Templates

Jim Fawcett

CSE687 – Object Oriented Design

Summer 2017

Agenda

- Basic Templates and Specialization
  - Simple Examples of template functions and classes
- Template parameters
  - Types, integral values, and function pointers
- Template members
  - Member functions of classes may be templates
- Partial Template Specialization
  - Smart pointers

Simple Examples

- Templates are patterns used to generate code, instances of which, differ only in the symbolic use of a type name.
- Templates allow us to write one function for an unspecified type, and let the compiler fill in the details for specific type instances.
- Consider the template function:

```
template<class T>
T max(const T& t1, const T& t2)
{
   return ((t2 > t1) ? t2 : t1);
}
```

- This works beautifully as long as types we instantiate have operations used to implement the function, in this case a magnitude comparison.

Instantiation

- We instantiate the function, for some specific type, by invoking it:

```
int y = 1;
int x = max(2,y);
```

- The compiler infers template parameter type from the type of y.

- What do you think happens with:

```
string s = max(“zebra”,”aardvark”);
```

We will come back to this later.

Template Functions

- A function template is a declaration for a function which uses an unspecified type T as a formal parameter, return value, or local variable.

```
template <typename T>
void swap(T& t1, T& t2) {
   T temp = t1;
   t1 = t2;
   t2 = temp;
}
```

- None of the processing in this example depends on the actual type represented by the generic parameter T. That’s a requirement for templates to succeed.

- Another requirement is that the generic type, T, possess the operations used in the function definition. In this swap function, T is required to have an assignment operator=(…) and a copy constructor, to build the temporary type T object temp, or compiler generated copy and assigment must be valid.

- If these conditions are not met, compilation of the swap function may fail, or, if the compiler can generate implied operations, the function’s semantics may not be what you expect.

Template Classes

- Template-based classes are classes which depend on an unspecified type as a:
  - Formal parameter or return value of a member function.
  - Local variable for a member function.
  - Composed member of the class.

```
template <typename T> class stack {
public:
  stack();
  void push(const T& t);
  T pop(void);
      …
};

template <typename T>
stack<T>::stack() { … }
    …
```

See Handouts/CSE687-OnLine/code/Templates/Stack for more details.

Template Class Instantiation

- We instantiate a class by declaring the type of its parameter, like this:

```
stack<string> myStack;
stack<int> yourStack;
```

- The stack<string> class and stack<int> class are distinct types. Should stack<T> declare a static data member, that member would be shared by all stack<int> instances, but would not be shared by instances of stack<int> with instances of stack<string>.
- Instantiation happens at application compile time, when the compiler sees the concrete type associated with the template parameter. No code can be generated before that time, as the size of T instances is not known.
  - The consequence of this is that you must put all template function and class method bodies in the header file, included with application code, so the compiler sees the definition at the time the parameter is associated with a concrete type.
  - Thus, we have a general rule:

Header files contain only:

» Declarations
» Template declarations and parameterized“definitions”
» Inline function definitions

Rewrite Rules

- A template-based class can be derived from a class that depends on a specific type, say int, by using the following rules:
  - Replace
    ```
    class classname { … };
    ```
    with
    ```
      template <class T> class className { … };
    ```
  - For each of the member functions, replace
    ```
      returnType className::memName(…) { …}
    ```
    with
    ```
    template <class T>
      returnType className<T>::memName(…) {…}
    ```
  - Replace every object definition
    ```
      className obj;
    ```
    with
    ```
    className<T> obj;
    ```
  - Replace every occurrence of the specific type, say int, with the generic type name T.

Template Parameters

Template parameters may be:

- Types defined in applications and libraries

  ```
  template<class T> stack { … }
  stack<Widget> myStack;
  ```

- Native types, e.g., int, double, …

  ```
  template<class T, int i> buffer
  { … }
  buffer<char,255> pathbuff;
  ```

- Other templates, called template template paramters

  ```
  template<class T, template
  <class T>  class V> myClass { …
  }
  ```

  ```
  myClass<int, stack> mc;
  ```
  instead of
  ```
  myClass<int, stack<int> > mc;
  ```

- Constant expressions

  ```
  enum thread_type { default,
  terminating };
  template <thread_type>
  class Thread { … };
  ```

  ```
  Thread<default> myThread;
  ```

- Address of a function or object with external linkage

  ```
  template<class R, class A,
  R(*fptr)(A)> class delegate { … }
  ```

  ```
  delegate<int,double,mfunc> del;
  ```

Template Members

- A class may define members to be templates

```
template <class T> class smartPtr {
public:
   template <class U> smartPtr(U* pU);
   smartPtr();
        …
```

- This class declares a smart pointer and its template member, smartPtr(U* pU), defines a promotion constructor from type U* to smartPtr<T>.

Template Specializations

- The max function:

  ```
  template<class T>
  T max(const T& t1, const T& t2)
  {
   return ((t2 > t1) ? t2 : t1);
  }
  ```

  is fairly useless for literal strings. Their type is const char*, e.g., an address, so max returns whichever string has the higher memory address, not the lexocographically higher. Enter specialization.

- The C++ language guarantees that, should we replace T with a specific type, then that specialization will be used whenever the compiler’s type inference yields that type, e.g.:

  ```
  typedef char* pChar;
  template<>
  const pChar max(const pChar& s1, const pChar& s2)
  {
      return ((strcmp(s1,s2)>0) ? S1 : s2);
  }
  ```

Template Specializations

- A template specialization is the narrowing of the number of types a template accepts as arguments, possibly to one.
  - `template <class T> widget`
    – generic
  - template <class T*> widget
    `- partial specialization to pointers`
  - template <> widget<std::string>
    `- full specialization to one concrete type.`
  - See PTS.cpp in cse687-OnLine/code/templates
- We define specializations to treat those cases in some special manner.

Specialization Targets

- All template classes can be specialized
  - `template <class T> class stack {…};`
  - `template<> class stack<int> {…};`
  - Specializations need not provide exactly the same members. You may add or delete members.

    -
      ```
      template <class U, class V>
      class Widget {…};
      ```

    -
      ```
      template <class U>
      class Wiget <U,int> {…}; // may add, modify, remove
                               // members to suit int param
      ```

Partial Template Specialization

- A template class with two or more parameters may be specialized partially, as well as completely.
  - Fully generic class:

    ```
    template<class U, class V> Widget { … };
    Widget<string,double> myWidget;
    ```

  - Partial specialization:

    ```
    template<class U> Widget<U,double> { … };
    Widget<string,double> yourWidget;
    ```

  - Complete specialization:

    ```
    template<> Widget<string,double> { … };
    Widget<string,double> ourWidget;
    ```

Containers

- Containers are classes which implement data structures containing objects of other classes:
  - `stacks, lists, queues, …`

- Often the logic necessary to manage a container is independent of its contents type, using the type only symbolically. These containers may be implemented with templates to yield a generic container.

- If an operation of the generic type is needed by the container, say assignment or comparison, then the type used to instantiate the container must provide it. Otherwise compilation will fail.

- Example: if we expect to support binary search on a generic array of objects: `array<T> myArray`, then the type T will need to support assignment and comparison so we can sort the array.

Container Types

- Containers are classified as either:
  - **Homogeneous**

    Containers hold only on type of object. They generalize an array of primitive types.
  - **Heterogeneous**

    Containers hold more than one type of object, usually all belong to a common derivation hierarchy.
    - If a container expects a reference to a base class object, it will accept a reference to any type which is derived from the base class.
    - Upward type compatibility is the basis for flexible run-time creation and management of objects.

Template Container Types

- If we implement a container with templates, it will be homogeneous if we instantiate it with a a class with no derivations or with a primitive type, e.g.:

  ```
  array<char> myHomogeneousArray;
  ```

  If, however, we instantiate the container with pointers to the base class of a derivation hierarchy, then the same design yields a heterogeneous container, e.g.:

  ```
  array<myBaseClass*> myHeterogeneousArray;
  ```

  The moral of this story is that you should always design template containers as implicitly homgeneous, allowing instantiation to determine the container type.

Container Semantics

- Containers are said to have either
  - <u>Value semantics</u>:
    Objects are copied into the container
  - <u>Reference semantics</u>:
    The container is passed a reference or pointer to an external object.

Reference Semantics

- Container holds references to objects, e.g., pointers, C++ references, or handles.
- An object can be "in" more than one container simultaneously
- Insertion does no copying
- When containers are destroyed, their objects are not automatically destroyed
  - So you have to manage those resources without help from the container

Value Semantics

- Container holds the objects themselves.
- No object lives in more than one container.
- Objects are copied into the container, so the container owns its objects.
  - All contained objects must be the same size.
  - Types must supply copy and assignment semantics
- When the container is destroyed, the objects are also destroyed.

Container Semantics

- It is easy to simulate reference semantics by using containers designed with value semantics, but holding reference objects, e.g.:

```
array<myObjects*> CollectionOfPointers
```

- The moral of this story is that you should always design containers with value semantics. If the client wants reference semantics that's easy to arrange by using references to objects.
  - The client then takes responsibility to manage the references properly.

Container Iterators

- Iterators are objects designed to traverse a specific container type.
- Often the container must grant the iterator friendship status or provide member accessor functions designed to allow the iterator to do its job.
- A container/iterator class pair generalizes the operations of an array/pointer pair for primitive types.
  - The iterator constructor makes the iterator “point” to an object in the container.
  - Often, the iterator has knowledge and access necessary to traverse some, perhaps complex, internal structure in the container, contained object-by-object.

Iterator Issues

- Either the designer or client must take care not to destroy a container, or change allocations in a container, if it has active iterators. The result would be a dangling reference.
  - We will see later that it is easy to create invalid iterators on std::vector objects by inserting or removing elements.
- Iterators are especially useful in cases where a container may need to have more than one simultaneous traversal.
  - This is a common need when implementing graph algorithms, for example.
- There may also be more than one type of useful traversal, so iterators can be designed for each type.
  - Depth first search and breath first search on graphs are an example.

Conclusion

- Virtually everything we’ve discussed in this presentation is a part of the design or use of the Standard Template Library (STL).
- We will see all of these illustrated with examples, throughout this course.