CS 440/540: Container in C

Due September 23rd, 11:59 PM.

(This document was last modified on Thursday, September 12, 2019 at 05:17:52 PM.)

The topic of the assignment is how to do containers in C. This will give you some understanding of what goes in behind-the-scenes in C++, and also serve as a good review of C.

A secondary goal of this assignment is to give you experience in a form of test-driven development. I do not tell you exactly what to write, but rather I give you a piece of code (the test) that you must conform to.

The assignment is not to be completed in strict C, but rather more like C++ without classes. You may not use any C++ classes, inheritance, templates, member functions, virtual member functions, `dynamic_cast`, etc. Also do not use any of the standard C++ containers such as `std::vector` or `std::list`. You will need to use C++ references. You must compile with `g++`, not `gcc`.

You do not need to handle any error conditions not explicitly specified. For example, you may assume that the iterator returned by `end()` is never incremented. If the user does increment this, then this will cause undefined behavior.

You may find these two posts at Hacker News, and of course the accompanying referenced posts, useful.

A Container in C

You will use a macro to implement a “template” for a double-ended queue container, known as a deque, in C. As a “template”, it will be able to contain any type without using `void *` (which violates type safety). The requirements are given via the `test.cpp` program, linked below. You are to write the macro so that this program will compile and run properly. You must implement the deque as a circular, dynamic array. In other words, you are not allowed to use a linked list.

You should use two `struct` types: one to represent the container itself and a another for the iterator. The `struct` type for the container itself would contain any bookkeeping information that you need. The iterator `struct` type would contain any bookkeeping information that you need to maintain the iterator.

Since the container must have slightly different source code for each type, you cannot code it directly. Instead, you need to write a long macro, named `Deque_DEFINE()`. The macro takes one argument, which is the contained type, provided as a typedef if it is not already single word. This macro will contain all the definitions needed for the container. By making the name of the contained type part of the generated classes and functions, a separate instance of the source code is generated for each contained type.

For example, the `struct` for a container containing `MyClass` objects would be named `Deque_MyClass`, while the `struct` for a container of `int` variables would be named `Deque_int`. A container for pointers to `MyClass` objects would be created by first defining a `typedef`:

```
typedef MyClass *MyClassPtr;
```

and then instantiating the container code by `Deque_DEFINE(MyClassPtr)`.

I strongly suggest that you first do this part as regular code. When you are sure it is working, then convert it to a macro as the last step. Also, after converting it to a macro, save the original. If you find a bug, fix it in the original, then convert to a macro again.

An example of how to use a macro as a template mechanism is here.

Evaluation

Your code should compile with no warnings. Your code must not have any fixed limits. You must be able to put any number elements into your deque, etc. Your program must not have any memory leaks, nor any other types of memory errors, such as using data before it is initialized, writing to beyond a valid memory region, etc. Your program must not output any debugging messages. Use the `valgrind` command to test your submission.

Your code should be in one file, `Deque.hpp`. Your header files must have include guards.

The test program for the assignment is here. Correct output is given here. Your performance will of course not give exactly the same numbers. Your code must work with the unaltered `test.cpp`. (You are of course allowed to modify it on your own, as long as your submission still works with the unaltered version.) We may also test your code with other test files, but we will conform to the API used in the provided `test.cpp` file.

Your grade will be based on correctness. Correctness will include whether or not your output is correct, and whether or not you can scale, or have any memory (or other resource) leaks. Correctness will also depend on whether or not you have followed instructions, such as not using inheritance or member functions. If your code does not work correctly with the provided `test.cpp`, you must provide your own test code, which we will use to assign partial credit, along with an explanation in your `README.txt` as to what works. To receive credit for a functionality, your test code must test it. In other words, if you don't work with our `test.cpp`, and your supplied test code only tests `push_back()`, then you will not receive credit for being able to `push_front()`, even if you claim to be able to do it.

Performance will only be a factor for this assignment if you are unusually slow.

The test program is provided as a means for you to check whether or not your program is as expected. In other words, it tests conformance. It is NOT intended to be especially helpful in debugging, however. You may find it hard to understand, etc. So you will almost certainly need to write additional test code to help you find your bugs. Also, we may update the test program periodically.

You may of course develop your program on any machine, but it will be tested on the Linux classroom machines, and your grade will be determined by how it works on those.