ECS 34: Programming Assignment #5

Instructor: Aaron Kaloti

Fall 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2:
  - Fixed a syntax error towards the end of the skeleton version of `graph.cpp` provided on Canvas.
  - For convenience, the contents of `graph1.txt` and `graph2.txt` are now displayed in the last code listing in this document.
  - Added a section on ranged-based `for` loops to the recommended reading.
- v.3:
  - Grading breakdown.
  - Autograder details.
  - Added lines in `graph.hpp` to delete the copy constructor and copy assignment constructor. You do not need to include these lines.
  - Modified `demo_graph.cpp` so that it does not needlessly use any copy constructors.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Wednesday, November 25. Gradescope will say 12:30 AM on Thursday, November 26, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

You should use the `-Wall`, `-Werror`, and `-std=c++11` flags when compiling. The autograder will use these flags when it compiles your code.

3 Grading Breakdown

This assignment will be out of 100 points. Below is the approximate worth of each part. For the constructor, the worth is referring to the input validation. It is hard to put an exact worth for each part because the test cases that fall under the last category call many of the methods.

- Constructor: 8
- `getNumVertices()`: 3
- `getAdjacencyMatrix()`: 10
- `getAdjacencyLists()`: 10
- `getBFSOrdering()`: 13
- `getDFSOrdering()`: 13
- `getTransitiveClosure()`: 13
- General usage of instances of the class and multiple methods: 30
  - Much of the cases for this will reuse parts from previous cases, so it indirectly increases the point value of those previous cases.

4 Submitting on Gradescope

You should only submit your C++ files with the names indicated below. You may be penalized for submitting additional files. You have infinite submissions until the deadline.

During the 10/02 lecture, I talked about how to change the active submission, just in case that is something that you find yourself wanting to do.

You will submit the following C++ files:

- `graph.hpp`
- `graph.cpp`

4.1 Regarding Autograder

Your output must match mine *exactly*. In certain situations (where explicitly specified), the autograder will take into account multiple possible outputs (e.g. multiple possible DFS orderings).

There is a description about how to interpret some of the autograder error messages in the directions for the first two programming assignments. I will not repeat that description here.

4.1.1 Memory Leaks

Probably at least one autograder test case will check for memory leaks. However, there should be no reason for you to explicitly[1] use dynamic memory allocation in this assignment, memory leaks should not be an issue at all, unless you forget to close a file.

4.1.2 Number of Visible vs. Hidden Cases

Below is the breakdown of visible vs. hidden test cases on each part.

[1] I say "explicitly" because standard library containers such as `std::vector` use dynamic memory allocation without you having to care.

Cases	No. Cases	No. Visible	No. Hidden
01 - 04	4	2	2
05 - 06	2	2	0
07 - 09	3	2	1
10 - 12	3	2	1
13 - 14	2	2	0
15 - 16	2	2	0
17 - 18	2	1	1
19 - 23	5	4	1

4.1.3 Autograder Code

See `test_visible.cpp` for the visible test cases' inputs. I will release `test_hidden.cpp`, which contains the hidden test cases' inputs, after the deadline. Below are some notes that may help you to understand `test_visible.cpp`:

- In C++, it would seem that for me to define variables within any part of the body of a `switch` statement, I must put curly braces around the body of the relevant `case` label, even if the variables do not have the same names as variables in other parts of the `switch` statement.
- There is some common code used by `test_visible.cpp` and `test_hidden.cpp`, so I included that common code in `test_utils.hpp` (declarations) and `test_utils.cpp` (definitions).
- For convenience, and to expose you to more features of C++ object-oriented programming, towards the beginning of `test_utils.cpp`, I define what are called "overloaded operators" that allows the autograder C++ code to print out complicated objects, such as the 2D vectors of booleans used to represent an adjacency matrix, with the same kind of syntax (i.e. using `std::cout <<`) that is used to print other variables.
- To expose you to namespaces, the helper functions in `test_utils.hpp` and `test_utils.cpp` are in the `namespace TestUtils`.

Compilation: You can compile `test_visible.cpp` in the manner shown below. Don't forget to pass the test case number as a command-line argument.

```
$ g++ -Wall -Werror -std=c++11 test_visible.cpp test_utils.cpp graph.cpp -o test_visible
$ ./test_visible 1
std::runtime_error exception caught with message: Invalid graph
No exception was thrown.
$
```

5 Prerequisite Concepts

Before beginning this assignment, you should acquire an understanding of the following C++ concepts. Below each concept, I make a recommendation as to how to go about studying the concept. Whenever I mention "the C++ book" or make any reference to a book, I am referring to *C++ Crash Course: A Fast-Paced Introduction* by Josh Lospinoso, one of the required textbooks for this course.

1. **Classes**: In chapter 2, read the sections "User-Defined Types" (just the subsection "Plain-Old Data Classes") and "Fully Featured C++ Classes".
2. **References**: In chapter 3, the section "References" on p.77 may help. You can think of references as being like pointers that cannot be set to null, cannot be made to reference something else, and do not need to be explicitly dereferenced. In C, we have a function take a pointer to an object as argument if we want the function to modify that object. In C++, we typically have the function take a reference as argument in this scenario. If you wanted to allow the possibility that a null pointer could be passed as argument, then you would need to have the function take a pointer instead, but otherwise, you should default to references.
3. **`const`, More on References, etc.**: In chapter 3, you should read the section "Usage of Pointers and References." It covers many useful topics, including the below.
   - **Const arguments**: The other benefit of taking a reference (or pointer) as argument is that the object referenced by the reference or pointer will not have to be copied. This especially matters when you are dealing with large objects (i.e. instances of classes that have large members and/or many members). There are scenarios in which we want the function to take a reference as argument (to avoid making a copy) without allowing the function to modify the referenced object; to do this, we would mark the argument as `const`, making the argument a "`const` reference".
   - **Const methods**: In all of the methods of `class UnweightedGraph` that you will implement, you may notice the use of `const` after the argument list. This makes the method `const`, meaning that the method cannot modify its members.

4. `auto` **keyword**: In chapter 3, you should read the section "auto Type Deduction".
5. **Exceptions**: In chapter 4, in the section "Exceptions", you should read the subsections from "The throw Keyword" through "Handling Exceptions". You can stop before "User-Defined Exceptions".
6. **Standard library containers**: `std::vector`, `std::stack`, and `std::queue`.
   - I talk about what you need to know about these STL containers below. I would probably not recommend the C++ book reading for these concepts for now, because of how technical the reading gets for these concepts, but see chapter 13 if you are curious.
   - You *might* find `std::unordered_set` useful too, but it is not necessary. (`std::unordered_set` and `std::unordered_map` are each implemented using a hash table.)
7. **Ranged-Based `for` Loops**: In chapter 8, you should read the section "Ranged-Based for loops". Hopefully, you should find these similar to the `for` loops in Python (when they did not use `range()`). Probably, reading the subsection titled "Usage" is enough. The "Range Expressions" section goes into how ranged-based `for` loops work behind the scenes using iterators, something that we will eventually talk about.
8. **File I/O**: You should familiarize yourself with how C++ does file I/O (the `<fstream>` header). You may find the "File Streams" subsection of chapter 16 to be helpful. Since C++ includes all of C for the most part, you can read files the same way in which you read them in C (e.g. with `FILE` pointers, `fgets()`, and/or `fscanf()`) if you want to, but if you are going to mention C++ on your résumé, it might raise eyebrows if you are not using the "modern" and typically preferable features that C++ added.

5.1 Vectors

The standard library `std::vector` type, which you can access by including `<vector>`, represents a dynamic array. By dynamic, I mean that it can grow and shrink its size as desired, just like a Python list and in contrast to the fixed-sized C arrays that you used in the previous assignments. You will eventually learn about how vectors are implemented, but for this assignment, it suffices to know how to use them. Below is an example.

```
$ cat try_vector.cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v;  // default constructor: empty vector
    v.push_back(8);
    v.push_back(17);
    v.push_back(12);
    std::cout << v[2] << '\n';  // array indexing works
    std::cout << v.at(1) << '\n';  // at() method throws exception if
                                   // given out-of-range index.
    std::cout << "Size: " << v.size() << '\n';
    std::vector<int> v2(5, 3);  // use one of the non-default constructors;
                                // vector of 5 ints, each 3.
    for (unsigned i = 0; i < v2.size(); ++i)  // iterate in C-style way
        std::cout << v2[i] << ' ';
    std::cout << '\n';
    for (int x : v2)  // iterate with foreach loop (not supported before C++11);
                      // isn't useful if we need the index.
        std::cout << x << ' ';
    std::cout << std::endl;
}
$ g++ -Wall -Werror -std=c++11 try_vector.cpp -o try_vector
$ ./try_vector
12
17
Size: 3
3 3 3 3 3
3 3 3 3 3
$
```

If we want a vector of characters, we use `std::vector<char>`. If we want a vector of floating-point numbers, we use `std::vector<float>`. In other words, the type goes in between the angle brackets. This makes use of a concept called **templates** (or, as your book more generally refers to the concept, **compile-time polymorphism**). You will learn more about this concept eventually. Below are the methods of `std::vector` that I most commonly use. You should look up the others and acquire a general familiarity with what capabilities are available to you.

- The constructor that lets you specify all of the elements.

- `push_back()`
- `resize()`
- `size()`

5.1.1 `std::vector<bool>`

Whenever an adjacency matrix is mentioned, I use `std::vector<bool>` to represent each row. Those of you with prior experience in C++ (or who have read ahead enough) may know that `std::vector<bool>` has a special interpretation as a dynamic bitset. It's perhaps[2] possible that `std::vector<char>` or `std::vector<int>` would be faster for each row in an adjacency matrix, even if it would waste space, but I went with `std::vector<bool>` since we are not too concerned with precise space vs. time tradeoffs right now.

5.2 Stacks

Below is an example in which I use the standard library `std::stack` that C++ provides.

```
$ cat try_stack.cpp
#include <iostream>
#include <stack>

int main()
{
    std::stack<int> s;
    s.push(8);
    s.push(15);
    s.push(19);
    std::cout << s.size() << '\n';
    std::cout << s.top() << '\n';
    s.pop();  // pop 19
    s.pop();  // pop 15
    std::cout << s.top() << std::endl;
    return 0;
}
$ g++ -Wall -Werror -std=c++11 try_stack.cpp -o try_stack
$ ./try_stack
3
19
8
$
```

The `std::stack` implementation prioritizes speed over convenience and safety, which is why[3]:

1. `pop()` and `top()` are separate methods[4].
2. `pop()` and `top()` exhibit undefined behavior if called while the stack is empty.

5.3 Queues

Below is an example in which I use the standard library `std::queue` that C++ provides.

```
$ cat try_queue.cpp
#include <iostream>
#include <queue>

int main()
{
    std::queue<int> q;
    q.push(37);
    q.push(65);
    q.push(49);
    q.push(15);
    std::cout << q.size() << '\n';
    std::cout << q.back() << '\n';
    std::cout << q.front() << '\n';
    q.pop();
    std::cout << q.front() << '\n';
```

[2] It may depend on the compiler.

[3] Reference: section 12.1 of *The C++ Standard Library* (Second Edition) by Nicolai M. Josuttis. In section 12.1.3, the author provides a stack implementation that prioritizes convenience and safety instead of speed.

[4] You could probably write a macro or macro function that essentially combines `pop()` and `top()` into one function call.

```
}
$ g++ -Wall -Werror -std=c++11 try_queue.cpp -o try_queue
$ ./try_queue
4
15
37
65
$
```

6 Your Task: `class UnweightedGraph`

In `graph.cpp`, implement all of the methods for `class UnweightedGraph` that are declared in `graph.hpp`. You should read all of the comments in `graph.hpp`, and you should start with the skeleton version of `graph.cpp` that is on Canvas. As mentioned above, you will submit `graph.hpp`, so you are allowed to modify it. However, **you are not allowed to modify the return value, name, argument types, number of arguments, or `const`-ness of any of the required methods** (the ones that are already on there). Below are reasons that you might modify `graph.hpp`:

- You may want to add at least one `private` member variable to the class.
- If you want to add helper methods, it is preferable that they be private, since there is no reason that code outside of `graph.cpp` should call those helper methods. Where possible, I personally prefer helper functions that are only declared/defined in `graph.cpp` (and thus should be `static` and do not appear at all in `graph.hpp`), but such helper functions have the drawback that they cannot access private members of the class `UnweightedGraph`.

Make sure to compile with C++11. With `g++`, this means using the `-std=c++11` flag, as shown in the example below. It looks like as of recently, `g++` seems to default to using C++11, at least on my Ubuntu 18.04 laptop. The autograder will use the `-std=c++11` flag.

If you wish, you could create typedefs for some of the long type names.

Below are examples of how your code should behave. You can find `demo_graph.cpp` on Canvas.

```
$ cat demo_graph.cpp
#include "graph.hpp"

#include <iostream>
#include <vector>

void printAdjMatrix(const std::vector<std::vector<bool>>& matrix)
{
    for (auto row : matrix)
    {
        for (bool val : row)
            std::cout << val << ' ';
        std::cout << '\n';
    }
}

void printGraphInfo(const UnweightedGraph& graph)
{
    std::cout << "Number of vertices: " << graph.getNumVertices() << '\n';
    std::cout << "Adjacency matrix:\n";
    auto adjMatrix = graph.getAdjacencyMatrix();
    printAdjMatrix(adjMatrix);
    std::cout << "Adjacency lists:\n";
    auto adjLists = graph.getAdjacencyLists();
    for (unsigned i = 0; i < adjLists.size(); ++i)
    {
        std::cout << "Vertex #" << i << ": ";
        for (int neighbor : adjLists[i])
            std::cout << neighbor << ' ';
        std::cout << '\n';
    }
    std::cout << "One BFS ordering (many are possible), starting at 0:\n";
    std::vector<int> ordering = graph.getBFSOrdering(0);
    for (int v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "Another BFS ordering, starting at 2:\n";
    ordering = graph.getBFSOrdering(2);
    for (int v : ordering)
```

```
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "One DFS ordering, starting at 3:\n";
    ordering = graph.getDFSOrdering(3);
    for (int v : ordering)
        std::cout << v << ' ';
    std::cout << '\n';
    std::cout << "Transitive closure:\n";
    auto tc = graph.getTransitiveClosure();
    printAdjMatrix(tc);
}

/**
 * Notice that argv is an array of character pointers, even in C++.
 * Although C++ has strings, because C++ is for the most part a superset of C,
 * C++ is stuck with many aspects of C that it cannot change, and this means
 * that you will from time to time encounter scenarios in which you must
 * deal with "C-style strings".
 */
int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        std::cerr << "No graph files provided.\n";
        return 1;
    }
    std::string filename = argv[1];
    UnweightedGraph graph(filename);
    std::cout << "=== " << filename << " ===\n";
    printGraphInfo(graph);
}
$ g++ -Wall -Werror -std=c++11 demo_graph.cpp graph.cpp -o demo_graph
$ cat graph_files/graph1.txt
4
0 1
1 0
1 2
1 3
3 0
$ ./demo_graph graph_files/graph1.txt
=== graph_files/graph1.txt ===
Number of vertices: 4
Adjacency matrix:
0 1 0 0
1 0 1 1
0 0 0 0
1 0 0 0
Adjacency lists:
Vertex #0: 1
Vertex #1: 0 2 3
Vertex #2:
Vertex #3: 0
One BFS ordering (many are possible), starting at 0:
0 1 2 3
Another BFS ordering, starting at 2:
2
One DFS ordering, starting at 3:
3 0 1 2
Transitive closure:
1 1 1 1
1 1 1 1
0 0 1 0
1 1 1 1
$ cat graph_files/graph2.txt
6
1 0
2 1
5 2
1 5
1 3
3 0
$ ./demo_graph graph_files/graph2.txt
=== graph_files/graph2.txt ===
Number of vertices: 6
```

```
Adjacency matrix:
0 0 0 0 0 0
1 0 0 1 0 1
0 1 0 0 0 0
1 0 0 0 0 0
0 0 0 0 0 0
0 0 1 0 0 0
Adjacency lists:
Vertex #0:
Vertex #1: 0 5 3
Vertex #2: 1
Vertex #3: 0
Vertex #4:
Vertex #5: 2
One BFS ordering (many are possible), starting at 0:
0
Another BFS ordering, starting at 2:
2 1 0 5 3
One DFS ordering, starting at 3:
3 0
Transitive closure:
1 0 0 0 0 0
1 1 1 1 0 1
1 1 1 1 0 1
1 0 0 1 0 0
0 0 0 0 1 0
1 1 1 1 0 1
$
```

UCDAVIS
COMPUTER SCIENCE