EXCEPTION HANDLING

10.1 Multiple catch statement

```
#include<iostream.h>
#include<conio.h>
void num(int k)
{
try
{
if(k==0)throw k;
else
if(k>0)throw 'p';
else
if(k<0)throw .0;
}
catch(char g)
{
        cout<<"\n caught a positive value";
}
catch(int j)
{
        cout<<"\n caught a null value";
}
catch(double f)
{
          cout<<"\n caught a negative value";
}
}
int main()
{
   num(0);
   num(-5);
   num(-1);
   getch();
}
```

10.2 Catching Multiple Exception\ Generic catch block

```
#include<iostream.h>
#include<conio.h>
void num(int k)
{
    try
    {
    if(k==0)throw k;
```

```
    else
    if(k>0)throw 'p';
    else
    if(k<0)throw .0;
    cout<<"\n ---------TRY BLOCK-----------";
    }
    catch(...)
    {
         cout<<"\n caught an exception";
    }

}
int main()
{
   num(0);
   num(5);
   num(-1);
   getch();
}
```

10.3 Rethrowing an Exception

```
#include<iostream.h>
#include<conio.h>
void sub(int j,int k)
{
    try
    {
        if(j==0)
        throw j;
        else
        cout<<"\n subtraction:"<<j-k;
    }
    catch(int)
    {
         cout<<"caught a null value";
         throw;
    }
}
int main()
{
   try
   {
      sub(8,5);
      sub(0,8);
   }
   catch(int)
```

```
   {
        cout<<"caught a null value inside main";
   }
   getch();
}
```

10.4 Specifying Exception

```
#include<iostream.h>
#include<conio.h>
class num
{
     public:
          float a;
          void get()
          {
              cout<<"enter a number:";
              cin>>a;
          }
          friend void result(num &,num &);
};
void result(num &n,num &m)
{
     float r;
     r=n.a/m.a;
     cout<<"division is"<<r;
}
int main()
{
   num o1,o2;
   o1.get();
   o2.get();
   try
   {
          if(o2.a==0)
          throw (o2.a);
          else
          result(o1,o2);
   }
   catch(float k)
   {
          cout<<"exception caught:"<<k;
    }
    getch();
}
```

TEMPLATES

11.1 Introduction

- The templates provide generic programming by defining generic classes.
- A function that works for all C++ data types is called as a generic function.
- In templates generic data types are used as arguments and they can handle a verity of data types.
- Templates helps the programmer to declare a group of functions or classes. When used with functions they are called function templates.

  Ex:

  We can create a template for function square to find the square of an data type including int , float, long & double. The templates associated with classes are called as class template.

11.2 Need of Template:

- A template is a technique that allows using a single function or class to work with different data types.
- Using template we can create a single function that can process any type of data that is the formal arguments of template functions are of template type.
- So they can accept data of any type such as int, long, float, etc. Thus a single function can be used to accept values of different data type.
- Normally we overload functions when we need to handle different data type. But this approach increases the program size and also more local variables are created in memory.
- A template safely overcomes all this limitations and allows better flexibility to the program.

  Ex:

```
#include <iostream.h>
#include <conio.h>
template <class T>
class data
{
        Public:
                data (T c)
                {
                        cout<<c<<sizeof(c);
                }
};
void  main()
{
        clrscr();
        data <char> h('A');
        data <int > i(100);
        data <float> j(3.12);
        getch();
}
```

11.3 Normal Function Template:

- A normal function is not a member function of any class.
- The difference between a normal and member function is that :-
  - ✓ Normal functions are defined outside the class.
  - ✓ They are not members of any class and hence can be invoke directly without using object and dot operator.
  - ✓ But member functions are class members and hence need to be invoke using objects of the class to which they belong.

Ex:

/* Write a program to find square of a number using normal template function*/

```
#include <iostream.h>
#incude <conio.h>
template <class T>
void square (T x)
{
        cout<<"square="<<x*x;
}
void main()
{
        clrscr();
        int i; char j; double k;
        cout<<"Enter values for i, j & k :";
        cin>>i>>j>>k;
        square(i);
        square(j);
        square(k);
        getch();
}
```

11.4 Member Function Template:

```
#include <iostream.h>
#include <conio.h>
template <class T>
class sqr
{
        public:
                sqr (T c)
```

```
        {
                cout<<"Square="<<c*c;
        }
};
```

11.5 Working of Function Templates:

- After compilation the compiler can’t case with which type of data the template function will work.
- When the template function called at that moment from the type of argument passed to the template function, the compiler identified the data type.
- Every argument of the template type is then replaced with the identified data type and this process is called as instantiating.
- So according to different data types respective versions of template functions are created.
- The programmer need not write separate functions for each data type.

```
/* Write a program to define data members of template types */
#include <iostream.h>
#include <conio.h>
template <calss T>
class data
{
        T x;
        public:
                data (T u)
                {
                        X=u;
                }
                void show (T y)
                {
                        cout<<x<<y;
                }
};
void main()
{
        Clrscr();
        data<char> C (‘B’);
        data<int> i (100);
        data<double>d (48.25);
        c.show (‘A’);
        i.show (65);
        d.show (68.25);
}
```

/* Write a program to define a constructor with multiple template variables */

```
#include <iostream.h>
#include <conio.h>
template <class T1,class T2>
class data
{
        Public:
                Data(T1 a,T2 b)
                {
                        cout<<a<<b;
                }
};
void main()
{
        clrscr();
        data <int, float> h(2,2.5);
        data <int, char> i(15,'c');
        data < float, int > j(3.12,50);
}
```

11.6 Overloading of Template Functions :

- Template functions can be overloaded by normal function or template function.
- While invoking these functions an error occurs if no accurate match is made.
- No implicit conversion is carried out in parameters of template function.

  Ex:

```
#include <iostream.h>
#include <conio.h>
template <calss T>
void show (T c)
{
        cout<<"Template variable c="<<c;
}
void show(int f)
{
        cout<<"Integer variable f="<<f;
}
void main()
{
        clrscr ();
        show ('c');
        show (50);
        show (50.25);
```

```
}

/* Member function templates */

#include <iostream.h>
#include <conio.h>
template <calss T>
class data
{
        public:
                data (T c);
};
template <class T>
data <T> :: data (T c)
{
        cout<<"c="<<c;
}
void main()
{
        clrscr();
        data <char> h ('A');
        data <int> i (100);
        data <float> j (3.12);
}
```

11.7 Exception Handling With Class Template:

```
#include <iostream.h>
#include <conio.h>
class sq{};
template <calss T>
class square
{
        T s;
        public:
                square (T x)
                {
                        sizeof(x)==1 ? throw sq() : s=x*x;
                }
        void show()
        {
                cout<<"square="<<s;
        }
};
```

```
void main()
{
        try
        {
                square <int> i(2);
                i.show();
                square <char> c('c');
                c.show();
        }
        catch(sq)
        {
                cout<<"Square of character can't be calculated.";
        }
}
```

11.8 Class Templates with Overloaded Operators :

```
#include <iostream.h>
#include <conio.h>
template <calss T>
class num
{
        private:
                T number;
        public:
                num()
                {
                        number=0;
                }
                void input()
                {
                        cout<<"Enter a number :";
                        cin>>number;
                }
                num operator +(num);
                void show()
                {
                        cout<<number;
                }
};
template <class T>
num <T> num <T> :: operator +(num <T>c)
{
        num <T> tmp;
```

```
        tmp.number = number + c.number;
        return (tmp);
}
void main()
{
        clrscr();
        num <int> n1,n2,n3;
        n1.input();
        n2.input();
        n3=n1+n2;
        cout<<"n3=";
        n3.show();
        getch();
}
```

11.9 Class Template and Inheritance:

- The template class can also act as base class.
- There are 3 cases:-
    - ✓ Derive a template class and add new member to it, the base class must be of template type.
    - ✓ Derive a class from non-template class add new template type members to derive class.
    - ✓ Derive a class from a template base class and omit the template features in the derive class.
- This can be done by declaring the type while deriving the class.
- All the template based variables are substituted with basic data type.

  How To Derive A Class Using A Template Base Class:

```
#include <iostream.h>
#include <conio.h>
template <calss T>
class one
{
        protected:
                T x,y;
        void display()
        {
                cout<<x<<y;
        }
};
template <class S>
class two : public one <S>
{
        S z;
        public:
                two (S a, S b, S c)
                {
```

```
            x=a; y=b; z=c;
        }
        void show ()
        {
            cout<<x<<y<<z;
        }
};
void main()
{
    clrscr();
    two <int> i (2,3,4);
    i.show();
    two <float> f (1.1,2.2,3.3);
    f.show();
}
```

11.10 Difference between Templates and Macros:

- Macros are not type safe. That is a macro defined for integer operation can't accept float data.
- They are expanded with no type checking.
- It is difficult to find errors in macros.
- In case a variable is post incremented (a++) or post decremented (a--) the operation is carried out twice for a macro.

11.11 Guidelines For Templates:

- Templates are applicable when we want to create type secure classes that can handle different data types with same member functions.
- The template classes can also be involved in inheritance.
- The template variables allow us to assign default values.

```
template <class T, int x=20>
class data
{
    T num[x];
}
```

- All template arguments declared in the template argument list should be used for definition of formal arguments. Otherwise it will give compilation error.
  - ✓
    ```
    template <class T>
    T show()
    {
        return x;
    }
    ```
  - ✓
    ```
    template <class T>
    void show(int y)
    ```

```
{
        T tmp;
}
```

NAMESPACES

12.1 Namespace Scope:

- C++ allows variables with different scopes such as local, global, etc with different blocks and classes. This can be done using keyword namespace.
- All classes, templates & functions are defined inside the name space std.
- The statement using namespace std tells the compiler that the members of this namespace are to be used in the current program.

12.2 Namespace Declaration:

- It is same as the class declaration, except that the name spaces are not terminated by (;) semicolons.

Ex:

```
namespace num
{

   int n;
   void show (int k)
   {
         cout<<k;
   }
}
num :: n=50;
using namespace num
n=10;
show (15) ;
```

12.3 Accessing elements of a name space:

1) Using Directive
2) Using Declaration

1) Using Directive:

- This method provides access to all variables declared with in the name space.
- Here we can directly access the variable without specifying the namespace name.

Ex:

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

```
namespace num
{
    int n;
    void show()
    {
            cout<<"n="<<n;
    }
}
int main ()
{
    using namespace num;
    cout<<"Enter a number :";
    cin>>n;
    show();
    getch();
}
```

2) **Using Declaratives**:

- Here the namespace members are accessed through scope resolution operator and name space name.

```
#include <iostream.h>
#include<conio.h>
namespace num
{
        int n;
        void show()
        {
                cout<<"n="<<n;
        }
}
int main()
{
        cout<<"Enter value of n :";
        cin>>num :: n;
        num :: show();
        getch();
}
```

12.4 Nested namespace:

- When one name space is declared inside another namespace it is known as nested namespace.

12.5 Anonymous namespaces:

- A namespace without name is known as anonymous namespaces.
- The members of anonymous namespaces can be accessed globally in all scopes.
- Each file posses separate anonymous namespaces.

```
#include<iostream.h>
#include<conio.h>
namespace   num
{
    int j=200;
    namespace num1 // nested namespace
    { int k=400;}
}
namespace //anonymous namespace
{int j=500;}
void main()
{
    cout<<" j= "<<num :: j;
    cout<<" k= "<<num :: num1 :: k;
    cout<<" j= "<<j;
}
```

12. 6 Function in namespace:

```
#include<iostream.h>
#include<conio.h>
namespace fun
{
        int add (int a,int b)
        {
                Return (a+b);
        }
        int mul (int a,int b)
}
int fun :: mul (int a, int b)
{ return (a*b);}
int main()
{
        using namespace fun;
        cout<<"Addition :"<<add(20,5);
        cout<<" Multiplication :"<<mul(20,5);
}
```

12.7 Classes in namespace:

```
#include<iostream.h>
#include<conio.h>
namespace A
{
	class num
	{
		private:
			int t;
		public:
			num (int m)
			{
				T=m;
			}
			void show()
			{
				Cout<<t;
			}
	};
}
void main()
{
//indirect access using scope access operator
	A :: num n1 (500);
	n1.show();
//direct access using directive
	using namespace A;
	num n2(800);
	n2.show();
}
```

12.8 Namespace Alias:

- It is designed to specify another name to existing namespace.
- It is useful if the previous name is long.
- We can specify a short name as alias and call the namespace members.

```
#include<iostream.h>
#include<conio.h>
namespace number
{
	Int n;
	Void show()
	{
		Cout<<” n= “<<n;
```

```
        }
}
Int main()
{
        namespace num=number;
        num :: n = 200;
        number :: show ();
}
```

- The standard namespace std contains all classes, templates and functions needed by a program.

12.9 Explicit Keyword :

- It is to declare class constructors to be explicit constructors.
- Any constructor called with one argument performs implicit conversion in which the type received by the constructor is converted to an object of the class in which the constructor is define. This conversion is automatic.
- If we don't wants such automatic conversion to take place. We may da so by declaring the onr argument constructor as explicit.

Ex:

```
class ABC
{
        Int m;
        public:
                explicit ABC (int i)
                {
                        m=i;
                }
________________
    ________
       __
};
void main()
{
        ABC abc1 (100);
        ABC abc2=100;
}
```

12.10 Mutable keyword:

- If we want to create a constant object or function but would like to modify a particular data item only, we can make that particular data item modifiable by declaring it as mutable.

Ex:

```
Class ABC
{
        private:
                mutable int m;
        public:
                explicit ABC (int x=0)
                {
                        m=x;
                }
        void change () const
        {
                m=m+10;
        }
        int display () const
        {
                return m;
        }
}
void main()
{
        const ABC abc (100);
        abc.display();
        abc.change();
        abc.display();
}
```

12.11 Manipulating Strings:

- A string is a sequence of character we use null terminated character arrayes to store and manipulate strings. These strings are called C strings or C style string.
- ANSI standard C++ provides a new class called string. This class is very large and includes many constructors, member functions and operators.
- For using the string class we must include the string data type in the program.

Creating String Objects:

1) string s1; (null string) //using constructor with no argument
2) string s2("xyz"); //using one argument constructor
3) s1=s2; //assigning string objects
4) cin>>s1; //reading through keyboard
5) getline (cin s1);

12. 12 Manipulating String Objects:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
	string  s1("12345");
	string  s2("abcde");
	cout<<s1<<s2;
	s1.insert(4,s2);
	cout<<s1;
	s1.erase(4,5);
	cout<<s1;
	s2.replace(1,3,s1);
	cout<<s2;
}
```

12.13 Relational Operator:

- These operators are overloaded and can be used to compare string objects.

```
void main()
{
	string  s1("ABC");
	string  s2("XYZ");
	string  s3=s1+s2;
	if(s1 != s2)
		cout<<"s1 is not equal to s2.";
	if(s1 > s2)
		cout<<"s1 greater than s2.";
	else
		cout<<"s2 greater than s1.";
	int x=s1 . compare (s2);
	if(x==0)
		cout<<"s1 == s2";
	else if (x>0)
		cout<<"s1 > s2";
	else
		cout<<"s1 < s2";
}
```

12.14 Accessing Characters In String:

- They are used to access sub strings and individual characters of a string.

Ex:

```
int main()
{
string  s("one two three four");
for(int i=0;i<s.length();i++)
cout<<s.at(i);
for(int j=0;j<s.length();j++)
cout<<s[j];
int x1=s.find("two");
cout<<x1;
int x2=s.find-first-of('t');
cout<<x2;
int x3=s.find-last-of('r');
cout<<x3;
cout<<s.substr(4,3);
}
string  s1("Road");
string  s2("Read");
s1.swap(s2);
```