CS241 Principles and Practice of Problem Solving

Lecture 8: Containers and Algorithms

Yuye Ling, Ph.D.

Shanghai Jiao Tong University
John Hopcroft Center for Computer Science

October 10, 2019

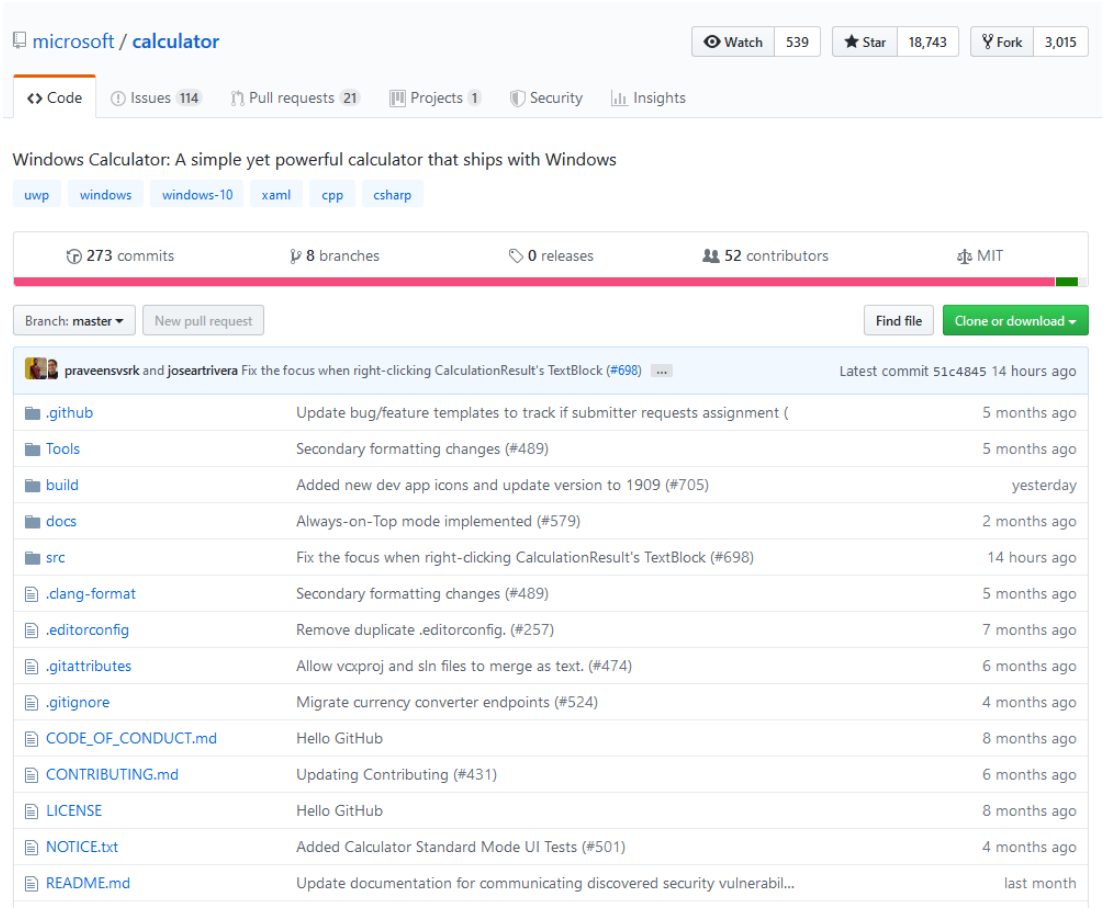
Copyright Notice

A large portion of the contents in the following pages come from Prof. Bjarne Stroustrup's slides.
The original contents could be found here.

Lingering issues from Lecture 6

```
1  Token Token_stream::get()  // read a Token from the Token_stream
2  {
3      if (full) { full=false; return buffer; }  // check if we already have a
        Token ready
4      char ch;
5      cin >> ch; // note that >> skips whitespace (space, newline, tab, etc.)
6      switch (ch) {
7      case '(': case ')': case ';': case 'q': case '+': case '-': case '*': case '
        /':
8          return Token{ch}; // let each character represent itself
9      case '.':
10     case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '
        7': case '8': case '9':
11         { cin.putback(ch); // put digit back into the input stream
12         double val;
13         cin >> val; // read a floating-point number
14         return Token{number ,val};
15         }
16     default:
17         error("Bad token");
18     }
19 }
20
```

How is Windows Calculator implemented?



microsoft / calculator

Watch 539 | Star 18,743 | Fork 3,015

Code | Issues 114 | Pull requests 21 | Projects 1 | Security | Insights

Windows Calculator: A simple yet powerful calculator that ships with Windows

uwp windows windows-10 xaml cpp csharp

273 commits | 8 branches | 0 releases | 52 contributors | MIT

Branch: master | New pull request | Find file | Clone or download

praveensvsrk and joseartrivera Fix the focus when right-clicking CalculationResult's TextBlock (#698) ... Latest commit 51c4845 14 hours ago

.github	Update bug/feature templates to track if submitter requests assignment (	5 months ago
Tools	Secondary formatting changes (#489)	5 months ago
build	Added new dev app icons and update version to 1909 (#705)	yesterday
docs	Always-on-Top mode implemented (#579)	2 months ago
src	Fix the focus when right-clicking CalculationResult's TextBlock (#698)	14 hours ago
.clang-format	Secondary formatting changes (#489)	5 months ago
.editorconfig	Remove duplicate .editorconfig. (#257)	7 months ago
.gitattributes	Allow vcxproj and sln files to merge as text. (#474)	6 months ago
.gitignore	Migrate currency converter endpoints (#524)	4 months ago
CODE_OF_CONDUCT.md	Hello GitHub	8 months ago
CONTRIBUTING.md	Updating Contributing (#431)	6 months ago
LICENSE	Hello GitHub	8 months ago
NOTICE.txt	Added Calculator Standard Mode UI Tests (#501)	4 months ago
README.md	Update documentation for communicating discovered security vulnerabil...	last month

A quick recap after the break

What is the objective (programming part)?

A quick recap after the break

What is the objective (programming part)?

- Try to understand the essence behind the programming paradigm (in order to build a real software or to solve a real problem)

A quick recap after the break

What is the objective (programming part)?

- Try to understand the essence behind the programming paradigm (in order to build a real software or to solve a real problem)
- Lecture 1: Error handling
- Lecture 2: C++ in 4 slides
- Lecture 5 & 6: Write a simple program

A quick recap after the break

What is the objective (programming part)?

- Try to understand the essence behind the programming paradigm (in order to build a real software or to solve a real problem)
- Lecture 1: Error handling
- Lecture 2: C++ in 4 slides
- Lecture 5 & 6: Write a simple program
- Now what?

A quick recap after the break

What is the objective (programming part)?

- Try to understand the essence behind the programming paradigm (in order to build a real software or to solve a real problem)
- Lecture 1: Error handling
- Lecture 2: C++ in 4 slides
- Lecture 5 & 6: Write a simple program
- Now what?
- Generic programming in this lecture

What are the four keys in C++

What are the four keys in C++

- Map to hardware
- Classes
- Inheritance
- Parameterized types

What are the four keys in C++

- Map to hardware
- Classes *OOP*
- Inheritance *OOP*
- Parameterized types

What are the four keys in C++

- Map to hardware
- Classes *OOP*
- Inheritance *OOP*
- Parameterized types *GP*

What are the four keys in C++

- Map to hardware
- Classes *OOP*
- Inheritance *OOP*
- Parameterized types *GP*

Question: Let's think about OOP and GP. Are they compatible?

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"
- Why are we interested in GP?

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"
- Why are we interested in GP? Or what are the aims (or benefits) from the perspective of an end user?

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"
- Why are we interested in GP? Or what are the aims (or benefits) from the perspective of an end user?
  - Increase correctness

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"
- Why are we interested in GP? Or what are the aims (or benefits) from the perspective of an end user?
  - Increase correctness
  - Greater range of uses

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"
- Why are we interested in GP? Or what are the aims (or benefits) from the perspective of an end user?
  - Increase correctness
  - Greater range of uses
  - Better performance

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"
- Why are we interested in GP? Or what are the aims (or benefits) from the perspective of an end user?
  - Increase correctness
  - Greater range of uses
  - Better performance (?)

Generic programming

- Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. (Wikipedia)
- It is sometime called "lifting"
- Why are we interested in GP? Or what are the aims (or benefits) from the perspective of an end user?
  - Increase correctness
  - Greater range of uses
  - Better performance (?)
- In C++, this is technically implemented by using `template`

Example: summation

Summation over a double array

```
1 double sum(double array[], int n)
2 {
3     double s = 0;
4     for (int i = 0; i < n; ++i ) s = s + array[i];
5     return s;
6 }
7
```

Example: summation

Summation over a double array

```
1  double sum(double array[], int n)
2  {
3      double s = 0;
4      for (int i = 0; i < n; ++i ) s = s + array[i];
5      return s;
6  }
7
```

Summation over an integer list

```
1  struct Node { Node* next; int data; };
2  int sum(Node* first)
3  {
4      int s = 0;
5      while (first) {
6          s += first->data;
7          first = first->next;
8      }
9      return s;
10 }
11
```

Summation: abstraction

Now we want to abstract/lift/generalize this summation procedure, what should we do?

Summation: abstraction

Now we want to abstract/lift/generalize this summation procedure, what should we do?
It is equivalent to ask you to write a piece of pseudo-code.

Summation: abstraction

Now we want to abstract/lift/generalize this summation procedure, what should we do?
It is equivalent to ask you to write a piece of pseudo-code.

```
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
}
```

Summation: abstraction

Now we want to abstract/lift/generalize this summation procedure, what should we do?
It is equivalent to ask you to write a piece of pseudo-code.

```
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
}
```

This is called algorithm.

Summation: implement the algorithm

For this task, we only need three operations (on the data structure):

Summation: implement the algorithm

For this task, we only need three operations (on the data structure):

1. determine whether we are reaching the end (`not at end`)
2. `get value`
3. `get next data element`

Summation: implement the algorithm

For this task, we only need three operations (on the data structure):

1. determine whether we are reaching the end (`not at end`)
2. `get value`
3. `get next data element`

```
1  template<class Iter, class T>   // Iter should be an Input_iterator
2                                  // T should be something we can + and =
3  T sum(Iter first, Iter last, T s)   // T is the accumulator type
4  {
5      while (first!=last) {
6          s = s + *first;
7          ++first;
8      }
9      return s;
10 }
11
```

What is STL?

What is STL?

- STL is a *software library*.

What is STL?

- STL is a *software library*.
- It is part of the ISO C++ Standard Library

What is STL?

- STL is a *software library*.
- It is part of the ISO C++ Standard Library
- Specifically, STL provides an extensible framework dealing with data in a C++ program.

What is STL?

- STL is a *software library*.
- It is part of the ISO C++ Standard Library
- Specifically, STL provides an extensible framework dealing with data in a C++ program.
  - Include ∼10 containers and ∼60 algorithms

What is STL?

- STL is a *software library*.
- It is part of the ISO C++ Standard Library
- Specifically, STL provides an extensible framework dealing with data in a C++ program.
  - Include ∼10 containers and ∼60 algorithms
  - Other organizations provide more containers and algorithms in the style of the STL (e.g. boost, SGI, microsoft)

What do we mean here by data?

What do we mean here by data?

- Indeed, it refers to data structure

What do we mean here by data?

- Indeed, it refers to data structure in the form of containers

What do we mean here by data?

- Indeed, it refers to data structure in the form of containers
- Data structure is important, because it manifests the logical relationship between the elements.

What do we mean here by data?

- Indeed, it refers to data structure in the form of containers
- Data structure is important, because it manifests the logical relationship between the elements.
- Before we talk about containers in C++, let's recall what we are using in C for data structure?

What do we mean here by data?

- Indeed, it refers to data structure in the form of containers
- Data structure is important, because it manifests the logical relationship between the elements.
- Before we talk about containers in C++, let's recall what we are using in C for data structure?
- array (primitive)

What do we mean here by data?

- Indeed, it refers to data structure in the form of containers
- Data structure is important, because it manifests the logical relationship between the elements.
- Before we talk about containers in C++, let's recall what we are using in C for data structure?
- array (primitive)
- Why do we dislike array?

What do we mean here by data?

- Indeed, it refers to data structure in the form of containers
- Data structure is important, because it manifests the logical relationship between the elements.
- Before we talk about containers in C++, let's recall what we are using in C for data structure?
- array (primitive)
- Why do we dislike array?
- We have to reinvent the wheel all the time

STL Containers

- Sequence containers
  - Array
  - Vector
  - Deque (Double-ended queue)
  - List
  - Forward-list

STL Containers

- Sequence containers
  - Array
  - Vector
  - Deque (Double-ended queue)
  - List
  - Forward-list
- Associative containers
  - Set/Multiset
  - Map/Multiamp

STL Containers

- Sequence containers
  - Array
  - Vector
  - Deque (Double-ended queue)
  - List
  - Forward-list
- Associative containers
  - Set/Multiset
  - Map/Multiamp
- Unordered containers
  - Unordered set/multiset
  - Unordered map/multiamp
  - HashTable

STL Containers

- Sequence containers
  - Array
  - Vector
  - Deque (Double-ended queue)
  - List
  - Forward-list
- Associative containers
  - Set/Multiset
  - Map/Multiamp
- Unordered containers
  - Unordered set/multiset
  - Unordered map/multiamp
  - HashTable

Container class templates

Sequence containers:

array	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers: http://www.cplusplus.com/reference/stl/

unordered_set	Unordered Set (class template)
unordered_multiset	Unordered Multiset (class template)
unordered_map	Unordered Map (class template)
unordered_multimap	Unordered Multimap (class template)

What do we mean here by dealing with?

What do we mean here by dealing with?

- Initialize
- Collect
- Organize
- Retrieve
- Add
- Delete
- ...

What do we mean here by dealing with?

- Initialize
- Collect
- Organize
- Retrieve
- Add
- Delete
- ...

There are some universal algorithms that are used to process the data
Basic operations for those containers are identical

Design ideals

Separation of concerns

Design ideals

Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators

Design ideals

Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
  - What do these sentences mean?

Design ideals

Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
  - What do these sentences mean?
  - Each container has its own iterator types

Design ideals

Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
  - What do these sentences mean?
  - Each container has its own iterator types
  - Two levels of abstraction/lifting/generalization

Design ideals

Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
  - What do these sentences mean?
  - Each container has its own iterator types
  - Two levels of abstraction/lifting/generalization: algorithm→iterator

Design ideals

Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
  - What do these sentences mean?
  - Each container has its own iterator types
  - Two levels of abstraction/lifting/generalization: algorithm→iterator; iterator(data structure)→ data

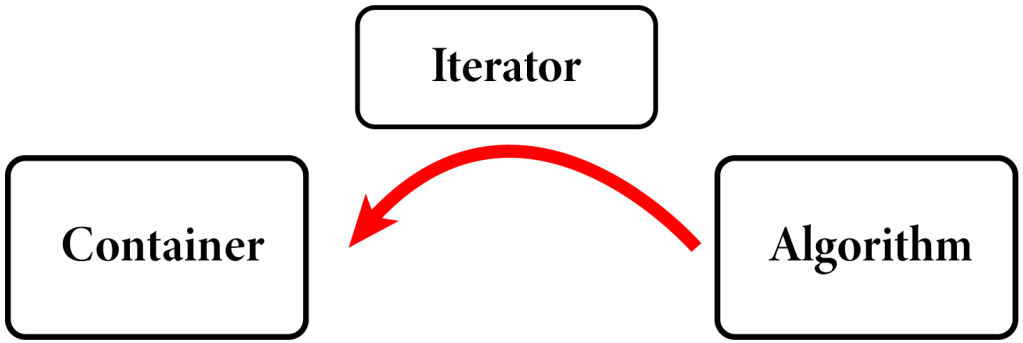
Design ideals

Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
  - What do these sentences mean?
  - Each container has its own iterator types
  - Two levels of abstraction/lifting/generalization: algorithm→iterator; iterator(data structure)→ data
  - Different from pointer

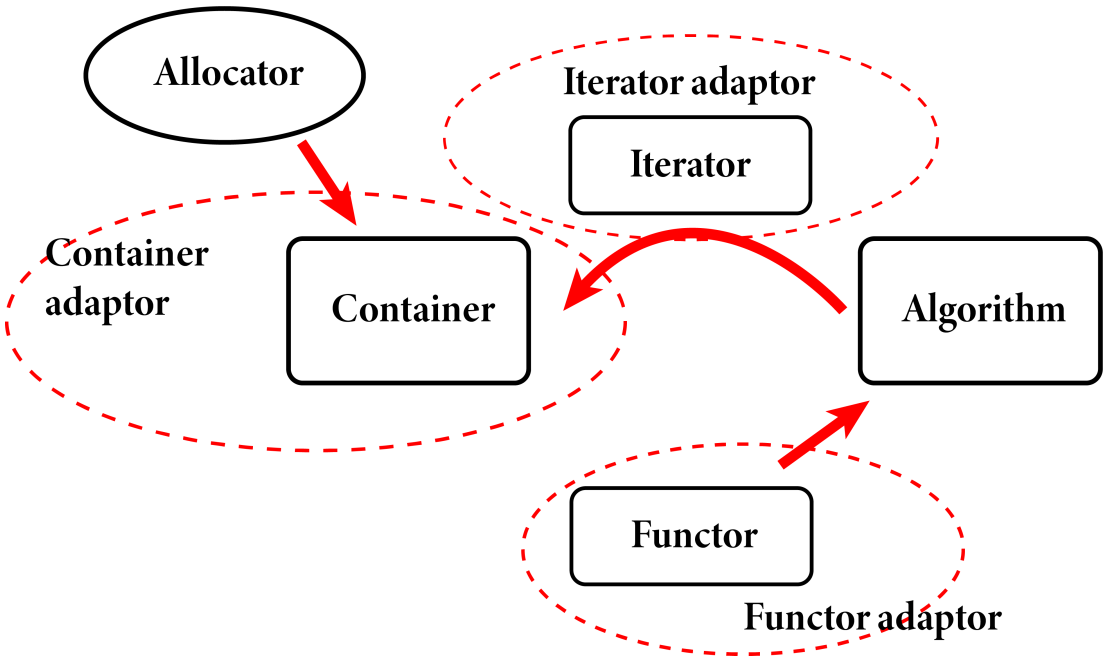
Basic model of STL

- Container
- Iterator
- Algorithm



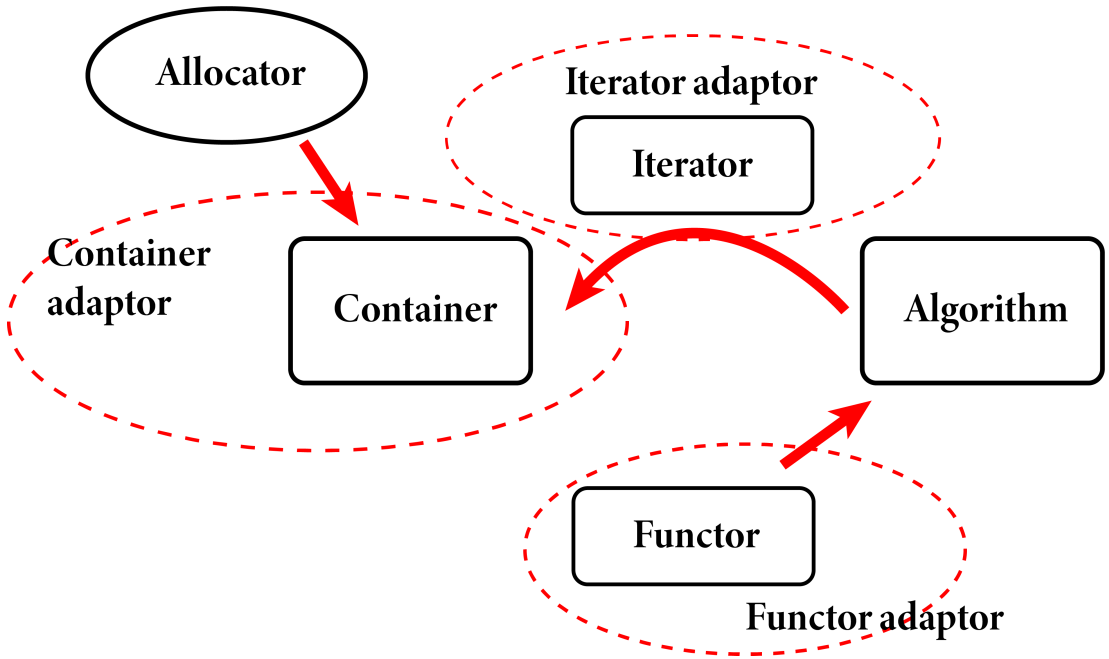
Complete model of STL

- Container
- Iterator
- Algorithm
- Allocator
- Adaptor
- Functor



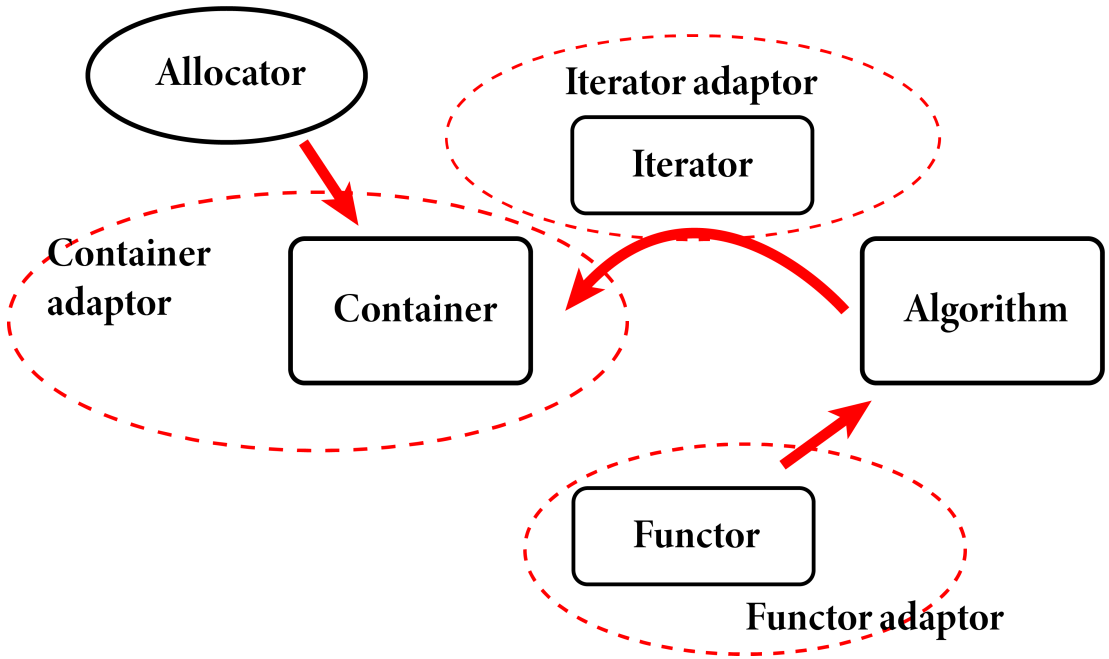
Complete model of STL

- Container
- Iterator
- Algorithm
- Allocator
- Adaptor
  - Container adaptor: pile, stack, etc
- Functor

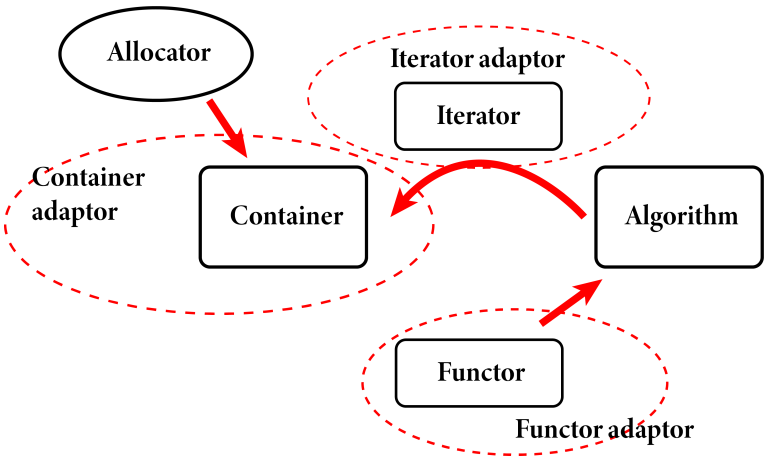


Complete model of STL

- Container
- Iterator
- Algorithm
- Allocator
- Adaptor
  - Container adaptor: pile, stack, etc
  - Does not support iterator
- Functor



Example




```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
        6);
11     cout << count_if(vi.begin(), vi.end(),
        not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Example (Cont'd)

- Container:
- Allocator:
- Algorithm:
- Iterator:
- Functor:
- Adaptors:

```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
        6);
11     cout << count_if(vi.begin(), vi.end(),
        not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Example (Cont'd)

- Container: vector
- Allocator:
- Algorithm:
- Iterator:
- Functor:
- Adaptors:

```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
       6);
11     cout << count_if(vi.begin(), vi.end(),
       not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Example (Cont'd)

- Container: vector
- Allocator: allocator<int>
- Algorithm:
- Iterator:
- Functor:
- Adaptors:

```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
       6);
11     cout << count_if(vi.begin(), vi.end(),
       not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Example (Cont'd)

- Container: vector
- Allocator: allocator<int>
- Algorithm: count_if
- Iterator:
- Functor:
- Adaptors:

```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
       6);
11     cout << count_if(vi.begin(), vi.end(),
       not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Example (Cont'd)

- Container: vector
- Allocator: allocator<int>
- Algorithm: count_if
- Iterator: vi.begin(), vi.end()
- Functor:
- Adaptors:

```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
       6);
11     cout << count_if(vi.begin(), vi.end(),
       not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Example (Cont'd)

- Container: vector
- Allocator: allocator<int>
- Algorithm: count_if
- Iterator: vi.begin(), vi.end()
- Functor: less<int>()
- Adaptors:

```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
       6);
11     cout << count_if(vi.begin(), vi.end(),
       not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Example (Cont'd)

- Container: vector
- Allocator: allocator<int>
- Algorithm: count_if
- Iterator: vi.begin(), vi.end()
- Functor: less<int>()
- Adaptors: bind2nd, not1

```
1  #include <vector>
2  #include <algorithm>
3  #include <functional>
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      int ia[6] = {27, 210, 12, 47, 109, 83};
10     vector<int, allocator<int>> vi(va, ia +
       6);
11     cout << count_if(vi.begin(), vi.end(),
       not1(bind2nd(less<int>(), 40)));
12     return 0;
13 }
14
```

Iterators

An iterator is an object (like a pointer) that points to an element inside the container.

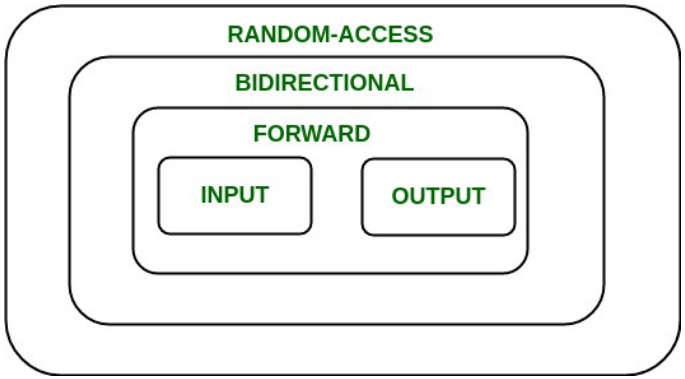
Iterators

An iterator is an object (like a pointer) that points to an element inside the container.
It can:

- Point to elements
- Iterate through container (via ++)
- But, not all iterators are born equal

Types of iterators

Iterators are classified into five categories depending on the functionality they implement:



Input and output iterators

Input

Input iterators are iterators that can be used in sequential input operations

Input and output iterators

Input

Input iterators are iterators that can be used in sequential input operations

- Input: from the program's perspective

Input and output iterators

Input

Input iterators are iterators that can be used in sequential input operations

- Input: from the program's perspective
- Sequential:

Input and output iterators

Input

Input iterators are iterators that can be used in sequential input operations

- Input: from the program's perspective
- Sequential:
  1. "++" has to be supported

Input and output iterators

Input

Input iterators are iterators that can be used in sequential input operations

- Input: from the program's perspective
- Sequential:
  1. "++" has to be supported
  2. Each element could only be dereferenced once

Input and output iterators

Input

Input iterators are iterators that can be used in sequential input operations

- Input: from the program's perspective
- Sequential:
  1. "++" has to be supported
  2. Each element could only be dereferenced once

Can you give an example for input iterator?

Input and output iterators

Input

Input iterators are iterators that can be used in sequential input operations

- Input: from the program's perspective
- Sequential:
  1. "++" has to be supported
  2. Each element could only be dereferenced once

Can you give an example for input iterator?

Output

Output iterators are ...

Forward iterators

Forward iterators are iterators that can be used to access the sequence of elements in a range in the direction that goes from its beginning towards its end.

Forward iterators

Forward iterators are iterators that can be used to access the sequence of elements in a range in the direction that goes from its beginning towards its end.

- Both read and write are supported

Forward iterators

Forward iterators are iterators that can be used to access the sequence of elements in a range in the direction that goes from its beginning towards its end.

- Both read and write are supported
- Range:

Forward iterators

Forward iterators are iterators that can be used to access the sequence of elements in a range in the direction that goes from its beginning towards its end.

- Both read and write are supported
- Range: support multiple dereferencing of a single element

Forward iterators

Forward iterators are iterators that can be used to access the sequence of elements in a range in the direction that goes from its beginning towards its end.

- Both read and write are supported
- Range: support multiple dereferencing of a single element
- Direction: only "++" are supported

Forward iterators

Forward iterators are iterators that can be used to access the sequence of elements in a range in the direction that goes from its beginning towards its end.

- Both read and write are supported
- Range: support multiple dereferencing of a single element
- Direction: only "++" are supported

Example: forward_list

Bidirectional iterators

Bidirectional iterators are iterators that can be used to access the sequence of elements in a range in both directions (towards the end and towards the beginning).

Random-access iterators

Random-access iterators are iterators that can be used to access elements at an arbitrary offset position relative to the element they point to.

Random-access iterators

Random-access iterators are iterators that can be used to access elements at an arbitrary offset position relative to the element they point to.

Question:

- Is a pointer an iterator?

Random-access iterators

Random-access iterators are iterators that can be used to access elements at an arbitrary offset position relative to the element they point to.

Question:

- Is a pointer an iterator?
- What type of iterator is most resemblant to a pointer?

Quick summary

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

Functors

Functors are objects that can be treated as though they are a function or function pointer.

Functors

Functors are objects that can be treated as though they are a function or function pointer.

Function version

```
1 bool odd(int i) { return i%2; }
2 odd(7);
3
```

Functors

Functors are objects that can be treated as though they are a function or function pointer.

Function version

```
1 bool odd(int i) { return i%2; }
2 odd(7);
3
```

Functor version

```
1 struct Odd {
2     bool operator()(int i) const { return i%2; }
3 };
4 Odd odd; // make an object odd of type Odd
5 odd(7);
6
```

Functors

Functors are objects that can be treated as though they are a function or function pointer.

Function version

```
1 bool odd(int i) { return i%2; }
2 odd(7);
3
```

Functor version

```
1 struct Odd {
2     bool operator()(int i) const { return i%2; }
3 };
4 Odd odd; // make an object odd of type Odd
5 odd(7);
6
```

What's the difference?

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor?

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor? It is an object!

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor? It is an object!
- Benefits:
  1. Statefulness:

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor? It is an object!
- Benefits:
  1. Statefulness: it can save current state (or data)

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor? It is an object!
- Benefits:
  1. Statefulness: it can save current state (or data)
  2. Performance:

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor? It is an object!
- Benefits:
  1. Statefulness: it can save current state (or data)
  2. Performance: inline

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor? It is an object!
- Benefits:
  1. Statefulness: it can save current state (or data)
  2. Performance: inline
  3. Parameterization

Why we prefer functors?

Both versions fulfill the requirements. Why bother?

- What is functor? It is an object!
- Benefits:
  1. Statefulness: it can save current state (or data)
  2. Performance: inline
  3. Parameterization

```
1  cout << count_if(vi.begin(), vi.end(), not1(bind2nd(less<int>(), 40)));
2
```

std::less

C++11 version

```
1 template <class T> struct less {
2     bool operator() (const T& x, const T& y) const {return x<y;}
3     typedef T first_argument_type;
4     typedef T second_argument_type;
5     typedef bool result_type;
6 };
7
```

For this specific case, `less` is also known as a predicate.

std::less

C++11 version

```
1 template <class T> struct less {
2     bool operator() (const T& x, const T& y) const {return x<y;}
3     typedef T first_argument_type;
4     typedef T second_argument_type;
5     typedef bool result_type;
6 };
7
```

For this specific case, `less` is also known as a predicate.

- A predicate is a function or a function object that takes arguments and returns a bool

Back to the example

- `bind2nd(less<int>(), 40)`

Back to the example

- `bind2nd(less<int>(), 40)` : functor adaptor

Back to the example

- `bind2nd(less<int>(), 40)` : functor adaptor
- `not1(bind2nd(less<int>(), 40))`

Back to the example

- `bind2nd(less<int>(), 40)` : functor adaptor
- `not1(bind2nd(less<int>(), 40))` : another functor adaptor

Back to the example

- `bind2nd(less<int>(), 40)` : functor adaptor
- `not1(bind2nd(less<int>(), 40))` : another functor adaptor
-
```
count_if(vi.begin(), vi.end(),
  not1(bind2nd(less<int>(), 40)))
```

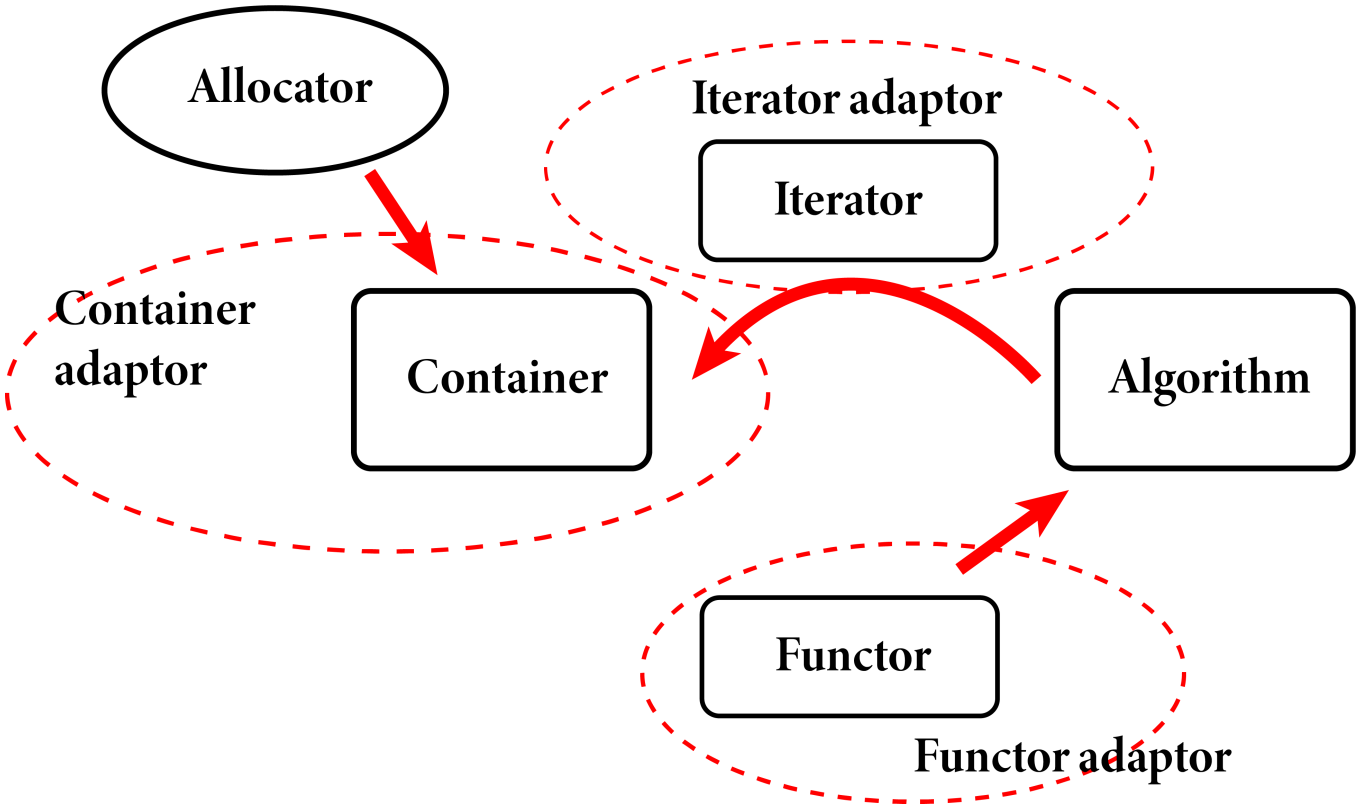
Back to the example

- `bind2nd(less<int>(), 40)` : functor adaptor
- `not1(bind2nd(less<int>(), 40))` : another functor adaptor
-
```
count_if(vi.begin(), vi.end(),
  not1(bind2nd(less<int>(), 40)))
```

Back to the example

- `bind2nd(less<int>(), 40)` : functor adaptor
- `not1(bind2nd(less<int>(), 40))` : another functor adaptor
-
```
count_if(vi.begin(), vi.end(),
  not1(bind2nd(less<int>(), 40)))
```

Now we know that functors provide us with a lot of flexibilities. But how is this connected to the bigger picture?


Allocator
Iterator adaptor
Iterator
Container
adaptor
Container
Algorithm
Functor
Functor adaptor

Policy parameterization

Whenever you have a useful algorithm, you eventually want to parameterize it by a "policy".

Policy parameterization

Whenever you have a useful algorithm, you eventually want to parameterize it by a "policy".
For example, we need to parameterize `sort` by the comparison criteria

```
1  struct Record {
2      string name; // standard string for ease of use
3      char addr[24]; // old C-style string to match database layout
4      // ...
5  };
6
7  vector<Record> vr;
8  // ...
9  sort(vr.begin(), vr.end(), Cmp_by_name()); // sort by name
10 sort(vr.begin(), vr.end(), Cmp_by_addr()); // sort by addr
11
```

Summary

- We explained the idea of generic programming with the help of STL.

Summary

- We explained the idea of generic programming with the help of STL.
- Core ideal: separation of concerns

Summary

- We explained the idea of generic programming with the help of STL.
- Core ideal: separation of concerns
- Back to the question raised at the beginning: does GP conflict with OOP?

Summary

- We explained the idea of generic programming with the help of STL.
- Core ideal: separation of concerns
- Back to the question raised at the beginning: does GP conflict with OOP?
- Yes and No

Summary

- We explained the idea of generic programming with the help of STL.
- Core ideal: separation of concerns
- Back to the question raised at the beginning: does GP conflict with OOP?
- Yes and No
- Stroustrup: "C++ is a general-purpose programming language with a bias towards systems programming that:"
  1. is a better C
  2. supports data abstraction
  3. supports object-oriented programming
  4. supports generic programming.