Lecture review: Generic Programming (templates)

In C++, a template is a simple and yet very powerful tool. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types.

Function templates

Let say we want to implement `abs()` function for all data types. One way is to use function overloading, but it is not a good solution because we need to write many overload functions for each data type like **`int`**, **`float`**, **`double`**, **`long`**, etc.

```
int abs(int x) {
    return (x >= 0) ? x : -x;
}
float abs(float x) {
    return (x >= 0) ? x : -x;
}
...
```

The problem with the above approach is that 1) we need to write the same code for different data types, and 2) it won't work for new user-defined data types. The generic programming provides a way to write a single *function template* that can be used by compiler to automatically generate the required function for a specific data type. A simple implementation of `abs()` function template is given below.

```
template <typename T>
T abs(T x) {
    return (x >= 0) ? x : -x;
}
```

To define a function template, the keyword **`template`** is used, followed by the template parameter list inside `< >` which is a comma-separated list of parameters. The keyword **`typename`** is used to define type parameters. The function template `abs()` can be used for any data type. For example,

```
int main() {
    int a = -5;
    float b = -5.5;
    cout << abs(a) << endl;  // 5
    cout << abs(b) << endl;  // 5.5
    return 0;
}
```

When calling template function, the compiler will automatically generate the required function from the template. This process is called *template instantiation*.
In the above example, since the variable `a` has type **`int`**, the compiler will generate a function **`int`** `abs(`**`int`** `x)` from the template. Similarly, the compiler will generate a function **`float`** `abs(`**`float`** `x)` from the template for the variable `b`.

Class templates

Like function templates, *class templates* are a useful way to define a class that can work with any data type. For example, we can define a `Vector3D` class template that can use any data type for its elements.

```
template <typename T>
class Vector3D {
    T x, y, z;
  public:
    Vector3D(T x, T y, T z) : x(x), y(y), z(z) {}
    T magnitude() {
        return sqrt(x*x + y*y + z*z);
    }
};
```

The class template `Vector3D` can be used for any data type. For example,

```
int main() {
    Vector3D<int> v1(1, 2, 3);
    Vector3D<float> v2(1.5, 2.5, 3.5);
    cout << v1.magnitude() << endl;  // 3.74166
    cout << v2.magnitude() << endl;  // 4.30116
    return 0;
}
```

The compiler will generate two classes from the template: `Vector3D<int>` and `Vector3D<float>`.

Measure execution time of a function

To measure the execution time of a function, we can use the following code snippet.

```
#include <chrono>
using namespace std::chrono;
// note the timepoint before the function call
auto start = high_resolution_clock::now();
func(); // the function call of which we want to measure the execution time
auto stop = high_resolution_clock::now(); // time after the function call

// Subtract stop and start timepoints and cast it to required unit.
auto duration = duration_cast<microseconds>(stop - start);

cout << duration.count() << endl; // print the duration
```

In the above code, we have used **`auto`** keyword to avoid typing long type definitions. The `<chrono>` library provides the function `now` to get the timepoint at this instant. The `high_resolution_clock` is a clock that provides the smallest possible tick period.
The `duration_cast` function is used to convert the difference between two timepoints to a specific unit of time. Predefined units are `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes`, and `hours`. To get the value of duration, we use the `count()` member function on the duration object.

Lab Exercises

Exercise 1

Develop a template function `min3` that takes three arguments of the same type and returns the least of these arguments. For example, `min3(1,0,2)` would return `0` and `min4(1.5, 3, 0.5)` would return `0.5`.

Exercise 2

(a) Develop a template function `sum` that computes the sum of zero or more elements (of the same type) that are stored contiguously in memory. The template should have a single parameter that is the type of the elements to be processed by the function. The function has two parameters: 1) a pointer to the first element in the range to be summed, and 2) a pointer to one-past-the-last element in the range to be summed. The function should return the sum of the elements in the range.

(b) Write a program to test the `sum` function.

(c) Also, test the `sum` function with the code below.

```
int main() {
    int iarr[] = {1, 2, 3, 4, 5};
    double darr[] = {1.2, 2.3, 4.9, 5.1};
    cout << sum(iarr, iarr+5) << endl;  // 15
    cout << sum(darr, darr+4) << endl;  // 13.5
    return 0;
}
```

Exercise 3

Develop a template class `Complex` that represents a complex number and is parameterized on the type `T` used to represent the real and imaginary parts of the complex number. So,for example, `Complex<float>` would be a complex number with the real and imaginary parts represented with `float`.

Exercise 4

In this exercise you will convert the `Seq` class, which stores a sequence of `int` values (and is implemented with a simple linked list), to a template sequence class which can be instantiated to instances of different types. Thus, with the class, you can create sequences of `int`s, sequences of `double`s, etc. You will then write a small program to demonstrate the class.

Begin by downloading the files `Seq.h`, `Seq.cpp`. The `Seq.h` file contains the declaration of the `Seq` class. The `Seq.cpp` file contains the implementation of the `Seq` class.

Your task is to convert the `Seq` class to a template class, and write a test `main` function that demonstrates that it works. To convert the class to a template class, you must:

1. Put the entire class implementation into the `Seq.h` file. (You need to move the implementations from `Seq.cpp` to `Seq.`);
2. Change the class definition and each function definition to a template, by prefacing each with the line `template <typename T>`.
3. Re-write the class qualifier (`Seq::`) that appears before the function name in each function implementation as `Seq<T>::`

4. Replace all occurrences of **int** in type declarations for variables which are sequence elements to `T`. (There might be some **int** variable which are not sequence elements, so should not be changed.)

For example here is the add method before and after these changes:

```
void Seq::add(int x){
    Node *p = new Node; //temporary node
    // Assign appropriate values to the new node
    p -> data = x;
    p -> next = first;
    // Make first point to the new node
    first = p;
}
```

and the "templated" version

```
template <class T>
void Seq<T>::add(T x){
    Node *p = new Node; //temporary node
    // Assign appropriate values to the new node
    p -> data = x;
    p -> next = first;
    // Make first point to the new node
    first = p;
}
```

To use the `Seq` template class in a program, when you declare a variable of type `Seq`, you need specify what type it is to be, for example:

- `Seq<int>` is a sequence of **int**s, and
- `Seq<string>` is a sequence of `string`s

Write a test program that uses the `Seq` template class to make an **int** sequence, a `string` sequence, and one other type of sequence. Use output (with `cout` and the `display` function) to clearly demonstrate that the class works. The program should use each of the main `Seq` functions (`add`, `insertAt`, `remove`) at least once on each type, and use the `display` function (along with other output to the screen to make clear what is being shown) as many times as needed to demonstrate that the functions work. Do not make your test program more complex than is needed to accomplish this, though.

Exercise 5 .....

In this exercise, we will measure the execution time of `Seq` class functions and compare it with the execution time of `std::vector` functions.

Create a linked list of 1 million random **int** values using the `Seq` class and measure the time taken to insert an element at the beginning of the list. Do the same with the `std::vector` class.