Lecture 10 STL summary

#CS106L

- **Assignment 2 Preview**
- **BIG STL RECAP and Mystery Activity**
- **Let's Put It All Together!**

Do not use directly `using namespace std;`

different rules for invalidated containers:

- iterator to erasure/addition point always invalidated.
- `vector`: all iterators, pointers, and references invalidated.
- `deque`: all iterators, pointers, and references invalidated.(unless erasure point was front or back)
- `list/set/map/forward_list`: all other iterators, pointers and references are still valid.

Q1:This code is buggy!

```
void erase_all(vector<int>& vec, int val) {

        for (auto iter = vec.begin(); iter != vec.end(); ++iter) {

        if (*iter == val)  vec.erase(iter);


        }
```

```
}
```

The modified code is as follows:

```
void erase_all(vector<int>& vec, int val) {

        for (auto iter = vec.begin(); iter != vec.end();) {

        if (*iter == val)  iter = vec.erase(iter);

        else               ++iter;

        }

}
```

This function is basically removing all the elements that are equal to the given value from the vector. It uses the `iter = vec.erase(iter)` idiom. This is a common pattern when removing elements from a container while iterating through it. Because erase invalidates iterators, we need to update the iterator to the next element after the one that was erased, which is returned by the erase function. By doing so, we can ensure that the next element is not skipped while iterating through the container.

It's important to note that this function is not very efficient as it has a O(N) time complexity because it has to iterate through all elements of the vector and check each one individually. An alternative way of achieving the same result would be using the `std::remove` or `std::remove_if` algorithm which has a linear time complexity.

```
void remove_all(vector<int>& vec, int val) {

    vec.erase(std::remove(vec.begin(), vec.end(), val), vec.end());
}
```

```
void remove_all(vector<int>& vec, int val) {

    vec.erase(std::remove_if(vec.begin(), vec.end(), [val](int i){ return

i==val;}), vec.end());

}
```

Q2:trailing return type?

in some cases, it can be difficult or impossible to specify the return type before the function name, especially when the return type depends on template arguments. C++11 introduced a feature called "trailing return type" that allows you to specify the return type after the function name and parameters, using the `->` syntax.

The syntax for a trailing return type is as follows:

```
auto function_name(parameters) -> return_type { // function body }
```

The `auto` keyword is used as a placeholder for the return type, and the `-> return_type` syntax is used to specify the actual return type. The return type can be specified using any valid C++ type, including `auto`, `decltype`, and `std::decay`.

Here is an example of a function that uses a trailing return type to return the sum of two template arguments:

```
template<typename T, typename U>

auto add(T a, U b) -> decltype(a + b) {

    return a + b;

}
```

In this example, the function `add` takes two template arguments `T` and `U`, and returns the sum of the two arguments. The return type is specified using the `decltype` keyword, which deduces the type of the expression `a + b`. This means that the return type will be the same as the type of `a + b`, regardless of the types of `T` and `U`.

Trailing return type is useful when the return type of a function template depends on the template arguments. It allows the return type to be deduced automatically from the function body, rather than having to specify it explicitly. This can make the code more readable, and it can help to avoid errors when the return type changes.

Q3:decltype?

`decltype` is a C++ keyword that can be used to determine the type of an expression. It is often used in combination with template functions and type traits to deduce the types of function arguments or template parameters.

The basic syntax for `decltype` is:

```
decltype(expression)
```

`decltype` returns the type of the expression passed to it. The expression can be any valid C++ expression, including variables, function calls, and operator expressions.

Here are some examples of using `decltype`:

```
int x = 5;

decltype(x) y;   // y has type int

std::vector<int> v;

decltype(v)::value_type w;   // w has type int
```

In the first example, `decltype(x)` returns `int`, so the variable `y` is declared as an `int`. In the second example, `decltype(v)` returns `std::vector<int>`, and the `::value_type` member type is `int`, so the variable `w` is declared as an `int`.

`decltype` is particularly useful when working with template functions, as it allows you to deduce the types of function arguments or template parameters automatically.

Q4: `std::transform`?

`std::transform` is a standard algorithm in C++ that applies a given function to each element in a range and stores the result in a different range, specified by the user. The function takes four arguments:

- the first is the start of the input range,
- the second is the end of the input range,
- the third is the start of the output range,
- the fourth is the function that will be applied to each element.

Here is an example of how to use `std::transform` to square each element of a vector:

```
        std::vector<int> v = {1, 2, 3, 4, 5};

        std::vector<int> squared(v.size());

        std::transform(v.begin(), v.end(), squared.begin(), [](int x){ return x*x;
 });
```

It's also possible to use `std::transform` to modify the input range in place, by passing the same range as both the input and output range.

```
        std::vector<int> v = {1, 2, 3, 4, 5};

        std::transform(v.begin(), v.end(), v.begin(), [](int x){ return x*x; });
```

The function applies the lambda function to each element of the input range, and stores the result in the same element of the input range.

It's important to note that the input range and the output range can be of different types, and they don't have to be the same size, but they have to have the same number of elements. Also, it's important to note that the output range should be initialized before using `std::transform` or the result will be undefined.

`std::transform` works by applying the given function to each element in the input range, one at a time, and storing the result in the corresponding element of the output range. It starts at the first element of the input range, applies the function to that element, and stores the result in the first element of the output range. Then, it moves to the next element in the input range, applies the function to that element, and stores the result in the next element of the output range. This process continues until all elements in the input range have been processed. The algorithm returns an iterator pointing to the element following the last element of the output range.

This algorithm is useful in cases where you want to apply a specific operation to each element of a range, and store the results in a new range or modify the input range. It's particularly useful when combined with other STL algorithms, like `std::sort` and `std::unique`, to perform complex operations on ranges of data.

It's important to note that `std::transform` has a linear time complexity, which means that the algorithm takes O(n) time, where n is the number of elements in the input range.

Q5: `back_inserter`?

`std::back_inserter` is a helper function in C++ that creates an "insert iterator" for a container. An insert iterator is a type of iterator that allows you to insert elements into a container, rather than just reading or modifying existing elements.

Here is an example of how to use `std::back_inserter` to insert elements into a vector:

```
        std::vector<int> v;

        std::copy(input.begin(), input.end(), std::back_inserter(v));
```

It's also possible to use `std::back_inserter` with other algorithms like `std::transform` to insert the result of the operation into the container:

```
        std::vector<int> v;

        std::transform(input.begin(), input.end(), std::back_inserter(v), [](int x)
 { return x*x; });
```

In this example, `std::back_inserter(v)` creates an insert iterator that can be used to insert elements at the end of the vector `v`. The `std::transform` function is used to apply the lambda function to each element in the input range `input.begin()` to `input.end()`, and insert the result into the vector.

`std::back_inserter` is a helper function that can be used with any container that has a `push_back` member function, like `std::vector`, `std::deque`, `std::list`, etc. It's important to note that using an insert iterator can be less efficient than using the `emplace_back` member function, which allows you to construct the element in-place.

In summary, `std::back_inserter` is a helper function that creates an insert iterator that can be used to insert elements into a container, specifically at the end of the container, it works by calling the container's `push_back` member function on each element that is inserted. It allows you to use standard algorithms like `std::copy` and `std::transform` to insert elements into a container, rather than having to use a loop and call `push_back` manually.

It's important to note that the container passed to `std::back_inserter` should be pre-allocated with enough capacity or the container should have amortized constant time complexity for push_back operation, otherwise, the use of `std::back_inserter` can lead to poor performance due to the need of reallocating the container multiple times.

Q6:Why `using namespace std` is not a good idea?

The main problem with using `using namespace std;` is that it can lead to naming conflicts. If you use it, all the identifiers in the `std` namespace are in the same scope as your own identifiers, which means that if you or other libraries you are using have an identifier with the same name as an identifier in the `std` namespace, the compiler will not be able to distinguish between them, and it will lead to errors or unexpected behavior.

Another problem with using `using namespace std;` is that it can make your code less readable. If you use it, it's harder to tell where an identifier comes from, which can make it harder to understand what your code is doing.Also,it pollutes the global namespace, and it could lead to unexpected behavior if the same identifier is defined in the global namespace and in the `std` namespace.

For these reasons, it's generally considered a best practice to avoid using `using namespace std;` in large or complex programs and instead use the `std::` prefix for identifiers from the `std` namespace, or use specific `using` statements for the identifiers that you need.

Q7:`#include` and `using`?

`#include` is a preprocessor directive that tells the compiler to include the contents of a specified file in the current source file, before the compilation process starts. The file that is included can contain any valid C++ code, including definitions of types, functions, and variables. The most common use of `#include` is to include the contents of standard C++ library headers, such as `<iostream>`.

On the other hand, `using` is a C++ keyword that allows you to specify which namespaces or identifiers should be visible in the current scope, without having to prefix them with the namespace or class name.

- `using namespace N;` : allows all names declared in namespace N to be visible without the prefix N::

- `using N::name;` : allows the name declared in namespace N to be visible without the prefix N::

Additionally, it's important to note that, including a file with `#include` can increase the compilation time and the size of the binary, so it's important to keep the number of includes to a minimum and use forward declarations where possible.