CIT237
Chapter 16: Exceptions and Templates (Part 2: Templates)

November 13, 2019

Reminders

- Quiz 6 will be held at the start of class on Wednesday, November 20.
- The material covered on Quiz 6 will be:
  - Lectures of October 28 through November 13.
  - Chapters 15, 16 and 17.
- Project 3:
  - We discussed Project 3 in class last week.
  - The due date is December 2.

Function Templates

- Function template: a pattern for a function that can work with many data types
- When written, parameters are present in the template for the data types
- When called, compiler generates code for specific data types in function call

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

What gets generated when `times10` is called with an `int`:	What gets generated when `times10` is called with a `double`:

`int times10(int num)`
`{`
`  return 10 * num;`
`}`
	`double times10(double num)`
	`{`
	`  return 10 * num;`
	`}`

Calling a Function Template

- Call a template function in the usual manner:

```
int ival = 3;
double dval = 2.55;
cout << times10(ival); // displays 30
cout << times10(dval); // displays 25.5
```

Multiple Type Parameters

- Can define a template to use multiple data type parameters:

```
template<class T1, class T2>
```

- Example:

```
template<class T1, class T2>      // T1 and T2 will be
double mpg(T1 miles, T2 gallons) // replaced in the
{                                 // called function
return miles / gallons          // with the data
}                                 // types of the
                                  // arguments
```

Overloading a Function Template

- Function templates can be overloaded.
- Each template must have a unique parameter list

```
template <class T>
T sumAll(T num) ...
template <class T1, class T2>
T1 sumall(T1 num1, T2 num2) ...
```

Function Template Notes

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition
- All data type parameters specified in template prefix must be used in template definition

Where to Start When Defining Function Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types.
- However, developing a complex template "from scratch" may be difficult.
- Instead, develop a "normal" function using the usual data types first; then after you get the "normal" function working, convert it to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template

Class Templates

- Suppose you needed to create several C++ classes, and they were all indentical except that they each operate on a *different* numeric data type.
- For example, in Chapter 14 of our textbook, there is a class called **IntArray**.

  (See pages 858 – 863)
- It could be that we would also need classes called **LongArray**, **FloatArray**, and **DoubleArray**.

SimpleVector.h

- The textbook includes a *class template* called **SimpleVector**, and declares it in a file called **SimpleVector.h**.
- This template specifies the code for creating an array of any numeric primitive type.
- Program 16-11 (pages 1018-1019) is a sample program that utilizes the SimpleVector class.

What about Non-numeric Data Types?

- Can we make a class template for other data types?
  - Yes, but not all operations have meaning for all data types.
  - For example, how do you increment a **string** object?