Reusing Code II
Class Templates

Topics Covered

- Reading: 326-337 (review), 673-698
- Function Template Review
- Class Templates
  - Defining a class template and member functions
  - Using a class template
  - Class templates, pointers, and non-type arguments
  - Class templates, inheritance and specialization
- Fun with Facets
- Homework

What are generic functions?

- A function is code that performs an action
  - The action can be expressed as an algorithm
  - The algorithm usually doesn’t change, even when the type of data it processes does
    » Example: a bubble sort or swapping two values
    » Sorting ints is logically the same as sorting chars
- In C, we use macros to implement generics
  - Example: return absolute value of a number
    - \#define ABS(x) (((x)<0) ? (-(x)) : (x))
Macros & Inline Functions

- Compiler macros have several drawbacks
  - Don't pass arguments like real functions
  - Macros and side effects cause problems
  - Not type-safe, don't obey scoping rules
- Inline functions designed to fix these defects
  - As efficient as compiler macros
  - Type-safe and free from side effects
  - Only drawback? Not generic
    » Overloading solves this problem to some extent

Function Templates I

- What are function templates?
  - A "blueprint" to create functions
  - Function templates use a "type placeholder"
  - In use, compiler generates actual function
- How do you write a function template?
  - Precede definition with template <class T>
  - Replace type names in function with T

```cpp
int ABS(int n) { return n < 0? -n : n; } // normal
T ABS(T n) { return n < 0 ? -n : n; } // template
```

Function Templates II

- How do you use a function template?
  - Put function definition in a header file
    » Cannot (yet) separately compile templates
    » Definition must be seen to be instantiated
  - Call just like a regular function
  - Compiler generates template function

```cpp
cout << ABS(-3); // generates ABS(int);
cout << ABS(-1.5); // makes ABS(double);
```
Class Templates

-Templates can also generate classes
  - Much more common use of templates
  - Called class templates
  - Generate template classes

- Class templates often used for collections
  - Arrangement of values in particular order
  » Examples: vector, list, table, bag, set
  » Allows for common, generic operations

- C++ standard library uses template classes

The Stack Again

- A simple ADT example is a stack
  - Logically, only four operations:
    » Create a stack
    » Destroy a stack
    » Add an item to the top of the stack (push)
    » Retrieve the item on the top of the stack (pop)
  - Implementation may use additional methods
    » For instance, empty(), or full()

Stacks II

- Here is a simple class that meets these specs:
  
  ```
  class IntStack
  {
    int * stack; // Use array in this case
    int ptr, sz; // Top and size of stack
  public:
    IntStack(int size=10);
    ~IntStack();
    int pop();
    bool push(int n); // Return true if OK
  };
  ```
Stacks III

- Here are the constructor and destructor
  - `IntStack::IntStack(int size) : ptr(0), sz(size)`
    - `stack = new int[sz];`
  - `IntStack::~IntStack() { delete[] stack; }

Stacks IV

- Here are the `pop()` and `push()` methods
  - `int IntStack::pop() { assert(ptr); return stack[--ptr]; }`
  - `bool IntStack::push(int n) { if (ptr < sz) { stack[ptr++] = n; return true; } return false; }

Stacks V

- Use the `IntStack` like this:
  - ```
      IntStack s(20);
      int n, count = 0;
      cout << "Enter up to 20 integers or q.;"
      while (cin >> n && s.push(n))
        count++;
      cout << "Thanks. Here are your ints:
      for (int i = 1; i < count; i++)
        cout << s.pop() << " ",
      cout << s.pop() << endl;
  ```
A Stack Class Template I

- Mostly same as creating function template
  - 1. Add template <class T> in front
  - 2. Replace instances of managed type with T

```cpp
template <class T>
class Stack {
    T * stack;
    T pop();
};
```

A Generic Stack Class II

- When writing member functions definitions
  - Precede each with:
    ```cpp
template <class T>
    Stack<T>::Stack(int size) : ptr(0), sz(size) {
        stack = new T[sz];
    }
    ```
  - For class type in definition use Stack<T>

A Generic Stack Class III

- Template functions are implicitly generated
  - Compiler cannot deduce type of T in class templates
  - Instantiate by putting actual type in brackets
    ```cpp
    Stack<int> istack;   // holds ints
    Stack<double> dstack; // doubles
    Stack<string> sstack;  // strings
    ```
  - Example: CTemplates02.cpp
  - Cannot create a stack of references. Pointers?
Given the following type declarations:

```c++
template <class T> class Array;
enum Status { ... }
typedef string *Pstring;
```

Which, if any, of the following def's are in error?

- `Array < int* & > pri (1024);`
- `Array< Array<int> > aai(1024);`
- `Array< complex<double> > acd(1024);`
- `Array< Status > as( 1024 );`
- `Array< Pstring > aps( 1024 );`

---

Containers that hold pointers and those that hold values (or objects) are fundamentally different:
- You have to consider that when designing class
- Different semantics for assignment, comparison, etc.
- Compiler happily creates a meaningless class.

Use container classes with pointers when:
- Want to apply an operation to an existing collection
  - Sorting a list of pointers is "cheaper" than sorting list
- Objects need to be "present" in several containers

Some restrictions on containers holding pointers:
- **Rule 1:** User must supply valid pointers
  - User is responsible for allocating memory, etc.

Example of pointer misuse [CTemplate03.cpp]

```c++
Stack<char *> pstack; // instead of Stack<string>
char * input;         // instead of string
while (cin >> input && strcmp(input, "quit")
    & pstack.push(input))
    count++;
```
Incorrectly Using Pointers II

- Can you use an array instead?
  - Rule 2: Class may require certain behavior
  - Derived types like arrays usually can't be used
  - Often require assignment and comparison
- How about just allocating memory on heap?
  - Stack<Char *> pstack;
  - Char * input = new char[100]; // instead of string
  - while (cin >> input && strcmp(input, "quit")
    & pstack.push(input));
  - count++;

Correctly Using Pointers

- There are two situations where you want to use pointers with templates
  - 1. You already have the pointers, but you want to process them [or present] in different ways
     - This is the idea behind index files in databases
     - "Pointers" let you access data in different order
  - 2. You need to use a collection with polymorphism
     - Can't store an array of references
     - Objects don't work polymorphically

Non-Type Arguments I

- Here's a different version of an Array class
  - template <class T, int n>
  - class Array {
  - T ar[n]: // Notice this is a regular array
  - };
  - Second argument n is "non-type" or "expression"
- Use it to instantiate a class like this:
  - Array<double, 12> eggWeight;
Non-type Arguments II

- There are some restrictions on expression args
  - They must be integer, enum, ref, or pointer types
    - double is out, double & or double * is OK
- Template instantiation occurs at compile-time
  - Argument must be known [constant]
    - Array<double, 12> eggWeight; // OK
    - const int n = 144;
    - Array<float, n> grossWeight; // OK
    - int m = rand();
    - Array<int, m> oops; // Doesn't work

Non-type Arguments III

- Why do this instead of T* version of Array?
  - Data is created on stack, may be more efficient
- Any drawbacks? Yes, several. [Non-type.cpp]
  - Size must be a constant
    - Can't have arrays that are dimensioned at run-time
    - Can't modify or take the address of n
  - Each instantiation creates a new class if n changes
    - Can't intermix different versions of class
- The T* version is probably preferable

Inheritance and Templates

- Templates can be combined with other forms of code reuse such as inheritance & composition
  - Inheritance to create specialized versions
    - template <class T>
    - class Array {
        T* data;
    };
    - template <class T>
    - class ArrayRC : public Array<T> {
    };

Stephen Gilbert  C++ Programming II  Page 22

Stephen Gilbert  C++ Programming II  Page 23

Stephen Gilbert  C++ Programming II  Page 24
Composition and Templates I

- You can create a class template that uses another class template as a data member

```cpp
template <class T>
class Stack {
    Array<T> ar; // Implement using array
};
Stack<int> si;
```

- Instantiates both Array<int> and Stack<int>

Composition and Templates II

- What if you don’t want to use an array?
  - The class T type can be a template type
  - This is called an adapter type

```cpp
template <class T, class C>
class Stack {
    C data; // The implementation container
};
Stack<int, vector<int> > ias; // note spaces
Stack<double, list<double> > dls;
```

Templates can have multiple arguments

- This allows you to associate values at runtime
- [Unlike a struct, where you associate at compile-time]

```cpp
template <class T1, class T2>
class Pair {  T1 a; T2 b; …
};
Pair<char *, int> ratings[4];
```

- Good place to put the typedef to work

```cpp
typedef Pair<char *, int> ratingType;
ratingType rating("Charthouse", 4);
```
Default Template Arguments

- Unlike function templates, class templates can have default arguments
  - You could make sure your Stack class always used a vector by default like this:

```cpp
template <class T, class C = vector<T> >
class Stack {
    C data; // Container for the data
};
```

- Stack<int> ia; // Uses vector for implementation
- Stack<int, list<int> >; // Uses list

Template Instantiations I

- Instantiation means "creating an instance"
  - For objects it refers to object creation
  - For templates it refers to class generation

- Implicit instantiation
  - Occurs when you create an instance of class

```cpp
Array<int> *pi; // no instantiation
pi = new Array<int>(10); // instantiation
```

- Not all functions will necessarily be generated
- Creating pointers doesn't necessarily create object

Template Instantiations II

- Explicit Instantiation
  - Declare the class using the keyword template
  - May be necessary if you refer to an object indirectly
    - Through a run-time generated function pointer, etc.

```cpp
Array<int> *pi; // implicit instantiation
template Array<string>;
```

- The Array<string> class is generated, even though no Array<string> objects are created
Template Specializations

- Allow you to create a version for types that require special handling
  - A SortedArray for C-style strings for example
    ```cpp
    template <class T> class SortedArray { ... }
    template <> class SortedArray<char *> { }
    ```
  - Notice that the type migrates from before to after
    ```cpp
    SortedArray<int> scores; // Use general
    SortedArray<char *> dates; // Use specialized
    ```

Partial Specialization

- Means specialization of portion of type args
  - If you have two args, you can specialize one
    ```cpp
    template <class T1, class T2> class Pair { ...; }
    template <class T1> class Pair<T1, int> { ...; }
    ```
  - Allows specialized pointer versions
    ```cpp
    template <class T> class X { ...; }
    template <class T*> class X { ...; }
    ```
- Partial specialization may not be implemented

Standard I/O

- The first standard library written for C was the standard I/O library
  - The philosophy behind stdio is that everything can be converted to and from characters
- The problems with stdio are that:
  - I/O functions are not easily extended
    » Can't add new format types for scanf, printf
  - Characters are not always 8-bit chars
  - I/O is fairly parochial; hard to internationalize
Standard I/O II

- The first version of C++ I/O had similar problems
  - Only real improvement was ability to easily extend input and output for new types
  - Character size assumptions and internationalization problems remained
- Standard C++ I/O attempts to solve this problem
  - Stream classes are templates, not char streams
  - C++ locales handle internationalization
  - The facet classes provide standardized formatting

Locales Again

- A locale is a container for local customs
  - What characters are used
  - How characters are collated and sorted
  - How numbers are read and written
  - How dates and times are read and written
  - What language is used
- A facet is a class which manages a custom
  - C++ comes with several standard facets
  - All facets are class templates which you extend

Standard Facets

- Every Std C++ implementation supplies these
  - Included in the <locale> header [not <locale>]
  - String comparison: collate
  - Numeric I/O: numpunct, num_get, num_put
  - Monetary I/O: moneypunct, money_get, money_put
  - Time/Date I/O: time_get, time_put
  - Character Classification: ctype, codecvt
  - Message Retrieval: messages
Facets II

- What is the advantage of using facets?
  - Don't have to rewrite common formatting code
    » You simply create a new class from a basic one
    » You override some virtual functions to customize your specific formatting rules
  - Facets make it easy to "plug in" new customs
- For more information, contact Bjarne @:
  - http://www.research.att.com/~bs/3rd_loc0.html

Using a Facet I

- First step is to create a facet object [Facet01, 02]
  - Normally use the global function use_facet()
  - Template function that takes a facet specialization as a template parameter, and a locale as arg
    ```
    const money_put<char>& formatter1 =
    use_facet< money_put<char> > ( cout.getloc() );
    const time_put<char>& formatter2 =
    use_facet< time_put<char> > ( cout.getloc() );
    ```

Using a Facet II

- Use facet member functions to do output
  - Here's how you'd print using money_put.put()
    ```
    showbase(cout);
    formatter.put( 
      cout,      // where's the output going to start
      false,    // use international currency symbol?
      cout.fill(),  // character to use for fill
      125457.L); // Amount [in cents] as a long double
    ```
- Similar process for other facets
Customizing a Facet I

- Facet implementations are not perfect [Facet03]
  - VC++ 6 uses a non-standard form of use_facet
  - BC++ blows up if you try to change locales
    - Seems to work OK if you build a custom locale
  - money_put relies on class moneypunct
  - Create a custom version of moneypunct
    - Install your version of moneypunct into locale
  - imbue your stream with custom version of locale

Customizing a Facet II

- Create a public subclass of moneypunct<char>
  - Regular class, not a class template
  - Override virtual functions to supply format info

```cpp
class MyMoneyPunct : public moneypunct<char>
{
    public:
        char do_decimal_point() const { return '.'; }
        string do_curr_symbol() const { return "US$"; }
    }
```

Customizing a Facet III

- What virtual functions should you override?

```cpp
char do_decimal_point() const; // "."
char do_thousands_sep() const; // ""
string do_grouping() const; // "3" means every 3
string do_curr_symbol() const; // "5"
string do_positive_sign() const; // Usually empty
string do_negative_sign() const; // "-" or "()"
int do_frac_digits() const; // Usually 2
pattern do_pos_format() const; // [symbol,sign,none,value]
pattern do_neg_format() const;
```
Using a Customized Facet

To use the new facet, "install it" into a locale
- Use new operator to create the facet
- The locale will handle deleting facet

```cpp
locale oldLoc = cout.getloc();
locale newLoc(locale(), new MyMoneyPunct);
cout.imbue(newLoc);
ios_base::fmtflags oldFmt = cout.flags();
showbase(cout);
// Use money_put facet here
cout.flags(oldFmt);
cout.imbue(oldLoc);
```

Homework

- Homework 18: Due on Sunday, April 18th
  - Use the basic Array class template [Array01.h]
  - Derive a new GrowableArray class
  - Add constructor, resize(), subscript operator
- Homework 19: Due Sunday, April 25th
  - Add a facet to the Xollar class
  - Create the XollarMoneyPunct class
  - Add an insertion operator to Xollar