Implementing generic types

With C++ templates

Starting Savitch Chapter 17

C++ templates

- Like "blueprints" for the compiler to use in creating class and function definitions
- Always involve one or more *parameterized types*
  - e.g., function template to compare object sizes:

```
template <typename T1, typename T2>
int sizeComp(T1 const &o1, T2 const &o2)
{ return (sizeof o1 - sizeof o2); }
```

  - e.g., class template for a list that holds any type:

```
template <typename DataType>
class List { /* here refer to DataType objects */ };
```

- Can use either keyword `typename` or `class` in a "template prefix" – e.g., `template <class T>`

Function templates

- An alternative to function overloading
  - But code for concrete types created only as needed
    - And the programmer does not have to write it!
  - Compiler deduces types if user doesn't specify:

```
int x = sizeComp('a', 7);
    // compiler uses template to create sizeComp(char, int)
```

  - To specify: `x = sizeComp<int, int>('a', 7.5);`

```
    // compiler uses template to create sizeComp(int, int)
```

- Better choice than macros too
  - Strictly type-checked, and no nasty side effects
- See `~mikec/cs32/demos/templates/greater.cpp`

More function template issues

- Template definition must be in header file – so compiler can know how to define the functions
  - i.e., cannot be defined in a separate `.cpp` file
- Sometimes *specialized* for particular types
  - Tells compiler to use specialized version instead of creating a new definition – e.g. greater for char*:

```
template <>   // <> does not show a type parameter
char * &greater<char *>(char *s, char *t)
{  /* would use strcmp to compare s and t, instead of operator< */  }
```

    - Empty parameter types – exact types everywhere else
  - No type conversions though (must be exact match), so usually better to just overload instead of specialize

Defining class templates

- Idea: “generalize” data that can be managed by a class

```
template<typename T>
class Pair {
public:
    Pair();
    Pair(T firstVal, T secondVal);
    void setFirst(T newVal);
    void setSecond(T newVal);
    T getFirst() const;
    T getSecond() const;
private:
    T first; T second;
};
```

Class template member functions

- All methods need template prefix – e.g., constructor:

```
template<class T>
Pair<T>::Pair(T val1, T val2)
     : first(val1), second(val2) { }
```

- Similarly setter and getter functions:

```
template<class T>
void Pair<T>::setFirst(T newVal)
{    first = newVal;   }
template<class T>
T Pair<T>::getFirst() const { return first; }
```

- See `~mikec/cs32/demos/templates/complex` example

Note: each function definition is itself a template

More class template notes

- Mostly design just like any class
  - Can have friends – usually do
  - Can be a base class or a derived class
- Careful though: `MyTemplate<T1>` ≠ `MyTemplate<T2>`
  - That is, there is no inheritance or any other kind of formal relationship between the two classes
    - e.g., cannot cast an object of one to an object of the other
  - Why?
    - Compiler defines completely different classes!

Class templates in OO design

- An alternative to using an inheritance hierarchy
  - More flexible, as template classes stand alone
  - More efficient than using virtual functions
- Both are ways to have objects with independent behaviors, but all sharing a common interface
- The STL is mostly template classes and functions
  - Ditto the Java Collections Framework by the way
- Even a `string` is actually a specialization of a template, defined as follows in `namespace std`:
  - `typedef basic_string<char> string;`
  - Also: `typedef basic_string<wchar_t> wstring;`