Programming Methodology

with C++

Paul Kim

powered by LaTeX 2ε

Contents

Part I

Algorithms

Chapter 1

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: $T(n) \geq c \times f_O(n) \Rightarrow T(n) = O(f_O(n))$ Best-case scenarios can be described via Big-O functions.
   2) Big-Omega: $T(n) \leq c \times f_\Omega(n) \Rightarrow T(n) = \Omega(f_\Omega(n))$ Worst-case scenarios can be described via Big-Omega functions.
   3) Big-Theta: $T(n) \geq c \times f(n)$ and $T(n) \leq c' \times f(n) \Leftrightarrow T(n) = O(f(n)) = \Omega(f(n)) \Rightarrow T(n) = \Theta(f(n))$ 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)

Chapter 2

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

```
6  int max_sum1(int* arr, int arrsize) {
7      int maxSum = 0;
8      for (int i = 0; i < arrsize; i++) {
9          for (int j = i; j< arrsize; j++) {
10             int thisSum = 0;
11             for (int k = i; k <= j; k++) thisSum += arr[k];
12             if (maxSum < thisSum) maxSum = thisSum;
13         }
14     } return maxSum;
15 }
```

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

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

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

```
18 int max_sum2(int* arr, int arrsize) {
19     int maxSum = 0;
20     for (int i = 0; i < arrsize; i++) {
21         int iSum = 0;
22         for (int j = i; j < arrsize; j++) {
23             iSum += arr[j];
24             if (maxSum < iSum) maxSum = iSum;
25         }
26     } return maxSum;
27 }
```

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

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

2.3 Divide and Conquer

```
30 int max_sum3(int* arr, int left, int right) {
31     if (left >= right) return arr[left];
32     else { int hereMax = 0;
33         int leftSum = max_sum3(arr, left, ((left + right) / 2) - 1);
34         int rightSum = max_sum3(arr, ((left + right) / 2) + 1, right);
35         if (leftSum >= rightSum) hereMax = leftSum;
36         else hereMax = rightSum;
37         int leftMax = arr[(left + right) / 2], leftTemp = 0;
38         int rightMax = arr[(left + right) / 2], rightTemp = 0;
39         for (int i = (left + right) / 2; i >= left; i--) {
40             leftTemp += arr[i];
41             if (leftMax < leftTemp) leftMax = leftTemp;
42         } for (int i = (left + right) / 2; i <= right; i++) {
43             rightTemp += arr[i];
44             if (rightMax < rightTemp) rightMax = rightTemp;
45         } int midSum = leftMax + rightMax - arr[(left + right) / 2];
46         if (hereMax < midSum) hereMax = midSum;
47         return hereMax;
48     }
49 }
```

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

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

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

```
52 int max_sum4(int* arr, int arrsize) {
53     int maxSum = arr[0], thisSum = 0;
54     for (int i = 0; i < arrsize; i++) {
55         thisSum += arr[i];
56         if (thisSum > maxSum) maxSum = thisSum;
57         else if (thisSum < 0) thisSum = 0;
58     }  return maxSum;
59 }
```

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.

Chapter 3

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

```
4   void quicksort(double *arr, int begin, int end) {
5       double pivot = arr[begin];
6       int i = begin, j = end;
7       while (i <= j) {
8           while (arr[i] < pivot) i++;
9           while (arr[j] > pivot) j--;
10          if (i <= j) {
11              swap(arr[i], arr[j]);
12              i++; j--;
13          }
14      } if (begin < j) quicksort(arr, begin, j);
15      if (end > i) quicksort(arr, i, end);
16  }

87  int main(int argc, char *argv[]) {
88      double *input_array = new double[2500];
89      for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

91      double *quick_arr = new double[2500];
92      for (int i = 0; i < 2500; i++) quick_arr[i] = input_array[i];
93      quicksort(quick_arr, 0, 2499);
94      if (check(quick_arr)) cout << "Quicksort Validated" << endl;

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

```
18  void mergesort(double * arr, int begin, int end) {
19      if (begin < end) {
20          int centre = begin + (end - begin) / 2;
21          mergesort(arr, begin, centre);
22          mergesort(arr, centre + 1, end);
23          int n1 = centre - begin + 1;
24          int n2 = end - centre;
25          double* L = new double[n1 + 1];
26          double* R = new double[n2 + 1];
27          for (int i = 0; i <= n1 - 1; i++) L[i] = arr[begin + i];
28          for (int j = 0; j <= n2 - 1; j++) R[j] = arr[centre + j + 1];
29          L[n1] = (double)INT_MAX;
30          R[n2] = (double)INT_MAX;
31          int a = 0, b = 0;
32          for (int k = begin; k <= end; k++) {
33              if (L[a] <= R[b]) arr[k] = L[a++];
34              else arr[k] = R[b++];
35          }
36      }
37  }

87  int main(int argc, char *argv[]) {
88      double *input_array = new double[2500];
89      for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

96      double *merge_arr = new double[2500];
97      for (int i = 0; i < 2500; i++) merge_arr[i] = input_array[i];
98      mergesort(merge_arr, 0, 2499);
99      if (check(merge_arr)) cout << "Mergesort Validated" << endl;

120     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

```
46  void insertionsort(double* arr, int size) {
47      for (int i = 1; i <= size - 1; i++) {
48          double key = arr[i];
49          int j = i - 1;
50          while (j >= 0 && arr[j] > key) {
51              arr[j + 1] = arr[j];
52              j--;
53          } arr[j + 1] = key;
54      }
55  }

87  int main(int argc, char *argv[]) {
88      double *input_array = new double[2500];
89      for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

101     double *insertion_arr = new double[2500];
102     for (int i = 0; i < 2500; i++) insertion_arr[i] = input_array[i];
103     insertionsort(insertion_arr, 2500);
104     if (check(insertion_arr)) cout << "Insertionsort Validated" << endl;

120     return 0;
121 }
```

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

```
57  void stoogesort(double* arr, int begin, int end) {
58      if (begin >= end) return;
59      else if (end - begin == 1) {
60          if (arr[begin] > arr[end]) swap(arr[begin], arr[end]);
61      } else {
62          int d = (end - begin + 1) / 3;
63          stoogesort(arr, begin, end - d);
64          stoogesort(arr, begin + d, end);
65          stoogesort(arr, begin, end - d);
66      }
67  }

87  int main(int argc, char *argv[]) {
88      double *input_array = new double[2500];
89      for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);
```

```
106     double *stooge_arr = new double[2500];
107     for (int i = 0; i < 2500; i++) stooge_arr[i] = input_array[i];
108     stoogesort(stooge_arr, 0, 2499);
109     if (check(stooge_arr)) cout << "Stoogesort Validated" << endl;

120     return 0;
121 }
```

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

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

3.5 Heap Sort

```
69  void heapsort(double *arr, int n, int i) {
70      int largest = i;
71      int l = 2 * i + 1;
72      int r = 2 * i + 2;
73      if (l < n && arr[l] > arr[largest]) largest = l;
74      if (r < n && arr[r] > arr[largest]) largest = r;
75      if (largest != i) {
76          swap(arr[i], arr[largest]);
77          heapsort(arr, n, largest);
78      }
79  }

87  int main(int argc, char *argv[]) {
88      double *input_array = new double[2500];
89      for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

111     double *heap_arr = new double[2500];
112     for (int i = 0; i < 2500; i++) heap_arr[i] = input_array[i];
113     for (int i = 1249; i >= 0; i--)
114         heapsort(heap_arr, 2500, i);
115     for (int i = 2499; i >= 0; i--) {
116         swap(heap_arr[0], heap_arr[i]);
117         heapsort(heap_arr, i, 0);
118     } if (check(heap_arr)) cout << "Heapsort Validated" << endl;

120     return 0;
121 }
```

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

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

Chapter 4

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

```
3  int Searcher(int* a, int l, int r) {
4      int m = l + (r - l) / 2; // Prevents overflow
5      if (l == r) return a[l];
6      else if (a[l] < a[r]) return a[l];
7      else if (a[m] < a[r]) {
8          if (a[m - 1] <= a[m]) return Searcher(a, l, m - 1);
9          else return a[m];
10     } else return Searcher(a, m + 1, r);
11 }

34 int main(int argc, char *argv[]) {
35     int n; std::cin >> n;
36     int* input_array = new int[n];
37     for (int i = 0; i < n; ++i) input_array[i] = int(rand() % n);
38     sort(input_array, 0, n - 1); // See Previous Chapter
39     std::cout << Searcher(input_array, 0, n - 1);

42     return 0;
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:

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

Chapter 5

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

```
3  int quickselect(int* a, int k, int l, int r) {
4      if (l == r) return a[l];
5      int x = a[r], i = l;
6      for (int j = l; j < r; ++j) {
7          if (a[j] <= a[r]) {
8              std::swap(a[i], a[j]);
9              ++i;
10         }
11     } std::swap(a[i], a[r]);
12     if (k == i - l + 1) return a[i];
13     else if (k < i - l + 1) return quickselect(a, k, l, i - 1);
14     else return quickselect(a, k - i + l - 1, i + 1, r);
15 }

17 int main(int argc, char *argv[]) {
18     int n; std::cin >> n;
19     int* input_array = new int[n];
```

```
20      for (int i = 0; i < n; ++i) input_array[i] = int(rand() % n);
21      int k; std::cin >> k;
22      std::cout << quickselect(input_array, k, 0, n - 1);
23      return 0;
24  }
```

Chapter 6

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

Chapter 7

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

```
4  int brute(int* price, int n) {
5      if (n <= 0) return 0;
6      int q = INT_MIN;
7      for (int i = 0; i < n; ++i) {
8          q = std::max(q, price[i] + brute(price, n - i - 1));
9      } return q;
10 }
```

```
35 int main(int argc, char *argv[]) {
36     int n; std::cin >> n;
37     int* pricing = new int[n + 1];
38     pricing[0] = 0;
39     for (int i = 0; i < n; ++i) pricing[i + 1] = pricing[i] + rand() % n;

47     std::cout << brute(pricing, n + 1);

48     return 0;
49 }
```

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 Dynamic Programming: Top-Down Approach

```
12 int topdown(int* price, int n, int* memo) {
13    if (memo[n] >= 0) return memo[n];
14    int q = INT_MIN;
15    if (n == 0) q = 0;
16    else {
17        for (int i = 1; i <= n; ++i) {
18            q = std::max(q, price[i] + topdown(price, n - i, memo));
19        }
20    } memo[n] = q;
21    return q;
22 }

35 int main(int argc, char *argv[]) {
36     int n; std::cin >> n;
37     int* pricing = new int[n + 1];
38     pricing[0] = 0;
39     for (int i = 0; i < n; ++i) pricing[i + 1] = pricing[i] + rand() % n;

41     std::cout << " ";
42     int* arr = new int[n + 1];
43     for (int i = 0; i <= n; ++i) {
44         arr[i] = INT_MIN;
45     } std::cout << topdown(pricing, n, arr);

48     return 0;
49 }
```

The most critical reason for the brute force algorithm's exponentaily long time conplexity is that brute force repeats sonving what is essentially the same set of divisions. For example, when processing the cutting of a 9 inch rod, brute force algorithm processes the same 2, 3, and 4-inch dsivision as 6 different methods $(2,\ 3,\ 4),\ (3,\ 2,\ 4), \cdots$. This can be solved by recording results of previous iterations of the recursion, or the **memoisation** of previous data. With the top-down method presented above, the new array `int* arr` is a storage array for the previously processed results.

7.3 Dynamic Programming: Bottom-Up Approach

```
24 int bottomup(int* price, int n) {
25     int val[n + 1];
26     val[0] = 0;
27     for (int i = 1; i <= n; ++i) {
28         int q = INT_MIN;
29         for (int j = 0; j < i; ++j) {
30             q = std::max(q, price[j] + val[i - j - 1]);
31         } val[i] = q;
```

```
32      } return val[n];
33  }

35  int main(int argc, char *argv[]) {
36      int n; std::cin >> n;
37      int* pricing = new int[n + 1];
38      pricing[0] = 0;
39      for (int i = 0; i < n; ++i) pricing[i + 1] = pricing[i] + rand() % n;

40      std::cout << bottomup(pricing, n + 1);

48      return 0;
49  }
```

Unusual circumstances can lead to failures in top-down algorithms where not all possible outcomes are tested. The bottom-up approach above often shows strength in constant factors and overall stability, and can be mathematically proven in its ability to always process all subproblems, although such proof will not be provided in this manual.
Both Top-Down and Bottom-Up approaches show a time complexity of $T(n) = \Theta\left(n^2\right)$.

Chapter 8

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 $\Omega\left(2^n\right)$ 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;
        t1[i] = rand() % n; t2[i] = rand() % n;
    } int e1 = rand() % n, e2 = rand() % n;
    int x1 = rand() % n, x2 = rand() % n;
    int way1 = e1 + a1[0];
    int way2 = e2 + a2[0];
    for (int i = 1; i < n; i++) {
        int temp = way1;
        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  }
```

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

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

Chapter 9

Matrix-Chain Multiplication Algorithm

The objective of this challenge is to calculate the minimum amount of scalar multiplications a group of matrices needs to return the complete and correct matrix product. Consider three matrices $\mathbf{A}[10][100]$, $\mathbf{B}[100][5]$, and $\mathbf{C}[5][50]$. There are two ways to multiply these matrices:

1. $(\mathbf{A}\times\mathbf{B})\times\mathbf{C}$: $\mathbf{A}\times\mathbf{B}$ requires $10\times100\times5=5000$ scalar multiplications. $\mathbf{AB}\times\mathbf{C}$ requires $10\times5\times50=2500$. Therefore the total amount of scalar multiplications is $5000+2500=7500$.

2. $\mathbf{A}\times(\mathbf{B}\times\mathbf{C})$: $\mathbf{B}\times\mathbf{C}$ requires $100\times5\times50=25000$ scalar multiplications. $\mathbf{A}\times\mathbf{BC}$ requires $10\times100\times50=50000$. Therefore the total amount of scalar multiplications is $25000+50000=75000$.

Given a chain of n matrices and an integer array of $n+1$ elements where each matrix $\mathbf{A_i}$ shows the dimesion of $(p_{i-1},\ p_i)$, the following solutions are algorithms that minimise scalar multiplication amounts.

9.1 Optimal Solution

```
4  int main() {
5      int n; std::cin >> n;
6      int* p = new int[n + 1]; int** m = new int*[n];
7      for (int i = 0; i < n; ++i) std::cin >> p[i] >> p[i + 1];
8      for (int i = 0; i < n; ++i) {
9          m[i] = new int[n]; if (i != 0) m[i][i] = 0;
10     } for (int l = 2; l <= n; ++l) {
11         for (int i = 1; i <= n + 1 - l; ++i) {
12             int j = i + l - 1;
13             m[i][j] = INT_MAX;
14             for (int k = i; k < j; ++k) {
15                 int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
16                 if (q < m[i][j]) m[i][j] = q;
17             }
18         }
19     } std::cout << m[1][n];
20     delete[] p;
```

```
21      return 0;
22  }
```

Matrix chain multiplication has $T(n) = \Theta\left(n^2\right)$ subproblems, each featuring no more than $n - 1$ choices. Therefore the time complexity of this algorithm is:

$$T(n) = \Theta\left(n^3\right)$$

Part II

C++

Chapter 1

Variables and Class

1.1 The Basics

```
#include <iostream>

int main(void) {
    std::cout << "Hello World!";
    return 0;
}
```

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 ¡++¿

Chapter 2

Functions

Chapter 3

`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 guard**s:

```
#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
```

Chapter 4

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 ¡++¿

Chapter 5

Object-Orientated Programming

5.1 ¡++¿

5.2 ¡++¿

5.3 ¡++¿

5.4 ¡++¿

Chapter 6

Defining Classes with OOP

Chapter 7

Member Functions

7.1 ¡++¿

7.2 ¡++¿

7.3 ¡++¿

7.4 ¡++¿

Chapter 8

Namespace and STL

8.1 ¡++¿

8.2 ¡++¿

8.3 ¡++¿

8.4 ¡++¿

Chapter 9

Constructors and Destructors

9.1 ¡++¿

9.2 ¡++¿

9.3 ¡++¿

9.4 ¡++¿

Chapter 10

Public or Private, Friend Declarations

10.1 ¡++¿

10.2 ¡++¿

10.3 ¡++¿

10.4 ¡++¿

Chapter 11

Copy Constructors

11.1 ¡++¿

11.2 ¡++¿

11.3 ¡++¿

11.4 ¡++¿

Chapter 12

Operator Overloading and the Rule of Three

12.1 ¡++¿

12.2 ¡++¿

12.3 ¡++¿

12.4 ¡++¿

Chapter 13

Protected and Private Derivations

13.1 ¡++¿

13.2 ¡++¿

13.3 ¡++¿

13.4 ¡++¿

Chapter 14

Virtual Functions

14.1 ¡++¿

14.2 ¡++¿

14.3 ¡++¿

14.4 ¡++¿

Chapter 15

Pure Virtual Functions

Chapter 16

Reusing Copy Control Members

16.1 ¡++¿

16.2 ¡++¿

16.3 ¡++¿

16.4 ¡++¿

Chapter 17

¡++¿

17.1 ¡++¿

17.2 ¡++¿

17.3 ¡++¿

Chapter 18

¡++¿

18.1 ¡++¿

18.2 ¡++¿

18.3 ¡++¿

Chapter 19

¡++¿

19.1 ¡++¿

19.2 ¡++¿

19.3 ¡++¿

Chapter 20

¡++¿

20.1 ¡++¿

20.2 ¡++¿

20.3 ¡++¿

Chapter 21

¡++¿

21.1 ¡++¿

21.2 ¡++¿

21.3 ¡++¿

Chapter 22

¡++¿

22.1 ¡++¿

22.2 ¡++¿

22.3 ¡++¿

Chapter 23

¡++¿

23.1 ¡++¿

23.2 ¡++¿

23.3 ¡++¿

Chapter 24

¡++¿

24.1 ¡++¿

24.2 ¡++¿

24.3 ¡++¿