Object-Oriented Programming (OOP)
Lecture No. 36

Recap – Member Templates

- A class template may have member templates
- They can be parameterized independent of the class template

Recap –Template Specializations

- A class template may not handle all the types successfully
- Explicit specializations are required to deal such types

Member Templates Revisited

- An ordinary class can also have member templates

```
class ComplexSet {
…
   template< class T >
   insert( Complex< T > c )
   { // Add "c" to the set }
};
```

… Member Templates Revisited

```
int main() {
  Complex< int > ic( 10, 5 );
  Complex< float > fc( 10.5, 5.7 );
  Complex< double > dc( 9.567898, 5 );
  ComplexSet cs;
  cs.insert( ic );
  cs.insert( fc );
  cs.insert( dc );
  return 0;
}
```

Partial Specialization

- A partial specialization of a template provides more information about the type of template arguments than that of template
- The number of template arguments remains the same

Example – Partial Specialization

```
template< class T >
class Vector { };

template< class T >
class Vector< T* > { };
```

Example – Partial Specialization

```
template< class T, class U, class V >
class A {};

template< class T, class V >
class A< T, T*, V > {};

template< class T, class U, int I >
class A< T, U, I > {};

template< class T >
class A< int, T*, 5 > {};
```

Example – Complete Specialization

```
template< class T >
class Vector { };

template< >
class Vector< char* > { };
```

Example – Complete Specialization

```
template< class T, class U, class V >
class A {};

template< >
class A< int, char*, double > {};
```

Function Templates

- A function template may also have partial specializations

Example – Partial Specialization

```
template< class T, class U, class V >
void func( T, U, V );

template< class T, class V >
void func( T, T*, V );

template< class T, class U, int I >
void func( T, U );

template< class T >
void func( int, T, 7 );
```

Example

- Consider the following template

```
template< typename T >
bool isEqual( T x, T y ) {
   return ( x == y );
}
```

Complete Specialization

- We have already used this complete specialization

```
template< >
bool isEqual< const char* >(
   const char* x, const char* y ) {
   return ( strcmp( x, y ) == 0 );
}
```

Partial Specialization

- Following partial specialization deals with pointers to objects

```
template< typename T >
bool isEqual( T* x, T* y ) {
   return ( *x == *y );
}
```

Using Different Specializations

```
int main() {
  int i, j;
  char* a, b;
  Shape *s1 = new Line();
  Shape *s2 = new Circle();
  isEqual( i, j );   // Template
  isEqual( a, b );   // Complete Sp.
  isEqual( s1, s2 ); // Partial Sp.
  return 0;
}
```

Non-type Parameters

- Template parameters may include non-type parameters
- The non-type parameters may have default values
- They are treated as constants
- Common use is static memory allocation

Example – Non-type Parameters

```
template< class T >
class Array {
private:
  T* ptr;
public:
  Array( int size );
  ~Array();
  …
};
```

Example – Non-type Parameters

```
template< class T >
Array<T>::Array() {
  if (size > 0)
     ptr = new T[size];
  else
     ptr = NULL;
}
```

Example – Non-type Parameters

```
int main() {
  Array< char > cArray( 10 );
  Array< int > iArray( 15 );
  Array< double > dArray( 20 );

  return 0;
}
```

Example – Non-type Parameters

```
template< class T, int SIZE >
class Array {
private:
  T ptr[SIZE];
public:
  Array();
  …
};
```

…Example – Non-type Parameters

```
int main() {
  Array< char, 10 > cArray;
  Array< int, 15 > iArray;
  Array< double, 20 > dArray;

  return 0;
}
```

Default Non-type Parameters

```
template< class T, int SIZE = 10 >
class Array {
private:
  T ptr[SIZE];
public:
  void doSomething();
  …
}
```

… Default Non-type Parameters

```
int main() {
  Array< char, 15 > cArray;
  return 0;
}
// OR

int main() {
  Array< char > cArray;
  return 0;
}
```

Default Type Parameters

- A type parameter can specify a default type

```
template< class T = int >
class Vector {
   …
}

Vector< > v;  // Vector< int > v;
```