

PA0: C++ Compiler Setup

Overview

You will install a C++ compiler for your operating system of choice. You will then complete a deque interface and use your compiler to compile and test your project. You will also compile and test the project on the CISE servers.

Note: Start early so if you run into any technical issues you have time to ask the TAs for help. Be sure to register for a CISE account at <https://register.cise.ufl.edu/> early as it can take time for your account to be created.

Structure

This project is broken up into four parts:

1. Install the GNU G++ compiler on your computer
2. Complete the project interface
3. Compile and run the project on your computer
4. Compile and run the project on the CISE servers

Compiler Installation

There are many possible ways to compile C++ code. For this class, we will be using G++, the C++ compiler from the [GNU Compiler Collection \(GCC\)](#).

Windows

On Windows, we will install GCC through the Windows Subsystem for Linux, a compatibility layer for running Linux tools and executables from Windows. You will need Windows 10 for this. Talk to a TA if you have an earlier version of Windows and are unable to upgrade to Windows 10.

1. Open a Powershell window as Administrator and run the following command: `Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux`.
2. When prompted, restart your computer.
3. Go to <https://www.microsoft.com/store/productId/9NBLGGH4MSV6> or search for “Ubuntu” in the Windows Store and install the app.
4. Launch the new app (search for “Ubuntu” in the start menu or look for the app in the app list).
5. Wait for the distro initialization to complete.
6. When prompted, enter a username and password.
7. Update the distro’s packages by running the following command: `sudo apt update && sudo apt upgrade`.
8. Install GCC and other needed tools by running the following command: `sudo apt install build-essential`.

Linux

The following instructions are for Ubuntu. If you are running a different distro, you will need to find the corresponding commands for your distro.

1. Open your terminal of choice.
2. Update the distro’s packages by running the following command: `sudo apt update && sudo apt upgrade`.
3. Install GCC and other needed tools by running the following command: `sudo apt install build-essential`.

MacOS

On MacOS, the “g++” command is an alias to the Clang compiler. To get the real G++ compiler, we will need to install it through [Brew](#).

1. Open the Terminal app.
2. Install make by running the following command: `xcode-select --install`
3. Install Brew by running the following command: `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`
4. Run the following command to update brew's packages: `brew update && brew upgrade`
5. Run the follow command to install GCC: `brew install gcc@9`
6. Run the following commands to update your PATH environment variable and alias g++-9 to g++:
`echo "PATH=\"/usr/local/bin:$PATH\"" >> ~/.bash_profile` and `sudo ln -s /usr/local/bin/g++-9 /usr/local/bin/g++`
7. Close all Terminal sessions for changes to take effect:
 - a. Hold command and press tab to view open programs
 - b. While holding command key tab until Terminal instance is selected
 - c. Press Q once Terminal icon is selected to quit all Terminal sessions
8. Verify g++ is installed by running `g++` in your terminal and verify you get the following output:

```
g++: fatal error: no input files
compilation terminated.
```

Once you have finished installing GCC, run `g++ -v` in your terminal and take a screenshot of the output.

Complete the Project Interface

Now that you've installed the compiler, you can complete the project interface. In this project, you will be completing the interface for a deque, backed by a doubly linked list.

Start by downloading the project zip folder from Canvas and extract it. You can use any text editor to open the project source files. If you are using an IDE like Visual Studio or CLion, take care that you are **only** using it to edit the source files, **not** to compile and run the code. You must do that on the terminal using the provided makefile (covered in the next section).

When completing the interface, you may add new public or private functions or member variables. However, you may not modify or remove existing functions or member variables. Additionally, you may not change the way the tests are written. E.g. You may not modify the tests so that a new function you create is called in the test case. Do not create a main function. The testing framework will generate it automatically.

You may use anything in the C++ STL (other than the deque class) to complete the project. However, you must back your deque with a doubly linked list you've created. I.e., you may not use a vector or list as the backing structure for your deque. We encourage you to explore the new features added in the C++11, C++14, and C++17 standards, such as smart pointers.

Interface Details

T refers to the type specialization of the templated deque.

Method	Description
<code>deque()</code>	Constructs a deque object, initializing necessary member variables, etc. The deque has no maximum size: the size is only limited by the amount of memory in your computer.

void push_front(T data)	Adds <i>data</i> to the front of the deque.
void push_back(T data)	Adds <i>data</i> to the back of the deque.
void pop_front()	Removes the item at the front of the deque. Throws an <code>std::runtime_error</code> if the deque is empty.
void pop_back()	Removes the item at the back of the deque. Throws an <code>std::runtime_error</code> if the deque is empty.
T front()	Returns (by value) the item at the front of the deque. Throws an <code>std::runtime_error</code> if the deque is empty.
T back()	Returns (by value) the item at the back of the deque. Throws an <code>std::runtime_error</code> if the deque is empty.
size_t size()	Returns the size of the deque().
bool empty()	Return true if the deque is empty and false otherwise.

File Details

To aid in completing the project, we have provided a project skeleton on Canvas. Here is a short description of the files provided:

- deque.h - Contains the declaration and definition of the deque class and its member functions.
- catch.hpp – Catch framework header file. **Do not** modify this file.
- catch_main.cpp – Separate compilation unit for the catch framework to speed up the compilation process. **Do not** modify this file.
- tests.cpp – Contains sample Catch unit tests. You may add your own test cases to this file.
- makefile – Contains rules for compiling the project. **Do not** modify this file.

Testing

To test your implementation, we will use the [Catch](#) testing framework. Sample test cases are provided to show how your class will be used and to show how Catch tests are written so you can write your own. A basic catch test has the following format:

```
TEST_CASE("Test case name", "[Test case tags]") {
    // test case setup
    // test case assertions
    // test case tear down
}
```

For example, a basic deque unit test may look like the following:

```
TEST_CASE("Test push_front", "[deque]") {
    deque<int> dq;
    dq.push_front(1);
    dq.push_front(2);
    CHECK(dq.back() == 1);
    CHECK(dq.front() == 2);
}
```

Compile & Run the Project Locally

To compile the project, we will use [make](#), a tool for controlling the generation of executables and non-source files from the project's source files. For our purposes, you don't need to know the details of make as we will provide a makefile for you, but if you're curious here's a quick primer.

Make is made up of rules and targets, which are placed in a file aptly named “makefile”. Rules take the following basic form:

```
target: dependencies
      commands
```

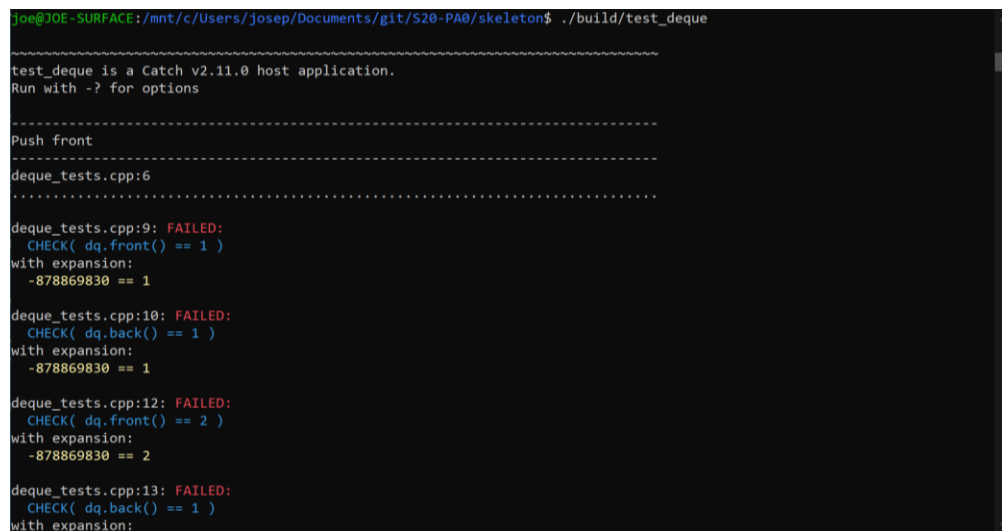
target is the name of the executable or file that will be created when the rule finishes running.

dependencies are a list of files on which the *target* depends. This allows make to skip executing a rule if the timestamp of the target is newer than the timestamps of all the dependencies.

commands is a list of commands (such as `g++`) that will be used to generate the *target* from the *dependencies*.

To compile the project, open your terminal and use the `cd` command to browse to the folder containing the project source files and makefile. Note that on the Windows Subsystem for Linux you can browse to your Windows user directory by `cd`ing to `/mnt/c/<windows_user_name>`. Once inside the project folder, run `make` to compile the project. The first time you run this it may take a few seconds, since Catch takes a long time to compile. Subsequent runs should be much faster since catch will not need to compile again.

Once the project is compiled, run the project with the command `./build/test_deque`. The output will look like the following by default:



```
joe@JOE-SURFACE:/mnt/c/Users/josep/Documents/git/S20-PA0/skeleton$ ./build/test_deque

test_deque is a Catch v2.11.0 host application.
Run with -? for options

-----
Push front
-----
deque_tests.cpp:6
.....

deque_tests.cpp:9: FAILED:
  CHECK( dq.front() == 1 )
with expansion:
  -878869830 == 1

deque_tests.cpp:10: FAILED:
  CHECK( dq.back() == 1 )
with expansion:
  -878869830 == 1

deque_tests.cpp:12: FAILED:
  CHECK( dq.front() == 2 )
with expansion:
  -878869830 == 2

deque_tests.cpp:13: FAILED:
  CHECK( dq.back() == 1 )
with expansion:
```

Beginning of output with no files modified

```

deque_tests.cpp:104: FAILED:
CHECK( dq.size() == 2 )
with expansion:
140736609485498 (0x7fffc9d82ba) == 2

deque_tests.cpp:106: FAILED:
CHECK( dq.size() == 3 )
with expansion:
140736609485498 (0x7fffc9d82ba) == 3

deque_tests.cpp:108: FAILED:
CHECK( dq.size() == 2 )
with expansion:
140736609485498 (0x7fffc9d82ba) == 2

deque_tests.cpp:110: FAILED:
CHECK( dq.size() == 1 )
with expansion:
140736609485498 (0x7fffc9d82ba) == 1

deque_tests.cpp:112: FAILED:
CHECK( dq.size() == 0 )
with expansion:
140736609485498 (0x7fffc9d82ba) == 0

=====
test cases: 10 | 1 passed | 9 failed
assertions: 34 | 7 passed | 27 failed

joe@JOE-SURFACE:/mnt/c/Users/josep/Documents/git/S20-PA0/skeleton$

```

End of output with no files modified

Your goal is to implement the interface so that all our sample test cases pass. Your output should then look like the following:

```

joe@JOE-SURFACE:/mnt/c/Users/josep/Documents/git/S20-PA0/src$ make
mkdir -p ./build
g++ -std=c++17 -MMD -MP -Wall -Wextra -pedantic -g deque_tests.cpp -c -o build/deque_tests.o
g++ -std=c++17 -MMD -MP -Wall -Wextra -pedantic -g catch_main.cpp -c -o build/catch_main.o
g++ -o build/test_deque build/deque_tests.o build/catch_main.o
joe@JOE-SURFACE:/mnt/c/Users/josep/Documents/git/S20-PA0/src$ ./build/test_deque
=====
All tests passed (34 assertions in 10 test cases)

joe@JOE-SURFACE:/mnt/c/Users/josep/Documents/git/S20-PA0/src$

```

Output with all tests passing

Note that the program's return code will be the number of failed tests. Also note that the sample tests provided will ensure your implementation correctly conforms to the interface but are not meant to be exhaustive. You are encouraged to write your own test cases to ensure your project works properly under all edge cases.

Run `make` and `./build/test_deque` and take a screenshot of the output. For the screenshot you do not need to have all the tests passing.

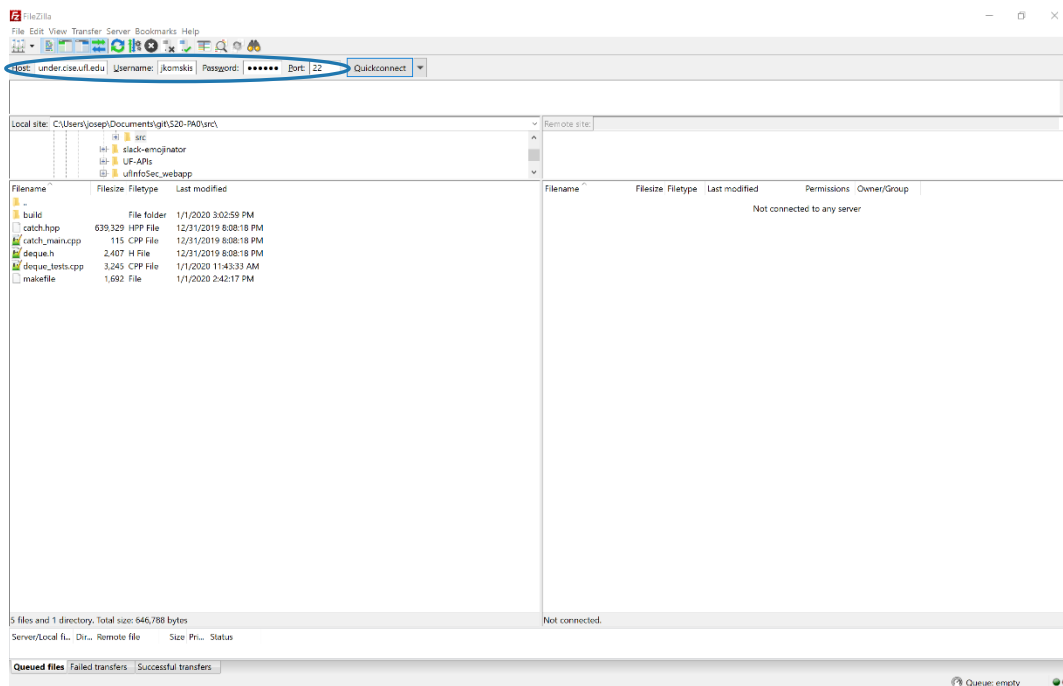
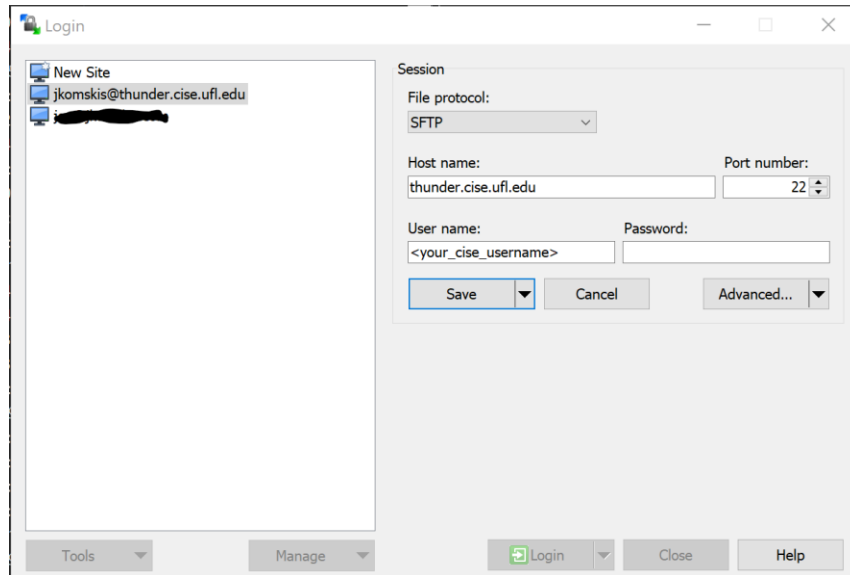
Compile & Run the Project Remotely

We will compile and test your project on the CISE servers. This will allow you to test your project on the exact same environment we will use for testing. Compiling and testing the project locally on your machine should give the same result as compiling and testing on the CISE servers, but there is always a small chance for some disparities. Thus, we recommend testing your project on the CISE servers so you can be absolutely sure your code will work when we test it.

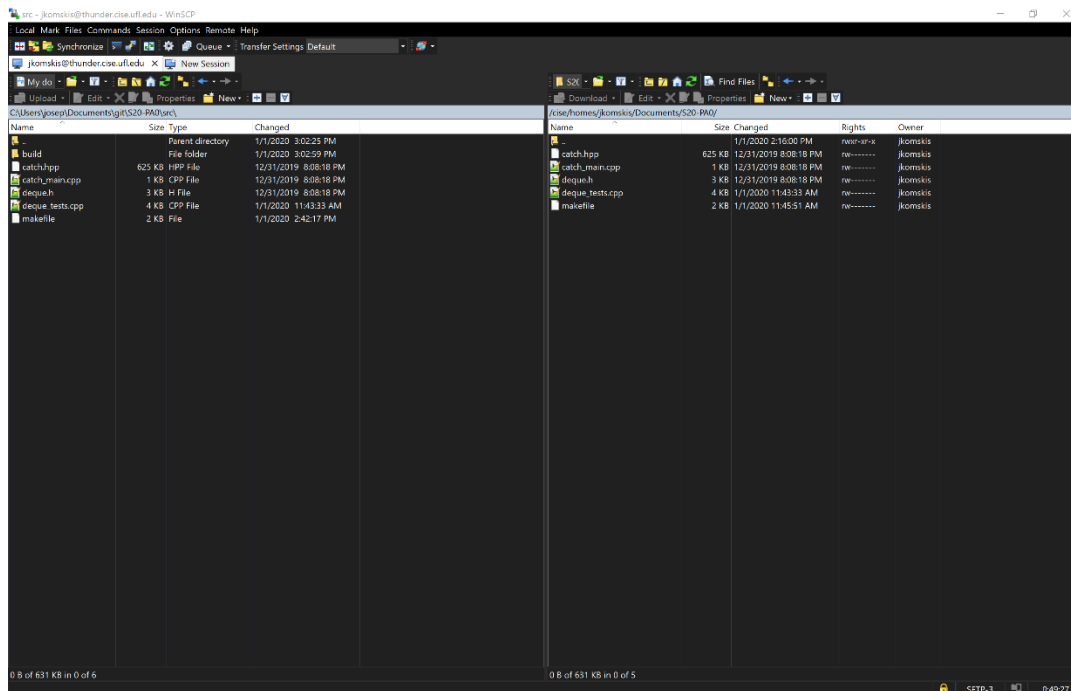
CISE has two servers for students: `storm.cise.ufl.edu` and `thunder.cise.ufl.edu`. We will be using `thunder`. These servers run Ubuntu and provide no GUI, only a terminal environment. The steps for compiling the project on the CISE servers are the same as compiling locally. If you have not already, register for an account at <https://register.cise.ufl.edu/>. The account creation can take some time.

First, you will need to transfer your project files to the server using an SFTP client. We recommend [WinSCP](#) for Windows (download and install the exe from the website) and [FileZilla](#) for MacOS (download and install the dmg from the website) and Linux (install with `sudo apt install filezilla`).

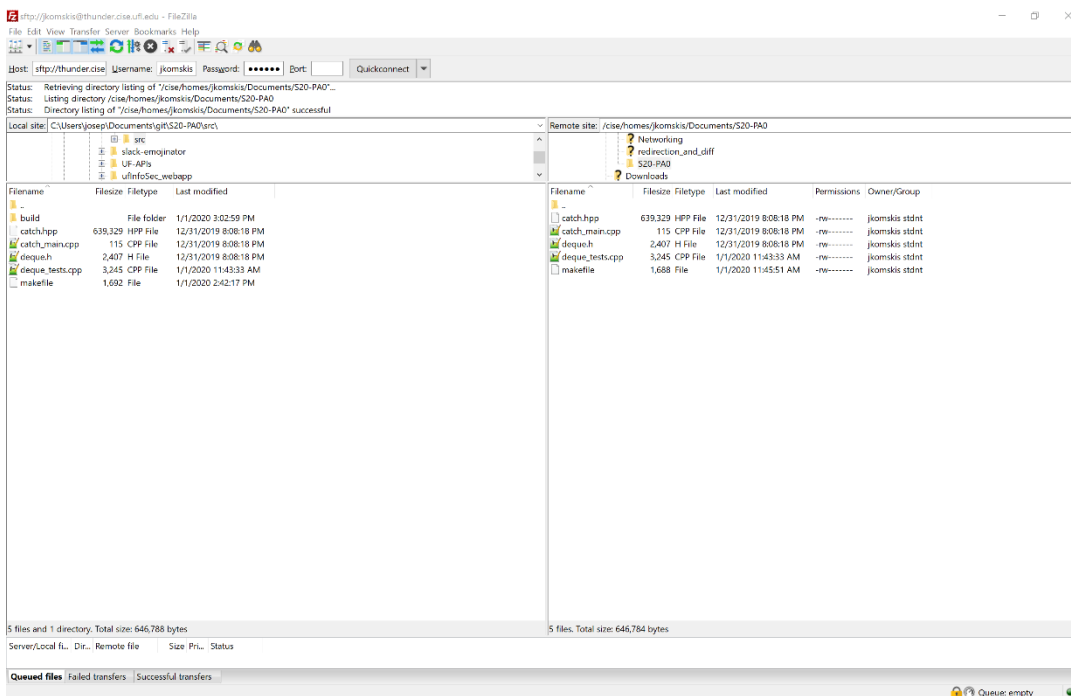
Once installed, open your SFTP client and configure the connection as follows:



Then, connect to the server. The left panel shows your local files and the right panel shows your user files on the server. In the left panel, browse to the folder containing your project folder, and drag and drop it to the right panel (make a note of where you transfer the file on the server).



This is what WinSCP should look like after transferring the files. `C:\Users\josep\Documents\git\S20-PA0\src` is my local project directory, and `/cise/homes/jkomskis/Documents/S20-PA0` is the remote directory where I copied my files.



This is what FileZilla should look like after transferring the files. `C:\Users\josep\Documents\git\S20-PA0\src` is my local project directory, and `/cise/homes/jkomskis/Documents/S20-PA0` is the remote directory where I copied my files.

Now that the project files are transferred, you will need to open a terminal session on the server. We will do this using SSH. Open a terminal as you would when compiling and run the following command to open an SSH session on the server: `ssh <you_cise_username>@thunder.cise.ufl.edu`. Enter your password when prompted. You now have a session opened on the server, and you can compile and test your project the

same way you would on your local machine: `cd` into the folder running your project files, run `make` to compile the project, then run `./build/test_deque` to run the tests. If you get different outputs on your local machine and the CISE servers and can't figure out why, you can ask a TA to help you resolve the issue.

Run `make` and `./build/test_deque` and take a screenshot of the output. For the screenshot you do not need to have all the tests passing.

Submission

Submit the following files on canvas:

- Screenshot of the output from the “Compiler Installation” section.
- Your completed deque.h file.
- Screenshot of the output from the “Compile & Run the Project Locally” section.
- Screenshot of the output from the “Compile & Run the Project Remotely” section.

When submitting, attach each file to your submission **separately**. **Do not** zip or tar the files.

FAQ

My program has undefined behavior, segmentation faults, memory leaks, etc. and I don't know why! How can I find the problem?

Fortunately, newer versions of G++ have built in [sanitizers](#) from Google to automatically find and report these kinds of problems. If you wish to use them, modify the makefile as follows:

Find the lines:

```
CPPFLAGS = -std=c++17 -MMD -MP -Wall -Wextra -pedantic -g# -fsanitize=address -fsanitize=memory -fsanitize=undefined -fno-omit-frame-pointer
LINKFLAGS = # -fsanitize=address -fsanitize=memory -fsanitize=undefined
```

Remove the '#' characters before `-fsanitize=address...`, the lines should then read:

```
CPPFLAGS = -std=c++17 -MMD -MP -Wall -Wextra -pedantic -g -fsanitize=address -fsanitize=memory -fsanitize=undefined -fno-omit-frame-pointer
LINKFLAGS = -fsanitize=address -fsanitize=memory -fsanitize=undefined
```

Lastly, recompile and rerun your program. If the sanitizers detect any issues, information about them will be printed to the terminal.