Advanced Programming

Standard Template Library (STL)

The C++ Standard Template Library (STL) is a powerful library that provides a set of common data structures and algorithms. This supplementary material focuses on the algorithms provided by STL, as well as the concepts of functors, lambdas, and allocators.

1 Algorithms

STL algorithms are generic functions that operate on sequences of elements. These sequences are usually provided by containers (e.g., 'vector', 'list', 'set'), and the algorithms perform tasks such as searching, sorting, modifying, and more.

STL algorithms are divided into several categories:

- **Non-modifying Sequence Operations**: Operations that do not change the elements of the container.
- **Modifying Sequence Operations**: Operations that alter the elements of the container.
- **Sorting and Searching**: Operations for ordering elements and searching for specific elements.
- **Numeric Operations**: Operations that perform numerical computations on sequences.

1.1 Non-modifying Sequence Operations

1.1.1 find

The `find` algorithm searches for a specific value in a range and returns an iterator to the first occurrence of the value.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    auto it = std::find(vec.begin(), vec.end(), 3);
    if (it != vec.end()) {
        std::cout << "Found: " << *it << std::endl;
    } else {
        std::cout << "Not found" << std::endl;
    }
    return 0;
}
```

The `find` algorithm makes use of value–based criteria. Alternatively, for boolean criteria we can use `find_if` or `find_if_not`. The `find_if` algorithm finds the first element that matches a predicate that returns `true`, whilst `find_if_not` finds the first element that matches a predicate* that returns `false`.

*A predicate is a function or a function object that returns a `bool`. Unary predicates take one argument. These predicates are particularly useful testing conditions.

Let's see an example

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

// Let's define a couple of predicates
bool isOdd(int i) {
    return i % 2 != 0;  // returns true if i is odd
}
bool isEven(int i) {
    return i % 2 == 0;  // returns true if i is even
}

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // We have two options for finding the first odd value
    // 1. using find_if and isOdd
    auto it = std::find_if(vec.begin(), vec.end(), isOdd);
    if (it != vec.end()) {
        std::cout << "Found first odd value (using find_if): " << *it << std::endl;
    }
    // 2. using find_if_not and isEven
    auto it2 = std::find_if_not(vec.begin(), vec.end(), isEven);
    if (it2 != vec.end()) {
        std::cout << "Found first odd value (using find_if_not): " << *it2 << std::endl;
    }

    // Likewise, for finding the first even value
    // 1. using find_if and isEven
    auto it3 = std::find_if(vec.begin(), vec.end(), isEven);
    if (it != vec.end()) {
        std::cout << "Found first even value (using find_if): " << *it3 << std::endl;
    }
    // 2. using find_if_not and isOdd
    auto it4 = std::find_if_not(vec.begin(), vec.end(), isOdd);
    if (it4 != vec.end()) {
        std::cout << "Found first even value (using find_if_not): " << *it4 << std::endl;
    }
    return 0;
}
```

1.1.2 count

The `count` algorithm counts the number of occurrences of a specific value in a range.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 3, 5};
    // let's count the occurrences of 3 in vec
    int count = std::count(vec.begin(), vec.end(), 3);
    std::cout << "Count of 3s: " << count << std::endl;
    return 0;
}
```

1.1.3 for_each

The `for_each` algorithm applies a function to each element in a range.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// define a function
int increment(int &n) {
    return ++n;
}

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // apply the function to each element in vec
    std::for_each(vec.begin(), vec.end(), increment);
    for (int i : vec){
        std::cout << i << std::endl;
    }
    return 0;
}
```

1.2 Modifying Sequence Operations

1.2.1 copy

The `copy` algorithm copies elements from one range to another

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec1 = {1, 2, 3, 4, 5};
    std::vector<int> vec2(5);
    // Let's see vec2 before copy
    std::cout << "vec2 before copy" << std::endl;
    for (int i : vec2) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
    // copy
    std::copy(vec1.begin(), vec1.end(), vec2.begin());
    // after copy
    std::cout << "vec2 after copy" << std::endl;
    for (int i : vec2) {
        std::cout << i << " ";
    }
    return 0;
}
```

In this example the target container `vec2` is created with the same size of the source container `vec` (in this case 5). Try yourself changing the size of the target container and check the output.

1.2.2 transform

The `transform` algorithm applies a function to a range of elements and stores the result in another range.

Recall the example for the `for_each` algorithm. By applying the function `triple` to each element in `vec` the values were modified inplace. If want to preserve the original values in `vec`, we would need to create a new container `vec2`, as shown in the example for the `copy` algorithm and use `for_each` on this copy:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// define a function
int increment(int &n) {
    return ++n;
}

int main() {
    std::vector<int> vec1 = {1, 2, 3, 4, 5};
    std::vector<int> vec2(5);
    // copy values
    std::copy(vec1.begin(), vec1.end(), vec2.begin());
    // apply the function to each element in vec2
    std::for_each(vec2.begin(), vec2.end(), increment);
    for (int i : vec2) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

The `transform` algorithm reduces the procedure to one line

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// define a function
int increment(int n) {
    return ++n;
}

int main() {
    std::vector<int> vec1 = {1, 2, 3, 4, 5};
    std::vector<int> vec2(5);
    // using transform
    std::transform(vec1.begin(), vec1.end(), vec2.begin(), increment);
    for (int i : vec2) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

1.3 Sorting and Searching

1.3.1 sort

The `sort` algorithm sorts the elements in a range according to a specific criterion. This algorithm uses ascending order by default, but custom comparison functions can be used to change the sorting order.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// create a comparison function for sorting in descending order
bool descend(int a, int b){
        return a > b;
}

```

```cpp
int main() {
    std::vector<int> vec = {4, 2, 3, 1, 5};
    // sort in ascending order (default)
    std::sort(vec.begin(), vec.end());
    for (int i : vec) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // sort in descending order
    // using our comparison function descend as sorting criterion
    std::sort(vec.begin(), vec.end(), descend);
    for (int i : vec) {
        std::cout << i << " ";
    }
    return 0;
}
```

1.3.2 binary_search

The `binary_search` algorithm checks whether a value exists in a sorted range. It returns a boolean indicating the presence of the value.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    bool found = std::binary_search(vec.begin(), vec.end(), 3);
    if (found) {
        std::cout << "Element found" << std::endl;
    } else {
        std::cout << "Element not found" << std::endl;
    }
    return 0;
}
```

Recall the example for the `find` algorithm. Can you see similarities/differences?

1.4 Numeric Operations

1.4.1 accumulate

The `accumulate` algorithm computes a cumulative operation of a range of elements. The default operation is sum, but custom cumulative operations can be used.

```cpp
#include <iostream>
#include <numeric>
#include <vector>
#include <string>

// Define a custom operation for joining elements as a string
auto join = [](std::string a, int b){
    return std::move(a) + '-' + std::to_string(b);
};

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
```

```
13
14      // default operation: Sum
15      // Note the argument 0. This is the initial value for the operation
16      int sum = std::accumulate(vec.begin(), vec.end(), 0);
17      std::cout << "Sum: " << sum << std::endl;
18
19      // using an operation from std namespace
20      // std::multiplies<int>(): integer multiplication
21      // Note the argument 0. This is the initial value for the operation
22      int prod = std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
23      std::cout << "Product: " << prod << std::endl;
24
25      // using custom operation join
26      // Note the range bounds and the initial value
27      std::string s = std::accumulate(std::next(vec.begin()), vec.end(),
28                                      std::to_string(vec[0]),
29                                      join);
30      std::cout << "String: " << s << std::endl;
31      return 0;
32  }
```

Note the definition of the custom operation. The syntax used corresponds to a *lambda expression*. Lambda expressions are explained below.

1.4.2 partial_sum

The `partial_sum` algorithm computes the partial sums of a range of elements and stores them in another range.

```
1   #include <iostream>
2   #include <numeric>
3   #include <vector>
4
5   int main() {
6       std::vector<int> vec = {1, 2, 3, 4, 5};
7       std::vector<int> result(vec.size());
8       std::partial_sum(vec.begin(), vec.end(), result.begin());
9       int c = 1;
10      for (int i : result) {
11              std::cout << "Sum of " << c << " first elements of vec: ";
12          std::cout << i << std::endl;
13          ++c;
14      }
15      return 0;
16  }
```

2 Function Objects (Functors) and Lambdas

2.1 Function Objects (Functors)

A functor is an object that can be called as if it were a function. Functors are useful in situations where you want to pass a function as an argument to an algorithm but need to maintain state.

2.1.1 Example of a Functor

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>
4
```

```cpp
// create the function object Multiply
struct Multiply {
    int factor;
    Multiply(int f) : factor(f) {}
    void operator()(int &n) const {
        n *= factor;
    }
};

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // use the function object as argument in an algorithm
    std::for_each(vec.begin(), vec.end(), Multiply(2));
    for (int i : vec) {
        std::cout << i << " ";
    }
    return 0;
}
```

2.2 Lambda Expressions in C++

Lambda expressions are a convenient way to define anonymous functions in C++. They are especially useful in STL algorithms where simple function objects are needed.

2.2.1 Example of a Lambda Expression

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// Define a lambda expression
auto triple = [](int &n) {
    return n *= 3;
};

int main() {
        std::vector<int> vec = {1, 2, 3, 4, 5};
    // pass the lambda expression to an algorithm
    std::for_each(vec.begin(), vec.end(), triple);
    for (int i : vec) {
        std::cout << i << " ";
    }
        std::cout << std::endl;
    // This can be simplified by passing the operation
    // following the lambda expression syntax
    std::for_each(vec.begin(), vec.end(), [](int &n) { n *= 2; });
    for (int i : vec) {
        std::cout << i << " ";
    }

    return 0;
}
```

You may have noticed the square brackets (`[]`). In the previous example brackets remain empty. However, these brackets are for capturing variables from the enclosing local scope of a lambda expression. This allows lambdas to use local variables in their bodies.

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

```cpp
#include <algorithm>

int main() {
    // define the variable factor in the enclosing scope of the lambda expression
    int factor = 2;
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // capture the variable and use it in the body of the lambda expression
    std::for_each(vec.begin(), vec.end(), [factor](int &n) { n *= factor; });
    for (int i : vec) {
        std::cout << i << " ";
    }
    return 0;
}
```

3 References

- Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference.*
- Bjarne Stroustrup, *The C++ Programming Language.*
- https://en.cppreference.com/w/