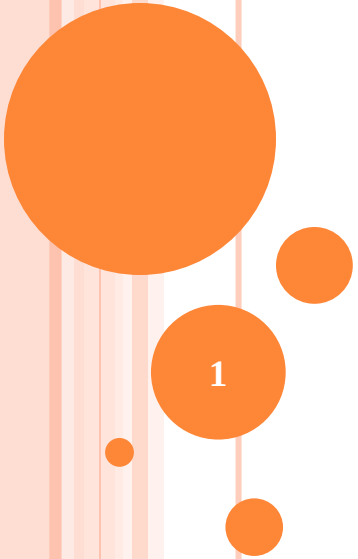
Templates



Templates

- Type-independent patterns that can work with multiple data types.
  - Generic programming
  - Code reusable
- Function Templates
  - These define logic behind the algorithms that work for multiple data types.
- Class Templates
  - These define generic class patterns into which specific data types can be plugged in to produce new classes.

Function and function templates

- C++ routines work on specific types. We often need to write different routines to perform the same operation on different data types.

```
int maximum(int a, int b, int c)
 {
      int max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

Function and function templates

```
float maximum(float a, float b, float c)
 {
      float max = a;
      if (b > max) max = b;
      if (c > max) max = c;
      return max;
}
```

FUNCTION AND FUNCTION TEMPLATES

```
double maximum(double a, double b, double c)
 {
     double max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

The logic is exactly the same, but the data type is different.

Function templates allow the logic to be written once and used for all data types – generic function.

Function Templates

- Generic function to find a maximum value (see maximum example).

```
Template <class T>
T maximum(T a, T b, T c)
 {
     T max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

- Template function itself is incomplete because the compiler will need to know the actual type to generate code. So template program are often placed in .h or .hpp files to be included in program that uses the function.
- C++ compiler will then generate the real function based on the use of the function template.

Function Templates Usage

- After a function template is included (or defined), the function can be used by passing parameters of real types.

```
Template <class T>
T maximum(T a, T b, T c)
…
int i1, i2, i3;
…
Int m = maximum(i1, i2, i3);
```

- maximum(i1, i2, i3) will invoke the template function with T==int. The function returns a value of int type.

Function Templates Usage

- Each call to `maximum()` on a different data type forces the compiler to generate a different function using the template. See the maximum example.
  - One copy of code for many types.

```
int i1, i2, i3;
// invoke int version of maximum
cout << "The maximum integer value is: "
   << maximum( i1, i2, i3 );
// demonstrate maximum with double values
double d1, d2, d3;
// invoke double version of maximum
cout << "The maximum double value is: "
   << maximum( d1, d2, d3 );
```

Another example

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

Usage

```
template< class T >
void printArray( const T *array, const int count );

char  cc[100];
int      ii[100];
double dd[100];
……
printArray(cc, 100);
printArray(ii, 100);
printArray(dd, 100);
```

Usage

```
template< class T >
void printArray( const T *array, const int count );

char  cc[100];
int      ii[100];
double dd[100];
myclass xx[100];  <- user defined type can also be used.
……
printArray(cc, 100);
printArray(ii, 100);
printArray(dd, 100);
printArray(xx, 100);
```

Use of template function

- Can any user defined type be used with a template function?
  - Not always, only the ones that support all operations used in the function.
  - E.g. if myclass does not have overloaded << operator, the printarray template function will not work.

Class template

- So far the classes that we define use fix data types.
- Sometime is useful to allow storage in a class for different data types.
- See simplelist1 (a list of int type elements) example
  - What if we want to make a simple list of double type?
    - Copy paste the whole file and replace int with double
    - Make use of typedef in C++, See simplelist2.
      - Still need to change one line of code for a new type.

Class template

- Function templates allow writing generic functions that work on many types.
- Same idea applies to defining generic classes that work with many types -- extract the type to be a template to make a generic classes.
- See simplelist3

Class template

- To make a class into a template, prefix the class definition with the syntax:

  template< class T >

  - Here T is just a type parameter. Like a function parameter, it is a place holder.
  - When the class is instantiated, T is replaced by a real type.
- To access a member function, use the following syntax:
  - className< T >:: memberName.
  - SimpleList < T > :: SimpleList()
- Using the class template:
  - ClassName<real type> variable;
  - SimpleList < int > list1;

Another Class Template Example

- MemoryCell template can be used for any type Object.

- Assumptions
  - Object has a zero parameter constructor
  - Object has a copy constructor
  - Copy-assignment operator

- Convention
  - Class templates declaration and implementation usually combined in a single file.
  - It is not easy to separate them in independent files due to complex c++ syntax.
  - This is different from the convention of separating class interface and implementation in different files.

```
1   /**
2    * A class for simulating a memory cell.
3    */
4   template <typename Object>
5   class MemoryCell
6   {
7     public:
8       explicit MemoryCell( const Object & initialValue = Object( ) )
9         : storedValue( initialValue ) { }
10      const Object & read( ) const
11        { return storedValue; }
12      void write( const Object & x )
13        { storedValue = x; }
14    private:
15      Object storedValue;
16  };
```

Class Template Usage Example

- MemoryCell can be used to store both primitive and class types.
- Remember
  - `MemoryCell` is not a class.
  - It's a class template.
  - `MemoryCell<int>`, `MemoryCell<string>` etc are classes.

```
1   int main( )
2   {
3       MemoryCell<int>    m1;
4       MemoryCell<string> m2;( "hello" );
5
6       m1.write( 37 );
7       m2.write( m2.read( ) + "world" );
8       cout << m1.read( ) << endl << m2.read( ) << endl;
9
10      return 0;
11  }
```