by Leedehai

# Chapter 16(2) A Glimpse into The Standard Template Library

Good references:

Chapter 16 and Appendices F, G, *C++ Primer Plus*, 6th edition, by Stephen Prata

Part II, *C++ Primer*, 5th edition, by Stanley Lippman, et al.

*The C++ Standard Library A Tutorial and Reference*, 2nd edition, by Nicolai Josuttis

Golden rule: **using makes familiar, practice makes perfect.**

## 16.8 Functors, namely function objects

- Many STL algorithms use *function object*s, also known as *functor*s. A functor is any object (NOT necessarily class object) that can be used with a function-call operator `()` in the manner of a function (Yes, the definition is as simple as it is.)

  > The term "functor" is also used in category theory, a ramification of mathematics. Category theory has practical applications in programming language theory, notably in the study of monads in functional programming.

- This generalized idea includes normal function names, pointers to functions, and class objects for which the operator `()` is overloaded (its operator function is `operator()()`).
- Functor conceptual categories
  - A *generator* is a functor that can be called with no arguments.
  - A *unary function* is a functor that can be called with one argument.
  - A *binary function* is a function is a functor that can be called with two arguments.
  - A unary function that returns a `bool` value is a *predicate*.
  - A binary function that returns a `bool` value is a *binary predicate*.
- A simple example of a functor
  - The `list` template (declared in `<list>`) has a `remove_if()` member that takes a predicate as an argument. It applies the predicate to each member in the indicated range, removing those elements for which the predicate returns `true`.
  - For example. the following code would remove all elements greater than 100:

    ```cpp
    bool tooBig(int a) {return a > 100;} // this is a predicate
    std::list<int> scores {10, 30, 50, 70, 90, 105, 110};
    scores.remove_if(tooBig); // remove_if() takes a predicate
    ```

  - Suppose you want to remove every element whose value is greater than a designated value. It would be nice if you could pass the cutoff value to `tooBig()` as a second argument so you could use the function with different values, but a predicate, which is required by `remove_if()`, can have but one argument. To circumvent this restriction, you can use class members instead of function argument to convey this additional information:

```cpp
template <typename T> class TooBig {
private: T cutoff;
public: TooBig(const T &c) : cutoff(c) {}
        bool operator()(const T &a) { return a > cutoff; }
};
```

  - The `TooBig` class template is used in this way:

    ```cpp
    int val[] = {10, 30, 50, 70, 90, 105, 110};
    TooBig<int> tooBig(105); // declares a predicate
    std::list<int> scores(val, val + sizeof(val)); // range constructor
    scores.remove_if(tooBig);
    scores.remove_if(TooBig<int>(105)); // more concise
    ```

- Another example of a functor
  - The `for_each()` function template, briefly used in an example in 16.4, takes a unary functor as the third argument:

    ```cpp
    template <class InputIterator, class Functor>
    Functor for_each(InputIterator first, InputIterator last, Functor f);
    ```

    A function that fits the third argument looks like this:

    ```cpp
    void showReview(const Review &);
    ```

    This makes the identifier `showReview` has the type `void (*)(const Review &)`, so this is the type that is assigned to the template argument `Functor`. With a different function call, the `Functor` argument could represent a class type that has an overloaded `()` operator. Ultimately, the `for_each()` code will have an expression using `f()`. In the `showReview()` example, `f` is a pointer to a function, and `f()` invokes the function. If the final `for_each()` argument is an object instead of a function pointer, then `f` becomes the object that invokes the object's overloaded `()` operator.

- Predefined functors in the STL
  - To illustrate the functionalities of these predefined functors, it is helpful to roll out a STL algorithm `transform()` (declared in `<algorithm>`). It has several versions, two of which are listed below (the other two involve rvalue reference):

    ```cpp
    // version #1: "op" functor is applied to each element
    template <class InputIter, class OutputIter, class UnaryOperation>
    OutputIter transform(InputIter first, InputIter last,
                         Output dest_first, UnaryOperation op);
    // version #2: "op" functor is applied to each element pair
    template <class InputIter1, class InputIter2, class OutputIter,
              class BinaryOperation>
    OutputIter transform(InputIter first1, InputIter last1, /* first */
                         InputIter first2,                  /* second */
                         OutputIter dest_first, BinaryOperation op);
    ```

- Calculating the square root (with `sqrt()` defined in `<cmath>`) of each element in a `vector`, and display the results.

```
const int N = 5; double arr[N] = {36, 39, 42, 45, 48};
std::vector<double> vd(arr, arr + N);
std::ostream_iterator<double, char> out(std::cout, " ");
std::transform(vd.begin(), vd.end(), out, std::sqrt); // use ver. #1
```

- Calculating the addition (with template class `plus` defined in `<functional>`) of each element pair, and display the results.

```
const int N = 5; double a[N] = {...}; double b[N] = {...};
std::vector<double> va(a, a + N); std::vector<double> vb(b, b + N);
std::ostream_iterator<double, char> out(std::cout, " ");
std::transform(va.begin(), va.end(), vb.begin, out, plus<double>());
```

  Here, rather than create a named object, the code uses the `plus<double>` class's default constructor to construct a functor to do the adding. This functor overloaded the `()` operator and can takes in two arguments, thus it is a binary functor.

- The STL has some predefined functor class templates for some ordinary operations. They can be instantiated with any built-in or any user-defined types.

| Operator | Class template | Operator | Class template |
|---|---|---|---|
| `+` | `plus` | `>` | `greater` |
| `-` (subtract) | `minus` | `<` | `less` |
| `*` | `multiplies` | `>=` | `greater_equal` |
| `/` | `divides` | `<=` | `less_equal` |
| `%` | `modulus` | `&&` | `logical_and` |
| `-` (negate) | `negate` | `\|\|` | `logical_or` |
| `==` | `equal_to` | `!` | `logical_not` |
| `!=` | `not_equal_to` | | |

## 16.9 Algorithms

- *Algorithm*s, or *algorithm function*s, are part of the STL. They have two main features:
  - They use templates to provide generic types.
  - They use iterators to provide a generic representation for accessing data.
- Four groups of algorithms
  - Non-modifying sequence operations, declared in `<algorithm>` (formerly `algo.h`). These operations operate on each element in a range and leaved a container unchanged. e.g. `find()`, `for_each()`.
  - Mutating sequence operations, declared in `<algorithm>` (formerly `algo.h`). These operations

    also operate on each element in a range, and may alter the contents of a container. e.g. `transform()`, `random_shuffle()`, `copy()`.
  - Sorting and related operations, declared in `<algorithm>` (formerly `algo.h`). It include several sorting functions and a variety of other functions, including the set operations. e.g. `sort()`.
  - Generalized numeric operations, declared in `<numeric>` (formerly, too, `algo.h`). They include functions to sum the content of a range, calculate the inner product of two containers, calculate partial sums, calculate adjacent differences, etc. Typically `vector` is the container most likely to be used with them since they are characteristic of arrays.

  > For a complete summary of these functions, see Appendix G, *C++ Primer Plus*, 6th edition, by Stephen Prata.

- Algorithm functions vs. container methods
  - Sometimes you have a choice between using an STL algorithm function and a STL container method. Usually, the method is better, because (1) the method should be optimized for a particular container, and (2) being a member function, it can use a template class's memory management facilities and resize a container when needed.

    > For example, template class `list` has a method called `remove()`, and there is also an STL algorithm function called `remove()`. Their syntaxes are different, though.

  - Although the methods are usually better suited, the non-method functions are more general.
- A brief tour of the algorithms, see page 870, *C++ Primer*, 5th edition, by Stanley Lippman, et al.

  □