chapter-9 Templates.

Benifit of object oriented programming is reusability.

C++ can implement the concepts of reusability with the help of templates.

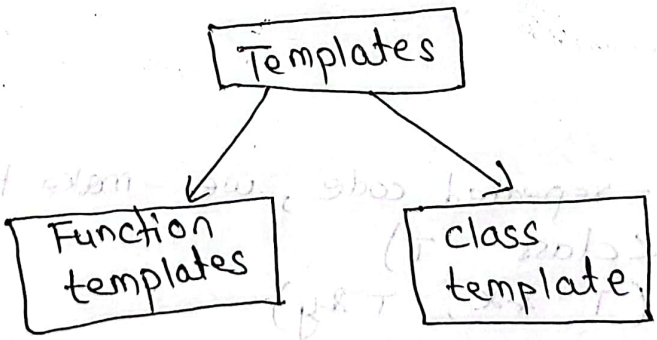
Templates in C++ is an interesting feature that is used for generic programming. Generic programming is an approach of programming where generic types are used as parameters in algorithms to work for a variety of data types.

Templates :

Templates in C++ is defined as a blueprint or formula for creating a generic class or a function.

In simple,
You can create a single function or single class to work with different data types using templates.

Types of templates.



Function template.

Function template in C++ is a single function template that works with multiple data types.

Syntax:

```
template <class type>
return_type function_name(parameter_list)
{
// body of a function.
}.
```

Eg:

```
#...
Void swap(int &x, int &y)
{
int t;
t=x;
x=y;
y=t;
}
Void swap(double &x, double &y)
{
double t;
t=x;
x=y;
y=t;
}
Void swap(float &x, float &y)
{
float t;
t=x;
x=y;
y=t;
}
```

To eliminate repeated code, we make templates.

Eg:

```
template <class T)
Void swap(T &x, T &y)
{
T t;
t=x;
x=y;
y=t;
}.
```

```
int main()
{
int a=5, b=7;
swap(a,b);
float d=2.5, c=7.3;
swap(d,c);
double e=2.1, f=3.6;
swap(e,f);
return 0;
}
```

Function template for different types of data types.

Eg:

```
template <class T1, class T2>
void testfunc (T1 a, T2 b)
{
cout << a << " " << b << endl; }
int main()
{
int inum = 5;
float fnum = 7.5;
test_func (inum, fnum);
test_func (20, "c++");
test_func (15.3, 17L);
return 0;
}
```

Case I: when all of the template parameters are not used in function argument.

Example:

```
template <class T1, class T2>
void testfunc (T2 a)
{
T1 b;
//....
}
```

then a call to a function as

```
int num;
testfunc(num); // error cannot deduce T1
```

fails to compile because compiler cannot deduce (replace) the the template parameter T1, as T2 is only used in the function parameter list. So to resolve this problem the function instantiation is done explicitly by specifying the template parameter as

```
test_func <float>(inum); // T1 is double and T2 is int.
```

Also,

```
testfunct <float, int>(inum);
```

Also,

```
test_func <int, float>(inum);
```

Overloading Function Template.

When using function templates we can declare several function templates with same name and even declare a combination of function templates and non template functions (normal functions) with the same name.

Overloading function templates can be of two types:

(a) Overloading with functions.
(b) Overloading with other templates.

(a) Overloading with functions.

If the function templates definition is not suitable for some specific data type then it is necessary to override the function template by defining normal function of same name for specific type.

If a normal function is an exact match than the function template then the normal function is selected when the function is called.

Eg:

```
#include <iostream>
#include <cstring>
using namespace std;
template <class T>
T find_max (T a, T b)
{
 T result;
 if (a>b)
    result = a;
 else
    result=b;
 return result;
 }
// for string data types.
char *find_max (char *a, char *b)
 {
  char *result;
    if (strcmp(a,b) >0)
       result = a;
    else
       result=b;
    return result;
   }
int main()
 {
int i1 = 15, i2 = 20;
cout <<" Greater is" << find_max (i1, i2) << endl;
float f1 = 40000.05, f2 = 38000.44;
cout <<" Greater is" << find_max (f1, f2) << endl;
double d1 = 55.05, d2 = 67.777;
cout <<" Greater is" << find_max (d1, d2) << endl;
char c1 = 'a', c2 = 'A';
cout <<" Greater is" << find_max (c1, c2) << endl;
char str1[] = "apple", str2[] = "Zebra";
cout <<" Greater is" << find_max (str1, str2);
return 0;
 }.
```

⑤. Overloading with other Template.

Along with overloading the function template with normal function, the function template can be overloaded with other function templates. That is, we can have multiple function templates with the same name but with different parameter list. Similar to overloading of normal functions, overloaded function templates must differ either in terms of number of parameters or their type.

Eg:

```
# include <iostream>
using namespace std;
template <class T>
void func (T a , T b)
{
cout <<" func (T a , T b) :"<<a<<','<<b<<endl;
}
template <class T1 , class T2>
void func (T1 a, T2 b)
 {
cout << "func (T1 a, T2 b) :"<<a <<','<< b<<endl;
 }
int main()
 {
 func (1,2);
 func (1, 2.5);
 func (1.7, 3);
 return 0;
 }.
```

Class Template.

Similar to the function templates, we can declare class templates that operate on any types of data. A class that operates on any type of data is called class template.

```
const int MAX = 20;
class intStack
 {
private:
 int arr[MAX];
 int top;
 public:
  intStack();
  Void push(int data);
  int pop();
  int size();
  };
```

```
const int MAX = 20;
class floatStack
   {
   private:
     float arr[MAX];
     int top;
  public:
   floatStack();
   void push(float data);
   float pop();
   int size();
    };
```

If we need to store data of other types in the stack then we need to declare stack class for each and every data type.

In this case, if we would create a single class specification that would work for any data type then the repeated code for different data types can be eliminated.

The class template models a generic class which supports similar operation for different data types.

Syntax:

```
template <class template_type,....>
class class_name
    {
   private:
      // data member of template type or non template type.

      //....
   public:
     // function members with template type argument
     // and return type
     };
```

Eg:

```
const int MAX = 20;
template <class T>
class Stack
 { private:
       T arr[MAX];
       int top;
```

```
public:
  stack();
  void push(T data);
  T pop();
  int size();
  };
```

How object is created?

```
class_name <data_type> object;
```

Eg:

```
stack <int> i;
stack <float> f;
```

Using multiple data types in a class.

```
#include <iostream>
using namespace std;
template <class T1, class T2>
class test
 {
 private:
     T1 a;
     T2 b;
 public:
   test() {}
   test (T1 n1, T2 n2)
    {
    a = n1;
    b = n2;
    }
 Void display()
   {
 cout <<"Data:" <<a <<" and" <<b <<endl;
 };
 int main()
 {
 test <int, float> myobj (5, 7.39);
 myobj.display();
 return 0;
 }.
```

Function Definition of class Template.

When a member function of a template class is defined outside the class, the member function must explicity be declared as a template. The member function of class template is defined outside the class in the following form.

Syntax:

```
template <class template_type>
class class_name
{
 private :
      template_type variable_name;
      //.....
  public :
      return_type function_name(template_type arg);
      //.....
   };
template <class template_type>
return_type class_name <template_type> ::
            function_name (template_type arg)
      { // body of function template.
       }.
```

Example:

```
#include<iostream>
using namespace std;
const int size=5;
template <class T>
class Array
  {
 private :
     T arr[size];
  public:
     Void get_array();
     T find_max();
     T find_min();
    };
template <class T>
void Array<T> :: get_array()
  {
  for (int i= 0; i< size ; ++i)
     cin >> arr[i];
  }
template <class T>
T Array<T> :: find_max()
     {
```

```
T max = arr[0];
for (int i = 1; i < size; ++i)
{
if (arr[i] > max)
    max = arr[i];
  }
return max;
}
template <class T>
T Array<T>:: find min()
{
T min = arr[0];
for (int i=1; i<size; ++i)
  {
    if (arr[i] < min)
        min = arr[i];
  }
  return min;
}
int main()
{
Array <int> a1;
cout <<" Enter integer numbers :";
a1.get_ array();
cout <<" Largest integer number is :" << a1.find_max();
cout <<"Smallest integer number is :" << a1. find_min();
Array <float> a2;
cout << "Enter floating numbers :";
a2.get_array();
cout <<" Largest floating number is :" << a2. find_max();
cout << "Smallest floating number is :" << a2. find_min();
return 0;
}.
```

Non-Template Type Arguments.

In addition to template parameter, the template specification for a generic class can have non template type parameters too.

Syntax: Example:

```
#include <iostream>
using namespace std;
template <class T, int size>
class Array
  {
  private:
     T arr[size];
  public:
     void get_array ();
     T find_max ();
     T find_min ();
   };
template <class T, int size>
void Array <T, size>:: get_array()
  {
  for (int i=0; i<size; ++i)
     cin >> arr[i];
  }
template <class T, int size>
T Array<T, size>:: find_max()
  {
   T max = arr[0];
   for (int i=1 ; i< size; ++i)
   {
    if (arr[i] > max)
       max = arr [i];
       }
    return max;
  }
template <class T, int size>
T Array <T, size>:: find_min()
  {
   T min = arr [0];
   for (int i=1; i<size ; ++i)
     {
      if (arr[i]< min)
          min= arr[i];
       }
      return min;
  }
```

```
int main()
{
Array <int, 6> a1;
cout << " Enter integer numbers: ";
a1.get_array();
cout << "Largest integer number is: " << a1.find_max();
cout << " Smallest integer number is: " << a1.find_min();
Array <float, 5> a2;
cout << "Enter floating numbers:";
a2.get_array();
cout << "Largest floating number is: " << a2.find_max();
cout << "Smallest floating number is: " << a2.find_min();
return 0;
}.
```

Default Arguments with Class Template.

```
template <class template_type = default_data_type>
class class_name
    {
    // data and function declarations.
    };
```

Eg:

```
template <class T=int>
class test
{
  //....
};

test <float> ft;  // template argument is float.
test <> int; // template argument int when not supplied
```

Derived class template.

Similar to the inheritance of normal class, the class template can also be inherited. Inheritance with class template provides a mechanism for building new data types from existing ones which has some form of common behaviors and properties.

Providing template arguments to the base class.

(Only base as template class)

```
template <class T>
class base
  {
  private:
     T data;
  public:
    base () {}
    base (T a)
      {
      data = a;
      }
    void display()
     {
     cout <<" data :" << data;
     }
   };
class derived1 : public base<int>
   {
   public:
     derived1 () { }
     derived1 (int a) : base <int> (a) { }
     };
```

```
derived1 obj(5);
obj.display()
```

Template argument of the base class with the data type we create with template argument of derived class.

(Constructor on both classes) (Both base and derived are template)

```
template <class T>
class derived2 : public base<int>
 {
 private:
    T data;
 public:
    derived2 (int a, T b) : base<int>(a) {data=b;}
    void display()
      {
```

```
cout << " in base";
base<int>::display();
cout << "in derived data: " << data;
  }
};

derived2<float> ob2(10, 12.5);
ob2.display();
```

Introduction to standard template library (STL)

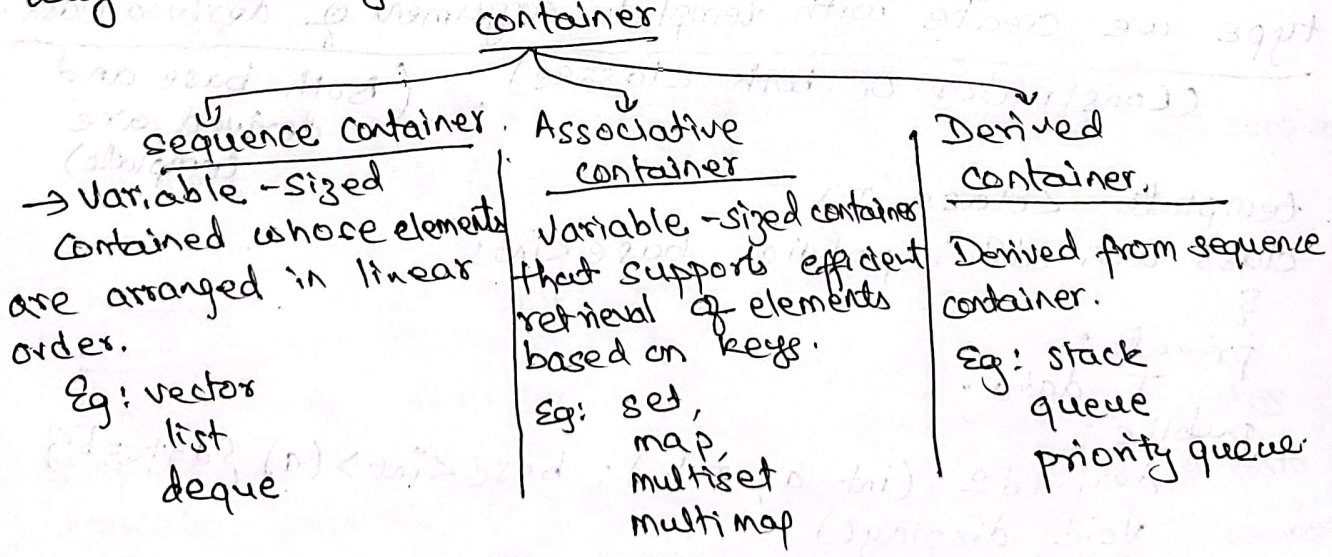
The C++ STL is a powerful set of C++ template classes to provide general purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues and stacks.

Components of STL.

1. Container.
2. Iterators.
3. Algorithms.

1. Containers.

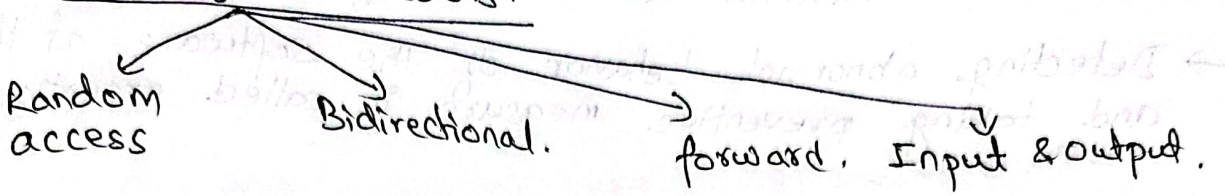
The STL container are data structures capable of storing objects of any data type in an organized way in memory.



Iterators.

STL iterators, which have the properties similar to those of pointers, are used by programs to manipulate the STL - container elements.

Types of Iterators.



Algorithms.

STL algorithms are functions that perform common data manipulations such as searching, sorting, copying and compairing data.

Algorithms are implemented as template functions. Some of these are find, search, swap, count, merge, fill etc.

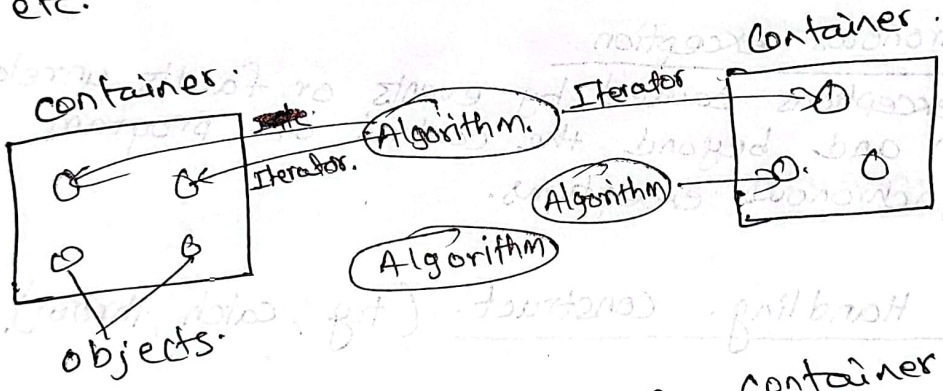


fig: Relationship between container, algorithm and objects.

chapter-10.
Exception Handling.

→ Exceptions are errors that occur at run time. They are also called runtime anomalies (irregularity).

→ Detecting abnormal behavior of the software at the runtime and taking preventive measure is called exception handling.

Types of exceptions:

ⓐ Synchronous exception.
ⓑ Asynchronous exception.

ⓐ Synchronous exception.

The exceptions which occur due to problem in input data or inappropriate way of handling data are called synchronous exceptions.

ⓑ Asynchronous exception.

The exceptions caused by events or faults unrelated to the program and beyond the control of program are called asynchronous exceptions.

Exception Handling construct. (try, catch, throw).

try:

The part of the code that can generate exception or call of function that generates exception should be placed within 'try' block. It also contains throw.

catch:

The part of the code to handle appropriate exceptions should be placed within catch block.

⇒ The condition for exception is indicated by throwing the exceptions.
⇒ The try block must immediately followed by catch block.
⇒ There can be multiple catch block that can handle exceptions of different type and each catch block must

immediately be followed by previous catch block.

⇒ Any type of exceptions can be thrown but user-defined type or objects is better to be thrown as it can provide more information.

Syntax:

```
try
{
  if (error-occuring code)     // raising exception.
   throw variable / object
}
catch (type_id1)
 {
    // action for handling exception)
  }
 catch (type-id2)
  {
   // action for handling exception.
  }.
```

Steps taken During Exception Handling.

1. Hit the exception.
2. Throw the exception.
3. Catch the exception.
4. Handle the exception.

Try block throwing an exception.

```
#include <iostream>
using namespace std;
int main()
 {
  int a,b;
  cout <<"Enter values of a and b";
  cin>>a;
  cin>> b;
  int x = a-b;
```

```
try
  {
    if (x != 0)
    {
      cout << "Result (a/x) = " << a/x << "\n";
    }
    else
    {
       throw (x);
    }
  }
  catch (int i)        // catches the exception.
  {
  cout << " Exception caught : x = " << x << "\n";
  }
 cout << "END";
 return 0;
 }.
```

Multiple exception handling.

```
# include <iostream>
 using namespace std;
 void test (int x)
   {
     try
      { if (x == 1) throw x;    //int
        else
            if (x==0) throw 'x';   //char
        else
            if (x == -1) throw 1.0 ; // double
        cout << " End of try block";
      }
     catch (char c)
       {
        cout << " Caught an character";
       }
     catch (int m)
            {cout << " caught an integer "
             }
     catch (double d)
          {
           cout << " Caught a double"
          } }
```

```
int main()
{
cout << "Testing multiple catches";
cout << "x == 1";
test(1);
cout << "x == 0";
test(0);
cout << "x == -1";
test(-1);
cout << "x == 2";
test(2);
return 0;
}
```

Catching all exceptions.

```
try {
  ...
  ...
}
catch (...)
{
cout << "Caught all exceptions";
}
```

Advantage over Conventional Error Handling

Conventional error handling is inconvenient because every call must surrounded with an if...else statements to handle error or call the error routine. This increases size of program and makes code bulky. Exception handling separates the error handling code from other code making program more redable.

Rethrowing Exceptions.

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

```
# include <iostream>
using namespace std;
void divide (double x, double y)
    {
    cout << "Inside function \n";
    try {
```

```
if (y == 0.0)
        throw y;        // Throwing double.
else
    cout << "Division = " << x/y ;
 }
catch (double)
{                       // catch double.
 cout <<" Caught double inside function ";
 throw;                 // Rethrowing double.
 }
 cout << " End of function";
}
int main()
 {
 cout << " Inside main \n";
 try
    {
      divide (10.5, 2.0);
      divide (20.0, 0.0);
    }
  catch (double)
     {
      cout << " Caught double inside main ";
     }
   cout << " End of main";
  return 0;
  }
```

Exceptions with Arguments.

→ Sometimes application demands more information about cause of exceptions.

→ Hence object which carries more description about its design & specification is passed as arguments.

```
#
class myException
   {
   public:
        int number;
        char description [20];
```

```
myException()
 {
 number = 0;
 description[0] = '\0';
 }
MyEx myException(int n, char *desc)
    {
     number = n;
     strcpy(description, desc);
     }
};

double sqroot(double n)
{
if (n<0)
    throw myException(1, "Error: negative number");
else
    return sqrt(n);
}
int main()
  {
  double dn = 23.5;
  try
    {
     double sqrt = sqroot(dn);
     cout << "Square root:" << sqrt;
    }
catch(myException e)
   {
   cout << "Error occured while performing sq.root";
   cout << "Error Number:" << e.number;
   cout << "Error Description:" << e.description;
   }
}
```

Exception Specification for a function.

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding throw list.

Syntax:

```
return_type (arg_list) throw(type_list)
{
...//body
}
```

Eg:

```
Void func1() ; // can throw any exceptions.
Void func2() throw(); // can't throw any exceptions.
Void func3() throw(x); // throws only x exceptions. and derived from x
Void func4() throw(x,y); // throws exceptions x and y. and derived from x and y.
```

Handling uncaught and unexpected exceptions.

```
#include <exception>
#include <iostream>
using namespace std;
void myhandler()
  {
  cout << "Inside new terminate handler";
  abort();
  }
int main()
 set_terminate(myhandler);
 try
   {
    cout << "Inside try block";
    throw 100;
   }
 catch (char a)    //won't catch an int exception
   {
    cout << "Inside catch block";
   }
  return 0;
}
```