General Directed Weighted Graph

Contents

1 Change Log

- **2022-07-26**: Clarified `operator->` and printing empty graphs (see operator-> and printing empty graphs)
- **2022-07-18**: Fixed a typo in `insert_edge` (see this post for details)
- **2022-07-14**: Clarified gdwg.internal (see this post for details)
- **2022-07-11**: Initial Release

2 The Task

Write a `graph` library type in C++, in `include/gdwg/graph.hpp`.

In this assignment, you will write a *generic directed weighted graph* (GDWG) with value-semantics in C++. Both the data stored at a node and the weight stored at an edge will be parameterised types. The types may be different. For example, here is a graph with nodes storing `std::string` and edges weighted by `int`:

```
using graph = gdwg::graph<std::string, int>;
```

Formally, this directed weighted graph $G = (N, E)$ will consist of a set of nodes N and a set of weighted edges E.

All nodes are unique, that is to say, no two nodes will have the same value and shall not compare equal using `operator==`.

Given a node, an edge directed into it is called an *incoming edge* and an edge directed out of it is called an *outgoing edge*. The *in-degree* of a node is the number of its incoming edges. Similarly, the *out-degree* of a node is the number of its outgoing edges. Given a directed edge from `src` to `dst`, `src` is the *source node* and `dst` is known as the *destination node*.

Edges can be reflexive, that is to say, the source and destination nodes of an edge could be the same.

G is a multi-edged graph, as there may be two edges from the same source node to the same destination node with two different weights. Two edges from the same source node to the same destination node cannot have the same weight.

2.1 Definitions [gdwg.definitions]

1. Some words have special meaning in this document. This section precisicely defines those words.

    - *Preconditions*: the conditions that the function assumes to hold whenever it is called; violation of any preconditions results in undefined

    - *Effects*: the actions performed by the function.

    - *Postconditions*: the conditions (sometimes termed observable results) established by the function.

    - *Returns*: a description of the value(s) returned by the function.

    - *Throws*: any exceptions thrown by the function, and the conditions that would cause the exception.

    - *Complexity*: the time and/or space complexity of the function.

    - *Remarks*: additional semantic constraints on the function.

    - *Unspecified*: the implementation is allowed to make its own decisions regarding what is unspecified, provided that it still follows the explicitly specified wording.

2. An *Effects* element may specify semantics for a function `F` in code using the term *Equivalent to*. The semantics for `F` are interpreted as follows:

    - All of the above terminology applies to the provided code, whether or not it is explicitly specified. [*Example*: If `F` has a *Preconditions* element, but the code block doesn't explicitly check them, then it is implied that the preconditions have been checked. —*end example*]

    - If there is not a *Returns* element, and `F` has a non-`void` return type, all the return statements are in the code block.

    - *Throws*, *Postconditions*, and *Complexity* elements always have priority over the code block.

3. Specified complexity requirements are upper bounds, and implementations that provide better complexity guarantees meet the requirements.
4. The class synopsis is the minimum text your header requires to compile most tests (this doesn't mean that it will necessarily link or run as expected).
5. Blue text in code will link to C++ Reference or to another part of this document.

6. This section makes use of [stable.names]. A stable name is a short name for a (sub)section, and isn't supposed to change. We will use these to reference specific sections of the document. [*Example*:

> Student: Do we need to define `gdwg::graph<N, E>::operator!=`?
>
> Tutor: [other.notes] mentions that you don't need to so you can get used to C++20's generated operators.

—*end example*]

2.2 Constructors [gdwg.ctor]

It's very important your constructors work. If we can't validly construct your objects, we can't test any of your other functions.

`graph();`

1. *Effects*: Value initialises all members.
2. *Throws*: Nothing.

`graph(std::initializer_list<N> il);`

3. *Effects*: Equivalent to: `graph(il.begin(), il.end());`

```
template<typename InputIt>
graph(InputIt first, InputIt last);
```

4. *Preconditions*: Type `InputIt` models *Cpp17InputIterator* and is indirectly readable as type `N`.
5. *Effects*: Initialises the graph's node collection with the range `[first, last)`.

`graph(graph&& other) noexcept;`

6. *Postconditions*:

- `*this` is equal to the value `other` had before this constructor's invocation.
- `other.empty()` is `true`.
- All iterators pointing to elements owned by `*this` prior to this constructor's invocation are invalidated.
- All iterators pointing to elements owned by `other` prior to this constructor's invocation remain valid, but now point to the elements owned by `*this`.

`auto operator=(graph&& other) noexcept -> graph&;`

7. *Effects*: All existing nodes and edges are either move-assigned to, or are destroyed.
8. *Postconditions*:

- `*this` is equal to the value `other` had before this operator's invocation.
- `other.empty()` is `true`.
- All iterators pointing to elements owned by `*this` prior to this operator's invocation are invalidated.
- All iterators pointing to elements owned by `other` prior to this operator's invocation remain valid, but now point to the elements owned by `*this`.

9. *Returns*: `*this`.

`graph(graph const& other);`

10. *Postconditions*: `*this == other` is `true`.

`auto operator=(graph const& other) -> graph&;`

11. *Postconditions*:

    - `*this == other` is `true`.

    - All iterators pointing to elements owned by `*this` prior to this operator's invocation are invalidated.

12. *Returns*: `*this`.

2.3 Modifiers [gdwg.modifiers]

`auto insert_node(N const& value) -> bool;`

1. *Effects*: Adds a new node with value `value` to the graph if, and only if, there is no node equivalent to `value` already stored.
2. *Postconditions*: All iterators are invalidated.
3. *Returns*: `true` if the node is added to the graph and `false` otherwise.

`auto insert_edge(N const& src, N const& dst, E const& weight) -> bool;`

4. *Effects*: Adds a new edge representing `src` → `dst` with weight `weight`, if, and only if, there is no edge equivalent to `value_type{src, dst, weight}` already stored. [*Note*: Nodes are allowed to be connected to themselves. —*end note*]

5. *Postconditions*: All iterators are invalidated.
6. *Returns*: `true` if the edge is added to the graph and `false` otherwise.
7. *Throws*: `std::runtime_error("Cannot call gdwg::graph<N, E>::insert_edge when either src or dst node does not exist")` if either of `is_node(src)` or `is_node(dst)` are `false`. [*Note*: Unlike Assignment 2, the exception message must be used verbatim. —*end note*]

`auto replace_node(N const& old_data, N const& new_data) -> bool;`

8. *Effects*: Replaces the original data, `old_data`, stored at this particular node by the replacement data, `new_data`. Does nothing if `new_data` already exists as a node.
9. *Postconditions*: All iterators are invalidated.
10. *Returns*: `false` if a node that contains value `new_data` already exists and `true` otherwise.
11. *Throws*: `std::runtime_error("Cannot call gdwg::graph<N, E>::replace_node on a node that doesn't exist")` if `is_node(old_data)` is `false`. [*Note*: Unlike Assignment 2, the exception message must be used verbatim. —*end note*]

`auto merge_replace_node(N const& old_data, N const& new_data) -> void;`

12. *Effects*: The node equivalent to `old_data` in the graph are replaced with instances of `new_data`. After completing, every incoming and outgoing edge of `old_data` becomes an incoming/ougoing edge of `new_data`, except that duplicate edges shall be removed.
13. *Postconditions*: All iterators are invalidated.
14. *Throws*: `std::runtime_error("Cannot call gdwg::graph<N, E>::merge_replace_node on old or new data if they don't exist in the graph")` if either of `is_node(old_data)` or `is_node(new_data)` are `false`. [*Note*: Unlike Assignment 2, the exception message must be used verbatim. —*end note*]
15. [*Note*: The following examples use the format (N_{src}, N_{dst}, E). [*Example*: Basic example.

- Operation: `merge_replace_node(A, B)`
- Graph before: (A, B, 1), (A, C, 2), (A, D, 3)
- Graph after : (B, B, 1), (B, C, 2), (B, D, 3)

—*end example*][*Example*: Duplicate edge removed example.

- Operation: `merge_replace_node(A, B)`
- Graph before: (A, B, 1), (A, C, 2), (A, D, 3), (B, B, 1)
- Graph after : (B, B, 1), (B, C, 2), (B, D, 3)

—*end example*][*Example*: Diagrammatic example.

—*end example*] —*end note*]

`auto erase_node(N const& value) -> bool;`

16. *Effects*: Erases all nodes equivalent to `value`, including all incoming and outgoing edges.
17. *Returns*: `true` if `value` was removed; `false` otherwise.
18. *Postconditions*: All iterators are invalidated.

`auto erase_edge(N const& src, N const& dst, E const& weight) -> bool;`

20. *Effects*: Erases an edge representing `src` → `dst` with weight `weight`.
21. *Returns*: `true` if an edge was removed; `false` otherwise.
22. *Postconditions*: All iterators are invalidated.
23. *Throws*: `std::runtime_error("Cannot call gdwg::graph<N, E>::erase_edge on src or dst if they don't exist in the graph")` if either `is_node(src)` or `is_node(dst)` is `false`. [*Note*: Unlike Assignment 2, the exception message must be used verbatim. —*end note*]
24. *Complexity*: $O(\log(n) + e)$, where n is the total number of stored nodes and e is the total number of stored edges.

`auto erase_edge(iterator i) -> iterator;`

25. *Effects*: Erases the edge pointed to by `i`.
26. *Complexity*: Amortised constant time.
27. *Returns*: An iterator pointing to the element immediately after `i` prior to the element being erased. If no such element exists, returns `end()`.
28. *Postconditions*: All iterators are invalidated. [*Note*: The postcondition is slightly stricter than a real-world container to help make the assingment easier (i.e. we won't be testing any iterators post-erasure). —*end note*]

`auto erase_edge(iterator i, iterator s) -> iterator;`

29. *Effects*: Erases all edges between the iterators `[i, s)`.
30. *Complexity* $O(d)$, where d=`std::distance(i, s)`.
31. *Returns*: An iterator equivalent to `s` prior to the items iterated through being erased. If no such element exists, returns `end()`.

32. *Postconditions*: All iterators are invalidated. [*Note*: The postcondition is slightly stricter than a real-world container to help make the assingment easier (i.e. we won't be testing any iterators post-erasure). *—end note*]

`auto clear() noexcept -> void;`

33. *Effects*: Erases all nodes from the graph.
34. *Postconditions*: `empty()` is `true`.

2.4 Accessors [gdwg.accessors]

`[[nodiscard]] auto is_node(N const& value) -> bool;`

1. *Returns*: `true` if a node equivalent to `value` exists in the graph, and `false` otherwise.
2. *Complexity*: $O(\log(n))$ time.

`[[nodiscard]] auto empty() -> bool;`

3. *Returns*: `true` if there are no nodes in the graph, and `false` otherwise.

`[[nodiscard]] auto is_connected(N const& src, N const& dst) -> bool;`

4. *Returns*: `true` if an edge `src` → `dst` exists in the graph, and `false` otherwise.
5. *Throws*: `std::runtime_error("Cannot call gdwg::graph<N, E>::is_connected if src or dst node don't exist in the graph")` if either of `is_node(src)` or `is_node(dst)` are `false`. [*Note*: Unlike Assignment 2, the exception message must be used verbatim. *—end note*]

`[[nodiscard]] auto nodes() -> std::vector<N>;`

6. *Returns*: A sequence of all stored nodes, sorted in ascending order.
7. *Complexity*: $O(n)$, where n is the number of stored nodes.

`[[nodiscard]] auto weights(N const& src, N const& dst) -> std::vector<E>;`

8. *Returns*: A sequence of weights from `src` to `dst`, sorted in ascending order.
9. *Complexity*: $O(\log(n) + e)$, where n is the number of stored nodes and e is the number of stored edges.

10. *Throws*: `std::runtime_error("Cannot call gdwg::graph<N, E>::weights if src or dst node don't exist in the graph")` if either of `is_node(src)` or `is_node(dst)` are `false`. [*Note*: Unlike Assignment 2, the exception message must be used verbatim. —*end note*]

`[[nodiscard]] auto find(N const& src, N const& dst, E const& weight) -> iterator;`

11. *Returns*: An iterator pointing to an edge equivalent to `value_type{src, dst, weight}`, or `end()` if no such edge exists.
12. *Complexity*: $O(\log(n) + \log(e))$, where n is the number of stored nodes and e is the number of stored edges.

`[[nodiscard]] auto connections(N const& src) -> std::vector<N>;`

13. *Returns*: A sequence of nodes (found from any immediate outgoing edge) connected to `src`, sorted in ascending order, with respect to the connected nodes.
14. *Complexity*: $O(\log(n) + e)$, where e is the number of outgoing edges associated with `src`.
15. *Throws*: `std::runtime_error("Cannot call gdwg::graph<N, E>::connections if src doesn't exist in the graph")` if `is_node(src)` is `false`. [*Note*: Unlike Assignment 2, the exception message must be used verbatim. —*end note*]

2.5 Iterator access [gdwg.iterator.access]

`[[nodiscard]] auto begin() const -> iterator;`

1. *Returns*: An iterator pointing to the first element in the container.

`[[nodiscard]] auto end() const -> iterator;`

2. *Returns*: An iterator denoting the end of the iterable list that `begin()` points to.
3. *Remarks*: `[begin(), end())` shall denote a valid iterable list.

2.6 Comparisons [gdwg.cmp]

`[[nodiscard]] auto operator==(graph const& other) -> bool;`

1. *Returns*: `true` if `*this` and `other` contain exactly the same nodes and edges, and `false` otherwise.
2. *Complexity*: $O(n + e)$ where n is the sum of stored nodes in `*this` and `other`, and e is the sum of stored edges in `*this` and `other`.

2.7 Extractor [gdwg.io]

`friend auto operator<<(std::ostream& os, graph const& g) -> std::ostream&;`

1. *Effects*: Behaves as a formatted output function of `os`.
2. *Returns*: `os`.
3. *Remarks*: The format is specified thusly:

```
[source_node$_1$] [edges$_1$]
[source_node$_2$] [edges$_2$]
...
[source_node$_n$] [edges$_n$]
```

[source_node$_1$], ..., [source_node$_n$] are placeholders for all nodes that the graph stores, sorted in ascending order. [edges$_1$], ..., [edges$_n$] are placeholders for

```
(
  [node$_n$_connected_node$_1$] | [weight]
  [node$_n$_connected_node$_2$] | [weight]
  ...
  [node$_n$_connected_node$_n$] | [weight]
)
```

where [node$_n$_conencted_node$_1$] | [weight], ..., [node$_n$_connected_node$_n$] | [weight] are placeholders for each node's connections and corresponding weight, also sorted in ascending order. [*Note*: If a node doesn't have any connections, then its corresponding [edges$_n$] should be a line-separated pair of parentheses —*end note*]

[*Example*:

```
using graph = gdwg::graph<int, int>;
auto const v = std::vector<graph::value_type>{
  {4, 1, -4},
  {3, 2, 2},
  {2, 4, 2},
  {2, 1, 1},
  {6, 2, 5},
  {6, 3, 10},
  {1, 5, -1},
  {3, 6, -8},
  {4, 5, 3},
  {5, 2, 7},
};

auto g = graph{};
for (const auto& [from, to, weight] : v) {
  g.insert_node(from);
  g.insert_node(to);
  g.insert_edge(from, to, weight)
}

g.insert_node(64);
auto out = std::ostringstream{};
out << g;
auto const expected_output = std::string_view(R"(1 (
  5 | -1
)
2 (
  1 | 1
  4 | 2
)
3 (
  2 | 2
  6 | -8
)
4 (
  1 | -4
  5 | 3
)
5 (
  2 | 7
)
6 (
  2 | 5
  3 | 10
)
64 (
)
)");
CHECK(out.str() == expected_output);
```

—*end example*]

Note: The empty graph should print an empty string. i.e.

```
auto g = graph<int, int>();

auto oss = std::ostringstream{};

oss << g;
```

```
CHECK(oss.str().empty());
```

2.8 Iterator [gdwg.iterator]

```
template<typename N, typename E>
class graph<N, E>::iterator {
public:
  using value_type = graph<N, E>::value_type;
  using reference = value_type;
  using pointer = void;
  using difference_type = std::ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;

  // Iterator constructor
  iterator() = default;

  // Iterator source
  auto operator*() -> reference;
  // auto operator->() -> pointer not required

  // Iterator traversal
  auto operator++() -> iterator&;
  auto operator++(int) -> iterator;
  auto operator--() -> iterator&;
  auto operator--(int) -> iterator;

  // Iterator comparison
  auto operator==(iterator const& other) -> bool;
private:
  explicit iterator(unspecified);
};
```

1. Elements are lexicographically ordered by their source node, destination node, and edge weight, in ascending order.
2. Nodes without any connections are not traversed.
3. [*Note*: `gdwg::graph<N, E>::iterator` models `std::bidirectional_iterator`. —*end note*]

2.8.1 Iterator constructor [gdwg.iterator.ctor]

`iterator();`

1. *Effects*: Value-initialises all members.
2. *Remarks*: Pursuant to the requirements of `std::forward_iterator`, two value-initialised iterators shall compare equal.

`explicit iterator(unspecified);`

3. *Effects*: Constructs an iterator to a specific element in the graph.
4. *Remarks*: There may be multiple constructors with a non-zero number of parameters.

2.8.2 Iterator source [gdwg.iterator.source]

`auto operator*() -> reference;`

1. *Effects*: Returns the current `from`, `to`, and `weight`.

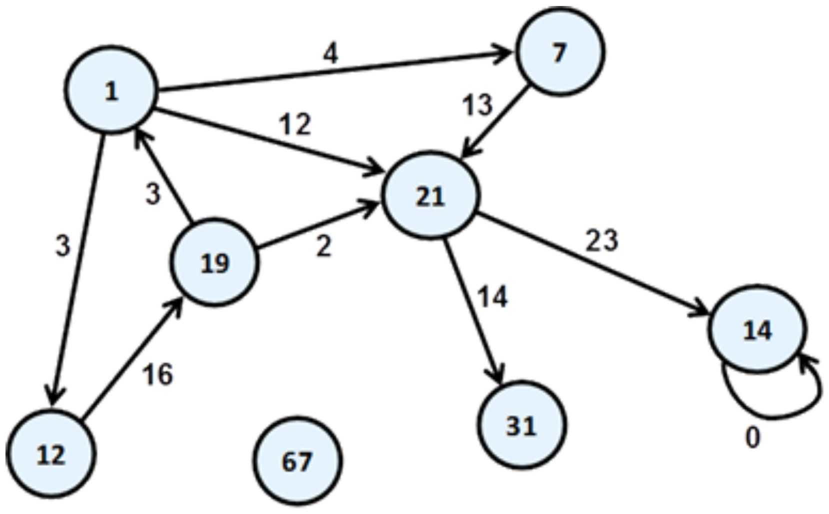
2.8.3 Iterator traversal [gdwg.iterator.traversal]

`auto operator++() -> iterator&;`

1. *Effects*: Advances `*this` to the next element in the iterable list.

[*Example*: In this way, your iterator will iterator through a graph like the one below producing the following tuple values when deferenced each time:

`gdwg::graph<int, int>`



```
(1, 7, 4)
(1, 12, 3)
(1, 21, 12)
(7, 21, 13)
(12, 19, 16)
(14, 14, 0)
(19, 1, 3)
(19, 21, 2)
(21, 14, 23)
(21, 31, 14)
```

—*end example*]

2. *Returns*: `*this`.

```
auto operator++(int) -> iterator;
```

3. *Effects*: Equivalent to:

```
auto temp = *this;
++*this;
return temp;
```

```
auto operator--() -> iterator&;
```

4. *Effects*: Advances `*this` to the previous element in the iterable list.
5. *Returns*: `*this`.

```
auto operator--(int) -> iterator;
```

6. *Effects*: Equivalent to:

```
auto temp = *this;
--*this;
return temp;
```

2.8.4 Iterator comparison [gdwg.iterator.comparison]

```
auto operator==(iterator const& other) -> bool;
```

1. *Returns*: `true` if `*this` and `other` are pointing to the same elements in the same iterable list, and `false` otherwise.

2.9 Compulsory internal representation [gdwg.internal]

Your graph is **required** to own the resources (nodes and edge weights) that are passed in through the insert functions. This means creating memory on the heap and doing a proper copy of the values from the caller. This is because resources in your graph should outlive the caller's resouce that was passed in in case it goes out of scope. For example, we want the following code to be valid.

```
auto main() -> int {
  gdwg::graph<std::string, int> g;
  {
    std::string s1{"Hello"};
    g.insert_node(s1);
  }

  // Even though s1 has gone out of scope, g has its own
  //  copied resource that it has stored, so the node
  //  will still be in here.
  std::cout << g.is_node("Hello") << "\n"; // prints 'true';
}
```

Your graph is **required** to use smart pointers (*however* you please) to solve this problem.

1. For each node, you are only allowed to have one underlying resource (heap) stored in your graph for it. This means every `N` can only be stored once per graph instance.
2. For each edge, you should avoid using unnecessary additional memory wherever possible.

- [*Hint*: In your own implementation you're likely to use some containers to store things, and depending on your implementation choice, somewhere in those containers you'll likely use either `std::unique_ptr<N>` or `std::shared_ptr<N>` —*end hint*]

2.9.1 But why smart pointers [gdwg.internal.rationale]

You could feasibly implement the assignment without any smart pointers, through lots of redundant copying. For example, having a massive data structure like:

```
std::map<N, std::vector<std::pair<N, E>>>
```

You can see that in this structure you would have duplicates of nodes when trying to represent this complex structure. This takes up a lot of space. We want you to build a space efficient graph.

2.10 Other notes [other.notes]

You must:

- Include a header guard in `include/gdwg/graph.hpp`
- Use C++20 style and methods where appropriate
- Make sure that *all appropriate member functions* are `const`-qualified
- Leave a moved-from object in a state with no nodes.

- Implement this class within the namespace `gdwg`.
- Assignment 2 asked you to implement `operator!=` because you'll see it in a lot of production codebases, and it's important that you know how to write it correctly. To get used to how C++20 handles `operator!=`, we're asking that you **do not** implement an overload for `operator!=` in Assignment 3.

You must not:

- Write to any files that aren't provided in the repo (e.g. storing your vector data in an auxilliary file)
- Add additional members to the **public** interface.

You:

- Should try to mark member functions that will not throw exceptions with `noexcept`
- Are not required to make any member function explicit unless directly asked to in the spec.

2.10.1 `const`-correctness [const.correctness]

We have deliberately removed the `const`-qualifiers for most member functions from the specification. **You are required to work out which functions must be `const`-qualified.** You must ensure that each operator and member function appropriately either has:

- A `const` member function; or
- A non-`const` member function; or
- Both a `const` and non-const member function

Please think carefully about this. The function declarations intentionally do not specify their constness, except for the `begin()` and `end()` member functions. These are `const`-qualified to help you out.

In most cases you will only need a single function in the overload set.

2.10.2 Member types [gdwg.types]

For convenience's sake, inside the `graph` class you will likely have a member type `value_type`, defined like so:

```
class graph {
public:
  struct value_type {
    N from;
    N to;
    E weight;
  };
  // ...
};
```

As noted in the compulsory internal representation section, you are unlikely to want to use this directly within your representation. However, it is used by the `iterator` type, and is good practice to have for a container.

3 Getting Started

If you haven't done so already, clone this repository.

```
$ git clone gitlab@gitlab.cse.unsw.edu.au:COMP6771/22T2/students/z5555555/ass3.git
```

(Note: Replace z5555555 with your zid)

Navigate inside the directory. You can then open vscode with `code .` (note the dot).

3.1 Running your tests

Similar to the first tutorial, you simply to have to run `Ctrl+Shift+P` and then type `Run Test` and hit enter. VS Code will compile and run all of your tests and produce an output.

3.2 Adding more tests

Part of your assignment mark will come from the quality and extensiveness of tests that you write.

You can add more test files to the `test/graph/` directory. Simply copy `test/graph/graph_test1.cpp` into another file in that directory.

Note, everytime that you add a new file to the `test/graph/` directory you will need to add another few lines to `test/CMakeLists.txt`. You can once again, simply copy the test reference for `graph_test1.cpp` and rename the appropriate parts. Every time you update `CMakeLists.txt` in any repository, in VSCode you should codess `Ctrl+Shift+P` and run `Reload Window` for the changes to take effect.

4 Marking Criteria

This assignment will contribute 30% to your final mark.

The assessment for the assignment recognizes the difficulty of the task, the importance of style, and the importance of appropriate use of programming methods (e.g. using while loops instead of a dozen if statements).

50%	**Correctness** The correctness of your program will be determined automatically by tests that we will run against your program. You will not know the full sample of tests used prior to marking.
25%	**Your tests** You are required to write your own tests to ensure your program works. You will write tests in the `test/` directory. At the top of **ONE** file you will also include a block comment to explain the rationale and approach you took to writing tests. Please read the Catch2 tutorial or review lecture/tutorial content to see how to write tests. Tests will be marked on several factors. These include, but are not limited to: • Correctness — an incorrect test is worse than useless. • Coverage - your tests might be great, but if they don't cover the part that ends up failing, they weren't much good to you. • Brittleness — If you change your implementation, will the tests need to be changed (this generally means avoiding calling functions specific to your implementation where possible - ones that would be private if you were doing OOP). • Clarity — If your test case failed, it should be immediately obvious what went wrong (this means splitting it up into appropriately sized sub-tests, amongst other things).
20%	**C++ best practices** Your adherence to good C++ best practice in lecture. This is **not** saying that if you conform to the style guide you will receive full marks for this section. This 20% is also based on how well you use modern C++ methodologies taught in this course as opposed to using backwards-compatible C methods. Examples include: Not using primitive arrays and not using pointers. We will also penalise you for standard poor practices in programming, such as having too many nested loops, poor variable naming, etc.
5%	**clang-format** In your project folder, run the following commands on all cpp/h files in the \`source\` and \`test\` directory. `$ clang-format-11 -style=file -i /path/to/file.cpp` If, for each of these files, the program outputs nothing (i.e. is linted correctly), you will receive full marks for this section (5%). A video explaining how to use clang-format can be found HERE.

5 Originality of Work

The work you submit must be your own work. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted.

The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such submissions.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct.

Do not provide or show your assignment work to any other person — apart from the teaching staff of COMP6771.

If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

Note you will not be penalized if your work has the potential to be taken without your consent or knowledge.

The following actions will result in a 0/100 mark for this assignment, and in some cases a 0 for COMP6771:

- Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
- Submitting any other person's work. This includes joint work.

The lecturer may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

PLEASE NOTE: We have a record of ALL previous submissions of this assignment submitted. If you find a solution from a friend, or online, we will find it and you will receive 0 for the assignment and potentially 0 for the course. Trust me, at least 1 person does it every term and I encourage you not to think you'll get lucky.

6 Submission

This assignment is due *Monday 1st of August at 19:59:59 (Week 10)*.

Our systems automatically record the most recent push you make to your master branch. Therefore, to "submit" your code you simply need to make sure that your master branch (on the gitlab website) is the code that you want marked for this task.

It is your responsibiltiy to ensure that your code can be successfully demonstrated on the CSE machines (e.g. vlab)
from a fresh clone of your repository. Failure to ensure this may result in a loss of marks.

7 Late Submission Policy

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 0.2% up to 120 hours late, after which it will receive 0.

For example if an assignment you submitted with a raw awarded mark of 90% was submitted 5 hours late, the late submission would have no effect (as maximum mark would be 99%).

If the same assignment was submitted 72 hours late it would be awarded 85%, the maximum mark it can achieve at that time.

This late penalty has been amended from the original specification, and you should not assume it will be the same for future assignments.