Lecture Topics

- extensibility: philosophy vs. reality
- templates
  - motivation & basics
  - using function templates
  - template checking and binding [remainder Thursday]
  - avoiding code explosion
  - miscellany
  - the Standard Template Library
  - specializing containers

Administrivia

- HW #2
  - out today
  - get code from web page
  - nominally due in two weeks…

Philosophy vs. Reality

- "…no type…can have operations added after its definition is complete." (Str, p. 81)
- functions valid on a class instance
  - defined at compile time (as with Simula)
  - in contrast with Smalltalk
    - which allows use of dynamic extensions, e.g., `base_pointer->new_func_name ();`
    - can't rule out questionable uses at compile time
    - thus implies run-time checking
- extensibility
  - consider base class with some non-virtual functions
  - attractive for performance reasons
  - Can I create a derived class that overrides a non-virtual function in the code of the base class?
  - No: I have to change the base class definition (add virtual keyword).
  - Java provides "final" to allow optimization
    - same issue!
    - to make an extension
    - I have to go in and remove "final." Not much different.
- implications of definition may be subtle
  - "operations" may be implicit in the definition
  - changing the definition may
    - implicitly change the meaning of code using the definition
    - without producing any errors or warnings
  - implicit conversions
    - single-argument constructors
    - cast operators

Templates

- Str. Ch. 15
- templates provide a general mechanism for type polymorphism
  - write one piece of code (possibly several associated pieces)
  - tailor many implementations to specific types
- most important application: container classes
  - wide variety of algorithms operate on “things”
  - require a small number of basic relations on “things”
  - e.g., a total order
  - examples: lists, sets, heaps, extensible arrays, various search trees, maps (from key to item)
- earlier alternatives for C/C++
  - build using generic types and callbacks/casting
  - example: quicksort in standard C library; arguments include
    - an array pointer
    - number of elements in the array
    - size of elements
    - a callback function for comparisons (must be deterministic!)
  - use the preprocessor
  - example
    - one in Section 2.9.2 (unnecessarily weak binding)
    - pass types, new type names, associated strings, etc. as macro parameters
    - replace in code as necessary
    - multiple instantiations not easy to manage, so more of an external code-generation facility
    - N.B. preprocessor is slightly more powerful than templates, but substantially more error-prone

- goals for templates in C++
  - easy to use
  - efficient
  - general
  - fast compilation and linking
  - simple and portable

- basic model
  - templates for classes or functions
  - parameterized by one or more types and highly-restricted constants
  - definition creates nothing
    - template is instantiated when used
    - unused pieces need not be able to compile

- typical use
  - definitions provided in header file
  - instantiated and often inlined for each compilation unit
  - appear as weak symbols, so linker keeps one copy at most
  - inlining can cause code explosion (more later)

- examples

```
// Apply Newton's method with a given precision
// to a given function, starting from a certain point
template<class F> F newton
    (F (*func) (F arg), F precision, F start) { ... }

// a doubly-linked list of a given class
template<class T> class DoubleList { ... };
```

Using Function Templates

- types can usually be omitted
  - specify a (possibly empty) prefix of necessary types
  - other types must be deduced by compiler
  - example: calling **`newton`** with a callback that takes and returns a **`double`** implies that **`class F`** is **`double`**
- deduction can sometimes require type conversion
- and thus a single template may create an overloaded function

- regardless
  - function templates participate in overloading
  - including those appearing as class members

- resolution is slightly different when template versions of a name exist
  - an otherwise **equal** match between a template and a non-template version
  - i.e., an ambiguous pair according to the matching rules
  - is silently routed to the non-template version
  - (exact matches necessary to allow specialization, but more general rule is questionable)

- note
  - you are **allowed** to include type arguments when calling a function
  - non-template versions cannot match a call with type arguments

- some things cannot be deduced
  - return type
  - internal types (example: define **`newton`** function to use doubles in interfaces, but use something else internally)

Template Checking and Binding

- a given template (in whole or part) may not make sense for some types
  - many argued for explicit constraint mechanism
  - What's the rationale for avoiding? constraints force code replication
    - example
    - some types may allow sorting (say, a function **`sort`**)
    - some functions in a template class use **`sort`**
    - constraints demand that any type provide it
    - forcing creation of a separate template for types without sort
    - instead, simply avoid using functions that require sort with types that don't provide it
- delaying instantiation is not without drawbacks!
  - delayed errors
  - possible unintended variations due to contextual changes (remember call stealing?)
- questions without obvious answers
  - When are templates checked?
  - When are calls bound (i.e., overload resolution)?
  - note: textbook answer and today's g++ are not the same!
- the rules according to the textbook
  1. names that do not depend on any of the class arguments of a template: checked and bound at template definition
  2. names that depend on template's class arguments: checked and bound at first use in compilation unit (**not** at point of variable declaration, etc.)
- **`g++`** rules are…weird
  - #1 is always bound at instantiation, but #2 is bound in definition if the template class type is built-in (e.g., **`int`**)
  - run **`bind-example.cc`** from the web page