Moving templates in front of remaining slides on inheritance

For the sake of assignment 5 –
covering chapter 12 before
finishing chapter 11

C++ templates

- Like “blueprints” for the compiler to use in creating class and function definitions
  - Repeat – *the compiler writes the code for you*
- 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{…};
```

Function templates

- Alternative to function overloading
  - But code for concrete types created only as needed
    - And the programmer does not have to write it!
  - Compiler deduces parameter types if not specified

    ```
    int x = sizeComp(‘a’, 7);  // now the compiler will use
      the template to create sizeComp(char, int)
    x = sizeComp<int, int>(‘a’, 7.5); // specify(int,int)
    ```

  - And no casts or run-time conversions required
- Better choice than macros
  - Strictly type-checked, and no nasty side effects
- See `greater` example in …`/demo09/function_template`

More function templates

- Template definition must be in header file
  - Compiler must know how to define function
    - So template cannot be in separate `.cpp` file
- Can specialize for particular types
  - Tells the compiler to use specialized version instead of creating a new definition
    - In this case, okay to declare in `.h` and implement in `.cpp`
  - e.g., `template <> int const &greater<int>(…);`
    - No template parameters – exact types everywhere else
  - No type conversions are made – must be exact match
    - So it is usually better to just overload instead of specialize

Class templates

- Alternative to inheritance – and more flexible
  - No cosmic superclass in C++ (like `java.lang.Object`)
- Objects are always a particular type
  - e.g., `List<int>` is unrelated to `List<char>`
    - i.e., not a hierarchy like inheritance provides
  - User must specify the type – not deduced by compiler
    - Unless default type in definition: `<typename T = int>`
- Can grant friendship to functions or classes
- Can be specialized, fully or partially
- Can be derived classes, and can be base classes

Implementing class templates

- All but specializations must be in header file
  - Compiler can’t write the class without the blueprint
    - Note: the separate compilation model using the `export` keyword (Nagler pp. 392-6) does not work with `g++` yet
  - Simplest way is implicit inline inside class definition
- If implement outside class (but still in header file) – must parameterize class name wherever it is used
  - See `Complex` example in …`/demo09/class_template`
- Specialized functions may be in a `.cpp` file
  - But declare in header to let compiler know not to create

Back to inheritance topics

Inheriting functions

- Function hiding – if function defined in derived class with same *name* as function(s) in base class
  - Hides *all* non-virtual base class functions with same name
  - But can do `using Base::`*`name`* to unhide
- Manager functions are *never* inherited
  - But still often must access – e.g., always need base’s ctor
    - Can use `Base(`*arg list*`)` in derived class’s initializer list
    - In `operator=` and others – use scope resolution `Base::operator=(…)`
- Upcasts – base pointer/reference for derived instance OK
  - Never upcast with arrays – different sizes ruin pointer arithmetic
  - Called “object slicing” if derived instance copied to base instance

`virtual` functions

- Polymorphism is not automatic in C++
  - Function must be declared `virtual` in base class
    - Otherwise derived class will hide it, not override it
    - Virtual functions stay virtual for all descendants
  - See …`/demo08/loans/` example
- Note: dtors *must* be virtual to allow derivation
- Abstract base classes – any class with a "pure virtual" function – cannot be instantiated *per se*
  - e.g., `virtual void func() = 0;` // pure virtual
    - Derived classes must implement or they are abstract too
  - All instances are actually derived class instances