Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 21/24

Session 21. More on Standard Template Library: Introduction to Generic Algorithms and No Raw Loop

- Introduction to Generic Algorithms
- Algorithms and Lifting: An example
- Iota as a Generic algorithm
- The classification of Generic Algorithms
- Simple STL algorithms: The find family algorithms and all_of/none_of/any_of family algorithms
- More STL algorithms: For_each, Adjacent_find and Reverse algorithms
- Predicates, Function Objects, and Lambda functions
- Writing a few programs using generic algorithms
- Q&A

150 min (incl. Q & A)

Algorithms

Algorithms

- In mathematics and computer science, an *algorithm* is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing, and automated reasoning tasks.

Wikipedia contributors. (2018, September 7). Algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved 19:20, September 9, 2018, from https://en.wikipedia.org/w/index.php?title=Algorithm&oldid=858547889

Algorithms

- In mathematics and computer science, an *algorithm* is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing, and automated reasoning tasks.

Wikipedia contributors. (2018, September 7). Algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved 19:20, September 9, 2018, from https://en.wikipedia.org/w/index.php?title=Algorithm&oldid=858547889

Algorithms

- In mathematics and computer science, an *algorithm* is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing, and automated reasoning tasks.

Wikipedia contributors. (2018, September 7). Algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved 19:20, September 9, 2018, from https://en.wikipedia.org/w/index.php?title=Algorithm&oldid=858547889

- Algorithms, both the standard-library algorithms and the user's own ones, are important:

  - Each name a specific operation, documents an interface, and specifies semantics.
  - Each can be widely used and known by many programmers.

  - Bjarne Stroustrup

Algorithms

- In mathematics and computer science, an *algorithm* is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing, and automated reasoning tasks.

Wikipedia contributors. (2018, September 7). Algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved 19:20, September 9, 2018, from https://en.wikipedia.org/w/index.php?title=Algorithm&oldid=858547889

- Algorithms, both the standard-library algorithms and the user's own ones, are important:
  - Each name a specific operation, documents an interface, and specifies semantics.
  - Each can be widely used and known by many programmers.

  \- Bjarne Stroustrup

- Algorithm vs. Random-code

Algorithms

- In mathematics and computer science, an *algorithm* is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing, and automated reasoning tasks.

Wikipedia contributors. (2018, September 7). Algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved 19:20, September 9, 2018, from https://en.wikipedia.org/w/index.php?title=Algorithm&oldid=858547889

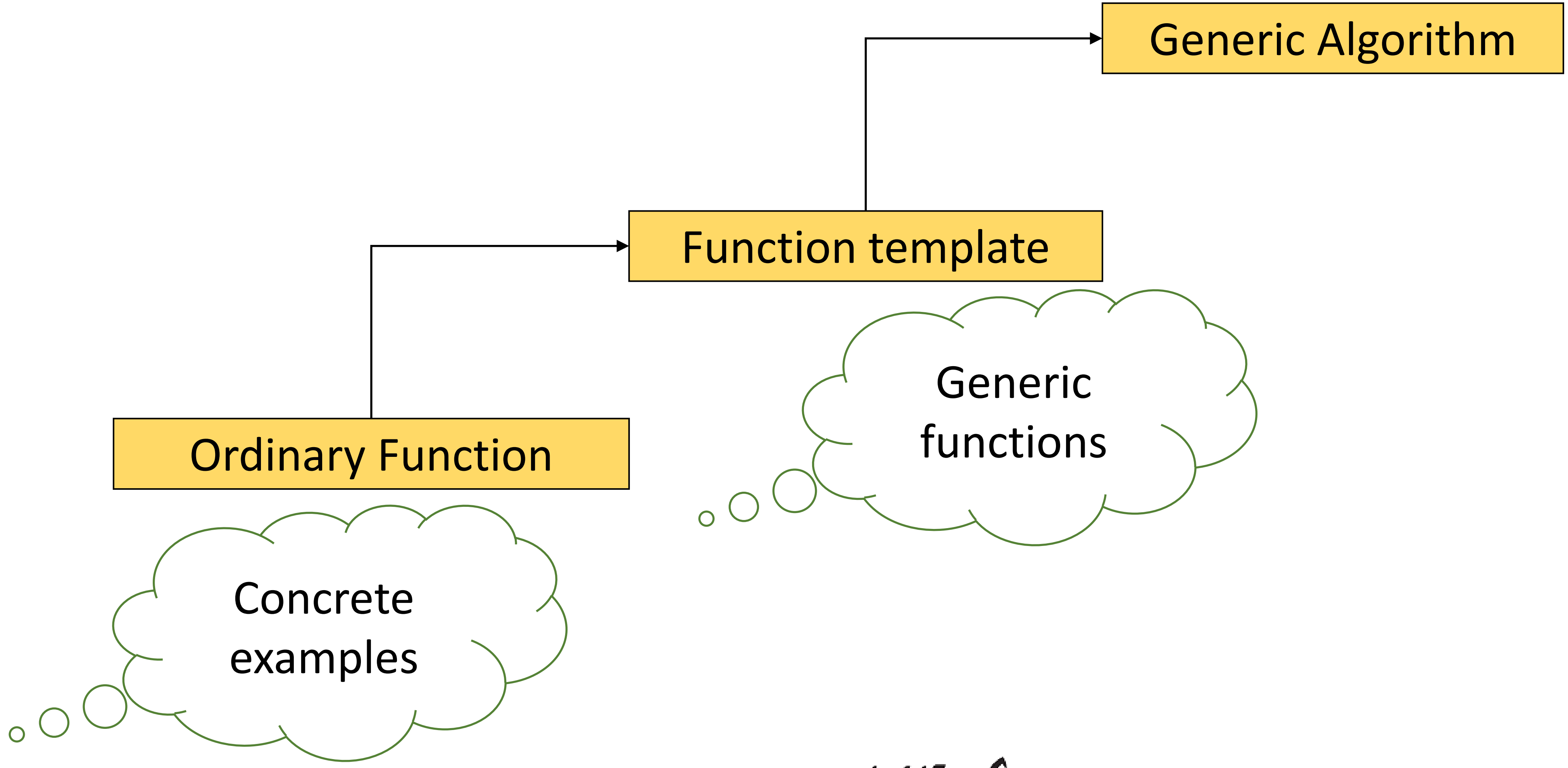
- Algorithms, both the standard-library algorithms and the user's own ones, are important:
  - Each name a specific operation, documents an interface, and specifies semantics.
  - Each can be widely used and known by many programmers.

  \- Bjarne Stroustrup

- Algorithm vs. Random-code
- Generic algorithms vs. Hand-written loop

Generic Algorithms

Algorithms and Lifting



Algorithms Lifting- *very* simple example

- Fill incremental algorithm:

- A loop

```
int a[10];
for (int i = 0; i != 10; ++i)
    a[i] = i;
```

```
double b[10];
for (int i = 0; i != 10; ++i)
    a[i] = static_cast<double>(i);
```

- An ordinary function

```
void fill_incr(int*, int*)
{
    for (int i = 0; i != 10; ++i)
        a[i] = i;
}
// user code
int a[10];
fill_incr(a, a + 10);
```

```
struct Node {
    Node* next;
    double data;
};
void fill_incr(Node* first, Node* last)
{
    int i = 0;
    while (first != last) {
        first->data = static_cast<double>(i);
        ++i;
        first = first->next;
    }
}
// user code
// make singly linked list and pass the two pointers to nodes
```

- They are just two implementations of *fill incremental* algorithm.

fill incremental algorithm

- Pseudo-code

```
void fill_incr(container)
{
    T i = 0;
    while (not at end) {
        assign i to current element of container
        go to next data element
        ++i;
    }
}
```

- Operations related to Container data structure:
  - Not at end
  - Get current element position
  - Go to next element

- Operations related to Actual data :
  - Initialize to zero
  - Assignment
  - Add

fill incremental algorithm cont.

- Pseudo-code

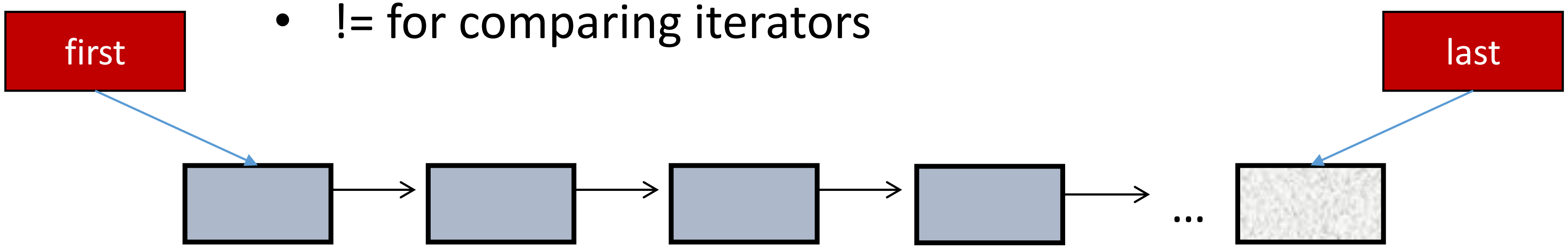
```
void fill_incr(container)
{
    T i = 0;
    while (not at end) {
        assign i to current element of container
        go to next data element
        ++i;
    }
}
```

- Template function

```
// complete STL-like code
template<typename Iter, typename Val>
void fill_incr(Iter first, Iter last)
{
    Val v = 0;
    while (first != last) {
        *first++ = v++;
    }
}
```

- Iterator model

- * for accessing current value
- ++ for moving forward to the next element
- != for comparing iterators



fill incremental algorithm- further generalization

- Initial value

```
// complete STL-like code
template<typename Iter, typename Val>
void fill_incr(Iter first, Iter last, Val v)
{
    while (first != last) {
        *first++ = v++;
    }
}
```

- Forward iterator concept

```
// complete STL-like code
template<typename ForwardIter, typename Val>
void fill_incr(ForwardIter first, ForwardIter last, Val v)
{
    while (first != last) {
        *first++ = v++;
    }
}
```

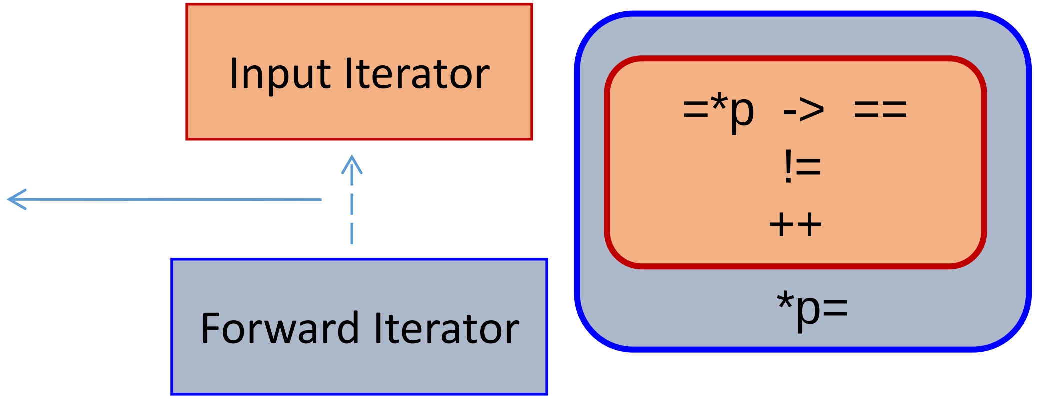
Iota

- iota is a new generic algorithm in C++11.
- More than 30 new generic algorithms were added to C++11.

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
```

- *Requires*: T shall be convertible to ForwardIterator's value type. The expression ++val, where val has type T, shall be well formed.
- *Effects*: For each element referred to by the iterator i in the range [first,last), assigns *i = value and increments value as if by ++value.
- *Complexity*: Exactly last - first increments and assignments.

It's not inheritance!



iota and fundamental types

- Here it is a complete program that uses iota with fundamental types:

```
// iota_practice.c++
#include <numeric>
#include <vector>
#include <list>
#include <iostream>
#include <array>
int main()
{
  using namespace std;
  vector<int> vi(1000000);
  list<double> lst(1000000);
  array<char, 26> lower_case; // array is new container
  vector<long long> vll(10); // long long is a new fundamental data type
  iota(vi.begin(), vi.end(), 0); // 0, 1, 2, ..., 999999
  iota(lst.begin(), lst.end(), 0.0); // 0.0, 1.0, 2.0, … 999999.0
  iota(lower_case.begin(), lower_case.end(), 'a'); // 'a', 'b', … 'z'
  iota(vll.begin(), vll.end(), 0LL); // 0LL, 1LL, 2LL, … 9LL
  for (auto c : lower_case) cout << c << ‘ ‘; // range-based for loop
  cout << ‘\n’;
  return 0;
}
```

- C++11: array container, long long data type, range-based for loop and auto

iota cont.

- A typical/likely implementation of iota

```
namespace std {
  template<class ForwardIterator, class TYPE_>
  void iota(ForwardIterator first, ForwardIterator last, TYPE_ t)
  {
     for (auto it = first; it != last; ++it, ++t) // prefix ++
       *it = t;
  }
}
```

Iterator requirement

Template type requirement

- Pre increment vs. Post increment

- The value of ++x, is the new (that is, incremented) value of x. The value of x++, is the old value of x.

```
y = ++x; // y = (x += 1) or ++x; y = x;
y = x++; // y = (t = x, x += 1; t) or t = x; x++; y = t;
```

- ++x means to increment x and return the new value, while x++ is to increment x and return the old value. iota uses prefix ++ increment.

iota and user-defined types

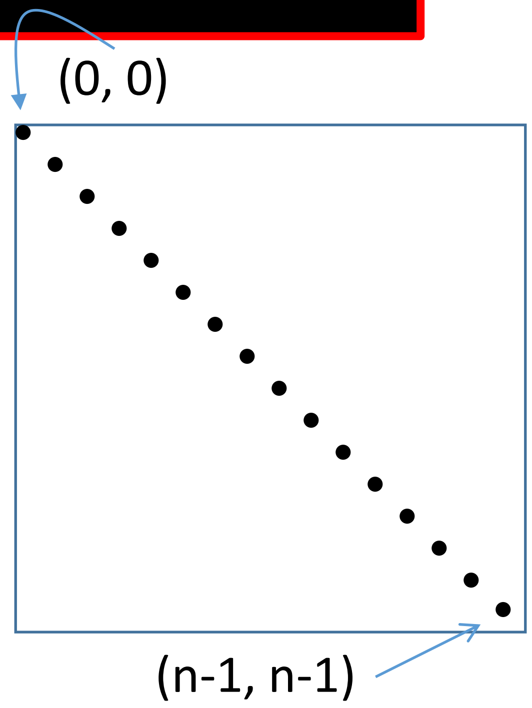
- Rational numbers, 2D points and SE Closing prices, …

1.

```
// rational.h
class Rational { // non-normalized run-time rational number
  int numerator_, denominator_;
public:
  Rational() : numerator_{0}, denominator_{1} {} // new initialization
                                                  // syntax
  // other ctor(s), relop(s), other member functions
  Rational& operator++() { numerator_ += denominator_; return *this; }
};
```

2.

```
// point.h
class Point { // 2D point
  int x_, y_;
public:
  Point() : x_{0}, y_{0} {}
  // ctor(s), graphics-related member functions
  Point& operator++() { ++x; ++y; return *this; }
};
```



iota and user-defined types

3.

```
// price.h
class price_stepper { // price stepper for a typical securities exchange
  double price_;
public:
  static const double STEP{0.05}; // new syntax for uniform initialization
  static const double FACE_VALUE{1000.00};
  price_stepper() : price_{FACE_VALUE} {} // default ctor
  price_stepper(double price) : price_{price} {}
  operator double() const { return price_; }
  price_stepper& operator++() { price_ += STEP; return *this; }
};
```

⋮

4.

```
// gadget.h
class FuturisticGadget { // An Unidentified futuristic gadget: year 2050
public:
 FuturisticGadget& operator++() { /* … */ }
};
```

ألفا
ALPHA

iota and user-defined data types

- Complete program

```
#include <array>
#include <numeric>
#include <vector>
#include <list>
#include <iostream>
#include “point.h”
#include “rational.h”
#include “price.h”
#include “gadget.h”

int main()
{
 using namespace std;
  array<Point> vp(10000);
  list<Rational> lr(100000);
  vector<price_stepper> p_list(100000);
  forward_list<FuturisticGadget> fg_list(5);
  iota(vp.begin(), vp.end(), Point()); // [point(0, 0), point(9999, 9999)]
  iota(lr.begin(), lr.end(), Rational()); // [0/1, 1/1, 2/1, …, 100000/1]
  // …
  return 0;
}
```

- Template duck typing:

If it looks like a duck, walks like a duck, and quacks like a duck..., so it’s a *duck*.

STL Algorithms

STL Algorithms

STL Algorithms

Generic algorithms

STL Algorithms

Generic algorithms

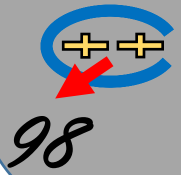
for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find, count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n, generate, generate_n, remove, remove_copy, remove_if, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element, lower_bound, upper_bound, equal_range, binary_search, partition, stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference, set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,

ALPHA آلفا

STL Algorithms

Generic algorithms

for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find,
count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n,
generate, generate_n, remove, remove_copy, remove_if, remove_copy_if,
unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element,
lower_bound, upper_bound, equal_range, binary_search, partition,
stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference,
set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,



STL Algorithms

Generic algorithms

for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find,
count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n,
generate, generate_n, remove, remove_copy, remove_if, remove_copy_if,
unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element,
lower_bound, upper_bound, equal_range, binary_search, partition,
stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference,
set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,

98

all_of, any_of, none_of,
copy_if, copy_n, move, move_backward,
is_sorted, is_sorted_until, is_partitioned, partition_copy, partition_point,
iota

STL Algorithms

Generic algorithms

for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find,
count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n,
generate, generate_n, remove, remove_copy, remove_if, remove_copy_if,
unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element,
lower_bound, upper_bound, equal_range, binary_search, partition,
stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference,
set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,

C++98

all_of, any_of, none_of,
copy_if, copy_n, move, move_backward,
is_sorted, is_sorted_until, is_partitioned, partition_copy, partition_point,
iota

C++11

ALPHA آلفا

STL Algorithms

Generic algorithms

for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find,
count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n,
generate, generate_n, remove, remove_copy, remove_if, remove_copy_if,
unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element,
lower_bound, upper_bound, equal_range, binary_search, partition,
stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference,
set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,

98

all_of, any_of, none_of,
copy_if, copy_n, move, move_backward,
is_sorted, is_sorted_until, is_partitioned, partition_copy, partition_point,
iota

11

for_each_n, sample,
reduce, transform_reduce, exclusive_scan, inclusive_scan,
transform_ _scan, transfor_inclusive_scan

STL Algorithms

Generic algorithms

for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find,
count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n,
generate, generate_n, remove, remove_copy, remove_if, remove_copy_if,
unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element,
lower_bound, upper_bound, equal_range, binary_search, partition,
stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference,
set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,

C++98

all_of, any_of, none_of,
copy_if, copy_n, move, move_backward,
is_sorted, is_sorted_until, is_partitioned, partition_copy, partition_point,
iota

C++11

for_each_n, sample,
reduce, transform_reduce, exclusive_scan, inclusive_scan,
transform_exclusive_scan, transfor_inclusive_scan

C++17

STL Algorithms

C++98 → ~ 60 algorithms

C++11 → ~ 20 algorithms

C++17 → ~ 10 algorithms

Generic algorithms

for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find,
count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n,
generate, generate_n, remove, remove_copy, remove_if, remove_copy_if,
unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element,
lower_bound, upper_bound, equal_range, binary_search, partition,
stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference,
set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,

98

all_of, any_of, none_of,
copy_if, copy_n, move, move_backward,
is_sorted, is_sorted_until, is_partitioned, partition_copy, partition_point,
iota

11

17

for_each_n, sample,
reduce, transform_reduce, exclusive_scan, inclusive_scan,
transform_ _scan, transfor_inclusive_scan

STL Algorithms

C++98 → ~ 60 algorithms

C++11 → ~ 20 algorithms

C++17 → ~ 10 algorithms

- Non-modifying algorithms
- Mutating algorithms
- Sorting, Searching and related algorithms
- Set operations
- Generalized numeric algorithms

Generic algorithms

98

for_each, find, find_if , find_if_not, find_end, find_first_of, adjacent_find, count, mismatch, equal, is_permutation, search, search_n,
copy, copy_backward, swap_ranges,
transform, replace, replace_if, replace_copy, replace_copy_if, fill, fill_n, generate, generate_n, remove, remove_copy, remove_if, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, shuffle,
sort, stable_sort, partial_sort, partial_sort_copy, nth_element, lower_bound, upper_bound, equal_range, binary_search, partition, stable_partition, merge, inplace_merge,
Includes, set_union, set_intersection, set_difference, set_symmetric_difference,
accumulate, inner_product, partial_sum, adjacent_diiference,

11

all_of, any_of, none_of,
copy_if, copy_n, move, move_backward,
is_sorted, is_sorted_until, is_partitioned, partition_copy, partition_point,
iota

17

for_each_n, sample,
reduce, transform_reduce, exclusive_scan, inclusive_scan, transform_[illegible]_scan, transfor_inclusive_scan

STL Algorithms classification

- The C++ committee standardization draft classifies the STL algorithms in 6 categories:
- STL algorithms

1. Non-modifying sequence operations
2. mutating sequence operations
3. Sorting and related functions

   3.1 Sorting

   3.1 Binary search

4. Set operations on sorted structures
5. Heap operations
6. Generalized numeric operations

STL Algorithms

- A pre-built library of general-purpose algorithms designed to solve specific problems.
- There are about 90 standard algorithms defined in <algorithm>.
- All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types.
- All algorithms are generic and function template.
- They operate on *sequences* defined by a pair of iterators (for input) or single iterator (for output). Ex. find and copy
- We have a collection of family algorithms: find family algorithms, for_each family algorithms, copy family algorithms, accumulate family algorithms, …
- STL algorithms show the power of function name overloading.
- Almost all algorithms have the parallel overload version.
- The complexity: O(n), O(log n), O(n * log(n)), …
- Other algorithms: Heap operations, minimum & maximum, clamp, lexicographical comparison, permutation generation, mathematical functions f[illegible]ing types, and C libraries.

Non-modifying algorithms

- A non-modifying algorithm just reads the values of elements of its input sequences; it does not rearrange the sequence and does not change the values of the elements.
  - Do not modify the input sequence.
  - Do not emit a result sequence.
  - *Algorithm* will not cause side-effects in input sequence.
  - *Function object*, if present, may cause side-effects by modifying itself, the sequence (in certain cases, e.g. for_each), or its environment.

the Find family algorithms

the Find family algorithms

- find

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, T value);
```

the Find family algorithms

- find

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, T value);
```

- find_if

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

the Find family algorithms

- find

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, T value);
```

- find_if

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

- find_if_not

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pred);
```

the Find family algorithms

- find

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, T value);
```

- find_if

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

- find_if_not

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last, Predicate pred);
```

the Find algorithm- details

the Find algorithm- details

- *Returns:* The first iterator i in the range [first,last) for which *i == value. Returns last if no such iterator is found.
- *Complexity:* At most last - first applications of the corresponding predicate → $\mathcal{O}(n)$

the Find algorithm- details

- *Returns:* The first iterator i in the range [first,last) for which *i == value. Returns last if no such iterator is found.
- *Complexity:* At most last - first applications of the corresponding predicate → $\mathcal{O}(n)$
- implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value) return first;

    return last;
}
```

the Find algorithm- details

- *Returns:* The first iterator i in the range [first,last) for which *i == value. Returns last if no such iterator is found.
- *Complexity:* At most last - first applications of the corresponding predicate → $\mathcal{O}(n)$

- implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value) return first;

    return last;
}
```

- pre-increment rather than post-increment:

  ++first vs. first++

the Find algorithm- details

- *Returns:* The first iterator i in the range [first,last) for which *i == value. Returns last if no such iterator is found.
- *Complexity:* At most last - first applications of the corresponding predicate → $\mathcal{O}(n)$

- implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value) return first;

    return last;
}
```

- pre-increment rather than post-increment:

  ++first vs. first++

the Find algorithm- Another implementation

the Find algorithm- Another implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (InputIterator p = first; p != last; ++p)
        if (*p == value) return p;

    return last;
}
```

the Find algorithm- Another implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (InputIterator p = first; p != last; ++p)
        if (*p == value) return p;

    return last;
}
```

- Extra variable: p

the Find algorithm- Another implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (InputIterator p = first; p != last; ++p)
        if (*p == value) return p;

    return last;
}
```

- Extra variable: p
- Performance matters exactly for small, frequently used functions that deal with a lot of data

the Find algorithm- Another implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (InputIterator p = first; p != last; ++p)
        if (*p == value) return p;

    return last;
}
```

- Extra variable: p
- Performance matters exactly for small, frequently used functions that deal with a lot of data

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value) return first;

    return last;
}
```

the Find algorithm- Another implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (InputIterator p = first; p != last; ++p)
        if (*p == value) return p;

    return last;
}
```

Good

- Extra variable: p
- Performance matters exactly for small, frequently used functions that deal with a lot of data

Better

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value) return first;

    return last;
}
```

the Find algorithm- Another implementation

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (InputIterator p = first; p != last; ++p)
        if (*p == value) return p;

    return last;
}
```

Good

- Extra variable: p
- Performance matters exactly for small, frequently used functions that deal with a lot of data

Better

```
template<class InputIter, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    for (; first != last; ++first)
        if (*first == value) return first;

    return last;
}
```

- Generic algorithms vs. Hand-written loop

the Find_if algorithm- details

- *Returns:* The first iterator i in the range [first, last) for which the following condition hold: pred(*i) != false. Returns last if no such iterator is found.
- *Complexity:* At most last-first applications of the corresponding predicate → $\mathcal{O}(n)$

- implementation

```
template<class InputIter, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred)
{
    for (; first != last; ++first)
        if (pred(*first)) return first;

    return last;
}
```

- To find specific value vs. To find an element that fulfills a specific requirement

- Predicate: *Unary* predicate

→ Predicate

Predicates

Definition: A *predicate* is something that we can invoke to return true or false.

Predicate

- function or pointer to function
- function object
- lambda expression

- Predicate as a function:

```
inline bool greater_than_zero(int n){  return n > 0; }
void f()
{
    vector<int> v = {-10, -7, -3, 0, 1, 4, 9, 10, 20, 31, 40, 41, 45, 64, 99};
    vector<int>::const_iterator cit = find_if(v.begin(), v.end(), greater_than_zero);
    if (cit != v.end())
        cout << *cit << '\n';
}
```

- A *predicate* is called for each element and returns a boolean value, which the algorithm uses to perform its intended action.

FindFamilyAlgoTest

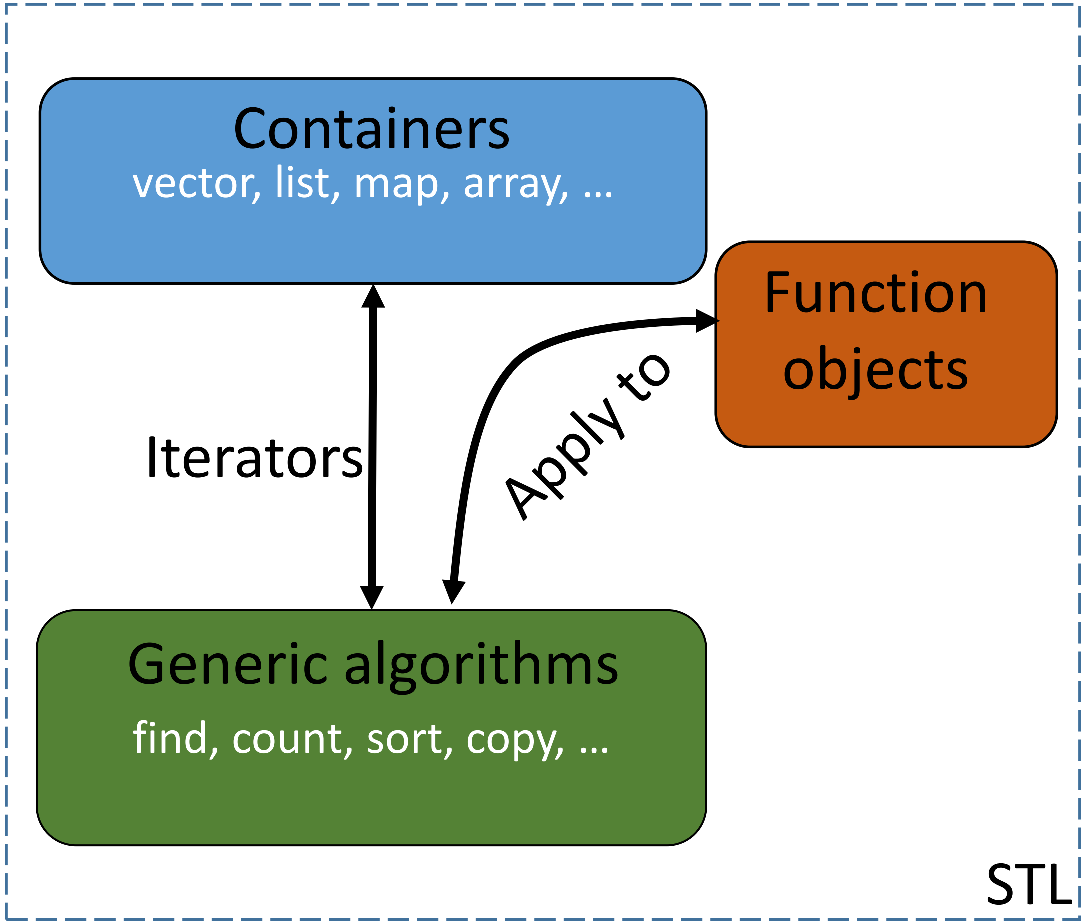
Prog.

Function object

Definitions:

Function object or *functor* is used to define objects that can be called like functions.

Function objects are objects with an operator() defined.



- Function object = Policy object

Function object- an example

```
template<typename T>
class GreaterThan {
    const T val; // value to compare against
public:
    constexpr GreaterThan(const T& v) : val{v} {}
    constexpr bool operator()(const T& x) const { return x > val; } // call operator (predicate)
};
```

- high-qualified class
- The function called operator() implements the “function call”, “call” or “application” operator ().
- x > val not val > x;
- Using function object

```
GreaterThan<int> gti{42};
GreaterThan<string> gts{“Omega”};
void f(int n, const string& s)
{
    bool b1 = gti(n); // true if n > 42
    bool b2 = gts(s); // true if s > “Omega”
}
```

using Greater-than function object

- Predicate as a function object:

```
void f()
{
    vector<int> v = {-10, -7, -3, 0, 1, 4, 9, 10, 20, 31, 40, 41, 45, 64, 99};
    vector<int>::const_iterator cit = find_if(v.begin(), v.end(), GreaterThan<int>(0));
    if (cit != v.end())
        cout << *cit << ‘\n’;
}
```

- Predicate as a lambda expressions:

```
void f()
{
    vector<int> v = {-10, -7, -3, 0, 1, 4, 9, 10, 20, 31, 40, 41, 45, 64, 99};
    vector<int>::const_iterator cit = find_if(v.begin(), v.end(), [](int n )  { return n > 0; } );
    if (cit != v.end())
        cout << *cit << ‘\n’;
}
```

Lambda expressions

- A *Lambda expressions*, is a simplified notation for defining and using an *anonymous* function object. provide a concise way to create simple function objects.

- Lambda expressions a.k.a Lambda functions

- A lambda expression consists of a sequence of parts:
    - A possibly empty *capture list*, specifying what names from the definition environment can be used in lambda expression's body.
    - An optional *parameter list*, specifying what arguments the lambda expression requires
    - An optional *mutable* specifier, indicating that the lambda expression's body may modify the state of the lambda.
    - An optional *noexcept* specifier
    - An optional return type declaration
    - A *body*, specifying the code to be executed.

Lambda expressions- Anatomy

[Capture list $_{opt}$] (Parameters $_{opt}$) -> Return Type $_{opt}$ { Body }

Lambda introducer

Mandatory

Mandatory

[ ] ($_{opt}$) -> $_{opt}$ {}

- Lambda introducer: []
- The first character of lambda expression is always [.
- The simplest lambda expression:

```
[] {} // empty lambda
[] { cout << "Hello, world\n"; }
```

- If a lambda expression does not take any arguments, the argument list can be omitted. Thus the minimal lambda expression is []{}

Lambda expressions- An example

- C++98: using predicates
- C++11: using lambda expressions

```
#include <iostream>
#include <vector>
#include <algorithm>
using std::vector; using std::find_if; using std::cout;

vector<int> v = {-10, -7, -3, 0, 1, 4, 9, 10, 20, 31, 40, 41, 45, 64, 99};
bool gt_zero(int n) { return n > 0; } // named inline function

int main()
{
    vector<int>::const_iterator cit1 = find_if(v.begin(), v.end(), gt_zero);
    auto cit2 = find_if(v.begin(), v.end(),
                        [](int n) { return n > 0; }); // anonymous inline lambda function
    if (cit1 != v.end()) cout << *cit1 << '\n';
    if (cit2 != v.end()) cout << *cit2 << '\n';

    return 0;
}
```

- Lambda expressions allow that to be done “inline” without having to name a function or function object and use it elsewhere.

Capture and capture list

- Some lambda expressions require no access

```
#include <algorithm>
#include <cmath>
vector<int> v = {50, -10, 20, -30};
void vector_sort(const vector<int>& v) {
     sort(v.begin(), v.end()); // sort values: [-30, -10, 20, 50]
     sort(v.begin(), v.end(),
          [](int a, int b) { return std::abs(a) < std::abs(b); } );
                                  // sort absolute values [-10, 20, -30, 50]
}
```

- []: an empty capture list: no local names from surrounding context
- [&]: implicitly capture by reference.
- [=]: implicitly capture by value.
- [*capture-list*]: explicit capture;
- [&, *capture-list*]: implicitly capture by reference all local variables with names not mentioned in the list. Variable names in the capture list are captured by value.
- [=, *capture-list*]: implicitly capture by value all local variables with names not mentioned in the list. Variables named in the capture list are captured by reference.

Capture and capture list- an example

```
#include <algorithm>
#include <cmath>

void vector_sort(const vector<int>& v) {
    bool sensitive = true;
    // …
    sort(v.begin(), v.end(),
         [sensitive](int a, int b) {
                return sensitive ? a < b :  // sort values
                       std::abs(a) < std::abs(b); } ); // sort absolute values
}
```

- The capture of sensitive is done “by value”.

- The choice between capturing by value and by reference is basically the same as the choice for function arguments. We use a reference if we need to write to the captured object or if it is large.

Lambda expressions- sum and product

- capture all local variables with reference

```
int sum = 0;
long long product = 1;
for_each( values.begin(), values.end(), [&](int i){ sum += i; product *= i; });
```

Namespace names

- We don’t need to capture namespace variables (including global variables), because they are accessible.

```
template<typename U, typename V>
ostream& operator<<(ostream& os, const pair<U, V>& p) // global function
{
    return os << '{' << p.first << ', ' << p.second << '}';
}

void print_all(const map<string, int>& m, const string& label)
{
    cout << label << ":\n{\n";
    for_each(m.begin(), m.end(), [](const pair<string, int>& p) { cout << p << '\n'; }
    cout << "}\n";
}
```

Lambda expressions cont.

```
#include <iostream>
int main()
{
    // capture nothing, take zero argument and return nothing
    [] { std::cout << "Hello, lambdas\n"; }; // return void
    return 0;
}
```

- No output

```
#include <iostream>
int main()
{
        auto Lambda = []{ std::cout << "Hello, lambdas\n"; }; // definition
        Lambda(); // lambda function call
        return 0;
}
```

```
auto Lambda = []() -> void { std::cout << "Hello, lambdas\n"; }; // verbose
```

- Lambda with arguments

```
#include <iostream>
auto sum = [](int x, int y){ return x + y; };

int main()
{
        sum();
        return 0;
}
```

Lambda expressions- under the hood

[Capture list $_{opt}$] (Parameters $_{opt}$) -> Return Type $_{opt}$ { Body }

```
class __functor {
private:
        CaptureTypes __captures;
public:
        __functor( CaptureTypes captures )
                : __captures( captures ) { }
        auto operator() ( params ) -> ret { statements; }
};
```

ALPHA

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant

Q & A