UNIT 1 - INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

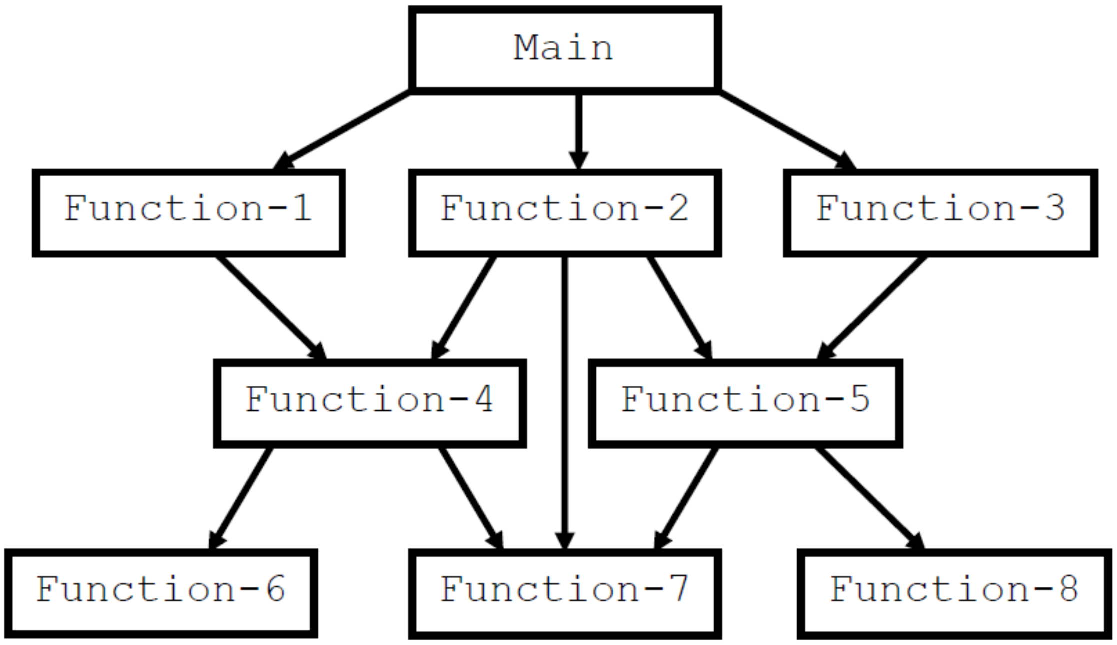
INTRODUCTION TO OBJECT ORIENTED PROGRAM

Procedure Oriented Programming (POP)

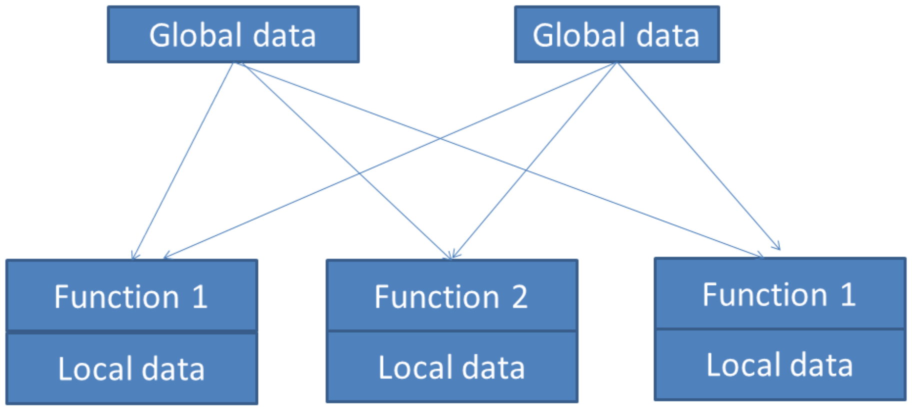
The high level languages, such as BASIC, COBOL, C, FORTRAN are commonly known as Procedure Oriented Programming.

Using this approach, the problem is viewed in sequence of things to be done, like reading, processing and displaying or printing. To carry out these tasks the function concepts must be used.

- ♦ This concept basically consists of number of statements and these statements are organized or grouped into functions.

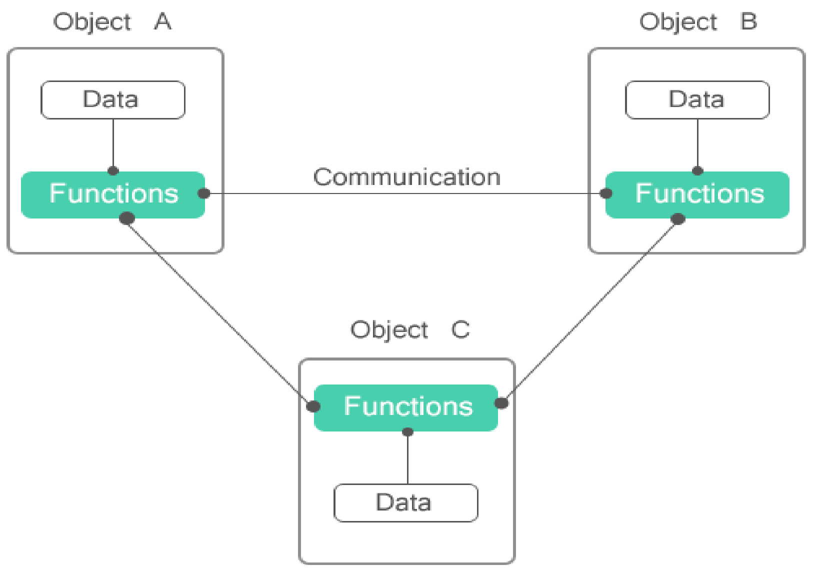


- ♦ While developing these functions the programmer must care about the data that is being used in various functions.
- ♦ A multi-function program, the data must be declared as global, so that data can be accessed by all the functions within the program & each function can also have its own data called local data.
- ♦ The global data can be accessed anywhere in the program. In large program it is very difficult to identify what data is accessed by which function. In this case we must revised about the external data and as well as the functions that access the global data. At this situation there is so many chances for an error.



Object Oriented Programming (OOP)

- ♦ This programming approach is developed to reduce the some of the drawbacks encountered in the Procedure Oriented Programming Approach.
- ♦ The OO Programming approach treats data as critical element and does not allow the data to freely around the program.
- ♦ It bundles the data more closely to the functions that operate on it; it also protects data from the accidental modification from outside the function.
- ♦ The object oriented programming approach divides the program into number of entities called objects and builds the data and functions that operates on data around the objects.
- ♦ The data of an object can only access by the functions associated with that object.



Comparison of Programming Paradigms

S. No.	Procedure Oriented Programming (C)	Object Oriented Programming (C++)
1.	Programs are divided into smaller Sub-programs known as functions.	Programs are divides into objects & Classes.
2.	New data and functions can be added easily.	New data and functions can be added easily.
3.	It is a Top-Down Approach	It is a Bottom-Up Approach
4.	Data cannot be secured and available to all the function	Data can be secured and can be available in the class in which it is declared.
5.	Here, the reusability is not possible, hence redundant code cannot be avoided.	Here, We can reuse the existing one using the Inheritance concept.
6.	It does not have any access specifier.	It has access specifiers named Public, Private, Protected, etc.
7.	Data can move freely from function to function in the system.	Objects can move and communicate with each other through member functions.
8.	Overloading is not possible.	Overloading is possible in the form of Function Overloading and Operator Overloading.
9.	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, Small talk.

Concepts of OOPS

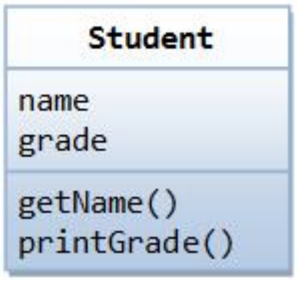
The general concepts of OOPS comprises the following.

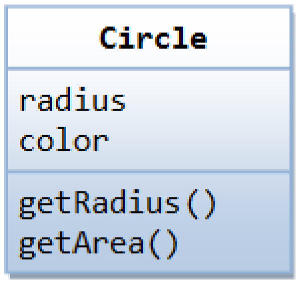
1. Object
2. Class
3. Data abstraction
4. Inheritance
5. Polymorphism
6. Dynamic Binding
7. Message passing.

1. Object

Object is an entity that can store data and, send and receive messages. They are runtime entities; they may represent a person, a place a bank account, a table of data or any item that the program must handle. It is an instance of a class.

They may also represent user-defined data such as vectors, time and lists. When a program is executed, the object interacts by sending messages to one another. Each object contain data and code to manipulate the data objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.





2. Classes

A class is a collection of objects of similar type. Classes are user defined data types and behave like the built in types of a programming language. For example mango, apple and orange are members of the class fruit. Then the statement FRUIT MANGO; will create an object mango belonging to the class fruit. The syntax used to create an object is no different than the syntax used to create an integer object in C. If **fruit** has been defined as a class, then the statement

fruit mango;

will create an object **mango** belonging to the class **fruit**.

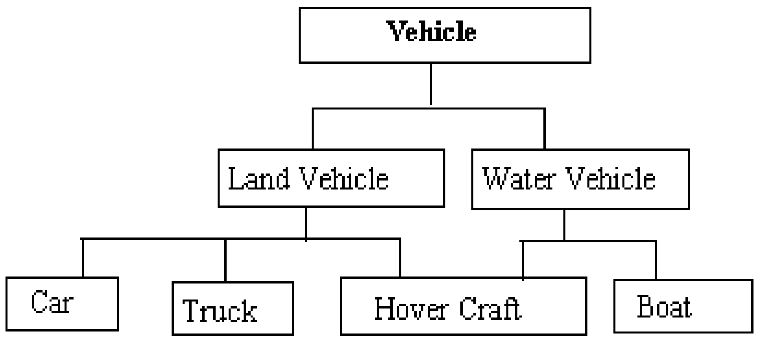
3. **Data abstraction and encapsulation:**

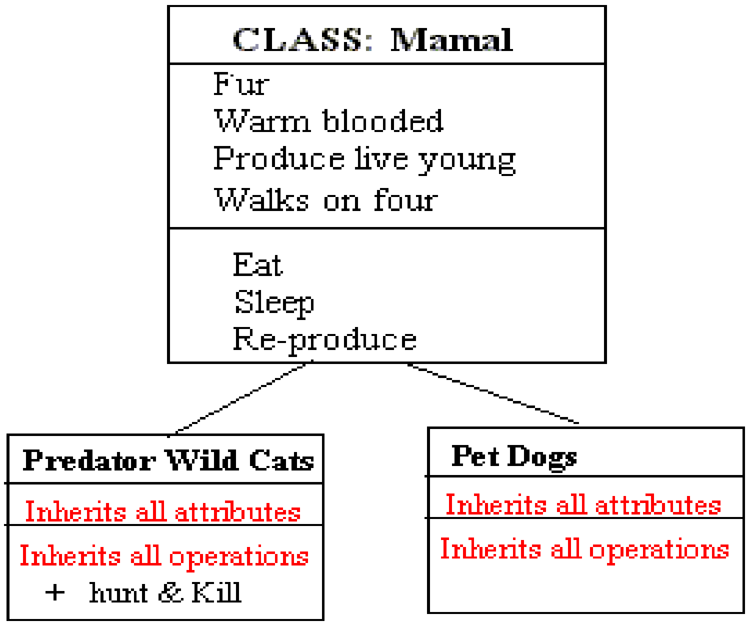
The wrapping up of data and its functions into a single unit (class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can assess it> these functions provide the interface between the objects data and the program> this insulation of data from direct access by the program is called **DATA HIDING (or data abstraction).** Since the classes use the concept of data abstraction they are known as ABSTRACT DATA TYPES (ADT)

4. **Inheritance:**

In heritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example the bird **robin** is a part of the class **flying birds** which again a part of **bird**. As given in the diagram below each derived class shares common characteristics with the class from which it is derived.

In **OOP**, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants.

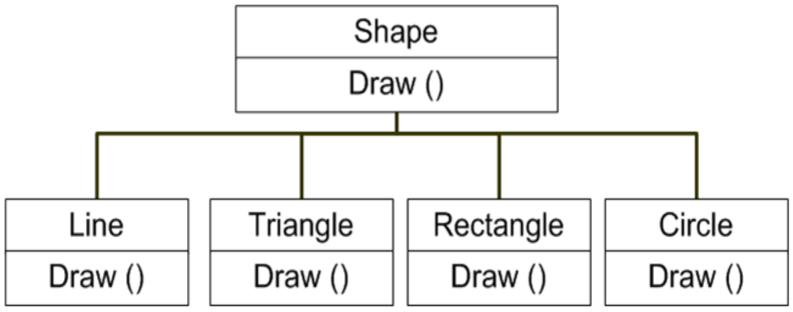




5. Polymorphism:

Polymorphism means the ability to take more than one form. For example an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example consider the operation addition. For two numbers, the operation will generate a sum . if the operands are strings, then the operation would produce a third string by concatenation.

Here in the below given diagram a single function draw () does different operation according to the behavior of the type derived. I.e. Draw () function works in different form.



Polymorphism plays an important role in allowing objects having internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in inheritance.

6. Dynamic Binding

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call is associated with a polymorphic reference depends on the dynamic type of that reference.

7. Message Communication

An object oriented program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Benefits or Advantages of OOPS

- The complexity of software can be managed easily.
- Data hiding concept help the programmer to build secure programs
- Through the class concept we can define the user defined data type
- The inheritance concept can be used to eliminate redundant code
- The message-passing concept helps the programmer to communicate between different objects.
- New data and functions can be easily added whenever necessary.
- OOPS ties data elements more closely to the functions that operates on.

Basics of C++ Programming:

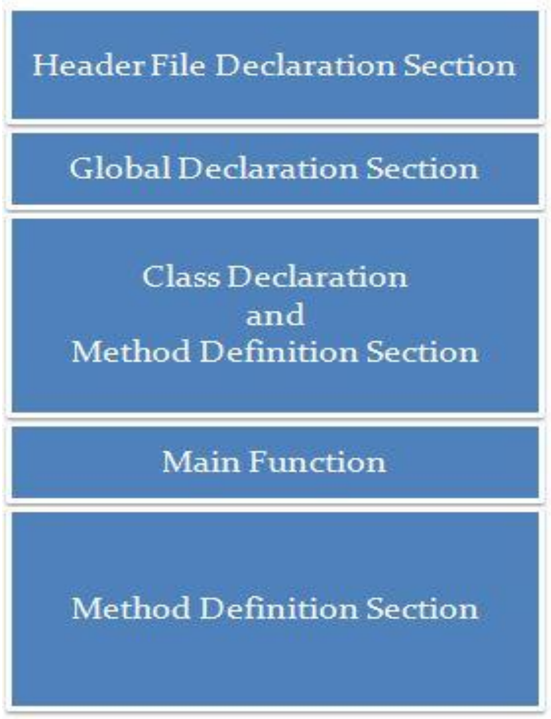
C++ was developed by BJARNE STROUSSTRUP at AT&T BELL Laboratories in Murry Hill, USA in early 1980's.

Strousstrup combines the features of 'C' language and 'SIMULA67' to create more powerful language that support OOPS concepts, and that language was named as "C with CLASSES". In late 1983, the name got changed to C++.

The idea of C++ comes from ‘C’ language increment operator (++) means more additions.

C++ is the superset of ‘C’ language or extension of ‘C’ language, most of the ‘C’ language features can also applied to C++, but the object oriented features (Classes, Inheritance, Polymorphism, Overloading) makes the C++ truly as Object Oriented Programming language.

Structure of C++ Program


Header File Declaration Section
Global Declaration Section
Class Declaration and Method Definition Section
Main Function
Method Definition Section

Section 1 : Header File Declaration Section

1. Header files used in the program are listed in this section.
2. Header File provides Prototype declaration for different library functions.
3. We can also include user define header file.
4. Basically all preprocessor directives are written in this section.

Include files provides instructions to the compiler to link functions from the system library.

Eg: #include <iostream.h>

#include	–	Preprocessor Directive
iostream.h	–	Header File

Section 2 : Global Declaration Section

1. Global Variables are declared here.
2. Global Declaration may include
    - Declaring Structure
    - Declaring Class
    - Declaring Variable

Section 3 : Class Declaration Section

1. Actually this section can be considered as sub section for the global declaration section.
2. Class declaration and all methods of that class are defined here.
3. A class is a way to bind and its associated functions together. It is a user defined datatype. It must be declared at class declaration part.

Section 4 : Main Function

1. Each and every C++ program always starts with main function.
2. This is entry point for all the function. Each and every method is called indirectly through main.
3. We can create class objects in the main.
4. Operating system call this function automatically.

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

Program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical and of the program.

Section 5 : Method Definition Section

1. This is optional section . Generally this method was used in C Programming.
2. Member function definition describes how the class functions are implemented. This must be the next part of the C++ program.

Input / Output Statements

Input Stream

Syntax:

```
cin >> var1 >> var2 >>;
```

cin is a keyword, it is an object, predefined in C++ to correspond to the standard input stream.

>> is the extraction or get from operator. Extraction operation (>>) takes the value from the stream object on its left and places it in the variable on its right.

Eg:

```
cin>>x;
cin>>a>>b>>c;
```

Output Stream:

Syntax:

```
cout<<var1<<var2;
```

cout is a keyword, predefined object of standard output stream

<< is called the insertion or put to operator. It directs the contents of the variable on its right to the object on its left. Output stream can be used to display messages on output screen.

Eg:

```
cout<<a<<b; cout<<”value of x is”<<x;
cout<<”Value of x is”<<x<<”less than”<<y;
```

Tokens

The smallest individual units in a program are known as tokens. C++ has the following tokens

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords

- It has a predefined meaning and cannot be changed by the user
- Keywords cannot be used as names for the program variables.

Keywords supported by C++ are:

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

The specific C++ Keywords

There are several keywords specific to C++

asn	catch	class	delete
friend	inline	new	operator
private	protected	public	template
this	throw	try	virtual

Identifiers

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer.

Rules for naming these identifiers:

1. Only alphabetic characters, digits and underscores are permitted.
2. The name cannot start with a digit.
3. Uppercase and lowercase letters are distinct.
4. A declared keyword cannot be used as a variable name.

(i) Variables:

It is an entity whose value can be changed during program execution and is known to the program by a name.

A variable can hold only one value at a time during program execution.

Eg:

Allowable variable names	Invalid names	
i	1_B	– 1st letter must be alphabet
sum	$xy	- 1st letter must be alphabet
A_B	x+b	- special symbol '+' not allowed
A-1B		

Declaration of Variables

Syntax

datatype variablename;

There are 2 features of variables which are supported by C++. They are,

(i) Dynamic Initialization of Variable

(ii) Reference Variable

(i) Dynamic initialization of variable

In C language, al the variables should be declared only at the beginning of the program. But in C ++, they can be initialized wherever it is necessary in the program.

Eg. Program:

```
void main()
{
  int sum=0;
for(int i=1;i<=10;i++)
{
     sum=sum+i;
}
int avg=sum/10;                    //Dynamic initialization of avg
cout<<"Sum is"<<sum;
cout<<"Average is"<<avg;
}
```

Output:

```
Sum is 55
Average is 5.5
```

(ii) Reference Variable:

A reference variable provides an alias (alternative name) for a previously defined variable.

For example, if we make the variable total a reference to the variable sum, then total & sum can be used interchangeably to represent that variable.

A reference variable is created as follows:

datatype & ref_name = var_name;

Eg:

```
float sum = 100;
float  &  total  =  sum;
cout << sum << total;
```

Both the variables refer to the same data object in the memory

i.e. total sum

$\boxed{100}$

<u>Eg. Program:</u>

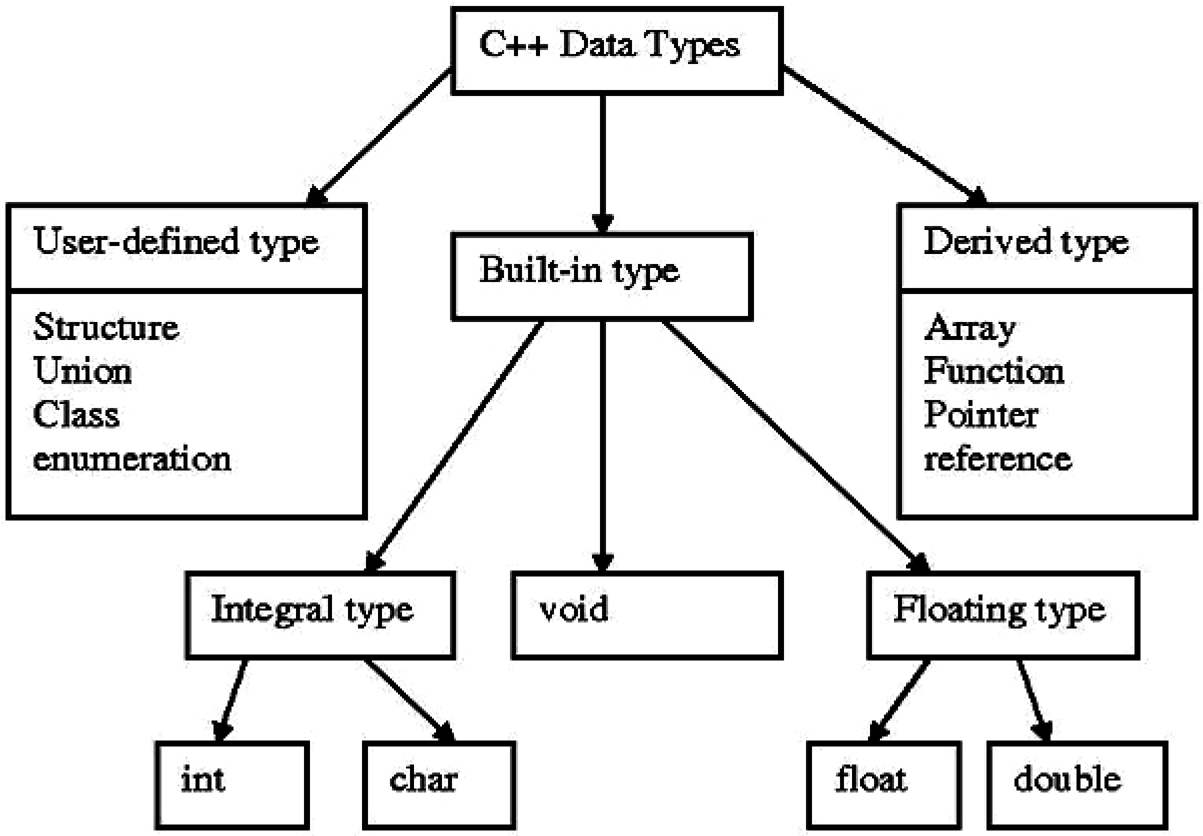
```
void main()
{
  int sum=0;
for(int i=1;i<=10;i++)
{
     sum=sum+i;
}
int avg=sum/10;          //Dynamic initialization of avg
int &total=sum;          //Alias name for sum is total
cout<<”Sum is”<<total;
cout<<”Average is”<<avg;
}
```

Output:

```
    Sum is 55
    Average is 5.5
```

Datatype

It is the type of data, that is going to be processed within the program. A variable can be declared anywhere in the program before its first use.



Type Conversion:

(i) Implicit type conversion

(ii) Explicit type conversion

Implicit type conversion

It will be done by the compiler, by following the rule of lower type converted to higher type.

Eg: int y = 10;

float z = 10.5,x;

x = y+z; (y is converted to float type by compiler) x = 10.0 + 10.5

x= 20.5 (result var. x is must be float)

Explicit type conversion

It will be performed by the programmer. According to the need of this in the program.

Syntax: datatype (var)

Eg: int y = 10;

float z = 2.5;(resultant type of y+z is float, that is converted explicitly to int type)

x = int (y + z);

Now the result is of int type.

Constants

A quantity that does not change is known as constants.

Types of constants:

- Integer constants- - Eg: 123, 25 – without decimal point
- Character constants - Eg: ‘A’, ‘B’, ‘*’, ‘1’
- Real constants - Eg: 12.3, 2.5 - with decimal point

Strings

A sequence of characters is called string. String constants are enclosed in double quotes as follows

“Hello”

Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.

Types of Operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment & decrement Operators
6. Conditional Operators
7. Bitwise Operators

8. Special Operators
9. Manipulators
10. Memory allocate / delete Operators

An expression is a combination of variables, constants and operators written according to the syntax of the language.

Arithmetic Operators

C++ has both unary & binary arithmetic operators.

- Unary operators are those, which operate on a single operand.
- Whereas, binary operators on two operands +, -, *, /, %

<u>Examples for Unary Operators:</u>

int x = 10;

int y = -x; (The value of x after negation is assigned to y ie. y becomes –10.)

int x = 5;

int sum = -x;

<u>Examples for binary Operators:</u>

int x = 16, y=5;

x+y = 21; (result of the arithmetic expression), x-y = 11; x*y=80;

/ - Division Operator

Eg:

x = 10, y = 3;

x/y=3; (The result is truncated, the decimal part is discarded.)

% - Modulo Division

The result is the remainder of the integer division only applicable for integer values.

x=11, y = 2 x%y = 1

Relational Operators

- A relational operator is used to make comparison between two expressions.
- All relational operators are binary and require two operands.
- <, <=, >, >=, ++, !=

Relational Expression

Expression1 relational operator Expression2

Expression1 & 2 – may be either constants or variables or arithmetic expression.

Eg:

a < b (Compares its left hand side operand with its right hand side operand)

10 = = 15

a != b

- ♦ An relational expression is always return either zero or 1, after evaluation.

Eg: (a+b) <= (c+d)

$\searrow$

arithmetic expression

Here relational operator compares the relation between arithmetic expressions.

Logical Operators

&&	-	Logical AND
!!	-	Logical OR
!	-	Logical NOT

Logical operators are used when we want to test more than one condition and make decisions.

Eg: (a<b) && (x= =10)

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression. Like simple relational expressions, a logical expression also yields a value of one or zero, according to the truth table.

Operand 1	Operand 2	AND	OR	NOT OP1	NOT OP2
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

Assignment Operators

Assignment operators are used to assign the result of an expression to a variable.

Eg: a = 10;

a = a + b; x = y;

Chained Assignment:

Syntax: Variable operator = operand

OP = is called as shorthand assignment operator.

VOP = exp

is equivalent to v = v op exp Eg: x+=y; $\Longrightarrow$ x = x+y;

Increment & Decrement Operators

- Increment ++, this operator adds 1 to the operand
- Decrement --, this operator subtracts 1 from the operand
- Both are unary operators

Eg:

m = 5;

y = ++m; (adds 1 to m value)

x = --m; (Subtracts 1 from the value of m)

Types

- Pre Increment / Decrement OP
- Post Increment / Decrement OP

If the operator precedes the operand, it is called pre increment or pre decrement. Eg: ++i, --i;

If the operator follows the operand, it is called post increment or post decrement. Eg: ++i, --i;

In the pre Increment / Decrement the operand will be altered in value before it is utilized for its purpose within the program.

Eg: x = 10; Y = ++x;

- 1st x value is getting incremented with 1.
- Then the incremented value is assigned to y.

In the post Increment / Decrement the value of the operand will be altered after it is utilized.

Eg: y = 11; x = y++;

- 1st x value is getting assigned to x & then the value of y is getting increased.

Conditional Operator

? :

General Form is

Conditional exp ? exp 1 : exp 2;

Conditional exp - either relational or logical expression is used. Exp1 & exp 2 : are may be either a variable or any statement.

Eg:

(a>b)?a:b;

- Conditional expression is evaluated first.
- If the result is '1' is true, then expression1 is evaluated.
- If the result is zero, then expression2 is evaluated.

Eg: lar = (10>5)?10:5;

Bitwise Operators

Used to perform operations in bit level

<u>Operators used</u>:

&	-	Bitwise AND
|	-	Bitwise OR
^	-	Exclusive OR
<<	-	Left shift
>>	-	Right shift
~	-	One's complement

Special Operators

- sizeof
- comma(,)

- size of operators returns the size the variable occupied from system memory.

Eg: var = sizeof(int)

cout<<var; **Ans: 2**

x = size of (float);

cout << x; **Ans: 4**

int y; x = sizeof (y);

cout<<y; **Ans: 2**

Precedence of Operators

Name	Operators	Associativity
Unary	-,++,--,!, sizeof	R → L
Mul, div & mod	*, /, %	L → R
Add, Sub	+, -	L → R
Relational	<. <=, >, >=	L → R
Equality	= =, !=	L → R
Logical AND	&&	L → R
Logical OR	||	L → R

Example

```
x     =10, y = 2m z = 10.5
a      = (x+y) – (x/y)*z;
       = 12 – 5 * 10.5;
       = 12 – 52.5;
a     = -42.5
```

Manipulators

Manipulators are operators used to format the data display. The commonly used manipulators are endl, setw.

endl manipulators

It is used in output statement causes a line feed to be inserted, it has the same effect as using the newline character “\n” in ‘C’ language.

```
#include <iostream.h>
main()
{
        int a=10, b=20;
        cout << “C++ language” <<
        endl; cout << “A value: “ << a
        << endl; cout << “B value:” <<
        b << endl;
}
```

O/P:

```
C++ language
A value: 10
B value: 20
```

Setw Manipulator

The setw manipulator is used or specify the field width for printing, the content of the variable. We need to include a header file "iomanip.h" in order to use setw in the program.

Syntax: **setw(width);**

where width specifies the field width. int a = 10;

cout << " A value" << setw(5) << a << endl;

Output:

```
A value    10
```

Note: Remaining operators will be discussed in 4th unit.

Symbolic Constant

Symbolic constants are constants to which symbolic names are associated with numbers for the purpose of readability and ease of handling.

1. #define preprocessor directive
2. const keyword
3. enumerated data type

1. #define preprocessor directive

It associates a constant value to a symbol and is visible throughout the function in which it is defined.

Syntax: **#define symbol name constant value**

Eg

```
#define     max_value
100 #define pi 3.14
```

The value of symbolic constant will not change throughout the program.

2. const keyword

Syntax: **const datatype var = constant;**

```
Eg 1:   const int max = 100;
main()
{
        char x[max];
        ---------
        ---------
}
```

Eg 2: const size = 10;

The above statement is valid; default symbolic constant type is integer. Here size is of type int.

3. Enumerated datatype

It assigns to value to variable of that type.

Syntax: enum enum-type-name {enum-list};

Eg.

enum colors {red,green,blue}; //Default values start from zero.

i.e) red=0,green=1,blue=2

enum shapes {circle=5,square,rectangle}; //Default values for enum list start from 5. i.e) circle=5,square=6,rectangle=7

Eg. Program:

```
enum week {sun,mon,tue,wed,thu,fri,sat};
void main()
{
  for(int i=sun;i<=sat;i++)
    cout<<i<<"\t";
}
```

Output:

```
    0      1      2      3      4      5      6
```

CONTROL INSTRUCTIONS

- In real world, several activities are sequenced, or repeated based on some decisions.
- Constructing control instructions can program such activities.

Types of control Instruction

1. Sequential Control Instruction
2. Selection Control Instruction
3. Loop Control Instruction
4. Case Control Instruction

SEQUENTIAL CONTROL INSTRUCTION

- Here instructions are executed sequential manner
- That is the same order in which they appear in the program.
- By default the instructions in a program are executed sequentially.

Example:

```
#include<iostream.h>
void main()
{
int a,b;
cout<<”Enter the value of a&b”;
cin>>a>>b;
int x=a+b;
cout<<”Sum of a & b is”<<x;
}
```

```
RUN
Enter the value of a& b
10 5
Sum of a & b is 15
```

In the above program instructions are executed one after another, in which they appear in the program.

SELECTION CONTROL INSTRUCTION

- Many times, we want a set of instructions to be executed in one situation, and an entirely different set of instruction to be executed in another situation.
- This kind of situation is dealt in C++ by constructing selection control instructions.

The following Statements are supporting to construct selection control instructions: 1. simple if statement

2. if-else statement
3. Nested if- else statement
4. Else-if statement

1. Simple if statement(one way decision stmt):

It is a powerful decision making statement, which is used to control the sequence of the execution of statements.

Syntax:

```
If(test expression)
{
        statement Block;
}
statement-x;
```

Execution Procedure:

The statement Block may be a single statement or a group of statements. If the test expression is true, the statement block is executed; Otherwise the statement block will be skipped and the execution will jump to the statement x. Remember when the condition is true both the statement block and statement x are executed in sequence.

Example:

```
if( category ==”sports”)
{
        marks = marks + bouns_marks;
}
cout<<marks;
```

The program tests the type of category of the student. If the student belongs to the SPORTS category, then additional bouns_marks are added to his marks befor they are printed. For others, bouns_marks are not added.

2. The if-else statement (two way decision stmt)

It performs some action even when the test expression fails.

Syntax:

```
if( test expression)
{
        true block statements;
}
else
{
        false block statements;
{
statement x;
```

Execution Procedure:

If the test expression is true, then the true block statement, immediately following the statement are executed; otherwise, the false block statements are executed. In either case, either true or false block will be executed, not both. In both the cases, the control is transferred subsequently to statement x.

Example:

```
# include<iostream.h>
void main()
{
int age;
cout<<"Enter your age"; cin>>age;
if((age>12)&&(age<20))
{
cout<<"you are a teen aged person":
}
else
{
cout<<"You are not a teen aged person";
}
cout<<"Program terminated";
}
```

```
Run1
Enter your age 16
You are a teen aged person Program Terminated
Run 2
Enter your age 23
You are not teen aged person Program Terminated
```

3. Nesting of if else statements(Multi way decision stmt)

When a series of decisions involved, we may have to use more than on if – else statement is nested form as follows:

```
if(test condition 1)
{
        if( test condition 2)
        {
                statement 1;
        }
        else
        {
                statement 2;
        }
}
else
{
        statement 3;
}
Statement x;
```

Execution Procedure:

If the condition 1 is false, the statement 3 will be executed; otherwise it continues to perform the second test. If the condition 2 is true, the statement –1 will bee evaluated; otherwise statement 2 will be executed and then the control, is transferred to statement x.

Example: Program to find largest of 3 nos

```
#Include<iostream.h>
void main()
{
int a,b,c;
cout<<" three nos";
cin>>a>>b>>c;
If(a>b)
{
if(a>c)
{
cout<<"a is greatest";
}
else
{
cout<<"c is greatest";
}
}
else
{
if(b>c)
{
cout<<"b is greatest";
}
else
{
```

```
cout<<"c is greatest";
}
}
Run:
Enter 3 nos: 24 56 34
b is greatest
```

4. else – if statement

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if.

It takes the following general form:

```
If( test condition1)
{
statement1;
}
else if (test condition2)
{
statement 2;
}
else if(test condition3)
{
statement 3;
}
else
{
statement 4;
}
statement x;
```

Execution Procedure:

This construct is known as else if ladder. The condition evaluated from the top, downwards. As soon as the true condition is found, the statement associated with it is executed and the control is transferred to the statement x(skipping the rest of the ladder). When all the conditions become false, then the final else containing the statement will be executed. Let us consider an example of grading the student in an academic institution. The grading is done according to the following rules:

Average marks	Grade
80-100	Honours
60-79	First Division
50- 69	Second Division
40- 49	Third Division
0-39	Fail

This grading can be done using the else if ladder as follows:

```
#include<iostream.h>
void main()
{
int marks;
cout<<"enter ur marks";
cin>>marks;
If(marks>79)
Grade="Honours";
else if(marks>59)
Grade="First Division";
else if(marks>49)
Grade="Second Division";
else if(marks>39)
Grade="Third Division";
else
Grade="Fail";
cout<<Grade;
}
Run
Enter ur mark 67
First Division
```

LOOP CONTROL INSTRUCTIONS

- Loop cause a section of code to be executed repeatedly until a termination condition is met.
- A program loop therefore consist of two segments, one known as the body of the loop and the other known as the control statement.
- Control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

The following Statements are supporting to construct loop control instructions:

1. while statement
2. do - while statement
3. for statement

1. while statement(Entry controlled Loop)

While is used when the number of iterations to be performed is not known in advance.

Syntax:

```
While(test condition)
{
        body of the loop;
}
Statement x;
```

Execution Procedure:

The test condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop.

Example: Display 1----N numbers

```
#include<iostream.h>
void main()
{
int n;
cout<<"how many integers to be displayed";
cin>>n;
int I=1;
while(I<=n)
{
cout<<I<<endl;
I++;
}
cout<<"Program Terminated";
}
Run
How many integers to be displayed: 5
1 2 3 4 5
Program Terminated
```

PROBLEMS:

1. Write a C++ program to find reverse of a given number
2. Write a C++ program to find sum of the digits in a given number.
3. Write a C++ program to check the given no. is Armstrong or not.
4. Write a C++ program to check the given no. is Palindrome or not.

2. do-while statement (Exit controlled loop stmt)

- Some times, it is desirable to execute the body of a while loop only once,, even if the test expression evaluates to false during the first iteration.
- This requires testing of termination expression at the end of the loop rather than the beginning as in the while loops.
- So, the do- while loop is called bottom tested loop.
- The loop is executed as long as the test condition remains true.

Syntax:

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

statement x;

Execution Procedure:

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test condition in the while statement is evaluated. I the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true.

When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

Example: Display 1----N numbers

```
#include<iostream.h>
void main()
{
int n;
cout<<”how many integers to be displayed”;
cin>>n;
int I=1;
do
{
cout<<I<<endl;
I++;
}while(I<=n);
cout<<”Program Terminated”;
}
```

```
Run
How many integers to be displayed: 5
1 2 3 4 5
Program Terminated
```

3. for loop statement

For loop is useful while executing a statement a fixed number of times. For statement is the compact way to express a loop.

Syntax

```
for(initialization; condition; increment/decrement)
{
        Body of the loop;
}
statement x;
```

Execution Procedure:

The Initialization part is executed only once. Next the test condition is evaluated. If the test evaluates to false, then the next statement after the for loop is executed. If the test expression evaluates to true, then body of the loop is executed. After executing the body of the loop, the increment/ decrement part is executed. The test is evaluated again and the whole process is repeated as long as the test expression evaluates to true.

Example: display numbers 1…..n using for loop

```
#include<iostream.h>
void main()
{
int n;
cout<<"how many integers to be displayed";
cin>>n;
int I;
for(I=1;I<=n;I++)
{
cout<<I<<endl;
}
cout<<"Program Terminated";
}
```

```
Run
How many integers to be displayed: 5
1 2 3 4 5
Program Terminated
```

PROBLEMS:

1. Generate fibonacci series.
2. Check the given number is perfect or not.
3. Sum of 1+2+…..+n
4. Find factorial of a given number
5. Sum of 1+1/2+1/3+…..+1/n
6. Sum of 1+1/2!+1/3!…..+1/n!
7. Display the following pyramids

*	1	1	1
* *	1 2	2 2	0 1
* * *	1 2 3	3 3 3	1 0 1
* * * *	1 2 3 4	4 4 4 4	0 1 0 1
* * * * *	1 2 3 4 5	5 5 5 5 5	1 0 1 0 1

BREAK Statement

- We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test.
- The keyword break allows us to do this.
- It is used to terminate the loop. When the keyword break is used inside any c++ loop, control automatically transferred to the first statement after the loop.
- A break usually associated with if statement.
- **Syntax : break;**

Example:

```
#include<iostream.h>
void main()
{
int I;
for(I=1;I<=10;I++)
{
if(I<=6)
break;
cout<<I;
}
}
```

Output: 1 2 3 4 5

Here the cout statement print value of I upto 5 when I reaches 6 the if statement is true. So the break statement transfer the control to the outside of the for loop.

Example: to determine whether a no is prime or not.

```
// a prime number which is divisible only by 1 and itself.
#include<iostream.h>
void main()
{
int n;
cout<<"Enter a number"; cin>>num;
for(I=2;I<num;I++)
{
if(num%I==0)
{
cout<<" number is not prime"; break;
}
}
if(num==I)
{
cout<<"Prime number";
}
}
```

- In this progam the moment num%I turns out to be 0, num is exactly divisible by I, the message "not prime no" is printed and the control breaks out of the while loop.
- Why does the program require the if statement after the while loop at all ?
    i. It jumped out because the number proved to be not prime.

    ii. The loop causes to an end because the value of I became equal to num.
- When the loop terminates in the second case, it means that there was no number between 2 & num-1 that would exactly divide num. That is num is indeed a prime.

- If this is true, the program should print out the message “Prime number”.

Problems:

1. Generate Prime number series (2 3 5 7 11 13 17 ….)

CONTINUE Statement

- In some programming situation we want to take the control to the beginning of the loop, by passing the statement inside the loop, which have not yet been executed.
- The keyword continue allows us to do this. When the keyword continue is encountered inside a c++ loop, control automatically passes to the beginning of the loop
- Continue statement usually associated with if statement.

  Syntax: continue;

  Example: sum of the +ve numbers(2, -3, 4, -2, 9, 5)

```
int Sum=0;
for(int I=0;I<10;I++)
{
cin>>no;
if(no<0)
continue;
sum=sum+no;
}
cout<<sum;
```

In the above program when the entered number is less than 0, then it is –ve number, so move the control to read the next number without performing summation.

Unconditional statement

- goto is the unconditional statement. Which is used to move the control anywhere inside the program.
- Goto require a lable in order to identify the place where the control is to be moved. A label is a valid variable name & a colon must follow it.
- The label is placed immediately before the statement where the control is to be transferred.
- General form:

```
Goto label;                    label:
………                          statement;
………                          ………
label:                         Goto label;
statement;                         ………
```

the label can be anywhere in the program either befor or after the goto statement.

Example:

```
main()
{
Int x,y;
read:
cin>>x;
if(x<0)
goto read;
y=x*x;
cout<<x<<y;
}
```

CASE CONTROL INSTRUCTION

- The control statement, which allows user to make a decision from the number of choices, is called a switch case statement.
- It allows user to execute a group of statements from several available group of statements.

Syntax:

```
switch(  expression)
{
case value1:
block1;
break;
case value2:
block2;
break;
case value3:
block3;
break;
………
………
default:
deafult block;
}
statement x;
```

Rules to be followed to construct case control instructions

1. The expression in switch statement must be an integer value or a character constant.
2. No case values are identical
3. Each case block must end with break statement.
4. The case keyword must terminate with colon(:)
5. Default is optional.

Execution Procedure:

- When switch is executed, the value of the expression is successively compared against the values value1, value2 –etc.
- If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.
- The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement transferring the control to the statement x following the switch.
- The default is optional case, when present, it will be executed if the value of the expression does not match with any case values.
- If not present action takes place if all matches fails and the control goes to the statement x.

Example: Read a number between 0-9 and print it in words.

```
cout<<"enter any number between 0-9";
```

```
cin>>num;
switch(num)
{
case 0: cout<<"ZERO"; break;
case 1: cout<<"ONE"; break;
case 2: cout<<"TWO"; break;
…………
…………
deault:
cout<<"Num is not within 0-9";
}
```

RUN

```
Enter a no between 0-9 9
NINE
```

The input value matches with case 9, so the corresponding block is getting executed. Remaining cases are all skipped from execution.

Problem:

Write a C++ program to read any integer number & display it in words.

ARRAYS

What are arrays?

For understanding the arrays properly, let us consider the following program.

```
main( )
{
int x;
x = 5; x = 10;
cout<<x;
}
```

- This program will print the value of x as 10. Why so? Because when a value 10 is assigned to x the earlier value of x, ie., 5, is lost. This ordinary variables are capable of holding only one value at a time..

- However, there are situations in which we want to store more than one value at a time in a single variable.
- For example, suppose we wish to arrange the percentage of marks obtained by 100 students in ascending order.
- In such a case we have two options to store these marks in memory:
  - (i) Construct 100 variables to store percentage of marks obtained by 100 different student ie., each variable containing one student's marks.
  - (ii) Construct one variable capable of storing or holding all the hundred values.
- The second one is better. The reason is it would be much easier to handle one variable than handling 100 different variables.
- An array is a group of logically related data items of the same data type addressed by a common name, and all the items are stored in contiguous memory locations.

Array Declaration

Like other normal variables, the array variable must be defined before it use.

<u>Syntax:</u>

Datatype Arrayname[array-size];

Arraysize – indicates the maximum number of elements the array can hold.

Example:

```
int marks[100]; // Integer array of size 100
float salary[25]; //Floating print array of size 25
char name[50]; //character array of size 50
```

Accessing Array Elements

- Once an array variable is defined, its element can be accessed by using an index or position.

<u>Syntax:</u> Arrayname[index];

- To access a particular element in the array, specify the array name followed by an integer constant or variable (array index) enclosed within square braces.
- Array index indicates the element of the array, which has to be accessed.

Example:

name[4]; //Accesses the 5th element of the array name.

- Note that, in an array of N elements, the first element is indexed by zero & the last element of an array is indexed by N-1
- The loop used to read the elements of the array is

```
for(int i=0; i<5; i++)
{
        cin>>name[i];
}
```

- The variable i varies from 0 to N-1.
- Note that, the expression age[i] can also be represented as i[age], similarly, the expression age[3] is equivalent to 3[age].

Array Initialization at Definition (at compile time)

- Arrays can be initialized at the point of their definition as follows:

    datatype array-name[size] = {list of values separated by comma};

    For instance, the statement, int age[5] = {19,21,16,1,50};

- Defines an array of integers of size 5.
- In this case, the 1st element of the array age is initialized with 19, 2nd with 21, and so on.
- The array size is omitted when the array is initialized during at compile time.

    int age[ ] = {19,21,16,1,50};

- In such a cases, the compiler assumes the array size to be equal to the number of elements enclosed within the curly braces.
- Hence, the above statement, size of the array is considered as five.

```
int age[5] = {19,21,16,1,50};
                (or)
int age[ ] = {19,21,16,1,50};
```

Example: Sum of Array Elements

```
#include<iostream.h>
void main( )
{
int a[10];
cout<<"Enter the no. of elements, max<10>";
cin>>n;
cout<<"Enter elements";
for(int I = 0; I<n; I++)
{
cin>>a[I];
}
int sum = 0;
for(int I = 0; I<n; I++)
{
sum = sum + a[I];
}
cout<<"Sum of entered elements"<<sum;
}
```

Problems:

1. Find largest/Smallest element in a given array.
2. Search an element in a given array
3. Arrange an array in ascending order

Two Dimensional Array

Matrix is a two dimensional array & two subscripts are required to access each element.

Declaration:

datatype Array-name[size1][size2];

size1 – no. of rows in a matrix.

size2 – no. of columns in a matrix.

Example:

int x[3][3];

Representation of 2-D array in memory:

Matrix is allocated into the memory according to row wise (row by row allocation).

Example:

$$A = \begin{matrix} 1 & 5 \\ 3 & 4 \end{matrix}$$

Accessing 2-D Array elements:

The elements of a 2-D array can be accessed by the following statement A[i][j]

i – row number

j – column number

Initialization of 2-D Array

A 2-dimenstional array can be initialized during its definition.

datatype matrixname [row size][col size] = {elements of first row, elements of 2nd row… elements of n-1 row};

Example:

int a[3][3] = {1,2,3,4,5,6,7,8,9} or

for more readability each row elements can be grouped:

int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; or

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

Row size can be omitted.

Example: // Read & display a matrix

```
#include<iostream.h> void main( )
{
int a[5][5];
cout<<”Enter row and col value of a matrix max<5>”;
cin>>r>>c;
cout<<”Enter elements in a Matrix”;
for(int i = 0; i<n; i++)
{
for(int j = 0; j<n; j++)
{
cin>>a[i][j];
}
}
cout<<”Display of given Matrix”; for(i = 0; i<n; i++)
{
for(int j = 0; j<n; j++)
{
cout<<a[i][j]<<setw(5);
}
cout<<endl;
}
}
```

Run:

```
Enter row and col value of a matrix max<5> 2 2
Enter elements in a Matrix";
2       4
6       8
Display of given Matrix
2       4
6       8
```

Problems:

1 Perform addition of two matrixes

2 Perform subtraction of two matrixes

3 Perform Multiplication of two matrixes

4 Find Transpose of a matrix

Pointers

A pointer is a variable whose value is the address of another variable. Like any variables, we must declare a pointer variable at the beginning of the program. We can create pointer to any variable type as given in te below examples.

The general format of a pointer variable declaration is as follows:-

datatype *pointervariable;

Examples:

```
int  *a;          //pointer to an integer variable
float *b;         //pointer to a float variable double
double *c         //pointer to a double variable
char *d;          //pointer to a character variable
```

- * → Represent Value at address Operator

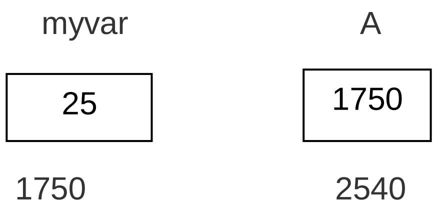
- & → Represent Address Operator

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as*address-of operator*. For example:

```
int myvar=25;
int *A;
A=&myvar;
```

This would assign the address of variable myvar to A; by preceding the name of the variable myvar with the *address-of operator* (&), we are no longer assigning the content of the variable itself to A, but its address.

The actual address of a variable in memory cannot be known before runtime. The values contained in each variable after the execution of this are shown in the following diagram:



/* Add Two Numbers using Pointer */

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int num1, num2, *a, *b, c=0;
        cout<<"Enter the two number :";
        cin>>num1>>num2;
        ptr1 = &num1;
        ptr2 = &num2;
        c = *a + *b;
        cout<<"Sum of the two number is "<<c;
        getch();
}
```

Output:

```
Enter the two number : 5  10
Sum of the two number is   15
```

STRINGS

- String is a group of characters
- A group of character can be stored in a character array.
- A string is an array of characters or character array.
- Strings are used in Programming languages for storing and manipulating text, such as words, names, sentences.
- End of the string is marked by the NULL (‘\0’) character.
- String constants are enclosed in double quotes
- Example: “Hello world”
- A string is stored in memory by using ASCII codes of the characters that form the string.

String Representation in Memory

String Declaration:

An array of characters representing a string is defined as follows:

char array_name[size];

size of the array must be an integer value.

Eg. Char name[50]; Defines an array & reserves 50 bytes of memory for string a set of characters. The length of the string cannot exceeds49 since; one storage location must be reserved for string the end of the string marker.

Initialization at compile time:

Char array_name = {list of characters separated by comma};

For example:

Char month = {‘A’,’P’,’R’,’I’,’L’,’/0’}; Or

Char month = “April”;

It has the same effect as the above statements.

The compiler takes care of storing the ASCII codes of the characters of the string in memory, and also stores the NULL character at the end.

Example: Read & Display a string

```
#include<iostream.h>
void main()
{
char name[50];
cout<<"ENTER YOUR NAME<49-max>";
cin>>name;
cout<<"Your name is "<<name;
}
```

Run:

```
Enter your name : Archanna
Your name is Archanna
```

- In main () the statement in >> name; reads characters & stores them into the variable name cout << name outputs the contents of the string variable name.

String Manipulation

- C++ has several built-in fuctions such as strlen(), strcat(), strcmp(), strcpy()..etc, for string manipulation.
- To use these functions, the header file **string.h** must be included in the program using the statement.

  #include<string.h>

i. **String Length**

- The string function strlen() returns the length of a given string.
- The length of the string excludes the end of string character('\0').

Syntax: int var=strlen(string variable);

Example:

```
#include<iostream.h>
#include<string.h>
void main()
{
char s1[20];
cout<<"Enter Your name";
cin>>s1;
cout<<"Strlen(s1)"<<strlen(s1);
}
```

Run:

```
Enter Your name: Smrithi
Strlen(s1):7
```

ii. <u>String Copy</u>

- The string function strcpy() copies the contents one string to another.

Syntax: strcpy(string1, string2); string1→Destination String string2 → Source String

Source string is copied into destination string.

Example:

```
#include<iostream.h>
#include<string.h>
void main() {
char s1[20],s2[20];
cout<<"Enter Your name";
cin>>s1;
strcpy(s2,s1);
cout<<"Strcpy(s2,s1): content of s2 is:"<< s2;
}
```

Run:

```
Enter Your name: Smrithi
Strcpy(s2,s1): content of s2 is: Smrithi
```

<u>iii. String concatenation:</u>

- The string function strcat() concatenates two string resulting in a single string.

Syntax: strcat(string1, string2); string1→Destination String string2 → Source String

- The destination & Source strings are concatenated and the resultant string is stored in the destination string.

Example:

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

```
void main()
{
char s1[20],s2[20];
cout<<"Enter Your first name"; cin>>s1;
cout<<"Enter Your second name"; cin>>s2;
strcat(s2,s1);
cout<<"Strcat(s2,s1): content of s2 is:"<< s2;
}
```

Run:

```
Enter Your First name: Roy Enter Your Second name: Joy
Strcat(s2,s1): content of s2 is:  RoyJoy
```

iii. String Compare:

- The string function strcmp() compares two strings, character by character.
- It returns an integer, whose value is
    - **< 0** if first string is alphabetically higher than second string.
    - **= = 0** If both are identical
    - **> 0** if first string is alphabetically lower than second string.

Syntax: strcmp(string1,string2);

- ASCII codes of each character in the given strings are compared. Once it find mismatch ASCII code it stop comparing and returns the difference between the ASCII value of mismatched string is returned.

Example:

```
#include<iostream.h>
#include<string.h>
void main()
{
char s1[20],s2[20];
cout<<"Enter name1"; cin>>s1;
cout<<"Enter name2"; cin>>s2;
int i=strcmp(s2,s1);
if( i= = 0 )
cout<<"Name1 and name2 are same";
else if(i<0)
cout<<" Name1 is alphabetically higher than name2 ";
else
cout<<" Name1 is alphabetically lower than name2 ";
}
Run1:
Enter name1: Roy Enter name2: Joy
Name1 is alphabetically lower than name2
Run2:
Enter name1: Roy Enter name2: Roy
Name1 and name2 are same
Run3:
Enter name1: Roy Enter name2: roy
Name1 is alphabetically higher than name2
```

ARRAYS OF STRINGS

An array of strings is a two dimensional array of characters and is defined as follows:

Syntax: char arry_name[row size][col size];

row size-> No of strings can be stored, col size-> no of characters in each string.

Example: char person[10][15]; → It defines an array of strings which can store name of 10 persons and each name cannot exceeds 14 characters. Last one character is used to represents the end of a string.

Example: Array of strings storing names

```
#include<iostream.h>
#include<string.h>
void main()
{
char person[10][15];
cout<<"How many persons <max-10>"; cin>>n;
for(int i=0;i<n;i++)
{
cout<<"enter person"<<i+1<<endl; cin>>person[i];
}
cout<<"Person names";
for( i=0;i<n;i++)
{
cout<<person[i]<<endl;
}
}
Run:
Enter Persons <max-100> 2
Enter person1: Roy Enter person2: Joy
Person names
Roy Joy
```

Problem:

Write a C++ program to perform sorting of N names in descending order.

Need for Structures:

- Arrays can be used to represent a group of data items that belongs to the same type, such as int or float.
- However, if we want to represent a collection of data items of different types using a single name, then we cannot use array.
- C++ supports a user-defined data type known as STRUCTURE.
- A Structure is convenient tool for handing a group of logically related data items.

Structure Declaration:

Declaration of a structure specifies the grouping of various data items into a single unit.

Syntax:

```
Struct structure name
{
datatype member1;
datatype member2;
--------------------
datatype member'n';
};
```

- Structure declaration can be written in a program either above main() or inside main()
- The structure declaration starts with the structure header, which consist of the keyword struct followed by a structure name.
- Structure name can be used for creating structure variables.
- The individual members of the structure are enclosed between the curly braces and they can be of the same or different data type.

For example:

```
struct  student
{
int roll_no; char name[25]; char branch[15]; int marks;
};
structure name -> student
```

Structure definition:

Structure definition creates structure variables and allocates storage space for them. Structure variable can be created at the point of declaration itself, or by using the structure name explicitly as and when required.

Syntax:

struct structure name var1,var2….; struct keyword is an optional.

Var1,var2 -> structure variables.

Example: struct student s1; (or)

student s1;

Memory allocation of the members the structure student:

```
struct  student
{
int roll_no; char name[25]; char branch[15]; int marks;
};
student s1;
```

The structure variables can be created during the declaration of a structure as follows:

```
struct  student
{
int roll_no; char name[25]; char branch[15]; int marks;
}s1;
```

Accessing Structure Members:

C++ provides the period or dot(.) operator to access the members of a structure independently.

Dot operator connects a structure variable and its member.

Syntax: **structure variable. member name**

```
struct  student
{
int roll_no; char name[25]; char branch[15]; int marks;
}s1;
```

Each member of the structure can be accessed using structure variable. S1.name

S1.roll_no

S1.branch

S1.marks

Structure Initialization

Similar to the standard data type, structure variables can be initialized at the point of their definition.

Consider the following structure declaration

```
struct  student
{
int roll_no; char name[25];
char branch[15]; int marks;
};
```

The members of the structure student can be initialized during variable creation itself. Student s1={5,”abc”,”computer”,240};

Example: Structure member initialization at the point of definition: #include<iostream.h>

//structure declaration struct date

{

int day; int month; int year; };

void main()

{

date d1={14,10,1983}; date d2={03,11,1984}; cout<<”Birthday”;

cout<<d1.day<<”-“<<d1.month<<”-“<<d1.year; cout<<d2.day<<”-“<<d2.month<<”-“<<d2.year;

}

Output:

Birthday: 22-07-2016 03-11-1984

Difference between structure and class

S.No.	Structure	Class
1.	Default access level for its members are public	Default access level for its members are public and protected
2.	It can not be inherited	It can be inherited
3.	It contains data members	It contains data members and member functions
4.	We cant initialize value to the variables inside the struct body	We can assign values to variables inside the class
5.	There is no data hiding	Data is hidden

UNION

Union Declaration:

Declaration of a union specifies the grouping of various data items into a single unit.

Syntax:

union union name

{

datatype member1;

datatype member2;

----

datatype member’n’;

```
};
union union name var1,var2,…;
```

The initialization, accessing of union members through union variable are similar to structure.

Eg. Program:

```
union person
{
char name[25];
int age;
float salary;
}p1;
void main()
{
cout<<"Enter details of the person";
cin>>p1.name>>p1.age>>p1.salary;
cout<<"Name:" << p1.name;
cout<<"Age:" << p1.age;
cout<<"Salary:" << p1.salary;
}
```

Difference between structure and Union

S.No.	**Structure**	**Union**
1.	Allocates storage space for all its members separately.	Allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2.	We can access all members of structure at a time.	We can access only one member of union at a time.
3.	It occupies higher memory space.	It occupies lower memory space over structure.

4.	Structure example:
	struct student
	{
	int mark;
	char name[6];
	double average;
	};
		Union example:
		union student
		{
		int mark;
		char name[6];
		double average;
		};

5.	For above structure, memory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

FUNCTIONS

Introduction

- It is difficult to implement a large program even if it is algorithm is available.
- To implement such a program in an easy manner, it should be split into a number of independent tasks, which can be easily designed, implemented, and managed.
- This process of splitting a large program into small manageable tasks and designing them independently is called Modular Programming.
- A repeated group of instruction in a program can be organized as a function.
- A function is a set of program statements that can be processed independently.

Advantages

- Reduction in the amount of work and development time.
- Program and function debugging is easier.
- Reduction in size of the program due to code Reusability.

Function Components:

- Function declaration or prototype
- Function Definition
- Function call
- Function parameters
- Function return statement

1. Function Prototype or Declaration:

It provides the following information to the compiler

- The name of the function
- The type of the value returned(optional, default is an integer)

- The number and type of the arguments that must be supplied in a call to the function.

Syntax:

Return type function_name(argu1,argu2,……argun);

Return type- specifies the data type of the value in the return statement.

Fun_name- name of the function.

Argu1,argu2…**argun** – type of argument to be passed from calling function to the called function

Examples:

int max(int,int); → It informs the compiler that the function max has 2 arguments of the type integer. The function max() returns an integer values.

void max(); → It informs the compiler that the function max has no arguments, max() is not returning any value.

max(); → It informs the compiler that the function max has no arguments. The function max() returns an integer values.

- In function prototype default return type is integer.

2. Function definition:

The function itself is referred to an function definition.

Syntax:

```
return type function_name(argu1,argu2,……argun) //function declarator
{
        function body;
}
```

- The first line of the function definition is known as function declarator, and is followed by the function body.
- The function delcarator and declaration must use the same function name, the number of arguments, the arguments type and the return type.
- Function definition is allowed to write in a program either above or below the main ().
- If the function is defined before main (), then function declarator is optional.

Example:

```
int  max(int x, int y)
{
if(x>y)
return(x);
else
return(y);
}
```

- For this function max() definition, it is declaration must be : int max(int,int);

3. **Function call:**

- A function, which gets life only when a call to the function is made.
- A function call specified by the function name followed by the arguments enclosed in parenthesis and terminated by a semi colon.

Syntax:

Function_name(argu1,argu2,……argun) ;

- If a function contains a return type the function call is of the following form:
- var= Function_name(argu1,argu2,……argun) ;

Example:

- c=max(a,b);
- Executing the call statement causes the control to be transferred to the first statement in the function body and after execution of the function body the control is returned to the statement following the function call.

```
// Greatest among 2 numbers #include<iostream.h>
void main()
{
int a,b;
int max(int,int); //function declaration cout<<"Enter any 2 integers"; cin>>a>>b;
int c= max(a,b); cout<<"Greatest is: "<<c;
}
int max(int x,int y)
{
```

```
if (x>y) return(x);
else
return(y);
}
Run:
Enter any 2 integers
40
35
Greatest is: 40
```

- The max() returns the maximum of the parameters a and b. The return value is assigned to local variable c in main().

4. Function Parameters

- The parameters specified in the function call are known as **actual parameters** and those specified in the function declarator (definition) are known as **formal parameters**
- For example in the main(), the statement c=max(a,b); passes the parameters(actual parameters) a and b to max().
- The parameters x and y are formal parameters.
- When a function call is made, a one to one correspondence is established between the actual and the formal parameters.
- The scope of the formal parameters is limited to its function only.

5. Function Return

- Functions can be grouped into two categories:
  - i. A Function does not have a return value (void function)
  - ii. Functions that have a return value.
- The statements: return(x); and return(y);in function max() are called function return statements. The caller must be able to receive the value returned by the function.
- In the statement c=max(a,b) → The value returned by the function max() returning a value to the caller.

Limitation of return

A key limitation of the return statement is that it can be used to return only one item from a function.

Passing Data to Functions

The entity used to convey the message to a function is the function argument. It can be a numeric constant, a variable, multiple variables, user defined data type, etc.

i. **Passing constants as arguments**

The following program illustrates the passing of a numeric constant as an argument to a function. This constant argument is assigned to the formal parameter which is processed in the function body

```
// Greatest among 2 numbers #include<iostream.h>
void main()
{
int a,b;
int max(int,int); //function declaration int c= max(40,35);
cout<<"Greatest is:  "<<c;
}
int max(int x,int y)
{
if (x>y)
return(x);
else
return(y);
}
Run:
Enter any 2 integers 40 35
Greatest is: 40
```

In main(), the statement c=max(40,35); invoke the function max with the constants.

ii. **Passing variable as arguments:**

Similarly to constants, varables can also be passed as arguments to a function.

```
// Greatest among 2 numbers #include<iostream.h>
void main()
{
int a,b;
int max(int,int); //function declaration
cout<<”enter two integer ”; cin>>a>>b;
int c= max(a,b); cout<<”Greatest is: “<<c;
}
int max(int x,int y)
{
if (x>y)
return(x);
else
return(y);
}
Run:
Enter any 2 integers 40 35
Greatest is: 40
```

In main(), the statement c=max(a,b); invoke the function max with the values of a & b

PARAMETER PASSING

- Parameter passing is a mechanism for communication of data and information between the calling function and the called function.
- It can be achieved by either by passing values or address of the variable.
- C++ supports the following 3 types of parameter passing schemes:
    1. **Pass by Value**
    2. **Pass by Address**
    3. **Pass by Reference**

i. Pass by Value

- The default mechanism of parameter passing is called pass by value.

- Pass by value mechanism does not change the contents of the argument variable in the calling function, even if they are changed in the called function.
- Because the content of the actual parameter in a calling function is copied to the formal parameter in the called function.
- Changes to the parameter within the function will affect only the copy (formal parameters)
- And will have no effect on the actual argument.

Example:

```
#include<iostream.h>
#include<iomanip.h>
void swap(int x,int y)
{
int t;
cout<<”value of x& y in swap() before exchange”;
cout<<x<<setw(5)<<y<<endl;
t=x;
x=y;
y=t;

cout<<”value of x & y in swap() after exchange”;
cout<<x<<setw(5)<<y<<endl;
}
void main()
{
int a,b;
cout<<”enter two integers”;
cin>>a>>b;
swap(a,b);
cout<<”value of a and b on swap(a,b) in main()”; cout<<a<<setw(5)<<b;
}
```

```
Run:
enter two integers 30 50
value of x& y in swap() before exchange 30 50
value of x& y in swap() after exchange 50 30
value of a and b on swap(a,b) in main() 30 50
```

Explanation:

- In main(), the statement swap(a,b) invokes the function swap()and assigns the contents of the actual parameters a & b to the formal parameters x & y respectively
- In swap() function, the input parameters are exchanged, however it is not reflected in the calling function; actual parameters a & b do not get modified.

ii. Pass by Address:

- C++ provides another means of passing values to a function known as pass by address mechanism.
- Instead of passing the value, the address of the variable is passed.
- In function, the address of the argument is copied into a memory location instead of the value.

Example:

```
#include<iostream.h>
#include<iomanip.h>
void swap(int *x,int *y)
{
int t; t=*x; *x=*y; *y=t;
}
void main()
{
int a,b;
cout<<”enter two integers”; cin>>a>>b;
swap(&a,&b);
cout<<”value of a and b after calling swap() in main()”; cout<<a<<setw(5)<<b;
}
```

```
Run:
enter two integers
30
50
value of a and b after calling swap() in main()"; 50 30
```

Explanation:

- In main(), the statement swap(&x, &y) invokes the function swap and assigns the address of the actual parameters a and b to the formal parameters x & y respectively.
- In swap(), the statement t=*x; assigns the contents of the memory location pointed to by the pointer (address) stored in the variable x. similarly, the paramets y holds the address of the parameter b.
- Any modification to the memory contents using these address will be reflected in the calling function, the actual parameter a & b gets modified.

iii. Pass by Reference

- Passing parameters by reference has the functionality of pass by address and the syntax of pass by value.
- Any modification made through the formal parameter is also reflected in the actual parameter.
- To pass as argument by reference, the function call is similar to that of call by value.
- In function declarator, those parameters, parameters, which are to be received by reference, must be preceded by the address (&)operator.
- The reference type formal parameters are accessed in the same way as normal value parameters.
- However, any modification to them will also be reflected in the actual parameters.

Example:

```
#include<iostream.h>
#include<iomanip.h>
void swap(int &x,int &y)
{
```

```
int t=x; x=y; y=t;
}
void main()
{
int a,b;
cout<<"enter two integers"; cin>>a>>b;
swap(a,b);
cout<<"value of a and b after swap(a,b) in main()"; cout<<a<<setw(5)<<b;
}
Run:
enter two integers 30 50
value of a and b after swap(a,b) in main() 50    30
```

Explanation:

- In main(), the statement swap(a, b); is translated into swap(&a,&b); internally during compilation.
- The function declarator void swap(int &a,int &b) indicates that the formal parameters are of reference type and hence, they must be bound to the memory location of the actual parameters
- Thus any access made to the reference formal parameters in the swap() reflects to the actual parameters.

DEFAULT ARGUMENTS

- Normally a function should specify all the arguments used in the function definition.
- In a c++ function call, when one or more arguments are omitted, the function may be defined to take default values for the omitted arguments by providing the default values in the function prototype.
- Hence the feature of default arguments allows the same function to be called with fewer arguments than defined in the function prototype.
- To establish a default value, the function declaration must be used.
- The compiler checks the function prototype with the arguments in the function call to provide default values to those arguments, which are omitted.

- Default arguments reduce the burden of passing arguments explicitly at the point of the function call.

Example:

```
#include<iostream.h>
void greatest(int = 50;int=25,int =35);
void main()
{
greatest();
greatest(10);
greatest(75,12);
greatest(15,2,55);
}
void greatest(int x,int y,int z)
{
if((x>y)&&(x>z))
cout<<"I st number is greatest";
else if(y>z))
cout<<"II nd number is greatest";
else
cout<<"III rd number is greatest";
}
```

- In the main(), when the compiler encounters the statement greatest(), it is replaced by the statement greatest(50,25,35); Internally substituting the missing arguments. Similarly when the compilers encounters the statement greatest (10); it is replaced by the statement greatest(10,25,35); Internally substituting the remaining two missing arguments and so on for all the remaining function calls.
- Variable names can be omitted while assigning default values in the prototype.

Scope and Namespaces

Scope

- The entities, such as variables, functions need to be declared before being used in C++. The point in the program where this declaration happens influences its visibility.
- An entity declared outside any block has global scope, meaning that its name is valid anywhere in the code. While an entity declared within a block, such as a function, has local scope, and is only visible within the specific block in which it is declared, but not outside it.
- Variables with local scope are known as local variables.
- For example, a variable declared in the body of a function is a local variable that extends until the end of the the function (i.e., until the brace } that closes the function definition), but not outside it.

EXAMPLE 1:

```
int p;        // global variable
int function1()
{
  int q;      // local variable
  q = 0;
}

int function2()
{
  p = 1;  // ok: p is a global variable
  q = 2;  // wrong: q is not visible from this function
}
```

EXAMPLE 2:

```
// inner block scopes
#include <iostream.h>
using namespace std;
```

```
void main ()
{
  int x = 10;
  int y = 20;
  {
    int x;   // ok, inner scope.
    x = 50;  // sets value to inner x
    y = 50;  // sets value to (outer) y
    cout << "inner block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "outer block:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
}
```

```
OUTPUT:
inner block:
x: 50
y: 50
outer block:
x: 10
y: 50
```

Namespaces

- For example, we might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

- A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Defining a Namespace:

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

SYNTAX:

```
namespace namespace_name
{
      // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
namespace_name :: code;          // code could be variable or function.
```

EXAMPLE PROGRAM:

```
#include <iostream.h>
using namespace std;
// first name space
namespace first_space
{
  void func()
  {
    cout << "Inside first_space" << endl;
  }
}
// second name space
namespace second_space
{
  void func()
 {
        cout << "Inside second_space" << endl;
    }
}
int main ()
{
     // Calls function from first name space.
    first_space::func();
     // Calls function from second name space.
    second_space::func();
     return 0;
}
```

OUTPUT:

```
Inside first_space
Inside second_space
```

The using directive:

The using directive:

We can also avoid prepending of namespaces with the **using namespace** directive. This directive tells the compiler that the subsequent code is making use of function or variable in the specified namespace. The namespace is thus implied for the following code:

EXAMPLE PROGRAM:

```
#include <iostream>
using namespace std;
// first name space
namespace first_space{
    void func(){
        cout << "Inside first_space" << endl;
    }
}
// second name space
namespace second_space{
  void func(){
    cout << "Inside second_space" << endl;
  }
}
using namespace first_space;
int main ()
{
    func();// This calls function from first name space.
    return 0;
}
OUTPUT:
Inside first_space
```

Source Files and Programs

CPP File:

A CPP file is a source code file written in C++, a popular programming language that adds features such as object-oriented programming to C. The file can be a standalone program or one of many files references in a development project. It must be compiled by a C++ compiler for the target platform before run.

CPP files are most commonly edited with programs that provide syntax highlighting. You can still open CPP files using any text editor, but programs that provide syntax highlighting, auto completion, and other helpful tools are most often used.

Object File:

An object file is a file containing object code, meaning relocatable format machine code that is usually not directly executable. There are various formats for object files, and the same object code can be packaged in different object files. An object file also works like an Application Extension (.dll).

Compilation

Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file. This step doesn't create anything the user can actually run.

Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. For instance, if we compile three separate files, we will have three object files created as output, each with the name <filename>.o or <filename>.obj (the extension will depend on your compiler). Each of these files contains a translation of our source code file into a machine language file. (we can't run them)

Linking

Linking refers to the creation of a single executable file from multiple object files. In this step, it is common that the linker will complain about undefined functions (commonly, main itself). During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file. If this isn't the case, there's no way the compiler would know -- it doesn't look at the contents of more than one file at a time. The linker, on the other hand, may look at multiple files and try to find references for the functions that weren't mentioned.

The compiler does its thing, and the linker does its thing -- by keeping the functions separate, the complexity of the program is reduced. Another advantage is that this allows the creation of large programs without having to redo the compilation step every time a file is changed. Instead, using so called "conditional compilation", it is necessary to compile only those source files that have changed; for the rest, the object files are sufficient input for the linker. Finally, this makes it simple to implement libraries of pre-compiled code: just create object files and link them just like any other object file. (The fact that each file is compiled separately from information contained in other files, incidentally, is called the "separate compilation model").

Knowing the difference between the compilation phase and the link phase can make it easier to hunt for bugs. Compiler errors are usually syntactic in nature -- a missing semicolon, an extra parenthesis. Linking errors usually have to do with missing or multiple definitions. If you get an error that a function or variable is defined multiple times from the linker, that's a good indication that the error is that two of your source code files have the same function or variable.