Object-Oriented Programming (OOP)
Lecture No. 32

Motivation

- Following function prints an array of integer elements:

```
void printArray(int* array, int size)
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

...Motivation

- What if we want to print an array of characters?

```
void printArray(char* array,
                              int size)
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

...Motivation

- What if we want to print an array of doubles?

```
void printArray(double* array,
                              int size)
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

...Motivation

- Now if we want to change the way function prints the array. e.g. from

```
1, 2, 3, 4, 5
```

to

1 - 2 - 3 - 4 - 5

...Motivation

- Now consider the `Array` class that wraps an array of integers

```
class Array {
   int* pArray;
   int size;
public:
   …
};
```

...Motivation

- What if we want to use an `Array` class that wraps arrays of double?

```
class Array {
   double* pArray;
   int size;
public:
   …
};
```

...Motivation

- What if we want to use an `Array` class that wraps arrays of boolean variables?

```
class Array {
   bool* pArray;
   int size;
public:
   …
};
```

...Motivation

- Now if we want to add a function sum to `Array` class, we have to change all the three classes

Generic Programming

- Generic programming refers to programs containing generic abstractions
- A generic program abstraction (function, class) can be parameterized with a type
- Such abstractions can work with many different types of data

Advantages

- Reusability
- Writability
- Maintainability

Templates

- In C++ generic programming is done using templates
- Two kinds
  - Function Templates
  - Class Templates
- Compiler generates different type-specific copies from a single template

Function Templates

- A function template can be parameterized to operate on different types of data

Declaration

```
template< class T >
void funName( T x );
// OR

template< typename T >
void funName( T x );
// OR

template< class T, class U, … >
void funName( T x, U y, … );
```

Example – Function Templates

- Following function template prints an array having almost any type of elements:

```
template< typename T >
void printArray( T* array, int size )
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

...Example – Function Templates

```
int main() {
  int iArray[5] = { 1, 2, 3, 4, 5 };
  void printArray( iArray, 5 );
     // Instantiated for int[]

  char cArray[3] = { 'a', 'b', 'c' };
  void printArray( cArray, 3 );
     // Instantiated for char[]
  return 0;
}
```

Explicit Type Parameterization

- A function template may not have any parameter

```
template <typename T>
T getInput() {
   T x;
   cin >> x;
   return x;
}
```

...Explicit Type Parameterization

```
int main() {
  int x;
  x = getInput();      // Error!

  double y;
  y = getInput();      // Error!
}
```

...Explicit Type Parameterization

```
int main() {
  int x;
  x = getInput< int >();

  double y;
  y = getInput< double >();
}
```

User-defined Specializations

- A template may not handle all the types successfully
- Explicit specializations need to be provided for specific type(s)

Example – User Specializations

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

... Example – User Specializations

```
int main {
  isEqual( 5, 6 );     // OK
  isEqual( 7.5, 7.5 );     // OK
  isEqual( "abc", "xyz" );
     // Logical Error!
  return 0;
}
```

... Example – User Specializations

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

... Example – User Specializations

```
int main {
  isEqual( 5, 6 );
     // Target: general template
  isEqual( 7.5, 7.5 );
     // Target: general template

  isEqual( "abc", "xyz" );
     // Target: user specialization
  return 0;
}
```