# Student's Manual for Programming Methodology with C++ $\,$

Paul Kim

## Contents

I	Algorithms	7		
1	Time Complexity Analysis	8		
2	Finding the Maximum Subarray Sum	9		
	2.1 Cubic Brute Force Algorithm	9		
	2.2 Quadratic Brute Force Algorithm	10		
	2.3 Divide and Conquer	10		
	2.4 Kadane's Algorithm: A Linear, Incremental Solution	11		
3	Various Ways of Sorting	12		
	3.1 Quick Sort	12		
	3.2 Merge Sort	13		
	3.3 Insertion Sort	14		
	3.4 Stooge Sort	14		
	3.5 Heap Sort	15		
4	Minimum Search on Rotated Array			
	4.1 Solution	16		
5	Finding the $k$ -th Smallest Number	17		
	5.1 Sort-and-Find	17		
	5.2 Quick Select	17		
6	Finding the Closest Pair of Dots	19		
	6.1 Naive Solution	19		
	6.2 Application of the Merge Sort Algorithm	19		
7	Rod Cutting Algorithm	20		
	7.1 Brute Force	20		
	$7.2 <++> \dots$	20		
	7.3 <++>	20		

	Part		Cha	pter	0
--	------	--	-----	------	---

8	Asse	embly-Line Scheduling Algorithm	21
	8.1	Brute Force	21
	8.2	The Fastest Way	21
9	Mat	rix-Chain Multiplication Algorithm	23
	9.1	<++>	23
	9.2	<++>	23
	9.3	<++>	23
II	C-	++	24
1	Vari	ables and Class	25
	1.1	The Basics	25
		1.1.1 Commenting	25
		1.1.2 Types and Variables	25
	1.2	<++>	25
2	Fund	ctions	26
	2.1	Function Definition	26
	2.2	Procedural Abstraction	26
	2.3	Argument Passing Mechanism	26
	2.4	Inline Functions	26
	2.5	Recursive Functions	26
3	iost	tream Headers	27
	3.1	Keyboard Input and Screen Output	27
	3.2	File In/Output	27
	3.3	Header Files	27
4	Arra	ays and Pointers	28
	4.1	Array	28
	4.2	Pointer	28
	4.3	<++>	28
	4.4	<++>	28
5	Obje	ect-Orientated Programming	29
	5.1	<++>	29
	5.2	<++>	29
	5.3	<++>	29
	F 4		20

6	Defining Classes with OOP	30
	6.1 <++>	30
	$6.2  <++> \dots $	30
	6.3 <++>	30
	6.4 <++>	30
7	Member Functions	31
	7.1 <++>	31
	$7.2  <++> \dots $	31
	7.3 <++>	31
	7.4 <++>	31
8	Namespace and STL	32
	8.1 <++>	32
	8.2 <++>	32
	8.3 <++>	32
	8.4 <++>	32
9	Constructors and Destructors	33
	9.1 <++>	33
	9.2 <++>	33
	9.3 <++>	33
	9.4 <++>	33
10	Public or Private, Friend Declarations	34
	10.1 <++>	34
	$10.2 <++> \dots $	34
	$10.3 <++> \dots $	34
	10.4 <++>	34
11	Copy Constructors	35
	11.1 <++>	35
	11.2 <++>	35
	11.3 <++>	35
	11.4 <++>	35
12	Operator Overloading and the Rule of Three	36
	12.1 <++>	36
	$12.2 <++> \dots $	36
	12.3 <++>	36
	19.4 < + + >	36

13	Protected and Private Derivations	37
	13.1 <++>	37
	13.2 <++>	37
	13.3 <++>	37
	13.4 <++>	37
14	Virtual Functions	38
	14.1 <++>	38
	14.2 <++>	38
	14.3 <++>	38
	14.4 <++>	38
15	Pure Virtual Functions	39
	15.1 <++>	39
	15.2 <++>	39
	15.3 <++>	39
	15.4 <++>	39
16	Reusing Copy Control Members	40
		40
	16.2 <++>	40
		40
	16.4 <++>	40
17	<++>	41
	17.1 <++>	41
	17.2 <++>	41
	17.3 <++>	41
18	<++>	42
	18.1 <++>	42
	18.2 <++>	42
	18.3 <++>	42
19	<++>	43
	19.1 <++>	43
	19.2 <++>	43
	19.3 <++>	43
20	<++>	44
	20.1 <++>	44
	20.2 <++>	44
	20.3 <++>	11

6 Part - Chapter 0

21	<++>	45
	21.1 <++>	45
	21.2 <++>	45
	21.3 <++>	45
22	<++>	46
	22.1 <++>	46
	22.2 <++>	46
	22.3 <++>	46
23	<++>	47
	23.1 <++>	47
	23.2 <++>	47
	23.3 <++>	47
24	<++>	48
	24.1 <++>	48
	24.2 <++>	48
	24.3 <++>	48

## Part I

## Algorithms

### Time Complexity Analysis

- 1. The running time of an algorithm is relevent to the amount of input. Therefore the running time is a function of the amount of input: T(n)
- 2. Definitions of Time Complexity: with a positive constant c,
  - 1) Big-O:  $T(n) \ge c \times f_O(n) \Rightarrow T(n) = O(f_O(n))$  Best-case scenarios can be described via Big-O functions.
  - 2) Big-Omega:  $T(n) \leq c \times f_{\Omega}(n) \Rightarrow T(n) = \Omega\left(f_{\Omega}(n)\right)$  Worst-case scenarios can be described via Big-Omega functions.
  - 3) Big-Theta:  $T(n) \ge c \times f(n)$  and  $T(n) \le c' \times f(n) \Leftrightarrow T(n) = O(f(n)) = \Omega(f(n)) \Rightarrow T(n) = \Theta(f(n))$ Best- and Worst-case scenarios are the same in Big-Theta functions.
  - 4) Small-O:  $T(n) = O(f_o(n)) \neq \Theta(f_o(n)) \Rightarrow T(n) = o(f_o(n))$

Constants are ignored, and only the highest degree of the polynomial's monomials are relevant to Time Complexity Analysis.

- 3. Running Time Calculations
  - 1) Summations for Loops: One loop sequence of running time f(i) is equivalent to:

$$T(n) = \sum_{i=1}^{n} f(i)$$

Two loop sequences of running time f(i, j) is equivalent to:

$$T(n) = \sum_{j=1}^{n} \sum_{i=1}^{n} f(i, j)$$

- 2) Selective Controls: Worst-case scenario,  $T(n) = \max(T_1(n), T_2(n), \cdots)$ . Best-case = minimum.
- 3) Recursion: T(n) = f(T(n')) (점화식)

## Finding the Maximum Subarray Sum

The objective of this challenge is to find the maximum value of the sum of elements in a subarray of a given array. If all integers are negative, said maximum value is the sum of a subarray equivalent to the 'empty set', which is zero.

#### 2.1 Cubic Brute Force Algorithm

```
int max_sum1(int* arr, int arrsize) {
    int maxSum = 0;
    for (int i = 0; i < arrsize; i++) {
        for (int j = i; j < arrsize; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) thisSum += arr[k];
            if (maxSum < thisSum) maxSum = thisSum;
        }
    } return maxSum;
}</pre>
```

This algorithm utilises three loops and a function of constant time in the innermost loop. Therefore, the time complexity analysis goes:

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 = O(n^3)$$

Obviously this method is quite wasteful in both memory and timekeeping. The following two algorithms are substantial progessions from this algorithm:

### 2.2 Quadratic Brute Force Algorithm

```
int max_sum2(int* arr, int arrsize) {
    int maxSum = 0;
    for (int i = 0; i < arrsize; i++) {
        int iSum = 0;
        for (int j = i; j < arrsize; j++) {
            iSum += arr[j];
            if (maxSum < iSum) maxSum = iSum;
        }
    } return maxSum;
}</pre>
```

This algorithm utilises two loops and a function of constant time in the innermost loop. Therefore, the time complexity analysis goes:

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} 1 = O(n^2)$$

#### 2.3 Divide and Conquer

```
int max_sum3(int* arr, int left, int right) {
30
        if (left >= right) return arr[left];
31
        else { int hereMax = 0;
            int leftSum = max_sum3(arr, left, ((left + right) / 2) - 1);
33
            int rightSum = max_sum3(arr, ((left + right) / 2) + 1, right);
34
            if (leftSum >= rightSum) hereMax = leftSum;
35
            else hereMax = rightSum;
            int leftMax = arr[(left + right) / 2], leftTemp = 0;
            int rightMax = arr[(left + right) / 2], rightTemp = 0;
            for (int i = (left + right) / 2; i >= left; i--) {
39
                 leftTemp += arr[i];
40
                 if (leftMax < leftTemp) leftMax = leftTemp;</pre>
            } for (int i = (left + right) / 2; i <= right; i++) {</pre>
42
                 rightTemp += arr[i];
                 if (rightMax < rightTemp) rightMax = rightTemp;</pre>
            } int midSum = leftMax + rightMax - arr[(left + right) / 2];
45
            if (hereMax < midSum) hereMax = midSum;</pre>
            return hereMax;
47
        }
    }
```

This algorithm utilises a divisive recursion and selection controls of constant time within each recursion. Therefore, the time complexity analysis goes:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^kT\left(\frac{n}{2^k}\right) + nk, \quad \therefore T(n) = 2^{\log_2 n}T(1) + n\log_2 n = n\log_2 n + \varepsilon n = O(n\log n)$$

While these two algorithms are quite effective compared to Cubic Brute Force, this last algorithm manages to provide the fastest solution possible:

#### 2.4 Kadane's Algorithm: A Linear, Incremental Solution

```
int max_sum4(int* arr, int arrsize) {
    int maxSum = arr[0], thisSum = 0;
    for (int i = 0; i < arrsize; i++) {
        thisSum += arr[i];
        if (thisSum > maxSum) maxSum = thisSum;
        else if (thisSum < 0) thisSum = 0;
    }
} return maxSum;
}</pre>
```

This algorithm utilises one loop and a function of constant time in the loop. Therefore, the time complexity analysis goes:

$$T(n) = \sum_{i=1}^{n} 1 = O(n)$$

Since, essentially, the challenge requires a system to read each data at least once, the O(n) function above is obviously the best solution.

## Various Ways of Sorting

The objective of this challenge is sorting given algebraic (in this case, double) elements of an aray in incrasing order. Some elements may be of equal value.

#### 3.1 Quick Sort

```
void quicksort(double *arr, int begin, int end) {
         double pivot = arr[begin];
         int i = begin, j = end;
         while (i <= j) {
             while (arr[i] < pivot) i++;</pre>
             while (arr[j] > pivot) j--;
             if (i <= j) {
10
                  swap(arr[i], arr[j]);
                  i++; j--;
13
         } if (begin < j) quicksort(arr, begin, j);</pre>
14
         if (end > i) quicksort(arr, i, end);
15
    }
16
     int main(int argc, char *argv[]) {
87
         double *input_array = new double[2500];
88
         for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);</pre>
89
         double *quick_arr = new double[2500];
         for (int i = 0; i < 2500; i++) quick_arr[i] = input_array[i];</pre>
         quicksort(quick_arr, 0, 2499);
93
         if (check(quick_arr)) cout << "Quicksort Validated" << endl;</pre>
94
120
         return 0;
121
    }
```

Quick Sort is a recursive algorithm that does the following:

- 1. Choose a "pivot" element: Line 6
- 2. Swap elements that are larger than the pivot with elements that are smaller but on the righthand side of the chosen element: Lines 8:14
- 3. Continue until all elements are sorted, then recursively proceed with the left and right subarrays of the pivot: Lines 15, 16

Using two half-recursions results in the following time complexity analysis:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^k T\left(\frac{n}{2^k}\right) + kn = O(n\log n)$$

#### 3.2 Merge Sort

```
void mergesort(double * arr, int begin, int end) {
19
         if (begin < end) {
             int centre = begin + (end - begin) / 2;
             mergesort(arr, begin, centre);
             mergesort(arr, centre + 1, end);
22
             int n1 = centre - begin + 1;
23
             int n2 = end - centre;
             double* L = new double[n1 + 1];
             double* R = new double[n2 + 1];
26
             for (int i = 0; i <= n1 - 1; i++) L[i] = arr[begin + i];
27
             for (int j = 0; j \le n2 - 1; j++) R[j] = arr[centre + j + 1];
28
             L[n1] = (double)INT_MAX;
             R[n2] = (double)INT_MAX;
             int a = 0, b = 0;
             for (int k = begin; k <= end; k++) {</pre>
32
                 if (L[a] <= R[b]) arr[k] = L[a++];</pre>
33
                 else arr[k] = R[b++];
34
             }
35
         }
    }
37
     int main(int argc, char *argv[]) {
87
         double *input_array = new double[2500];
88
         for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);</pre>
         double *merge_arr = new double[2500];
         for (int i = 0; i < 2500; i++) merge_arr[i] = input_array[i];</pre>
97
         mergesort(merge_arr, 0, 2499);
         if (check(merge_arr)) cout << "Mergesort Validated" << endl;</pre>
         return 0;
    }
121
```

Merge Sort is also a recursive algorithm dependent on two half-recursions. Time complexity analysis goes:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^k T\left(\frac{n}{2^k}\right) + kn = O(n\log n)$$

#### 3.3 Insertion Sort

```
void insertionsort(double* arr, int size) {
         for (int i = 1; i <= size - 1; i++) {
             double key = arr[i];
             int j = i - 1;
49
             while (j \ge 0 \&\& arr[j] > key) {
                 arr[j + 1] = arr[j];
51
                 j--;
             } arr[j + 1] = key;
         }
54
    }
55
     int main(int argc, char *argv[]) {
87
         double *input_array = new double[2500];
88
         for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);
         double *insertion_arr = new double[2500];
101
         for (int i = 0; i < 2500; i++) insertion_arr[i] = input_array[i];</pre>
102
         insertionsort(insertion_arr, 2500);
104
         if (check(insertion_arr)) cout << "Insertionsort Validated" << endl;</pre>
120
         return 0;
    }
```

Insertion Sort is a double-loop algorithm, therefore time complexity analysis is as follows:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \epsilon = O(n^2)$$

#### 3.4 Stooge Sort

```
void stoogesort(double* arr, int begin, int end) {
        if (begin >= end) return;
        else if (end - begin == 1) {
59
            if (arr[begin] > arr[end]) swap(arr[begin], arr[end]);
60
        } else {
61
            int d = (end - begin + 1) / 3;
            stoogesort(arr, begin, end - d);
            stoogesort(arr, begin + d, end);
64
            stoogesort(arr, begin, end - d);
65
        }
66
    }
    int main(int argc, char *argv[]) {
        double *input_array = new double[2500];
        for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);</pre>
```

```
double *stooge_arr = new double[2500];
for (int i = 0; i < 2500; i++) stooge_arr[i] = input_array[i];
stoogesort(stooge_arr, 0, 2499);
if (check(stooge_arr)) cout << "Stoogesort Validated" << endl;
return 0;
}</pre>
```

Stooge Sort is a recursive algorithm of three subarrays. The time complexity is:

$$T(n) = 3T\left(\frac{3}{2}n\right) + 1, \quad \therefore T(n) \approx O\left(n^{2.7}\right)$$

#### 3.5 Heap Sort

```
void heapsort(double *arr, int n, int i) {
         int largest = i;
70
         int 1 = 2 * i + 1;
71
         int r = 2 * i + 2;
72
         if (1 < n && arr[1] > arr[largest]) largest = 1;
         if (r < n && arr[r] > arr[largest]) largest = r;
         if (largest != i) {
             swap(arr[i], arr[largest]);
             heapsort(arr, n, largest);
77
         }
    }
     int main(int argc, char *argv[]) {
         double *input_array = new double[2500];
         for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);
89
         double *heap_arr = new double[2500];
111
         for (int i = 0; i < 2500; i++) heap_arr[i] = input_array[i];</pre>
112
         for (int i = 1249; i >= 0; i--)
113
             heapsort(heap_arr, 2500, i);
114
         for (int i = 2499; i >= 0; i--) {
             swap(heap_arr[0], heap_arr[i]);
116
             heapsort(heap_arr, i, 0);
117
         } if (check(heap_arr)) cout << "Heapsort Validated" << endl;</pre>
118
         return 0;
120
    }
121
```

Heap sort is a loop-and-recursive algorithm of time complexy analysis

$$T(n) = O(n \log n)$$

## Minimum Search on Rotated Array

The objective of this challenge is to find the smallest element in an array that was rotated at a random index after being sorted. Duplicate elements are not allowed for this challenge.

#### 4.1 Solution

```
int Searcher(int* a, int 1, int r) {
        int m = 1 + (r - 1) / 2; // Prevents overflow
        if (l == r) return a[l];
        else if (a[l] < a[r]) return a[l];</pre>
        else if (a[m] < a[r]) {
            if (a[m - 1] <= a[m]) return Searcher(a, 1, m - 1);</pre>
             else return a[m];
        } else return Searcher(a, m + 1, r);
10
    }
11
    int main(int argc, char *argv[]) {
34
        int n; std::cin >> n;
35
        int* input_array = new int[n];
        for (int i = 0; i < n; ++i) input_array[i] = int(rand() % n);</pre>
37
        sort(input_array, 0, n - 1); // See Previous Chapter
38
        std::cout << Searcher(input_array, 0, n - 1);</pre>
39
        return 0;
42
    }
43
```

By using a binary-search method with three nodes, the code above is successful in finding and returning the smallest element. This algorithm uses recursive functions, and shows the time complexity of the following:

$$T(n) = 2T\left(\frac{n}{2}\right) + \alpha = 4T\left(\frac{n}{4}\right) + 3\alpha = \dots = 2^k T\left(\frac{n}{2^k}\right) + \frac{k(k+1)}{2}\alpha, \quad \therefore T(n) = O(\log n)$$

### Finding the k-th Smallest Number

The objective of this challenge is to find the k-th smallest number in an unsorted array. Some elements may be of equal value.

#### 5.1 Sort-and-Find

This solution sorts the unsorted array and returning the value at index k. As seen before, time complexity analysis indicates that this approach requires more than  $O(n \log n)$  amount of time. Since the sorting algorithm has already been presented before, it will not be features again.

#### 5.2 Quick Select

This solution is modeled after the Quicksort algorithm. Unlike Quicksort, this algorithm only focuses on one side of the partition, and therefore its expected running time is  $T(n) = \Theta(n)$ .

```
int quickselect(int* a, int k, int l, int r) {
        if (1 == r) return a[1];
        int x = a[r], i = 1;
       for (int j = 1; j < r; ++j) {
            if (a[j] \le a[r]) \{
                std::swap(a[i], a[j]);
10
        } std::swap(a[i], a[r]);
11
        if (k == i - l + 1) return a[i];
        else if (k < i - l + 1) return quickselect(a, k, l, i - 1);
        else return quickselect(a, k - i + l - 1, i + 1, r);
14
15
    int main(int argc, char *argv[]) {
        int n; std::cin >> n;
        int* input_array = new int[n];
```

```
for (int i = 0; i < n; ++i) input_array[i] = int(rand() % n);
int k; std::cin >> k;
std::cout << quickselect(input_array, k, 0, n - 1);
return 0;
4 }</pre>
```

## Finding the Closest Pair of Dots

The objective of this challenge is to find the distance between the two closest dots on a two-dimensional plane. No two dots occupy the same coordinate.

- 6.1 Naive Solution
- 6.2 Application of the Merge Sort Algorithm

## Rod Cutting Algorithm

The objective of this challenge is to determine how to cut an imaginary rod to receive the maximum revenue. Revenue per rod length and initial rod lengths are given. Note that, if the price for the initial rod length is high enough, the optimal solution may require no cutting at all.

#### 7.1 Brute Force

Since there are n-1 places to either cut or leave, there are  $2^{n-1}$  ways in total to cut a rod of n inches. Therefore a brute force algorithm such as the one presented solves all possible outcomes and finds the maximum value. Time Complexity Analysis shows an exponential time function.

- 7.2 <++>
- 7.3 <++>

## Assembly-Line Scheduling Algorithm

The objective of this challenge is to move from entry to exit point via stations along two assembly lines. Each station has a fixed time of processing, and from a station one can move to the next station along the line OR the next station on the other assembly line. Elements can be of equal value.

#### 8.1 Brute Force

With n stations on each assembly line, there are  $2^n$  possible ways to choose stations. Therefore the brute force algorithm takes  $\Omega(2^n)$  of time complexity to be completely processed, making it infeasible.

### 8.2 The Fastest Way

```
#include <iostream>
    #include <ctime>
    int main(int argc, char *argv[]) {
        int n; std::cin >> n;
        srand((unsigned int)time(NULL));
        int* a1 = new int[n]; int* a2 = new int[n];
        int* t1 = new int[n]; int* t2 = new int[n];
        for (int i = 0; i < n; i++) {
            a1[i] = rand() % n; a2[i] = rand() % n;
10
            t1[i] = rand() % n; t2[i] = rand() % n;
11
        } int e1 = rand() % n, e2 = rand() % n;
12
        int x1 = rand() % n, x2 = rand() % n;
        int way1 = e1 + a1[0];
        int way2 = e2 + a2[0];
15
        for (int i = 1; i < n; i++) {
16
            int temp = way1;
17
            way1 = a1[i] + (way1 < way2 + t2[i - 1]) ? way1 : way2 + t2[i - 1];
            way2 = a2[i] + (way2 < temp + t1[i - 1]) ? way2 : temp + t1[i - 1];
```

```
20      } std::cout << (way1 + x1 < way2 + x2) ? way1 + x1 : way2 + x2;
21      return 0;
22  }</pre>
```

With a simple loop algorithm and a minimum-evaluating process every iteration, this process shows a time complexity of:

$$T(n) = \Theta(n)$$

## Matrix-Chain Multiplication Algorithm

- 9.1 <++>
- 9.2 <++>
- 9.3 <++>

Part II

### Variables and Class

#### 1.1 The Basics

```
#include <iostream>

int main(void) {

std::cout << "Hello World!";

return 0;
}</pre>
```

C++ is an extension of C.

#### 1.1.1 Commenting

Single-line comments use two forward slashes: // Comment 1 Multi-line comments use a forward slash and a star at each end to denote beginning and end: /\* Comment 2 \*/

#### 1.1.2 Types and Variables

- 1. Primitive, built-in types:
  - 1) void is used to determine functions and variables of no return value.
  - 2) bool, char, int, float, double and et cetra are used to return certain values.
  - 3) unsigned is used to prepend targets that are always of positive value.
- 2. Enumerations: enum is used to define groups of integer constants.

#### 1.2 <++>

## Functions

- 2.1 Function Definition
- 2.2 Procedural Abstraction
- 2.3 Argument Passing Mechanism
- 2.4 Inline Functions
- 2.5 Recursive Functions

### iostream Headers

- 3.1 Keyboard Input and Screen Output
- 3.2 File In/Output
- 3.3 Header Files
- 1. Header File: A file that allows the reusage of certain portions of source code. Header files are included to a source code via #include, which inserts the header file code at that specific location.
- 2. Headers are used for declaring functions, classes, et cetra, and since a header file can be used multiple times by multiple source codes, defining variables and classes must not happen in a header file.
- 3. Include Guards: Since a single header file can be included multiple times throughout a compiling process and cause compilation errors, prevention methods are supported by the compilers. This "guard from inclusion"s are called Include guards:

```
#ifndef _IOSHEADER_H_ // Check if header is yet undefined
#define _IOSHEADER_H_ // If checked, define header

#include <iostream>

class classy {
public:
    void std::cout << "Hello World!" << std::endl;
};

#endif // End selection control</pre>
```

## Arrays and Pointers

### 4.1 Array

1. Sequential container of objects of a single data type with fixed size: int array[3] = {5, 3, 2};

#### 4.2 Pointer

- 1. A pointer is a variable that holds the address of an object, enabling indirect access: int \*m = new int[4];
- 2. Adding an integer n to a pointer variable returns the address of an element displaced by n from the original element.
- 4.3 <++>
- 4.4 <++>

## Object-Orientated Programming

- 5.1 <++>
- 5.2 <++>
- 5.3 <++>
- 5.4 <++>

## Defining Classes with OOP

- 6.1 <++>
- 6.2 <++>
- 6.3 <++>
- 6.4 <++>

## Member Functions

- 7.1 <++>
- 7.2 <++>
- 7.3 <++>
- 7.4 <++>

## Namespace and STL

- 8.1 <++>
- 8.2 <++>
- 8.3 <++>
- 8.4 <++>

### Constructors and Destructors

- 9.1 <++>
- 9.2 <++>
- 9.3 <++>
- 9.4 <++>

## Public or Private, Friend Declarations

- 10.1 <++>
- 10.2 <++>
- 10.3 <++>
- 10.4 <++>

## Copy Constructors

- 11.1 <++>
- 11.2 <++>
- 11.3 <++>
- 11.4 <++>

Operator Overloading and the Rule of Three

- 12.1 <++>
- 12.2 <++>
- 12.3 <++>
- 12.4 <++>

#### Protected and Private Derivations

- 13.1 <++>
- 13.2 <++>
- 13.3 <++>
- 13.4 <++>

# Virtual Functions

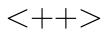
- 14.1 <++>
- 14.2 <++>
- 14.3 <++>
- 14.4 <++>

#### Pure Virtual Functions

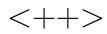
- 15.1 <++>
- 15.2 <++>
- 15.3 <++>
- 15.4 <++>

# Reusing Copy Control Members

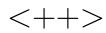
- 16.1 <++>
- 16.2 <++>
- 16.3 <++>
- 16.4 <++>



- 17.1 <++>
- 17.2 <++>
- 17.3 <++>

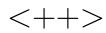


- 18.1 <++>
- 18.2 <++>
- 18.3 <++>

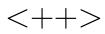


- 19.1 <++>
- 19.2 <++>
- 19.3 <++>

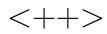
- 20.1 <++>
- 20.2 <++>
- 20.3 <++>



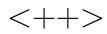
- 21.1 <++>
- 21.2 <++>
- 21.3 <++>



- 22.1 <++>
- 22.2 <++>
- 22.3 <++>



- 23.1 <++>
- 23.2 <++>
- 23.3 <++>



- 24.1 <++>
- 24.2 <++>
- 24.3 <++>