**What is template? What is the need of templates? How can you define a template? (OR)**

**Explain about generic programming.**

**(OR)**

**Explain about class templates with an example.**

**(OR)**

**Explain about member function templates with an example.**

The template provides generic programming by defining generic classes. A function that works for all C++ data types are called generic function.

Uses:

- A template is a technique that allows a single function or class to work with different data types.
- A single function can accept different types of values.
- Templates increases the flexibility of the program
- Template provides flexibility.

## Class Template

A class template can be defined as follows:

```
template <class T>
class  name_of_the_class
{
        Data members
        Member functions
}
```

template <class T> tells the compiler that the following declaration can use the template class data type. "T" is the variable of template type.

**Example**

```
# include <iostream.h>
template<class T> class
Data
{
        public:
           Data(T c)
           {
                cout<<"\n value = "<<c;
                cout<<"\tsize = "<<sizeof(c)<< " byte(s)";
           }
};

int main()
{
        Data <char> c('A');
        Data <int> i(100);
        Data <double> f(23.45);
}
```

**Output**

```
Value =A        Size = 1 byte(s)
Value =100     Size = 2 byte(s)
Value =23.45   Size = 8 byte(s)
```

**Class templates with more parameters**

The class templates can contain one or more parameters of generic type.

```
        template <class T1, class T2>
        {
                Data members
                Member functions
        }
```

**Example**

```
# include <iostream.h>
template<class T1, class
T2> class Data
{
        public:
           Data(T1 c, T2 d)
           {
                 cout<<"\n C="<< c << "\t D="<<d;
            }
};

int main()
{
        Data <char, double> c('D',10.5);
        Data <int, double> i(100, 23.456);
        Data <double,char> f(23.345,'K');
        Data <char, char*> k('R', "Rama");
        return 0;
}
```

**Output**

```
C= D C=        D= 10.5
100 C=         D= 23.456
23.345 C=      D= K
R              D= Rama
```

**Explain about functions templates with an example.**

**(OR)**

**Write about generic functions with an example.**

**(OR)**

**How a function template can work?**

Like class templates, function templates can also be defined as follows:

```
        template <class T>
        return type functions name (arguments of type T)
        {
                Statements;
        }
```

After compilation, the compiler cannot guess the type of data that a template function can work. When the template function is called, at that moment, the type of arguments passes to the template function replaces with the identical type. This process is called **instantiating**.

**Example**

```
# include <iostream.h>
template <class T>
T Max (T a, T b)
{
  return a > b ? a:b;
}
int main ()
{
        int i = 39;
        int j = 20;
        cout << "Max("<<i<<","<<j<<"): " << Max(i, j) <<
        endl; double f1 = 13.5;
        double f2 = 20.7;
        cout << "Max("<<f1<<","<<f2<<"): " << Max(f1, f2) <<
        endl; char c1 = 'Z';
        char c2 = 'A';
        cout << "Max("<<c1<<","<<c2<<"): " << Max(c1, c2) <<
        endl; return 0;
}
```

**Output**

```
Max (39, 20): 39
Max (13.5, 20.7): 20.7
Max (Z, A): Z
```

**Function template with more arguments:**

```
#include <iostream.h>
template<class T>
void add(T a, T b)
{
        T c;
        c=a+b;
        cout<<"\nSum="<<c;
}
int main()
{
        int a=10, b=20;
        add(a,b);
        double p=10.7,q=20.5;
        add(p,q);
        char c='A', d='D';
        add(c,d);
        return 0;
}
```

**Output**

```
Sum = 30
Sum = 31.2
Sum = á
```

**How can you overload a function template? Explain with an example.**

A Template function can also support overloading mechanism. It can be overloaded by normal function or template function. The compiler observes the following rules for choosing an appropriate function:

1. Searches for an accurate match of functions; if found, it is invoked.
2. Searches for a template function through which a function that can be invoked with an accurate match can be generated; if found, it is invoked
3. In case no match found, an error will be reported.

**Example:**

```
# include <iostream.h>
template<class T> void
show(T c)
{
        cout<<"\nTemplate variable C="<<c;
}
void show (int i)
{
        cout<<"\nInteger variable F ="<<f;
}
int main()
{
        show('A');
        show(10);
        show(23.45f);
        show(23.678);
        show("Rama");
       return 0;
}
```

**Output**

```
Template variable C= A
Integer variable F = 10
Template variable C= 23.45
Template variable C= 23.678
Template variable C= Rama
```

**Bubble Sort Using Function Templates**

```
# include <iostream.h>
template<class T> void
bsort(T a[], int n)
{
  for(int i=0;i<n-1;i++)
        {
            for(int j=i+1;j<n;j++)
              if(a[i] > a[j])
              {
                T temp = a[j];
                a[j] = a[i];
                a[i] = temp;
            }
        }
}
int main()
{
        int i;
        int a[5] = {4,1,5,-4,3};
        float f[5] = {4.4,1.0,5.7,-4.5,3.3};
        char c[5]={'A','N','A','N','D'};
        bsort(a,5);
        bsort(f,5);
        bsort(c,5);
        cout<< "\nSorted list:";
        for(i=0;i<5;i++)
                cout<< a[i]<<"  ";
        cout<< "\nSorted list:";
        for(i=0;i<5;i++)
                cout<< f[i]<<"  ";
        cout<< "\nSorted list:";
        for(i=0;i<5;i++)
                cout<< c[i]<<"  ";
        return 0;
}
```

**Output**

```
Sorted list: -4   1   3   4   5
Sorted list: -4.5   1.0   3.3   4.5   5.7
Sorted list: A   A   D   N   N
```

**Difference between templates and macros**

1. Macros are not type safe; that is, a macro defined for integer operations cannot accept float data. They expanded with no type checking.
2. It is difficult to errors in macros.
3. In case a variable is pre-incremented or decremented, the operation is carried out twice.

Consider the following macro:

```
#include<iostream.h>
#define max(a) a+ ++a
int main()
{
        int i=10, c;
        c = max(10);
        cout<<c;
        return 0;
}
```

**Output:**

**22**

The macro defined in the above macro definition is expanded twice. Hence, it is a serious limitation of macros. The lenition of this program can be removed using a template as shown below:

```
#include <iostream.h>
template<class T>
T max(T k)
{
        ++k;
        return k;
}
int main()
{
        int a = 10, c;
        c = max(a);
        cout<<c;
        return 0;
}
```

**Output**

```
11
```

**What is meant by exception? What are its uses? Explain about principles of exception handling. (Types of exceptions)**

Exception handling is used to trap the errors like logical errors. An exception is an abnormal termination of the program due to errors occurred in runtime. Exception handling mechanism helps us to prevent the abnormal termination of the program.

(OR)

Exceptions are run time anomalies or unusual conditions that a program may encounter during execution.

(OR)

An Exception is a problem that arises during the execution of a program.

Anomalies such as

- Division by zero
- Access to an array outside of its bounds
- Running out of memory
- Running out of disk space

A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. It was not a part of original C++. It is a new feature added to ANSI C++.

Exceptions are of 2 kinds

1. Synchronous Exceptions
   - Out of range
   - Over flow
2. Asynchronous Exceptions: Error that are caused by causes beyond the control of the program
   - Keyboard interrupts

In C++ only synchronous exception can be handled.

The purpose of exception handling mechanism is to create a routine that detects and report exceptional circumstances so that appropriate action can be taken. The mechanism has error handling code that perform the following tasks:

- Find the problem (Hit the exception)
- Inform that an error has occurred (Throw the exception)
- Receive the error information (Catch the exception)
- Take corrective action (handle the exception)

**Explain about exception handling mechanism with an example.**

**(OR)**

**Explain about try, catch, throw keywords with an example.**

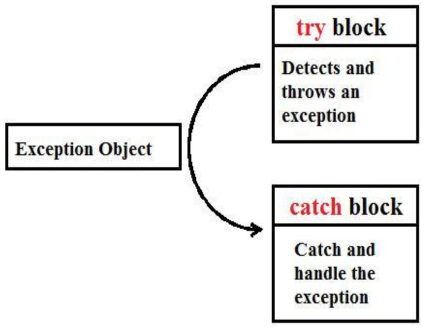
**(OR)**

**How exceptions can be handled? Explain with an example.**

In C++, Exception handling mechanism basically builds upon three keywords:

1. *try*
2. *throw*
3. *catch*

- When an exception is detected, it is thrown using a **throw** statement in the try block.
- A **catch** block defined by the keyword '*catch*' that catches the exception and handles it appropriately.


try block
Detects and throws an exception
Exception Object
catch block
Catch and handle the exception

**try keyword**

The try keyword is followed by a series of statements enclosed in curly braces. This keyword is used
at the starting of the exception.

Syntax:

```
try
{
        Statement 1;
        Statement 2;
        ------ -------
        Statement N;
}
```

**throw keyword**

The function of *'throw'* statement is to send the exception found. The *throw* statement is present

inside the try block. It has three forms and their syntax:

1. *throw(exception);*
2. *throw exception;*
3. *throw //used to re-throw a exception*

The argument object ***exception*** may be of any type, including constant. It is also possible to throw an object not intended for error handling. In any case, control is transferred to the *catch* statement.

**catch**

The catch block contains a series of statements enclosed in curly braces. It contains an argument of an exception type. When an exception is found the catch block is executed. In case of no exception is caught the catch block is ignored.

Syntax:

```
    catch (type arg)
    {
        statement 1;
        statement 2;
    }
```

- The type indicates the type of exception the catch block handles. The parameter *arg* is an *optional* parameter.
- The *catch* statement catches an exception whose type matches with the type of the 'catch' argument.
- If the parameter in the catch statement is named, then the parameter can be used in the exception handling code.
- If a catch statement does not match the exception it is skipped.

**Example:** program for divide by zero.

```
# include <iostream.h>
int main( )
{
        int a,b;
        cout<<"Enter values of a and b: ";
        cin>>a>>b;
        try {
            if(b!=0)
                cout<<"Result of
            (a/b):"<<a/b; else
               throw b;
        }
        catch(int e)
       {
           cout<<"Divide by zero error due to b = "<<e;
        }
}
```

**Output1:**

```
Enter values of a and b: 4  2
Result (a/b) : 2
```

**Output2:**

```
Enter values of a and b: 3  0
Divide by zero error due to b = 0
```

**How can you define multiple catch statements? Explain with an example.**

It is also possible to define multiple *catch* blocks. In the *try* block such programs also contain multiple *throw* statements based on certain conditions. The format of multiple *catch* statements is as follows:

```
        try
        {
                throw exception;
        }
        catch(type1 arg)
        {
                 // catch block 1
        }
        catch(type2 arg)
        {
                // catch block 2
        }
                ...     ...     ...
        catch(typeN  arg)
        {
                // catch block N
        }
```

Whenever an exception is thrown, the compiler searches for an appropriate matching catch block. If found, the matching catch block is executed. If no match catch block is found then the program is terminated.

**Example**

```
#include <iostream.h>
void  test(int x)
{
   try     {
          if(x==1)  throw x;
          else if(x==0)  throw 'x';
          else if(x==-1)  throw 1.0;
   }
   catch(int i)
   {
        cout<<"\nCaught an integer";
    }
   catch(char c)
   {
          cout<<"\nCaught a character";
   }
   catch(double d)
   {
         cout<<"\nCaught a double";
   }
}
int main()
{
```

```
          test(1);
          test(0);
          test(-1);
          test(2);
          return 0;
 }
```

**Output:**

```
Caught an integer
Caught a character
Caught a double
```

**Is there any possibility to caught more than one exception by a single catch block? Explain with an example.**

It is possible to define *single or default* catch block for one or more exceptions of different types. In such a situation, a single *catch* block is used to catch the exceptions thrown by multiple *throw* statements .

Syntax:

```
          catch
          {
                    //  Statements for handling all exceptions
          }
```

**Example**

```
# include <iostream.h>

void test(int x)
{
           try {
                    if(x==1)  throw x;
                    else if(x==0)  throw 'x';
                    else if(x==-1)  throw 1.0;
          }
          catch(…)
          {
                    cout<<"\nCaught an Exception";
          }
}
int main()
{
          test(1);
          test(0);
          test(-1);
          return 0;
}
```

**Output**

```
Caught an Exception
Caught an Exception
Caught an Exception
```

**Explain about re-throwing exception with an example.**

An exception is thrown from a catch block is called as re-throwing exception. The throw statement without any argument re-throws the exception.

Ex:

```
#include <iostream.h>
void test()
{
        try
        {
                throw "hello";
        }
        catch(char const *p)
        {
                cout<<"\n caught the exception";
                throw;
        }
}
int main()
{
        cout<<"\n in main";
        try
        {
                test();
        }
        catch(char const *p)
        {
                cout<<"\n caught in main";
        }
        cout<<"\n main end";
        return 0;
}
```

**Output:**

```
In main
caught the exception
caught in main
main end
```

**Write about specified exceptions with an example.**

It is also possible to restrict a function to throw certain type of exceptions by adding a throw list clause to the function definition.

**Syntax:**

```
        type function(arg-list) throw(type-list)
        {
                … … …
                … … …
        }
```

- The type-list specifies the type of exception that may be thrown.

- Throwing any other kind of exception will cause abnormal program termination.
- If you want to prevent a function from throwing no exception, you may do so by making the type-list empty.

*throw();*

Example

```
# include <iostream.h>
void test(int x) throw(int, double)
{
   if(x==1)  throw x;
         else if(x==0)  throw 'x';
         else if(x==-1)  throw 1.0;
}
int main()
{
   try
   {
      test(0);
      test(-1);
      test(1);
   }
   catch(char c)
   {
            cout<<"\nCaught a character";
   }
   catch(int  i )
   {
            cout<<"\nCaught an integer";
   }
   catch(double d)
   {
            cout<<"\nCaught a double";
   }
   return 0;
}
```

**Output**

```
terminate called after throwing an instance of 'char'
```