Introduction to OOPs: What Is Object-Oriented Programming, Encapsulation, Polymorphism, Inheritance, **C++ Overview:** The Origins of C++, The General Form of a C++ Program, different data types, operators, expressions, arrays and strings, Reference variables, Function Components, Argument passing, Inline functions, function overloading, function templates.

1.1What is Object Oriented Programming?

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods*.

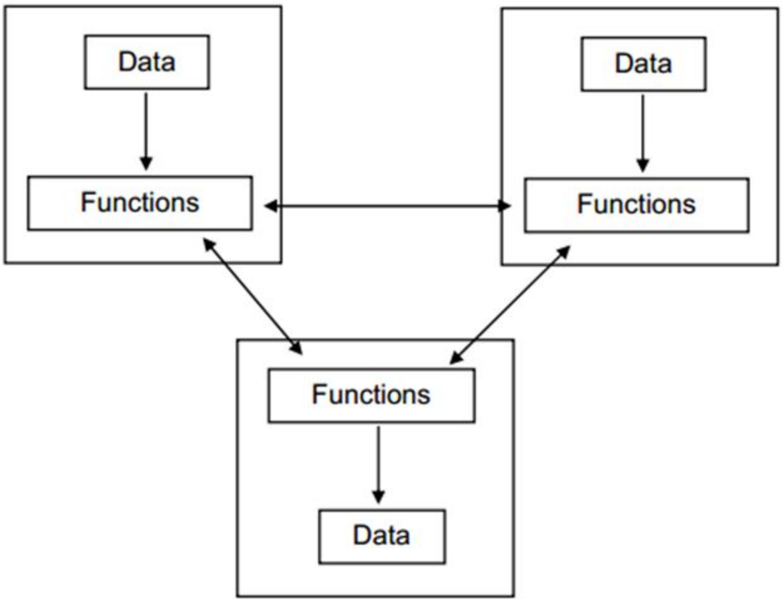


Fig 1: Object Oriented Programming

Building Blocks of OO language are:

- Class
- Object
- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

Class

Definition: A class is a user defined data type which binds data and functions together into single entity.

1.2 Encapsulation

Definition: The process of binding data and code together into single entity is called encapsulation.

- A class represents the protoype of a real world entity. Hence, a class, by its own will not have any physical existence. It can be treated as a user-defined data type.
- It consists of properties(known as data members) and behaviour(known as member functions)

1.3 Polymorphism

Definition: Polymorphism is a concept where one name can have many forms. Polymorphism (from Greek polys means "many, much" and morphē means "form, shape").

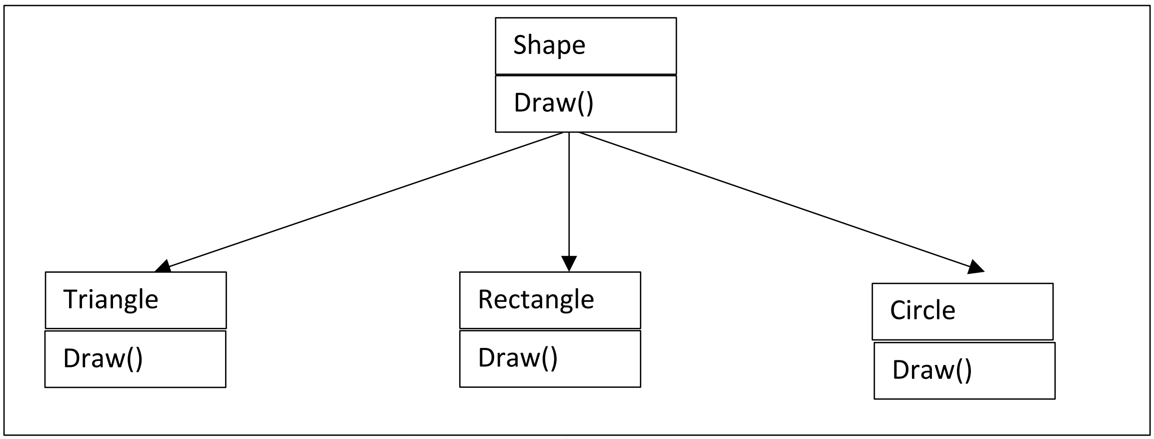


Figure 1.1 Polymorphism

Types of Polymorphism:

1. Static Polymorphism
   a. Function Overloading
   b. Operator overloading
2. Dynamic Polymorphism
   a. Virtual functions

1.4 Inheritance

Definition: Inheritance is a mechanism in which one class acquires the property of another class

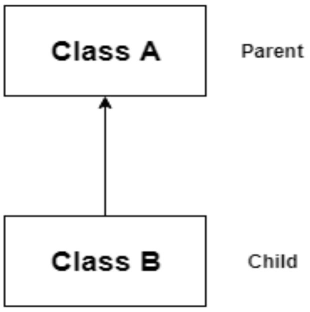


Figure 1.2 Inheritance

Why inheritance is required:

- Runtime polymorphism can be achieved
- Code reusability

1.5 C++ Overview: The origins of C++

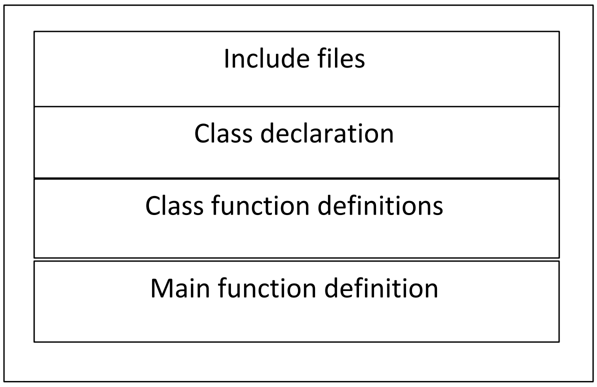
- During 1970 Dennis Ritchie created C Programming language on DEC PDP-11.
- C++ programming language is extension to C Language. Therefore called C++ as “Incremented C” means Extension to C.
- C++ was invented by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey.
- He initially called the new language “C with Classes.” However, in 1983 the name was changed to C++.

Versions of C++ Language

- There are several versions of C++ Programming Language –
  1. Visual C++
  2. Borland C++
  3. Turbo C++
  4. Standardize C++ [ANSI C++]

1.6 The General Form of a C++ Program

Structure of C++ Program



Include files

Class declaration

Class function definitions

Main function definition

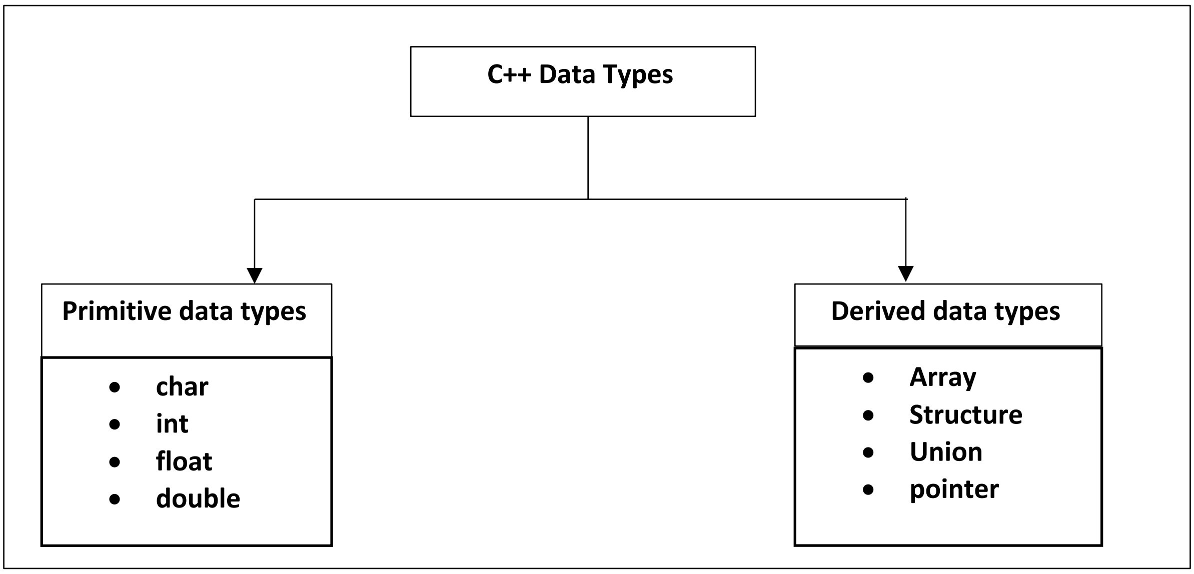
Structure of C++ Program

Example of C++

```
#include<iostream>
using namespace std;
class Abc
{
      int i;     //data variable
      void display(){ cout<<”Hello world”;} //member function
};
int main()
{
      Abc obj;//creating object
      Obj.display();//calling member function using class object
}
```

1.7 Different Data types

- Data types can be of two types.
    1. **Built-in Data types**
    2. **User-defined data types**



Built- in types

Type	**Size in bytes**	**Range**
Char	1	-128 to 127 (-2^7 + 1 to 2^7)
Int	4	-32, 768 to 32, 767 (-21^{14}+ 1 to 2^{15})
Float	4	Six digit of precision
double	8	Ten digit of precision

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

```
#include<iostream>
using namespace std;
int main(){
     cout<< ‟size of char :”<<sizeof(char)<<endl;
     cout<<””size of int :”<<sizeof(int)<<endl;
     cout<<””size of float :”<<sizeof(float)<<endl;
     cout<<””size of double :”<<sizeof(double)<<endl;
}
```

Output:

```
Size of char : 1
Size of int : 4
Size of float : 4
Size of double : 8
```

1.8 Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- Types of operators
  1. Arithmetic operators
  2. Relational operators
  3. Logical operators
  4. Bitwise operators
  5. Assignment operators
  6. Increment and decrement operator
  7. Additional operator

Arithmetic operators

Operator	operation
+ , -	Arithmetic addition and subtraction
*, /	Arithmetic multiplication and division
%	Modulus operator(gives remainder)

Relational Operators

Operator	Operation
<, >, <=, >=	Compare the values for greater than, less than
= =, !=	Compare the values for equality

Logical Operator

Operator	Operation
&&, ||, !	Logical Comparison of values

Bitwise operators

Operator	Operation
<<, >>	Left shift and right shift
&, |	Bitwise AND bitwise OR

Assignment Operators

Operator	Operation
=	Assignment operation
+=, -=, *=, /=	Compound assignment

Increment and decrement operator

Operator	Operation
++	Increments by one
--	Decrements by one

Additional operator

Operator	**Operation**
::	Scope resolution operator
.	**Member operator** used to reference individual members of classes
->	
endl	line feed operator (Adds new line)
new	Memory allocation operator
delete	Memory release operator
setw	Field width operator

1.9 Expressions

- Expression in c++ is a combination of constants, operands and operators that specify computation
- There are three types of expression:
    - Arithmetic Expression
        - Example: c = a + b;
    - Relational Expression
        - Example: if(a < b)
    - Logical Expression
        - Example: if ((a > b) && (a >c))

1.10 Arrays and Strings

Arrays:

- An array is a collection of a single data type. Each member of an array is associated by referring the index variable.
- Ex: int a[10];
- Here, a is an array variable of size 10 and its members are a[0],a[1]….a[9]. All these variables can store integer values. Array values can b initialized as

  int a[ ] ={15, 31, 42, 9, 6}
- Array can be classified into two parts as:
  1. Single dimensional Array.
  2. Multidimensional Array.

1. Single dimensional Array: (1-D Array)

- *An Array having only one dimension is called as single dimension Array.*
- **Example:**

```
int arr [5];            /* Integer array */
char str [10];            /* Character array */
```

2. Multi dimensional Array:

- *An Array having more than one dimension, then it is called as multidimensional Array.*
- **Example:**

```
int arr [5][5];        /* 2-D integer Array */
int  arr [2][3][2];   /* 3-D integer Array */
char str [8][4];       /* 2-D character Array */
```

Strings

- C++ provides following two types of string representations –
  - The C-style character string.
  - The string class
- **The c-style character string**
  - C-string is nothing but an array of characters or a string =s in c-string can be done as
    - char str[10];
  - Here, str is an array of characters or a string which can hold 10 characters and last character is being null character \0. A constant string is always enclosed within double quotes.
  - example

```
char str[ ] =”hello”
```

Example: C++ String to read a word

```
#include <iostream>
using namespace std;
int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: " << str << endl;
    cout << "\nEnter another string: ";
    cin >> str;
    cout << "You entered: "<<str<<endl;
    return 0;
}

Output:
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

- This is because the extraction operator >> works as scanf() in C and considers a space " " has a terminating character.

Example: String to read a line of text

```
#include <iostream>
using namespace std;
int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);
    cout << "You entered: " << str << endl;
    return 0;
}

Output:
Enter a string: Programming is fun.
You entered: Programming is fun.
```

- To read the text containing blank space, cin.get function can be used. This function takes two arguments.
- First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.
- In the above program, str is the name of the string and 100 is the maximum size of the array.

The String class

- In C++, you can also create a string object for holding strings.
- Unlike using char arrays, string objects has no fixed length, and can be extended as per your requirement.

Example: C++ strings using string data type

```
#include <iostream>
using namespace std;
int main()
{
    // Declaring a string object
    string str;
    cout << "Enter a string: ";
    getline(cin, str);
    cout << "You entered: " << str << endl;
    return 0;
}

Output:
Enter a string: Programming is fun.
You entered: Programming is fun.
```

- Instead of using cin>> or cin.get() function, you can get the entered line of text using getline().

- getline() function takes the input stream as the first parameter which is cin and str as the location of the line to be stored.

1.11 Reference variables

- **Definition:** A reference variable is an alias, that is, another name for an already existing variable.
- Reference variable shares the same copy of memory by creating an alias
- Once a reference is created, it cannot be later made to reference another object;
- References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created.
- **Syntax**:

```
type & variable1 = variable2
```

- Example

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

     int i=5;
     int &r=i;

     cout<<”value of i ” <<i;
     cout<<”value of i ” <<r;

     i=10;
     cout<<”value of i ” <<i;
     cout<<”value of i ” <<r;
r=20;
cout<<”value of i ” <<i;
cout<<”value of i ” <<r;

return 0;
}
```

Output

```
value of i=5
value of r=5
value of i=10
value of r=10
value of i=20
value of r=20
```

- Therefore, you can access the contents of the variable through either the original variable name or the reference.

Note:

References vs Pointers

- References are often confused with pointers but three major differences between references and pointers are −
- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

1.12 Function components

- A function (subroutine) is a sequence of instructions that performs a specific task.
- Functions are also known as procedure or subroutine in other programming languages.
- **Syntax(general form)**

```
return_type function_name( [ parameter_list ]  )
{
     //function body
}
```

- **Example**

```
void display( )
{
     cout<<”inside function display”;
}
```

➢ **Components of a function**

A function usually has 3 components. They are:

1. Function prototype/declaration
2. Function definition
3. Function call

i). Function Prototype/declaration

- Function declaration informs the compiler about the function's name, type and number of argument it receives and type of value it returns.
- If function is defined after function call () the prototype of function must be declared before main function. Function declaration informs the compiler that function definition is available in same program.
- Syntax for function declaration

```
return_type function_name( [type_list] );
```

- Example

```
void sum( int, int );
int volume( int );
```

ii) Function Definition

- Function definition consists of block of statements that specifies a specific task to be performed. Function definition start with function header consisting of return type, function name and list of formal argument.
- When a function is called it passes actual arguments to formal arguments of function definition. With a function call the control is transferred to the function definition.
- Syntax for function definition

```
return_type function_name( [ actual arguments ] )
{
     //block of statements
}
```

- Example

```
void  sum (int  x, int  y)     // function header; 'x' and 'y' are actual arguments
{
     cout << "Sum =" << x + y ;
}
```

iii) Function call

- Function call transfers the control to matching function name and arguments of a function definition. Function call supplies actual arguments to function definition.
- Syntax for function call

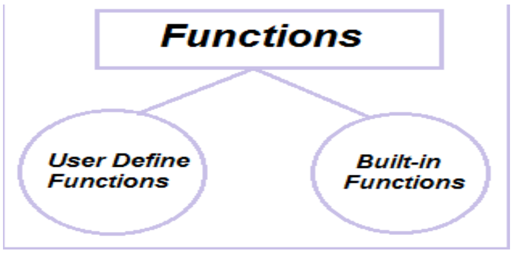
```
function_Name ([actual arguments]);
```

- Example

```
void sum ( int , int );       // function prototype
int  main ()
{
     sum(100, 200) ;      // function call
}
void  sum (int  x, int  y)      // function definition
{
     cout << "Sum =" << x + y ;
}
```

Types of Function

- **Built-in function**
  - Already defined in header files like <iostream>
  - Already in compiled (binary) form.
  - Ex: clrscr(), getch(), sin(), cos(), strcpy(), strcat(), .. etc



- **User-defined function**
  - C++ allows programmer to define their own function. A user-defined function groups code to perform a specific task and that group of code is given a name(identifier).

1.13 Argument Passing

- Argument passing is the process of initializing storage of function parameters with the values of function call arguments. In c++ there are three different ways of passing arguments to a function viz.
    1. Call-by-value
    2. Call-by-address/pointer
    3. Call-by-reference

Call-by-value

- Calling a function by passing values of the variables as arguments is called as call-by-value/pass-by-value.
- Many times, the arguments passed to a function will act as input to the function and the task carried out inside that function need not reflect the values of arguments.
- Instead, the result of the process has to be just returned to a calling function. In such situations, the variables are passed by-value to the function. Thus, in this method, the change in formal parameters does not reflect the original arguments.
- **P*rogram to that uses call-by-value method***

```
#include<iostream>
using namespace std;
void swap(int , int );
int main(){
     int a=10,b=25;
     cout<<”Before swapping:\n”;
     cout<<”a=”<<a<<”b”<<b<<endl;
     swap(a,b);//call-by-value
     cout<<”\n after swapping :”;
     cout<<”a=”<<a<<”b”<<b;
return 0;
}
void swap(int x,int y)
{
     int temp;
     temp=x;
     x=y;
     y=temp;
     cout<<”\n within function :”;
     cout<<x<<y;
}
```

```
Output:
Before Swapping:
a=10 b=25
within function: 25
10
After swapping
a=10 b=25
```

Call-by-address/pointer

- When the programmer wants the function arguments to be modified inside a function, the call-by-address method is used. Here, the addresses of the arguments are passed to a function in the form of pointers.

- ***Program that uses call by address method to swap two variables***

```
#include<iostream>
using namespace std;
void swap(int, int);
int main(){
     int a=10,b=25;
     cout<<”Before swapping:\n”;
     cout<<”a=”<<a<<”b”<<b<<endl;
     swap(&a,&b);//call by pointer
     cout<<”\n after swapping :”;
     cout<<”a=”<<a<<”b”<<b;
return 0;
}
void swap(int *x, int *y)
{
     int temp;
     temp = *x;
     *x = *y;
     *y = temp;
```

```
Output:
Before swapping
a=10 b=25
after swapping
a=25 b=10
```

Call-by-reference

- The references are being created to the actual arguments and are passed to a function. Since, references are just alias names for the variables, the modification done to the formal parameters inside the function will reflect the actual arguments.
- **Program that uses call-by-references method**

```
#include<iostream>
using namespace std;
void swap(int, int);
int main(){
     int a=10,b=25;
     cout<<”Before swapping:\n”;
     cout<<”a=”<<a<<”b”<<b<<endl;
     swap(a,b);// call by reference
     cout<<”\n after swapping :”;
     cout<<”a=”<<a<<”b”<<b;
return 0;
}
void swap(int &x, int &y)
{
     int temp;
     temp = x;
     x = y;
     y = temp;
}
```

```
Output:
Before Swapping
a=10 b=25
After Swapping
a=25 b=10
```

1.14 Inline Function

- When a function is declared inline, the compiler places a copy of the code of that function at each point where the function is called at compile time. To inline a function, place the keyword **inline** before the definition of function name.
- Inline functions should be small. The compiler ignores the inline qualifier in case defined function is more than a line. Inline function reduces the overhead of function calling, passing control, and returning a control.
- All the functions defined inside *class* definition are by default inline.

```
inline  int sum(int a, int b).      //’a’ and ‘b’ are formal parameters
{
     return (a>b) ? a : b;
}

int main()
{
     int max;
     max = sum(10, 20);      // 10 and 20 are actual parameters
     cout << “Max = ” << max;
}
```

- **Some Important points about Inline Functions**
  - We must keep inline functions small, small inline functions have better efficiency.
  - Inline functions do increase efficiency, but we should not make all the functions inline.
  - Because if we make large functions inline, it may lead to code bloat, and might affect the speed too.
  - It is advised to define large functions outside the class definition using scope resolution :: operator, because if we define such functions inside class definition, then they become inline automatically.

1.15 Function Overloading

- **Definition:** *Function overloading is the ability to create multiple functions of the* ***same name*** *with* ***different implementations.***
- Overloaded functions should have different **type**, **number** or **sequence** of parameters. Function overloading is a compile-time polymorphism.

```
//Program for finding area of circle and area of a triangle

#include<iostream>
using namespace std;

float area(int r){
     Return 3.14 * R *R;
}
float area(int b, int h)
{
     Return 0.5 * b *h;
}
int main(){
     int r=3,b=4,h=6;
     float a;

     a=area(r);      //area() function with one argument is called
     cout<<"area of circle="<<a;

     a=area(b,h);    //area() function with two argument is called
     cout<<"area of circle="<<a;
return 0;
}
```

- **Note** that, when the number of arguments and their data types of two of more functions are same, functions cannot be overloaded.
- for example, the below given functions, even though they are used for doing different tasks, cannot be declared.
    - `int myfun(int , char)`
    - `int myfun(int, char)`
- But, the declaration of the following two functions is possible as the argument list differ in their data types.
    - `int myfun(int, char)`
    - `int myfun(int, int)`

Function Overloaded Resolution

- The compiler decides which function is to be called among the set of overloaded functions. The function overload resolution involves the following steps.
    1. **Identify Candidate functions**
    2. **Viable functions**
    3. **Select best viable function(best match)**

1. **Identify candidate functions**
    - When a particular function call occurs, the compiler first list all the functions having same name as being called. These are known as candidate function.
    - And it identifies the properties of arguments like the total number of arguments and the data types of each of these arguments.

- For example,
  - int myfun(int, int);
  - int myfun(int, float);
  - int myfun(int, char);
  - int myfun(double int);
  - int myfun(char, int, float);
- And the call for the function with two integer parameters *int myfun(10,20)*
- then the following are the *candidate functions*
  - int myfun(int, int);
  - int myfun(int, float);
  - int myfun(int, char);
  - int myfun(double int);
- here, the function
  - int myfun(char, int, float); //*will not be a candidate as it has got three parameters*

2. **Viable functions**
   - Here, the compiler selects the functions from the list that it has got in step 1, those can be called with arguments specified. This set of functions is known as *viable function*.
   - There must exist implicit conversion that can convert each argument in the argument list to the type of its corresponding parameter in the function parameter list.
   - For example, in the above list of candidate functions, the *viable function* will be.
     - int myfun(int, int);
     - int myfun(int, float);
     - int myfun(double int);

3. **Select best viable function**
   - The compiler chooses the *best viable function*.
   - The compiler does this by *implicit conversion sequences* when best match is not found.
   - Example: Best match for function call **myfun(10, 20);'**
     - int myfun(int, int);

1.16 Function Templates

- **Definition**: *A function which defines a general set of operations that can be applied to various types of data is known as template function or generic function.*
- The concept of templates can be used in two different ways:
  1. Function templates(generic function)
  2. Class templates(Generic class)

- Templates are mechanism with which it is possible to use one class or function to handle different types of data.
- Using templates, one can design a single class/function that operates on data of many types, without having to explicitly write a code for each data type. When used with functions, they are known as *function templates and when used with class, they are called as class templates*.

- The general form for creating a generic function

```
template < class t_name > ret_type fun_name(para_list)
{
     //body of the generic function
}
```

- *Here,*
- *template* and *class* are keywords
- *t_name* :is any name given to the template that will act as a placeholder for a data type used by the function
- *ret_type* : is the return type of the function
- *fun_name:* is any vaild name given to the function
- *para_list* : is the list of parameters that the function uses and is optional.

Example (Single type parameter)

```
template <class T>
T  getMax(T  x, T  y)
{
return (x > y) ? x: y;
}
int main()
{    cout << getMax<int>(3, 7) << endl; // Calling myMax for int
     cout << getMax<double>(3.0, 7.0) << endl; // Calling myMax for double
     cout << getMax<char>('g', 'e') << endl;// call myMax for char
}
```

Example (Multiple type parameter)

```
template <typename RT, typename T1, typename T2 >
RT  getMax(T1  x, T2  y)
{
return (x > y) ? x: y;
}
int main()
{
     double a, b;
     a = getMax<double, int, double>(10, 10.5) << endl;
     b = getMax<double>(10.0, 10.5) << endl;
     cout << “a=” << a << “b=” << b;
}
```

```
// finding maximum of two integers and characters using template function.
#include<iostream>
template <class T> T max(T a, T b){
     T big;
     big= (( a>=b) ? a: b);
     return big;
}
int main(){
     int p,q;
     char x, y;
     cout<<”\n enter two integers “;
     cin>>p>>q;
     cout<<”\n enter two characters “;
     cin>>x>>y;
     cout<<”Biggest of two integers is :”<<max(p,q);
     cout<<”Biggest of two characters is :”<<max(x,y);
     return 0;
}
```

Overloading template function

- Just like normal function, a template function also can be overloaded. For doing so we just need another template with different number of arguments form the original one.

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

//template function with one parameter
Template<class T1> void fun(T1 a){
     Cout<<”\n Template function using one argument:”<<a<<endl;
}
//template function with two parameter
Template<class T1 , class T2> void fun(T1 a, T2 b){
     Cout<<”\n Template function using two argument:”<<a<<b;
}

int main()
{
     Int x, y;
     cout<<”enter two integers”;
     cin>>x>>y;
     fun(x)     //fun(T1) gets called here
     fun(x,y);  //fun(T1,T2)gets called here

return 0;
}
Output:
Enter two integers: 4 5
Template function using one argument 4
Template function using two argument 4 5
```

Default Arguments

- **Definition:** Few of the arguments having default values at the time of function declaration are known as default arguments.
- In some situation, a function may contain more parameters than required for its most common usage. That is, it may not be essential to pass a value for a particular, when a function is called. To allow the programmer to drop unnecessary parameter(s) during function call, C++ provides the facility of default arguments. Here the programmer needs to specify only those arguments that suit the exact situation.

- **Example for default argument**

```
#include<iostream>
using namespace std;
void fun(int);
int main(){
     fun(); //call with no parameter
     fun(30);  //call with parameter
     fun(); //call with no parameter
}
void fun (int x){
     cout<<x<<endl;
}
Output:
20
30
20
```

- **Rules of default Arguments**

I. The default arguments are defined in the function prototype or in the argument list of function definition if function is written before main.

```
void add(int a=10,int b=20,int c=30); // default arguments in prototype
int main()
{
     ------
}
```

II. Parameters with default arguments must be trailing parameters in the function declaration

```
void add(int a,int b,int c=30); // default parameter 'c' is trailing parameter
```

III. if you define a default argument for a parameter, all following (trailing) parameters must have default arguments.

```
void add(int a,int b=10,int c=30);   // All trailing parameters after 'b' are default.

void add(int a,int b=10,int c);    // Invalid as parameter 'c' not given default value
```

In the above example; if parameter “b‟ is default then all trailing parameters after “b‟ also must have default values.

IV. You must first supply any arguments that do not have default values

```
void add (int a, int b, int c=30);
int main()
{
     add(10);        // Invalid – must supply value for parameter ‘a’ and ‘b’
     add(10, 20);       // Valid
     add(2, 3, 4);       // Valid
}
void add( int a , int b , int c)
{
     cout <<  ‶Sum = ″ << a+b+c;

}
```

V. Default arguments are overwritten when calling function provides values for them.

```
void add (int a=10, int b=20, int c=30);
int main()
{
     add(2, 3, 4);       // overwrite a, b and c with values 2,3 and 4 resp.
}
void add (int a, int b, int c)
{
     cout <<  ‶Sum = ″ << a+b+c;
}
```

Frequently asked programs

1. READ A string and display the same

```
#include<iostream.h>
int main ( )
{    char str[50];
     cout<<“Enter a name:”;
     cin>> str;
     cout<<”Name is”<<str;
}
Output:
     Enter a name: Pankaj Kumar
     Name is: Pankaj     //(stop Reading at
     white space)
```

2. Program to Find sum and product of array elements

```
#include<iostream>
using namespace std;
int main ()
{
    int arr[10], n, i, sum = 0, pro = 1;
    cout << "Enter the size of the array : ";
    cin >> n;
    cout << "\nEnter the elements of the array : ";
    for (i = 0; i < n; i++)
    cin >> arr[i];
    for (i = 0; i < n; i++)
    {
        sum += arr[i];
        pro *= arr[i];
    }
    cout << "\nSum of array elements : " << sum;
    cout << "\nProduct of array elements : " << pro;
    return 0;
}
```

Output:

```
Enter the size of the array : 5
Enter the elements of the array : 1 2 3 4 5
Sum of array elements : 15
Product of array elements : 120
```

3. Find Biggest element in an arary

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

int main()
{
    int i, n;
    float arr[100];

    cout << "Enter total number of elements(1 to 100): ";
    cin >> n;
    cout << endl;

    for(i = 0; i < n; ++i)
    {
       cout << "Enter Number " << i + 1 << " : ";
       cin >> arr[i];
    }

    for(i = 1;i < n; ++i)
   {
       if(arr[0] < arr[i])
           arr[0] = arr[i];
    }
    cout << "Largest element = " << arr[0];

    return 0;
}
Output:
```

```
Enter total number of elements: 8

Enter Number 1: 23.4
Enter Number 2: -34.5
Enter Number 3: 50
Enter Number 4: 33.5
Enter Number 5: 55.5
Enter Number 6: 43.7
Enter Number 7: 5.7
Enter Number 8: -66.5

Largest element = 55.5
```

4. Write C++ program to copy one string another string.

```
#include <iostream>
int main()
{
    char s1[100], s2[100], i;

    printf("Enter string s1: ");
    scanf("%s",s1);

    for(i = 0; s1[i] != '\0'; ++i)
    {
        s2[i] = s1[i];
    }
```

```
              s2[i] = '\0';
              printf("String s2: %s", s2);

              return 0;
          }
Output:
```

```
Enter String s1: RNSIT
String s2:RNSIT
```

5. Write C++ program to copy one string another string using built-in function

```
     #include <iostream>
     #include <cstring>

     using namespace std;

     int main()
     {
         char s1[100], s2[100];

         cout << "Enter string s1: ";
         cin.getline(s1, 100);

         strcpy(s2, s1);

         cout << "s1 = "<< s1 << endl;
         cout << "s2 = "<< s2;

         return 0;
     }
Output
```

```
Enter string s1: RNSIT
s1 = RNSIT
s2 = RNSIT
```

6. Write C++ program to concatenate two strings using Built-in function.

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

     int main()
     {
      string s1, s2, result;
```

```
 cout << "Enter string s1: ";
 getline (cin, s1);

 cout << "Enter string s2: ";
 getline (cin, s2);

 result = s1 + s2;

 cout << "Resultant String = "<< result;

 return 0;
}
```

Output:

```
Enter string s1: C++ Programming
Enter string s2:  is awesome.
Resultant String = C++ Programming is awesome.
```

7. Write C++ program to concatenate two strings using User-Defined function.

```
#include <stdio.h>
int main()
{
   char str1[50], str2[50], i, j;
   cout<<”\nEnter first string:”;
   cin>>str1;
   cout<<nEnter second string:”;
   cin>>str2;

   for(i=0; str1[i]!='\0'; ++i);

   for(j=0; str2[j]!='\0'; ++j, ++i)
   {
      str1[i]=str2[j];
   }

   str1[i]='\0';
   cout<<”\nOutput: %s",str1;

   return 0;
}
```

```
Enter first string: C++ Programming
Enter second string:  is awesome.
Output= C++ Programming is awesome
```