Templates

Lecture 21

Non-type parameters and default types for class templates

- Non-type parameters can have default arguments and treated as constant

```
template <class T,int constant>
class A
{  T x;
  public:
  A(T i) { x=i; }
  void display()
 { cout<<"\n X : "<<x; cout<<" Constant : "<<constant; }};
void main()
{ A<int,30> a(5);
  a.display(); }
```

Templates and Friends

- Friendship can be established between class template
  - global function
  - Member function of another class
  - Entire class

Example

```
template <class T>
class A
{  T x;
  public:  A(T i) { x=i; }
  void display() {cout<<"\n X : "<<x;}
  friend void f(A<int>); };
void f(A<int> a)
{ cout<<"\n Member : "<<a.x; }
void main()
{ A<int> a(5);
  a.display();
  f(a);
}
```

Example

```
class B;
template <class T>
class A
{  T x;
  public:
  A(T i) { x=i; }
  void display() {cout<<"\n X : "<<x;}
  friend class B; };
class B
{ public:
  void display(A<int> a) {cout<<"\n Inside friend class : "<<a.x;}
};
void main()
{ A<int> a(5);   B b;   a.display();  b.display(a); }
```

Templates and ***static*** members

- Each class template specialization instantiated from a class template has its own copy of each static data member of the class template

Exception handling

Exception

- Is an indication of a problem that occurs during a program execution

Exception handling

- Enables programmers to create applications that can resolve (or handle) exception
- Handling an exception allows a program to continue executing as if no problem had been encountered

Note : - Programs are more robust and fault-tolerant

Exception handling

Perform a task

If the preceding task did not execute correctly – perform error processing

Perform next task

If the preceding task did not execute correctly – perform error processing

Exception handling in C++

- try (keyword)
  - Program statements that we want to monitor for exceptions are contained in a try block
- catch
  - Exception is caught using catch and process

try and catch

```
try
{
   // try block – statements to monitor
}
catch(type1 arg)
{ // catch block – statement after exception }
catch(type2 arg)
{ // catch block – statement after exception }
```

throw

- throw statement generates the exception
- syntax

  throw *exception*
- This statement should be written in either try block or from any function called within try block
- If no catch for throw, then program terminates

Example

```
void main( )
{ cout<<“ The program starts ”;
try {
   cout<<“ Inside try block”;
   throw 100;
   cout<<“ This will not execute” ;
}
catch(int k)
{ cout<<“ Caught”<<k ; } }
```

Example

```
void test( )
{
   cout<<“ Inside
    function”;
   throw 100;
}
```

```
void main( )
{ cout<<“ The program starts ”;
try {
   cout<<“ Inside try block”;
    test( );
   cout<<“ This will not
   execute” ;
}
catch(int k)
{ cout<<“ Caught”<<k ; } }
```

Catching class types

```
class myexp
{ private:
char str[20]; int what;
public:
myexp(char a[ ],int e)
{ strcpy(str,a); what=e; }
void display( )
{ cout<<str<<what; }
};
```

```
void main( )
{ int k; myexp
   e("Negative",0);
try
{
cin>>k;
If(k<0) throw e;
}
catch(myexp e)
{ e.display(); }}
```

Multiple catch

```
void main( )
{ int k ; cin>>k;
try {
  if (k<0)
   throw k;
  else
   throw “works”;
} catch(int j) { cout<<“ integer thrown”<<j ; }
catch(char *c) { cout<<“ char thrown”<<c; }
}
```

Exception and inheritance

Order of the catch statement should be as follow

- Write all the derived classes first
- Write the base class

Example

```
class B { public : void display() {cout<<“B” ; } };
class D : public B { void display() {cout<<“D” ; } };
void main( )
{  D derived;
try { throw derived ; }
catch(B b) { cout<<“ Caught in base !!”; }
catch(D d) { cout<<“ will not execute !!”; }
```

Catching all exceptions

```
catch(…)
{  cout<<“ Catches all type of exceptions”; }
```

Restricting exceptions

```
// restrict functions throw types
void test(int k) throw(int, char, double)
{
// function body
}
```

Re-throwing an exception

```
try {
throw 100 ;
}
catch(int k)
{ cout<<“inside catch throw the exception again” ;
   throw }
```

The terminate( ) function

- Standard C++ library function
- Prototype – void terminate( )
- Called when there is no matching catch
- By default terminate( ) calls abort( )
- The unexpected( ) function is called when a function attempts to throw an exception that is not allowed by its throw list

Setting the Terminate and Unexpected handlers

```
void my_Thandler() { cout<<“ Terminate handler”;
   abort(); }
void my_Uhandler() { cout<<“ Terminate handler”;
   abort(); }
void main( )
{ set_terminate(my_Thandler);
set_unexpected(my_Uhandler);
……
}
```

Constructors

```
#include<stdexcept>
using namespace std; int status=1;
class mynew
    { private : int x;
    public: mynew()
     { try { int i=0; cout<<(1/i); x=10; status=1;}
     catch(...) { cout<<"Exception handled !!"; status=0;} }
     void display() { cout<<"\n Value of x : "<<x; }
     ~mynew() { cout<<"Destructor called" ; } };

    void main()
    {
     mynew m;
     if(status) m.display();
     else cout<<"\n Object not created properly !!"; }
```

Stack unwinding

- When an exception is thrown but not caught in a particular scope, the function call stack is unwound, and an attempt is made to catch the exception in the next outer try…catch block