COMP6771 19T2 (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

Week 02 Tutorial Sample Answers

Advanced C++ Programming (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

1. Are you having any issues with the assignment? Any questions you want to ask your tutor?

2. What is the difference between each kind of iterator?
   - Input iterator
   - Output iterator
   - Forward iterator
   - Bidirectional iterator
   - Random-access iterator

   - An input iterator allows read and increment. However, it cannot make multiple passes when incrementing
   - An output iterator allows write and increment. However, it cannot make multiple passes when incrementing
   - A forward iterator allows read, write and increment
   - A bidirectional iterator allows read, write, increment, and decrement
   - A random-access iterator allows read, write, increment, decrement, +x, and -x

   Multiple passes means you can read an old value of an iterator (by copying the iterator and advancing a copy, for example).

3. We write some code (marks.cpp) to store information in a container and then access the middle element (to determine the median). What's wrong with this code? How could we modify this code to produce the intended result?

```
#include <iostream>
#include <list>

int main() {
  std::list studentMarks;
  studentMarks.push_back(63);
  studentMarks.push_back(67);
  studentMarks.push_back(69);
  studentMarks.push_back(74);
  studentMarks.push_back(82);

  std::cout << "Median: " << studentMarks[2] << "\n";
}
```

```
#include <iostream>
#include <list>
#include <vector>
#include <iterator>

int main() {
  // Manually iterating through a list (forward iteration)
  std::list<int> studentMarks1;
  studentMarks1.push_back(63);
  studentMarks1.push_back(67);
  studentMarks1.push_back(69);
  studentMarks1.push_back(74);
  studentMarks1.push_back(82);

  int medianIndex = (studentMarks1.size() / 2);
  auto it = studentMarks1.begin();
  std::advance(it, medianIndex);
  std::cout << "Median: " << *it << "\n";

  // Using a container (vector) with random access iterator
  std::vector<int> studentMarks2;
  studentMarks2.push_back(63);
  studentMarks2.push_back(67);
  studentMarks2.push_back(69);
  studentMarks2.push_back(74);
  studentMarks2.push_back(82);

  std::cout << "Median: " << studentMarks2.at(2) << "\n";
}
```

4. What is different about a const iterator compared to a non-const iterator?

A const iterator does not allow modification of the value that the iterator refers to.

5. Here is a C-style and C++-style method of forward iterating through an array. In each of these cases, how would we modify the code (rev.cpp) to iterate in the other direction? Why is the C++ way preferred?

```
#include <iostream>
#include <vector>

int main() {

  std::vector<int> temperatures{32, 34, 33, 28, 35, 28, 34};

  for (int i = 0; i < temperatures.size(); ++i) {
    std::cout << temperatures.at(i) << " ";
  }
  std::cout << "\n";

  for (const auto& temp : temperatures) {
    std::cout << temp << " ";
  }
  std::cout << "\n";

  for (auto iter = temperatures.cbegin();
       iter != temperatures.cend(); ++iter) {
    std::cout << *iter << " ";
  }
  std::cout << "\n";

}
```

```
#include <iostream>
#include <vector>

int main() {

  std::vector<int> temperatures{32, 34, 33, 28, 35, 28, 34};

  for (int i = temperatures.size() - 1; i >= 0; --i) {
    std::cout << temperatures.at(i) << " ";
  }
  std::cout << "\n";

  // Can't do it with for-range

  for (auto iter = temperatures.crbegin();
       iter != temperatures.crend(); ++iter) {
    std::cout << *iter << " ";
  }
  std::cout << "\n";

}
```

6. What kind of iterator is each of the following (and are the iterators const)?
   Which of these will compile, and which of these will not?
   Note: std::list is a doubly linked list, and std::forward_list is a singly linked list.

```
const std::vector<int> vec;
std::list<int> li;
std::forward_list<double> forward_li;
std::string s;

vec.begin();
vec.cbegin();
(*vec.begin())++;
li.cbegin();
li.rbegin();
forward_li.cbegin();
(*forward_li.cbegin())++;
forward_li.crbegin();
s.begin();
std::back_inserter(vec);
std::back_inserter(li);
std::istream_iterator<int>(std::cin);
std::ostream_iterator<int>(std::cout, " ");
```

```
const std::vector<int> vec;
std::list<int> li;
std::forward_list<double> forward_li;
std::string s;

vec.begin();  // const random access iterator
vec.cbegin();  // const random access iterator
(*vec.begin())++;  // Fails to compile (vector is const, attempt to access non-const iterator)
li.cbegin();  // const bidirectional iterator
li.rbegin();  // non-const bidirectional iterator
forward_li.cbegin();  // const forward iterator
(*forward_li.begin())++;  // forward iterator
(*forward_li.cbegin())++;  // fails to compile (attempt to modify non-const iterator)
forward_li.crbegin();  // Fails to compile (cannot get reverse iterator for forward iterators)
s.begin();  // non-const random access iterator
std::back_inserter(vec);  // Compiles. Can't invoke the dereference operator to insert anything be
std::back_inserter(li);  // Output iterator
std::istream_iterator<int>(std::cin);  // Input iterator
std::ostream_iterator<int>(std::cout, " ");  // Output iterator
```

7. Below are two functions minInt and minDouble. Use function overloading to improve the style of this code (overload.cpp)

```
int minInt(int a, int b) {
  return a < b ? a : b;
}

double minDouble(double a, double b) {
  return a < b ? a : b;
}
```

```
int min(int a, int b) {
  return a < b ? a : b;
}

double min(double a, double b) {
  return a < b ? a : b;
}
```

8. Using the above example, how could we further improve the clarity of the code with function templates?

```
template <typename T>
T min(T a, T b) {
  return a < b ? a : b;
}
```

9. Do you understand every part of this map code from lectures?

```
#include <iostream>
#include <map>
#include <string>

int main() {
  std::map<std::string, double> m;
  // The insert function takes in a key-value pair.
  std::pair<std::string, double> p1{"bat", 14.75};
  m.insert(p1);
  // The compiler will automatically construct values as
  // required when it knows the required type.
  m.insert({"cat", 10.157});
  // This is the preferred way of using a map
  m.emplace("cat", 10.157);

  // This is very dangerous, and one of the most common causes of mistakes in C++.
  std::cout << m["bat"] << '\n';

  auto it = m.find("bat"); // Iterator to bat if present, otherwise m.end()

  // This is a great example of when to use auto, but we want to show you what type it is.
  for (const std::pair<const std::string, double>& kv : m) {
    std::cout << kv.first << ' ' << kv.second << '\n';
  }
}
```

10. What is the relationship between containers, iterators, and algorithms?
    Why is this relationship so important, and how does it help us as programmers
    How does this relate to the DRY (don't repeat yourself) principle?

    Each container defines an iterator. Iterators act as an API which is used by algorithms. The algorithms, despite knowing nothing about the containers, are able to access them through zero-overhead abstractions through iterators.
    The relationship is very important, since it allows us to write generic code that doesn't need to be rewritten and retested for each different type. It also means that we can swap out types without the need to make many changes.
    It relates to the DRY principle in that we need to write m Containers + n algorithms, instead of m containers * n algorithms.