KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
DEPARTMENT OF CS, CA & IT
SYLLABUS

SUBJECT NAME: PROGRAMMING FUNDAMENTALS USING C / C++
SUBJECT CODE: 19CSU101
SEMESTER: I **CLASS: I BSc CS -A**

Instruction Hours / week: L: 5 T: 0 P: 0 **Marks:** Internal : **40** External : **60** Total: **100**
End Semester Exam : 3 Hours

Course Objectives

- To impart adequate knowledge on the need of programming languages and problem solving techniques.
- To develop programming skills using the fundamentals and basics of C Language.
- To enable effective usage of arrays, structures, functions, pointers and to implement the memory management concepts.
- To teach the issues in file organization and the usage of file systems.
- To learn the characteristics of an object-oriented programming language: data abstraction and information hiding, inheritance, and dynamic binding of the messages to the methods.

Course Outcomes (COs)

After the completion of this course, a successful student will be able to do the following:

1. Obtain the knowledge about the number systems this will be very useful for bitwise operations.
2. Develop programs using the basic elements like control statements, Arrays and Strings .
3. Solve the memory access problems by using pointers
4. understand about the dynamic memory allocation using pointers which is essential for utilizing memory
5. Understand about the code reusability with the help of user defined functions.
6. Develop advanced applications using enumerated data types, function pointers and nested structures.
7. Learn the basics of file handling mechanism that is essential for understanding the concepts in database management systems.
8. Understand the uses of preprocessors and various header file directives.
9. Use the characteristics of an object-oriented programming language in a program.
10. Use the basic object-oriented design principles in computer problem solving.

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, putcharetc), Formatted and Console I/O (printf(), scanf(), cin, cout), Using Basic Header Files (stdio.h, iostream.h, conio.hetc).

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. Herbtz Schildt. (2003). C++: The Complete Reference (4th ed.) McGraw Hill, New Delhi.
2. Bjarne Stroustrup. (2013). The C++ Programming Language(4th ed.). Addison-Wesley, New Delhi.
3. Bjarne Stroustroup. (2014). Programming, Principles and Practice using C++(2nd edAddison-Wesley, New Delhi.
4. Balaguruswamy, E. (2008). Object Oriented Programming with C++. Tata McGraw-Hill Education, New Delhi.
5. Paul Deitel., & Harvey Deitel. (2011). C++ How to Program (8th ed.). Prentice Hall, New Delhi.
6. John, R. Hubbard. (2000). Programming with C++- (2nd ed.). Schaum's Series.
7. Andrew Koeni., Barbara, E. Moo. (2000). Accelerated C++. Addison-Wesley.
8. Scott Meyers. (2005). Effective C++ (3rd ed.).Addison-Wesley,.
9. Harry, H. Chaudhary. (2014). Head First C++ Programming: The Definitive Beginner's Guide. LLC USA: First Create space Inc, O-D Publishing.
10. Walter Savitch.(2007) Problem Solving with C++, Pearson Education,.
11. Stanley, B. Lippman., Josee Lajoie., & Barbara, E. Moo. (2012). C++ Primer, 5th ed.). Addison-Wesley

WEB SITES

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)
Coimbatore - 641021.
(For the candidates admitted from 2019 onwards)
DEPARTMENT OF CS,CA & IT

Enable | Enlighten | Enrich
(Deemed to be University)
(Under Section 3 of UGC Act 1956)

SUBJECT :PROGRAMMING FUNDAMENTALS USING C / C++ **Class : I B.Sc (CS) B**
SEMESTER: I **SUBJECT CODE: 19CSU101**

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, putcharetc)

Formatted and Console I/O (printf(), scanf(), cin, cout)

Using Basic Header Files (stdio.h, iostream.h, conio.hetc)

Expressions, Conditional Statements and Iterative Statements:

Simple Expressions in C++ (Unary and 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,

Introduction C and 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 –

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.

Dennis Ritchie is known as the **founder of c language**.

It was developed to overcome the problems of previous languages such as B, BCPL etc.

Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and B-CPL.

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

Facts about C

- C 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.

Introduction -C++

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by BjarneStroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Note − A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

A Brief History of C:

The C programming language was developed at Bell Labs during the early 1970's. Quite unpredictably it derived from a computer language named B and from an earlier language BCPL. Initially designed as a system programming language under UNIX it expanded to have wide usage on many different systems. The earlier versions of C became known as K&R C after the authors of an earlier book, "The C Programming Language" by Kernighan and Ritchie. As the language further developed and standardized, a version know as ANSI (American National Standards Institute) C became dominant. As you study this language expect to see references to both K&R and ANSI C. Although it is no longer the language of choice for most new development, it still is used for some system and network programming as well as for embedded systems. More importantly, there is still a tremendous amount of legacy software still coded in this language and this software is still actively maintained.

A Brief History of C++:

BjarneStroustrup at Bell Labs initially developed C++ during the early 1980's. It was designed to support the features of C such as efficiency and low-level support for system level coding. Added to this were features such as classes with inheritance and virtual functions, derived from the Simula67 language, and operator overloading, derived from Algol68. Don't worry about understanding all the terms just yet, they are explained in easyCPlusPlus's C++ tutorials. C++ is best described as a superset of C, with full support for object-oriented programming. This language is in wide spread use.

Differences between C and C++:

Although the languages share common syntax they are very different in nature. C is a procedural language. When approaching a programming challenge the general method of solution is to break the task into successively smaller subtasks. This is known as top-down design. C++ is an object-oriented language. To solve a problem with C++ the first step is to design classes that are abstractions of physical objects. These classes contain both the state of the object, its members, and the capabilities of the object, its methods. After the classes are designed, a program is written that uses these classes to solve the task at hand.

PROEDURE ORIENTED PROGRAMMIMG (POP)

- **Procedural programming** uses a list of instructions to tell the computer what to do step-by-step.
- It based upon the concept of the *procedure call*.
- Procedures, also known as routines, or functions (not to be confused with mathematical functions), but similar to those used in functional 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.
- that most of the early programming languages are all procedural.
- Examples of procedural languages include **FORTRAN**, **COBOL** and **C**, **Pascal** which have been around since the 1960s and 70s.

Characteristics of Procedural oriented programming:-

- It focuses on process rather than data.
- It takes a problem as a sequence of things to be done such as reading, calculating and printing. Hence, a number of functions are written to solve a problem.
- A program is divided into a number of functions and each function has clearly defined purpose.
- Most of the functions share global data.
- Data moves openly around the system from function to function.

Drawback of Procedural oriented programming (structured programming):-

- Data is given a second class status even through data is the reason for the existence of the program.
- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.

- It is often difficult to design because the components function and data structure do not model the real world.

OBJECT ORIENTED PROGRAMMING

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are a few principle concepts that form the foundation of object-oriented programming –

Object

Object is a real time entity .This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class

Class is a blue print of an object.When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction

Showing essential features and hiding background details is called abstraction.

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation

Encapsulation is data and function into a single unit is called encapsulation.placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance

Inherit theproperties and (methods)behaviors from one class to another class. One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class. This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism

It ability to take more than one form.The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

main() **function**

main() function is the entry point of any C++ program. It is the point at which execution of program is started. When a C++ program is executed, the execution control goes directly to the main() function. Every C++ program have a main() function.

Syntax

```
void main()
{
...........
...........
}
```

In above syntax;

- **void:** is a keyword in C++ language, void means nothing, whenever we use void as a function return type then that function nothing return. here main() function no return any value.
- In place of void we can also use **int** return type of main() function, at that time main() return integer type value.
- **main:** is a name of function which is predefined function in C++ library.

Simple example of main()

Example

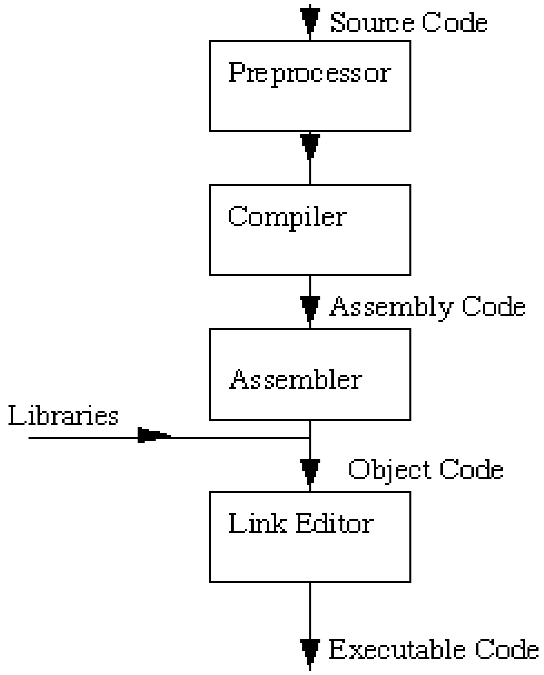
```
#include<iostream>
Using namespace std;
void main()
{
cout<<"This is main function";
}
```

Output

```
This is main function
```

Compiling and executing simple program in c++

The C Compilation Model:



The C Compilation Model:

The Preprocessor

We will study this part of the compilation process in greater detail later

The Preprocessor accepts source code as input and is responsible for

- removing comments
- Interpreting special ***preprocessor directives*** denoted by #.

For example

- #include -- includes contents of a named file. Files usually called ***header*** files. ***e.g***
    - #include <math.h> -- standard library maths file.
    - #include <stdio.h> -- standard library I/O file
- #define -- defines a symbolic name or constant. Macro substitution.
    - #define MAX_ARRAY_SIZE 100

C Compiler:

The C compiler translates source to assembly code. The source code is received from the preprocessor.

Assembler:

The assembler creates object code. On a UNIX system you may see files with a .o suffix (.OBJ on MSDOS) to indicate object code files.

Link Editor:

If a source file references library functions or functions defined in other source files the ***link editor*** combines these functions (with main()) to create an executable file. External Variable references resolved here also.

Variables

While doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

What are Variables?

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.

Basic types of Variables

Each variable while declaration must be given a data type, on which the memory assigned to the variable, depends. Following are the basic types of variables,

bool - For variable to store boolean values(True or False)

char - For variables to store character types.

int - for variable with integral values

float and double are also types for variables with large and floating point values

Declaration and Initialization:

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

***Example*:**

```
int i;      // declared but not initialized
char c;
int i, j, k;  // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i;   // declaration
i = 10;  // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10;         //initialization and declaration in same step
int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
inti,j;
```

```
i=10;
j=20;
int j=i+j;    //compile time error, cannot redeclare a variable in same scope
```

Scope of Variable

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

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

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables −

```
#include<iostream>
usingnamespacestd;

int main ()
{
// Local variable declaration:
int a, b;
int c;

// actual initialization
  a =10;
  b =20;
  c = a + b;

cout<< c;

return0;
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

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

```
#include<iostream>
usingnamespacestd;

// Global variable declaration:
int g;

int main  ( )
{
// Local variable declaration:
int a, b;

// actual initialization
  a =10;
  b =20;
  g = a + b;

cout<< g;

return0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example –

```
#include<iostream>
usingnamespacestd;

// Global variable declaration:
int g =20;

int main ()
{
// Local variable declaration:
int g =10;

cout<< g;

return0;
}
```

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

```
10
```

Defining Constants

There are two simple ways in C++ to define constants –

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

The #define Preprocessor

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

```
#define identifier value
```

Following example explains it in detail –

```
#include<iostream>
usingnamespacestd;
#define LENGTH 10
#defineWIDTH5
#define NEWLINE '\n'
intmain()
{
int area;

area= LENGTH * WIDTH;
cout<< area;
cout<< NEWLINE;
return0;
}
```

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

```
50
```

The constant –(const) Keyword

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

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

Following example explains it in detail –

```
#include<iostream>
usingnamespacestd;

intmain()
{
constint  LENGTH=10;
constint  WIDTH=5;
constchar NEWLINE ='\n';
int area;

area= LENGTH * WIDTH;
cout<< area;
cout<< NEWLINE;
return0;
}
```

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

```
50
```

Data Types

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types –

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers –

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767

long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

```
#include<iostream>

usingnamespacestd;

intmain()

{

cout<<"Size of char : "<<sizeof(char)<<endl;

cout<<"Size of int : "<<sizeof(int)<<endl;

cout<<"Size of short int : "<<sizeof(shortint)<<endl;

cout<<"Size of long int : "<<sizeof(longint)<<endl;

cout<<"Size of float : "<<sizeof(float)<<endl;

cout<<"Size of double : "<<sizeof(double)<<endl;

cout<<"Size of wchar_t : "<<sizeof(wchar_t)<<endl;


return0;

}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine –

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

Keywords In C++

The C++ Keywords are reserved words by the compiler. All keywords have been assigned a fixed meaning. They cannot be used as variable names because they have been assigned fixed jobs.

C++ Keyword List :

asm	**else**	**operator**	**template**
auto	enum	private	this
break	extern	protected	throw
case	float	public	try
catch	for	register	typedef
char	friend	return	union
class	goto	short	unsigned
const	if	signed	virtual
continue	inline	sizeof	void
default	int	static	volatile
delete	long	struct	while
double	new	switch	-

Data Types

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types –

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers –

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range

char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

```
#include<iostream>

usingnamespacestd;

intmain()

{

cout<<"Size of char : "<<sizeof(char)<<endl;

cout<<"Size of int : "<<sizeof(int)<<endl;
cout<<"Size of short int : "<<sizeof(shortint)<<endl;

cout<<"Size of long int : "<<sizeof(longint)<<endl;

cout<<"Size of float : "<<sizeof(float)<<endl;

cout<<"Size of double : "<<sizeof(double)<<endl;

cout<<"Size of wchar_t : "<<sizeof(wchar_t)<<endl;

return0;

}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine –

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

typedef Declarations

You can create a new name for an existing type using **typedef**. Following is the simple syntax to define a new type using typedef –

```
typedef type newname;
```

For example, the following tells the compiler that feet is another name for int –

```
typedefint feet;
```

Now, the following declaration is perfectly legal and creates an integer variable called distance –

```
feet distance;
```

Enumerated Types

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.

Creating an enumeration requires the use of the keyword **enum**. The general form of an enumeration type is −

```
enumenum-name { list of names } var-list;
```

Here, the enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, **green** will have the value 5.

```
enum color { red, green = 5, blue };
```

Here, **blue** will have a value of 6 because each name will be one greater than the one that precedes it.

Casting of Data Types-Typecasting

A cast is a special operator that forces one data type to be converted into another. As an operator, a cast is unary and has the same precedence as any other unary operator. Typecasting is a conversion of data in one basic type to another by applying external use of data type keywords.

Example: Program for Typecasting and displaying the converted values

```
#include<iostream>
using namespace std;
int main()
{
	inti = 69;
	float f = 4.5;
	char c = 'D';
	cout<<"Before Typecasting"<<endl;
	cout<<"---------------------------------------------"<<endl;
	cout<<"i = "<<i<<endl;
```

```
        cout<<"f = "<<f<<endl;
        cout<<"c = "<<c<<endl<<endl;
        cout<<"After Typecasting"<<endl;
        cout<<"------------------------------------------"<<endl;
        cout<<"Integer(int) in Character(char) Format : "<<(char)i<<endl;
        cout<<"Float(float) in Integer(int) Format : "<<(int)f<<endl;
        cout<<"Character(char) in Integer(int) Format : "<<(int)c<<endl;
        return 0;
}
```

Output:

```
Before Typecasting
------------------------------------------
i = 69
f = 4.5
c = D

After Typecasting
------------------------------------------
Integer(int) in Character(char) Format : E
Float(float) in Integer(int) Format : 4
Character(char) in Integer(int) Format : 68
```

In the above example, the variables of integer(int), float(float) and character(char) are declared and initialized **i = 69, f = 4.5, c = 'D'.**

In first cout statement, integer value converted into character according to ASCII character set and the character E is displayed.

In second cout statement, float value is converted into integer format. The displayed value is 4 not 4.5, because while performing typecasting from float to integer, it removes decimal part of float value and considers only integer part.

In third cout statement, character converted into integer. The value of 'D' is 69, printed as an integer. The integer format converts character into integer.

In the above diagram, 69 is an integer value and it is converted into character E by using typecasting format (char).

OPERATORS in C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ 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**

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language –

Assume variable A holds 10 and variable B holds 20, then –

Show Examples

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

Arithmetic operator example:

```
include<iostream.h>
int main()
{
```

```
intx,y,sum;
floataverage;
cout<<"Enter 2 integers : "<<endl;
cin>>x>>y;
sum=x+y;
average=sum/2;
cout<<"The sum of "<< x <<" and "<< y <<" is "<< sum <<"."<<endl;
cout<<"The average of "<< x <<" and "<< y <<" is "<< average <<"."<<endl;}
```

Output:

```
Enter 2 integers: 8   4
The sum of 4 and 8 is 12.
The average of 4 and 8 is 6.
```

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

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

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

Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then –

Show Examples

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

Logical operator example:

```
#include<iostream.h>
void main()
{
int a, b;
cout<<”\n Enter the a and b values:”;
cin>>a>>b;
if(a<b)&&(b>a)
cout<<”A is small”;
else
cout<<"B is big";
}
```

Output:

```
1) Enter the a and b values: 100    300          2) Enter the a and b values: 1    3
   A is small                                       B is big
```

Bitwise Operators

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

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

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

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

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

Show Examples

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

Bitwise operator example:

```
#include<iostream.h>
void main()
{
unsignedint a =60;               // 60 = 0011 1100
unsignedint b =13;               // 13 = 0000 1101
int c =0;
 c = a & b;                      // 12 = 0000 1100
cout<<"Line 1 - Value of c is : "<< c <<endl;
 c = a | b;                      // 61 = 0011 1101
cout<<"Line 2 - Value of c is: "<< c <<endl;
  c = a ^ b                      ;// 49 = 0011 0001
cout<<"Line 3 - Value of c is: "<< c <<endl;
  c =~a;                         // -61 = 1100 0011
cout<<"Line 4 - Value of c is: "<< c <<endl;
}
```

Output:

```
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
```

Assignment Operators- (optional)

There are following assignment operators supported by C++ language –

Show Examples

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

^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Operator Precedence

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

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

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

Show Examples

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

Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Try the following example to understand operators precedence concept available in C++. Copy and paste the following C++ program in test.cpp file and compile and run this program.

Check the simple difference with and without parenthesis. This will produce different results because (), /, * and + have different precedence. Higher precedence operators will be evaluated first –

```
#include<iostream>
usingnamespacestd;

main(){
int a =20;
int b =10;
int c =15;
int d =5;
int e;

  e =(a + b)* c / d;// ( 30 * 15 ) / 5
cout<<"Value of (a + b) * c / d is :"<< e <<endl;

  e =((a + b)* c)/ d;// (30 * 15 ) / 5
cout<<"Value of ((a + b) * c) / d is  :"<< e <<endl;

  e =(a + b)*(c / d);// (30) * (15/5)
cout<<"Value of (a + b) * (c / d) is  :"<< e <<endl;

  e = a +(b * c)/ d;//  20 + (150/5)
cout<<"Value of a + (b * c) / d is  :"<< e <<endl;

return0;
}
```

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

```
Value of (a + b) * c / d is :90
Value of ((a + b) * c) / d is  :90
Value of (a + b) * (c / d) is  :90
Value of a + (b * c) / d is  :50
```

Comments in C++

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.
C++ comments start with /* and end with */. For example –

```
/* This is a comment */

/* C++ comments can also
  * span multiple lines
*/
```

A comment can also start with //, extending to the end of the line. For example –

```
#include<iostream>
usingnamespacestd;

main(){
cout<<"Hello World";// prints Hello World

return0;
}
```

When the above code is compiled, it will ignore **// prints Hello World** and final executable will produce the following result –

```
Hello World
```

Within a /* and */ comment, // characters have no special meaning. Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example –

```
/* Comment out printing of Hello World:

cout<< "Hello World"; // prints Hello World

*/
```

Conditional statements

If statement

Syntax

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

Syntax

The syntax of an if statement in C++ is –

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

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

Example

```
#include<iostream>
using name space std;

int main ()
{
// local variable declaration:
int a =10;

// check the boolean condition
if( a<20)   10<20
{
// if condition is true then print the following
cout<<"a is less than 20;"<<endl;
}
cout<<"value of a is : "<< a <<endl;

return0;
}
```

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

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

if and else Statement

If statement

Syntax

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

Syntax

The syntax of an if statement in C++ is –

```
if(boolean_expression)
```

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

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

Example

```
#include<iostream>
using name space std;

int main ()
{
// local variable declaration:
int a =10;

// check the boolean condition
if( a<20)   10<20
{
// if condition is true then print the following
cout<<"a is less than 20;"<<endl;
}
cout<<"value of a is : "<< a <<endl;

return0;
}
```

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

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

The syntax of   if...else statement in C++ is –

An **if else** statement in programming is a conditional statement that runs a different set of statements depending on whether an expression is true or false

Syntax

The syntax of an if...else in C++

```
if(condition)
{
```

```
// Block For Condition Success
}
else
{
// Block For Condition Fail
}
```

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

Example

```
#include<iostream>
usingnamespacestd;

int main ()
{
// local variable declaration:
int a =100;
// check the boolean condition
if( a<20)
            {
// if condition is true then print the following
cout<<"a is less than 20;"<<endl;
            }
else
            {
// if condition is false then print the following
cout<<"a is not less than 20;"<<endl;
            }
cout<<"value of a is : "<< a <<endl;

return0;
}
```

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

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

if...else and else….if Statement

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

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

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

Syntax

The syntax of an if...else if...else statement in C++ is −

```
if(boolean_expression 1)

{

  // Executes when the boolean expression 1 is true

}

else if( boolean_expression 2)

{

  // Executes when the boolean expression 2 is true

}

else if(boolean_expression 3)

 {

  // Executes when the boolean expression 3 is true

}

else

{

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

Example

```
#include<iostream>
usingnamespacestd;
int main ()
{
// local variable declaration:
int a =100;
// check the boolean condition
if( a==10)
{
// if condition is true then print the following
cout<<"Value of a is 10"<<endl;
}
else if( a==20)
{
// if else if condition is true
cout<<"Value of a is 20"<<endl;
}
else if( a==30)
{
// if else if condition is true
cout<<"Value of a is 30"<<endl;
}
else
{
// if none of the conditions is true
cout<<"Value of a is not matching"<<endl;

}

cout<<"Exact value of a is : "<< a <<endl;



return0;

}
```

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

```
Value of a is not matching
Exact value of a is : 100
```

Switch statement:

Switch case statements are a substitute for long if statements that compare a variable to several integral values

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Switch is a control statement that allows a value to change control of execution.

Syntax

The syntax for a **switch** statement in C++ is as follows –

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

case constant-expression  :
statement(s);
break; //optional

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

The following rules apply to a switch statement –

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

Example

```
#include<iostream>
usingnamespacestd;

intmain()
{
char o;
float num1, num2;

cout<<"Enter an operator (+, -, *, /): ";
cin>> o;

cout<<"Enter two operands: ";
cin>> num1 >> num2;

switch (o)
    {
case'+':
cout<< num1 <<" + "<< num2 <<" = "<< num1+num2;
break;
case'-':
cout<< num1 <<" - "<< num2 <<" = "<< num1-num2;
break;
case'*':
cout<< num1 <<" * "<< num2 <<" = "<< num1*num2;
break;
case'/':
cout<< num1 <<" / "<< num2 <<" = "<< num1/num2;
break;
default:
// operator is doesn't match any case constant (+, -, *, /)
cout<<"Error! operator is not correct";
break;
    }

return0;
}
```

Output

```
Enter an operator (+, -, *, /): +


-


Enter two operands: 2.3


4.5


2.3 - 4.5 = -2.2
```

The - operator entered by the user is stored in `o` variable. And, two operands 2.3 and 4.5 are stored in variables `num1` and `num2` respectively.

Then, the control of the program jumps to

```
cout<< num1 << " - " << num2 << " = " << num1-num2;
```

Finally, the break statement ends the switch statement.

If break statement is not used, all cases after the correct case is executed

C++ while Loop

While loop is an entry controlled loop where the condition is checked at the beginning of the loop. The condition to be checked can be changed inside it. The control can exit a loop in two ways, when the condition becomes false or using **break** statement.

The syntax of a while loop is:

```
while (testExpression)
{
    // codes
}
```

where, testExpression is checked on each entry of the while loop.

How while loop works?

- The while loop evaluates the test expression.
- If the test expression is true, codes inside the body of while loop is evaluated.
- Then, the test expression is evaluated again. This process goes on until the test expression is false.
- When the test expression is false, while loop is terminated.

Example 1: C++ while Loop

```
// C++ Program to compute factorial of a number
// Factorial of n = 1*2*3...*n

#include<iostream>
usingnamespacestd;
int main()
{
int number, i =1, factorial =1;

cout<<"Enter a positive integer: ";
cin>> number;

while( i <= number)
              {
factorial*= i;//factorial = factorial * i;
++i;
              }

cout<<"Factorial of "<< number <<" = "<< factorial;
return0;
}
```

Output

```
Enter a positive integer: 4
Factorial of 4 = 24
```

In this program, user is asked to enter a positive integer which is stored in variable number. Let's suppose, user entered .

Then, the while loop starts executing the code. Here's how while loop works:

1. Initially, i = 1, test expression i <= number is true and factorial becomes 1.
2. Variable i is updated to 2, test expression is true, factorial becomes 2.
3. Variable i is updated to 3, test expression is true, factorial becomes 6.
4. Variable i is updated to 4, test expression is true, factorial becomes 24.
5. Variable i is updated to 5, test expression is false and while loop is terminated.

Do-while loop

Do-while loop is a variant of while loop where the condition isn't checked at the top but at the end of the loop, known as **exit controlled loop**. This means statements inside do-while loop are executed at least once and exits the loop when the condition becomes false or **break** statement is used. The condition to be checked can be changed inside the loop as well.

do-while loop is similar to while loop, however there is a difference between them: In while loop, condition is evaluated first and then the statements inside loop body gets executed, on the other hand in do-while loop, statements inside do-while gets executed first and then the condition is evaluated.

Syntax of do-while loop

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

```
#include<iostream>

usingnamespacestd;

int main ()

{
```

```
// Local variable declaration:

int a =10;



// do loop execution

do{

cout<<"value of a: "<< a <<endl;

    a = a +1;

}

while( a <20);

return0;

}
```

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

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

```
#include<iostream>
usingnamespacestd;
int main()
{
intnum=1;
do
{
cout<<"Value of num: "<<num<<endl;
num++;
  }
while(num<=6);
return0;
}
```

Output:

```
Value of num: 1
Value of num: 2
Value of num: 3
Value of num: 4
Value of num: 5
Value of num: 6
```

- ➢ While is entry control looping statements .
- ➢ Do…while is exit control looping statements.
- ➢ The most important **difference between While and Do-While** is that in **Do-While**, the block of code is executed at least once. ...
- ➢ In **While** loop, the condition is first tested and then the block of code is executed if the test result is true.
- ➢ In D0-**While**, the code is first executed and then the condition is checked.

for loop

for loop is an entry controlled loop, where entry controlled means the condition is checked at the beginning of loop. For loop is suitable to use when the number of times a loop runs is known or fixed.

Syntax of for loop

```
for (initialization; condition; increment/decrement)
{
   statement(s);
   ... ... ...
}
```

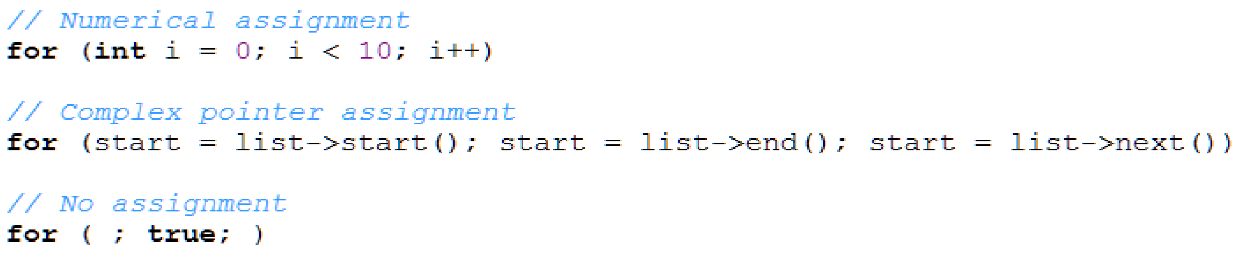
For loop consists of three components

- **Initialization**

  This is a part where a variable is initialized for the loop. It can be a simple numerical assignment or a complex pointer to the start of a list array. However, it is not mandatory to assign a variable. Loops without initialization would only have a semicolon "**;**".

-

For example:



```
// Numerical assignment
for (int i = 0; i < 10; i++)

// Complex pointer assignment
for (start = list->start(); start = list->end(); start = list->next())

// No assignment
for ( ; true; )
```

- **Condition**

  Here, condition for executing the loop is checked. It is evaluated on each loop and runs until the condition is satisfied, otherwise the control exits the loop. This is the only **mandatory** part of for loop.

- **Increment/Decrement**

  This part increments or decrements the value of a variable that is being checked. The control of a program shifts to this part at the end of each loop and doesn't necessarily have to be an increment/decrement statement as shown by the above diagram (Complex pointer assignment). It is also not mandatory to have any statement here as shown by the above diagram (No assignment).

A **for loop** terminates when a break, return, or goto (to a labeled statement outside the**for loop**) within statement is executed. A continue statement in a **for loop**terminates only the current iteration

```
#include<iostream>

usingnamespacestd;
int main()
{
for(int i=1; i<=6; i++)
        {
/* This statement would be executed
   * repeatedly until the condition
   * i<=6 returns false.
   */
cout<<"Value of variable i is: "<<i<<endl;
        }
return0;
}
```

Output:

```
Value of variable i is: 1
Value of variable i is: 2
Value of variable i is: 3
Value of variable i is: 4
Value of variable i is: 5
Value of variable i is: 6
```

Console I/O operations form:

Formatted and Console I/O

There are mainly two types of consol I/O operations form:

1. Unformatted consol input output
2. Formatted consol input output

C++ Basic Input/Output:

C++ I/O occurs in streams, which are sequences of bytes. If bytes flows from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device likes a display screen, a printer, a disk drive, or a network connection, etc, this is called **output operation**.

I/O Library Header Files:

There are following header files important to C++ programs:

Header File	Function and Description
<iostream>	This file defines the cin, cout, objects, which correspond to the standard input stream, the standard output stream.

Using in C-Language

<stdio>	This files defines the printf(), scanf() functions, which correspond to the standard input, the standard output

<conio>	This header declares several useful library functions for performing "console input and output" from a program like clrscr(),getch() functions.

The standard output stream (cout):

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" thestandard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream.h>
int main( )
{
charstr[] = "Hello C++";

cout<<"Value of str is : "<<str<<endl;
}
```

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

```
Value of stris : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The standard input stream (cin):

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
int main( )
{
char name[50];
cout<<"Please enter your name: ";
cin>> name;
cout<<"Your name is: "<< name <<endl;

}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the result something as follows:

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

```
cin>> name >> age;
```

This will be equivalent to the following two statements:

```
cin>> name;
cin>> age;
```

Printf and Scanf:

Printf():

Printf is a predefined function in "stdio.h" header file, by using this function, we can print the data or user defined message on console or monitor. While working with printf(), it can take any number of arguments but first argument must be within the double cotes (" ") and every argument should separated with comma (,) Within the double cotes, whatever we pass, it prints same, if any format specifies are there, then that copy the type of value. The scientific name of the monitor is called console.

Syntax:

```
    printf("user defined message");
```

Syntax:

```
prinf("Format specifers",value1,value2,..);
```

Example of printf() function:

```
int a=10;
double d=13.4;
printf("%f%d",d,a);
```

scanf():

scanf() is a predefined function in "stdio.h" header file. It can be used to read the input value from the keyword.

Syntax:

```
scanf("format specifiers",&value1,&value2,.....);
```

Format specifier	Type of value
%d	Integer
%f	Float
%lf	Double
%c	Single character
%s	String
%u	Unsigned int
%ld	Long int
%lf	Long double

Example of scanf function:

```
int a;
float b;
scanf("%d%f",&a,&b);
```

In the above syntax format specifier is a special character in the C language used to specify the data type of value.

Format specifier:

The address list represents the address of variables in which the value will be stored.

Example:

```
int a;
float b;
scanf("%d%f",&a,&b);
```

In the above example scanf() is able to read two input values (both int and float value) and those are stored in a and b variable respectively.

Syntax :

```
double d=17.8;
char c;
longint l;
scanf("%c%lf%ld",&c&d&l);
```

1) Unformatted consol input output operations

These input / output operations are in unformatted mode. The following are operations of unformatted consol input / output operations:

cin

It is the method to take input any variable / character / string.

Syntax:

cin>>variable / character / String / ;

Example:

```
#include<iostream>
Usingnamespacestd;

int main()
{
intnum;
charch;
stringstr;

cout<<"Enter Number"<<endl;
cin>>num; //Inputs a variable;
cout<<"Enter Character"<<endl;
cin>>ch; //Inputs a character;
cout<<"Enter String"<<endl;
cin>>str; //Inputs a string;

return 0;
}
```

Output

```
Enter Number
07
Enter Character
h
Enter String
Deepak
```

cout

This method is used to print variable / string / character.

Syntax:

cout<< variable / charcter / string;

Example:

```
#include<iostream>
usingnamespacestd;

int main()
{
intnum=100;
charch='X';
stringstr="Deepak";

cout<<"Number is "<<num<<endl; //Prints value of variable;
cout<<"Character is "<<ch<<endl; //Prints character;
cout<<"String is "<<str<<endl; //Prints string;

return 0;
}
```

Output

```
Number is 100
Character is X
String is Deepak
```

2) Formatted console input output operations

In formatted console input output operations we uses following functions to make output in perfect alignment. In industrial programming all the output should be perfectly formatted due to this reason C++ provides many function to convert any file into perfect aligned format. These functions are available in

header file <iomanip>. iomanip refers input output manipulations.

Syntax:

Set width

cout<<setw(int n);

Example:

```
#include<iostream>
#include<iomanip>
usingnamespacestd;

int main()
{
        int x=10;
        cout<<setw(20)<<variable;

        return 0;
}
```

Output

```
          10
```

fill(char)

This function is used to fill specified character at unused space.

Syntax:

cout<<setfill('character')<<variable;

Example:

```
#include<iostream>
#include<iomanip>
usingnamespacestd;
```

```
int main()
{
	int x=10;
	cout<<setw(20);
	cout<<setfill('#')<<x;

	return 0;
}
```

Output

```
##################10
```

precison(n)

This method is used for setting floating point of the output.

Syntax:

```
cout<<setprecision('int n')<<variable;
```

Example:

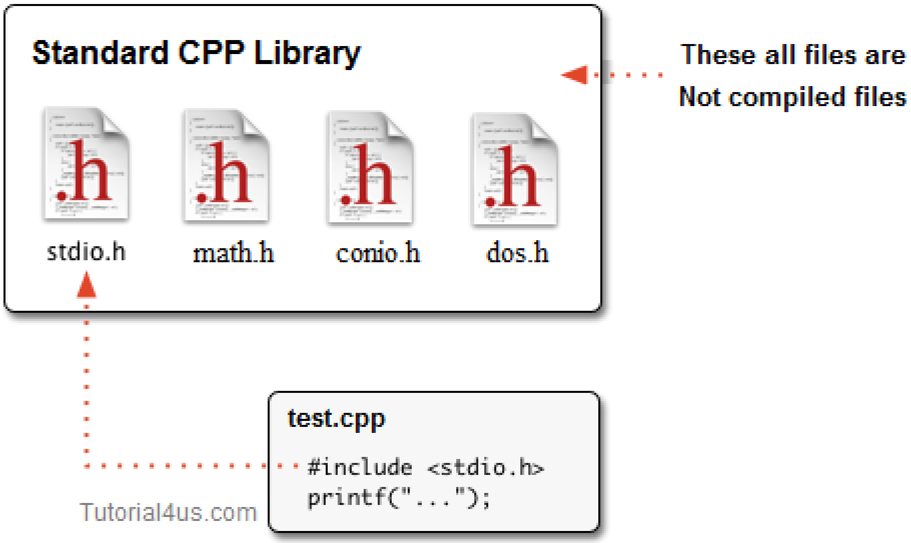
```
#include<iostream>
#include<iomanip>
usingnamespacestd;

int main()
{
	float x=10.12345;
	cout<<setprecision(5)<<x;

	return 0;
}
```

Output

```
10.123
```

Header Files in C++

Header files contain definitions of **Functions and Variables**, which is imported or used into any C++ program by using the pre-processor #include statement. Header file have an extension ".h" which contains C++ function declaration and macro definition.



Each header file contains information (or declarations) for a particular group of functions. Like **stdio.h** header file contains declarations of standard input and output functions available in C++ which is used for get the input and print the output. Similarly, the header file **math.h** contains declarations of mathematical functions available in C++.

Types of Header files

- **System header files:** It is comes with compiler.
- **User header files:** It is written by programmer.

Header File	Function and Description
<iostream>	This file defines the cin, cout, objects, which correspond to the standard input stream, the standard output stream.

Using in C-Language

<stdio>	This files defines the printf(), scanf() functions, which correspond to the standard input, the standard output
<conio>	This header declares several useful library functions for performing "console input and output" from a program like clrscr(),getch() functions.

Why need of header files

When we want to use any function in our C++ program then first we need to import their definition from C++ library, for importing their declaration and definition we need to include header file in program by using #include. Header file include at the top of any C++ program.

For example if we use clrscr() in C++ program, then we need to include, conio.h header file, because in conio.h header file definition of clrscr() (for clear screen) is written in conio.h header file.

Syntax

```
#include<conio.h>
```

See another simple example why use header files

Syntax

```
#include<iostream>

int main()
{
usingnamespacestd;
cout<<"Hello, world!"<<endl;
return0;
}
```

In above program print message on scree hello world! by using cout but we don't define cout here actually already cout has been declared in a header file called **iostream**.

How to use header file in Program

Both user and system header files are include using the pre-processing directive #include. It has following two forms:

Syntax

```
#include<file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directives.

Syntax

```
#include"file"
```

This form used for header files of our own program. It searches for a file named file in the directive containing the current file.

continue Statements

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

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

Syntax

The syntax of a continue statement in C++ is –

```
continue;
```

Example

```
#include<iostream>
usingnamespacestd;

int main ()
{
// Local variable declaration:
int a =10;
// do loop execution
Do
{
if( a ==15)
{
// skip the iteration.
      a = a +1;
continue;
}
cout<<"value of a: "<< a <<endl;
    a = a +1;
}
while( a <20);
return0;
}
```

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

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

break

The **break** statement has the following two usages in C++ –

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

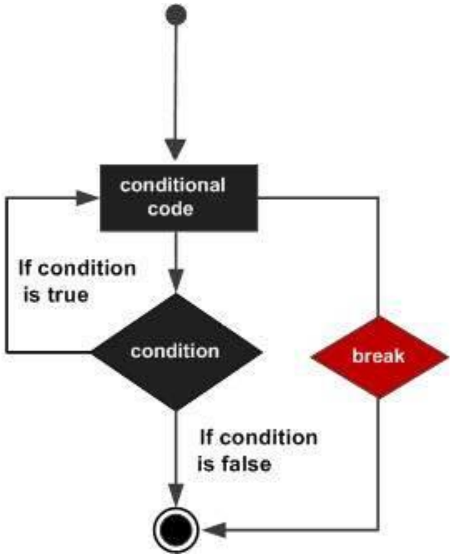
If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax of a break statement in C++ is –

```
break;
```

Flow Diagram



Example

```
#include<iostream>

usingnamespacestd;
```

```
int main (){
// Local variable declaration:
int a =10;
// do loop execution
do
{
cout<<"value of a: "<< a <<endl;
    a = a +1;
if( a >15)
{
// terminate the loop
break;
}
}
while( a <20);
return0;
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

Character I/O (getc, getchar, putc, putchar)

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

In this tutorial, we will learn about such functions, which can be used in our program to take input from user and to output the result on screen.

All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.

---

scanf() and printf() functions

The standard input-output header file, named stdio.h contains the definition of the functions printf()and scanf(), which are used to display output on screen and to take input from user

```
#include<stdio.h>

voidmain()
{
// defining a variable
int i;
/*
displaying message on the screen
asking the user to input a value
   */
printf("Please enter a value...");
/*
reading the value entered by the user
   */
scanf("%d",&i);
/*
displaying the number as output
   */
printf("\nYou entered: %d", i);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

You must be wondering what is the purpose of %d inside the scanf() or printf() functions. It is known as **format string** and this informs the scanf() function, what type of input to expect and inprintf() it is used to give a heads up to the compiler, what type of output to expect.

Format String	**Meaning**
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

We can also **limit the number of digits or characters** that can be input or output, by adding a number with the format string specifier, like "%1d" or "%3s", the first one means a single numeric digit and the second one means 3 characters, hence if you try to input 42, while scanf() has "%1d", it will take only 4 as input. Same is the case for output.

In C Language, computer monitor, printer etc output devices are treated as files and the same process is followed to write output to these devices as would have been followed to write the output to a file.

NOTE : printf() function returns the number of characters printed by it, and scanf() returns the number of characters read by it.

```
int i =printf("studytonight");
```

In this program printf("studytonight"); will return 12 as result, which will be stored in the variablei, because studytonight has 12 characters.

getchar() & putchar() functions

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The putchar() function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use putchar() method in a loop.

```
#include <stdio.h>

voidmain()
{
int c;
printf("Enter a character");
/*
      Take a character as input and
store it in variable c
   */
   c =getchar();
/*
display the character stored
in variable c
   */
putchar(c);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

---

gets() & puts() functions

The gets() function reads a line from **stdin**(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs. The puts() function writes the string strand a trailing newline to **stdout**.

str → This is the pointer to an array of chars where the C string is stored. (Ignore if you are not able to understand this now.)

```
#include<stdio.h>

voidmain()
{
/* character array of length 100 */
charstr[100];
printf("Enter a string");
gets(str);
puts(str);
getch();
}
```

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Difference between scanf() and gets()

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.

If you enter name as **Study Tonight** using scanf() it will only read and store **Study** and will leave the part after space. But gets() function will read it completely.

KARPAGAM ACADEMY OF HIGHER EDUCATION

COIMBATORE - 21

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

CLASS : I B.Sc COMPUTER SCIENCE

BATCH : 2019-2022

Part -A Online Examinations **(1 mark questions)**

SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++ **SUBJECT CODE: 19CSU101**

UNIT-I

S.No	Questions	OPT1	OPT2	OPT3	OPT4	Answer
1	The decomposition of a problem into a number of entities called___________	objects	classes	methods	messages	objects
2	OOPS follows______________ approach in program design	bottom-up	top-down	middle	top	bottom-up
3	Objects take up ______________in the memory	space	address	memory	bytes	space
4	_________________is a collection of objects of similar type	Objects	methods	classes	messages	classes
5	We can create ____________of objects belonging to that class	1	2	10	any number	any number
6	The wrapping up of data & function into a single unit is known as _______________	Polymorphism	encapsulation	functions	data members	encapsulation
7	__________________refers to the act of representing essential features without including the background details or explanations	encapsulation	inheritance	Dynamic binding	Abstraction	Abstraction

8	Attributes are sometimes called_______________	data members	methods	messages	functions	data members
9	The functions operate on the datas are called_______________	methods	data members	messages	classes	methods
10	_______________is the process by which objects of one class acquire the properties of objects of another class	polymorphism	encapsulation	data binding	Inheritance	Inheritance
11	___________________means the ability to take more than one form	polymorphism	encapsulation	data binding	none	polymorphism
12	The process of making an operator to exhibit different behaviors in different instances is known as _________________	function overloading	operator overloading	method overloading	message overloading	operator overloading
13	Single function name can be used to handle different types of tasks is known as ____________	function overloading	operator overloading	polymorphism	encapsulation	operator overloading
14	________________means that the code associated with a given procedure call is not known until the time of the call	late binding	Dynamic binding	Static binding	none	Dynamic binding
15	Objects can be____________	created	created & destroyed	permanent	temporary	created & destroyed
16	_______________helps the programmer to build secure programs	Dynamic binding	Data hiding	Data building	message passing	Data hiding
17	__________________techniques for communication between objects makes the interface descriptions with external systems much simpler	message passing	Data binding	Encapsulation	Data passing	message passing
18	Variables are declared in__________________	only in main()	anywhere in the scope	before the main() only	only at the beginning	anywhere in the scope
19	How many sections in C++?	2	4	1	5	4

20	____________________refers to permit initialization of the variables at run time	Dynamic initialization	Dynamic binding	Data binding	Dynamic message	Dynamic initialization
21	_____________________provides an alias for a previously defined variable	static variable	Dynamic variable	reference variable	address of an variable	reference variable
22	Reference variable must be initialized at the time of _____________	declaration	assigning	initialization	running	declaration
23	The ___________________is an exit-controlled loop	while	do-while	for	switch	do-while
24	The ________________is an entry-entrolled loop	while	do-while	for	switch	for
25	____________________is an entry-controlled one	while	do-while	for	switch	while
26	Error checking does not occur during compilation if we are using_______________	functions	macros	pre-defined functions	operators	macros
27	____________________is a function that is expanded in line when it is invoked	macros	inline function	predefined function	preprocessor macros	inline function
28	________________refers to the use of same thing for different purposes	overloading	Dynamic binding	message loading	none	overloading
29	_________________are extensively used for handling class objects	overloaded functions	methods	objects	messages	overloaded functions
30	____________________is used to reduce the number of functions to be defined	default arguments	methods	objects	classes	default arguments
31	Control structures are said to be_______________	programs	structured programs	statements	case statements	structured programs

32	________________________is a decision making statement	for	jump	break	if	if
33	The bool type data occupies ___________byte in memory	two	one	three	four	one
34	if-else-if ladder sometimes called________________	if-else-if nested	nested-if-else-if	if-else-if-staircase	if-else-if	if-else-if-staircase
35	How many statements are used to perform an unconditional transfer?	2	3	4	5	4
36	The label must start with___________	character	__	number	alphanumeric	character
37	________________statement is frequently used to terminate the loop in the switch case()	jump	goto	continue	break	break
38	______________statement does not require any condition	for	if	goto	while	goto
39	____________statement is used to transfer the control t pass on t the beginning of the block/loop	break	jump	goto	continue	continue
40	________________statement is a multiway branch statement	for	switch	if	while	switch
41	Every case statement in switch case statement terminates with	;	:	,	>>	:
42	How many types of loop control structure exist in c++?	1	3	2	4	3
43	The expression are separated by ____________in the for loop	:	;	,	++	;

44	Test is performed at the ____________of the for loop.	top	middle	end	program terminates	top
45	Condition is checked at the ____________of the loop in the do-while statement.	beginning	end	middle	program terminates	end
46	Every expression always return____________	0 or 1	1 or 2	-1 or 0	none	0 or 1
47	Which of the following loop statement uses 2 keyword?	do-while loop	for loop	if loop	while loop	do-while loop
48	The meaning of if(1) is________________	always false	always true	both(a) & (b)	none	always true
49	The for loop comprises of ______________actions	2	3	1	4	3
50	_____________statement present at the bottom of the switch case statements	default	case	label	none	default
51	__________________is an assignment statement that is used to set the loop control variables	Increment	declaring	Initialization	decrement	Initialization
52	Which of the following control expressions are valid for an of statement ?	an integer expression	a Boolean expression	either A or B	Neither A nor B	a Boolean expression
53	If the data is received from the input devices in sequence then it is called________.	Source stream	Object stream	Destination stream	Input stream.	Source stream
54	When the data is passed to the output devices it is called_____	Source stream	Object stream	Destination stream	Input stream.	Destination stream
55	The C++ have a number of stream classes that are used to work with _________ operations.	Console I/O	Console and file	formatted console	unformatted console	Console and file

56	The data accepted with default setting by I/O function of the language is known as-----	Formatted	Unformatted	Argumented	files	Unformatted
57	_________ is used as the input stream to read data.	Cout	Printf	Cin	Scanf.	Cin
58	cin and cout are ________ for input and output of data.	user defined stream	system defined stream	Pre defined stream	stream	Pre defined stream
59	.The data obtained or represented with some manipulators are called ______.	formatted data	unformatted data	extracted data	None.	formatted data
60	The output formats can be controlled with manipulators having the header file as	iostream.h	conio.h	stdlib.h	iomanip.h	iomanip.h
61	The _____ and ______ are derived classes from ios based class.	istream and ostream	source and destination stream	iostream and source stream	None.	istream and ostream
62	The manipulator << endl is equivalent to____	‘\t’	’\r’	’\n’	’\b’	’\n’
63	While loop is _______ statement.	entry controlled	exit controlled	branching	none	entry controlled
64	Do.. While loop is _____ statement.	entry controlled	exit controlled	branching	none	exit controlled
65	For loop is an _______ statement.	entry controlled	exit controlled	branching	none	entry controlled
66	______ statement causes loop to be continued with next	continue	goto	break	exit	continue

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2019 onwards)
DEPARTMENT OF CS, CA& IT

SUBJECT :PROGRAMMING FUNDAMENTALS USING C / C++ **Class : I B.Sc (CS) B**
SEMESTER: I **SUBJECT CODE: 19CSU101**

UNIT-II

Functions and Arrays:

FUNCTIONS

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.

ARRAYS :

Creating and Using One Dimensional Arrays

(Declaring and Defining an Array,

Initializing an Array,

Accessing,

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.

Functions In C++

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

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is 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, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

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

Defining a Function

The general form of a C++ function definition is as follows −

```
return_typefunction_name( parameter list )
 {
body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

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

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// 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_typefunction_name( parameter list );
```

For the above defined function max(), following is the function declaration −

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

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

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

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

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

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

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

```
#include<iostream>
usingnamespace std;

// function declaration
```

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

int main ()
{
// local variable declaration:
int a =100;
int b =200;
int ret;

// calling a function to get max value.
ret=max(a, b); // a,b are Actual Argument
cout<<"Max value is : "<< ret <<endl;

return0;
}

// function returning the max between two numbers
intmax(int num1,int num2) // num1 & num 2 are formals
{
// local variable declaration
int result;

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

return result;
}
```

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

```
Max value is : 200
```

Function Arguments

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

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function –

Sr.No	Call Type & Description
1	**Call by Value** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	**Call by Pointer**

	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.
3	**Call by Reference** This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

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

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

```
#include<iostream>
usingnamespace std;

intsum(inta,int b =20)
{
int result;
result= a + b;

return(result);
}
int main ()
{
// local variable declaration:
int a =100;
int b =200;
int result;

// calling a function to add the values.
result=sum(a, b);
cout<<"Total value is :"<< result <<endl;

// calling a function again as follows.
result= sum(a);
cout<<"Total value is :"<< result <<endl;

return0;
}
```

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

```
Total value is :300
Total value is :120
```

Call By Value

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

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

```
// function definition to swap the values.
voidswap(intx,int y){
int temp;

temp= x;/* save the value of x */
  x = y;/* put y into x */
  y = temp;/* put x into y */

return;
}
```

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

```
#include<iostream>
usingnamespace std;

// function declaration
voidswap(intx,int y);

int main (){
// local variable declaration:
int a =100;
int b =200;

cout<<"Before swap, value of a :"<< a <<endl;
cout<<"Before swap, value of b :"<< b <<endl;

// calling a function to swap the values.

swap(a, b);

cout<<"After swap, value of a :"<< a <<endl;
cout<<"After swap, value of b :"<< b <<endl;

return0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result –

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

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

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

Call By Reference

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

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

```
// function definition to swap the values.

void swap(int&x,int&y)

{

int temp;

temp= x;/* save the value at address x */

  x = y;/* put y into x */

  y = temp;/* put x into y */



return;

}
```

For now, let us call the function **swap()** by passing values by reference as in the following example –

```
#include<iostream>
usingnamespace std;

// function declaration
void swap(int&x,int&y);
int main ()
{
// local variable declaration:
int a =100;
int b =200;

cout<<"Before swap, value of a :"<< a <<endl;
cout<<"Before swap, value of b :"<< b <<endl;

/* calling a function to swap the values using variable reference.*/
swap(a, b);

cout<<"After swap, value of a :"<< a <<endl;
cout<<"After swap, value of b :"<< b <<endl;

return0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result –

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

Command line arguments in C/C++

It is possible to pass some values from the command line to your C or C++ programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main()

{ /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.

To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(intargc, char *argv[]) { /* ... */ }
```

or

```
int main(intargc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.

- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

For better understanding run this code on your linux machine.

```
// Name of program mainreturn.cpp

#include <iostream>
usingnamespacestd;

intmain(intargc, char** argv)
{
  cout<< "You have entered "<<argc
     << " arguments:"<< "\n";

  for(inti = 0; i<argc; ++i)
    cout<<argv[i] << "\n";

  return0;
}
```

Run on IDE

Input:

```
$ g++ mainreturn.cpp -o main

$ ./main geeks for geeks
```

Output:

```
You have entered 4 arguments:

./main

geeks

for

geeks
```

Note : Other platform-dependent formats are also allowed by the C and C++ standards; for example, Unix (though not POSIX.1) and Microsoft Visual C++ have a third argument giving the program's environment, otherwise accessible through getenv in stdlib.h: Refer C program to print environment variables for details.

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.

3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

https://ide.geeksforgeeks.org/index.php---ref

FUNCTION RETURNING VALUE

C++ Value- FUNCTION RETURNING VALUE With void functions, we use the function name as a statement in our program to execute the actions that function performs. With value-returning functions, the actions result in the return of a value, and that value can be used in an expression.

Here is the syntax template of a value-returning function prototype:

return DataTypefunctionName(DataTypeOfParameterList);

Here is the syntax template of a value-returning function definition:

```
return DataTypefunctionName(DataTypeWithParameterList)
{
            Statement
              .
              .
              .
            return value-returning-expression;
}
```

With void functions, we use the function name as a statement in our program to execute the actions that function performs. With value-returning functions, the actions result in the return of a value, and that value can be used in an expression. For example, let's write a function that returns the smallest of three input values.

Example

```
// Program PrintMin prints the smallest of three input values.
#include <iostream>
using namespace std;

int  Minimum(int, int, int);          // function prototype
// Post: Minimum returns the minimum of three distinct values.
```

```
int main()

{

int  one, two, three;

int  MIN;

cout<< "Input three integer values. Press return." <<endl;

cin>> one  >> two  >> three;

   MIN = Minimum(one, two, three); // function call

cout<< "The minimum value of the three numbers is " << MIN <<endl;

   return 0;

}

//*******************************************************

//Function definition with 3 parameters

//*******************************************************


int  Minimum(int first, int second, int third)

// Post: Returns minimum of three distinct int values.

{

   if (first <= second && first < third)

                    return first;

   else if (second <= first && second < third)

                    return second;

   else

                    return third;

}
```

The function prototype declares a function of data type int. This means that the value returned to the calling function is of type int. Because a value-returning function always sends one value back to the calling function, we designate the type of the value before the name of the function. We call this data type the function return data type or function type.

In the above example, the function invocation/call occurs in the output statement of the main function. The Minimum function is invoked/called and the returned value is immediately sent to the output stream. There are three parameters first, second and third. The three arguments sent from main are one, two and three. Note: in the function prototype, only the datatypes of the parameter list are needed. In the function heading (or function definition), however, both the parameters and their datatype are necessary.

Return from void functions in C++

Void functions are “void” due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions. Some of cases are listed below.

Inline function

C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Following is an example, which makes use of inline function to return max of two numbers –

Live Demo

```
#include <iostream>

using namespace std;

inline int Max(int x, int y)
 {
  return (x > y)? x : y;
}

// Main function for the program
int main()
{
cout<< "Max (20,10): " <<Max(20,10) <<endl;


cout<< "Max (0,200): " <<Max(0,200) <<endl;
cout<< "Max (100,1010): " <<Max(100,1010) <<endl;

  return 0;
}
```

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

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

DIFFERENTIATING BETWEEN DECLARATION AND DEFINITION OF FUNCTIONS

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.

Function Types

- Function with no argument and no return value
- Function with no argument but return value
- Function with argument but no return value
- Function with argument and return value

Defining a Function

The general form of a C++ function definition is as follows −

```
return_typefunction_name( parameter list )
 {
body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

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

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// 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_typefunction_name( parameter list );
```

For the above defined function max(), following is the function declaration −

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

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

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

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

A *function declaration* declares the identifier (i. e. the name) of the function. If the function declaration specifies they types of the function's parameters, that is a *function prototype.* Modern C and C++ do not allow function declarations that do not specify the types of the parameters.

A *function definition* is a function declaration that provides the code for the function itself.

Examples:

```
intsqr(int n);// declaration

intsqr(int n)
        {
        return n*n;// definition
}
```

A *definition* actually sets aside space for whatever the thing is that’s being defined. For a function, this includes the function body, which says what a function will do when its called. For the above declaration, the following is one potential definition/implementation:

```
intstrlen(char*s)
{
intlen=0;
while(s[len]!='\0')
++len;
returnlen;
}
```

Clearly, a definition is also a declaration. If you give a definition for a function, the compiler has all it needs to compile any subsequent uses of the function. But a declaration may not be a definition .Example:

```
void sum(int x, int y)
{
 int z;
 z = x + y;
cout<< z;
}

int main()
{
 int a = 10;
 int b = 20;
sum (a, b);
}
```

Here, **a** and **b** are sent as arguments, and **x** and **y** are parameters which will hold values of a and b to

perform required operation inside function.

- **Function body :** is he part where the code statements are written.

Declaring, Defining and Calling Function

Function declaration, is done to tell the compiler about the existence of the function. Function's return type, its name & parameter list is mentioned. Function body is written in its definition. Lets understand this with help of an example.

```
#include < iostream>
using namespace std;
int sum (int x, int y);   //declaring function
int main()
{
 int a = 10;
 int b = 20;
 int c = sum (a, b);   //calling function
cout<< c;
}
```

```
int sum (int x, int y)   //defining function
{
 return (X + y);
}
```

Here, initially the function is **declared**, without body. Then inside main() function it is **called**, as the function returns sumation of two values, hence z is their to store the value of sum. Then, at last, function is **defined**, where the body of function is mentioned. We can also, declare & define the function together, but then it should be done before it is called.

Function-parameters

Formal Parameters

Formal parameters are written in the function prototype and function header of the definition. Formal parameters are local variables which are assigned values from the arguments when the function is called.

Actual Parameters or Arguments

When a function is *called*, the values (expressions) that are passed in the call are called the *arguments* or *actual parameters* (both terms mean the same thing). At the time of the call each actual parameter is assigned to the corresponding formal parameter in the function definition.

For value parameters (the default), the value of the actual parameter is assigned to the formal parameter variable. For reference parameters, the *memory address* of the actual parameter is assigned to the formal parameter.

Value Parameters

By default, argument values are simply copied to the formal parameter variables at the time of the call. This type of parameter passing is called *pass-by-value*. It is the only kind of parameter passing in Java and C. C++ also has *pass-by-reference* (see below).

Reference Parameters

A *reference parameter* is indicated by following the formal parameter name in the function prototype/header by an ampersand (&). The compiler will then pass the *memory address* of the actual parameter, not the value. A formal reference parameter may be used as a normal variable, without explicit dereference - the compiler will generate the correct code for using an address. Reference parameters are useful in two cases:

- To *change* the value of actual parameter variables.
- To more efficiently pass large structures.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example –

```
#include<iostream>
usingnamespace std;

intsum(inta,int b =20)
{
int result;
result= a + b;

return(result);
}
int main ()
{
// local variable declaration:
int a =100;
int b =200;
int result;

// calling a function to add the values.
result=sum(a, b);
cout<<"Total value is :"<< result <<endl;

// calling a function again as follows.
result= sum(a);
cout<<"Total value is :"<< result <<endl;

return0;
}
```

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

```
Total value is :300
Total value is :120
```

Array

What is array?

An array is a collection of similar data-types & they are referenced by a common name.

[OR]

An array is a collection of data that holds fixed number of values of same type.

The Various types of Array those are provided by c as Follows:-

1. Single Dimensional Array
2. Two Dimensional Array
3. Three Dimensional array(Multi)

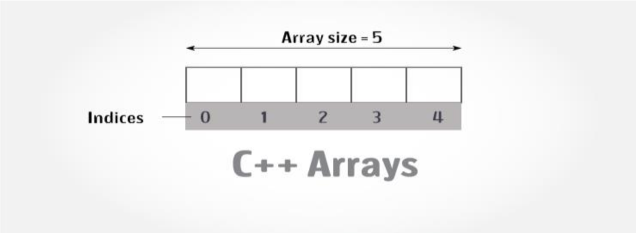
Declaring Arrays

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

```
int  age[100];
```

Here, the age array can hold maximum of 100 elements of integer type.

The size and type of arrays cannot be changed after its declaration.



How to declare an array in C++?

```
dataTypearrayName[arraySize];
```

For example,

```
float mark[5];
```

Here, we declared an array, *mark*, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

Elements of an Array and accessing

You can access elements of an array by using indices.Suppose you declared an array *mark* as above.

The first element is *mark[0]*,

second element is *mark[1]* and so on.

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

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

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

```
#include<iostream>

usingnamespace std;

#include<iomanip>

usingstd::setw;
```

```
int main ( )

{

intn[10];// n is an array of 10 integers

// initialize elements of array n to 0

for(inti=0;i<10;i++)

{

n[i]=i+100;// set element at location i to i + 100

}

cout<<"Element"<<setw(13)<<"Value"<<endl;


// output each array element's value

for(int j =0; j <10;j++)

{

cout<<setw(7)<< j <<setw(13)<< n[ j ]<<endl;

}

return0;

}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result –

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

Few key notes:

- Arrays have 0 as the first index not 1. In this example, *mark[0]* is the first element.
- If the size of an array is *n*, to access the last element, (n-1) index is used. In this example, *mark[4]* is the last element.
- Suppose the starting address of mark[0] is 2120d. Then, the next address, a[1], will be 2124d, address of a[2] will be 2128d and so on. It's because the size of a float is 4 bytes.

Initialize an array

It's possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

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

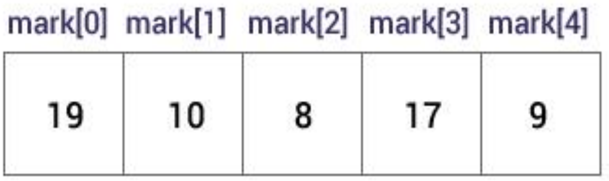
The number of values between braces { }can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

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

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

Another method to initialize array during declaration:

```
int mark[  ] = {19, 10, 8, 17, 9};
```



mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

Here,

```
mark[0] is equal to 19
mark[1] is equal to 10
mark[2] is equal to 8
mark[3] is equal to 17
mark[4] is equal to 9
```

One dimensional Array

A dimensional is used representing the elements of the array for example

int a[5]

The [] is used for dimensional or the sub-script of the array that is generally used for declaring the elements of the array For Accessing the Element from the array we can use the Subscript of the Array like this

a[3]=100

This will set the value of 4th element of array

So there is only the single bracket then it called the Single Dimensional Array

This is also called as the Single Dimensional Array

Example: C++ Array One dimensional Array

C++ program to store and calculate the sum of 5 numbers entered by the user using arrays.

```
#include<iostream>
usingnamespace std;
intmain()
{
int  numbers[5], sum =0;
cout<<"Enter 5 numbers: ";

// Storing 5 number entered by user in an array
// Finding the sum of numbers entered
for(inti=0;i<5;++i)
{
cin>> numbers[i];
      sum += numbers[i];
}
cout<<"Sum = "<< sum <<endl;
return0;
}
```

Output

```
Enter 5 numbers:
 3
4
5
4
2
Sum = 18
```

Two Dimensional Array

It is a collection of data elements of same data type arranged in rows and columns (that is, in two dimensions).

The Two Dimensional array is used for representing the elements of the array in the form of the rows and columns and these are used for representing the Matrix A Two Dimensional Array uses the two subscripts for declaring the elements of the Array

Like this int a[3][3]

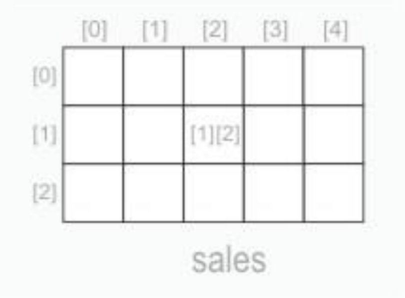
So This is the Example of the Two Dimensional Array In this first 3 represents the total number of Rows and the Second Elements Represents the Total number of Columns The Total Number of elements are judge by Multiplying the Numbers of Rows * Number of Columns in The Array in the above array the Total Number of elements are 9

Declaration of Two-Dimensional Array

Type arrayName[numberOfRows][numberOfColumn];

For example,

int Sales[3][5];



Initialization of Two-Dimensional Array

An two-dimensional array can be initialized along with declaration. For two-dimensional array initialization, elements of each row are enclosed within curly braces and separated by commas. All rows are enclosed within curly braces.

Here is the general form of a Multi dimensional array declaration −

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

Here is the general form of a Three (Multi) array declaration −

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

```
int A[4][3] = { {22, 23, 10},
          {15, 25, 13},
          {20, 74, 67},
          {11, 18, 14} };
```

Referring to Array Elements

To access the elements of a two-dimensional array, we need a pair of indices: one for the row position and one for the column position. The format is as simple as: name[rowIndex][columnIndex]

Examples:

```
cout<<A[1][2];    //print an array element
A[1][2]=13;       // assign value to an array element
cin>>A[1][2];     //input element
```

Using Loop to input an Two-Dimensional Array from user

```
int mat[3][5], row, col ;
for (row = 0; row < 3; row++)
for (col = 0; col < 5; col++)

cin>> mat[row][col];
```

Example 1:

Two Dimensional Array

C++ Program to display all elements of an initialisedtwo dimensional array.

```
#include<iostream>
usingnamespace std;

intmain( )
{


inttest[3][2]={
{2,-5},
{4,0},
{9,1}
};

// Accessing two dimensional array using
// nested for loops
for(inti=0;i<3;++i)
{
for(int j =0; j <2;++j)
{
cou t<<"test["<<i<<"]["<< j <<"] = "<< test[i][j]<<endl;
}
}

return0;
}
```

Example : 2

In this program, user is asked to entered the number of rows *r* and columns *c*. The value of *r* and *c* should be less than 100 in this program.The user is asked to enter elements of two matrices (of order r*c).Then, the program adds these two matrices, saves it in another matrix (two-dimensional array) and displays it on the screen.

Example: Add Two Matrices using Multi-dimensional Arrays

```
#include<iostream>
usingnamespace std;

intmain()
{
Int  r, c, a[100][100], b[100][100], sum[100][100],i, j;

cout<<"Enter number of rows (between 1 and 100): ";
cin>> r;

cout<<"Enter number of columns (between 1 and 100): ";
cin>> c;

cout<<endl<<"Enter elements of 1st matrix: "<<endl;

// Storing elements of first matrix entered by user.
for(i=0;i< r;++i)
for(j =0; j < c;++j)
{
cout<<"Enter element a"<<i+1<< j +1<<" : ";
cin>> a[i][j];
}

// Storing elements of second matrix entered by user.
cout<<endl<<"Enter elements of 2nd matrix: "<<endl;
for(i=0;i< r;++i)
for(j =0; j < c;++j)
{
cout<<"Enter element b"<<i+1<< j +1<<" : ";
cin>> b[i][j];
}
// Adding Two matrices
for(i=0;i< r;++i)
for(j =0; j < c;++j)
        sum[i][j]= a[i][j]+ b[i][j];

// Displaying the resultant sum matrix.
cout<<endl<<"Sum of two matrix is: "<<endl;
for(i=0;i< r;++i)
for(j =0; j < c;++j)
{
cout<< sum[i][j]<<"  ";
if(j == c -1)
cout<<endl;
}

return0;
}
```

Multi dimensional

The elements of an array can be of any data type, including arrays! An array of arrays is called a **multidimensional array**.

The Multidimensional Array are used for Representing the Total Number of Tables of Matrix A Three dimensional Array is used when we wants to make the two or more tables of the Matrix Elements for Declaring the Array Elements we can use the way like this

int a[3][3][3]

In this first 3 represents the total number of Tables and the second 3 represents the total number of rows in the each table and the third 3 represents the total number of Columns in the Tables

So this makes the 3 Tables having the three rows and the three columns

The Main and very important thing about the array that the elements are stored always in the Contiguous in the memory of the Computer

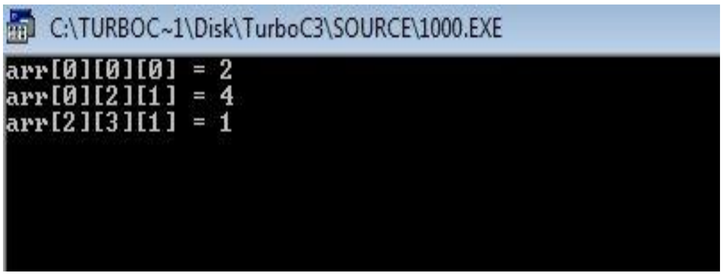
So, to initialize and print three(Multi) dimensional array, you have to use three **for** loops. Third **for** loop (the innermost loop) forms 1D array, second **for** loop forms 2D array and the third **for** loop (the outermost loop) forms 3D array, as shown here in the following program.

C++ Programming Code for Three(multi) Dimensional (3D) Array

A three dimensional (3D) array can be thought of as **an array of arrays of arrays**.

Following is a simple C++ program to initialize three-dimensional (3D) array of dimensions 3*4*2, then it will access some elements present in the array and display the element on the screen :

```
/* C++ Program - Three Dimensional Array Program */
#include<iostream.h>
#include<conio.h>
void main()
{
        int arr[3][4][2] = {
                                    {
                                          {2, 4},
                                          {7, 8},
                                          {3, 4},
                                        {5, 6}
                                },
                                {
                                        {7, 6},
                                        {3, 4},
                                        {5, 3},
                                        {2, 3}
                                },
                                {
                                        {8, 9},
                                        {7, 2},
                                        {3, 4},
                                        {5, 1}
                                }
                              };
        cout<<"arr[0][0][0] = "<<arr[0][0][0]<<"\n";
        cout<<"arr[0][2][1] = "<<arr[0][2][1]<<"\n";
        cout<<"arr[2][3][1] = "<<arr[2][3][1]<<"\n";
        getch();
}
```



```
arr[0][0][0] = 2
arr[0][2][1] = 4
arr[2][3][1] = 1
```

(https://codescracker.com/cpp/program/cpp-program-add-two-matrices.htm)

When the above C++ program is compile and executed, it will produce the following result:

Here, the outer array has three elements, each of which is a 2-D array of four 1-D arrays, each of which contains two integers. It means, a 1-D array of two elements is constructed first. Then four such 1-D arrays are placed one below the other to give a 2-D array containing four rows. Then, three such 2-D arrays are placed one behind the other to yield a 3-D array containing three 2-D arrays.

(http://ecomputernotes.com)

Enable | Enlighten | Enrich
KARPAGAM
ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)

KARPAGAM ACADEMY OF HIGHER EDUCATION

COIMBATORE - 21

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

CLASS : I B.Sc COMPUTER SCIENCE

BATCH : 2019-2022

Part -A Online Examinations **(1 mark questions)**

SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++ **SUBJECT CODE: 19CSU101**

UNIT-II

S.No	Questions	OPT1	OPT2	OPT3	OPT4	Answer
1	A member function can call another member function directly without using the _________ operator	Assignment	equal	dot	greater than	dot
2	A ______ member variable is initialized to zero when the first object of its class is created	Dynamic	constant	static	protected	static
3	_________ Variables are normally used to maintain values common to the entire class.	Private	protected	Public	static	static
4	When a copy of the entire object is passed to the function it is called as _________	Pass by reference	pass by function	pass by pointer	pass by value	pass by value
5	When the address of the object is transferred to the function it is called as _________	pass by reference	pass by function	pass by pointer	pass by value	pass by reference
6	A ________ function can be invoked like a normal function without the help of any object	Void	friend	inline	none of the above	friend

7	The ________ member variables must be defined outside the class.	Static	private	public	protected	Static
8	A friend function, although not a member function, has full access right to the _______ members of the class	Static	private	public	protected	private
9	Function should return a __________.	value	character	both (a) and (b)	none	value
10	________________function is useful when calling function is small	Built-in	Inline	user-defined	none.	Inline
11	c++ propouse a new future called ______________	function overloading	polymorphism	Inline function	calling function	Inline function
12	Which of the following cannot be passed to a function?	reference variables	arrays	class objects	header files	header files
13	Function should return a __________.	value	character	both (a) and (b)	none	value
14	________________function is useful when calling function is small	Built-in	Inline	user-defined	none.	Inline
15	Inline function needs more______________	variables	functions	memoryspace	control structures	memoryspace
16	Multiple function with the same name is known as _________________	function overloading	Encapsulation	inheritance	operator overloading	function overloading
17	The _____________ function creates a new set of variables and copies the values of arguments into them.	calling function	called function	function	function overloading	called function

18	Function contained within a class is called a _____________	built-in	member function	user-defined function	calling function	member function
19	In c++,Declarations can appear________________in the body of the function	Only at the top	middle	bottom	anywhere	anywhere
20	Modular structure of C language enables the program to be spli	structure	union	integers	function	function
21	The actual and formal arguments of functions must match in _	actual argum	formal arg	dummy pa	temporary variab	actual arguments
22	Functions receives the values passed by the calling function an	actual argum	formal arg	dummy pa	temporary variab	formal arguments
23	In looping process first step is _________	intialise cou	test for co	increment	execution statem	intialise counter
24	In case of for loop ______ section is executed before test condi	increment	intialise	testing	execution of state	increment
25	_____ statement is the mechanism for returning value to the cal	return	continue	break	goto	return
26	A function can return ____ value per call	one	zero	two	multiple	one
27	______ is a special case where a function calls itself.	recursion	subroutine	structure	none	recursion
28	___ is a group of related data items that share a common name.	variables	array	function	structure	array

29	A ______ is an array of characters.	string	variables	function	none	string
30	Individual values in array is referred as _______.	subscript	elements	subelement	none	elements
31	Any subscript between _______ are valid for an array of fifty el	0-49	0-56	0-48	0-46	0-49
32	Value in a matrix can be represented by _______ subscript.	1	3	2	4	2
33	arrays that do not have their dimensions explicitly specified are	unsized arra	undimensi	initialized a	to size arrays	unsized arrays
34	In ASCII character set the uppercase alphabet represent codes _	65 to 90	96 to 45	97 to 123	1 to 26	65 to 90
35	Modular structure of C language enables the program to be spli	structure	union	integers	function	function
36	The actual and formal arguments of functions must match in _	actual arguments	formal arguments	dummy parameters	temporary variables	actual arguments
37	Functions receives the values passed by the calling function an	actual arguments	formal arguments	dummy parameters	temporary variables	formal arguments
38	Process of calling a function using pointers to pass address of	call by value	call by reference	call by method	call by address	call by reference
39	The process of passing actual values of variable is known as __	call by value	call by reference	call by method	call by address	call by value

40	In pointers when function is called ______ are passed as actual	values	addresses	operators	none of the above	addresses

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2019 onwards)
DEPARTMENT OF CS,CA & IT

SUBJECT:PROGRAMMING FUNDAMENTALS USING C / C++ **Class : I B. Sc (CS) B**
SEMESTER: I **SUBJECT CODE: 19CSU101**

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

Utility (introduction) of structures

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person:

name1, citNo1, salary1, name2, citNo2, salary2

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

Declaring a structure

The struct keyword defines a structure type followed by an identifier (name of the structure). Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
char name[50];
int age;
float salary;
};
```

Here a structure person is defined which has three members: name, age and salary.

When a structure is created, no memory is allocated.

The structure definition is only the blueprint for the creating of variables. You can imagine it as a datatype. When you define an integer as below:

```
int foo;
```

The int specifies that, variable foo can hold integer element only. Similarly, structure definition only specifies that, what property a structure variable holds when it is defined.

Note: Remember to end the declaration with a semicolon **(;)**

Define a structure variable

Once you declare a structure person as above. You can define a structure variable as:

```
Person bill;
```

Here, a structure variable bill is defined which is of type structure Person.
When structure variable is defined, only then the required memory is allocated by the compiler.

Considering you have either 32-bit or 64-bit system, the memory of float is 4 bytes, memory of int is 4 bytes and memory of char is 1 byte.
Hence, 58 bytes of memory is allocated for structure variable bill.

Access the members of a structure

The members of structure variable is accessed using a **dot (.)** operator.
Suppose, you want to access age of structure variable bill and assign it 50 to it. You can perform this task by using following code below:

```
bill.age = 50;
```

Example: C++ Structure

C++ Program to assign data to members of a structure variable and display it.

```
#include<iostream>
usingnamespace std;

structPerson
{
char name[50];
int age;
float salary;
};
int main()
{
struct Person p1;
cout <<"Enter Full name: ";
cin.get(p1.name, 50);
cout <<"Enter age: ";
cin >> p1.age;
cout <<"Enter salary: ";
cin >> p1.salary;

cout <<"\nDisplaying Information."<< endl;
cout <<"Name: "<< p1.name << endl;
cout <<"Age: "<< p1.age << endl;
cout <<"Salary: "<< p1.salary;

return0;
}
```

Output

```
Enter Full name: Magdalena Dankova
Enter age: 27
Enter salary: 1024.4

Displaying Information.
Name: Magdalena Dankova
Age: 27
Salary: 1024.4
```

Here a structure Person is declared which has three members name, age and salary.
Inside main() function, a structure variable p1 is defined. Then, the user is asked to enter information and data entered by user is displayed.

Passing and returning structures from functions

Passing structure to function

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

A structure variable can be passed to a function in similar way as normal argument. Consider this example:

Example 1: C++ Structure and Function

```
#include<iostream>
usingnamespace std;

structPerson
{
char name[50];
int age;
float salary;
};

void displayData(Person);// Function declaration

int main()
{
structPerson p;

cout <<"Enter Full name: ";
cin.get(p.name,50);
cout <<"Enter age: ";
cin >> p.age;
cout <<"Enter salary: ";
cin >> p.salary;

// Function call with structure variable as an argument
displayData(p);

return0;
}

void displayData(Person p)
{
cout <<"\nDisplaying Information."<< endl;
cout <<"Name: "<< p.name << endl;
cout <<"Age: "<< p.age << endl;
cout <<"Salary: "<< p.salary;
}
```

Output

```
Enter Full name: Bill Jobs
Enter age: 55
Enter salary: 34233.4

Displaying Information.
Name: Bill Jobs
Age: 55
Salary: 34233.4
```

In this program, user is asked to enter the name, age and salary of a Person inside main()function.

Then, the structure variable p is to passed to a function using.

```
displayData(p);
```

The return type of displayData() is void and a single argument of type structure Person is passed.

Then the members of structure p is displayed from this function.

Returning structure from function

Example 2: Returning structure from function

```
#include<iostream>
usingnamespace std;

structPerson
{
char name[50];
int age;
float salary;
};

Person getData(Person);
void displayData(Person);

int main()
{

Person p;

p = getData(p);
displayData(p);

return0;
}

Person getData(Person p)
{

cout <<"Enter Full name: ";
cin.get(p.name,50);

cout <<"Enter age: ";
cin >> p.age;

cout <<"Enter salary: ";
cin >> p.salary;

return p;
}

void displayData(Person p)
{
cout <<"\nDisplaying Information."<< endl;
cout <<"Name: "<< p.name << endl;
cout <<"Age: "<< p.age << endl;
cout <<"Salary: "<< p.salary;
}In this program, the structure variable p of type structure Person is defined under main()function.
```

The structure variable p is passed to getData() function which takes input from user which is then returned to main function.p = getData(p);

Note: The value of all members of a structure variable can be assigned to another structure using assignment operator = if both structure variables are of same type. You don't need to manually assign each members.

Then the structure variable p is passed to displayData() function, which displays the information.

Manipulating individual members of structures

A function that would change the values of the members of a struct type variable. However, it produces an unexpected output and I can't figure out why this is happening. /*Program to test struct types*/

```
#include<iostream>
#include<cstring>
usingnamespace std;
struct myStruct
{
string a;
string b;
int c;
float d;
};

void assignValues(myStruct myobj)
{
Strcpy(myobj.a,”foobar”);
Strcpy(myobj.a,”Foobar”);
myobj.c =12;
myobj.d =15.223;
return (myobj);
}
int main()
{
myStruct x;
mystruct asignvalues(mystruct);
cout << x.a <<endl;
x=assignValues(x);
cout << x.a<<endl;
cout << x.b << endl;
cout << x.c << endl;
cout << x.d << endl;
}
```

Individual data members as structures

A structure variable has been defined, its member can be accessed through the use of dot (.) operator. For example, the following code fragment assigns 1740 to year element of birth_date structure variable declared earlier:

birth_date.year = 1740 ;

Syntax to Access Structure Member in C++

The structure variable name followed by a period or dot (.) and the element name references to that individual structure element. The syntax to access structure element is shown here:

structure-name.element-name

Remember that the first component or element of an expression involving the dot (.) operator is the name of specific structure variable (birth_date in this case), not the name of structure specifier (date).

The structure members are treated just like other variables. So, to print year of birth_date, you can simply write:

```
cout << birth_date.year ;
```

In same fashion, to read day, month and year of joining_date, you can simply write :

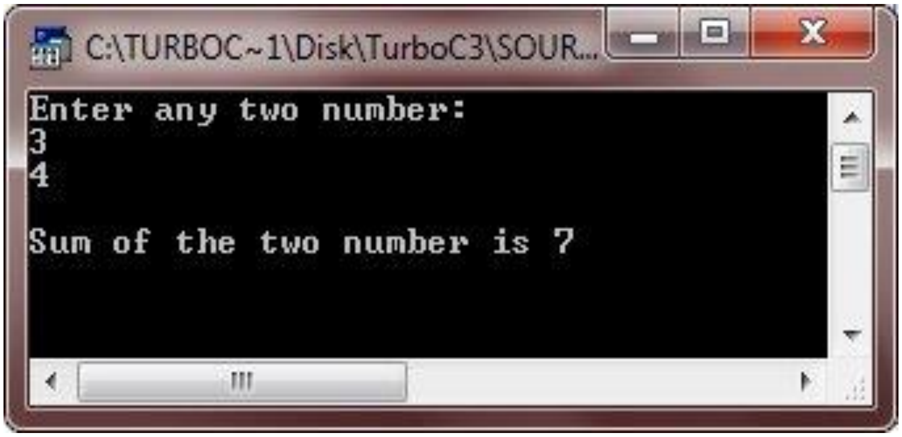
```
cin >> joining_date.day >> joining_date.month >> joining_date.year ;
```

C++ Access Structure Members Example

Here is an example, demonstrating how to access members of a structure in C++

```
/* C++ Access Structure Member */

#include<iostream.h>
#include<conio.h>

struct st
{
        int a;    // structure member
        int b;    // structure member
        int sum;  // structure member
} st_var;              // structure variable

void main()
{
        clrscr();
        cout<<"Enter any two number:\n";
        // accessing structure member a and b
        cin>>st_var.a>>st_var.b;
        // accessing structure member sum, a, and b
        st_var.sum = st_var.a + st_var.b;
        // accessing structure member sum
        cout<<"\nSum of the two number is "<<st_var.sum;
        getch();
}
```

Here is the sample run of this C++ program:



```
Enter any two number:
3
4

Sum of the two number is 7
```

Array Of Structure

The structure and the array both are C++ derived types. While arrays are collections of analogous elements, structures assemble elements under one roof. Thus both the array and the structure allow several values to be treated together as a single data object.

The arrays and structures can be combined together to form complex data objects. There may be structures contained within an array ; also there may be an array as an element of a structure. Let's discuss various combinations of arrays and structures.

Since an array can contain similar elements, the combination having structures within an array is an array of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to store addresses of 100 members of the council, you need to create an array.

Now, to declare a 100-element array of structures of type addr (defined in previous chapters), we will write :

struct addr mem_addr [100];

This creates 100 sets of variables that are organised as defined in the structure addr. To access a specific structure, index the structure name. For instance, to print the houseno of structure 8, write :

cout << mem_add[7].houseno ;

Always remember for all arrays in C++, indexing begins at 0.

An array of structures may even contain nested structures. For example, you can even create an array having structures emp type which is a nested structure (already defined in previous chapter) :

emp sales_emp[100] ;

The above declaration creates an array sales_emp to store 100 structures of emp type.

C++ Structure Array Example

Here is an example program, demonstrating structure array (array of structure) in C++. This program creates an array of structures

```
#include<iostream.h>
#include<conio.h>
void main ( )
{
          struct student
        {
            int subject1 ;
            int subject2 ;
            int subject3 ;
        };
             int i , n, total;
             float av ;
             clrscr( );
             struct student st[20];
             cout<<" \n Enter the Number of Students : " ;
             cin>> n ;
             for (i =0; i<n; i++)
                {
                     cout<<"\nEnter Marks of three Subjects of "<<i+1<<" Student : " ;
                     total = 0 ;
                     cin>> st[i].subject1 >>st[i].subject2>>st[i].subject3;
                     total = st[i].subject1+st[i].subject2+st[i].subject3;
                     av = (float) total /3 ;
                     cout<<"\nAVERAGE Marks of "<<i+1<<" Student is = "<< av ;
                 }
                     getch( );
}
```

UNION

Union is also like a Structure means Unions is also used for Storing data of different data types But the Difference b/w Structure and Union is that Structure Consume the Memory of addition of all elements of Different Data Types but a Union Consume the Memory that is Highest among all variables. It Consume Memory of highest variables and it will share the data among all the other Variables Suppose that if a union Contains variables Like Int ,Float ,Char then the Memory Will be consumed of Float variable because float is highest among all variables of data types etc. We can declare a Union in a Structure and Vice-versa.

- A union is a user-defined type in which all members share the same memory location.
- This means that at any given time a union can contain no more than one object from its list of members.

- It also means that no matter how many members a union has, it always uses only enough memory to store the largest member.
- Unions can be useful for conserving memory when you have lots of objects and/or limited memory.
- However they require extra care to use correctly because you are responsible for ensuring that you always access the last member that was written to.

The union is declared as:

union union-type-name

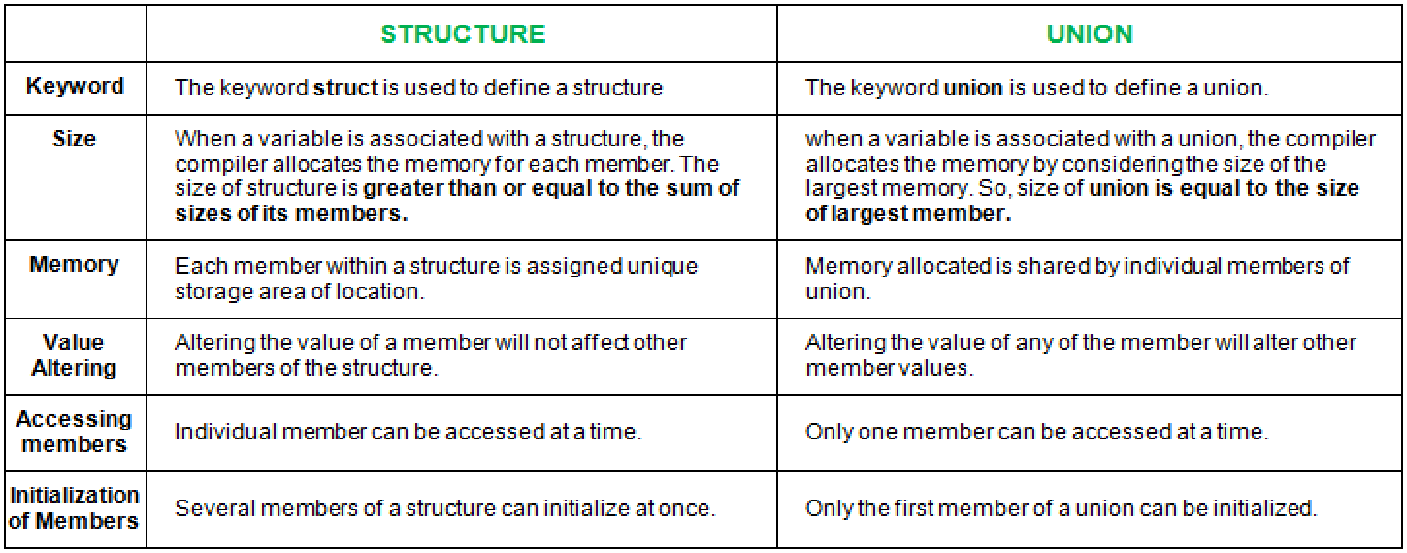
{

union-list

} union-variable;

Difference between Structure and Union.

Unions:



	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Unions in C++ are a user defined data type that uses the same memory as other objects from a list of objects. At an instance it contains only a single object.

Syntax:

```
union union-type-name

{
type member-name;
type member-name;
}union-variables;
```

Example :

```
#include <iostream.h>

union Emp
{
int num;
double sal;
};

int main()
{
Emp value;
value.num = 2;
cout <<"Employee Number::"<< value.num<<"\nSalary is:: "<< value.sal << endl;value.sal = 2000.0;
cout <<"Employee Number::"<< value.num<<"\nSalary is:: "<< value.sal << endl;
return 0;
}
```

Result :

```
Employee number is::2
Salary is::2.122e-314
Employee number is::0
Salary is::2000
```

In the above example, only "value.num" is assigned, but still the "val.sal" gets a value automatically, since the memory locations are same.

Manipulating individual members unions

```
#include<conio.h>
#include<iostream.h>
void main()
{
    union student
     {
          char grade;
          int rollno;
          float marks;
          double fees;
     }s;
          clrscr();
          cout<<"Enter the Garde of the Student : ";
          cin>>s.grade;
          cout<<"Grade is :"<<s.grade<<endl;
          cout<<"Enter the RollNo of the Student : ";
```

```
            cin>>s.rollno;
            cout<<"Rollno is:"<<s.rollno<<endl;
            cout<<"Enter the Marks of the Student : ";
            cin>>s.marks;
            cout<<"Marks are :"<<s.marks<<endl;
            cout<<"Enter the Fees of the Student : ";
            cin>>s.fees;
            cout<<"Fees paid is: "<<s.fees;
            getch();
   }
```

Understanding a Pointer Variable

What is a Pointer?

A pointer is a variable that holds a memory address where a value stored. Every variable is located under unique location within a computer's memory and this unique location has its own unique address, the memory address. However, pointer is a different beast, because it holds the memory address as its value and has an ability to “point” (hence pointer) to certain value within a memory, by use of its associated memory address.

A pointer is declared using the * operator before an identifier.

C++ Pointer Declaration

The general form of a pointer declaration is as follows :

```
type ∗var_name ;
```

where type is any valid C++ data type and var_name is the name of the pointer variable. Following declarations declares pointers of different types :

```
int ∗iptr ;     //creates an integer pointer iptr
char ∗cptr ;     //creates a character pointer cptr
float ∗fptr ;     //creates a float pointer fptr
```

When we say that iptr is an integer pointer, it means that the memory location being pointer to by iptr can hold only integer values. In other words, it can stated that iptr is a pointer to integer. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointers correctly.

Two special operators ∗ and & are used with pointers. The & is a unary operator that returns the memory address of its operand. For example,

```
int i = 25 ;    // declares an int variable i
int ∗iptr ;     // declares an int pointer iptr
iptr = &i ;     // stores the memory address of i into iptr
```

The above given assignment statement stores the memory address of i (& returns memory address of its operand) into iptr so that iptr is now pointing to the memory location of i. The expression &i will return an int pointer value because i is an integer thus, &i will return a pointer to integer. Similarly, if ch is a character variable, then &ch will return a char pointer value.

Note - The operand of & operator is as ordinary available but operand of ∗ is a pointer variable.

Using '∗' operator, changing/accessing value being pointed to by pointer (its state) is called Dereferencing.

Dereferencing refers to changing/accessing state of the pointer.

The pointer operator ∗ (called "at address" sign) is the same sign as multiply operator ∗ and the pointer operator & (called "address of" sign) is same as bitwise AND operator &. But these operators have no relationship to each other, Both pointer operator & and ∗ have a higher precedence than all other arithmetic operators except the unary minus, with which they have equal precedence.

Using Pointers in C++

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

```
#include <iostream>
using namespace std;
int main ()

{
  int  var = 20;   // actual variable declaration.
  int  *ip;        // pointer variable

  ip = &var;       // store address of var in pointer variable

  cout << "Value of var variable: ";
  cout << var << endl;

  // print the address stored in ip pointer variable
  cout << "Address stored in ip variable: ";
  cout << ip << endl;

  // access the value at the address available in pointer
  cout << "Value of *ip variable: ";
  cout << *ip << endl;

  return 0;
}
```

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

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined –

```
#include <iostream>

using namespace std;
int main ()
{
  int  var1;
  char var2[10];

  cout << "Address of var1 variable: ";
  cout << &var1 << endl;

  cout << "Address of var2 variable: ";
  cout << &var2 << endl;

  return 0;
}
```

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

```
Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6
```

Dereferencing a Pointer

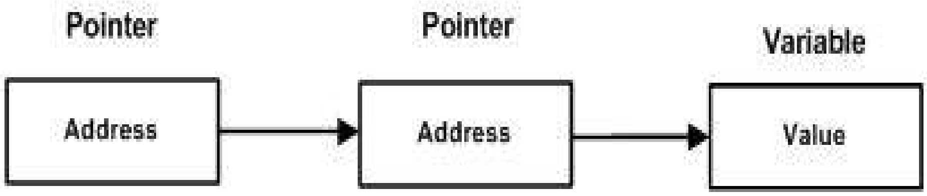
The **dereference** operation follows a pointer's reference to get the value of it is pointing to. When the dereference operation is used correctly, it's really simple. It just accesses the value of it points to. The only restriction is that the pointer must have an object for the dereference to access. Almost all bugs in pointer code involve violating that one restriction. A pointer must be assigned an object that it refers before dereference operations will work.

We **dereference** a pointer by using *, the **deference** operator.

```
cout << "myScore = " << *ptr << endl;
```

Pointers to Pointers

A pointer to a pointer is a form of multiple indirections or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int –

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example –

```
#include <iostream>
 using namespace std;
 int main ()
{
  int  var;
  int  *ptr;
  int  **pptr;

  var = 3000;

  // take the address of var
  ptr = &var;

  // take the address of ptr using address of operator &
  pptr = &ptr;

  // take the value using pptr
  cout << "Value of var :" << var << endl;
  cout << "Value available at *ptr :" << *ptr << endl;
  cout << "Value available at **pptr :" << **pptr << endl;

  return 0;
}
```

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

```
Value of var :3000
Value available at *ptr :3000
Value available at **pptr :3000
```

Pointers to Structure

A pointer variable can be created not only for native types like (int, float,double etc.) but they can also be created for user defined types like structure.

Example :

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

struct Distance
{
   int feet;
   float inch;
};

int main()
{
   Distance *ptr, d;

   ptr = &d;

   cout << "Enter feet: ";
   cin >> (*ptr).feet;
   cout << "Enter inch: ";
   cin >> (*ptr).inch;

   cout << "Displaying information." << endl;
   cout << "Distance = " << (*ptr).feet << " feet " << (*ptr).inch << " inches";

   return 0;
}
```

Output

```
Enter feet: 4
Enter inch: 3.5
Displaying information.
```

```
Distance = 4 feet 3.5 inches
```

In this program, a pointer variable `ptr` and normal variable `d` of type structure `Distance` is defined.

The address of variable `d` is stored to pointer variable, that is, `ptr` is pointing to variable `d`. Then, the member function of variable `d` is accessed using pointer.

Note: Since pointer `ptr` is pointing to variable `d` in this program,`(*ptr).inch` and `d.inch` is exact same cell. Similarly, `(*ptr).feet` and `d.feet`is exact same cell.

The syntax to access member function using pointer is ugly and there is alternative notation -> which is more common.

```
ptr->feet is same as (*ptr).feet

ptr->inch is same as (*ptr).inch
```

Passing Pointers to Function

Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.

Let's first consider an example that will swap two numbers i.e., interchange the values of two numbers.

```
#include <iostream>
using namespace std;
void swap( int *a, int *b )
{
        int t;
        t = *a;
        *a = *b;
        *b = t;
}
int main()
{
        int num1, num2;
        cout << "Enter first number" << endl;
        cin >> num1;
        cout << "Enter second number" << endl;
        cin >> num2;
        swap( &num1, &num2);
        cout << "First number = " << num1 << endl;
        cout << "Second number = " << num2 << endl;
        return 0;
}
```

Swapping means to interchange the values. **void swap(int *a, int *b)** - It means our function 'swap' is taking two pointers as argument. So, while calling this function, we will have to pass the address of two integers (**call by reference**). **int t; t = *a;** We took any integer t and gave it a value '*a'. ***a = *b** - Now, *a is *b. This means that now the values of *a and *b will be equal to that of *b. ***b = t;** - Since 't' has an initial value of '*a', therefore, '*b' will also contain that initial value of '*a'. Thus, we have interchanged the values of the two variables.

Since we have done this swapping with pointers (we have targeted on address), so, this interchanged value will also reflect outside the function and the values of 'num1' and 'num2' will also get interchanged.

In the above example, we passed the address of the two variables (num1 and num2) to the swap function. The address of num1 is stored in 'a' pointer and that of num2 in 'b' pointer. In the swap function, we declared a third variable’t’ and the values of 'a' and 'b' (and thus that of num1 and num2) gets swapped.

Returning Pointer from Functions

C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.

C++ allows returning an array from a function, similar way C++ allows you to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example –

```
int * myFunction()
{
  .
  .
  .
}
```

Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function, so you would have to define the local variable as **static** variable.

Now, consider the following function, which will generate 10 random numbers and return them using an array name which represents a pointer i.e., address of first array element.

```
#include <iostream>
#include <ctime>

using namespace std;

// function to generate and retrun random numbers.
int * getRandom( )
{
  static int  r[10];

  // set the seed
  srand( (unsigned)time( NULL ) );

  for (int i = 0; i < 10; ++i)
 {
    r[i] = rand();
    cout << r[i] << endl;
  }

  return r;
}

// main function to call above defined function.
int main () {
  // a pointer to an int.
  int *p;

  p = getRandom();
  for ( int i = 0; i < 10; i++ )
 {
    cout << "*(p + " << i << ") : ";
    cout << *(p + i) << endl;
  }

  return 0;
}
```

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

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
```

```
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

Using Arrays as Pointer

Using Arrays as Pointer can define arrays to hold a number of pointers.Following is the declaration of an array of pointers to an integer –

```
int *ptr[MAX];
```

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers which will be stored in an array of pointers as follows

```
#include <iostream>
using namespace std;
const int MAX = 3;

int main ()
{
  int  var[MAX] = {10, 100, 200};
  int *ptr[MAX];

  for (int i = 0; i < MAX; i++)
 {
    ptr[i] = &var[i]; // assign the address of integer.
  }

  for (int i = 0; i < MAX; i++)
{
    cout << "Value of var[" << i << "] = ";
    cout << *ptr[i] << endl;
  }

  return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Passing Arrays to functions

C++ Program to display marks of 5 students by**passing** one-dimensional **array** to a **function**. When an **array** is passed as an argument to a **function**, only the name of an **array** is used as argument. display(marks); Also notice the difference while **passing array** as an argument rather than a variable.

Arrays can be passed to a function as an argument. Consider this example to pass one-dimensional array to a function:

Example 1: Passing One-dimensional Array to a Function

C++ Program to display marks of 5 students by passing one-dimensional array to a function.

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

void display(int marks[5]);

int main()
{
   int marks[5] = {88, 76, 90, 61, 69};
   display(marks);
   return 0;
}

void display(int m[5])
{
   cout << "Displaying marks: "<< endl;

   for (int i = 0; i < 5; ++i)
   {
      cout << "Student "<< i + 1 <<": "<< m[i] << endl;
   }
}
```

Output

```
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69
```

- When an array is passed as an argument to a function, only the name of an array is used as argument.display(marks);
- Also notice the difference while passing array as an argument rather than a variable.

```
void display(int m[5]);
```

- The argument marks in the above code represents the memory address of first element of array marks[5].
- And the formal argument int m[5] in function declaration converts to int* m;. This pointer points to the same address pointed by the array marks.
- That's the reason, although the function is manipulated in the user-defined function with different array name m[5], the original array marks is manipulated.
- C++ handles passing an array to a function in this way to save memory and time.

Pointers vs References

Pointers vs References in C++ C and **C++** support**pointers** which is different from most of the other programming languages. ... A **pointer** needs to be dereferenced with * operator to access the memory location it points to. **References** : A **reference** variable is an alias, that is, another name for an already existing variable.

Example :

Following example makes use of references on int and double −

```
#include <iostream>

using namespace std;

int main ()
 {
  // declare simple variables
  int    i;
  double d;

  // declare reference variables
  int&    r = i;
  double& s = d;

  i = 5;
  cout << "Value of i : " << i << endl;
  cout << "Value of i reference : " << r  << endl;

  d = 11.7;
  cout << "Value of d : " << d << endl;
  cout << "Value of d reference : " << s  << endl;
  return 0;
}
```

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

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

References are usually used for function argument lists and function return values. So following are two important subjects related to C++ references which should be clear to a C++ programmer –

Declaring and initializing references.

On the surface, both references and pointers are very similar, both are used to have one variable provide access to another. With both providing lots of same capabilities, it's often unclear what is different between these different mechanisms. In this article, I will try to illustrate the differences between pointers and references.

Pointers: A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with * operator to access the memory location it points to.

References : A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer is also implemented by storing the address of an object. A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the * operator for you.

```
int i = 3;

// A pointer to variable i (or stores

// address of i)

int *ptr = &i;

// A reference (or alias) for i.

int &ref = i;
```

Example :

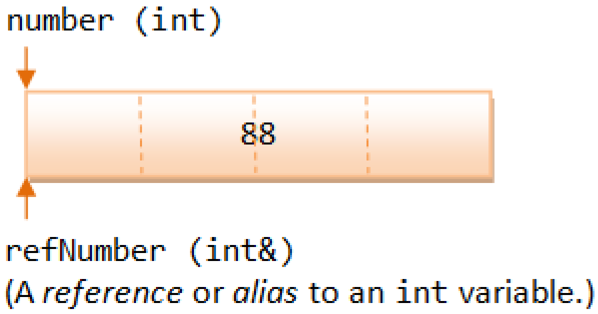
The syntax is as follow:

```
type &newName = existingName;
// or
type& newName = existingName;
// or
type & newName = existingName;  // I shall adopt this convention
```

It shall be read as "*newName is a reference to exisitngName*", or "*newNew is an alias of existingName*". You can now refer to the variable as *newName* or*existingName*.

For example,

```
/* Test reference declaration and initialization (TestReferenceDeclaration.cpp) */
#include <iostream>
using namespace std;

int main()
{
  int number = 88;                          // Declare an int variable called number
  int & refNumber = number;                // Declare a reference (alias) to the variable number
                                           // Both refNumber and number refer to the same value

  cout << number << endl;                 // Print value of variable number (88)
  cout << refNumber << endl;             // Print value of reference (88)

  refNumber = 99;                        // Re-assign a new value to refNumber
  cout << refNumber << endl;
  cout << number << endl;              // Value of number also changes (99)

  number = 55;                        // Re-assign a new value to number
  cout << number << endl;
  cout << refNumber << endl;      // Value of refNumber also changes (55)
}
```



Function's Return Value

You can also pass the return-value as reference or pointer. For example,

```
/* Passing back return value using reference (TestPassByReferenceReturn.cpp) */
#include <iostream>
using namespace std;

int & squareRef(int &);
int * squarePtr(int *);

int main( )
{
  int number1 = 8;
  cout <<  "In main( ) &number1: " << &number1 << endl;  // 0x22ff14
  int & result = squareRef(number1);
  cout <<  "In main() &result: " << &result << endl;         // 0x22ff14
  cout << result << endl;                                      // 64
  cout << number1 << endl;                                     // 64

  int number2 = 9;
  cout <<  "In main() &number2: " << &number2 << endl;  // 0x22ff10
  int * pResult = squarePtr(&number2);
  cout <<  "In main() pResult: " << pResult << endl;  // 0x22ff10
  cout << *pResult << endl;                               // 81
  cout << number2 << endl;                                // 81
}

int & squareRef(int & rNumber)
{
  cout <<  "In squareRef(): " << &rNumber << endl;  // 0x22ff14
  rNumber *= rNumber;
  return rNumber;
}

int * squarePtr(int * pNumber)
{
  cout <<  "In squarePtr(): " << pNumber << endl;  // 0x22ff10
  *pNumber *= *pNumber;
  return pNumber;
}
```

Reference as function argument

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

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

For now, let us call the function **swap()** by passing values by reference as in the following example –

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

// function declaration
void swap(int &x, int &y);

int main () {
  // local variable declaration:
```

```
 int a = 100;
  int b = 200;
 cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  /* calling a function to swap the values using variable reference.*/
  swap(a, b);

  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result −

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

Structure with union as members

Structures group members (data and functions) to create new data types. Structures encapsulate data members (usually different data types), much like functions encapsulate program statements. Unions are like structures, but data members overlay (share) memory, and unions may access members as different types. We use structures and unions in applications that need user-defined types, such as databases, windows, and graphics.

```
#include <iostream>
using namespace std;
union sab
{
int a;
}
struct stud
{
Int regno;
union sab x;
};
Struct stud s;
s.regno=10;
s.x.a=20;
cout<<s.regno<<s.x.a;
}
```

Union with structures as members

Declaration and Initialization of structure starts with struct keyword. Declaration and Initialization of union starts with union keyword.Structure allocates different memory locations for all its members while union allocates common memory location for all its members. The memory occupied by a union will be large enough to hold the largest member of the union.

```
#include<iostream.h>

 struct Employee1
 {
                 int Id;
                 char Name[25];
                 long Salary;
 };

 union Employee2
 {
                 int Id;
                 char Name[25];
                 long Salary;
 };

  void main()
  {

                 cout << "\nSize of Employee1 is : " << sizeof(Employee1);
                 cout << "\nSize of Employee2 is : " << sizeof(Employee2);

  }
```

Output :

```
                 Size of Employee1 is : 31
                 Size of Employee2 is : 25
```

Example 2:

```
#include <iostream>
using namespace std;
struct stud
{
int regno;
};
Union sample
{
struct stud x;
int a;
};

int main()
{
union sample s;
s.x.regno=10;
cout<<s.x.regno;
s.a=20;
cout<<s.a;
}
```

KARPAGAM
ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)

KARPAGAM ACADEMY OF HIGHER EDUCATION
COIMBATORE - 21

DEPARTMENT OF COMPUTER SCIENCE,CA & IT
CLASS : I B.Sc COMPUTER SCIENCE

BATCH : 2019-2022

Part -A Online Examinations **(1 mark questions)**
SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++ **SUBJECT CODE: 19CSU101**

UNIT-III

S.No	Questions	OPT1	OPT2	OPT3	OPT4	Answer
1	C++ supports all the features of ____________ as defined in C	structures	union	objects	classes	structures
2	A structure can have both variable and functions as _________	objects	classes	members	arguments	members
3	The class _________ describes the type and scope of its members	calling function	declaration	objects	none of the above	declaration
4	The class ___________ describes how the class function are implemented	Function definition	declaration	arguments	none of the above	Function definition
5	The keywords private and public are known as __________ labels	Static	dynamic	visibility	const	visibility
6	The class members that have been declared as _________ can be accessed only from within the class	Private	public	static	protected	Private
7	The class members that have been declared as _________ can be accessed from outside the class also	Private	Public	static	protected	Public

8	The variables declared inside the class are called as _________	Function variables	data members	member function	data variables	data members
9	The symbol ______ is called the scope resolution operator	>>	::	<<	::*	::
10	A member function can call another member function directly without using the _________ operator	Assignmen t	equal	dot	greater than	dot
11	A ______ member variable is initialized to zero when the first object of its class is created	Dynamic	constant	static	protected	static
12	_________ Variables are normally used to maintain values common to the entire class.	Private	protected	Public	static	static
13	When a copy of the entire object is passed to the function it is called as _________	Pass by reference	pass by function	pass by pointer	pass by value	pass by value
14	The ________ member variables must be defined outside the class.	Static	private	public	protected	Static
15	A friend function, although not a member function, has full access right to the ______ members of the	Static	private	public	protected	private
16	__________ enables an object to initialize itself when it is created	Destructor	constructor	overloading	none of the above	constructor
17	________ destroys the objects when they are no longer required	Destructor	constructor	overloading	none of the above	Destructor
18	The __________ is special because its name is the same as the class name.	Destructor	static	constructor	none of the above	constructor

19	A constructor that accepts no parameters is called the __________ constructor	Copy	default	multiple	none of the above	default
20	Constructors are invoked automatically when the ________ are created	Datas	classes	objects	none of the above	objects
21	Constructors cannot be _________	Inherited	destroyed	both a & b	none of the above	Inherited
22	Constructors cannot be _________	Destroyed	virtual	both a & b	none of the above	virtual
23	Constructors make _________ calls to the operators new and delete when memory allocation is required	Explicit	implicit	function	none of the above	implicit
24	The constructors that can take arguments are called _________ constructors	Copy	multiple	parameterized	none of the above	parameterized
25	The constructor function can also be defined as ________ function	Friend	inline	default	none of the above	inline
26	When a constructor can accept a reference to its own class as a parameter, in such cases it is	Multiple	copy	default	none of the above	copy
27	When more than one constructor function is defined in a class, then the constructor is said to be	Multiple	copy	default	overloaded	overloaded
28	C++ complier has a _________ constructor, which creates objects, even though it was not defined in the class.	Explicit	default	implicit	none of the above	implicit
29	A _________ constructor is used to declare and initialize an object from another object	Default	copy	multiple	parameterized	copy

30	The process of initializing through a copy constructor is known as ________ initialization	Overloaded	multiple	copy	none of the above	copy
31	A ______ constructor takes a reference to an object of the same class as itself as an argument	Delete	new	copy	none of the above	copy
32	Allocation of memory to objects at the time of their construction is known as ________ construction	Static	copy	dynamic	none of the above	dynamic
33	We can create and use constant objects using ______ keyword before object declaration.	Static	new	const	none of the above	const
34	A destructor is preceded by ______ symbol	Dot	asterisk	colon	tilde	tilde
35	_________ is used to allocate memory in the constructor	Delete	binding	free	new	new
36	_________ is used to free the memory	new	delete	clrscr()	none of the above	delete
37	Which is a valid method for accessing the first element of the array item?	item(1)	item[1]	item[0]	item(0)	item[0]
38	Which of the following statements is valid array declaration?	int number (5);	float avg[5];	double [5] marks;	counter int[5];	float avg[5];
39	An object is an _________ unit	group	individual	both a&b	none of the above	individual
40	Public keyword is terminated by a ________	Semicolon	comma	dot	colon	colon

41	Private keyword is terminated by a _________	semicolon	comma	dot	colon	colon
42	The memory for static data is allocated only ________	twice	thrice	once	none of the above	once
43	Static member functions can be invoked using ________ name	class	object	data	function	class
44	The _________ doesn’t have any argument	constructor	copy constructor	destructor	none of the above	destructor
45	The _________ also allocates required memory .	constructor	destructor	both a & b	none of the above	constructor
46	Any constructor or destructor created by the complier will be _______	private	public	protected	none of the above	public
47	_________ releases memory space occupied by the objects	constructor	destructor	both a & b	none of the above	destructor
48	Constructors and destructors are automatically inkoved by _________	operating system	main()	complier	object	complier
49	Constructors is executed when ________	object is destroyed	object is declared	both a & b	none of the above	object is declared
50	The destructor is executed when __________	object goes out of scope	when object is not used	when object contains	none of the above	object goes out of scope
51	The members of a class are by default ________	protected	private	public	none of the above	private

52	The ________ is executed at the end of the function when objects are of no used or goes out of scope	destructor	constructor	inheritance	none of the above	destructor

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.

Differentiating between static and dynamic memory allocation

Memory in your C++ program is divided into two parts –

- **The stack** – All variables declared inside the function will take up memory from the stack.
- **The heap** – This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

Following are the differences between Static Memory Allocation and Dynamic Memory Allocation:

Static Memory Allocation	Dynamic Memory Allocation
In static memory allocation, memory is allocated **before the execution** of the program begins.	In Dynamic memory allocation, memory is allocated **during the execution** of the program.
Memory allocation and deallocation actions are not performed during the execution.	Memory allocation and deallocation actions are performed during the execution.
Static memory allocation, pointer is required to access the variables.	It does not require pointers to allocate the variables dynamically.
It performs execution of program faster than dynamic memory.	It performs execution of program slower than static.
It requires more memory space.	It requires less memory space.
The data in static memory is allocated permanently.	The data in dynamic memory is allocated only when program unit is active.

New Operator

- New operator is used to dynamically allocate the memory.
- The **new** keyword is used to allocate the memory space dynamically followed by a data type specifier.

Syntax:

```
newint;        //dynamically allocates an integer
new float;     //dynamically allocates an float
```

- For creating an array dynamically, use the same form but put the square brackets ([]) with a size after the data type.

Syntax:

```
newint[10];        //dynamically allocates an array of 10 integers.
new float[10];    //dynamically allocates an array of 10 floats.
```

- In the above syntax, the allocated spaces have no names, but the new operator returns the starting address of the allocated space and stored it in a pointer.

Syntax:

```
int *ptr;            //declare a pointer ptr
ptr = new int;    //dynamically allocate an int and load address into ptr
```

Delete Operator

- Delete operator is used to deallocate the memory.
- This operator deallocates the memory previously allocated by the new operator.
- If the memory allocated dynamically to a variable is not required anymore, you can free up the memory with delete operator.

Syntax:

```
delete variable_name;
```

Example:

```
delete ptr;    //Releases memory pointed to by ptr
```

- Using delete operator the memory becomes available again for other requests of dynamic memory.

Example : Demonstrating how new & delete operators work

```
#include <iostream>
using namespace std;
int main ()
{
    int *ptr  = NULL;       // Pointer initialized with null
    ptr  = new int;         // Request memory for the variable
    *ptr = 12345;        // Store value at allocated address
    cout<< "Value of Pointer Variable *ptr : " << *ptr<<endl;
    delete ptr;          // free up the memory.
    return 0;
}
```

OR

```
#include<iostream>
usingnamespace std;

int main ()
{
double*pvalue= NULL;// Pointer initialized with null
pvalue=newdouble;// Request memory for the variable

*pvalue=29494.99;// Store value at allocated address
cout<<"Value of pvalue : "<<*pvalue<<endl;

deletepvalue;// free up the memory.

return0;
}
```

If we compile and run above code, this would produce the following result –

```
Value of pvalue : 29495
```

Use of malloc, calloc and free functions:

Use of malloc

The malloc() function in C++ allocates a block of uninitialized memory and returns a void pointer to the first byte of the allocated memory block if the allocation succeeds.

The malloc() function in C++ allocates a block of uninitialized memory and returns a void pointer to the first byte of the allocated memory block if the allocation succeeds.

If the size is zero, the value returned depends on the implementation of the library. It may or may not be a null pointer.

malloc() prototype

```
void* malloc(size_t size);
```

This function is defined in <cstdlib> header file.

---

malloc() Parameters

- size: An unsigned integral value which represents the memory block in bytes.

The malloc() function returns:

- a pointer to the uninitialized memory block allocated by the function.
- null pointer if allocation fails.
-

Example 1: How malloc() function works?

```
#include<iostream>
#include<cstdlib>
usingnamespace std;

int main()
{
        int*ptr;
        ptr=(int*)malloc(5*sizeof(int));

        if(!ptr)
        {
                cout<<"Memory Allocation Failed";
                exit(1);
        }
        cout<<"Initializing values..."<<endl<<endl;
        for(int i=0; i<5; i++)
        {
                ptr[i]= i*2+1;
        }
        cout<<"Initialized values"<<endl;

        for(int i=0; i<5; i++)
        {
                /* ptr[i] and *(ptr+i) can be used interchangeably */
                cout<<*(ptr+i)<<endl;
        }

        free(ptr);
        return0;
}
```

When you run the program, the output will be:

```
Initializing values...

Initialized values
1
3
5
7
9
```

Calloc function

The calloc() function in C++ allocates a block of memory for an array of objects and initializes all its bits to zero.

The calloc() function returns a pointer to the first byte of the allocated memory block if the allocation succeeds.

If the size is zero, the value returned depends on the implementation of the library. It may or may not be a null pointer.

calloc() prototype

```
void* calloc(size_tnum, size_t size);
```

The function is defined in <cstdlib> header file.

calloc() Parameters

- num: An unsigned integral value which represents number of elements.
- size: An unsigned integral value which represents the memory block in bytes.

free

(Free the memory allocated using malloc, calloc or realloc)

free functions frees the memory on the heap, pointed to by a pointer. Signature of free function is

void free(void* ptr);

- ptr must be pointing to a memory which is allocated using malloc, calloc or realloc.
- If ptr is called on a memory which is not on heap or on a dangling pointer, then the behavior is undefined.
- If ptr is NULL, then free does nothing and returns (So, its ok to call free on null pointers).

int x = 2;

int* ptr = &x;

free(ptr); //UNDEFINED.

int *ptr2; // UN initialized, hence dangling pointer

free(ptr2); //UNDEFINED

int *ptr3 = NULL;

free(ptr3); //OK.

Crashes in malloc(), calloc(), realloc(), or free() are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

Opening and closing a file

The **iostream** standard library, which provides **cin**and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types –

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.

2	**ifstream** This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only. Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstreamoutfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstreamafile;
```

```
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

```
#include<fstream>

#include<iostream>

usingnamespace std;

int main ()

{

char data[100];

// open a file in write mode.

ofstreamoutfile;

outfile.open("afile.dat");

cout<<"Writing to the file"<<endl;

cout<<"Enter your name: ";
cin.getline(data,100);
// write inputted data into the file.
outfile<< data <<endl;
cout<<"Enter your age: ";
cin>> data;
cin.ignore();
// again write inputted data into the file.
outfile<< data <<endl;
// close the opened file.
outfile.close();
// open a file in read mode.
ifstreaminfile;
infile.open("afile.dat");
cout<<"Reading from the file"<<endl;
infile>> data;
// write the data at the screen.
cout<< data <<endl;
// again read the data from the file and display it.
infile>> data;
cout<< data <<endl;
// close the opened file.
infile.close();
return0;
}
```

When the above code is compiled and executed, it produces the following sample input and output –

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

Random Access of Files (File Pointers)

Using file streams, we can randomly access binary files. By random access, you can go to any position in the file as you wish (instead of going in a sequential order from the first character to the last). Earlier in this chapter we discussed about a bookmarker that will keep moving as you keep reading a file.

This bookmarker will move sequentially but you can also make it move randomly using some functions. Technically this bookmarker is a file pointer and it determines as to where to write the next character (or from where to read the next character). We have seen that file streams can be created for input (ifstream) or for output (ofstream).

For ifstream the pointer is called as ‘get’ pointer and for ofstream the pointer is called as ‘put’ pointer. fstream can perform both input and output operations and hence it has one ‘get’ pointer and one ‘put’ pointer. The ‘get’ pointer indicates the byte number in the file from where the next input has to occur.

The ‘put’ pointer indicates the byte number in the file where the next output has to be made. There are two functions to enable you move these pointers in a file wherever you want to:

seekg ( ) - belongs to the ifstream class
seekp ( ) - belongs to the ofstream class

We’ll write a program to copy the string "Hi this is a test file" into a file called mydoc.txt. Then we’ll attempt to read the file starting from the 8th character (using the seekg( ) function).

Strings are character arrays terminated in a null character (‘\0’). If you want to copy a string of text into a character array, you should make use of the function:

```
strcpy (character-array, text);
```

to copy the text into the character array (even blank spaces will be copied into the character array). To make use of this function you might need to include the string.h header file.

```
#include <iostream.h>
```

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

int main( )
{
ofstream out("c:/mydoc.txt",ios::binary);
char text[80];
strcpy(text,"Hi this is a test file");
out<<text;
out.close( );
ifstream in("c:/mydoc.txt",ios::binary);
in.seekg(8);
cout<<endl<<"Starting from position 8 the contents are:"<<endl;

while ( !in.eof( ) )
{
charch;
in.get(ch);
if ( !in.eof( ) )
   {
cout<<ch;
   }
}

in.close( );
return 0;
}
```

The output is:

```
Starting from position 8 the contents are:
is a test file
```

As you can see, the output doesn’t display, "Hi this " because they are the first 7 characters present in the file. We’ve asked the program to display from the 8th character onwards using the seekg( ) function.

```
in.seekg(8);
```

will effectively move the bookmarker to the 8th position in the file. So when you read the file, you will start reading from the 8th position onwards.

The following fragment of code is interesting:

```
while ( !in.eof( ) )
{
charch;
in.get(ch);
if ( !in.eof( ) )
   {
cout<<ch;
   }
}
```

You might be wondering as to why we need to check for the EOF again using an 'if' statement. To understand the reason, try the program by removing the 'if' statement. The result will be surprising and interesting. Think over it and you will be able to figure out the logic.

The syntax for seekg( ) or seekp( ) is:

```
seekg(position, ios::beg)
seekg(position, ios::cur)
seekg(position, ios::end)
```

By default (i.e. if you don't specify 'beg' or 'cur' or 'end') the compiler will assume it as ios::beg.

ios::beg – means that the compiler will count the position from the beginning of the file.
ios::cur – means the compiler starts counting from the current position.
ios::end – it will move the bookmarker starting from the end of the file.
Just like we have 2 functions to move the bookmarker to different places in the file, we have another 2 functions that can be used to get the present position of the bookmarker in the file.

For input streams we have: tellg( )

For output streams we have :tellp( )

You would think that the value returned by tellg ( ) and tellp ( ) are integers. They are like integers but they aren't.

The actual syntax for these functions will be:

```
streampostellg ( );
```

wherestreampos is an integer value that is defined in the compiler (it is actually a typedef).

Of course you can say:

```
int position = tellg ( );
```

Now, the variable ‘position’ will have the location of the bookmarker. But you can also say:

```
streampos position = tellg( );
```

This will also give the same result. ‘streampos’ is defined internally by the compiler specifically for file-streams.

Similarly, the syntax of seekg ( ) and seekp ( ) was mentioned as:

```
seekg(position, ios::beg)
```

Again in the above syntax, ‘position’ is actually of type ‘streampos’.

preprocessors

The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).

You already have seen a #include directive in all the examples. This macro is used to include a header file into the source file.

There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc. Let us see important directives −

The #define Preprocessor

The #define preprocessor directive creates symbolic constants. The symbolic constant is called a macro and the general form of the directive is −

```
#define macro-name replacement-text
```

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example −

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

#define PI 3.14159

int main ()
{
```

```
cout<< "Value of PI :" << PI <<endl;

return 0;
}
```

Now, let us do the preprocessing of this code to see the result assuming we have the source code file. So let us compile it with -E option and redirect the result to test.p. Now, if you check test.p, it will have lots of information and at the bottom, you will find the value replaced as follows –

```
$gcc -E test.cpp >test.p
```

```
...
int main ()

 {
cout<< "Value of PI :" << 3.14159 <<endl;
return 0;
}
```

Function-Like Macros

You can use #define to define a macro which will take argument as follows –

Live Demo

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
int i, j;

   i = 100;
   j = 30;

cout<<"The minimum is " << MIN(i, j) <<endl;

return 0;
}
```

If we compile and run above code, this would produce the following result –

```
The minimum is 30
```

Conditional Compilation

There are several directives, which can be used to compile selective portions of your program's source code. This process is called conditional compilation.

The conditional preprocessor construct is much like the ‘if’ selection structure. Consider the following preprocessor code –

```
#ifndef NULL
#define NULL 0
#endif
```

You can compile a program for debugging purpose. You can also turn on or off the debugging using a single macro as follows –

```
#ifdef DEBUG
cerr<<"Variable x = " << x <<endl;
#endif
```

This causes the cerr statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG. You can use #if 0 statment to comment out a portion of the program as follows –

```
#if 0

code prevented from compiling
#endif
```

Let us try the following example –

Live Demo

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main ()
{
int i, j;

   i = 100;
   j = 30;

#ifdef DEBUG
cerr<<"Trace: Inside main function" <<endl;
#endif

#if 0
  /* This is commented part */
cout<< MKSTR(HELLO C++) <<endl;
#endif

cout<<"The minimum is " << MIN(i, j) <<endl;

#ifdef DEBUG
cerr<<"Trace: Coming out of main function" <<endl;
#endif

return 0;
}
```

If we compile and run above code, this would produce the following result –

```
The minimum is 30
Trace: Inside main function
Trace: Coming out of main function
```

The # and ## Operators

The # and ## preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes.

Consider the following macro definition –

Live Demo

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

#define MKSTR( x ) #x

int main ()
{

cout<< MKSTR(HELLO C++) <<endl;

return 0;
}
```

If we compile and run above code, this would produce the following result –

```
HELLO C++
```

Let us see how it worked. It is simple to understand that the C++ preprocessor turns the line –

```
cout<< MKSTR(HELLO C++) <<endl;
```

Above line will be turned into the following line –

```
cout<< "HELLO C++" <<endl;
```

The ## operator is used to concatenate two tokens. Here is an example –

```
#define CONCAT( x, y )  x ## y
```

When CONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CONCAT(HELLO, C++) is replaced by "HELLO C++" in the program as follows.

Live Demo

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

#define concat(a, b) a ## b
int main() {
intxy = 100;

cout<<concat(x, y);
return 0;
}
```

If we compile and run above code, this would produce the following result –

```
100
```

Let us see how it worked. It is simple to understand that the C++ preprocessor transforms –

```
cout<<concat(x, y);
```

Above line will be transformed into the following line –

```
cout<<xy;
```

Predefined C++ Macros

C++ provides a number of predefined macros mentioned below –

Let us see an example for all the above macros –

Live Demo

```
#include <iostream>
```

```
using namespace std;

int main () {
cout<< "Value of __LINE__ : " << __LINE__ <<endl;
cout<< "Value of __FILE__ : " << __FILE__ <<endl;
cout<< "Value of __DATE__ : " << __DATE__ <<endl;
cout<< "Value of __TIME__ : " << __TIME__ <<endl;

return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Value of __LINE__ : 6
Value of __FILE__ : test.cpp
Value of __DATE__ : Feb 28 2011
Value of __TIME__ : 18:52:48
```

#line

When we compile a program and there happens any errors during the compiling process, the compiler shows the error that have happened preceded by the name of the file and the line within the file where it has taken place.

The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that it appears when an error takes place. Its form is the following one:

```
#line number "filename"
```

Where number is the new line number that will be assigned to the next code line. The line number of successive lines will be increased one by one from this.
filename is an optional parameter that serves to replace the file name that will be shown in case of error from this directive until other one changes it again or the end of the file is reached. For example:

```
#line 1 "assigning variable"
int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 1.

#error

This directive aborts the compilation process when it is found returning the error that is specified as parameter:

```
#ifndef __cplusplus
#error A C++ compiler is required
#endif
```

This example aborts the compilation process if the defined constant __cplusplus is not defined.

Macro Functions

#define can be used to make macro functions that will be substituted in the source before compilation. A preprocessor function declaration comprises a macro name immediately followed by parentheses containing the function's argument. Do not leave any space between the name and the parentheses. The declaration is then followed by the function definition within another set of parentheses. For example, a preprocessor macro function to give bigger value of the two looks like this:

```
#define MAX(a,b) (a > b ? a : b)
```

When we use macro functions, however, unlike regular functions, they do not perform any kind of type checking. Because of this drawbacks, inline functions are usually preferable to macro functions. But because macros directly substitute their code, they reduce the overhead of a function call.

```
#define MAX(a,b) (a > b ? a : b)

#include <iostream>
using namespace std ;

inlineint max(int a, int b) {return (a > b ? a: b);}

int main()
{
 int x = 10, y = 20;

cout<< "Macro Max(x,y) = " << MAX(x,y) <<endl;
cout<< "inline max(x,y) = " << max(x,y) <<endl;

return 0;
}
```

Output is:

```
Macro Max(x,y) = 20
inline max(x,y) = 20
```

One of the common mistakes we make when we use Macro is to forget what Macro is suppose to do. In the following example, if we miss parenthesis around it, it will give us unexpected result.

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

```
#define SQUARE(n) ((n)*(n))
int main()
{
int j = 64/SQUARE(4);
printf("j = %d",j);

return 0;
}
```

surprisingly, it prints out j = 64 instead of j = 4.
Why?
Because j = 64/4*4 but not j = 64/(4*4).

So, we need to use the following Macro to get intended answer.

```
#define SQUARE(n) (n*n)
```

Here is another example which may give unexpected results:

```
#include <stdio.h>
#define SQR(n)(n*n)

int main()
{
int a, b = 3;
   a = SQR(b+2);        // a = (b+2*b+2) = 3+2*3+2 = 11 not 25
printf("%d\n", a);
return 0;
}
```

So, in this case, the macro should be:

```
#define SQR(n)((n)*(n))
```

KARPAGAM ACADEMY OF HIGHER EDUCATION
COIMBATORE - 21

DEPARTMENT OF COMPUTER SCIENCE,CA & IT
CLASS : I B.Sc COMPUTER SCIENCE

BATCH : 2019-2022

Part -A Online Examinations **(1 mark questions)**

SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++ **SUBJECT CODE: 19CSU101**

UNIT-IV

S.No	Questions	OPT1	OPT2	OPT3	OPT4	Answer
1	_________ is used as the input stream to read data.	Cout	Printf	Cin	Scanf	Cin
2	cin and cout are ________ for input and output of data.	user defined	system defined	Pre defined stream	none	system defined
3	The data obtained or represented with some manipulators are called ______.	formatted data	unformatted	extracted data	None.	formatted
4	The output formats can be controlled with manipulators having the header file as	iostream.h	conio.h	stdlib.h	iomanip.h	iomanip.
5	The _____ and ______ are derived classes from ios based class.	istream and ostream	source and destination	iostream and source	None.	istream and
6	The manipulator << endl is equivalent to____	‘\t’	’\r’	’\n’	’\b’	’\n’
7	Precision() is an __________ format function	Manipulator	Istream	ios	user defined	ios
8	Width of the output field is set using the ______	width()	iomanip.h	Manipulator	None	width()
9	Stream and stream classes are used to implement its I/O operations with the ______	the console and disk files	cin and cout	manipulators	none	the console

10	The interface supplied by an I/O system which is independent of actual device is called _____	stream	class	object	none.	stream
11	A _____ is a sequence of bytes.	Stream	class	object	none	Stream
12	The _____ streams automatically open when the program begins its execution	user defined	predefined	input	output	predefine
13	The class that is defined to various streams to deal with both the console and disk files is called _________	stream class	derived class	object	none	stream class
14	____ provide an interface to physical devices through buffers.	stream buffer	iostream	ostream	istream	stream buffer
15	The _____ are called as overloaded operators	>> and <<	+ and –	* and &&	– and .	>> and <<
16	The >> operator is overloaded in the ________	istream	ostream	iostream	None	istream
17	The ____ functions are used to handle the single character I/O operation.	get() and put()	clrscr() and getch()	cin and cout	None	get() and put()
18	____ functions are used to display text more efficiently by using the line oriented i/o functions.	getline() and write()	cin and cout	get() and put()	none	getline() and
19	The getline() reads character input to the ______ line	datatype	function	variable	none	variable
20	_____ is used to clear the flags specified.	width()	precision()	setf()	unsetf()	unsetf()
21	_____ is used to specify the required field size for displaying an output value	width()	self	fill()	none	width()
22	By default the floating numbers are printed with ______ after the decimal point.	5 digits	6	7	8	6
23	____ returns the setting in effect until it is reset	width	precision()	setf()	fill()	precision
24	A _______ is a collection of related data stored in a particular area on a disk.	Field	File	Row	Vector	File
25	File streams act as an ________ between programs and files.	interface	converter	translator	operator	interface

26	Ifstram, Ofstream, Fstream are derived form __________.	iostream	ostream	streambuff	fstreambase	fstreamb
27	Classes designed to manage the ________ files are declared in fstream.	random	sequential	disk	tape	sequentia
28	_________ is to set the file buffer to read and write.	filebuf	filestream	thread	package	filebuf
29	________ inherits get(), getline(), read(), seekg(), and tellg() from istream.	conio	ifstream	fstream	iostream	ifstream
30	Put(), seekp(), tellp(), and write() functions are inherited by ofstream from _______	ostream	fstream	ifstream	istream	ostream
31	______ inherits all functions from istream and ostream through iostream	file stream	ofstream	fstream	ifstream	fstream
32	The eof ( ) stands for _____.	end of file	error opening file	error of file	none of the above	end of file
33	Command line arguments are used with ________ function	main()	member function	with all function	none of the above	main()
34	The close() function _________.	closes the file	closes all files opened	closes only read mode	none	closes the file
35	The write() function writes ___________.	single character	object	string	none of these	single character
36	Feof function is used to test	End of file condition	Beginning of file	Middle of the file	Previous file position	End of file
37	__________ is a another memory allocation function th	Malloc()	Realloc()	Calloc()	Free()	Calloc()
38	With the dynamic run time allocation it is responsible t	Malloc()	Realloc()	Calloc()	Free()	Free()
39	List , queue and stack are all inherently	One dimensional	Two dimensional	Multi-dimensional	Hierarchal	One dimensio
40	Program that processes the source code before it passes	Preprocessor	Function	Library function	structure function	Preprocessor
41	C preprocessor offers a special feature known as	Unconditional	Debugging statement	Macro compilation	Conditional compilation	Conditional

42	Fopen() is used for	Create a file	Close a file	Read a file	Write a file	Create a file
43	FILE is a	Keyword	Identifier	Constant	variable	Keyword
44	FILE is a	Function	Structure	Defined data type	I/O function	Defined data type
45	Getc() is used for	Write a character	Read a character	Append a character	None of the above	Read a character
46	Fseek() is used for	Gives current	Gives previous	Sets the position to	Sets desired point	Sets desired
47	Putw() is used for	Write a integer	Read a character	Append a character	None of the above	Write a integer
48	FILE is a structure defined in---	Not defined in I/O library	I/O library	Input library	output library	I/O library
49	Filename specified in FILE concept should have	Primary name and	Secondary name and	Only Primary	Only optional period	Primary name and
50	W mode is used for	Reading and writing	Only reading	Only writing	none	Only writing
51	Filename and mode should be specified in	Double quotation	Single quotation	With tilde symbol	None	Double quotation
52	Fprintf and fscanf function is used for	For printing and reading	Only for reading	Only for writing	Scanning the variables	For printing

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2019 onwards)
DEPARTMENT OF CS, CA & IT

SUBJECT :PROGRAMMING FUNDAMENTALS USING C / C++	**Class : I B. Sc (CS) B**
SEMESTER: I	**SUBJECT CODE: 19CSU101**

UNIT-V

Using Classes in C++:

Principles of Object-Oriented Programming,

Defining & Using Classes,

Class Constructors,

Constructor Overloading,

Copy Constructors,

Function overloading in classes,

Class Variables &Functions,

Objects as parameters,

Specifying the Protected and Private access

Overview of Template classes and their use.

Overview of Function Overloading and Operator Overloading:

Overloading functions

Operators Overloading

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.

Principles of Object-Oriented Programming

Object oriented programming language is a feature that allows a mode of modularizing programs by forming separate memory area for data as well as functions that is used as object for making copies of modules as per requirement

Characteristics of OOPS

- Giving more importance to data than to function.
- Programs are divided into classes and their member function.
- New data items and functions can be added whenever essential.
- Data is private and prevented from accessing external functions.
- Objects can communicate with each other through functions.

The Key concepts of OOPS

- Objects
- Classes
- Abstraction
- Encapsulation
- Inheritance
- polymorphism

Object: Objects are basic run-time entities in an object-oriented system, objects are instances of a class these are defined user defined data types.

ex:

```
Classperson
{
  Charname[20];
  intid;
public:
  voidgetdetails()
{

}
};

Intmain()
{
  person p1; //p1 is a object
}
```

Run on IDE

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each others data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Class: Class is a blueprint of data and functions or methods. Class does not take any space.
Syntax for class:

```
classclass_name
{
 private:
   //data members and member functions declarations
 public:
   //data members and member functions declarations
 protected:
   //data members and member functions declarations
};
```

Run on IDE

Class is a user defined data type like structures and unions in C.

By default class variables are private but in case of structure it is public. in above example person is a class.

Encapsulation

Wrapping up(combing) of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding.

Data abstraction:

Data abstraction refers to, providing only needed information to the outside world and hiding implementation details. For example, consider a class Complex with public functions as getReal() and getImag(). We may implement the class as an array of size 2 or as two variables. The advantage of abstractions is, we can change implementation at any point, users of Complex class wont't be affected as out method interface remains same. Had our implementation be public, we would not have been able to change it.

Inheritance: inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. Inheritance provides re usability. This means that we can add additional features to an existing class without modifying it.

Polymorphism: polymorphism means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.C++ supports operator overloading and function overloading. Operator overloading is the process of making an operator to exhibit different behaviors in different instances is known as operator overloading. Function overloading is using a single function name to perform different types of tasks. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Defining & Using Classes

A class is a logical method to organize data and functions in the same structure. They are declared using keyword **class**, whose functionality is similar to that of the C keyword **struct**, but with the possibility of including functions as members, instead of only data.

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows −

```
class Box
{
  public:
double length;   // Length of a box
double breadth;  // Breadth of a box
double height;   // Height of a box
};


                                 OR
```

```
class classname
 {
   Access - Specifier :
   Member Varibale Declaration;
   Member Function Declaration;
}
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

```
#include <iostream>
#include<conio.h>
using namespace std;

// Class Declaration

class person
{
   //Access - Specifier
public:

   //Variable Declaration
   string name;
   int number;
};

//Main Function

int main()
 {
   // Object Creation For Class
   person obj;

   //Get Input Values For Object Varibales
cout<< "Enter the Name :";
cin>> obj.name;

cout<< "Enter the Number :";
cin>>obj.number;

   //Show the Output
cout<< obj.name << ": " <<obj.number<<endl;

getch();
   return 0;
}
```

Sample Output

```
Enter the Name :Byron
Enter the Number :100
Byron: 100
```

Class Variable and Functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them –

```
class Box
{
public:
double length;        // Length of a box
double breadth;        // Breadth of a box
double height;        // Height of a box
doublegetVolume(void);// Returns box volume
};
```

Member functions can be defined within the class definition or separately using **scope resolution operator,:** –. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume()** function as below –

```
class Box
 {
public:
double length;      // Length of a box
double breadth;     // Breadth of a box
double height;      // Height of a box

doublegetVolume(void)
 {
return length * breadth * height;
     }
};
```

If you like, you can define the same function outside the class using the **scope resolution operator** (::) as follows –

```
double Box::getVolume(void)
{
return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows –

```
Box myBox;         // Create an object

myBox.getVolume();  // Call member function for the object
```

Let us put above concepts to set and get the value of different class members in a class –

Live Demo

```
#include<iostream>

usingnamespacestd;

classBox{
public:
double length;// Length of a box
double breadth;// Breadth of a box
double height;// Height of a box

// Member functions declaration
doublegetVolume(void);
voidsetLength(doublelen);
voidsetBreadth(doublebre);
voidsetHeight(doublehei);
};

// Member functions definitions
doubleBox::getVolume(void)
{
return length * breadth * height;
}

voidBox::setLength(doublelen)
{
length=len;
}
voidBox::setBreadth(doublebre)
{
breadth=bre;
}
voidBox::setHeight(doublehei)
{
height=hei;
}

// Main function for the program
int main()
{
BoxBox1;// Declare Box1 of type Box
BoxBox2;              // Declare Box2 of type Box
double volume =0.0;// Store the volume of a box here

// box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);

// box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

// volume of box 1
volume=Box1.getVolume();
cout<<"Volume of Box1 : "<< volume <<endl;

// volume of box 2
volume=Box2.getVolume();
cout<<"Volume of Box2 : "<< volume <<endl;
return0;
}
```

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

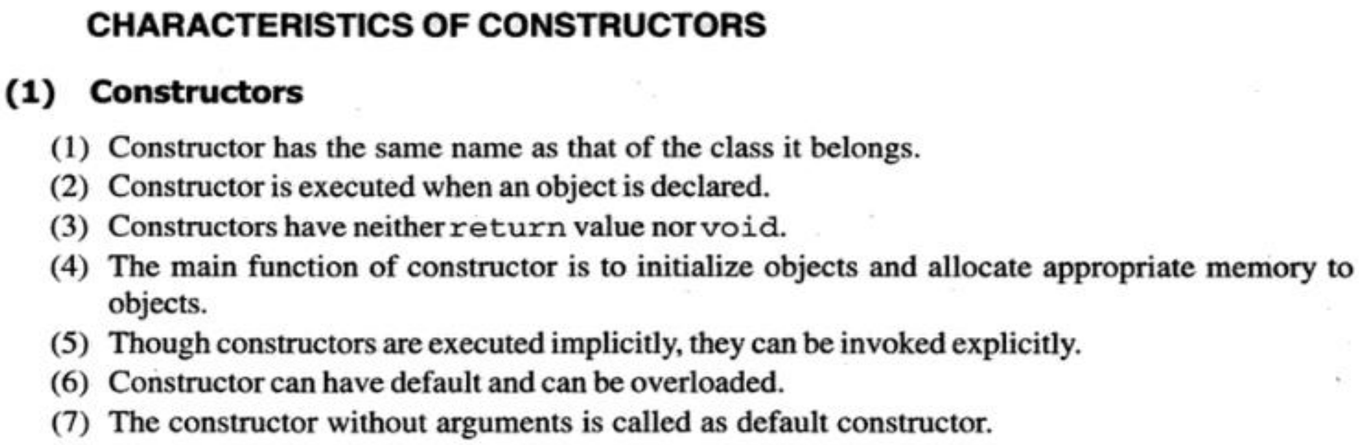
```
Volume of Box1 : 210
Volume of Box2 : 1560
```

https://books.google.co.in/books?id=rA0SWk4dQ-0C&printsec=frontcover&source=gbs_ViewAPI&redir_esc=y#v=onepage&q&f=false

The Class Constructor

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.



CHARACTERISTICS OF CONSTRUCTORS

(1) Constructors

(1) Constructor has the same name as that of the class it belongs.
(2) Constructor is executed when an object is declared.
(3) Constructors have neither `return` value nor `void`.
(4) The main function of constructor is to initialize objects and allocate appropriate memory to objects.
(5) Though constructors are executed implicitly, they can be invoked explicitly.
(6) Constructor can have default and can be overloaded.
(7) The constructor without arguments is called as default constructor.

SYNTAX

Constructor has the same name as that of the class and it does not have any return type. Also, the constructor is always public.

```
... .. ...
class temporary
{
private:
        int x;
        float y;
public:
        // Constructor
        temporary(): x(5), y(5.5)
        {
                // Body of constructor
        }
        ... ..  ...
};

int main()
{
        Temporary t1;
        ... .. ...
}
```

Above program shows a constructor is defined without a return type and the same name as the class.

```
#include<iostream>

usingnamespacestd;

classLine
{
public:
voidsetLength(doublelen);
doublegetLength(void);
Line();// This is the constructor
private:
double length;
};

// Member functions definitions including constructor

Line::Line(void)
{
cout<<"Object is being created"<<endl;
}
voidLine::setLength(doublelen)
{
length=len;
}
doubleLine::getLength(void)
{
return length;
}

// Main function for the program
int main()
{
Lineline;

// set line length
line.setLength(6.0);
cout<<"Length of line : "<<line.getLength()<<endl;

return0;
}
```

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

```
Object is being created

Length of line : 6
```

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.

Example Constructor overloading

```
// Source Code to demonstrate the working of overloaded constructors
#include<iostream>
usingnamespacestd;

classArea
{
private:
int length;
int breadth;

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

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

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

intAreaCalculation()
{
return length * breadth;
}

voidDisplayArea(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);

return0;
}
```

For object A1, no argument is passed while creating the object.Thus, the constructor with no argument is invoked which initializes 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 initializes lengthto 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
```

copy constructor

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to −

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here −

Syntax

```
classname (constclassname&obj)
{
  // body of constructor
}
```

Here, **obj** is a reference to an object that is being used to initialize another object.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Example
{
  // Member Variable Declaration
int a, b;
public:
  //Normal Constructor with Argument
Example(int x, int y)
 {
    // Assign Values In Constructor
    a = x;
    b = y;
cout<< "\nIm Constructor";
  }
  //Copy Constructor with Obj Argument
Example(constExample&obj)
{
    // Assign Values In Constructor
    a = obj.a;
    b = obj.b;
cout<< "\nIm Copy Constructor";
  }
void Display()
 {
cout<< "\nValues :" << a << "\t" << b;
  }
};
int main()
{
  //Normal Constructor Invoked
  Example Object(10, 20);


  //Copy Constructor Invoked - Method 1
  Example Object2(Object);
  //Copy Constructor Invoked - Method 2
  Example Object3 = Object;
Object.Display();
Object2.Display();
Object3.Display();
  // Wait For Output Screen
getch();
return 0;
}
```

Sample Output

```
Im Constructor
Im Copy Constructor
Im Copy Constructor
Values :10     20
Values :10     20
Values :10     20
```

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 features of Object Oriented Programming.

OR

The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.

Important points

- In the inheritance the class which is give data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class

Advantage of inheritance

If we develop any application using this concept than that application have following advantages,

- Application development time is less.
- Application takes less memory.
- Application execution time is less.
- Application performance is enhance (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistence results and less storage cost.

Types of Inheritance

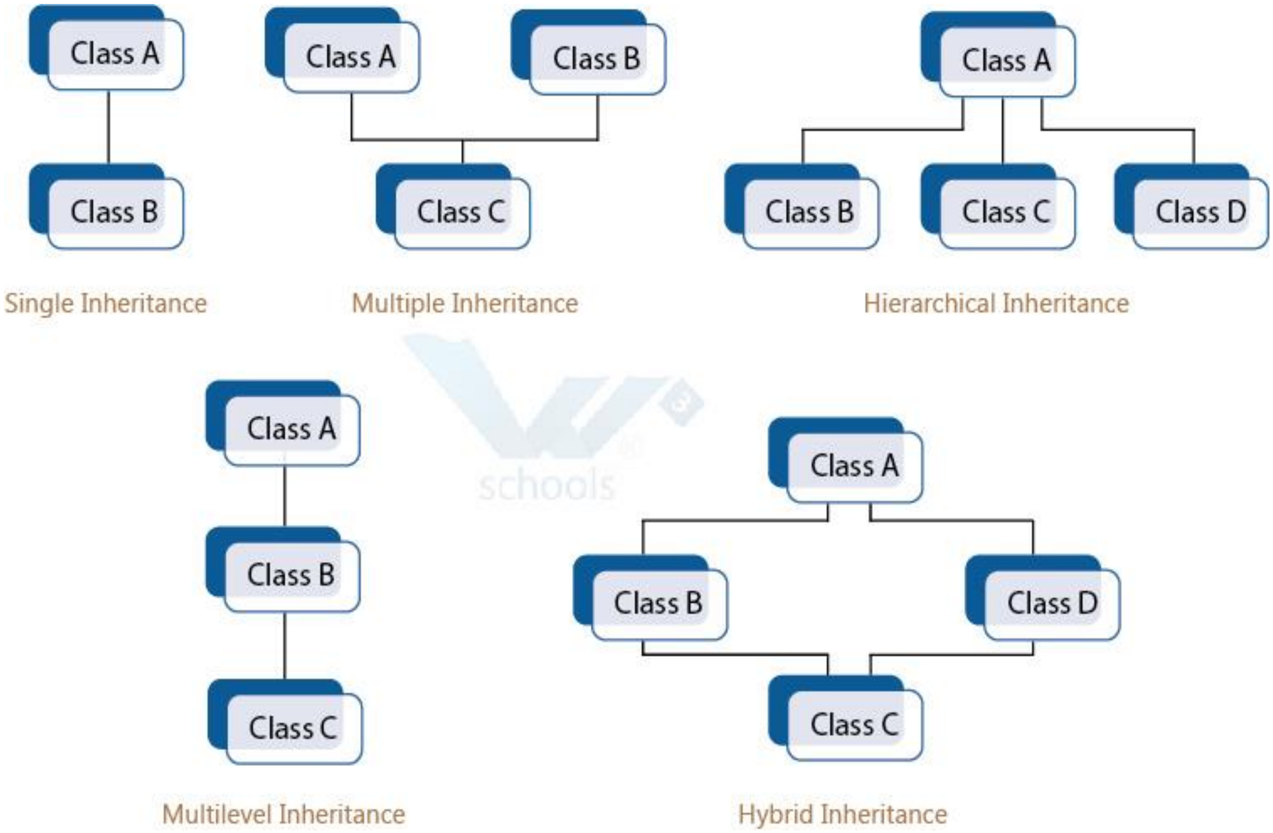
Based on number of ways inheriting the feature of base class into derived class it have five types they are:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

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.

Modes of Inheritance(Access Specifiers)

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.



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:

```
classsubclass_name : access_modebase_class
{
  //body of subclass
};
```

```
// C++ program to explain
// Single inheritance
#include <iostream>
usingnamespacestd;

// base class
Class Vehicle
{
  public:
    Vehicle( )
    {
      cout<< "This is a Vehicle"<<endl;
    }
};

// sub class derived from two base classes
classCar: publicVehicle
{

};

// main function
intmain ( )
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Run on IDE

Output:

```
This is a vehicle
```

Multiple Inheritances:

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:

```
classsubclass_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>
usingnamespacestd;

// first base class
ClassVehicle
 {
  public:
    Vehicle()
    {
      cout<< "This is a Vehicle"<<endl;
    }
};

// second base class
ClassFourWheeler
{
  public:
    FourWheeler()
    {
      cout<< "This is a 4 wheeler Vehicle"<<endl;
    }
};

// sub class derived from two base classes

Class Car: publicVehicle, publicFourWheeler
 {

};

// main function
Intmain()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return0;
}
```

Run on IDE

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>
Usingnamespacestd;

// base class
ClassVehicle
{
 public:
   Vehicle()
   {
     cout<< "This is a Vehicle"<<endl;
   }
};
ClassfourWheeler: publicVehicle
{
public:
   fourWheeler()
   {
     cout<<"Objects with 4 wheels are vehicles"<<endl;
   }
};
// sub class derived from two base classes
ClassCar: publicfourWheeler
{
  public:
    car()
    {
      cout<<"Car has 4 Wheels"<<endl;
    }
};

// main function
Intmain()
{
   //creating object of sub class will
   //invoke the constructor of base classes
   Car obj;
   return0;
}
```

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>
usingnamespacestd;

// base class
ClassVehicle
{
  public:
    Vehicle()
    {
      cout<< "This is a Vehicle"<<endl;
    }
};


// first sub class
classCar: publicVehicle
{

};

// second sub class
ClassBus: publicVehicle
{

};

// main function
intmain()
```

```
{
   // creating object of sub class will
   // invoke the constructor of base class
   Car obj1;
   Bus obj2;
   return0;

}
```

Run on IDE

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
```

Exception Handling in C++

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

Run Time Errors - They are also known as exceptions. An exception caught during run time creates serious issues.

Errors hinder normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling technics in our code.

In C++, Error handling is done by three keywords:-

- Try
- Catch

- Throw

Syntax:

```
Try
{
//code
throw parameter;
}
catch(exceptionname ex)
{
//code to handle exception
}
```

Try

Try block is intended to throw exceptions, which is followed by catch blocks. Only one try block.

Catch

Catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks.

Throw

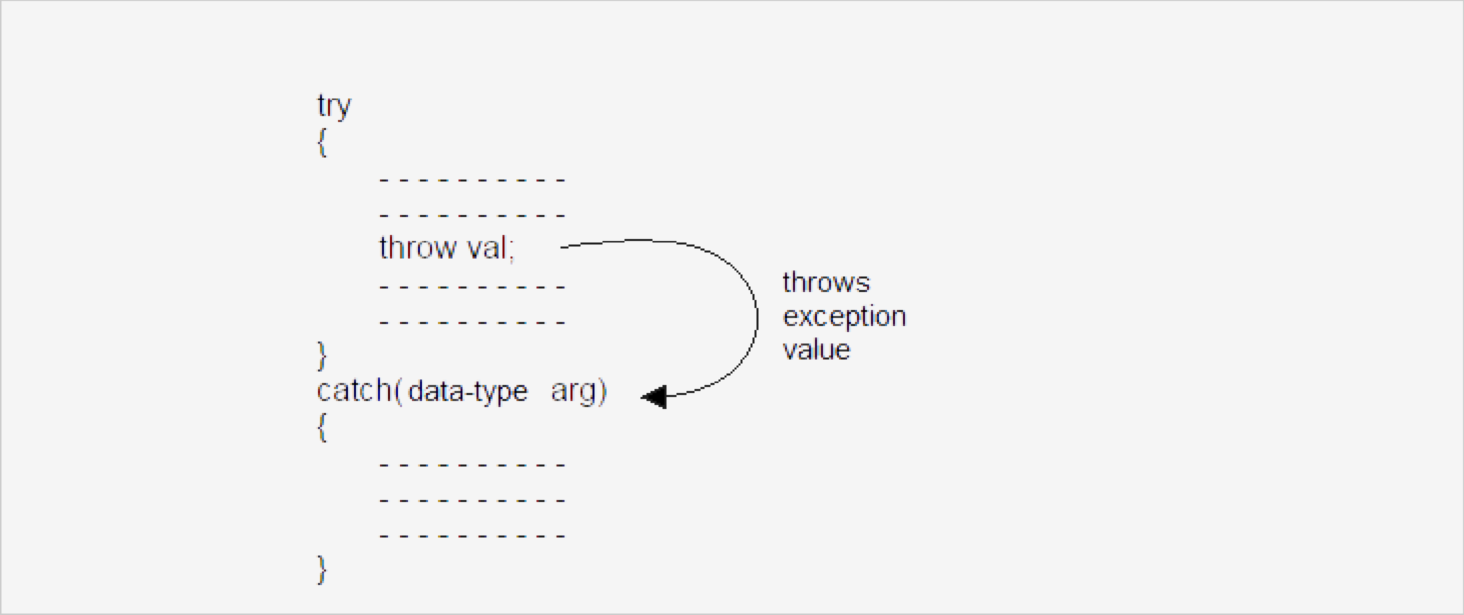
It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.

Example of Exception

Below program compiles successful but the program fails during run time.

```
#include <iostream>
#include<conio.h>
using namespace std;
intmain()
{
int a=10,b=0,c;
c=a/b;
return0;
}
```

Implementation of try-catch, throw statement



```
try
{
      - - - - - - - - - - -
      - - - - - - - - - - -
      throw val;
      - - - - - - - - - - -
      - - - - - - - - - - -
}
catch( data-type   arg)
{
      - - - - - - - - - - -
      - - - - - - - - - - -
      - - - - - - - - - - -
}
```

Example of simple try-throw-catch

```
     #include<iostream.h>
     #include<conio.h>
void main()
     {
int n1,n2,result;

cout<<"\nEnter 1st number : ";
cin>>n1;

cout<<"\nEnter 2nd number : ";
cin>>n2;

try
         {
if(n2==0)
throw n2;       //Statement 1
else
            {
result = n1 / n2;
cout<<"\nThe result is : "<<result;
            }
         }
catch(int x)
         {
cout<<"\nCan't divide by : "<<x;
```

```
        }

cout<<"\nEnd of program.";

    }

Output :

        Enter 1st number : 45
        Enter 2nd number : 0
        Can't divide by : 0
        End of program
```

The catch block contain the code to handle exception. The catch block is similar to function definition.

```
catch(data-type arg)
        {
            - - - - - - - - - -
            - - - - - - - - - -
            - - - - - - - - - -
        };
```

Data-type specifies the type of exception that catch block will handle, Catch block will recieve value, send by throw keyword in try block.

A **single try statement** can have multiple catch statements. Execution of particular catch block depends on the type of exception thrown by the throw keyword. If throw keyword send exception of integer type, catch block with integer parameter will get execute.

Example of multiple catch blocks

```
    #include<iostream.h>
    #include<conio.h>
void main()
    {
int a=2;

try
        {

if(a==1)
throw a;                //throwing integer exception

else if(a==2)
throw 'A';              //throwing character exception

else if(a==3)
throw 4.5;              //throwing float exception

          }
catch(int a)
          {
cout<<"\nInteger exception caught.";
          }
catch(char ch)
          {
cout<<"\nCharacter exception caught.";
          }
catch(double d)
          {
cout<<"\nDouble exception caught.";
          }

cout<<"\nEnd of program.";

     }

Output :

          Character exception caught.
          End of program.
```

The above example will caught only three types of exceptions that are integer, character and double. If an exception occur of long type, no catch block will get execute and abnormal program termination will occur. To avoid this, We can use the catch statement with three dots as parameter (...) so that it can handle all types of exceptions.

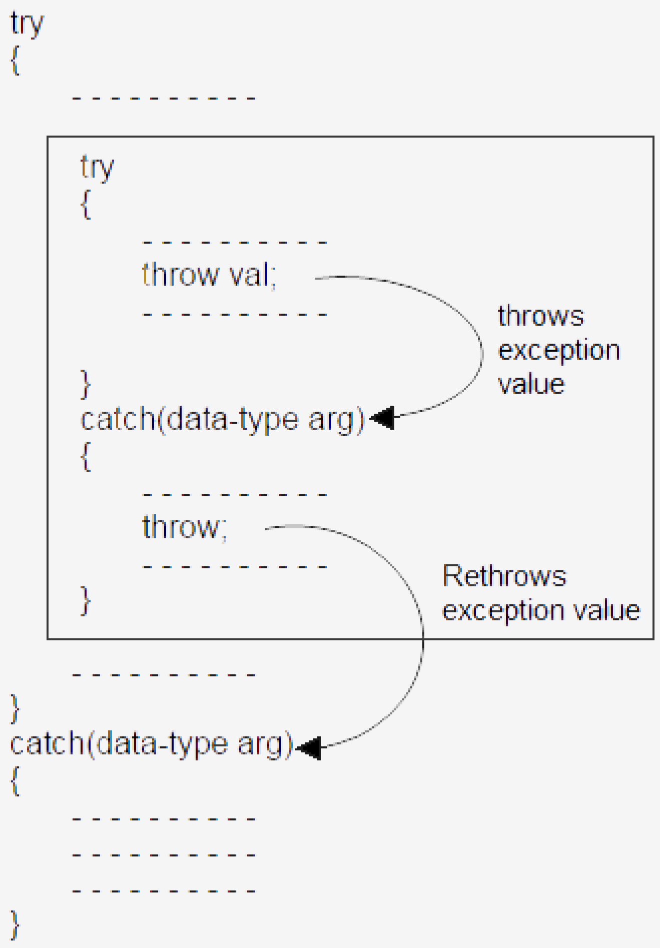
Example to catch all exceptions

```
     #include<iostream.h>
     #include<conio.h>
void main()
     {
int a=1;

try
```

```
            {

if(a==1)
throw a;                //throwing integer exception

else if(a==2)
throw 'A';              //throwing character exception

else if(a==3)
throw 4.5;              //throwing float exception

            }
catch(...)
            {
cout<<"\nException occur.";
            }

cout<<"\nEnd of program.";

      }

Output :

Exception occur.
            End of program.
```

Rethrowing exception is possible, where we have an inner and outer try-catch statements (Nested try-catch). An exception to be thrown from inner catch block to outer catch block is called rethrowing exception.

Syntax of rethrowing exceptions



```
try
{
    - - - - - - - - - -
    try
    {
        - - - - - - - - - -
        throw val;
        - - - - - - - - - -

    }
    catch(data-type arg)
    {
        - - - - - - - - - -
        throw;
        - - - - - - - - - -
    }
    - - - - - - - - - -
}
catch(data-type arg)
{
    - - - - - - - - - -
    - - - - - - - - - -
    - - - - - - - - - -
}
```

Example of rethrowing exceptions

```

    #include<iostream.h>
    #include<conio.h>
void main()
    {
int a=1;

try
        {
try
            {
throw a;
            }
catch(int x)
            {
cout<<"\nException in inner try-catch block.";

throw x;
            }
            }
catch(int n)
            {
cout<<"\nException in outer try-catch block.";
            }

cout<<"\nEnd of program.";

      }

Output :

Exception in inner try-catch block.
Exception in outer try-catch block.
            End of program.
```

We can restrict the type of exception to be thrown, from a function to its calling statement, by adding throw keyword to a function definition.

Example of restricting exceptions

```
      #include<iostream.h>
      #include<conio.h>

void Demo() throw(int ,double)
      {
int a=2;

if(a==1)
throw a;               //throwing integer exception

else if(a==2)
throw 'A';             //throwing character exception

else if(a==3)
throw 4.5;             //throwing float exception

      }

void main()
      {

try
            {
```

```
Demo();
            }
catch(int n)
            {
cout<<"\nException caught.";
            }

cout<<"\nEnd of program.";

      }
```

The above program will abort because we have restricted the Demo() function to throw only integer and double type exceptions and Demo() is throwing character type exception.

KARPAGAM ACADEMY OF HIGHER EDUCATION
COIMBATORE - 21

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

CLASS : I B.Sc COMPUTER SCIENCE

BATCH : 2019-2022

Part -A Online Examinations **(1 mark questions)**

SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++ **SUBJECT CODE: 19CSU101**

UNIT-V

	Questions	OPT1	OPT2	OPT3	OPT4	Answer
1	C++ supports all the features of ___________ as defined in C	structures	union	objects	classes	structures
2	A structure can have both variable and functions as	objects	classes	members	arguments	members
3	The class _________ describes the type and scope of its members	calling function	declaration	objects	none of the above	declaration
4	The class __________ describes how the class function are implemented	Function definition	declaration	arguments	none of the above	Function definition
5	The keywords private and public are known as _________ labels	Static	dynamic	visibility	const	visibility
6	The class members that have been declared as ________ can be accessed only from within the class	Private	public	static	protected	Private
7	The class members that have been declared as ________ can be accessed from outside the class also	Private	Public	static	protected	Public
8	The variables declared inside the class are called as _________	Function variables	data members	member function	data variables	data members
9	The functions which are declared inside the class are known as ______	Member function	member variables	data variables	function overloading	Member function
10	The class variables are known as ________	Functions	members	objects	none of the above	objects

11	The symbol ______ is called the scope resolution operator	>>	::	<<	::*	::
12	__________ enables an object to initialize itself when it is created	Destructor	constructor	overloading	none of the above	constructor
13	The __________ is special because its name is the same as the class name.	Destructor	static	constructor	none of the above	constructor
14	A constructor that accepts no parameters is called the __________ constructor	Copy	default	multiple	none of the above	default
15	Constructors are invoked automatically when the ________ are created	Datas	classes	objects	none of the above	objects
16	Constructors cannot be _________	Inherited	destroyed	both a & b	none of the above	Inherited
17	Constructors cannot be _________	Destroyed	virtual	both a & b	none of the above	virtual
18	Constructors make _________ calls to the operators new and delete when memory allocation is required	Explicit	implicit	function	none of the above	implicit
19	The constructors that can take arguments are called _________ constructors	Copy	multiple	parameterized	none of the above	parameterized
20	The constructor function can also be defined as ________ function	Friend	inline	default	none of the above	inline
21	When a constructor can accept a reference to its own class as a parameter, in such cases it is called as _________ constructors	Multiple	copy	default	none of the above	copy
22	When more than one constructor function is defined in a class, then the constructor is said to be _________	Multiple	copy	default	overloaded	overloaded
23	C++ compiler has a _________ constructor, which creates objects, even though it was not defined in the class.	Explicit	default	implicit	none of the above	implicit
24	A _________ constructor is used to declare and initialize an object from another object	Default	copy	multiple	parameterized	copy
25	The process of initializing through a copy constructor is known as ________ initialization	Overloaded	multiple	copy	none of the above	copy

26	_________ is used to free the memory	new	delete	clrscr()	none of the above	delete
27	Which is a valid method for accessing the first element of the array item?	item(1)	item[1]	item[0]	item(0)	item[0]
28	Which of the following statements is valid array declaration?	int number (5);	float avg[5];	double [5] marks;	counter int[5];	float avg[5];
29	An object is an _________ unit	group	individual	both a&b	none of the above	individual
30	Public keyword is terminated by a ________	Semicolon	comma	dot	colon	colon
31	Private keyword is terminated by a _________	semicolon	comma	dot	colon	colon
32	The memory for static data is allocated only ________	twice	thrice	once	none of the above	once
33	Static member functions can be invoked using ________ name	class	object	data	function	class
34	When a class is declared inside a function they are called as ________ classes.	global	invalid	local	none of the above	local
35	_________ releases memory space occupied by the objects	constructor	destructor	both a & b	none of the above	destructor
36	Constructors and destructors are automatically inkoved by _________	operating system	main()	complier	object	complier
37	Constructors is executed when ________	object is destroyed	object is declared	both a & b	none of the above	object is declared
38	The destructor is executed when __________	object goes out of scope	when object is not used	when object contains nothing	none of the above	object goes out of scope
39	The members of a class are by default ________	protected	private	public	none of the above	2
40	The ________ is executed at the end of the function when objects are of no used or goes out of scope	destructor	constructor	inheritance	none of the above	destructor
41	The statement catches the exception ________ .	catch	try	template	throw.	catch

42	In a multiple catch statement the number of throw statements are .	same as catch	twice than catch	only one	none.	only one
43	The exception is generated in _________block.	try	catch	finally	throw.	try
44	The exception handling one of the function is implicitly invoked.	abort	exit	assert	none.	abort
45	The exception handling mechanism is basically built upon ______ keyword	try	catch	throw	all the above	all the above
46	The point at which the throw is executed is called _________.	try	catch	throw point	exceptions	throw point
47	A template function may be overloaded by _______ function	template	normal	stream	exception	template
48	________function returns true when an input or output operation has failed	eof()	fail()	bad()	good()	fail()
49	.In ________ inheritance, the base classes are constructed in the order in which they appear in	Hybrid	Multipath	Hierarchical	Multiple	Multiple
50	The ________ function takes no operator.	Operator +()	Operator –()	Friend	Conversion	operator -( )
51	In overloading of binary operators, the ________ operand is used to invoke the operator function.	Right-hand	Arithmetic	Left-hand	Multiplicatio	left-hand
52	________ functions may be used in place of member functions for overloading a binary	Inline	Member	Conversion	Friend	Friend
53	The operator that cannot be overloaded is	Sizee of	+	-	=	single of
54	The friend functions cannot be used to overload the operator.	::	?:	.	=	::
55	________ is called compile time polymorphism.	Operator overloading	Function overloading	Overloading unary operator	Overloading	operator overloading
56	________ feature can be used to add two user-defined operator data types.	Function	Overloading	Arrays	Pointers	overloading
57	________ operator cannot be overloaded.	=	+	?:	–	?:

58	Operator overloading is done with the help of a special function called function.	Conversion	Operator	User-defined	In-built.	operator
59	________ functions must either be member functions or friend functions.	Operator	User-defined	Static Member	Overloading	operator
60	The overloading operator must have atleast operand that is of user-defined data	Two	Three	One	Four	one
61	________ operator function should be a class member.	Arithmetic	Relational	Casting	Overloading	casting
62	The casting operator must not have any	Arguments	Member	Return type	Operator	arguments
63	The casting operator function must not specify a type.	User-defined type	Return	Member	In-built	return
64	The operator that cannot be overloaded is .	Casting	Binary	Unary	Scope resolution	scope resolution
65	The friend function cannot be used to overload operator.	+	-	( )	::	( )
66	________ operator cannot be overloaded by friend function.	[]	*	.	?:	?:
67	The operator that cannot be overloaded by friend function is	.	::	->	Single of	::
68	Operator overloading is called ________	Function Overloading	Compile time	Casting operator	Temporary object	Compile time polymorphism
69	Overloading feature can add two ________ data types.	In-built	Enumerated	User-defined	Static	User-defined
70	The mechanism of deriving a new class from an old one is called	Operator overloading	Inheritance	Polymorphism	Access mechanism	polymorphism