Object Oriented Programming by C++

Generic Programming

Template, STL, Iterator, Lambda Function

2017. 8.

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr
Web: http://mobilelab.khu.ac.kr/

Textbook & Copyright

- Textbook: http://python.cs.southern.edu/cppbook/progcpp.pdf
- Sample Codes: https://github.com/halterman/CppBook-SourceCode

Fundamentals of

C++

Programming

DRAFT

Richard L. Halterman
School of Computing
Southern Adventist University

July 21, 2017

Preface

Legal Notices and Information

Permission is hereby granted to make hardcopies and freely distribute the material herein under the following conditions:

- The copyright and this legal notice must appear in any copies of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely use this document in their classes as a primary or optional textbook under the conditions specified above.

A local electronic copy of this document may be made under the terms specified for hard copies:

- The copyright and these terms of use must appear in any electronic representation of this document made in whole or in part.
- None of material herein can be sold or otherwise distributed in an electronic form for commercial purposes without written permission of the copyright holder.
- Instructors at any educational institution may freely store this document in electronic form on a local server as a primary or optional textbook under the conditions specified above.

Additionally, a hardcopy or a local electronic copy must contain the uniform resource locator (URL) providing a link to the original content so the reader can check for updated and corrected content. The current standard URL is `http://python.cs.southern.edu/cppbook/progcpp.pdf`.

If you are an instructor using this book in one or more of your courses, please let me know. Keeping track of how and where this book is used helps me justify to my employer that it is providing a useful service to the community and worthy of the time I spend working on it. Simply send a message to `halterman@southern.edu` with your name, your institution, and the course(s) in which you use it.

The source code for all labeled listings is available at

`https://github.com/halterman/CppBook-SourceCode`.

Contents

- Generic Function
- Generic Function Evolution
- Class Template
- Standard Template Library (STL)
- Containers
- Iterators
- Iterators & Template
- Lambda Function
- Closure
- Standard Algorithms

Not a Specific term, Generic term

```
Define integers.
Calculate integers.
Compare integers.
Find maximum integer.
```

Specific term - Integer

```
Define SOMETHING.
Calculate SOMETHING.
Compare SOMETHING.
Find maximum SOMETHING.
```

Generic term - Something

template Statement

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

/*
 *  less_than(a, b)
 *      Returns true if a < b; otherwise, returns
 *      false.
 */
template <typename T>
bool less_than(T a, T b) {
    return a < b;
}

int main() {
    std::cout << less_than(2, 3) << '\n';
    std::cout << less_than(2.2, 2.7) << '\n';
    std::cout << less_than(2.7, 2.2) << '\n';

    std::string word1 = "ABC", word2 = "XYZ";
    std::cout << less_than(word1, word2) << '\n';
    std::cout << less_than(word2, word1) << '\n';
}
```

```
1
1
0
1
0
```

- **T** is a type parameter.
- Specific functions (for each data types) are generated by C++ automatically - but invisible to the programmer

template Statement (using class term)

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

/*
 *  less_than(a, b)
 *      Returns true if a < b; otherwise, returns
 *      false.
 */
template <class T>
bool less_than(T a, T b) {
    return a < b;
}

int main() {
    std::cout << less_than(2, 3) << '\n';
    std::cout << less_than(2.2, 2.7) << '\n';
    std::cout << less_than(2.7, 2.2) << '\n';

    std::string word1 = "ABC", word2 = "XYZ";
    std::cout << less_than(word1, word2) << '\n';
    std::cout << less_than(word2, word1) << '\n';
}
```

```
1
1
0
1
0
```

- Same with `typename` statement case

template Statement (enhanced version)

```cpp
#include <iostream>
#include <string>

/*
 *  less_than(a, b)
 *     Returns true if a < b; otherwise, returns
 *     false.
 */
template <typename T>
bool less_than(const T& a, const T& b) {
    return a < b;
}

int main() {
    std::cout << less_than(2, 3) << '\n';
    std::cout << less_than(2.2, 2.7) << '\n';
    std::cout << less_than(2.7, 2.2) << '\n';

    std::string word1 = "ABC", word2 = "XYZ";
    std::cout << less_than(word1, word2) << '\n';
    std::cout << less_than(word2, word1) << '\n';
}
```

```
1
1
0
1
0
```

- Speed up by using ***call by reference***.
- Keep safety using `const` statement

Multiple Type Combination

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

/*
 *  less_than(a, b)
 *      Returns true if a < b; otherwise, returns
 *      false.
 */
template <typename T, typename V>
bool less_than(const T& a, const V& b) {
    return a < b;
}

int main() {
    std::cout << less_than(2, 3) << '\n';
    std::cout << less_than(2.2, 2) << '\n';
    std::cout << less_than(2, 2.2) << '\n';

    std::string word1 = "ABC", word2 = "XYZ";
    std::cout << less_than(word1, word2) << '\n';
    std::cout << less_than(word2, word1) << '\n';
}
```

```
1
0
1
1
0
```

- Type of a and b can be different

Code Example

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

template <typename T>
T sum(const std::vector<T>& v) {
    T result = 0;
    for (T elem : v)
        result += elem;
    return result;
}

int main()
{
    std::vector<double> v {10.0, 20.0, 30.0};
    std::vector<int> w {10, 20, 30};
    std::cout << sum(v) << '\n';
    std::cout << sum(w) << '\n';
}
```

```
60
60
```

Non-type Parameters

Listing 19.4: templatescale.cpp

```
#include <iostream>

template <int N>
int scale(int value) {
    return value * N;
}

int main() {
    std::cout << scale<3>(5) << '\n';
    std::cout << scale<4>(10) << '\n';
}
```

- *N is* 3 *or* 4

```
15
40
```

Listing 19.5: templatescale2.cpp

```
#include <iostream>

template <typename T, int N>
T scale(const T& value) {
    return value * N;
}

int main() {
    std::cout << scale<double, 3>(5.3) << '\n';
    std::cout << scale<int, 4>(10) << '\n';
}
```

- *N is* 3 *or* 4
- *T is* double *or* int

```
15.9
40
```

Class definition using Template Statement

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

template <typename T>
class Point {
public:
    T x;
    T y;
    Point(T x, T y): x(x), y(y) {}
};

int main()
{
    Point<int> pixel1(10, 10);
    Point<double> pixel2(10.0, 20.0);
}
```

- With class templates we can specify the pattern or structure of a class of objects in a type-independent way.
- Rather than providing two separate classes, we can write one class template let the compiler instantiate the coordinates as the particular program requires.

Use Visual Studio !! Goorm can't support user defined header file

Code Review

Listing 19.7-19.8

Polymorphism Application

Definition of STL

- The C++ Standard Template Library
  - Leverages templates to provide a rich collection of standard generic containers and algorithms to manipulate the containers and process the elements they contain
  - Contains a number of generic functions and classes built with templates
  - includes std::vector class as we already familiar

Standard Containers

- A container is ***a holder object that stores a collection of other objects*** (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.
- The container manages the storage space for its elements and ***provides member functions to access them***, either directly or through iterators (reference objects with similar properties to pointers).
- Containers replicate structures very commonly used in programming: ***dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)***, etc.

Reference: http://www.cplusplus.com/reference/stl/

Standard Containers

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

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

Associative containers:

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

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

- We already used `vector` frequently

Reference: http://www.cplusplus.com/reference/stl/

Standard Containers Example (List)

```
#include <iostream>
#include <list>   // Use the standard doubly linked list class

int main() {
    bool done = false;
    char command;
    int value;
    std::list<int> mylist;    // Initially empty

    while (!done) {
        std::cout << "I)nsert <item>  P)rint  L)ength  E)rase Q)uit >>";
        std::cin >> command;
        switch (command) {
          case 'I':   // Insert a new element into the list
          case 'i':
            if (std::cin >> value)
                mylist.push_back(value);
            else
                done = true;
            break;
          case 'P':  // Print the contents of the list
          case 'p':
            for (const auto& elem : mylist)
                std::cout << elem << ' ';
            std::cout << '\n';
            break;
          case 'L':  // Print the list's length
          case 'l':
            std::cout << "Number of elements: " << mylist.size() << '\n';
            break;
          case 'E':  // Erase the list
          case 'e':
            mylist.clear();
            break;
          case 'Q':  // Exit the loop (and the program)
          case 'q':
            done = true;
            break;
        }
    }
}
```

Standard Pointer-like Objects

- An iterator is ***an object that allows a client to traverse and access elements*** of a data structure in an implementation independent way
- C++ defines ***two global functions, std::begin and std::end***, that produce iterators to the front and back, respectively, of a data structure like a vector or static array.
- Containers defined int the STL provide begin and end methods that serve the same purpose; ***for example, if v is a std::vector, std::begin(v) returns the same iterator as the call v.begin()***
- Functions in the standard library that ***accept iterators as arguments*** rather than arrays or vectors work equally well with both vectors and arrays

Standard Pointer-like Objects

- Iterators provides the following methods:
  - operator*: used to ***access the element at the itertator's current position***. The syntax is exactly like pointer dereferencing
  - operator++: used to ***move the iterator to the next element*** within the data structure. The syntax is exactly like pointer arithmetic
  - operator!=: used to determine ***whether two iterator objects currently refer to different elements*** within the data structure

Iterators Example 1

```
#include <iostream>
#include <vector>

int main() {
    // Make a simple integer vector
    std::vector<int> vec {10, 20, 30, 40, 50};

    // Direct an iterator to the vector's first element
    std::vector<int>::iterator iter = std::begin(vec);

    // Print the element referenced by the iterator
    std::cout << *iter << '\n';

    // Advance the iterator
    iter++;

    // See where the iterator is now
    std::cout << *iter << '\n';
}
```

```
10
20
```

- The type of `iter` is `std::vector<int>::iterator`
- This complicated expression indicates that iterator is a type defined within the `std::vector<int>` type
- A shorter way to express this statement takes advantage of the compiler's ability to infer the variable's type from its context:

```
auto iter = std::begin(vec);
```

Iterators Example 2

```
#include <iostream>
#include <vector>

int main() {
    // Make a simple integer vector
    std::vector<int> vec {10, 20, 30, 40, 50};

    // Print the contents of the vector
    for (auto iter = std::begin(vec);
         iter != std::end(vec);
         iter++)
        std::cout << *iter << ' ‘;

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

```
10 20 30 40 50
```

- Checks each time through the loop to ensure the iterator object has not run off the back end of the vector.

Iterators Example 3

```
#include <iostream>
#include <vector>

int main() {
    // Make a static integer array
    int arr[] =  {10, 20, 30, 40, 50};

    // Print the contents of the array
    for (auto iter = std::begin(arr);
         iter != std::end(arr); iter++)
        std::cout << *iter << ' ';
    std::cout << '\n';
}
```

```
10 20 30 40 50
```

- The `std::begin` and `std::end` functions are overloaded to work with vector objects, arrays, and other container classes found in the standard library

Iterators Example 4

```
#include <iostream>
#include <vector>

// Print the contents of vector v, traversing with a
// caller supplied increment value (inc)
void print(const std::vector<int>& v, int inc) {
    for (auto p = std::begin(v); p != std::end(v); p += inc)
        std::cout << *p << ' ';
    std::cout << '\n';
}

// Print the contents of the vector v backwards,
// traversing with a caller supplied decrement value (dec)
void print_reverse(const std::vector<int>& v, int dec) {
    auto p = std::end(v);
    while (p != std::begin(v)) {
        p -= dec;
        std::cout << *p << ' ';
    }
    std::cout << '\n';
}

int main() {
   std::vector<int> vec(20);
   for (int i = 0; i < 20; i++)
       vec[i] = i;

   print(vec, 1);
   print(vec, 2);
   print(vec, 4);
   print(vec, 5);
   print(vec, 10);

   std::cout << '\n';

   print_reverse(vec, 1);
   print_reverse(vec, 2);
   print_reverse(vec, 4);
   print_reverse(vec, 5);
   print_reverse(vec, 10);
}
```

- The kind of iterator available to `std::vector` objects is known as a random access itertator.
- Random access iterators behave exactly like a pointers.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 2 4 6 8 10 12 14 16 18
0 4 8 12 16
0 5 10 15
0 10

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
18 16 14 12 10 8 6 4 2 0
16 12 8 4 0
15 10 5 0
10 0
```

Iterators Example 5

```
#include <iostream>
#include <vector>

using Iter = std::vector<int>::iterator;

// Count the elements in a vector of integers that match seek
int count_value(Iter iter_begin, Iter iter_end, int seek) {
    int cnt = 0;
    for (Iter cursor = iter_begin; cursor != iter_end; cursor++)
        if (*cursor == seek)
            cnt++;
    return cnt;
}

int main() {
    std::vector<int> a {34, 5, 12, 5, 8, 5, 11, 2};
    // Count multiple elements
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
    // Count single element
    std::cout << count_value(std::begin(a), std::end(a), 12) << '\n';
    // Count missing element
    std::cout << count_value(std::begin(a), std::end(a), 13) << '\n';
    a = {};  // Try an empty vector
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
}
```

- Searching through iterator

```
3
1
0
0
```

Iterators Example 6 (1/2)

```
#include <iostream>
#include <vector>
#include <array>
#include <list>
#include <string>

// Count the elements in a range that match seek.
// Type Iter is an iterator type working with a container that
// contains elements of type T.  Type T elements must be
// comparable with operator==.
template <typename Iter, typename T>
int count_value(Iter iter_begin, Iter iter_end, const T& seek) {
    int cnt = 0;
    for (auto cursor = iter_begin; cursor != iter_end; cursor++)
        if (*cursor == seek)
            cnt++;
    return cnt;
}

int main() {
    // Test with a vector of integers
    std::cout << "---Vector of integers--------------\n";
    std::vector<int> a {34, 5, 12, 5, 8, 5, 11, 2};
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
    a = {};  // Try an empty vector
    std::cout << count_value(std::begin(a), std::end(a), 5) << '\n';
    std::cout << count_value(std::begin(a), std::end(a), 8) << '\n';

    std::cout << "---STL array of integers-----------\n";
    // Test with a std::array of integers
    std::array<int, 8> arr {34, 5, 12, 5, 8, 5, 11, 2};
    std::cout << count_value(std::begin(arr), std::end(arr), 5) << '\n';
    arr = {};  // Try an empty array
    std::cout << count_value(std::begin(arr), std::end(arr), 5) << '\n';
    std::cout << count_value(std::begin(arr), std::end(arr), 8) << '\n';

    std::cout << "---Primitive C array of integers-----\n";
    // Test with a primitive C array of integers
    int carr[] = {34, 5, 12, 5, 8, 5, 11, 2};
    std::cout << count_value(std::begin(carr), std::end(carr), 5) << '\n';
    std::cout << count_value(std::begin(carr), std::end(carr), 8) << '\n';
```

- Heterogeneous types are consistently managed by `count_value()` function

```
---Vector of integers--------------
3
0
0
---STL array of integers-----------
3
0
0
---Primitive C array of integers-----
3
1
---Vector of strings---------------
3
0
0
---Linked list of strings----------
3
0
0
---Primitive C array of Points-----
2
1
0
```

Iterators Example 6 (2/2)

```
    std::cout << "---Vector of strings---------------\n";
    // Test with a vector of strings
    std::vector<std::string> b {"mae", "al", "pat", "mel", "al",
        "ray", "al"};
    std::cout << count_value(std::begin(b), std::end(b), "al") << '\n';
    b = {};
    std::cout << count_value(std::begin(b), std::end(b), "al") << '\n';
    std::cout << count_value(std::begin(b), std::end(b), "pat") << '\n';

    std::cout << "---Linked list of strings----------\n";
    // Test with a linked list of strings
    std::list<std::string> lst {"mae", "al", "pat", "mel", "al",
        "ray", "al"};
    std::cout << count_value(std::begin(lst), std::end(lst), "al") << '\n';
    lst = {};
    std::cout << count_value(std::begin(lst), std::end(lst), "al") << '\n';
    std::cout << count_value(std::begin(lst), std::end(lst), "pat") << '\n';


    std::cout << "---Primitive C array of Points-----\n";
    struct Point {
        int x;
        int y;
        bool operator==(const Point& other) {
            return x == other.x && y == other.y;
        }
    };
    // Test with a primitive array of Point objects
    Point pts[] =  {{5, 3}, {0, 0}, {5, 3}, {3, 5}, {2, 1}};
    std::cout << count_value(std::begin(pts), std::end(pts), Point{5, 3})
    << '\n';
    std::cout << count_value(std::begin(pts), std::end(pts), Point{3, 5})
    << '\n';
    std::cout << count_value(std::begin(pts), std::end(pts), Point{2, 3})
    << '\n';
}
```

```
---Vector of integers--------------
3
0
0
---STL array of integers-----------
3
0
0
---Primitive C array of integers-----
3
1
---Vector of strings---------------
3
0
0
---Linked list of strings----------
3
0
0
---Primitive C array of Points-----
2
1
0
```

What is this?

- Lambda Function:
  - An ***unnamed function object*** capable of capturing variables in scope
  - ***unnamed function object*** is
    - Simple: generally, single line
    - No name is required: execute generally one time and only when by invoked by s specific caller (= function, object, etc)

What is looks like?

```
[](int x, int y)->int { return  x * y; }
```

C++ supports the definition of anonymous functions via *lambda expressions*. The general form of a lambda expression is

[*capture list*] (*parameter list*) -> *return type* { *statements* }

where:

- *capture list* specifies the calling context to which the function has access (more on this follows)
- *parameter list* is a comma-separated list of parameters as you would find in any function definition
- *return type* is the type of the result the function returns
- *statements* are the statements as you would find in any function definition.

How can we use it? (its a function with no name!)

```
int evaluate(int (*f)(int, int), int x, int y) {
    return f(x, y);
}
```

`evaulate()` needs function,
thus we give lambda function.

```
[](int x, int y)->int { return  x * y; }
```

`evaulate()`
requires two integers

```
int val = evaluate([](int x, int y)->int { return  x * y; }, 2, 3);
```

finally *LabmdaFunction (2, 3);*
and returns *(2 * 3);*.

Simplified Format

Same Meaning

```
[](int x, int y)->int { return  x * y; }

[](int x, int y) { return  x * y; }
```

Code Example

```
#include <iostream>
using namespace std;

int evaluate(int (*f)(int, int), int x, int y)
{
    return f(x, y);
}

int main()
{
    int val;

    // type 1: Normal Statement
    val = evaluate([](int x, int y)->int { return x * y; }, 2, 3);
    cout << val << endl;

    // type 2: Simplified Statement
    val = evaluate([](int x, int y) { return  x * y; }, 2, 3);
    cout << val << endl;

    // type 3: Direct invokaction
    [](int x, int y) { std::cout << x << " " << y << '\n'; } (10, 20);

    // type 4: Lambda function invokation using auto variable
    auto f = [](int x) { return  5*x; };
    std::cout << f(10) << '\n';
}
```

```
6
6
10 20
50
```

Step.1 Function Object

- A function object is any object for which the function call operator is defined
- Class template ***`std::function<>`***
  - it is a ***general-purpose polymorphic function wrapper***
  - Instances of ***`std::function`*** can store, copy, and invoke any ***Callable target***:
    - functions, lambda expressions, bind expressions,
    - or other function objects,
    - as well as pointers to member functions and pointers to data members
  - The stored callable object is called the target of ***`std::function`***

Step.2 Variable Capturing

- See capture-list again @ Slide 27
  - One interesting aspect of lambda functions is that they can be used to create closures
  - A closure is a unit of code (in this case a function-like object) that can capture variables from its surrounding context
- What is it? See next slide…

Code Example

Listing 20.9: closurein.cpp

```
#include <iostream>
#include <functional>

int evaluate2(std::function<int(int, int)> f, int x, int y) {
    return f(x, y);
}

int main() {
    int a;
    std::cout << "Enter an integer: ";
    std::cin >> a;
    std::cout << evaluate2([a](int x, int y) {
                                if (x == a)
                                    x = 0;
                                else
                                    y++;
                                return x + y;
                            }, 2, 3) << '\n';
}
```

Function Object
std::function<>

Variable Capturing
in this case: ‘a’

Lambda Function

Understanding Variable Capturing

- So how does the magic of variable capture really work? It turns out that the way lambdas are implemented is by creating a small class; this class overloads the operator(), so that it acts just like a function. A lambda function is an instance of this class; when the class is constructed, any variables in the surrounding enviroment are passed into the constructor of the lambda function class and saved as member variables. This is, in fact, quite a bit like the idea of a functor that is already possible. The benefit of C++11 is that doing this becomes almost trivially easy--so you can use it all the time, rather than only in very rare circumstances where writing a whole new class makes sense.
- C++, being very performance sensitive, actually gives you a ton of flexibility about what variables are captured, and how--all controlled via the capture specification, []. You've already seen two cases--with nothing in it, no variables are captured, and with something, variables are captured.

Understanding Variable Capturing

C++ supports the definition of anonymous functions via *lambda expressions*. The general form of a lambda expression is

[*capture list*] (*parameter list*) -> *return type* { *statements* }

captures - a comma-separated list of zero or more captures, optionally beginning with a *capture-default*. Capture list can be passed as follows (see below for the detailed description):

- **`[a,&b]`** where *a* is captured by copy and *b* is captured by reference.
- **`[this]`** captures the current object (`*this`) by reference
- **`[&]`** captures all automatic variables used in the body of the lambda by reference and current object by reference if exists
- **`[=]`** captures all automatic variables used in the body of the lambda by copy and current object by reference if exists
- **`[]`** captures nothing

Reference: https://www.cprogramming.com/c++11/c++11-lambda-closures.html

Closure

Code Example

Listing 20.10: makeadder.cpp

```
#include <iostream>
#include <functional>

std::function<int(int)> make_adder() {
    int loc_val = 2;    // Local variable definition
    return [loc_val](int x){ return x + loc_val; };   // Returns a function
}

int main() {
    auto f = make_adder();
    std::cout << f(10) << '\n';
    std::cout << f(2) << '\n';
}
```

Function Object
std::function<>

Lambda Function

Variable Capturing
in this case: loc_val(=2)

```
12
4
```

Put it all together!!

- You understand:
  - Template & Standard Template Libraries
  - Iterators
  - Lambda Functions
  - Closures and Variable Capturing

- You can increase your productivity by using Standard Algorithm likes:
  - <algorithm> Library

    http://en.cppreference.com/w/cpp/algorithm

    http://www.cplusplus.com/reference/algorithm/
  - <numeric> Library

    http://en.cppreference.com/w/cpp/numeric

Code Example: std::for_each( )

- Applies the given function object f to the result of dereferencing every iterator in the range [first, last), in order.

Listing 20.12: `incelements.cpp`

```
#include <iostream>
#include <list>
#include <algorithm>

int main() {
    // Make a list of integers
    std::list<int> seq{5, 22, 6, -3, 8, 4};
    // Display the vector
    std::for_each(std::begin(seq), std::end(seq),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';
    // Increase each element in the vector by 1
    std::for_each(std::begin(seq), std::end(seq),
                  [](int& x) { x++; });
    // Redisplay the vector
    std::for_each(std::begin(seq), std::end(seq),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';
}
```

```
5 22 6 -3 8 4
6 23 7 -2 9 5
```

Code Example: std::copy( )

- Copies all elements in the range [first, last)

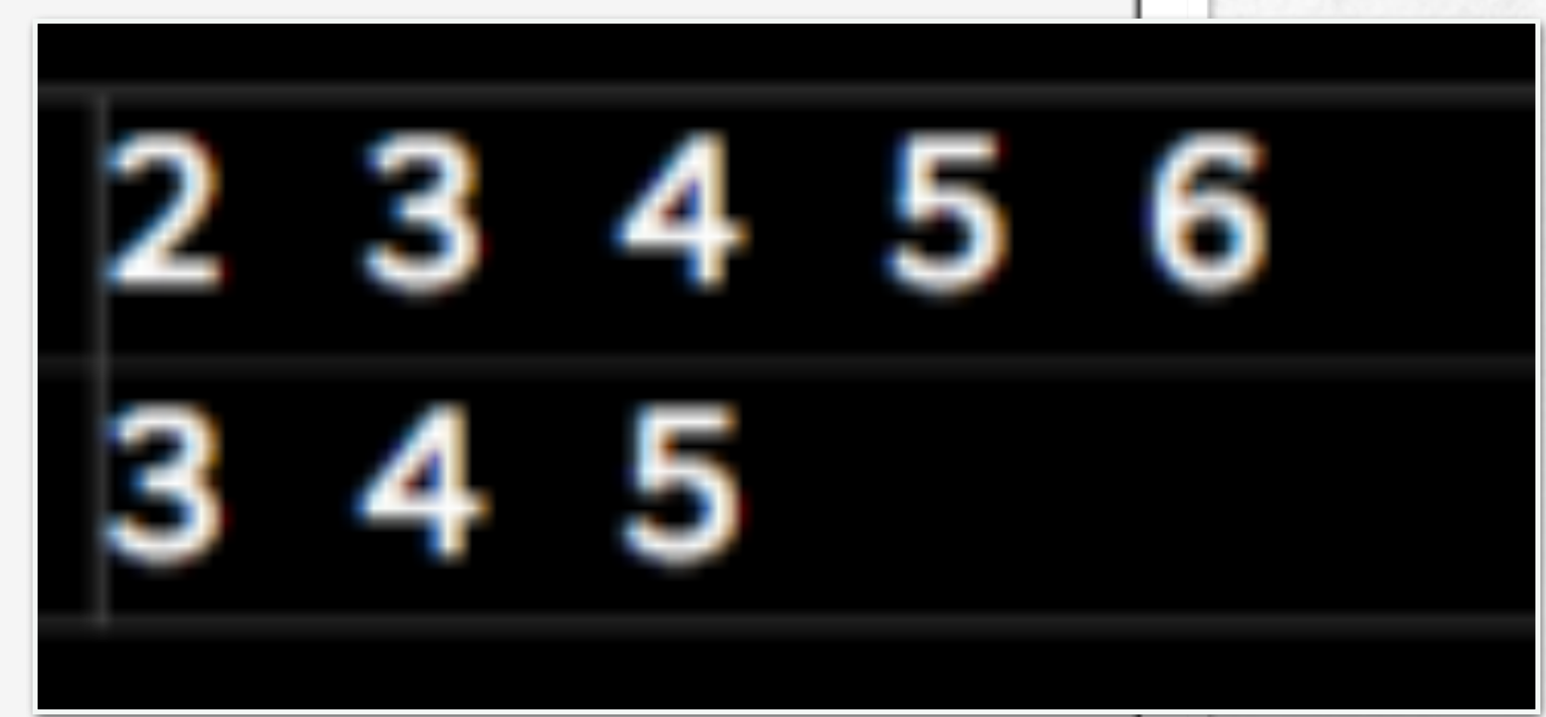
Listing 20.16: `trimvector.cpp`

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

int main() {
    // Make a vector of SIZE integers
    std::vector<int> seq { 2, 3, 4, 5, 6 };

    // Display seq
    std::for_each(std::begin(seq), std::end(seq),
                  [](int x) { std::cout << x << ' '; });
    std::cout << '\n';

    // Make a copy of vec with the first and last element trimmed off
    if (seq.size() >= 2) {
        // Make a vector large enough to hold trimmed values
        std::vector<int> seq2(seq.size() - 2);
        std::copy(std::begin(seq) + 1, std::end(seq) - 1,
                  std::begin(seq2));
        // Display seq2
        std::for_each(std::begin(seq2), std::end(seq2),
                      [](int x) { std::cout << x << ' '; });
        std::cout << '\n';
    }
}
```



Reference: http://en.cppreference.com/w/cpp/algorithm/copy

Code Example: std::transform( )

- Unary operation unary_op is applied to the range defined by [first1, last1)

Listing 20.17: uppercasestring.cpp

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

int main() {
    std::string name = "Fred",
                str = "abcDEF-GHIjkl345qw";

    std::cout << "Before: " << name << "    " << str << '\n';
    // Uppercase the strings
    std::transform(std::begin(name), std::end(name),
                   std::begin(name), std::toupper);
    std::transform(std::begin(str), std::end(str),
                   std::begin(str), std::toupper);
    std::cout << "After : " << name << "    " << str << '\n';
}
```

```
Before: Fred    abcDEF-GHIjkl345qw
After : FRED    ABCDEF-GHIJKL345QW
```


Reference: http://en.cppreference.com/w/cpp/algorithm/transform

Want know more about Namespace?

- Please read 20.6
- It's name conflicting problem, and its solution is grouping …
    - *group-name::function(or somthing)-name*
- Thus, programming with full-name may be better to avoid conflicting
    - Recommend `std::cout` instead of `using namespace std;`

Object Oriented Programming by C++

Sungwon Lee / Professor

Email: drsungwon@khu.ac.kr

Web: http://mobilelab.khu.ac.kr/