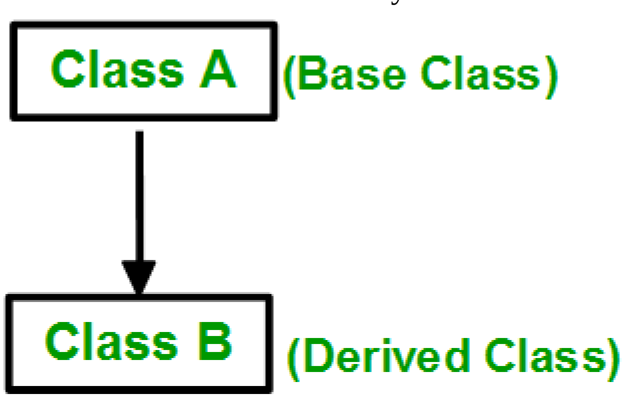
**Why and when to use inheritance?**

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods

fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes.

**Types of Inheritance in C++**

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.




**Syntax**:

```
class subclass_name : access_mode base_class
```

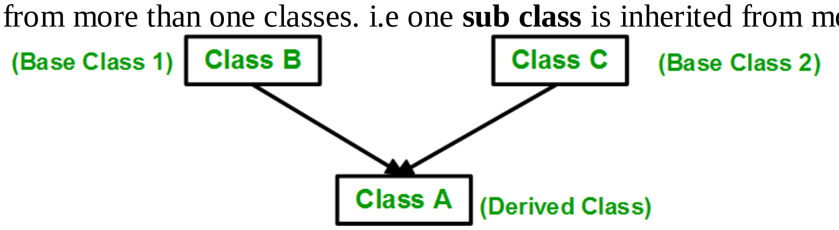
```
{
  //body of subclass
};
```

```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a vehicle
```

2. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.




**Syntax**:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

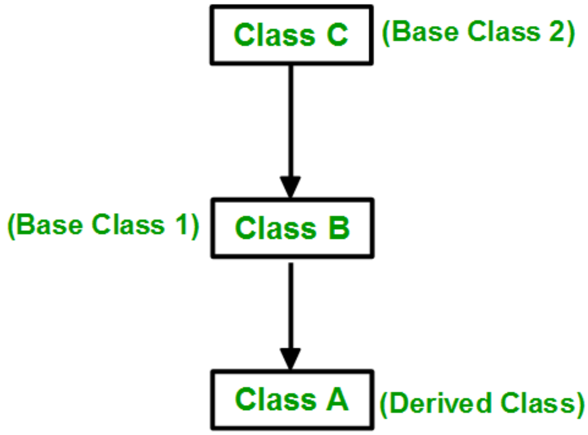
Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a Vehicle

This is a 4 wheeler Vehicle
```

**Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



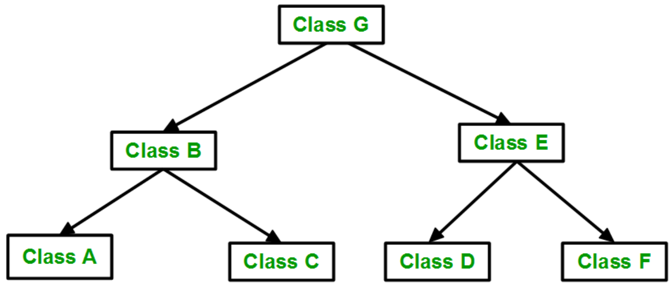

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
   public:
     car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};

// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
```

```
    Car obj;
    return 0;
}
```

output:

```
This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels
```

**Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.




```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};


// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// main function
```

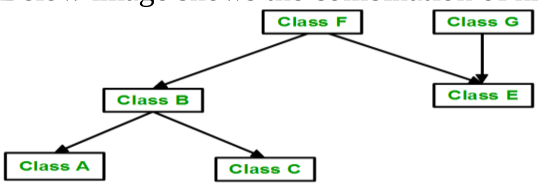
```
int main()
{
   // creating object of sub class will
   // invoke the constructor of base class
   Car obj1;
   Bus obj2;
   return 0;
}
```

Output:

```
This is a Vehicle

This is a Vehicle
```

**Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritance:




```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

//base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};

// first sub class
class Car: public Vehicle
```

```
{

};

// second sub class
class Bus: public Vehicle, public Fare
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle

Fare of Vehicle
```

**26.a) Explain in detail about overloading operators with example**

**Operator Overloading in C++**

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

**A simple and complete example**

```
#include<iostream>
using namespace std;

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}

   // This is automatically called when '+' is used with
   // between two Complex objects
   Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
```

```
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:

```
12 + i9
```

**b) Write note on catching all exceptions in c++.**

**Exception Handling in C++**

One of the advantages of C++ over C is Exception Handling. C++ provides following specialized keywords for this purpose.

*try*: represents a block of code that can throw an exception.
*catch*: represents a block of code that is executed when a particular exception is thrown.
*throw*: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

**Why Exception Handling?**

Following are main advantages of exception handling over traditional error handling.

***1)*** *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
***2)*** *Functions/Methods can handle any exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)
***3)*** *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.