6 Software Development with Templates, Iterators, and the STL

Abstract

This chapter aims to provide a better approach to writing code that's meant to be reused. We use templates to write template functions and template classes that are extreme utile. We'll provide iterators for container classes, allowing standard access to the items in a container. By following a standard approach we will write classes that are easier to use and maintain by others, while also taking advantage of the Standard Template Library.

Template Functions

Rather than writing several functions that take different inputs we can simply write one function to handle the different use cases. We can do this by writing one function and a typedef statement like so:

```
typedef ______ item;
item maximal (item a, item b) {
    if (a > b)
        return a;
    else
        return b;
}
```

From here we allow the programmer to fill in any type for the typedef that has a $>$ operator defined and a copy constructor (see pg. 96). However, the typedef approach limits us to one instance of the maximal function. What if we need two?

A more flexible solution is called a **template function**, which is like a function but with one caveat: the definition of a template function depends on an underlying data type. That data type can be defined as *Item* but it's not pinned down to just this specific type. When a template's used, the compiler examines the types of the arguments and automatically determines the datatype of *Item*.

Syntax for a Template Function

```
//Using a typedef statement
typedef int item;
item maximal(item a, item b) {
    if ( a > b)
        return a;
    else
        return b;
```

```
}

//Defining a template function
template <class Item>
Item maximal(Item a, Item b) {
    if ( a > b)
        return a;
    else
        return b;
}
```

The second definition, which uses the template definition, is known as a **tmeplate prefix**. The bracketed code defining *Item* as a class is called the **template parementer**.

Common programming practice is to capitalize the name of template parameters, to differentiate them from specific data types (Int versus int, Item versus item, etc).

An alternative to *class* is *typename*, but *typename* is not supported by older compilers.

Using a Template Function

A program can use a template function with any *Item* type that has the necessary features. In the scope of our previous example, the *Item* definition can be any of the basic data types, or any class that implements a >.

NOTICE: Unification errors regurgitate by the compiler are often the result of a failed template instantiation. Make sure that the template paremeter appears in hte paremeter list of the function.

See pg. 295 for code example.

The ¡algorithm¿ facility containts a swap, max, and min function.

Parameter Matching for Template Functions

The next template function function searches an array for hte biggest item and returns the index.

```
template <class Item>
size_t index_of_maximal(const Item data[], size_t n);
//Precondition: data is an array with at least n items,
//and n > 0.
//Postcondition: The return value is the index of a
//maximal item from data[0]...data[n-1].
```

However this function does not work as intended. The compiler can only do so much guess work on the information it's given. We must be as explicit as possible to ensure the compiler translates our functions properly. The arguments of a template function must therefore be the same.

A Template Function to Find the Biggest Item in an Array

To handle the inconsistency in the previous example we must change the specification slightly.

```
template <class Item, class SizeType>
size_t index_of_maximal(const Item data[], SizeType n);
```

Template Classes

In a similar way that template functions depend on an underlying data type, **template classes** also depends on an underlying data type.

Syntax for a Template Class

The syntax changes slightly between template function and class definitions. We must first change the templace class definition as seen below.

```
//Using a typedef statement
class bag {
    public:
        typedef int value_type;
}

//Using a template class
template <class Item>
class bag {
    public:
        typedef Item value_type;
}
```

The template expression at the beginning of the second example is a **template prefix**. It notifies the compiler that the following definition will use an unspecified data type. Our next step is to implement functions for the template class, which we can see in the typedef statement of the second example. However, outside the class definition there are a few things we need to do to notify the compiler of the proper data type to use.

- The template prefix is used directly before each function prototype and definition.
- Each use of the class name outside its definition is changed to the template class name (such as bag<Item>). The constructor's name simply remains bag.
- Within a class definition we still use the bag's type names, such as size_type or value_type. (It may be better practice to use Item instead of value_type, for semantic reasons).

– Outside member functions, it is necessary to preface usage with the *typename* keyword: typename bag<Item>::size_type.

```
//Overloading the addition operator for the bag class
template <class Item>
bag<Item> operator+(const bag<Item>& b1, const bag& b2)...

//Another example of template class usage
bag::size_type bag::count(const value_type& target) const...
```

The functions return type is bag::size_type, but it's returned before the compiler defines this as a member function. In order for the above statement to be correct we must add the *typename* keyword.

```
tmeplate <class Item>
typename bag<Item>::size_typ bag<Item>::count
                              (const Item& target) const...
```

And finally, we must make the implementation of our template class visible. In the header file we generally place the documentation and prototypes for functions, however we must now also include the actual implementations of all the functions. The reason for this inane requirement is purely a function of assisting the compiler. The recommended manner for avoiding this situation is to move the implementations to a sparate file, but place the include directive at the bottom of the header file (See Figure 6.2 pg.305 for an example).

More About the Template Implementation File

As you can see in Figure 6.2, the final include is of the syntax *bag4.template* as opposed to *bag4.cxx*, this is to remind us that we are using a template file as opposed to a standard source file.

Parameter Matching for Member Functions of Template Classes

While we must babysit the compiler for most function definitions, no such help is needed for member functions of a template class. Unlike ordinary template functions, the compiler is able to match a template data type with a regular data type.

Using the Template Class

Using template classes are fairly straightforward:

```
bag<char> letters;
bag<double scores;
```

Once this code appears, we consider the template parameter to be **instantiated**. In the letters class, *char* is instantiated, while *double* is instantiated in the scores class.

Figure 6.3 (pg.310) is an example of using different "bags" to create a "story" about "Life". A description of this program can be found on pg. 312 of the text.

11 Balanced Trees

Abstract

In this chapter we describe two programming projects that involve trees. The projects in question are improvements of the classes mentioned earlier in the text: the priority queue (pg. 433 Proj. 5) and the set class (similar to the bag class). We will also analyze the time performance of tree algorithms.

B-Trees

Section 10.5 implements the bag class with binary search trees, however their efficiency can sometimes go awry. We will analyze the problem and show one way to fix it.

The Problem of Unbalanced Trees

Let's assume that we have a binary tree that's left-aligned and populated with the values 1 through 5. This is troublesome since this implementation is essentially no better than a linked list; continue adding values to the left branch of the tree and the problem exacerbates. We can solve this problem in several ways, for example we could periodically balance the search trees (explained in Ch.10 pg.538). Another method is to use a B-Trees, where the leaves cannot become deep (relative to the number of nodes).

The B-Tree Rules

Every B-Tree depends on a positive constant integer $MINIMUM$ which serves as the number of entries held in a single node.

- The root may have as few as one entries while every other node has at least $MINIMUM$ entires.
- The maximum number of entries in a node is twice the value of $MINIMUM$.
- The entries of each B-Tree node are stored in a partially filled array, sorted from smallest to largest.
- The number of subtrees n below a nonleaf node is always $n = m + 1$ where m is the number of entires in the node.

Example 11.1. Consider a node that has 42 entries. This node will have 43 children according to our prevoius statement. Each subtree will be referred to as subtree 0, subtree 1,$\cdots$, subtree 42. $\diamond$

- For any nonleaf node the subtrees are ordered from least to greatest value

Example 11.2. Consider a nonleaf node that contains two integer entries 93 and 107. This nonleaf node must have three child nodes based upon example **??**. subtree0 must be less than the value 93, subtree1 must have a value between 93 and 107, and subtree2 must be greater than 107. ◇

– Every leaf in a B-Tree has the same depth.

The Set ADT with B-Trees

The rest of this section discusses methods for implementing B-Trees. I will refer to the text as much as possible to avoid typing out huge chunks of code.

Invariant for the Set Class Implemented with a B-Tree

1. The items of the set are stored in a B-Tree that comply with the aforementioned rules.
2. The number of entries in the tree's root are stored in the member variable data_count. The number of subtrees of the root is stored in the member variable child_count.
3. The root's entries are stored in data[0] through data[data_count-1].
4. If the root has subtrees, then these subtrees are stored in sets pointed to by the pointers subset[0] through subset[child_count-1].

Searching for an Item in a B-Tree

The set class (defined on pg. 551) has a member function (count) that determines whether an item called *target* appears in the set. Remember that count always returns 0 if *target* is not found, and 1 if it is.

We search B-Trees in much the same way we search binary search trees. The basic algorithm is as follows:

– Check the root, if $count = 1$ then we're done!

– If the root has no children, the search is also done.

– Make a recursive call to the children to check for *target*

We justify the recursive call by arguing that the subset of the root *has* to be smaller than the entire set.

Inserting an Item into a B-Tree

The *insert* member function defined in the set class adds a new item to the B-Tree. We'll attempt to implement a loose insertion which *may* result in an illegal tree.

The Loose Insertion into a B-Tree

Our initial condition is that the B-Tree is valid. Our postconditions are as follows:

- If the value is already set, leave the B-Tree unchanged.
- Otherwise add the entry to the set. The B-Tree remains valid, however it's possible that the number of entries in the root of the set may be higher than $MAXIMUM$.

So a basic algorithm for this insertion seems fairly straightforward, and begins much like the search algorithm: find the location of the root's entires that is not less than the new entry.

- Set local variable i equal to the first index, such that data[i] is less than the entry.
- If we find the value of i at data[i] return; there's nothing to do!
- If the root has no children, add a new entry to the root.
- Otherwise, save the value from the recursive call subset[i]→loose_insert(entry);

A Private Member Function to Fix an Excess in a Child

To fix a child with $MAXIMUM + 1$ entries, the child node is split into two nodes that each contain $MINIMUM$ entries. This leaves one extra entry, which is passed upward to the parent.

Back to the Insert Member Function

By using the loose insertion function defined above, we have only two steps to ensure a smooth insertion.

1. If loose insert returns false, we have nothing to do.
2. Otherwise, if the maximum value has been exceeded for the root, fix the tree with out fix_excess function.

The last step can be performed in two parts. We first copy all the pointers of the child nodes and clear the root, essentially bumping the whole tree down one node. We've changed the scope of the problem to our old root, and this is much simpler to fix. We split the old root node into a subset and shift the middle value up to our new root. Refer to page 562 for a pictoral representation of this shift.

Employing top Down Design

Essentially, top down design means we implement functions in such a way that each successive function call solves a smaller and smaller problem. Through this methodology it becomes easier to determine points of failure.

Removing an Item from a B-Tree

The *erase* member function of the set removes an entry from the B-Tree. Most of the work is accomplished with a private member function: *loose_erase* (think *loose_insert*). A simple algorithm for loosely removing a value from a tree looks something like this:

- If *target* does not exist, there's nothing to do
- If we found *target* and it has children, fix the root of the entire tree so that it no longer has zero entries.

The Loose Erase from a B-Tree

- Make a local variable i equal to the first index at data[i] that is not less than *target*.
- Deal with one of four possibilities
  1. The root has no children and we did not find *target*. Nothing to do.
  2. The root has no children and we found the *target*. Remove *target*.
  3. The root has children and we did not find *target*.
     - Make a recursive call to the child subsets.
     - Call a private member function *fix_shortage* to validate the B-Tree.
  4. The root has children and we did find *target*.
     - Instead of removing *target* we will remove the largest item in hte subset.
     - Take a copy of this largest item and place it into data[i] which contains the target.
     - The total effect is the same as removing *target*.

A Private Member Function ot Fix a Shortage in a Child

Example 11.3 (Transfer extra entry from subset[$i-1$])**.** Suppose that subset[$i-1$] has more than the min. number of entries.

- Transfer data[$i-1$] down to the front of subset[i]→data.
- Transfer the final item of subset[$i-1$]→data up one to replace data[$i-1$].
- If subset[$i-1$ has children, transfer the final child of subset[$i-1$] over to the front of subset[i].
- Consult the example provided on page 567 for a pictoral representation of this method.

◇

Example 11.4 (Transfer extra entry from subset$[i+1]$)**.** Compliment the method described in Case 1. ◇

Example 11.5 (Combine subset$[i]$ w/ subset$[i-1]$)**.** Suppose that $i > 0$ but it only has $MINIMUM$ entries.

1. – Transfer data$[i-1]$ down to the end of subset$[i-1]$→data, which removes the item from the root.
   – Shift data$[i]$, data$[i+1]$, and so on, leftward.
   – Subtract 1 from data_count and add one to subset$[i-1]$→data_count.
2. – Transfer all items and children from the current subset to the end of subset$[i-1]$
3. – Delete the node subset$[i]$ and shift subset$[i+1]$, subset$[i+2]$, and so on, leftward.

◇

Example 11.6 (Combine subset$[i]$ w/ subset$[i+1]$)**.** Implement the compliment of the previous example.

◇

Removing the Biggest Item from a B-Tree

Removing the biggest item is simple if the root has no children. We simply copy the last item of data into the reference parameter. If the root has children, we make a recursive call to remove the largest entry from the rightmost child.