COP 3331
OBJECT ORIENTED DESIGN
SPRING 2017

WEEK 12: TEMPLATES AND THE STANDARD TEMPLATE LIBRARY (STL)

SCHINNEL SMALL

UNIVERSITY OF SOUTH FLORIDA

FUNCTION TEMPLATES

FUNCTION TEMPLATES

- A function template is a pattern for a function that can work with many data types
- When written, parameters are substituted for the data types
- When called, the compiler generates code to handle specific data types in the call

WHY USE FUNCTION TEMPLATES?

- Recall: overloaded functions are convenient for performing the same operation on different data types
- However, each overloaded function must be defined individually
- A function template allows you to write one function definition that works for all of the different types

FUNCTION TEMPLATES

- Example: Instead of this:

```
int square(int number)
{
    return number * number;
}
```

```
double square(double number)
{
    return number * number;
}
```

- You can write this:

```
template <class T>
T square(T number)
{
        return number * number;
}
```

FUNCTION TEMPLATES

- The template is not an actual function, but a “mold” that the compiler uses to generate one or more functions
  - A function template uses no memory
- A *type parameter* is used for the parameters, return type and local variables instead of actual data types
- The generated code, or *template function* is created when the function is called
  - The compiler will examine the data type of the actual parameters to determine what data types to use

FUNCTION TEMPLATES

- Let's examine the function template:

```
template <class T>

T square(T number)
{
        return number * number;
}
```

- The line `template <class T>` is known as a *template prefix*
  - T represents the generic data type
    - More data types can be used, just separate by commas
    - e.g. `template<class T1, class T2>`

  - Note that the function definition is written as normal, expect all of the data types have been replaced by "T"

FUNCTION TEMPLATES - EXAMPLES

- See Examples 1-3 in the Week 12 Examples.doc on canvas

- Note: Function Templates should appear before all function calls
  - i.e. they should be placed at the top of a program/ header file

FUNCTION TEMPLATES

- Any/All data types specified in the template prefix must be used in template definition
- If a user defined class object is passed to a function, the class must contain code for an overloaded operator
  - e.g.: If we were to pass a fraction object into the square function, we would have to overload the * operator
  - If not overloaded, the compiler will generate a function with an error

FUNCTION TEMPLATES

- Function Templates can be overloaded!

- Options for overloading a function template:
  - Create two or more templates with different parameter lists
    - See Example 4 in Week 12 Examples.doc

  - Create a function template, and use alongside a regular function
    - Can coexist as long as they have different parameter lists

FUNCTION TEMPLATES

- Tip: the best way to write a function template is to:
  - Write a regular function
  - Test and debug the function thoroughly
  - Convert it to a template function:
    - Add template prefix
    - Replace data types with generic data types

FUNCTION TEMPLATES

- Starting in C++ 11, the key word `typename` may be used instead of `class` in the template prefix.

- Thus,

```
template<class T>
```

  may be written as

```
template<typename T>
```

- Test it out by modifying any of the previous examples!

CLASS TEMPLATES

CLASS TEMPLATES

- Classes can also be represented by templates
- Unlike functions, a class template is instantiated by supplying the type name (`int`, `float`, `string`, etc.) at object definition
- When a class object is created, type information is supplied to define the type of data members of the class

CLASS TEMPLATES

- Example: consider the following classes
  - Class used to join two integers by adding them:

```
class Joiner
{ public:
     int combine(int x, int y)
     {return x + y;}
};
```

  - Class used to join two strings by concatenating them:

```
class Joiner
{ public:
     string combine(string x, string y)
     {return x + y;}
};
```

CLASS TEMPLATES

- A single class template can capture the logic of both classes
- Like a function template, it is written with a template prefix that specifies the data type parameters

```
template <class T>
class Joiner
{
public:
    T combine(T x, T y)
        {return x + y;}
};
```

CLASS TEMPLATES

- To create an object of a class defined by a template, specify the actual parameters for the formal data types
  - Use as ordinary objects once defined

```
Joiner<double> jd;
Joiner<string> sd;
cout << jd.combine(3.0, 5.0);
cout << sd.combine("Hi ", "Ho");
```

- Prints `8.0` and `Hi Ho`

CLASS TEMPLATES

- If a member function is defined outside of the class, then the definition requires the template header to be prefixed to it

- the template name and type parameter list is to be used to refer to the name of the class

```
template<class T>
T Joiner<T>::combine(T x, T y)
      {return x + y;}
```

CLASS TEMPLATES

- For another example, see Example 5
  - 5a: RectTemplate (inspired from Rectangle class)
    - Referenced (and modified from week 5, slide 4 code)
  - 5b: Driver Program

CLASS TEMPLATES

- Class templates can inherit from other class templates:

```
template <class T>
class Rectangle
  { ... };

template <class T>
class Square : public Rectangle<T>
  { ... };
```

- Must use type parameter `T` everywhere base class name is used in derived class

INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY

THE STANDARD TEMPLATE LIBRARY

- The Standard Template Library (STL) is a library containing templates for frequently used data structures and algorithms
- The STL is not supported by many older compilers
- Programs can be developed faster and are more portable if they use templates from the STL

THE STANDARD TEMPLATE LIBRARY

- There are two important types of data structures in the STL: containers and iterators
- Containers are classes that store data and impose some organization on it
- Iterators are like pointers; they are mechanisms for accessing elements in a container

THE STANDARD TEMPLATE LIBRARY

- Two types of container classes in STL: sequence and associative containers
- sequence containers organize and access data sequentially
  - These include `array`, `vector`, `deque`, and `list`
- associative containers: use keys to allow data elements to be quickly accessed
  - These include `set`, `multiset`, `map`, and `multimap`

THE STANDARD TEMPLATE LIBRARY

- Sequence Containers
  - array: a fixed size container that is similar to an array
    - Essentially the object oriented version of an array
    - Introduced in C++ 11
    - Include the array header file
    - Syntax:
      ```
      array <int, 5> numbers;
      array <int, 5> numbers{1, 2, 3, 4, 5};
      ```
    - [ ] operator still does not do bounds checking (but you can overload the operator to fix that)

THE STANDARD TEMPLATE LIBRARY

- Sequence Containers
  - vector: a container that works like an expandable array
    - Values can be quickly added to the end of a vector
      - Other insertion points not as efficient
    - Include the vector header file
    - Syntax:
      ```
      vector <int> numbers;
      vector <int> numbers(10);
      ```
  - deque: a double ended queue
    - Like a vector, but values may be added from front or back
      - Other insertion points not as efficient
    - Include the deque header file
    - Syntax:
      ```
      deque <int> numbers;
      deque <int> numbers(4, 10);
      ```

THE STANDARD TEMPLATE LIBRARY

- Sequence Containers
  - list: a doubly linked list of data elements
    - Values can be quickly added at any point in the list
      - More on this later
    - Include the list header file
  - forward_list: a singly linked list of data elements
    - Values can be quickly added at any point in the list
      - More on this later… possibly
    - Include the forward_list header file

THE STANDARD TEMPLATE LIBRARY

- Associative Containers (some examples)
  - set: Stores a set of unique values that are sorted
    - No duplicates allowed
  - multiset: Stores a set of unique values that are sorted
    - duplicates allowed
  - map: Maps a set of keys to elements
    - Only one key per element allowed
  - multimap: Maps a set of keys to elements
    - Many keys per element allowed

THE STANDARD TEMPLATE LIBRARY

- Iterators are generalization of pointers
  - They are used to access information in containers

- Many types:
  - forward (uses `++`)
  - bidirectional (uses `++` and `--`)
  - random-access
  - input (can be used with `cin` and `istream` objects)
  - output (can be used with `cout` and `ostream` objects)

THE STANDARD TEMPLATE LIBRARY

- The type of container you have determines the iterator that you use
  - Examples: array, vector and deque use random access
  - list, set, multiset, map and mutimap use bidirectional

- To define an iterator, you specify the container

```
array<string, 3>::iterator it;
vector<int>::iterator iter;
```

THE STANDARD TEMPLATE LIBRARY

- Each container class defines functions that return iterators:
- `begin()`: points to the item at the start of the container
- `end()`: points to the location just past the end of the container
- See Example 6 for iterator example

THE STANDARD TEMPLATE LIBRARY

- Iterators support pointer-like operations. If `iter` is an iterator, then
  - `*iter` is the item it points to: this dereferences the iterator
  - `iter++` advances to the next item in the container
  - `iter--` backs up in the container

- The `end()` iterator points to past the end: it should *never* be dereferenced

THE STANDARD TEMPLATE LIBRARY

- Every container has an appropriate header file and list of member functions
- Example: vector

Function	Description
front(), back()	Returns a reference to the first, last element in a vector
size()	Returns the number of elements in a vector
capacity()	Returns the number of elements that a vector can hold
clear()	Removes all elements from a vector
push_back(value)	Adds element containing value as the last element in the vector
pop_back()	Removes the last element from the vector
insert(iter, value)	Inserts new element containing value just before element pointed at by iter

THE STANDARD TEMPLATE LIBRARY

- STL contains algorithms implemented as function templates to perform operations on containers.
  - Requires `algorithm` header file

- `algorithm` includes

`binary_search`	`count`
`for_each`	`find`
`find_if`	`max_element`
`min_element`	`random_shuffle`
`sort`	and others

THE STANDARD TEMPLATE LIBRARY

- Many STL algorithms manipulate portions of STL containers specified by a begin and end iterator

- Illustrated in Example 7