Unit→7

generic→ to operate on any data type, the type required being passed as a parameter.

Date.
Page No.

Function Templates & Exception Handling

⊛ Function templates:

Function templates are special functions that can operate with generic (comprehensive) types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

Template is simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types.

For example:- A mul() that can be used for multiplying integer, float, double etc. data types with the help of same one program. This means mul function (mul()) will accept any kind of data and can operate on it.

The main importance of the template is:

1) It eliminate the code duplication for different types and thus makes the program development easier and managable.
2) It allows us to generate a family of classes or functions to handle different data types.

If we create a specific class from class template that is known as the template class and the process of creating a template class is known as instantiation.

The general form of function template is:

(as with template type)

```
template<class T>
returntype function_name(arguments of type T)
{
    // Body of function with type T wherever
       appropriate
    ...
}
```

(keyword)

(T is a type like other variables we can keep it as our choice type)

(In <class T> we can replace this by typename also)

Date.
Page No.

Example 1 Program to display largest among two numbers of any data type using function templates.

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

template<class T>
T large(T n1, T n2){
        return (n1>n2)?n1:n2;
}

int main(){
    int i1, i2;
    float f1, f2;
    char c1, c2;

    cout<<"Enter two integers: \n";
    cin>>i1>>i2;
    cout<<large(i1,i2)<<" is larger."<<endl;

    cout<<"\n Enter two floating-point numbers: \n";
    cin>>f1>>f2;
    cout<<large(f1,f2)<<" is larger."<<endl;

    cout<<"\n Enter two characters: \n";
    cin>>c1>>c2;
    cout<<large(c1,c2)<<" has larger ASCII value.";

    return 0;
}
```

Date.
Page No.

⊛ Function Templates with Multiple Parameters:

We can use more than one generic data type in the template statement, using a comma-seperated list as shown below:

```
template <class T1, class T2, ...>
return_type function_name (arguments of types T1, T2, ...)
{
    ... (Body of function.
}
```

Program that we discussed in Example 1 is the example of function template with multiple parameters since we used more than one parameter in the function template.

⊛ Class templates:

A class template starts with a template keyword and a number of arguments it can accept. This is a definition of a class template in C++.

```
template <class T>
class add{};    // class template add.
```

Example:

```
#include <iostream>
using namespace std;
template <class t>
class greater { t a, b;
    public:
        greater (t first, t second){
            a = first;
            b = second;
        }
```

two arguments or more for multiple

template type of variable declaration a and b

Date
Page No.

```
t max (){
        t val;
        val = a>b ? a:b;
        return val;
      }
};

int main (){ clrscr();
        greater<int>obj1 (5,4);
        greater<float>obj2 (5.5, 16.5);
    cout<<"larger number = "<<obj1.max()<<endl;
    cout<<"larger number = "<<obj2.max()<<endl;
      }
```

Lesser imp

⊛ Template and Inheritance (Inheriting from a template class):

It is possible to inherit from a template class. All the usual rules for inheritence and polymorphism apply.

If we want the new, derived class to be generic it should also be a template class and pass its template parameter along to the base class.

When inheritance is applied with the template class, it helps to compose a hierarchical data structure known as container class. The following declaration illustrates the derivation of a class using template featured base class.

```
template<class T,...>
class XYZ { // Template type data members
                and member functions }
template<class T,...>
class ABC : public XYZ<T,...> { // Template type data members
                and member functions.
}
```

Date.
Page No.

Example:- Program to derive a class using template base class.

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

template<class T>
class one{
        protected:
            T x,y;
        void display (){ cout<<x; }
    };
template <class S>
class two: public one<S>{
        public:
            two (S a, S b, S c){
                x=a;
                y=b;
                z=c;
            }

    void show(){
    cout<<"\n x="<<x<<"y="<<y<<"z="<<z;
        }

    };

void main (){
        two<int> i(2,3,4);
        i.show ();
        two <float > f(1.1,2.2,3.3);
        f.show ();
    }
```

Output

```
x=2 y=3 z=4
x=1.1 y=2.2 z=3.3
```

Date.
Page No.

Exceptional Handling

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

Exceptions provide a way to transfer control from one part of program to another. C++ exception handling is built upon three keywords: try, catch and throw.

throw → A program throws an exception when a problem shows up. This is done using throw keyword.

catch → A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try → A try block identifies a block of code for which particular exceptions will be actived. It is followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is referred to as protected code, and the syntax for using try/catch is as follows:-

```
try {
   // protected code
}
catch (ExceptionName e1) {
   // catch block
}
catch (ExceptionName eN) {
   // catch block
}
```

Date.
Page No.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs.

```
double division(int a, int b) {
    if (b==0) {
        throw "Division by zero condition!";
    }
    return (a/b);
}
```

Catching Exceptions

The catch block following the try block catches any exception. We can specify what type of exception we want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch. The following code will catch an exception of ExceptionName type.

```
try { //protected code
}
catch (ExceptionName e) {
    //code to handle ExceptionName exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

Date.
Page No.

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

double division(int a, int b){
        if(b==0){
            throw "Division by zero condition!";
        }
        return (a/b);
    }

int main(){
    int x=50;
    int y=0;
    double z=0;
    try{
        z=division(x,y);
        cout<<z<<endl;
    }
    catch(const char*msg){
        cerr<<msg<<endl;
    }
    return 0;
}
```

Because we are raising an exception of type const char*, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would provide the following result:

Division by zero condition!

Date.
Page No.

Multiple Exceptions, Exceptions with arguments:

A single try statement can have multiple catch statements which is called multiple exception.

Example:

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

int main(){
    int choice;
    try{
    cout<<"Enter any choice:";
    cin>>choice;
  if(choice==0) cout<<"Hello Baby!"<<endl;
  else if(choice==1)throw(100);//throw integer value
  else if(choice==2) throw('x');//throw character value
  else cout<<"Bye !!"<<endl;
    }
    catch(int a){cout<<"Integer Exception"<<endl;}
  catch(char b){cout<<"Character Exception"<<endl;}
    return 0;
}
```

Output:

First Run → Enter any choice:0
Hello Baby!

Second Run → Enter any choice:1
Integer Exception

Third Run → Enter any choice:2
Character exception

Fourth Run → Enter any choice:6
Bye!!

if value out of program's used given

Date.
Page No.

⊛ Use of Exception handling:

Exception handling is used for the following things:

i) To develop a reliable and robust system.

ii) It is used when one system performs in a way that is not desired.

iii) It prevents the seperation of error handling code from the normal program code.

iv) Using this mechanism we can prevent user from seeing complex technical error messages.

v) Using this functions/Methods can handle any ~~exp~~ exceptions they choose.

vi) As the exception is passed back up the stack of calling function we can handle errors at any place we choose.