CS11 C++ Lab 5

Generic Collections

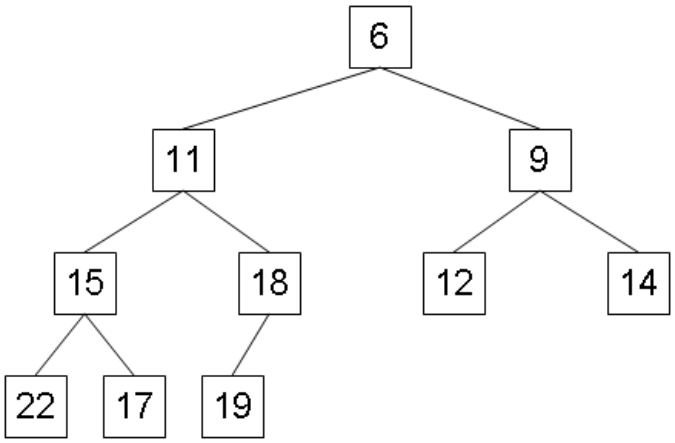
One of the best uses for templates is in building generic containers that can hold any kind of object. The template allows you to use the container in many different circumstances, without having to create a specialized copy of the class for each kind of element you are going to store into the container.

This week you will take a C implementation of a very interesting data structure called a *heap* (not to be confused with the "heap" where memory is dynamically allocated...), and turn it into a C++ template. You will also update the code to throw a few exceptions, since C doesn't have exceptions, and this heap would definitely benefit from being able to raise exceptions in certain circumstances. Finally, once all of that is done, you will write some test code to exercise your new heap template.

Heaps

A **heap** is a data structure that keeps its elements in a particular order. It is also hierarchical; it has a root, and every element has at most two child elements. The main characteristic of a heap is that each node's value is *smaller* than both of its child-node values. Heaps are very similar to binary trees, but they enforce slightly different ordering constraints than binary trees do.

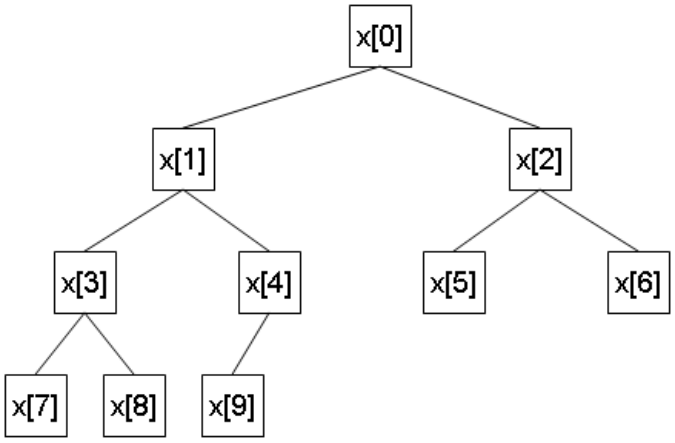
Here is a picture of a simple heap:



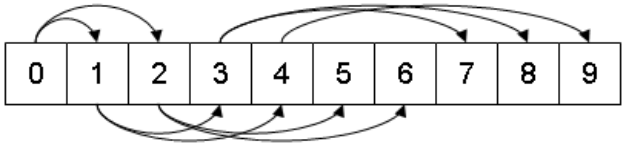
You can see that each node's value is smaller than the values of its children. However, the left child's value may be greater or smaller than the right child's value. (Heaps with parents that are smaller than their children are called *min-heaps*. It is also possible to construct a heap where each node's value is larger than its children; that is called a *max-heap*, as you might expect.) This characteristic is called the "order property," since it specifies the ordering of the values within the heap.

Heaps also have a second property, called the "shape property." This property simply states that all of a heap's leaves span at most two levels, and new leaves are added left to right. You can see that the above heap also satisfies the "shape property."

Another interesting characteristic of heaps is that they can be represented very easily in an array. This means that there isn't any storage overhead from managing node pointers or anything like that. Here is a picture of how you would lay out the above heap in an array:



If you lay this out in the array, you get:



The rules for navigating the heap inside the array are:

- Index 0 is the root.
- For a given index *idx* (in a zero-based array):
  - The parent node's index is: (*idx* - 1) / 2
  - The left child node's index is: 2 * *idx* + 1
  - The right child node's index is: 2 * *idx* + 2

Adding new elements to the heap is a reasonably efficient procedure, taking O(log N) time. In addition, getting the smallest value from the heap is also O(log N). Both of these operations require some reshuffling of the values in the heap, to make sure that the heap properties are maintained. But they are pretty simple to implement.

Heaps are often used as the underlying implementation of another useful data structure called a *priority queue*, where items with different priorities are thrown into a queue, and then the highest (or lowest) priority item is always available at the front of the queue. You can see that this is trivial to implement with a heap data structure.

This is a *very* brief overview of the heap data structure. You can read this page about heaps if you want more information (we are implementing a "binary min-heap"), or you can look at any decent algorithms textbook.

Your Task

Fortunately, you don't have to implement a heap from scratch this week. You can start with an implementation of a heap, but it's written in C, and it only works with `float` elements! So, what you will need to do is this:

1. **Download a copy of the heap implementation in C.** The files are:

   - `heap.h` - the "public interface" and data structure
   - `heap.c` - the implementation of the heap of floats
   - `main.c` - some simple test code to exercise the float heap

   Look over the source code so that you can get an idea of what's in those files. It's not necessary to understand *exactly* how the heap does what it does, but you should at least be aware of the different functions and how they are used within the heap.

   If you want to give the C code a try, you can compile and run it like this:

   ```
   gcc heap.c main.c -o heap
   heap
   ```

2. **Convert the C heap into a C++ class-template.**

   The C version of the heap can only store `float` values, but it would actually be useful for a lot of different data types. So, your goal for this week is to take this C code and make it far more generic by implementing it as a C++ class template.

   Your heap class-template code will exist entirely in a C++ header (`.hh`) file, since templates don't have corresponding source (`.cc`) files. You could call the file `heap.hh`, for example.

   Parameterize your template on two values:

   - The type of the element stored in the heap.
   - The maximum number of elements that can be stored in the heap.

   For example, to create a heap that can store up to 32 integers, you might instantiate your new heap template like this:

   ```
   Heap<int, 32> h;
   ```

   Note that the C code keeps track of individual "heap instances" by using a struct. Because C doesn't support objects, the various functions must take a pointer to the heap-struct that the function should actually operate on. However, your C++ functions shouldn't need to pass around a struct, since the C++ heap will be an object, and the functions will become member functions on the heap object.

   Thus, the data members in your class-template will be the contents of the `float_heap` struct in `heap.h`, appropriately modified to use the template parameters. Make sure those members are private, of course. (This should also make the `MAX_HEAP_ELEMS` constant unnecessary, since this is now one of the template parameters.)

   The functions in `heap.h` define the public interface of the heap functionality, so you should make these the public member-functions of your new heap template. Go ahead and copy the implementation of those functions into your template, but keep in mind that you will have to update it to work as a C++ class-template instead of a bunch of C functions. (For example, they shouldn't need to pass around a

struct-pointer when you are done with them!) Note that the `init_heap()` function serves the purpose of a constructor in the C code, so that functionality should go into your new C++ heap constructor.

Because your template is parameterized on the maximum number of elements in the heap, you should not have to dynamically allocate any memory in this class. Simply use the template's size parameter to specify the dimension of the array of elements. Also, because there is no dynamic allocation, your destructor shouldn't have to do anything.

The C implementation of the heap has several helper functions, and even a few macros! Make all of those into private helper functions in your template implementation. You should especially make sure to convert the C macros into C++ inline functions, so that they have some type-checking!

3. **Add exception reporting to the public interface.**

   The heap has two main operations, "add a new value," and "get the first value." Of course, the heap might be full when you try to add the new value, or the heap might be empty when you ask for the first value. The C implementation of the heap will trip an assertion when either of these conditions occur, but that isn't good at all. Instead, you should do this:

   - If the heap is already full when the caller tries to add another element, throw a `std::overflow_error` object. Pass an appropriate message to the exception's constructor, such as "heap full!"
   - If the heap is empty when the caller tries to get the first element, throw a `std::underflow_error` object. Again, pass an appropriate message to the exception constructor so that the cause will be obvious to whoever gets to catch the exception.

   Both of these exception classes are part of the C++ Standard Library, so you don't have to implement them yourself. They are declared in the `stdexcept` header, and are in the `std` namespace.

   Note that the heap code also contains many assertions, to catch programming bugs. Don't convert all of the assertions to exceptions; just throw exceptions in the particular situations specified above.

4. **Write some test code for your heap template.**

   You need to make sure your C++ heap implementation works properly. If you want, you can adapt the simple test code in `main.c` to exercise your new heap template. Of course, you should change the code from using C-style IO, to C++ stream IO with `cout`, and the filename should end with `.cc` instead of `.c`.

   Next, add a test to verify that your heap will throw the proper exceptions when it is misused. You should put this test in a separate function (called `test_exceptions` or something similarly descriptive), and then call it from `main`. This test can create a small heap (a max size of 4 elements, for example), and make sure that your exceptions are thrown at the proper times.

   The exception test should not just verify that an exception is thrown; it should make sure that the *right kind* of exception is thrown, and report an issue if either a different exception is thrown, or if no exception is thrown at all. You can do that with a try/catch statement of this form:

```
try {
    // Do something invalid, that should throw an exception.

    // If we get here then the operation didn't throw an exception!
    TODO;  // Report a failure - nothing was thrown.
}
catch (expected_exception &e) {
    TODO;  // Report success.
}
catch (...) {
    // If the function throws something other than what we expected,
    // this catch-block will run.

    TODO;  // Report a failure - the wrong thing was thrown.
}
```

   Obviously, you will need to fill in the details. These kinds of tests are called "negative tests," because the test is doing something that is known to be invalid, to make sure that the code responds to the bad usage correctly. Negative tests are actually very important in software development; many bugs can end up hiding in error-handling code when it is not properly exercised.

   Make sure that your heap template passes all of the tests you created! If your heap doesn't pass the "sorted order" test then double-check that you copied the code to the C++ template correctly. If you can't find the problem after more than a half-hour of debugging, ask for help.

5. **Try out your heap with some strings!**

   You may have noticed that the heap is careful to use only the < comparison operator to do all of its comparisons between `float` values. Your C++ template implementation should be the same; it should *only* use < for comparisons. This is done for a good reason; if you are careful in your template implementation then you can use it with *any* type that provides an implementation of the comparison operators.

   The C++ `string` class (declared in the `string` header) also supports comparison with the < operator. For example, you can do this:

```
string s1 = "red";
string s2 = "green";
```

```
cout << "s1 < s2 = "
     << (s1 < s2 ? "true" : "false") << endl;
```

This should print out "`s1 < s2 = false`" since "green" comes before "red" lexicographically.

Since `string` supports comparison, you should be able to use `string`s in your heap as well. This will be a good test for your heap template, to make sure that it is really a generic template.

Write another test (again, in a separate function) that constructs a heap of strings, and then add at least ten different strings to the heap, in some unsorted order. Then, fetch all ten values out of the heap and print them. If they are printed in sorted order then your heap works properly with class types as well as primitive types. This illustrates the power of both generic template classes, and C++ operator overloading.

You might want to adapt the code in `main.c` for this purpose; there really isn't any reason why you can't use the same kind of process to test your heap with strings this way.

Once you have completed all of these tasks, place your heap template and your test code into your `~/cs11/cpp/lab5` directory.

Last updated November 13, 2007.