Lecture 7 Templates

#CS106L

- **Template Functions**
- **Variadic Templates**
- **Concept Lifting**
- **Implicit Interfaces & Concepts**

---

the less than operator `<` is not defined for `strings`, so the comparison in the `my_minmax` function will not work as expected. If you want to compare strings lexicographically, you should use the `std::string::compare` function or the `std::less<std::string>` comparator.

Q1:template functions?

In C++, a template function is a way to create a single function that can operate on multiple types of data. A template function is defined with a template parameter, which is a placeholder for a specific type that will be provided when the function is called.

Here's an example of a simple template function:

```
//Be sure to inform the compiler that T is a type

template <typename T>

T add(T a, T b) {

    return a + b;

}
```

In this example, the function `add` takes two arguments of the same type `T`, and returns the sum of those two arguments. The `T` in the angle brackets is a template parameter, which is a placeholder for a specific type that will be provided when the function is called.

When the function is called, the compiler will generate a new version of the function with the specific type provided in place of the template parameter. For example, if we call the function as `add<int>(3, 4)`, the compiler will generate a new version of the function where `T` is replaced with `int`.

Here's an example of how you can use the function:

```
        int x = add<int>(3, 4);

        double y = add<double>(3.14, 2.72);
```

In this example, the function is called with two different types `int` and `double`, the compiler will generate two new versions of the function, one where `T` is replaced with `int` and another one where `T` is replaced with `double`

Template functions can be useful when you want to write a function that can operate on multiple types of data without having to write separate versions of the function for each type. It allows for code reusability and can make your code more readable and maintainable.

Q2:procedural programming?

Procedural programming is a programming paradigm that is based on the concept of procedures, also known as subroutines or functions. The main idea behind procedural programming is to break a program down into small, reusable blocks of code that can be executed in a specific order to accomplish a specific task.

In procedural programming, a program is composed of one or more procedures, each of which performs a specific task. These procedures are typically organized in a linear fashion, with each procedure being executed one after the other. The program is executed by calling the main procedure, which in turn calls other procedures as needed.

Procedural programming is characterized by the following features:

- Programs are divided into procedures that are executed in a specific order.
- Procedures are self-contained and have a specific purpose.

- Procedures may accept input parameters and return output values.
- Procedures may be called by other procedures, allowing for code reusability.
- Procedures may have local variables that are only visible within the scope of the procedure.
- Data and procedures are separate.

Q3:structured binding declarations?

Structured binding declarations is a feature in C++17 that allows you to assign the elements of a tuple or struct to individual variables. It uses the syntax `auto [var1, var2, ...] = expression;` where `var1`, `var2`, ... are the variable names you want to use and `expression` is the tuple or struct you want to extract the elements from.

For example, you could use structured binding to assign the elements of a tuple to individual variables like this:

```
        auto [x, y, z] = make_tuple(1, 2, 3);
```

This would create three variables `x`, `y`, and `z`, and assign the values 1, 2, and 3 to them respectively.

Similarly, for a struct, you could use structured binding like this:

```
        struct Point { int x, y; };

        Point p = {1, 2};

        auto [x, y] = p;
```

This would create two variables `x` and `y`, and assign the values of `p.x` and `p.y` to them respectively.

It is a useful feature which allows you to extract the elements of a tuple or struct in a more readable and concise way.

Q4:uniform initialization?

Uniform initialization is a feature in C++11 that allows you to initialize variables using a consistent syntax, regardless of the type of the variable or the constructor that is used to initialize it. It uses curly braces {} to enclose the initialization values, like this:

```
        int x{5};

        string s{"hello"};

        vector<int> v{1, 2, 3};
```

This feature is also known as "brace initialization" because it uses curly braces {} to initialize variables. The advantage of using uniform initialization is that it makes the code more readable and consistent, as the same initialization syntax is used for all types, regardless of whether they are built-in types, user-defined types, or aggregate types like arrays or structs.

It also does not require any additional constructors, and it allows for explicit conversion, for example:

```
        int x = {5.6}; // Compile error

        int y{5.6}; // warning in most of the cases
```

It also allows for aggregate initialization like this:

```
        struct MyStruct { int x, y; };

        MyStruct s = {1, 2}; // works only with uniform initialization.
```

In C++11, the use of uniform initialization is optional, but it is recommended to use it because it makes the code more consistent and readable. In C++14 and later, it is recommended to use uniform initialization by default because it is more powerful and expressive than the older initialization forms.

Q5:how to make this function generic?

```
int getInteger(const string& prompt, string& reprompt) {

        while (true) {

        cout << prompt;

        string line; int result; char extra;

        if (!getline(cin, line)) throw domain_error("[shortened]");
        istringstream iss(line);

        if (iss >> result && !(iss >> extra)) return result;

        cout << reprompt << endl;

        }

}
```

Here's an example of how you can make the `getInteger` function generic to work with different types of variables:

```
template <typename T>

T getValue(const string& prompt, const string& reprompt) {

    while (true) {

        cout << prompt;

        string line; T result; char extra;

        if (!getline(cin, line)) throw domain_error("[shortened]");

        istringstream iss(line);

        if (iss >> result && !(iss >> extra)) return result;

        cout << reprompt << endl;

    }

}
```

This way, the function can be called with any type, for example:

```
        int i = getValue<int>("Enter an integer: ", "Invalid input, try again.");

        double d = getValue<double>("Enter a double: ", "Invalid input, try
again.");
```

t's worth noting that if you want the function to work with user-defined types, you need to make sure that the `>>` operator is defined for this type, and the `istringstream` class should be able to extract values of this type from a string, otherwise the function will fail to work correctly.

Q6:how to write generic functions?

```
int countOccurences(const vector<int>& vec, int val) {

        int count = 0;

        for (size_t i = 0; i < vec.size(); ++i) {

        if (vec[i] == val) ++count;

        }

        return count;

}
```

Here's an example of how you can write a generic function to count the occurrences of a value in a container:

```
template <typename Container, typename T>

int countOccurences(const Container& container, const T& val) {

    int count = 0;

    for (const auto& element : container) {

        if (element == val) ++count;

    }

    return count;

}
```

You can use this function with any container that supports `begin()` and `end()` methods that return iterators, for example:

```
        vector<int> vec {1,2,3,4,5,6,7,8,1,1};

        int result = countOccurences(vec, 1);
        cout<<result<<endl;
```

It's worth noting that, as long as the `==` operator is defined for type T, the function will work with any type of container, such as `std::vector`, `std::list`, `std::set`, `std::deque`, etc. And also, the function will work with any type of value, as long as the `==` operator is defined for that type.

Q7(follow the above question):In your rewritten code, why

```
        int count = 0;
```

cannot be rewritten as:

```
        T count = 0;
```

In my previous example, I used `int count = 0;` instead of `T count = 0;` because `int` is a built-in type that is guaranteed to be able to hold any non-negative value, and it is also guaranteed to have the `++` operator defined. This means that we can increment the value of `count` in the loop, and it will always be safe to return the value of `count` as the result of the function.

On the other hand, when we use `T count = 0;`, The type T can be any type, so we can't guarantee that it can hold any non-negative value, or that it has the `++` operator defined. Therefore, it would not be safe to use `T count = 0;` in this function.

Another way to solve this problem is to use a default value for the count, for example:

```
template <typename Container, typename T>

T countOccurences(const Container& container, const T& val, T count = T()) {

    for (const auto& element : container) {

        if (element == val) ++count;

    }

    return count;

}
```

In this way, the function will work with any type that has a default constructor, and this will allow the function to work with user-defined types.

in C++, when we want to initialize a variable of a user-defined type, we can use the default constructor of that type. The default constructor is a constructor that can be called without any arguments, and it's automatically generated by the compiler if we don't define any constructors for a class.

For example, if you have a class `MyClass` and you want to create an instance of it, you can use the following code:

```
        MyClass obj;
```

This code will call the default constructor of the `MyClass` class, and it will create an instance of the class.

However, when we use a template, we don't know what type T is, so we can't use the following code:

```
        T count;
```

Because T could be any type, and we don't know if T has a default constructor or not.

To solve this problem, we can use the following code:

```
        T count = T();
```

This code will call the default constructor of the type T, and it will create an instance of T. The `T()` is called value initialization, and it will call the default constructor of T, if T is a built-in type, it will be zero-initialized.

So the `T count = T()` will initialize the variable count to a default-constructed value of the type T. This way, the function will work with any type that has a default constructor, regardless of whether it's a built-in type or a user-defined type, and this will allow the function to work with user-defined types.

In summary, `T()` is a way to ensure that the variable count is initialized to a default-constructed value of the type T, regardless of whether the type T has a default constructor defined or not.

Q8:varadic templates?

In C++, a variadic template is a type of template that can accept a variable number of template arguments. Variadic templates are useful when you want to write a function or class that can accept a variable number of arguments of any type.

In C++11, Variadic templates are implemented using the "ellipsis" (`...`) notation, which allows you to pass a variable number of arguments to a template. Here's an example of a simple variadic template function that takes a variable number of arguments and prints them to the console:

```
template <typename T, typename... Args>

void print(T first, Args... args) {

    cout << first << " ";

    print(args...);

}
```

In this example, the function `print` takes a first argument of type T and a variable number of arguments of any type, represented by the `Args...` template parameter. The function uses the `cout` stream to print the first argument and recursively calls itself with the remaining arguments.

Here's an example of how you can use the `print` function:

```
        print(1, 2, 3.14, "hello", 'A');
```

In this example, the function is called with five arguments, an `int`, an `int`, a `double`, a `string`, and a `char`. The function will print the values of these arguments to the console.

It's important to note that when using variadic template you will need a base case for recursion to prevent infinite loop, in the example above, the base case is an empty function call `print()`, in this case the function will stop calling itself.

In C++14 and later, there is another way to implement variadic templates using `std::integer_sequence` and `std::make_integer_sequence` which makes it more convenient and easier to use and maintain.

The `std::integer_sequence` is a template class that is used to hold a sequence of integers, and `std::make_integer_sequence` is a function template that generates an `std::integer_sequence` of a given type and size. These two templates can be used to generate a sequence of indices that correspond to the arguments passed to a function.

Here's an example of how you can use `std::integer_sequence` and `std::make_integer_sequence` to implement a variadic template function that prints all of its arguments:

```
template <typename... Args>

void print_args(Args... args) {

    (cout << ... << args) << endl;

}
```

In this example, the `print_args` function takes a variable number of arguments of any type, represented by the `Args...` template parameter. The `cout` stream is used to print the arguments passed to the function, the `...` operator is called the fold expression, it is used to expand the pack of arguments passed to the function, and the `endl` is used to end the output line.

Here's an example of how you can use the `print_args` function:

```
        print_args(1, 2, 3.14, "hello", 'A');
```

In this example, the function is called with five arguments, an `int`, an `int`, a `double`, a `string`, and a `char`. The function will print the values of these arguments to the console.

You can see that using `std::integer_sequence` and `std::make_integer_sequence` provides a more elegant and modern way to implement variadic templates, and it also allows you to handle the arguments passed to the function more easily.