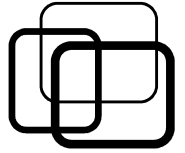


Standard Template Library

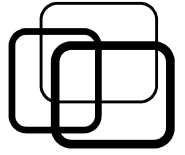
Inst. Nguyễn Minh Huy

Contents



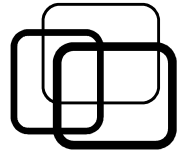
- Overview.
- Containers.
- Algorithms.

Contents



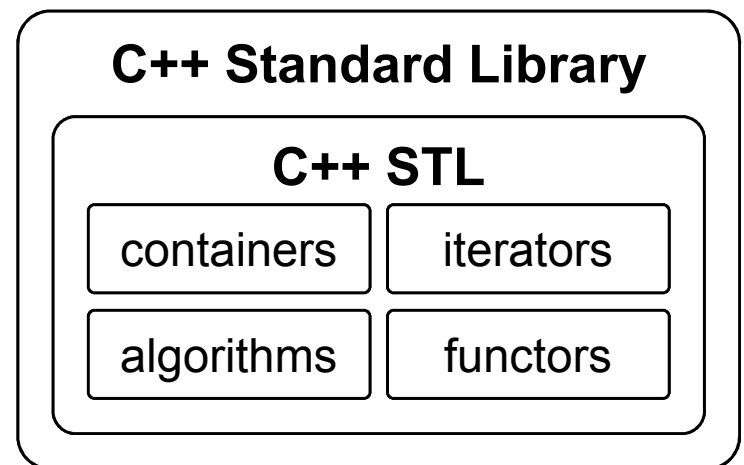
- **Overview.**
- Containers.
- Algorithms.

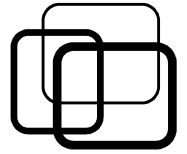
Overview



■ STL origin:

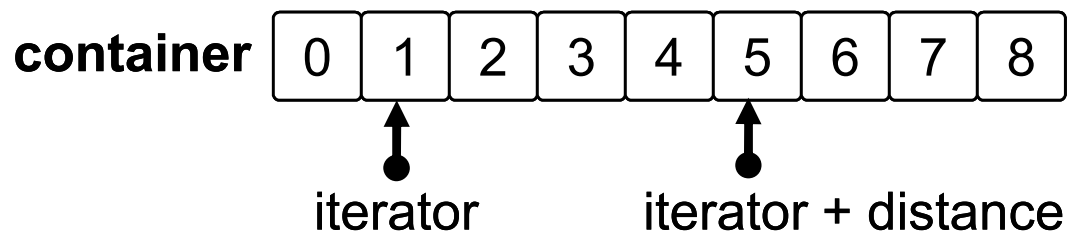
- Alexander Stepanov, 1994.
- Main part of C++ Standard Library.
- Use template extensively.
- Abstract data types and algorithms.
- Structure:
 - Containers.
 - Algorithms.
 - Iterators.
 - Functors.



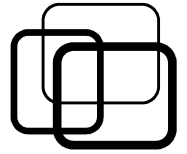


■ Iterator concept:

- An abstract pointer points to container element.
- An object to access items in container.
- Pointer operations:
 - Referencing: `<iterator> = <get position>`.
 - De-referencing: `*<iterator>`, `<iterator>->`.
 - Jumping: `++/--<iterator>`, `<iterator> +/- <distance>`.
 - Distance: `<iterator 1> - <iterator 2>`.



Overview



■ Iterator concept:

■ Referencing:

`<container type>::iterator <iterator> = <container position> +/- k.`

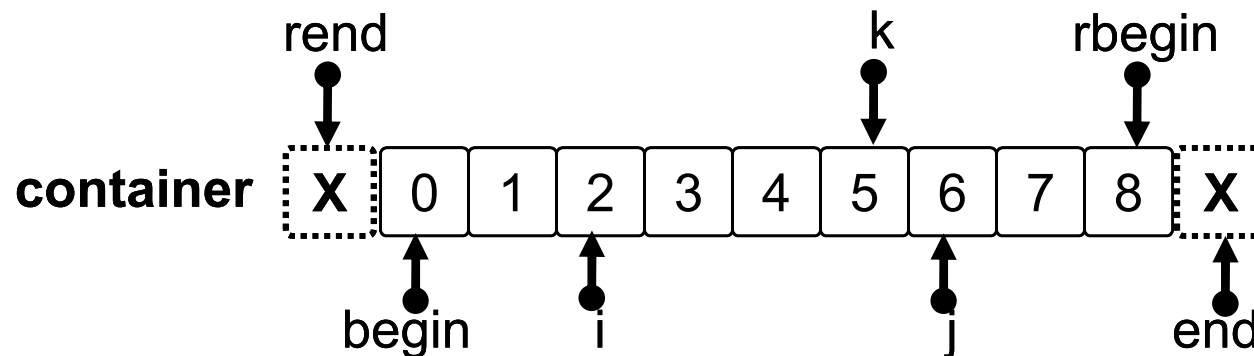
`<container position>`: `begin`, `end`, `rbegin`, `rend`.

```
std::vector<int> v {0, 1, 2, 3, 4, 5, 6, 7, 8};
```

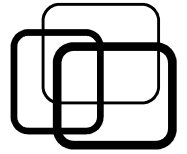
```
std::vector<int>::iterator i = v.begin() + 2;
```

```
auto j = v.end() - 3;           // auto type deducing.
```

```
auto k = v.rbegin() + 3;       // auto type deducing.
```



Overview

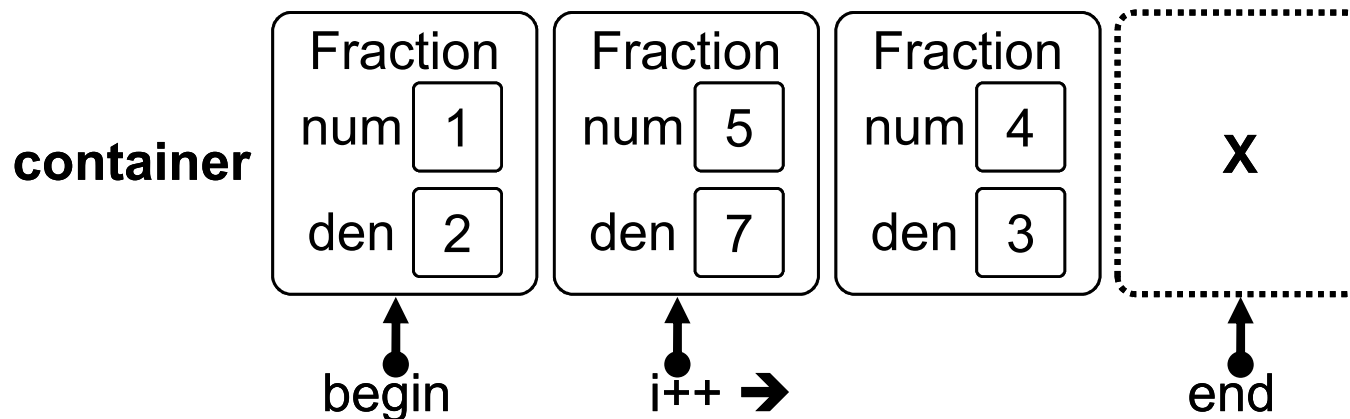


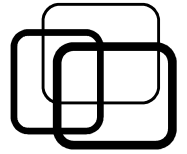
■ Iterator concept:

■ De-referencing and jumping:

```
struct Fraction {  
    int num, den;  
};
```

```
std::vector<Fraction> v { {1, 2}, {5, 7}, {4, 3} };  
for (auto i = v.begin( ); i != v.end( ); ++i )  
    std::cout << i-> num << '/' << i-> den << '\n';
```





■ Iterator concept:

■ Stream iterator:

- Treat stream like a container.
- Read/write stream in STL way.
- Begin position:

`std::istream_iterator<type>(<stream>).`

`std::ostream_iterator<type>(<stream>, <delimiter string>).`

- End position: `std::istream_iterator<type>().`

```
std::ifstream f("input.txt");
```

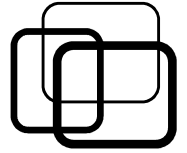
```
auto f_begin = std::istream_iterator<int>( f );
```

```
auto f_end = std::istream_iterator<int>( );
```

```
auto o = std::ostream_iterator<int>( std::cout, "\n" );
```

```
for ( auto i = f_begin; i != f_end; ++i )
```

```
    *o = *i;    // Read int from f and write to std::cout.
```

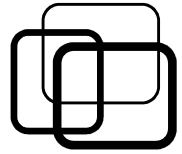



■ Iterator concept:

■ Insert iterator:

- Write value to iterator = insert value to container.
- Convenient way to insert element.
- Back insert position: **std::back_inserter**(<container>).
- Front insert position: **std::front_inserter**(<container>).
- Specified position: **std::inserter**(<container>, <iterator pos>).

```
std::ifstream f("input.txt");
auto f_begin = std::istream_iterator<int>( f );
auto f_end = std::istream_iterator<int>( );
std::vector<int> v;
auto o = std::back_inserter( v );
for ( auto i = f_begin; i != f_end; ++i )
    *o = *i; // Read int from f and push_back to v.
```



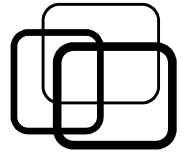
■ Functor concept:

- An object can be called like function.
- By defining operator ().

<return type> <class name>::**operator** ()(<arguments>).

```
class Power {  
private:  
    int m_expo;  
public:  
    Power( int expo ): m_expo( expo ) { }  
  
    float operator ( )( float base ) {  
        float res = 1;  
        for ( int i = 0; i < m_expo; ++i )  
            res *= base;  
        return res;  
    }  
};
```

```
int main()  
{  
    Power square( 2 );  
    float x = square( 3 );  
    float y = square( 5 );  
  
    Power cube( 3 );  
    float z = cube( 4 );  
    float t = cube( 6 );  
}
```

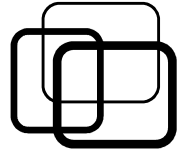


■ Functor concept:

- A functor can store state.

```
class EvenCounter {  
private:  
    int m_count; // Functor state.  
public:  
    EvenCounter( int start ):  
        m_count( start ) {  
    }  
  
    int operator ( )( int value ) {  
        if ( value % 2 == 0 )  
            ++m_count; // Update each call.  
        return m_count;  
    }  
};
```

```
int main()  
{  
    std::vector<int> v {1, 2, 3, 4, 5};  
  
    EvenCounter count( 0 );  
    for ( auto e: v )  
        count( e );  
  
    std::cout << count( 1 );  
}
```



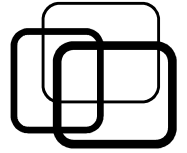
■ Functor concept:

■ Pre-defined functors (library <functional>):

- Arithmetics: `std::plus`, `std::minus`.
- Comparisons: `std::greater`, `std::less`, `std::equal_to`.
- Not complement: `std::not_fn`.

```
auto is_same = std::equal_to( );  
bool r1 = is_same( 1, 1 );    // r1 = true.  
bool r2 = is_same( 1, 2 );    // r2 = false.
```

```
auto is_different = std::not_fn( is_same );  
bool r3 = is_different( 1, 1 ); // r3 = false.  
bool r4 = is_different( 1, 2 ); // r4 = true.
```



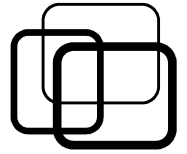
■ Functor concept:

■ Lambda expression: (C++11)

- An anonymous functor.
- Defined and used in-place.

```
[<captured states>] ( <arguments> ) -> <return type>
{
    // Functor body.
}
```

<capture states>: existing or declared variables.



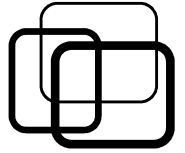
■ Functor concept:

■ Lambda expression: (C++11)

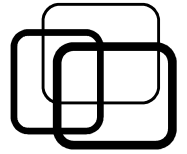
```
int main()
{ // Anonymous functor having cnt as state.
  auto countEven = [cnt = 0] ( int value ) {
    if ( value % 2 == 0 )
      ++cnt;
    return cnt;
  }

  std::vector<int> v {1, 2, 3, 4, 5};
  for ( auto e: v )
    countEven( e );
  std::cout << countEven( 1 );
}
```

Contents



- Overview.
- **Containers.**
- Algorithms.

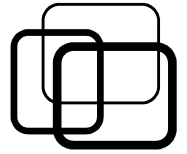


■ Basic concept:

- A collections of same type elements.
- Store elements in specific data structure.
- Features:
 - Common ways to work with different data structures.
 - General access by iterator.
 - Dynamic memory management.




■ Classifications:

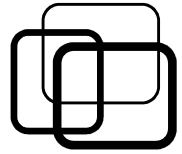
- Sequence containers.
- Associative containers.
- Container adaptors.



■ Sequence containers:

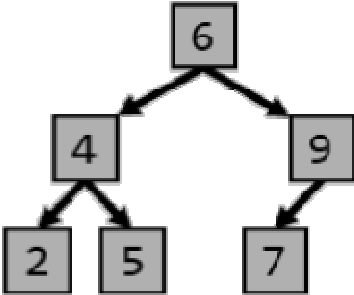
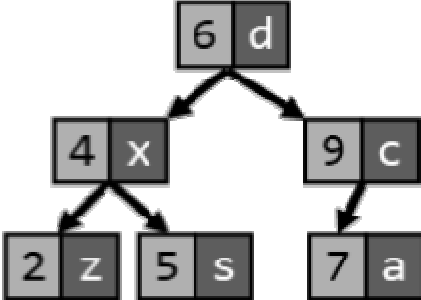
- Store elements in linear data structure.
- Insert orders are maintained.
- Access element by index.

Containers	Data structures	Features
<code>std::vector</code>		Random access Fast insert/delete end Low memory cost
<code>std::list</code>		Sequential access Very fast insert/delete
<code>std::deque</code>		Random access Fast insert/delete begin, end High memory cost

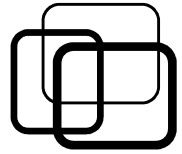


■ Associative containers:

- Store elements in binary search tree.
- Insert orders are not maintained.
- Access element by value.

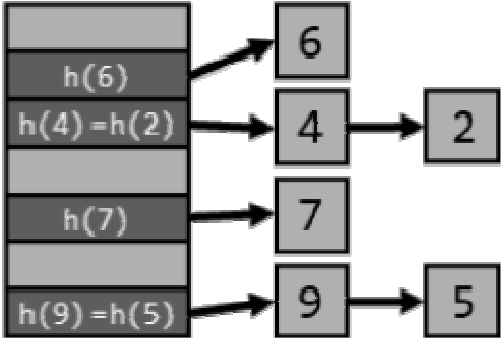
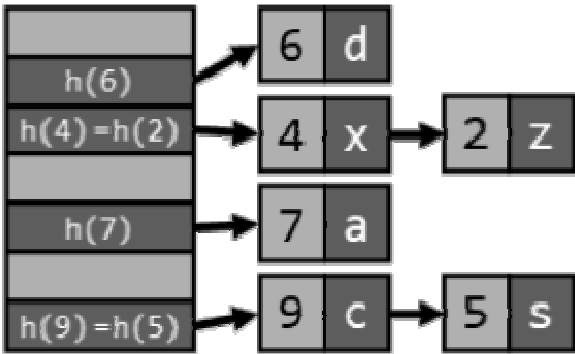
Containers	Data structures	Use cases
<code>std::set</code> <code>set::multiset</code>		Fast search values Fast insert/delete
<code>std::map</code> <code>std::multimap</code>		Fast search pairs Fast insert/delete

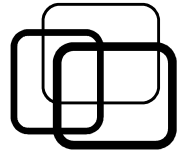
Containers



■ Unordered associative containers:

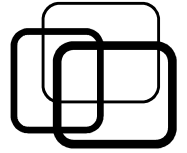
- Store elements in hash table.
- Faster but more memory cost.

Containers	Data structures	Use cases
<code>std::unordered_set</code> <code>set::unordered_multiset</code>		Very fast search values Very fast insert/delete Very high memory cost
<code>std::unordered_map</code> <code>std::unordered_multimap</code>		Very fast search pairs Very fast insert/delete Very high memory cost



■ Container adaptors:

- Wrapper of sequence container.
- Provide different API.
- `std::stack`: LIFO access (push, pop, top).
- `std::queue`: FIFO access (push, pop, front, back).
`std::stack<Fraction> s1; // Stack with std::deque as container.`
`std::queue<int, std::list<int> > s2; // Queue with std::list as container.`



■ Construct container:

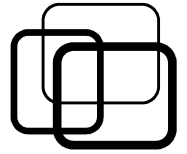
■ Constructor:

- `<container type> <container>{ element1, element2, ... }.`
- `<container type> <container>(<iter begin>, <iter end>).`

■ Assign (sequence container):

- `<container>.assign({ element1, element2, ... }).`
- `<container>.assign(<iter begin>, <iter end>).`

```
std::vector<int> v {1, 2, 3, 4, 5};           // v = {1, 2, 3, 4, 5}
std::list<int> l ( v.begin( ), v.end( ) );   // l = 1<->2<->3<->4<->5
v.assign( {6, 7, 8, 9} );                   // v = {6, 7, 8, 9}
l.assign( v.begin( ), v.end( ) - 1 );       // l = 6<->7<->8
```



■ Iteration:

■ Use iterator:

```
for (auto i = <container>.begin(); i != <container>.end(); ++i) {  
    // Process each element.  
}
```

■ Use range-based for (C++11):

```
for (auto &e: <container>) {  
    // Process each element.  
}
```

```
std::vector<int> v {2, 2, 1, 3, 1, 3, 5};
```

```
for (auto &e: v)
```

```
    std::cout << e << ' ';
```

```
// Print vector: 2 2 1 3 1 3 5
```

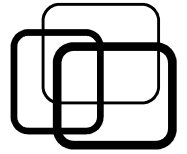
```
std::cout << '\n';
```

```
std::set<int> s( v.begin( ), v.end( ) );
```

```
for (auto &e: s)
```

```
// Print set: 1 2 3 5
```

```
    std::cout << e << ' ';
```



■ Insert/erase elements:

■ <container>.insert:

- Insert value: **insert**(<iter pos>, <value>).
- Insert range: **insert**(<iter pos>, <iter begin>, <iter end>).

■ <container>.erase:

- Erase value: **erase**(<iter pos>).
- Erase range: **erase**(<iter begin>, <iter end>).

```
std::vector<int> v {1, 2, 3, 4};
```

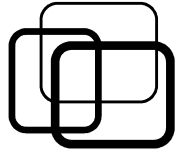
```
std::vector<int> t {5, 6};
```

```
v.insert( v.begin( ) + 2, -1 );           // v = {1, 2, -1, 3, 4}
```

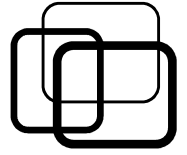
```
v.insert( v.end( ) - 1, t.begin( ), t.end( ) ); // v = {1, 2, -1, 3, 5, 6, 4}
```

```
v.erase( v.begin( ) + 4, v.end( ) );      // v = {1, 2, -1, 3}
```

Contents

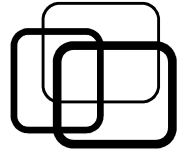


- Overview.
- Containers.
- **Algorithms.**



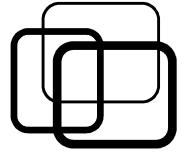
■ Initialize elements:

- `std::fill(<iter begin>, <iter end>, <value>)`.
- `std::generate(<iter begin>, <iter end>, <gen functor>)`.
- `std::generate_n(<iter begin>, n, <gen functor>)`.
- `std::iota(<iter begin>, <iter end>, <start value>)`.
- Practice:
 - Initialize a list of N random numbers.
 - Initialize a list of N integers from N to 1.



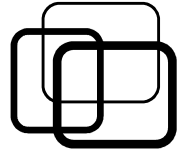
■ Compute elements:

- `std::accumulate`.
- `std::count`.
- `std::count_if`.
- `std::inner_product`.
- `std::sort`.
- Practice:
 - Compute distance of two N-D points.
 - Sort a list of integers in the following order:
 - Even numbers first, then odd numbers.
 - Even numbers in ascending.
 - Odd numbers in descending.



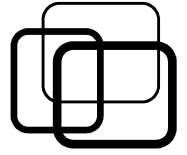
■ Copy elements:

- `std::copy`.
- `std::copy_if`.
- `std::remove_copy_if`.
- Practice:
 - Extract prime numbers from a list of integers.



■ Find elements:

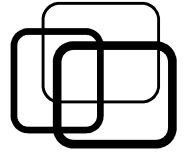
- `std::find.` `(First , Last , Value)`
- `std::find_if.` `(First , Last , Functor)`
- `std::find_if_not.`
- `std::adjacent_find.` `(First , Last , Functor)` Tìm hai phần tử kề nhau thỏa mãn điều kiện nào đó
- `std::search.` `(First1 , Last1 , First2 , Last2)` Tìm một dãy con
- Practice:
 - Check if a string contains all numbers or not. `find_if_not`
 - Check if a list of integers is an arithmetic sequence. `adjacent_find`



■ Remove elements:

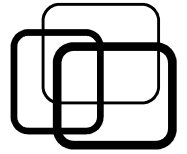
- `std::remove`.
- `std::remove_if`.
- `std::unique`. (First , Last)
- Remove and erase idom. `C.erase(std::remove() , C.end())`
- Practice:
 - 1 ➤ Delete all negative numbers in a list of integers.
 - 2 ➤ Delete multiple consecutive spaces in a string.

1. remove and erase idom (if $x < 0$)
2. remove and erase idom (if `prev = ' ' && cur`)



■ Transform elements:

- `std::transform.` `std::transform(first1 , last1 , first2 , output_des , functor)`
- `std::replace.` `(first , last , old_value , new_value)`
- `std::replace_if.` `(first , last , functor , new_valu`
- Practice:
 - Square root all perfect numbers in a list.
 - Capitalize first letter of each word in a string.

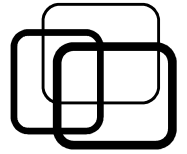


■ Overview:

- Generic data types and algorithms.
- Structure: containers, algorithms, iterators, functors.
- Iterator:
 - Pointer object pointing to container element.
 - Pointer operations: reference, de-reference, jump, distance.
 - Stream iterator, insert iterator.
- Functor:
 - Object acting like function, can store states.
 - By overload operator ().
 - Lambda: anonymous in-place functor.



Summary

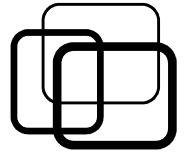


■ Containers:

- Same type elements in specific data structure.
- Dynamic memory management.
- Classifications:
 - Sequence: vector, list, deque.
 - Associative: set, multiset, map, multimap.
 - Unordered: unordered_set, unordered_map.
 - Adapter: stack, queue.
- Operations:
 - Constructor/assign.
 - Iteration: iterator for, range-based for.
 - Insert/erase.



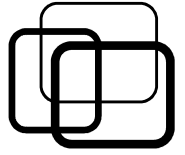
Summary



■ Algorithms:

- Initialize: fill, iota, generate, generate_n.
- Compute: accumulate, count, count_if, sort.
- Copy: copy, copy_if, remove_copy_if.
- Find: find, find_if, find_if_not, adjacent_find, search.
- Remove: remove, remove_if, unique.
- Transform: transform, replace, replace_if.

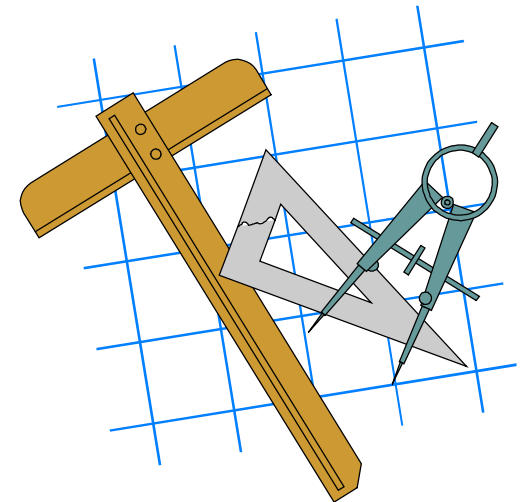


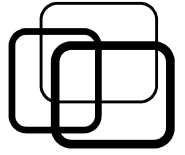


■ Practice 6.1:

Write C++ functions to do the following initializations:
(use Standard Template Library)

- a) Initialize a list of N integers as follow:
 - + Even indexes: $[1..(N+1)/2]$.
 - + Odd indexes: random numbers.
- b) Initialize a list of the first N prime numbers.





■ Practice 6.2:

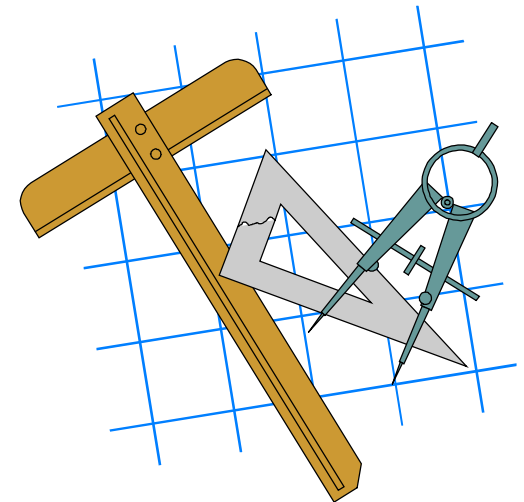
Write C++ function to check if a string is a palindrome or not.

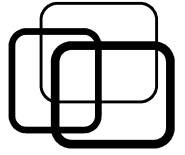
(use Standard Template Library)

Note: a palindrome is string that reads the same backward or forward, spaces and punctuations are not counted.

Example:

- “Race car”.
- “A man, a plan, a canal, Panama!”.





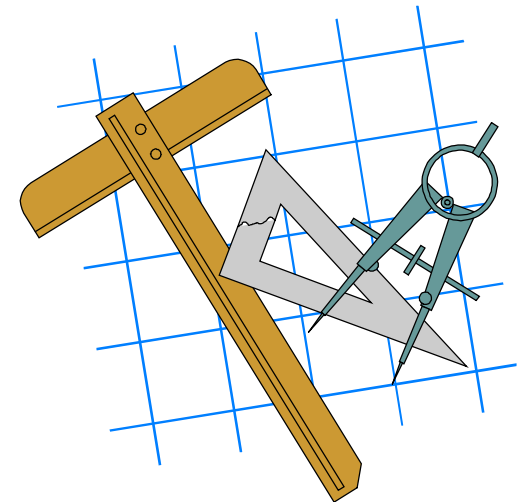
■ Practice 6.3:

Write C++ function to normalize string as follow:
(use Standard Template Library)

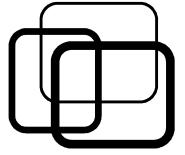
- Eliminate leading and trailing spaces.
- Eliminate multiple consecutive spaces or punctuations.
- Capitalize first letter of each words.

Example:

“ [[[the quick,,, brown fox]]] “
→ “[The Quick, Brown Fox]”



Practice



■ Practice 6.4:

File “INPUT.TXT” stores integers separated by spaces.

Write C++ function to filter out prime numbers from “INPUT.TXT” and write the result to file “OUTPUT.TXT” with integers separated by ‘\n’.

(use Standard Template Library)

INPUT.TXT

1 3 5

2 4

7 13 12

6 9 11

OUTPUT.TXT

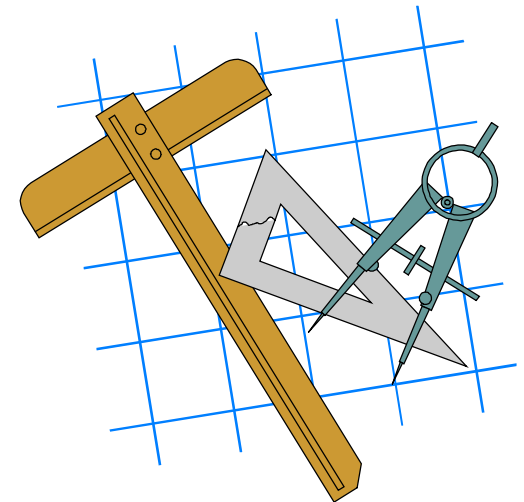
1

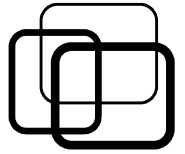
4

12

6

9





■ Practice 6.5:

A student has name and GPA.

Given a list of students, write C++ function to print student counts grouped by GPA in descending order of the counts.

(use Standard Template Library)

Student list:

{ {"John", 7}, {"Eve", 9}, {"Ander", 7}, {"Dora", 8}, {"Tom", 7}, {"Alex", 9} }

Output:

GPA 7: 3 students.

GPA 9: 2 students.

GPA 8: 1 students.

