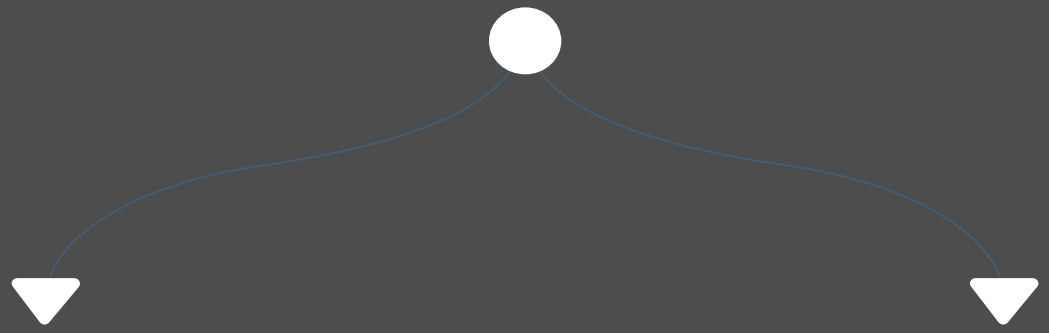
Writing Better C++

By
Vishal Chovatiya

Content



Applying STL

Applying C++ Moderness

CppIndia

Motivation

Developing standard library algorithm intuition

Practical Way Of Learning The standard library Algorithms

Compact & Expressive Code
= Less Maintenance
= Less Errors/Bugs
= Less Learning Curve

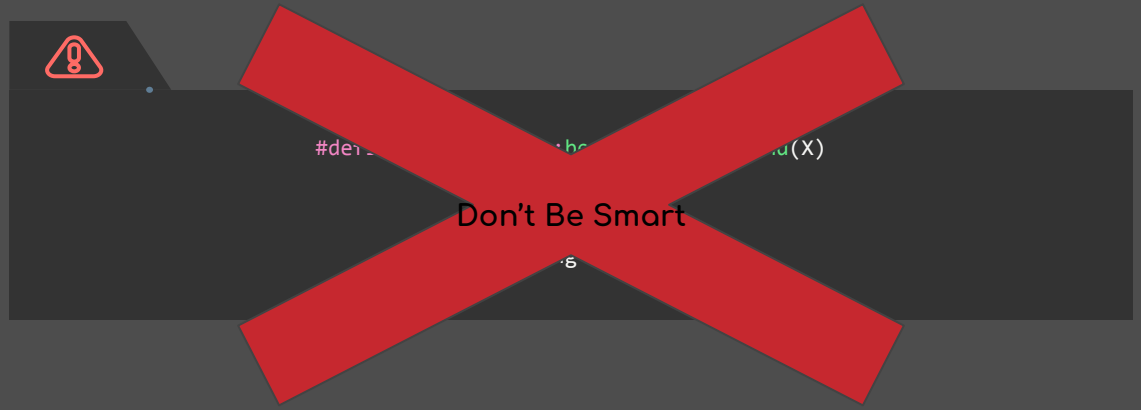
Interview preparation

Writing compact code
i.e. "no raw loops"

CppIndia

Source & Credits

- Conor Hoekstra
- Sean parent
- leetcode.com


Don't Be Smart

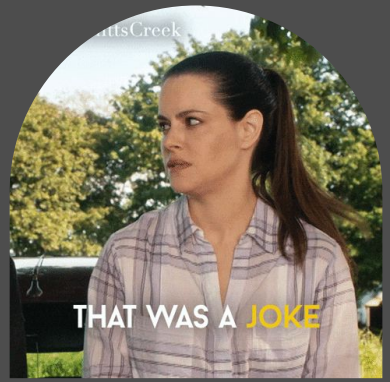
Reverse The String

```
auto str = "ABC"s;

// ?

assert(str == "CBA");
```

Reverse The String




```
auto str = "ABC"s;

str = string(rbegin(str), rend(str));

assert(str == "CBA");
```

Reverse The String

```
#define ALL(X) std::begin(X), std::end(X)
```

```
auto str = "ABC"s;

reverse(ALL(str));

assert(str == "CBA");
```

String To Lowercase

```
auto str = "ABAACCBDBB"s;

// ?

assert(str == "abaaccbdbb");
```

String To Lowercase

```
auto str = "ABAACCBDBB"s;

transform(ALL(str), begin(str), ::tolower);

assert(str == "abaaccbdbb");
```

Addition

```
vector<string>  numbers{"000", "111", "011"};
```

```
int binary_addition(const vector<string>& n) {
    return accumulate(ALL(n), 0,
    [](int res, const string &input) {
        return res + stoi(input, nullptr, 2);
    });
}
```

```
assert(binary_addition(numbers) == 10);
```

Palindrom

```
auto s = "aabaa"s;
```

```
bool is_palindrome(const string &s) {
    return s == string(rbegin(s), rend(s));
}
```

```
assert(is_palindrome(s));
```

Palindrom

```
auto s = "aabaa"s;

bool is_palindrome(const string &str) {
    auto s = begin(str);
    auto m = s + (str.length() / 2);
    return equal(s, m, rbegin(str));
}

assert(is_palindrome(s));
```

Slide To End

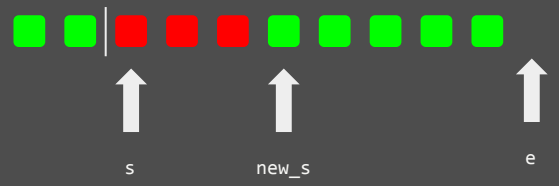
```
vector<string> words{
    "disagree",
    "useful",
    "1",
    "2",
    "3",
    "satisfy",
    "disgusted",
    "cub",
    "cooing",
    "wrong",
};
```

```
vector<string> slide_to_end(const vector<string> &words, uint32_t start, uint32_t cnt) {
    auto result(words);

    auto s = next(begin(result), start);
    auto e = end(result);
    auto new_s = next(s, cnt);

    rotate(s, new_s, e);

    return result;
}
```




```
assert((slide_to_end(words, 2, 3) == vector<string>{"disagree",
                                                    "useful",
                                                    "satisfy",
                                                    "disgusted",
                                                    "cub",
                                                    "cooing",
                                                    "wrong",
                                                    "1",
                                                    "2",
                                                    "3",
                                                    }));
```

Collect In Beginning

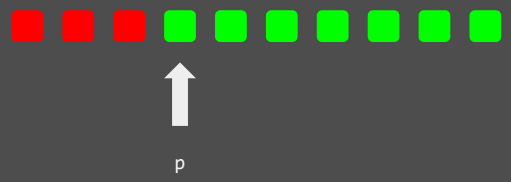
```
vector<string> words{
    "disagree",
    "temporary",
    "useful",
    "third",
    "permissible",
    "satisfy",
    "disgusted",
    "cub",
    "cooing",
    "wrong",
};
```

Sort by parity

Move zeros to end

What if we wants to segregate 0's, 1's, 2's, 3's?

```
vector<string> slide_to_begin(const vector<string> &words, uint32_t char_cnt) {
    auto result(words);
    auto p = [&](const auto &s) { return s.length() <= char_cnt; };
    stable_partition(ALL(result), p);
    return result;
}
```




```
assert((slide_to_begin(words, 5) == vector<string>{
        "third",
        "cub",
        "wrong",
        "disagree",
        "temporary",
        "useful",
        "permissible",
        "satisfy",
        "disgusted",
        "cooing",
    }));
```

Concentrate To Postition

```
vector<string> words{
    "disagree",
    "temporary",
    "useful",
    "third",
    "permissible",
    "thesis",
    "disgusted",
    "cub",
    "type",
    "satisfy",
};
```

```
vector<string> concentrate_to_pos(vector<string> &words, uint32_t pos) {
    auto p = [](const string &str) { return str[0] == 't'; };
    auto new_s = begin(words) + pos;

    stable_partition(begin(words), new_s, not_fn(p));
    stable_partition(new_s, end(words), p);

    return words;
}
```


```
assert((concentrate_to_pos(words, 3) == vector<string>{"disagree",
                                                        "useful",
                                                        "temporary",
                                                        "third",
                                                        "thesis",
                                                        "type",
                                                        "permissible",
                                                        "disgusted",
                                                        "cub",
                                                        "satisfy"}));
```

Concentrate To Postition

```
vector<string> words{
    "disagree",
    "temporary",
    "useful",
    "third",
    "permissible",
    "thesis",
    "disgusted",
    "cub",
    "type",
    "satisfy",
};
```

```
vector<string> concentrate_to_pos(vector<string> &words, uint32_t pos) {
    auto p = [](const string &str) { return str[0] == 't'; };
    auto new_s = begin(words) + pos;

    stable_partition(begin(words), new_s, not_fn(p));
    stable_partition(new_s, end(words), p);

    return words;
}
```


```
assert((concentrate_to_pos(words, 3) == vector<string>{"disagree",
                                                        "useful",
                                                        "temporary",
                                                        "third",
                                                        "thesis",
                                                        "type",
                                                        "permissible",
                                                        "disgusted",
                                                        "cub",
                                                        "satisfy"}));
```

Concentrate To Postition

```
vector<string> words{
    "disagree",
    "temporary",
    "useful",
    "third",
    "permissible",
    "thesis",
    "disgusted",
    "cub",
    "type",
    "satisfy",
};
```

```
vector<string> concentrate_to_pos(vector<string> &words, uint32_t pos) {
    auto p = [](const string &str) { return str[0] == 't'; };
    auto new_s = begin(words) + pos;

    stable_partition(begin(words), new_s, not_fn(p));
    stable_partition(new_s, end(words), p);

    return words;
}
```


```
assert((concentrate_to_pos(words, 3) == vector<string>{"disagree",
                                                        "useful",
                                                        "temporary",
                                                        "third",
                                                        "thesis",
                                                        "type",
                                                        "permissible",
                                                        "disgusted",
                                                        "cub",
                                                        "satisfy"}));
```

Top N Things

```
using language = string;
using popularity = uint64_t;

vector<pair<language, popularity>>  data{
    {"Mandarin", 1e+7},
    {"Spanish", 1e+2},
    {"Hindi", 1e+6},
    {"Tamil", 1e+5},
    {"English", 1e+9},
    {"Gujarati", 1e+0},
    {"Telugu", 1e+3},
    {"Kannada", 1e+5},
};
```

```
auto top_popular_langs(const vector<pair<language, popularity>> &words, uint8_t n) {
    auto result(words);

    auto p = [](const auto &l1, const auto &l2) {
        return l1.second > l2.second;
    };

    auto s = begin(result);
    auto m = s + n;
    nth_element(s, m, end(result), p);

    return decltype(result)(begin(result), begin(result)+n);
}
```

```
assert((top_popular_langs(data, 2) == decltype(data){
    {"Mandarin", 1e+7},
    {"English", 1e+9},
}));
```

Remove Duplicates

```
auto v = vector<int>{2, 5, 2, 1, 3, 2, 4, 5, 2,
                     7, 4, 2, 3, 5, 7, 5, 4, 3, 5, 5, 4};
```

```
set<int>    s(ALL(v)); // Order will not be guaranteed
assert((s == decltype(s){1, 2, 3, 4, 5, 7}));
```

```
auto e = copy_if(ALL(v), begin(v),
        [s = unordered_set<int>{}](int& v) mutable {
            if(!s.contains(v)) {
                s.insert(v);
                return true;
            }
            return false;
        });
v.erase(e, v.end());
```

```
const auto &[it, is_inserted] = s.insert(v);
return is_inserted;
```

```
assert((v == decltype(v){2, 5, 1, 3, 4, 7}));
```

Alphabetical Sort

```
vector<string> words{"Python",
                     "C",
                     "C++",
                     "JAVA",
                     "C_Sharp",
                     "ASSEMBLY",
                     "FORTRAN",
                     "HTML",
                     "NODEjs"};
```

```
assert((sort_lexicographically(words) == vector<string>{
                                         "ASSEMBLY",
                                         "C",
                                         "C++",
                                         "C_Sharp",
                                         "FORTRAN",
                                         "HTML",
                                         "JAVA",
                                         "NODEjs",
                                         "Python",
                                       }));
```

```
vector<string> sort_lexicographically(const vector<string> &words) {
    auto result(words);

    auto p = [](const auto &s1, const auto &s2) {
        return lexicographical_compare(ALL(s1), ALL(s2));
    };

    sort(ALL(result), p);
    return result;
}
```

Query For Sorted Vector

```
vector<int> v = {-11, -2, -3, -1, 1, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9};
```

1 How many time 5 is repeated?

```
<=      >
const auto [l, u] = equal_range(ALL(v), 5);
assert(distance(l, u) == 3);
```

2 How many negative elements are there?

```
auto p = [](const auto &n){ return n < 0;};
auto pp = partition_point(ALL(v), p);
assert(distance(begin(v), pp) == 4);
```

3 Insert at apt location to maintain container sorted.

```
auto pos = lower_bound(ALL(v), 2);
v.insert(pos, 2);
assert((v ==
    decltype(v){-11, -2, -3, -1, 1, 2, 3,
                3, 3, 4, 5, 5, 5, 6, 7, 8, 9}));
```

4 Remove all the elements greater than & equal to 5.

```
auto t = lower_bound(ALL(v), 5);
v.erase(t, end(v));
assert((v ==
    decltype(v){-11, -2, -3, -1,
                1, 2, 3, 3, 3, 4}));
```

Use Set Instead

```
multiset<int> s = {-11, -2, -3, -1, 1, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9};
```

1 How many time 5 is repeated?

```
const auto [l, u] = s.equal_range(5);
assert(distance(l, u) == 3);
```

2 How many negative elements are there?

```
auto u = s.upper_bound(0);
assert(distance(begin(s), u) == 4);
```

3 Insert at apt location to maintain container sorted.

```
auto pos = s.insert(2);

assert((s ==
    decltype(s){-11, -2, -3, -1, 1, 2, 3,
                3, 3, 4, 5, 5, 5, 6, 7, 8, 9}));
```

4 Remove all the elements greater than & equal to 5.

```
auto e = s.lower_bound(5);
s.erase(e, end(s));

assert((s == decltype(s){-11, -2, -3, -1,
                         1, 2, 3, 3, 3, 4}));
```

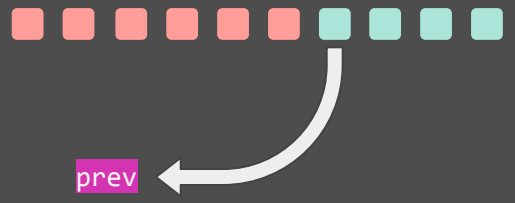
Frequency Sort

```
vector<int> v = {2, 5, 2, 1, 3, 2, 4, 5, 2, 7, 4, 2, 3, 5, 7, 5, 4, 3, 5, 5, 4};
```

```
unordered_map<int, int>   freq_of;
for (auto &&e : v) freq_of[e]++;
// [(7, 2), (4, 4), (3, 3), (1, 1), (5, 6), (2, 5)]
```

```
auto p = [] (auto &p1, auto &p2) { return p1.second > p2.second; };
set<pair<int, int>, decltype(p)>    s(ALL(freq_of));
// [(5, 6), (2, 5), (4, 4), (3, 3), (7, 2), (1, 1)]
```

```
for_each(ALL(s), [prev = begin(v)](auto &p) mutable {
    prev = fill_n(prev, p.second, p.first);
});
```




```
assert((v ==
    decltype(v){5, 5, 5, 5, 5, 5, 2, 2, 2, 2, 2,
                4, 4, 4, 4, 3, 3, 3, 7, 7, 1}));
```

Consecutive Unique

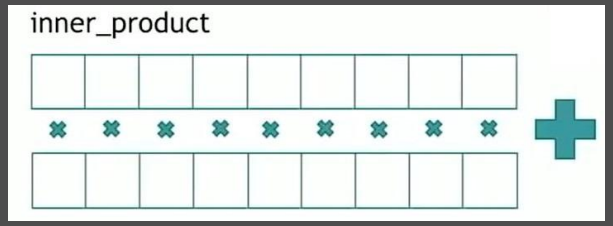
```
auto ints = "1          2  3   4 5  6   7     "s;
```

```
auto t = unique(ALL(ints), [](auto l, auto r){
    return l == ' ' && r == ' ';
});
```

```
ints.erase(t, end(ints));
```

```
assert(ints == "1 2 3 4 5 6 7 "s);
```

Count Equal Adjacent Characters



From Jonathan Boccara's Talk

```
auto str = "ABAAACCBDBB"s;Ṅ
```

```
int count_adjecent_element(const string& s){
    return inner_product(ALL(s), next(begin(s)),
                         0, plus<>(), equal_to<>());
}
```

```
assert(count_adjecent_element(str) == 4);
```

K Closest Points to Origin

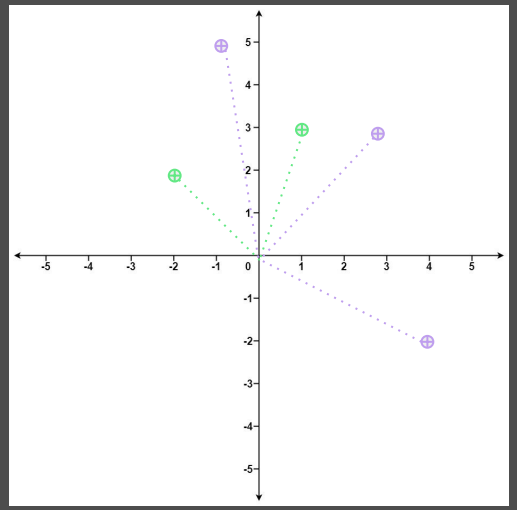
```
vector<pair<int, int>> v = {{3, 3}, {5, -1}, {-2, 4}};
uint32_t k = 2;
```

```
auto dist(const pair<int, int> &p1, const pair<int, int> &p2 = {0, 0}) {
    return sqrt(pow(p1.first - p2.first, 2)
       + pow(p1.second - p2.second, 2) * 1.0);
};

auto p = [](auto &p1, auto &p2) {
    return dist(p1) < dist(p2);
};
```

```
vector<pair<int, int>> o(k);
partial_sort_copy(ALL(v), ALL(o), p);
```

```
assert((o == decltype(o){{3, 3}, {-2, 4}}));
```



euclidean distance = √(x2 - x1)^2 + (y2 - y1)^2)

Input : [1, 3], [-2, 2], k = 1

Output : [-2, 2]

Distance b/w closest & farthest points

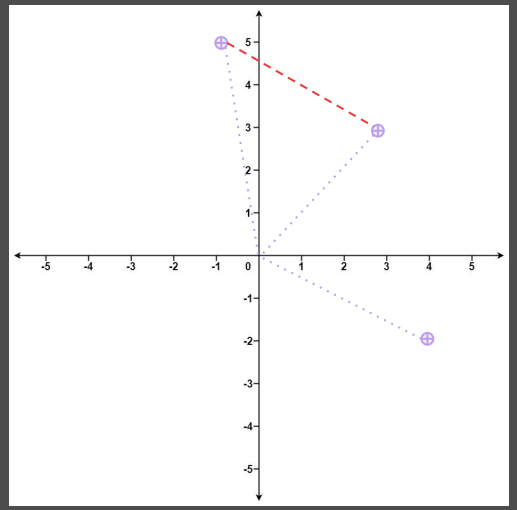
```
vector<pair<int, int>> v = {{3, 3}, {5, -1}, {-2, 4}};
```

```
auto dist(const pair<int, int> &p1, const pair<int, int> &p2 = {0, 0}) {
    return sqrt(pow(p1.first - p2.first, 2)
      + pow(p1.second - p2.second, 2) * 1.0);
};

auto p = [](auto &p1, auto &p2) {
    return dist(p1) < dist(p2);
};
```

```
const auto [min, max] = minmax_element(ALL(v), p);
auto d = dist(*max, *min);
```

```
assert(d == 4.4f); // comparison not valid
```



euclidean distance = √(x2 - x1)^2 + (y2 - y1)^2)

Search in Rotated Sorted Array

sorted array = [0, 1, 2, 4, 5, 6, 7]

sorted rotated array = [4, 5, 6, 7, 0, 1, 2]

t

```
vector<int> v = {4, 5, 6, 7, 0, 1, 2};
```

```
int sorted_rotated_search(const vector<int> &v, int k) {
    auto t = is_sorted_until(ALL(v));  // partition point
    auto s = begin(v);
    auto e = end(v);

    if(k > (*s))
        return binary_search(s, t, k);
    return binary_search(t, e, k);
}
```

```
assert(sorted_rotated_search(v, 7) == true);
```

Reverse Only Letters

```
auto s = "1ab2cd"s;

auto r = ""s;
copy_if(ALL(s), back_inserter(r), ::isalpha);

transform(ALL(s), begin(s),
[i = rbegin(r)](auto &c) mutable {
    return isalpha(c) ? *i++ : c;
});

assert(s == "1dc2ba"s);
```

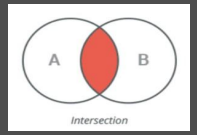
Set Algorithm

```
vector<int> v1{2, 3, 1, 5, 4};  // 1, 2, 3, 4, 5
vector<int> v2{8, 5, 7, 6, 4};  //          4, 5, 6, 7, 8
```

```
// Another way of sorting things
set<int>    s1(ALL(v1));
set<int>    s2(ALL(v2));
```

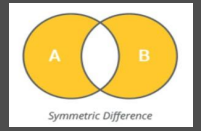
1 What are the common elements?

```
vector<int> i;
set_intersection(ALL(s1), ALL(s2), back_inserter(i));
assert((i == decltype(i){4, 5}));
```



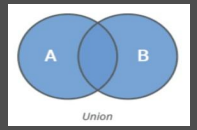
2 What are non-common elements?

```
vector<int> sd;
set_symmetric_difference(ALL(s1), ALL(s2),
                                  back_inserter(sd));
assert((sd == decltype(sd){1, 2, 3, 6, 7, 8}));
```



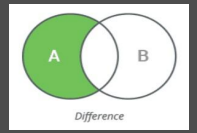
3 What are the unique elements?

```
vector<int> u;
set_union(ALL(s1), ALL(s2), back_inserter(u));
assert((u == decltype(u){1, 2, 3, 4, 5, 6, 7, 8}));
```



4 What are different elements in first set?

```
vector<int> diff;
set_difference(ALL(s1), ALL(s2), back_inserter(diff));
assert((diff == decltype(diff){1, 2, 3}));
```




Is that all you got?

User define literals

```
#define KB(n)    ((n) * 1024)
#define MB(n)    ((n) * KB(1024))

uint64_t kilo_bytes = KB(1);
uint64_t mega_bytes = MB(1);
```

```
uint64_t kilo_bytes = 1_KB;
uint64_t mega_bytes = 1_MB;
```

```
using ull = unsigned long long;

constexpr ull operator"" _KB(ull no) {
    return no * 1024;
}

constexpr ull operator"" _MB(ull no) {
    return no * (1024_KB);
}
```

```
assert(kilo_bytes == 1024);
assert(mega_bytes == (1024 * 1024));
```

Digit Separator

```
uint64_t addr = 0xA0000000;
```

```
uint64_t addr = 0xA000'0000;
```

```
uint32_t integer  = 1'0'0'0'000;
uint32_t binary   = 0b0001'0010'0111'1111;
```

Structured Binding

```
vector<uint32_t> v = {2, 1, 7, 3, 5, 4};

pair<decltype(begin(v)), decltype(begin(v))> p = minmax_element(ALL(v));

uint32_t min = *p.first;
uint32_t max = *p.second;

assert(min == 1);
assert(max == 7);
```

Structured Binding

```
vector<uint32_t> v = {2, 1, 7, 3, 5, 4};
```

```
const auto& [min, max] = minmax_element(ALL(v));
```

```
assert(*min == 1);
assert(*max == 7);
```

```
unordered_map<int, int>   freq_of;
for (const auto &e : v) freq_of[e]++;

for (const auto& [n, rep] : freq_of) {
    cout << n << " repeated " << rep << " times" << endl;
}
```

Fold Expression

```
template <typename... Args>
void print(Args... args) {
    ((cout << args << endl), ...);
}

print(1, 1.1, "Hello");
```

```
template <typename... Args>
auto push_all(vector<int> &v, Args... args) {
    return ((v.push_back(args)), ...);
}

auto v = vector<int>{};
push_all(v, 1, 2, 3);
```

```
template <typename... Args>
auto concat(Args... args) {
    return (... + args);
}

assert(concat("ABC "s, "XYZ"s) == "ABC XYZ"s);
assert(concat(1, 2, 3, 4, 5) == 15);
```

Finding Array Size

```
int x[20];
```

```
#define ARRAY_SIZE(x)    (sizeof(x)/sizeof(x[0]))
```

```
assert(ARRAY_SIZE(x) == 20);
```

```
template <typename T, size_t N>
constexpr size_t array_size(const T (&)[N]) {
    return N;
}
```

```
assert(array_size(x) == 20);
```

Higher order returning function

```cpp
uint32_t pc = 0xDEADBEAB;
uint32_t alu = 0xBEDA55;
```

```cpp
constexpr uint32_t ROM_ADRWIDTH = 16;
constexpr uint32_t ALU_ADRWIDTH = 8;
```

```cpp
uint32_t bitlength_upto(uint32_t data, uint8_t len){
    return data & ~(0xFFFF'FFFF << len)
}
```

```cpp
assert(bitlength_upto(pc, ROM_ADRWIDTH) == 0xBEAB);
assert(bitlength_upto(alu, ALU_ADRWIDTH) == 0x55);
```

```cpp
auto bitlength_upto = [](uint8_t len) {
    return [bit_len = len](uint32_t data) {
        return data & ~(0xFFFF'FFFF << bit_len);
    };
};
```

```cpp
auto upto_rom_addr = bitlength_upto(ROM_ADRWIDTH);
auto upto_alu_addr = bitlength_upto(ALU_ADRWIDTH);
```

```cpp
assert(upto_rom_addr(pc) == 0xBEAB);
assert(upto_alu_addr(alu) == 0x55);
```

Lambda Use-case

```
bool something::initialize() {
    if (!init_step0()) {
        cerr << "Step 0 failed\n";
        return false;
    }

    if (!init_step1()) {
        cerr << "Step 1 failed\n";
        return false;
    }

    if (!init_step2()) {
        cerr << "Step 2 failed\n";
        return false;
    }

    return true;
}
```

```
bool something::initialize() {
    const auto try_step = [](const char *msg, auto f) {
        if (f()) return true;
        cerr << msg << " failed\n";
        return false;
    };

    return try_step("Step 0", init_step0)
        && try_step("Step 1", init_step1)
        && try_step("Step 2", init_step2);
}
```

Scope Bound Objects

```
auto freq_of = map<int, int>{{3, 1}, {1, 1}, {2, 7}};

auto itr = freq_of.find(2);
if (itr != freq_of.end()) {
  assert(itr->second == 7);
}
else {
  // Do something...
}
// itr is still available & can be mess around
```

```
if (auto itr{freq_of.find(2)}; itr != freq_of.end()) {
  assert(itr->second == 7);
}
else {
  // Do something...
} // itr is not available here at all
```

```
switch (const int32_t c{getchar()}; c) {
    case 'a':
        cout << "move_left" << endl;
        break;
    case 'd':
        cout << "move_right" << endl;
        Break;
    // GNU Compiler Extension, not C++ standard
    case '0' ... '9':
        cout << "What you are doing ?" <<endl;
        break;
    default:
        cout << "invalid input: " << c << endl;
}
```

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

www.vishalchovatiya.com

linkedin.com/in/vishal-chovatiya

CppIndia

Split The String

```
auto path = "/root/home/vishal"s;
```

```
vector<string>  split(const string &s, char delim) {
    vector<string> result;
    stringstream ss(s);
    string item;

    while (getline(ss, item, delim)) {
        result.push_back(move(item));
    }

    return result;
}
```

```
assert((split(path, '/')
        == vector<string>{"", "root", "home", "vishal"}));
```

Split The String

```
auto path = "//root//home//vishal"s;
```

```
vector<string> split(const string& str, string_view pattern) {
    const auto r = regex(pattern.data());
    return vector<string>{
        sregex_token_iterator(ALL(str), r, -1),
        sregex_token_iterator()
    };
}
```

```
assert((split(path, '//')
        == vector<string>{"", "root", "home", "vishal"}));
```