Programming in Modern C++

Module M43: C++ Standard Library (Generic Programming): Part 1

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Understood object-oriented I/O of C++
- Learnt the major standard library components

Module Objectives

- To get an overview of Standard Library components of C++
- To understand generic programming for STL

Module Outline

Standard Library

Sources:

- Standard library, Wikipedia
- C math.h library functions

What is Standard Library?

- A *standard library in programming* is the library *made available across implementations of a language*
- These libraries are usually described in *language specifications (C/C++)*; however, they may also be determined (in part or whole) *by informal practices of a language's community (Python)*
- A language's standard library is *often treated as part of the language by its users*, although the *designers may have treated it as a separate entity*
- Many language specifications define a *core set that must be made available in all implementations*, in addition to *other portions which may be optionally implemented*
- The line between a *language and its libraries* therefore *differs from language to language*
- Bjarne Stroustrup, designer of C++, writes:

> *What ought to be in the standard C++ library? One ideal is for a programmer to be able to find every interesting, significant, and reasonably general class, function, template, etc., in a library. However, the question here is not, "What ought to be in some library?" but "What ought to be in the standard library?" The answer "Everything!" is a reasonable first approximation to an answer to the former question but not the latter. A standard library is something every implementer must supply so that every programmer can rely on it.*

- This suggests a *relatively small standard library*, containing only the constructs that *"every programmer" might reasonably require when building a large collection of software*
- **This is the philosophy that is used in the C and C++ standard libraries**

Source: Standard library, Wiki

C Standard Library: Common Library Components

Component	Data Types, Manifest Constants, Macros, Functions, ...
stdio.h	Formatted and un-formatted file input and output including functions • printf, scanf, fprintf, fscanf, sprintf, sscanf, feof, etc.
stdlib.h	Memory allocation, process control, conversions, pseudo-random numbers, searching, sorting • malloc, free, exit, abort, atoi, strtold, rand, bsearch, qsort, etc.
string.h	Manipulation of C strings and arrays • strcat, strcpy, strcmp, strlen, strtok, memcpy, memmove, etc.
math.h	Common mathematical operations and transformations • cos, sin, tan, acos, asin, atan, exp, log, pow, sqrt, etc.
errno.h	Macros for reporting and retrieving error conditions through error codes stored in a static memory location called errno • EDOM (parameter outside a function's domain – sqrt(-1)), • ERANGE (result outside a function's range), or • EILSEQ (an illegal byte sequence), etc.

A header file typically contains manifest constants, macros, necessary struct / union types, typedef's, function prototype, etc.

C Standard Library: math.h


```
/* math.h
 * This file has no copyright assigned and is placed in the Public Domain.
 * This file is a part of the mingw-runtime package.
 * Mathematical functions.
 */
#ifndef _MATH_H_
#define _MATH_H_
#ifndef __STRICT_ANSI__  // conditional exclusions for ANSI
// ...
#define M_PI 3.14159265358979323846  // manifest constant for pi
// ...
struct _complex {  // struct of _complex type
    double        x;        /* Real part */
    double        y;        /* Imaginary part */
};
_CRTIMP double __cdecl _cabs (struct _complex);  // cabs(.) function header
// ...
#endif /* __STRICT_ANSI__ */
// ...
_CRTIMP double __cdecl sqrt (double);  // sqrt(.) function header
// ...
#define isfinite(x) ((fpclassify(x) & FP_NAN) == 0)  // macro isfinite(.) to check if a number is finite
// ...
#endif /* _MATH_H_ */
```


Source: C math.h library functions

C++ Standard Library: Common Library Components

Component	Data Types, Manifest Constants, Macros, Functions, Classes, ...
iostream fstream	Stream input and output for standard I/O • cout, cin, endl, ..., etc. (Module 42)
string	Manipulation of string objects • Relational operators, IO operators, Iterators, etc.
memory	High-level memory management • Pointers: unique_ptr, shared_ptr, weak_ptr, & allocator etc.
exception stdexcept	Generic Error Handling • exception, bad_exception, unexpected_handler, & terminate_handler Standard Error Handling • logic_error, invalid_argument, domain_error, length_error, out_of_range, runtime_error, range_error, overflow_error, underflow_error, etc.

STL	Utilities
Containers	vector, deque, list, stack, queue, priority_queue, set, multiset, map, multimap C++11: array, forward_list, unordered_set/multiset/map/multimap
Iterators	begin & end, rbegin & rend. C++11: cbegin & cend, crbegin & crend,
algorithm	Non-Numerical: for_each, find, find_if, count, search, copy, move, swap, replace, fill, generate, remove, reverse, rotate, sort, binary_search, merge, min, max, ···
numeric	Numerical: accumulate, adjacent_difference, inner_product, and partial_sum. C++11: iota
Functions	equal_to, not_equal_to, greater, greater_equal, less, less_equal; plus, minus, multiplies, divides, modulus; logical_and, logical_not, logical_or. C++11: bit_and, bit_or, bit_xor

Imported from C Standard Library	
cmath	Common mathematical operations and transformations • cos, sin, tan, acos, asin, atan, exp, log, pow, sqrt, etc.
cstdlib	Memory alloc., process control, conversions, pseudo-rand nos., searching, sorting • malloc, free, exit, abort, atoi, strtold, rand, bsearch, qsort, etc.

namespace std for C++ Standard Library

C Standard Library	C++ Standard Library
• All names are global • stdout, stdin, printf, scanf	• All names are within std namespace • std::cout, std::cin • Use using namespace std; to get rid of writing std:: for every standard library name

W/o using

```
#include <iostream>


int main() {

    std::cout << "Hello World in C++"
              << std::endl;

    return 0;
}
```

W/ using

```
#include <iostream>
using namespace std;

int main() {

    cout << "Hello World in C++"
         << endl;

    return 0;
}
```

Standard Library: C/C++ Header Conventions

	C Header	C++ Header
C Program	Use `.h`. Example: `#include <stdio.h>` *Names in global namespace*	Not applicable
C++ Program	Prefix `c`, no `.h`. Example: `#include <cstdio>` *Names in `std` namespace*	No `.h`. Example: `#include <iostream>`

- A C std. library header is used in C++ with prefix '`c`' and without the `.h`. These are in `std` namespace:

```
#include <cmath>  // In C it is <math.h>
...
std::sqrt(5.0);   // Use with std::
```

It is possible that a C++ program include a C header as in C. Like:

```
#include <math.h> // Not in std namespace
...
sqrt(5.0);        // Use without std::
```

This, however, is not preferred

- **Using `.h` with C++ header files, like `iostream.h`, is disastrous. These are deprecated. It is dangerous, yet true, that some compilers do not error out on such use. Exercise caution.**

Generic Programming

Source:

- GENERIC PROGRAMMING, Sean Parent, code::dive conference 2018
- Chapter 20 The STL (containers, iterators, and algorithms), Bjarne Stroustrup
- Chapter 21 The STL (maps and algorithms), Bjarne Stroustrup

Common Programming Tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (for example, get the N^{th} element)
  - By value (for example, get the first element with the value `Chocolate`)
  - By properties (for example, get the first elements where `age < 64`)
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

Common Tasks have Common Goals

- We can (already) write programs that are very similar independent of the data type used (Recall templates)
  - Using an `int` is not that different from using a `double`
  - Using a `vector<int>` is not that different from using a `vector<string>`
- We would like to write common programming tasks so that we do not have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data
  - Finding a value in a `vector` is not all that different from finding a value in a `list` or an array
  - Looking for a `string` ignoring case is not all that different from looking at a `string` not ignoring case
  - Graphing experimental data with exact values is not all that different from graphing data with rounded values
  - Copying a file is not all that different from copying a `vector`

Ideals for Commonly used Common Codes

- Code that is
  - Easy to read
  - Easy to modify
  - Regular
  - Short
  - Fast
- Uniform access to data
  - Independently of how it is stored
  - Independently of its type
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - Retrieval of data
  - Addition of data
  - Deletion of data
- Standard versions of the most common algorithms
  - Copy, find, search, sort, sum, ...

Examples

- Sort a vector of strings
- Find a number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 800
- Find the first occurrence of the value 17
- Sort the telemetry records by unit number
- Sort the telemetry records by time stamp
- Find the first value larger than "Petersen"?
- What is the largest amount seen?
- Find the first difference between two sequences
- Compute the pairwise product of the elements of two sequences
- What are the highest temperatures for each day in a month?
- What are the top 10 best-sellers?
- What is the entry for "C++" (say, in Google)?
- What is the sum of the elements?

Generic Programming

- Generalize algorithms
  - Sometimes called *lifting an algorithm*
- The aim (for the end user) is
  - Increased correctness
    - Through better specification
  - Greater range of uses
    - Possibilities for re-use
  - Better performance
    - Through wider use of tuned libraries
    - Unnecessarily slow code will eventually be thrown away
- Go from the concrete to the more abstract
  - The other way most often leads to bloat

Lifting example: Concrete Algorithms


```
// Sum in Array: one concrete algorithm (doubles in array)
double sum(double array[], int n) { // data
    double s = 0;
    for (int i = 0;
            i < n;          // not at end
            ++i)            // get next data element
        s = s + array[i]; // get value
    return s;
}

// Sum in List: another concrete algorithm (ints in list)
struct Node { Node* next; int data; };
int sum(Node* first) { // data
    int s = 0;
    while (first) {           // not at end : terminates on null pointer
        s = s + first->data; // get value
        first = first->next; // get next data element
    }
    return s;
}
```

Lifting example: Abstract the data structure


```
// pseudo-code for a more general version of both algorithms

int sum(data) {                    // somehow parameterize with the data structure
    int s = 0;                     // initialize
    while (not at end) {           // loop through all elements
        s = s + get value;         // compute sum
        get next data element;
    }
    return s;                      // return result
}
```

- We need three operations (on the data structure):
  - `not at end`
  - `get value`
  - `get next data element`

Lifting example: Using template

```
// Concrete STL-style code for a more general version of both algorithms

template<class Iter, class T>       // Iter should be an Input_iterator
                                    // T should be something we can + and =
T sum(Iter first, Iter last, T s) { // T is the accumulator type
    while (first != last) {         // not at end
        s = s + *first;             // get value
        ++first;                    // get next data element
    }
    return s;
}
```

- Let the user initialize the accumulator

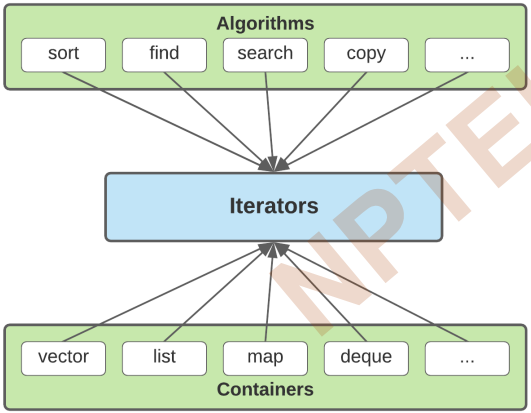
```
float a[] = { 1,2,3,4,5,6,7,8 }; // a[0], a[0], ..., a[7]
double d = 0;
d = sum(a, a+sizeof(a)/sizeof(*a), d); // [&a[0],&a[8])= {a[0], a[0], ..., a[7]}
```

Lifting example

- Almost the standard library accumulate
  - A bit for terseness is simplified
- Works for
  - arrays
  - `vector`s
  - `list`s
  - `istream`s

  ...
- Runs as fast as *hand-crafted* code
  - Given decent inlining
- The code's requirements on its data has become explicit
  - We understand the code better

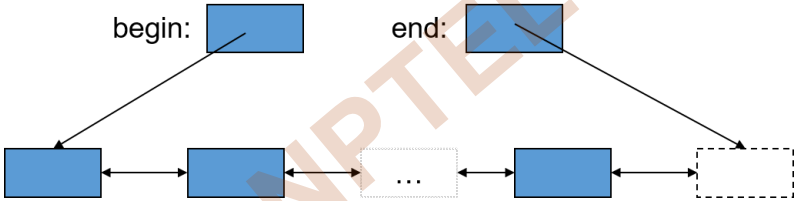
Basic Model: Algorithms ==> Iterators <== Containers



- **Separation of Concerns**
  - *Algorithms* manipulate data, but do not know about *Containers*
  - *Containers* store data, but do not know about *Algorithms*
  - *Algorithms* and *Containers* interact through *Iterators*
  - Each *Container* has its *own iterator types*

Basic Model: Iterators

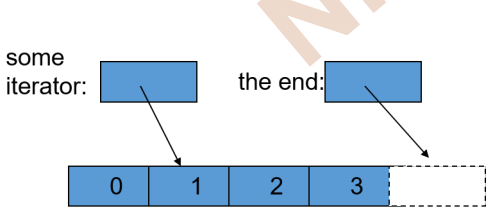
- A pair of iterators defines a sequence
  - The *beginning* (points to the *first element* – if any)
  - The *end* (points to the *one-beyond-the-last element*)

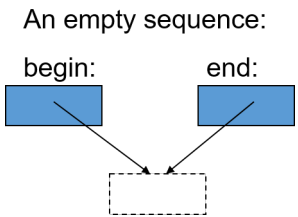


- An iterator is a type that supports the *iterator operations*
  - `++` Go to next element
  - `*` Get value
  - `==` Does this iterator point to the same element as that iterator?
- Some iterators support more operations (for example, `--`, `+`, and `[ ]`)

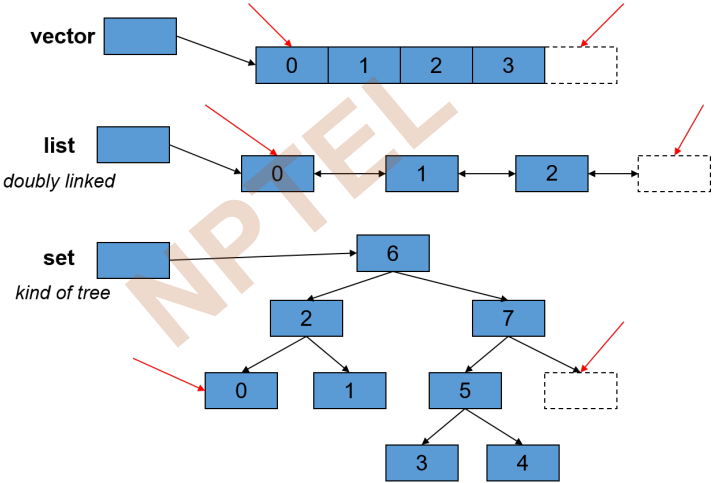
Basic Model: Algorithms + Iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is *one past the last element*
  - not *the last element*
  - That is necessary to elegantly represent an empty sequence
  - One-past-the-last-element is not an element
    - ▷ You can compare an iterator pointing to it
    - ▷ You cannot dereference it (read its value)
- Returning the end of the sequence is the standard idiom for *not found* or *unsuccessful*





Basic Model: Containers + Iterators



Algorithm: find()


```
// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val) {
    while (first != last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x) { // works for vector of ints
    vector<int>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}

void f(list<string>& v, string x) { // works for list of strings
    list<string>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}

void f(set<double>& v, double x) { // works for set of doubles
    set<double>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}
```

Algorithm: find_if()

```
// Find the first element that matches a criterion (predicate)
template<class In, class Pred>
In find_if(In first, In last, Pred pred) {
    while (first != last && !pred(*first)) ++first;
    return first;
}

void f(vector<int>& v) {
    vector<int>::iterator p = find_if(v.begin(), v.end, Odd()); // Here, a predicate takes
                                                                // one argument and returns a bool
    if (p != v.end()) { /* we found an odd number */ }
    // ...
}
```

- A predicate (often of one argument) is a function or a function object returns a `bool` given the argument/s. For example

```
// A function
bool odd(int i) { return i % 2; } // % is the remainder (modulo) operator
odd(7);                           // call odd: is 7 odd?

// A function object (Module 40)
struct Odd { bool operator()(int i) const { return i % 2; } };
Odd odd;    // make an object odd of type Odd
odd(7);     // call odd: is 7 odd?
```

Module Summary

- Overview of Standard Library components of C++
- Learnt fundamentals of generic programming