High Level Programming

Programming with the STL

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

This work is licensed under the license

CC BY-NC-ND 4.0

Attribution-NonCommercial-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form and for noncommercial purposes only.

BY: Credit must be given to you, the creator.

NC: Only noncommercial use of your work is permitted.
Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

ND: No derivatives or adaptations of your work are permitted.

To view a copy of the license, visit:
https://creativecommons.org/licenses/by-nc-nd/4.0/?ref=chooser-v1

The algorithm library

- Instead of defining each operation as a part of a container, the standard library defines a set of generic algorithms
  - **Generic:** because they operate on elements of different type
  - **Algorithms:** because they implement classical procedures, like sorting, searching, etc.
- Generic algorithms are included in four headers
  - Algorithm, numeric, memory, cstdlib
  - Algorithm defines the most relevant parts (more than 100 functions) to
    - Find, search, sort, combinatorics functions, set operations, etc.

The algorithm library

For more operations see the reference documentation

Type	Meaning
find	Algorithms to find an object.
binary_search	Algorithms to perform a binary search.
partitioning	Divide elements into two groups; the first group includes elements that satisfy a predicate; the second group those that do not satisfy it.
sort	Several sorting algorithm (stable, non-stable, etc.).
rotate (shuffle)	Rotate (randomly reorder) elements
permutation	Generate lexicographical permutation of a sequence.
set	Set algorithms (inclusion, union, intersection, etc.) on sorted sets.
min (max)	Minimum (maximum) value.
sum (difference)	Numeric algorithms.

The algorithm library

- It is essential to understand the structure of these algorithms rather than memorize their details
- They perform an operation on a range of elements
  - Ranges can be specified using pointers or any appropriate iterator type
  - In all following examples
    - **b** is the begin iterator
    - **e** the end iterator
    - **v** a value

The algorithm library

- Many algorithms require a predicate
  - A predicate is an expression that can be called and returns a value that can be used as a condition
  - The default version of the algorithm usually uses a standard predicate

    They allow you to specify conditions or criteria for certain operations like sorting, searching, and filtering
    - The operator is related to the type of the element, e.g., less than <, equal to ==, etc.
  - The extended version usually supplies its own **predicate** operator
    - In the following examples
      - **up** indicates a unary predicate (with one operand)
      - **bp** indicates a binary predicate (with two operands)

Search

- The library offers a variety of search functions
  - Different operations for sorted and unsorted ranges
    - In general, searching on sorted ranges is faster
    - Sorting will pay off for repeated lookups
- General semantics
  - Search operations return iterators pointing to the result
  - Unsuccessful operations are usually indicated by returning the **end** iterator

Search

- Several variants are possible

Type	Meaning
find(b,e,v)	Return an iterator to the first element in the input range equal to val.
find_if(b,e,up)	Return an iterator to the first element for which the predicate pred succeeds.
count(b,e,v)	Count matching elements.
count_if(b,e,up)	Count how many times pred succeeds.
all_of(b,e,up)	Return a bool if pred succeds for all elements (similarly for any_of and some_of).
search(b1,e1,b2,e2, bp)	Return an iterator to the first position of the input range at which the second range occur as a subsequence.

Examples

```
#include <algorithm>
#include <vector>

std::vector<int> v = {2, 6, 1, 7, 3, 7};

auto res1 = std::find(v.begin(), v.end(), 7);
// res1 refer the first value equal to 7 in the sequence

auto res2 = std::find(v.begin(), v.end(), 9);
// no 9 in the sequence; the end iterator is returned
if(res2 == v.end())
  std::cout << "Not found!";
```

This is a value

Binary search

- On sorted ranges, the library offers binary search operations
  - Require forward iterators but are faster with random iterators
  - These algorithms execute a logarithmic number of comparison
    - Complexity $O(log(N))$
    - However, when use with forward iterators make a linear number of iterator operations
  - They can employ custom comparison function
    - Please, see section of lambda functions

Binary search

- Elements in the input sequence must be sorted

several functions commonly used with sorted sequences

Type	Meaning
lower_bound(b,e,v)	Returns and iterator denoting the first element such that val is not less than that element. in the range of b, e
upper_bound(b,e,v)	Returns and iterator denoting the first element such that val is less than that element.
equal_range(b,e,v)	Return a pair: The first member returned by lower_bound and the second by upper_bound.
binary_search(b,e,v)	Return a bool indicating whether the sequence contains a value equal to val.

Examples

In C++, iterators for containers like vectors are used to navigate through the elements of the container. They provide a more generic way to access elements compared to indices, and they can be used with various standard library algorithms and functions. However, if you need the index of an element found by an algorithm like lower_bound, you must convert the iterator to an index using subtraction.

```
#include <algorithm>
#include <vector>

vector<int> arr1 = { 10, 15, 20, 25, 30, 35 };
vector<int> arr2 = { 10, 15, 20, 20, 25, 30, 35 };
vector<int> arr3 = { 10, 15, 25, 30, 35 };

// prints 2
cout <<
  lower_bound(arr1.begin(), arr1.end(), 20) - arr1.begin()
  << endl;

// prints 2
cout <<
  lower_bound(arr2.begin(), arr2.end(), 20) - arr2.begin();
  << endl;

// prints 2 (index of next higher)
cout <<
  lower_bound(arr3.begin(), arr3.end(), 20) - arr3.begin();
   << cout;
```

The lower bound function returns an iterator pointing to the first element that is not less than 20, which is 20 itself. The difference between this iterator and arr1.begin() gives the index of the found element, which is 2.

The reason you need to subtract arr1.begin() (or the equivalent for other vectors) is because the lower_bound function returns an iterator, not an index.

An iterator is similar to a pointer; it points to a memory location. Subtracting arr1.begin() from the iterator converts it to an index (an integer representing the position of the element within the vector). This is necessary if you want to get the index of the element found by lower_bound.

This is a value

Examples

```
#include <algorithm>
#include <vector>

vector<int> arr = { 10, 15, 20, 25, 30, 35 };

// Use binary_search to check if 15 exists
if (binary_search(arr.begin(), arr.end(), 15))
  cout << "15 exists in vector";
else
  cout << "15 does not exist";
cut << endl;

// Use binary_search to check if 23 exists
if (binary_search(arr.begin(), arr.end(), 23))
  cout << "23 exists in vector";
else
  cout << "23 does not exist";
```

boolean value is returned

This is a value

Binary search only lets you know if an element exists or not, if you want to know the position of the element then it's better to use lower_bound or upper_bound.

Sort

- The sort algorithm orders all elements
  - They need a random-access iterator So containers like strings, vectors, arrays
  - Each algorithms is given in two forms
    - The first one, use the operator "<" to compare elements
    - The second one, takes an extra parameters that specifies an ordering relation You can provide a custom function that defines the conditions for the sort.
  - Algorithms do not guarantee the order of equal elements
  - Usually, they need $O(N \cdot \log(N))$ comparisons

Sort

- All following functions have two versions
  - The first with a standard comparison function
  - The second with a third parameter (bp, i.e., a binary predicate) to specify the comparison operator

Stable sort -> For example, suppose you have a list of people sorted by their age. If two people have the same age, a stable sort algorithm would ensure that the person who appeared first in the original list retains their position in the sorted list.

Type	Meaning
sort(b,e,bp)	**Sort** an entire range.
stable_sort(b,e,bp)	As before, bur with a **stable** sorting procedure.
is_sorted(b,e,bp)	Returns a bool to indicate whether the range is sorted.
is_sorted_until(b,e,bp)	Checks if a (partial) range is sorted.
partial_sort(b,mid,e,bp)	Sorts all elements between mid-b and places those elements at the beginning of the range.

Examples

```
#include <algorithm>
#include <vector>

std::vector<unsigned> v={3,4,1,2};
...
std::sort(v.begin(),v.end());
// Now v is 1, 2, 3, 4
```

Sort uses the standard comparison function for integers (<)

```
#include <algorithm>
#include <vector>

std::vector<string> words = {...};

bool isShorter (const string &s1, contr string &s2) {
  return s1.size() < s2.size();
}


sort (words.begin(), words.end(), isShorter);
// Now the array word is alphabetically sorted
```

Ad-hoc comparison function

Sort uses an ad-hoc comparison function (predicate)

Examples

```
#include <vector>
#include <algorithm>
using namespace std;

vector<int> v = { 10, 10, 30, 30, 30, 100, 10,
                  300, 300, 70, 70, 80 };

std::pair<std::vector<int>::iterator,
std::vector<int>::iterator> ip;

// Sorting the vector v
sort(v.begin(), v.end());
// v becomes 10 10 10 30 30 30 70 70 80 100 300 300

// Using std::equal_range to compare elements with 30
ip = std::equal_range(v.begin(), v.begin() + 12, 30);

// Display the subrange bounds
cout << "30 is present in the sorted vector from index "
     << (ip.first - v.begin()) << " till "
     << (ip.second - v.begin());
```

Sort and other function used together

Sort uses the standard comparison function for integers (<)

Permutations

- The permutation algorithms generate lexicographic permutations of a sequence
  - The algorithms reorder a permutation to generate the next or previous permutation in a given sequence
    - The permutation are listed in lexicographical order based on the less than operator
    - Example: abc, acb, bac, bca, cab, cba
  - The algorithm may proceed forward and backward in the permutation
    - It requires a bidirectional iterator
  - A custom comparison function can be supplied (see belove)

Permutations

- The algorithms assume that the element of the sequence are unique
  - Please remind, simple permutation versus permutation with repetition

Type	Meaning
is_permutation(b1,e1,b2,bp)	Return true if there is a permutation of the second sequence with all elements of the first sequence.
next_permutation(b,e,bp)	Tranform the input sequence into the next sequence (or the first one if the input sequence is the last one).
prev_permutation(b,e,bp)	As before, but in reverse order.

Examples

Print all permutation of the string “abc”
abc, acb, bca, cba, bca, cba

```
#include <algorithm>
#include <string>
#include <iostream>

int main() {
  std::string s = "abc";
  std::sort(s.begin(), s.end());
  do {
    std::cout << s << '\n';
  } while(std::next_permutation(s.begin(), s.end()));
}
```

Sort uses the standard comparison function for integers (<)

With the string s=“aba” it prints
aba, baa, aab

Explanation:

std::string s = "abc";: Initialize a string s with the value "abc".
std::sort(s.begin(), s.end());: Sort the characters in the string s lexicographically. This is necessary to generate permutations in lexicographically sorted order.
do { ... } while(std::next_permutation(s.begin(), s.end()));: This loop generates and prints permutations of the string s using std::next_permutation. It repeatedly generates the next lexicographically greater permutation of the string until no more permutations are possible.
std::next_permutation(s.begin(), s.end()) generates the next permutation of the string s in lexicographically sorted order.
If a next permutation is possible, it rearranges the elements of the string s to the next lexicographically greater permutation and returns true, otherwise, it returns false indicating that no more permutations are possible.
Inside the loop body, std::cout << s << '\n'; prints the current permutation of the string s.

Set algorithms

- Set operations are possible on a sequence that is in sorted order

Type	Meaning
includes(b1,e1,b2,e2,bp)	Returns true if every element of the second sequence is present in the first sequence.
set_union(b1,e1,b2,e2,bp)	Create a sorted sequence with the elements that are in either sequence.
set_intersection(b1,e1,b2,e2,bp)	Create a sorted sequence with the elements that are in both sequences.
set_difference(b1,e1,b2,e2,bp)	Create a sorted sequence with the elements present in the first sequence but not in the second.
set_symmetric_difference (b1,e1,b2,e2,bp)	Create a sorted sequence of elements present in either sequence but not in both.

Examples

We need to introduce sets to better understand this example !

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <set>

int main() {
  int a[] = { 1, 3, 5 };
  int b[] = { 0, 2, 4, 6 };
  std::set<int> s;

  std::set_union (std::begin(a), std::end(a),
                  std::begin(b), std::end(b),
                  std::inserter (s, s.begin()));

  for (int x : s)
    std::cout << x << ' ';
  std::cout << std::endl;

  return 0;
}
```

Set union:
$s = a \cup b$

Examples

We need to introduce sets to better understand this example !

```
#include <iostream>
#include <set>
#include <algorithm>
int main() {
  std::set<int> a = {1, 2, 3, 4, 5};
  std::set<int> b = {3, 4, 5, 6, 7};
  std::set<int> in, dif;

  std::set_intersection(a.begin(), a.end(),
                        b.begin(), b.end(),
                        std::inserter(in, in.begin()));
  for (int num : in) {
    std::cout << num << " ";
  }

  std::set_difference(a.begin(), a.end(),
                      b.begin(), b.end(),
                      std::inserter(dif, dif.begin()));
  for (int num : dif)
    std::cout << num << " ";
  return 0;
}
```

Set intersection:
$s = a \cap b$

Set difference:
$s = a - b$

Algorithms and predicates

- In all previous examples, the predicates where
  - Standard or
  - Implemented through an external function
- In general, a predicate can be any callable object, i.e., an object that we can **call**
  - In C++, there are three types of callable objects
    - **Functions**
    - Classes overloading a function
    - **Lambda expressions**
- We need to analyze lambda expressions and see how to use them as an algorithm predicate

Lambda expressions

Lambda expressions

```
[capture_list] (parameter_list) -> return_type {body}
```

- Represent a **callable** unit of code
- It can be thought of as an **unnamed**, **inline function**
- **Like** any other function, a lambda has
  - A parameter list, a return type, and a function body
- **Unlike** any other function, a lambda
  - May be defined **inside** a function
  - Being an **internal** function has a **capture** list

They can also have a name

Lambda expressions

- The capture_list
  - Although a lambda may appear inside a function, it can use variables local to that function **only** if it specifies which variables it intends to use
  - Specifies which local variables will be used by the lambda expression
  - It may be empty

```
[capture_list] (parameter_list) -> return_type {body}
```

The capture list must always be present. It is eventually empty.

Lambda expressions

- Similarly to standard functions, lambdas can capture variables by value or by reference

Type	Meaning
[]	Empty capture list. The lambda use only local variables.
[v1,v2,...]	A comma-separated list of local variables. By default, variables are copied. When preceded by & are captured by reference.
[&]	**All** objects in the enclosing function are passed by reference.
[=]	**All** objects in the enclosing function are passed by value.
[&,v1,v2,…]	**All** variables are captured by reference **but** the ones in the list (captured by value).
[=,&v1,&v2,…]	**All** variables are captured by value **but** the ones in the list (captured by reference).

Lambda expressions

- The parameter_list
  - Is a comma-separated list of function parameters (used in the body)
    - Like any other function, the arguments are used to initialize the lambda's parameters
  - Arguments and parameter types must match
    - A lambda may not have default arguments

```
[capture_list] (parameter_list) -> return_type {body}
```

The parameter list has a standard format (as all other functions). It can be omitted.

Lambda expressions

- The return_type
  - Specifies the type of the object the function returns

```
[capture_list] (parameter_list) -> return_type {body}
```

Unlike other functions, lambda must use a **trailing return**. A trailing return follows the parameter list and is preceded by ->.
It can be omitted.

Lambda expressions

- If the body of a lambda includes
  - **Only** a return statement, the type of the lambda expression is deduced by the return statement
  - Any statement other than a return, that lambda is supposed to return **void**
  - In alll other cases, we need to define a return type using a **trailing return** type

```
[capture_list] (parameter_list) -> return_type {body}
```

Unlike other functions, lambda must use a **trailing return**. A trailing return follows the parameter list and is preceded by ->.
It can be omitted.

Lambda expressions

- The body
  - Includes the function body, i.e., its implementation

```
[capture_list] (parameter_list) -> return_type {body}
```

The body must always be present.

Examples

The parameter list

```
[](const string &a, const string &b)
  { return a.size() < b.size(); }
```

Lambda function to evaluate which string is shorter

This is how we call it within a stable sorting algorithm

```
stable_sort (words.begin(), words,end()
  [](const string &a, const string &b)
    { return a.size() < b.size(); }
  );
```

lambda

Sort a vector of integer values

```
std::vector<unsigned> v = {3, 4, 1, 2};
std::sort(v.begin(), v.end(),
  [](unsigned lhs, unsigned rhs) {return lhs > rhs;});
// v is now {4, 3, 2, 1}
```

lambda

```
[capture_list] (parameter_list) -> return_type {body}
```

Examples

```
#include <algorithm>
#include <vector>

std::vector<int> v = {2, 6, 1, 7, 3, 7};

auto it = std::find(v.begin(), v.end(), 7);
// it points to the first element equal to 7

int a = std::distance(v.begin(), it);
// Now a = 3, i.e., the index distance between
// iterator begin() and it
```

Standard comparison

```
auto it = std::find_if(
  v.begin(), v.end(),
  [](int val) { return (val % 2) == 1; }
);
// it points to the first odd element, i.e., 1
int a = std::distance(vec.begin(), it);
// Now a = 2, i.e., the index distance between
// iterator begin() and it
```

Lambda function

```
[capture_list] (parameter_list) -> return_type {body}
```

Examples

my_size is an object local to the "external" function

```
[my_size](const string &a)
  { return a.size() >= my_size; }
```

Captured value

Used inside the function to compare the string size

This is how we call it within the find_if algorithm to return an iterator to the firsts element that is at least as long as the given size

```
auto wc = find_if (words.begin(); words.end();
  [my_size](const string &a)
    { return a.size() >= my_size; }
  );
```

Examples

Passing a lambda function to a user function

Standard function

```
int callFunc(int (*func)(int, int), int arg1, int arg2) {
  return func(arg1, arg2);
}

auto lambda = [](int arg1, int arg2) {
  return arg1 + arg2;
};


int i = callFunc(lambda, 2, 4);
// Now i = 6


int j = lambda(5, 6);
// Now j = 11
```

Locally defined and **named** lambda function

Calling the standard function with lambda as a parametr

Direct call of a lambda function

```
[capture_list] (parameter_list) -> return_type {body}
```

Examples

Capture list

```
int i = 0;
int j = 42;

auto lambda1 = [i](){};      // i by-copy
auto lambda2 = [&i](){};     // i by-reference

auto lambda3 = [&j, i](){};  // j by-reference, i by-copy
auto lambda4 = [=, &i](){};  // j by-copy, i by-reference
```

Capture by Value: When you capture a variable by value, a copy of that variable is made and stored within the lambda function. Any changes made to the variable within the lambda function do not affect the original variable in the enclosing scope.Example:

```
int x = 10;
auto lambda = [x] () { std::cout << x << std::endl; };
x = 20; // Changing the value of x after lambda creation
lambda(); // Outputs 10
```

Capture by Reference: When you capture a variable by reference, the lambda function will access the original variable from the enclosing scope directly. Any changes made to the variable within the lambda function will affect the original variable in the enclosing scope. Example:

```
int y = 10;
auto lambda = [&y] () { std::cout << y << std::endl; };
y = 20; // Changing the value of y after lambda creation
lambda(); // Outputs 20
```

auto lambda4: This declares a lambda function named lambda4, and the auto keyword indicates that the lambda's type is deduced automatically.
[=, &i]: This is the capture list. = indicates that all variables from the enclosing scope should be captured by value, and &i specifies that the variable i should be captured by reference.
() {}: This is the parameter list and function body of the lambda function. In this case, the lambda takes no parameters (()), and the function body is empty ({}).

```
// ERROR: non-diverging capture types
auto lambda5 = [&, &i](){};

// ERROR: non-diverging capture types
auto lambda6 = [=, i](){};
```

Mixed Capture: You can mix capture modes within the same capture list. This allows you to capture some variables by value and others by reference.Example:

```
int x = 10;
int y = 20;
auto lambda = [&x, y] () { std::cout << x << " " << y << std::endl; };
x = 30;
y = 40;
lambda(); // Outputs 30 20
```

Example

- The capture is done at the definition, thus
  - In the capture by value, the **value** is persistent
  - In the capture by reference, the **reference** is persistent (**not** the value)

```
int i = 20;
auto lambda1 = [i]() { return i + 42; };
auto lambda2 = [&i]() { return i + 42; };
i = 0;

int a = lambda1();
// Now a = 20+42 = 62
int b = lambda2();
// Now b = 0+42 = 42
```

This lambda function captures the variable i by value ([i]), meaning it makes a copy of the value of i at the time of lambda creation.
Inside the lambda function, i is used in the expression i + 42. Since i was captured by value, it retains its original value (20) even if i is modified later in the enclosing scope.

The local value of i is retained

This lambda function captures the variable i by reference ([&i]), meaning it refers directly to the variable i in the enclosing scope.
Inside the lambda function, i is used in the expression i + 42. Since i was captured by reference, any changes to i in the enclosing scope are reflected inside the lambda function.

The current value of i is retained

Examples

The return type

The algorithm transform takes three iterators:
It transform the values included in the range specified by the first two iterators copying them to the third iterator

```
transform (v.begin(), v.end(), v.begin(),
  [](int i) { return i < 0 ? -i : i; }
);
```

There is only a return statement in the body; the type of the lambda is automatic

```
transform (v.begin(), v.end(), v.begin(),
  [](int i) -> int
    { if (i<0) return -i; else return i; }
);
```

Here, there are other statements, we need to define the return type with the trailing return

C++ versus C

- Which are the main differences between C macros and C++ lambdas

C (preprocessor)

```
#define MAX(A,B) (((A)>(B))?(A):(B))
#define LAMBDA(A,B) ((A)+(B))
```

C++

```
auto lambda = [](int arg1, int arg2) {
  return arg1 + arg2;
};
```

Macros	Lambdas
Are just a brute force text substitution mechanism	Are much less verbose than other library functions
Cannot be passed to as an argument to an algorithm	Are a much more general construct
Are the way to go, to achieve the shortest syntax possible	The preprocessor is strongly discouraged in C++

Exercise

- Which is the output generated b the following program?

```
int main() {
  int i, j;
  vector<int> v{0,1,2,3,4,5,6};
  auto l = [&](int i){ swap(v[i], v[v.size()-1-i]); };
    for (i=0, j=v.size()-1; i<j; i++, j--) {
      cout << v[i] << " ";
      l(i);
    }
  cout << "# ";
  for(auto e: v) {
    cout << e << " ";
  }
  return 1;
}
```

Exam 04.09.2023

This indicates that the lambda function captures variables from the enclosing scope by reference. The lambda will have access to variables v, i, and any other variables from the enclosing scope, and any changes made to these variables inside the lambda will affect the original variables in the enclosing scope.

{ swap(v[i], v[v.size()-1-i]); }: This is the function body of the lambda. It contains the code to be executed when the lambda is called. In this case, it swaps two elements of the vector v using the swap function. The elements being swapped are v[i] and v[v.size()-1-i], which swaps elements symmetrically around the midpoint of the vector v.

Solution

- Which is the output generated b the following program?

Exam
04.09.2023

```
int main() {
  int i, j;
  vector<int> v{0,1,2,3,4,5,6};
  auto l = [&](int i){ swap(v[i], v[v.size()-1-i]); };
    for (i=0, j=v.size()-1; i<j; i++, j--) {
      cout << v[i] << " ";
      l(i);
    }
  cout << "# ";
  for(auto e: v) {
    cout << e << " ";
  }
  return 1;
}
```

example of a range-based for loop combined with multiple loop variables.

Initialization: i=0, j=v.size()-1;
Two loop variables i and j are initialized. i starts from 0, and j starts from the last index of the vector v.
Condition: i<j;
The loop continues as long as the value of i is less than j.
Iteration: i++, j--
After each iteration of the loop, both i and j are incremented and decremented respectively.

```
0 1 2 # 6 5 4 3 2 1 0
```

Exercise

- Which is the output generated b the following program?

Exam 07.07.2023

```
auto lambda = []( std::string h )->bool{
   return ( h != "-" && h != "." );
};
int main() {
  std::string s("123.456.789-00");
  std::vector<std::string> num;
  for (int i = 0; i < s.length() ; i++) {
    num.push_back( s.substr(i, 1) );
  }
  cout << s << "#";
  for( auto z : num ) {
    if (lambda(z)) std::cout << z;
  };
  std::cout << '\n';
  return 0;
}
```

EXTRACTS SINGLE STRING

std::string substr (size_t pos, size_t len) const;
pos: The starting position of the substring to be extracted. It represents the index of the first character to include in the substring. The index is zero-based, so the first character in the string has index 0.
len: The length of the substring to extract. This parameter specifies the number of characters to include in the substring starting from the position pos. If this parameter is not specified, or if it exceeds the length of the string starting from pos, the returned substring will include characters up to the end of the string.
Returns: A new string containing the extracted substring

```
123.456.789-00#1234567890
```

Solution

- Which is the output generated b the following program?

Exam 07.07.2023

```
auto lambda = []( std::string h )->bool{
   return ( h != "-" && h != "." );
};
int main() {
  std::string s("123.456.789-00");
  std::vector<std::string> num;
  for (int i = 0; i < s.length() ; i++) {
    num.push_back( s.substr(i, 1) );
  }
  cout << s << "#";
  for( auto z : num ) {
    if (lambda(z)) std::cout << z;
  };
  std::cout << '\n';
  return 0;
}
```

```
123.456.789-00#1234567890
```

Exercise: Sorting Student Records

- Write a C++ program that manages a list of student records and performs the following tasks
  - Allow the user to input student records one by one. Each record should include the student's ID, name, and grade
  - Store the student records in a sequential container
  - Sort the students by
    - ID in ascending order
    - Name in alphabetical order
    - Grade in descending order
  - Use lambda functions to define custom sorting criteria for the sorting function

Example

```
Enter student records (ID, Name, Grade):
1 John 85.5
2 Alice 92.0
3 Bob 78.3
4 Sarah 88.7
5 Mike 75.2

Choose sorting criteria:
1. Sort by ID
2. Sort by Name
3. Sort by Grade

Enter your choice: 3

Sorted Student Records by Grade (descending order):
ID: 2, Name: Alice, Grade: 92.0
ID: 4, Name: Sarah, Grade: 88.7
ID: 1, Name: John, Grade: 85.5
ID: 3, Name: Bob, Grade: 78.3
ID: 5, Name: Mike, Grade: 75.2
```

Input

Output

Solution

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

struct Student {
  int id;
  std::string name;
  double grade;
};

// Function to display student records
void displayRecords(const std::vector<Student>& students) {
  for (const auto& student : students) {
    std::cout << "ID: " << student.id << ", Name: "
              << student.name << ", Grade: "
              << student.grade << std::endl;
  }
}
```

Data structure and output function

Solution

Sorting lambda functions

```
// Sort student records based on ID in ascending order
void sortByID(std::vector<Student>& students) {
  std::sort(students.begin(), students.end(),
    [](const Student& a, const Student& b) {
      return a.id < b.id;
  });
}
// Sort student records based on name in alphabetical order
void sortByName(std::vector<Student>& students) {
  std::sort(students.begin(), students.end(),
    [](const Student& a, const Student& b) {
      return a.name < b.name;
  });
}
// Sort student records based on grade in descending order
void sortByGrade(std::vector<Student>& students) {
  std::sort(students.begin(), students.end(),
    [](const Student& a, const Student& b) {
      return a.grade > b.grade;
  });
}
```

Solution

Main: Part 2

```
int main() {
  std::vector<Student> students;
  int id, choice;
  std::string name;
  double grade;

  std::cout << "Enter student records (ID, Name, Grade):\n";
  while (true) {
    std::cout << "> ";
    std::cin >> id >> name >> grade;
    if (id==0)
      break;
    students.push_back({id, name, grade});
  }

  std::cout << "\nChoose sorting criteria:\n";
  std::cout << "1. Sort by ID\n";
  std::cout << "2. Sort by Name\n";
  std::cout << "3. Sort by Grade\n";
  std::cout << "\nEnter your choice: ";
  std::cin >> choice;
```

Solution

Main: Part 2

```
  switch (choice) {
    case 1:
      sortByID(students);
      std::cout << "ID Sorting:\n";
      break;
    case 2:
      sortByName(students);
      ...
      break;
    case 3:
      sortByGrade(students);
      ...
      break;
    default:
      std::cout << "Invalid choice.\n";
      return 1;
    }
 displayRecords(students);
 return 0;
}
```