Object-Oriented Programming (OOP)
Lecture No. 35

Member Templates

- A class or class template can have member functions that are themselves templates

...Member Templates

```
template<typename T> class Complex {
  T real, imag;
public:
  // Complex<T>( T r, T im )
  Complex( T r, T im ) :
     real(r), imag(im) {}
  // Complex<T>(const Complex<T>& c)
  Complex(const Complex<T>& c) :
     real( c.real ), imag( c.imag ) {}
  …
};
```

...Member Templates

```
int main() {
  Complex< float > fc( 0, 0 );
  Complex< double > dc = fc; // Error
  return 0;
}
```

Because

```
class Complex<double> {
  double real, imag;
public:
  Complex( double r, double im ) :
     real(r), imag(im) {}
  Complex(const Complex<double>& c) :
     real( c.real ), imag( c.imag ) {}
  …
};
```

…Member Templates

```
template<typename T> class Complex {
  T real, imag;
public:
  Complex( T r, T im ) :
     real(r), imag(im) {}
  template <typename U>
  Complex(const Complex<U>& c) :
     real( c.real ), imag( c.imag ) {}
  …
};
```

...Member Templates

```
int main() {
  Complex< float > fc( 0, 0 );
  Complex< double > dc = fc; // OK
  return 0;
}
```

Because

```
class Complex<double> {
  double real, imag;
public:
  Complex( double r, double im ) :
     real(r), imag(im) {}
  template <typename U>
  Complex(const Complex<U>& c) :
     real( c.real ), imag( c.imag ) {}
  …
};
```

<float> Instantiation

```
class Complex<float> {
  float real, imag;
public:
  Complex( float r, float im ) :
     real(r), imag(im) {}
   // No Copy Constructor
  …
};
```

Class Template Specialization

- Like function templates, a class template may not handle all the types successfully
- Explicit specializations are provided to handle such types

...Class Template Specialization

```
int main() {
  Vector< int > iv1( 2 );
  iv1[0] = 15;
  iv1[1] = 27;
  Vector< int > iv2( iv1 );
  Vector< int > iv3( 2 );
  iv3 = iv1;
  return 0;
}
```

...Class Template Specialization

```
int main() {
  Vector< char* > sv1( 2 );
  sv1[0] = "Aamir";
  sv1[1] = "Nasir";
  Vector< char* > sv2( sv1 );
  Vector< char* > sv3( 2 );
  sv3 = sv1;
  return 0;
}
```

...Class Template Specialization

```
template<>
class Vector< char* > {
private:
  int size;
  char** ptr;
public:
  // Vector< char* >( int = 10 );
  Vector( int = 10 );
  Vector( const Vector< char* >& );
  virtual ~Vector();
```

...Class Template Specialization

```
  int getSize() const;
  const Vector< char* >& operator =(
          const Vector< char* >& );
  const char*& operator []( int );
  void insert( char*, int );
};
```

...Class Template Specialization

```
template<>
Vector<char*>::Vector(int s) {
  size = s;
  if ( size != 0 ) {
     ptr = new char*[size];
     for (int i = 0; i < size; i++)
         ptr[i] = 0;
  }
  else
     ptr = 0;
}
```

...Class Template Specialization

```
template<>
Vector< char* >::Vector(
         const Vector<char*>& copy ) {

  size = copy.getSize();
  if ( size == 0 ) {
     ptr = 0;
     return;
  }
```

...Class Template Specialization

```
  ptr = new char*[size];
  for (int i = 0; i < size; i++)
     if ( copy.ptr[i] != 0 ) {
          ptr[i] = new char[ strlen(
                    copy.ptr[i] ) + 1 ];
          strcpy(ptr[i], copy.ptr[i]);
     }
     else
          ptr[i] = 0;
}
```

...Class Template Specialization

```
template<>
Vector<char*>::~Vector() {
  for (int i = 0; i < size; i++)
     delete [] ptr[i];

  delete [] ptr;
}
```

...Class Template Specialization

```
template<>
int Vector<char*>::getSize() const {
  return size;
}
```

...Class Template Specialization

```
template<>
const Vector<char*>& Vector<char*>::
operator=(const Vector<char*>& right)
{
  if ( this == &right )
     return *this;
  for (int i = 0; i < size; i++)
     delete [] ptr[i];
  delete [] ptr;
```

...Class Template Specialization

```
size = right.size;
if ( size == 0 ) {
  ptr = 0;
  return *this;
}
ptr = new char*[size];
```

...Class Template Specialization

```
for (int i = 0; i < size; i++)
  if ( right.ptr[i] != 0 ) {
     ptr[i] = new char[strlen(
                right.ptr[i] ) + 1];
     strcpy( ptr[i], right.ptr[i] );
  }
  else
     ptr[i] = 0;
}
```

...Class Template Specialization

```
template<>
const char*& Vector<char*>::
         operator []( int index ) {
  if ( index < 0 || index >= size ) {
     cout << "Error: index out of
                              range\n";
     exit( 1 );
  }
  return ptr[index];
}
```

...Class Template Specialization

```
template<>
void Vector< char* >::insert(
                  char* str, int i ) {
  delete [] ptr[i];
  if ( str != 0 ) {
     ptr[i] = new char[strlen(str)+1];
     strcpy( ptr[i], str );
  }
  else
     ptr[i] = 0;
}
```

...Class Template Specialization

```
int main() {
  Vector< char* > sv1( 2 );
  sv1[0] = "Aamir"; // Error
  sv1.insert( "Aamir", 0 );
  sv1.insert( "Nasir", 1 );
  Vector< char* > sv2( sv1 );
  Vector< char* > sv3( 2 );
  sv3 = sv1;
  return 0;
}
```