Some Tips for Calling C++ Sorts from C

Sam McCauley

Written September 17, 2021; Last Edited September 18, 2021

As discussed in class, the standard method of sorting in C is to use `qsort`. Unfortunately, this function is not very fast for a few reasons.

There are several ways to deal with this; for example, a handwritten sort implementation is often going to be competitive with `qsort`. This handout focuses on another method: calling `C++` code from your `C` code.

Using the simple `std::sort` method, I was able to decrease the time sort took in my code by almost 50%, even on an array of `struct`s. Using primitive types and/or calling one of the Boost library methods may be even faster for some use cases.

1 Tips for Calling `std::sort`

1.1 Making a Wrapper Function

It is almost certainly useful to create a file with a *wrapper function*—a simple function that will call the `C++` code for you. This is helpful because we can build only this file with `g++` (the `C++` variant of `gcc`), allowing us to build the rest of our files with `gcc`.

Your file should probably look something like the following excerpt. I called mine `sort.cpp`:

```
1  #include <algorithm>
2  #include "struct_file.h"
3
4   bool int_struct_Comp(Int_Struct s1, Int_Struct s2) {
5           return s1.cost < s2.cost;
6   }
7
8   extern "C" void cpp_sort_int(int* A, int size){
9           std::sort(A, A + size);
10  }
11
12  extern "C" void cpp_sort_sumSolutions(Int_Struct* A, int size){
13      std::sort(A, A + size, int_struct_Comp);
14  }
```

Let's go through this file line by line.

Line 1: the `algorithm` is necessary to use `std::sort`.

Line 2: replace this line with an include of the file where you defined your `struct` (here, as an example, we have a `struct` called `Int_Struct` which contains a single `int`; this definition is stored in `struct_file.h`). If you're not sorting structs, just remove this line.

Line 4: comparison function for the objects you're sorting. Note: for `std::sort`, *you do not need to do this with primitive types.* `C++` knows how to sort `int`s, `float`s, etc. You only need to include this if you're sorting something like `struct`s.

Line 8: this is an example of a wrapper function that will sort integers. (Note that it does not pass the comparison function to `std::sort`.) It can be easily modified to sort doubles, etc. Note that `size` is the number of entries in `A`—unlike in `C`, you do not need to give the size of each entry.

Line 12: this is an example of a wrapper function that uses an arbitrary comparison function (from above) to sort an array. This is provided as the third argument to the function.

1.2 Calling the Wrapper Function

First, add declarations your wrapper functions to a header file (perhaps `sort.h`), and make sure that file is included wherever you call the functions from. You do not need to repeat `extern` in your header file. My `sort.h` looks like this:

```
void cpp_sort_int(int* A, size_t size);

void cpp_sort_sumSolutions(SumSolution* A, size_t size);
```

Then, substitute your call to `qsort` with a call to your wrapper function. You just need to pass in the array and the size. Something like this:

```
cpp_sort_sumSolutions(table, size);
```

1.3 Changing the Makefile

You need to make sure that your makefile compiles `sort.cpp`—and it needs to do so using `g++`, not `gcc`.

There are many ways to do this; here's one. First, add `sort.o` to the `OBJ` variable. Then, write a rule for `sort.o` that has dependency `sort.cpp`, and compiles it using `g++`. Note that the flags for `g++` are essentially identical to those for `gcc`. Overall, the relevant part of your makefile may look something like this:

```
#put any desired compilation flags here. Feel free to remove O2 or change to O3 or Ofast
#lm is to help find the math library
#make sure to run "make clean" if this is changed
#lstdc++ is necessary to include C++ code
CFLAGS=-Ofast -lstdc++ -lm

#flags to use when running make debug
#replaces CFLAGS
DEBUGFLAGS=-g -O0

#files to be kept up-to-date
OBJ = test.o twotowers.o sort.o

#include any files beyond .c files that depend on .o object files here
#(this likely only includes headers)
DEPS = twotowers.h

#only need test.out to build 'all' of project
all: test.out

#adds flags (set above) for make debug
#make sure to run "make clean" if no changes to source files
debug: CFLAGS=$(DEBUGFLAGS)
debug: test.out

#this rule says that every .o file needs to be compiled using the corresponding .c file
%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

#compile sort.o using g++
sort.o: sort.cpp sort.h
	g++ -std=c++11 -c -O3 -o sort.o sort.cpp

#this rule links the object files together and stores the output in test.out
test.out: $(OBJ)
	$(CC) -o $@ $^ $(CFLAGS)
```

A few things to note about this. First, it seems like there are two rules for `sort.o`: both the `sort.o` rule, and the `%.o` rule that matches any `.o` file. However, makefiles only take the most specific rule for a file—so for `sort.o` it will only use the `sort.o` rule.

We need to add the `-lstdc++` flag to the `gcc` compiler so that it can parse the `C++` portion of the code.

`-c` is necessary; it tells `g++` not to do linking yet (we'll do that later with `gcc`). `-std=c++11` is optional; if you do the `C++` code above it doesn't make a difference either way. (Modern `C++` uses `C++11` features pretty extensively, so it's not a bad idea to have.) You can change any optimization flags you want; `-O3` does seem to make a considerable difference in terms of sort speed.

Remember that if you change your makefile you should run `make clean` before `make`.

2 Boost sorting algorithms

Modern `C++` libraries provide sorting algorithms that can be faster than `std::sort`.

The Boost library, in particular, provides several such functions, and (just like the above) you can swap them out with your previous sort functions without too much effort.

Again we're going to create a `sort.cpp` file. But, there are a few steps first.

First, make sure Boost is installed on your computer. It is available already on the lab machines (you don't need to do anything for this step on the lab machines). I believe the version on the lab machines is 1.71.

Next, you must include a reference to the Boost library in your code. I added the following to my `sort.cpp`:

```
#include <boost/sort/sort.hpp>
```

After that, you should be ready to call a Boost library function.

2.1 Choosing a sort algorithm

Boost has a number of available sort functions; some optimized for particular kinds of data. The manual for these is available here: `https://www.boost.org/doc/libs/1_71_0/libs/sort/doc/html/index.html`.

In general, if sorting is the bottleneck for your algorithm, one thing you can do is play around with a few of their available options and see if any of them help. For example, they have dedicated functions for sorting integers, floats, strings, nearly-sorted data, etc.

For example, `pdqsort` is a general-purpose algorithm that's a lot like quicksort, but tailored to avoid many bad quicksort cases. I can call `pdqsort` with the following replacement in `sort.cpp`:

```
  //std::sort(A, A + size, sumSolutionComp);
  boost::sort::pdqsort(A, A + size, sumSolutionComp);
```

One thing to remember: we're focusing on single-processor efficiency in this course, so don't use any parallel algorithms.