Features of C++

- Default parameters
- Initializer list
- **explicit** constructor
- Constant member function
- Interface and Implementation
- Vectors, strings and pointers
- Function and class templates

```
/* A class for simulating an integer memory cell */
class IntCell
{
  public:
    /* Construct the IntCell. Initial value is 0. */
  IntCell()
    { storedValue = 0;}

  /* Construct the IntCell. Initial value
                             is initialValue. */
  IntCell(int initialValue)
    { storedValue = initialValue; }

  /* Return the stored value */
  int read()
    { return storedValue; }

  /* Change the stored value to x */
  void write( int x )
    { storedValue = x; }

  private:
    int storedValue;
};
```

```
class IntCell
{
  public:
/*1*/  explicit IntCell( int initialValue = 0 )
/*2*/     : storedValue( initialValue ) {}
/*3*/  int read() const
/*4*/    { return storedValue; }
/*5*/  void write( int x )
/*6*/    { storedValue =x; }
       private:
/*7*/    int storedValue;
};
```

Separation of Interface and Implementation

File IntCell.h

```
#ifndef _IntCell_H_
#define _IntCell_H_
class IntCell
{
  public:
     explicit IntCell( int initialValue = 0);
     int read() const;
     void write( int x );
  private:
     int storedValue;
};
#endif
```

Separation of Interface and Implementation

File IntCell.cpp

```
#include "IntCell.h"
/* Construct the IntCell with initialValue */
IntCell::IntCell( int initialValue )
                          : storedValue(initialValue)
{}
/* Return the stored value */
int IntCell::read() const
{ return storedValue;}
/* Store x */
void IntCell::write( int x )
{
   storedValue = x; }
```

Separation of Interface and Implementation

```
#include "IntCell.h"
int main()
{
    IntCell m;
    //Or, IntCell m( 0 ); but not IntCell m();
    m.write( 5 );
    cout << "Cell contents: " << m.read() << endl;
    return 0;
}
```

Separation of Interface and Implementation

- Preprocessor commands
- Scoping operator
- Signature must match exactly
- Objects are declared like primitive types
  - `IntCell obj1; // Zero parameter constructor`
  - `obj2( 12 ); // One parameter constructor`
  - `IntCell obj3 = 37; /* Error : Constructor is explicit */`
  - `IntCell obj4(); // Error : Function declaration`

Vectors and strings

```
#include <iostream.h>
#include "vector.h"
#include "mystring.h"

int main()
{
   vector<string> v( 5 );
   int itemRead = 0;
   string x;
   while( cin >> x)
   {
     if( itemRead == v.size())
        v.resize(v.size() * 2);
     v[ itemRead++] = x;
   }
   for (int i = itemsRead - 1; i >=0; i--)
       cout << v[i] << endl;
   return 0;
}
```

Pointers

```
int main()
{
/*1*/  IntCell *m;
/*2*/  m=new IntCell( 0 );
/*3*/  m->write( 5 );
/*4*/  cout << "Cell contents:"<< m->read() << endl;
/*5*/  delete m;
/*6*/  return 0;
}
```

Function Templates

```
template <class Comparable>
const Comparable & findmax(const vector<Comparable> & a)
{
/*1*/   int maxIndex=0;
/*2*/   for (int i=1;i <a.size();i++)
/*3*/      if (a[maxIndex] < a[i])
/*4*/           maxIndex = i;
/*5*/   return a[maxIndex];
}
```

Function Templates

```
int main()
{
 vector<int>    v1( 37 );
 vector<double> v2( 40 );
 vector<string> v3( 80 );
 vector<IntCell> v4( 75 );
 // Additional code to fill in the vectors
 cout << findMax( v1 ) << endl;
 cout << findMax( v2 ) << endl;
 cout << findMax( v3 ) << endl;
 cout << findMax( v4 ) << endl;
            // Error : operator < undefined
 return 0; }
```

Class Templates

```
// A class for simulating a memory cell.
template <class Object>
class MemoryCell
{
  public:
   explicit MemoryCell (const Object &initialValue=Object())
       : storedValue ( initialValue ) {}
   const Object & read() const
      { return storedValue; }
   void write( const Object & x )
      { storedValue = x; }
   private:
      Object storedValue; };
```

Class Templates

```
int main()
{
  MemoryCell<int>  m1;
  MemoryCell<string> m2( ‘‘hello’’ );
  m1.write( 37 );
  m2.write( m2.read() + ‘‘world’’ );
  cout << m1.read() << endl << m2.read() << endl;
  return 0; }
```

1.3 ADTs and Linear Structures

- Abstract Data Types (ADTs)
- The List ADT
- The Stack ADT
- The Queue ADT