Lecture 9

Templates

Namespaces

Exception handling

Templates
&
Namespaces

Objectives of this Session

- Templates
  - Identify need of Templates
  - Describe function Templates & class Templates
  - Distinguish between templates & macros
- Namespaces
  - Identify need of namespaces
  - State rules for using namespaces

Need of Templates

```
void swap(int &a,int &b) void swap (char &ch1,char &ch2)
{    int temp; {   char temp;
     temp = a;          temp = ch1;
     a = b ;        ch1 = ch2;
     b = temp;          ch2 = temp;
}     }

void swap (Complex &c1,Complex &c2)
{    Complex temp;
     temp = c1;
     c1 = c2;
     c2 = temp;
     Function
}
```

Function

↓

Function Overloading

↓

Generic Function

Templates

- With function overloading same code needs to be repeated for different data types which leads towards waste of time & space.
- Templates enable us to define generic functions or classes which avoids above repetition of code for different data types .
- Generally templates are used if same algorithm works well for various data types eg sorting algorithms.
- There can be function templates or class templates.

Function Template

template < class type > ret type FnName (parameter list)

template is a keyword used to create generic functions.

class type is a placeholder

```
// swap function using function template
template <class T> void swap (T &a, T&b)
{     {
    T temp;       int i = 10, ,j = 20;
    temp = a;          swap( i,j);
    a = b;        char x = 'A' , y = 'B' ;
    b = temp;          swap( x,y );
}          ……
//templates needs to be instantiated.       }
```

class Template

- Power of Templates is reusability of code.
- Eg
  - if Queue/Stack written as template or generic class, then you can have Queue/Stack of integers or floats or characters or objects ...

Stack as a template class

```
// part of stack.h
template <class T> class stack
{
    private :
      int m_size;
      T* m_pbuff;
      int m_top;
    public :
      stack ( );
      void push (T);
      T pop (void);
      ~ stack();
};
```

Stack as a template class

```
// part of stack.cpp
template <class T> stack <T> :: stack()   //constructor
{
    m_size = 10;
    m_top = -1;
    m_pbuff = new T [m_size];
}

// push()
template<class T> void stack <T> :: push(T val)
{
    assert ( m_top < m_size);
    m_pbuff [++m_top] = val;
}
```

Stack as a template class

```
// part of stack.cpp
// pop()
template<class T> T stack <T> :: pop()
{   assert(m_top > -1) ;
    return (m_pbuff [m_top--]);
}

template <class T> stack <T> :: ~stack() //destructor
{
    if (m_pbuff)
    delete[] m_pbuff;
}
```

Stack as a template class

```
// Part of main()
#include "stack.cpp" // include .cpp file
void main(void)
{
    stack <int> st1;
    st1.push(1);      st1.push(2);
    cout << "popped value is :"<< st1.pop();
    cout<< "popped value is :" << st2.pop();

    stack <char> st2;
    st2.push('A');
    st2.push('B');
    …
}
```

Namespaces

- Software development is a team effort, so it’s difficult to control names of variables, structures, classes, functions …
- Same variable names, structure name, class name leads towards re-declaration error
- Same situation for two or more functions with same signature in the same scope
- To avoid this use namespaces

Namespaces

```
// file a.h    // file b.h
void f();      class B
class A{
{    …
…    };
};   void f();
```

Compiler will throw re-declaration error for function f()

To avoid name clashes use namespaces

Namespace Syntax

namespace name

{

…

}

Namespaces

```
// file a.h    // file b.h
namespace space1    namespace space2
{    {
    void f();       class B
    class A         {
    {       …
    …          };
    };          void f();
}    }
```

While using Namespaces

```
main()
{
using namespace space1
…
using namespace space2
…
}
```

- using is resolved by compiler as a unique function as follows space1 :: f(), space2 :: f()

Namespaces

- C++ provides default global namespace
- Global namespace is implicitly declared and exists in every program
- All standard classes, objects, variables, functions, templates exists in this namespace
  - e.g. std :: cout
- Each user declared namespaces represent a distinct namespace scope

While using Namespaces …

- A namespace can be unnamed
  namespace
  {
  …
  }
- Unnamed namespace is unique to current file
- Unnamed namespaces is used to replace global static definitions

While using Namespaces …

- Namespace definition can appear only in global scope

```
void f()
{
 namespace err
 {
 …
 }
}
// local scope not allowed
```

- If classes are designed for reusability, namespaces should be used

using directive

- using directive exposes all names declared in a namespace to be in current scope

  namespace window

  {

  int val1 = 20;

  int val2 = 40;

  }
- window :: val1 = 10; // Access by ::
- using namespace window

  val2 = 30   // Access using directive

Exceptions

Objectives of this Session

- Exception Handling
  - Identify need of Exceptions handling.
  - State the C++ features for exception handling.

Industrial Grade Software

- Factors affecting robustness of software
  - program structure
  - logic or algorithm
  - Syntax & Data types
  - unexpected I/O
  - unusual but predictable problems

Handling Above Issues

- System crashes
- Inform user and exit gracefully (by exit housekeeping)
- Inform user and allow user to recover and continue
- Take corrective action and continue without disturbing user

What are exceptions ?

- Exceptions are runtime anomalies that a program may detect
- e.g.
  - Division by 0
  - Access to an array outside it’s bounds
  - Exhaustion of the free memory on heap

Exception Handling

- Exceptions in ‘C’ are handled through return & switch case constructs in caller function
- C++ provides built in features to raise and handle exceptions
- These language features activates a runtime mechanism to communicate exceptions between two unrelated portions of C++ program

C++ Features for Exception Handling

- Keywords : try, catch, throw
- try block is a block surrounded by braces in which exception may be thrown
- A catch block is the block immediately following a try block, in which exceptions are handled

How exceptions are used ?

```
try
{
    // something unusual but still predictable
}
catch (out of memory)
{
    // take some action
}
catch (File not found)
{
    // take other action
}
```

Exceptions : Execution Flow

```
try
{
// Calls to push, pop
functions
}
```

```
catch(Stack_Full)
{
      Error
      Messages
}
```

```
catch(Stack_Empty)
{
      Error
      Messages
}
```

```
throw Stack_Full()
// this throw expression is
part of push() function
```

```
throw Stack_Empty()
// this throw expression is
part of pop() function
```

Execution of catching exception

- When an exception is thrown, examine call stack.
- Stack is unwound,
- Destructors of the local objects on stack are invoked
- Steps 2 and 3 are continued till matching catch block is found.
- If matching catch block is not found till beginning of program i.e. main built in handler terminate() is called
- terminate() calls abort()

While using Exception Handling …

- Note
  - When an exception is raised, program flow continues after catch block.
  - Control never comes back to the point from where exception is thrown

Multiple catch blocks

- Execution is similar to switch-case
- Once a matched catch block signature is found, other catch blocks are not executed
- Order of catch blocks is important. Specific first and general at last.
- Most general is “Catch everything” indicated by catch(…)

casting Operators

- Explicit conversion is referred as cast
  - static cast
  - dynamic cast
  - const cast
  - reinterpret cast
- cast operators are sometimes necessary

- Explicit cast, allows programmer to momentarily suspend type checking

- Syntax
  cast_name<type>(expression);

casting Operators : const_cast

- casts away the constness of its expression
- e.g. const_cast

```
char *string_copy(char *);
const char  *pc_str;
char *pc = string_copy (const_cast<char*>(pc_str));
```

casting Operators : static_cast

- Any conversions which compiler performs implicitly can be made explicit using static_cast
- Warning messages for loss of precision will be turned off.
- reader, programmer and compiler all are made aware of fact of loss of precision.
- e.g. static_cast

```
double dval;
int ival;
ival += dval; // unnecessary promotion of ival to
 // double can be eliminated by using
 // explicit casting
ival += static_cast<int>(dval);
```

casting Operators : reinterpret_cast

- Complex <double> *pcom;
- Char *pc = reinterpret_cast < char*>(pcom)

- Reinterpret cast performs low level interpretation of bit pattern

- Is used to convert any data type to any other data type

- Most dangerous

Example

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

int main()
{
  int i;
  char *p = "This is a string";

  i = reinterpret_cast<int> (p); // cast pointer to integer

  cout << i;

  return 0;
}
```

casting Operators : dynamic_cast

- dynamic_cast operators are used to obtain pointer to the derived class

```
void company :: payroll (employee *person)
{
manager *pm = dynamic_cast<manager*>(person);
// if person at runtime refers to manager class, then dynamic cast is successful
if(pm)
// use pm to call bonus using pm -> bonus;
else
// use employee's member function
}
```

Example

```
class A { virtual void foo() {} };
class B : public A { ... };
void f(A* a)
{
B* b = dynamic_cast<B*>(a); // Will compile
B* b = static_cast<B*>(a); // Will compile
 }
```