Homework 01 Solutions

ECE 449/590, Fall 2022

1. *(20 points)* (Exercise 1-1 and 1-2)

   A. Are the following definitions valid? Why or why not?

   ```
   const std::string hello = "Hello";
   const std::string message = hello + ", world" + "!";
   ```

   B. Are the following definitions valid? Why or why not?

   ```
   const std::string exclam = "!";
   const std::string message = "Hello" + ", world" + exclam;
   ```

   Answer:

   Note that the operator `+` is left-associative.

   A. The first statement is obviously valid. The second statement is evaluated as

   ```
   const std::string message = (hello + ", world") + "!";
   ```

   The expression `hello + ", world"` is valid since `+` is overloaded to take two operands, one of type `std::string` (`hello`) and the other as a C-style string (`", world"`). Moreover, this expression return an object of type `std::string`. Therefore, `(hello + ", world") + "!"` is also valid and the second statement is valid.

   B. The first statement is obviously valid. The second statement is evaluated as

   ```
   const std::string message = ("Hello" + ", world") + exclam;
   ```

   The operator `+` is not overloaded with two operands both being C-style strings (`"Hello"` and `", world"`). Therefore, the expression `"Hello" + ", world"` is not valid and the second statement is not valid.

2. *(20 points)* The assignment operator `=` works with two operands `L` and `R` in the form `L=R`. For the following code to generate an output of `3 3 3 3`, what should be the associativity of `=` and what should be the result and the side effects of `L=R`?

```
int a(0), b(1), c(2), d(3);
a=b=c=d;
std::cout << a << " " << b << " " << c << " " << d << std::endl;
```

Answer:

The operator `=` is right-associative. The second statement is evaluated as

```
a=(b=(c=d));
```

For the expression `L=R`, the side effect is to assign value of `R` to `L` and the result can be an object that has the same value as `R` (later on we will learn that the result is a reference to `L`).

3. *(20 points)* (Exercise 4-8)

   If the following code is legal, what can we infer about the return type of `f`?

   ```
   double d = f()[n];
   ```

   Answer:

   The expression `f()[n]` is evaluated as `(f())[n]`, i.e. the operator `[]` is applied to what the function call `f()` returns.

   Therefore, the function `f` should return an object or a reference to an object where the operator `[]` is defined (either built-in like a C-aray or overloaded like `std::vector`). The result of the operator `[]` should be of the type `double` or some type that can be converted to `double`, like `int`.

4. *(20 points)* (Exercise 6-3 and 6-4, also see Chapter 6.1)

   A. The following program attempts to copy from `u` into `v`. What's wrong?

   ```
   std::vector<int> u(10, 100);
   std::vector<int> v;
   std::copy(u.begin(), u.end(), v.begin());
   ```

   B. Correct the above program. There are at least two possible ways to correct the program but you are only required to implement one.

   Answer:

   A. For `std::copy`, it's programmer's responsibility to ensure that the destinations are ready to accept the copied values. However, since `v` is empty, an iterator of it refers to nothing and won't be able to accept any value.

B. The first way is to ensure that the iterators refer to elements that do exist and thus can accept copied values. This can be achieved by ensuring `v` to have at least `u.size()` elements, e.g.,

```
std::vector<int> u(10, 100);
std::vector<int> v;
v.resize(u.size());
std::copy(u.begin(), u.end(), v.begin());
```

or you can choose to create those elements when constructing `v`,

```
std::vector<int> u(10, 100);
std::vector<int> v(u.size());
std::copy(u.begin(), u.end(), v.begin());
```

The second way is to use an iterator adaptor, which essentially creates an element at the destination before the value is copied, e.g.,

```
std::vector<int> u(10, 100);
std::vector<int> v;
std::copy(u.begin(), u.end(), std::back_inserter(v));
```

5. *(20 points)* Suppose `integers` is a container with `int` elements. Implement a function to sort `integers` from the largest to the smallest. (Hint: use `std::sort`.)

Answer:

`std::sort` may take a third parameter for the comparison between elements. The default semantics of the comparison is less-than (`<`) so if we pass in greater-than (`>`), numbers will be sorted from largest to smallest. You may also use `std::greater<int>` instead of the lambda function.

```
void sort_reversed(std::vector<int> &integers)
{
    return std::sort(integers.begin(), integers.end(),
        [](int x, int y) {return x > y;});
}
```