COMSC-200 Lecture Topic 14
Templates

□ Reference
Deitel Ch.14

□ Templates
1. function templates, and
2. class templates

added to Java in 2004 -- "generics"

□ Function Templates
alternative to overloading functions
  when overloading for different data types
    like `int avg(int, int);`
    and `double avg(double, double);`
C++ compiler overloads functions automatically
  based on *model* supplied by programmer
    *or a **template***
compiler generates copies of the function
  based on function calls found in program
the data-type of overloaded functions becomes
  a "parameter" in function definition
declaration syntax: (prototype)

```
template <class T>
```
or
```
template <typename A>
```
or
```
template <class A, class B>
```
substitute T (or A and B) for variable data-type where it appears in function proto and definition

□ Example: A `sortArray` Function
using overloaded functions:

```
void sortArray(int, int*);
void sortArray(int, Time*);
```
programmer writes 2 similar functions

using function templates *(a 2-line prototype)*:

```
template <class T>
void sortArray(int, T*);
```
programmer writes 1 template,
compiler generates copies as needed

overloading the `operator<` operator

□ Function Template Calls

```
int a[100] = {...};
sortArray(100, a);
```

Sometimes compiler needs "help"

```
template <class T, class U>
U convert(T& a);
...
int x =...;
...convert<int, double>(x)...
```
`#include<typeinfo>` *(no using statement)*

```
if(typeid(T)==typeid(int))...
```

□ Class Templates
extends template idea to class definitions
generic classes and parameterized data-types

class template "specializations":
  programmer *specifies* copies to make
  (instead of letting the compiler figure
  it out, as it does for function templates)
declaration syntax (class definition):

```
template <class T>
```
substitute T for variable data-type where it
  appears inside class definition
declaration syntax (member function definitions):

```
template <class T>
void Array<T>::output()
{
  ...
}
```

□ Example: An `Array` Class
`Array(int);` constructor
`int size() const;` member function
`T& operator[](int);` member function
`const T& operator[](int) const;` member function
  throwing exceptions
`ostream& operator<<(ostream&, const Array<T>&);` friend function
  must be without prototype or not templated
declarations:
  an `Array` of `int`s: `Array<int> a(10);`
  an `Array` of `Time`s: `Array<Time> t(10);`
multiple reference to same data-type do *not*
  result in redundant copies of
  the same template-generated class

□ Variations
non-type template parameters

```
template <class T, int size>
```
defaulted parameters

```
template <class T = int>
template <class T, int size=10>
```
  one or the other, but not both!

□ Template H Files
no prototypes, no CPP, just templates

□ The C++ Standard Template Library
consists of: containers, iterators, and algorithms
the `vector` template, and other containers
  for programmers to apply
the `string` class
  built-in application of `basic_string` template
  as applied to `char`s

Pseudocode for lab 14:

```
void Floor::addNewRider(const Rider& rider)
  // if added rider's destination is greater than the floor's location
    // add rider to the upRiders vector
  // else
    // add rider to the downRiders vector

vector<Rider> Floor::removeUpRiders(int max) // max = #of unused spaces on elevator
  // create an empty vector for riders to be removed
  // if there are any up riders...
    // create an empty vector for riders to remain on the floor
    // traverse the upRiders vector
      // if there are still spaces left on the elevator...
        // add an upRider to the vector of riders to be removed
      // else
        // add an upRider to the vector of riders to remain on the floor
    // replace the upRiders vector with the vector of remaining riders
  // return the vector of removed riders

vector<Rider> Floor::removeDownRiders(int max)
  // like removeUpRiders, but using the downRiders vector

bool Floor::isPreferredDirectionUp() const
  // if there are no downRiders, return true
  // if there are no upRiders, return false
  // if the ID of the first upRider (upRider[0]) is less than that of the first downRider...
    // return true
  // return false
```