Design Automation Renegades

Globetrotting Division

Boilerplate Code: Data Structures and Algorithms for Design Automation

Zhiyang Ong [1]

Report on
Common Data Structures and Algorithms
Found in Bolierplate Code for
Design Automation Software

March 8, 2016

[1]Email correspondence to: ongz@acm.org

Abstract

This report describes the design and implementation of common data structures and algorithms, as well as "computational engines" that are found in electronic design automation (EDA) software.

Data structures and algorithms for digital VLSI and cyber-physical system design include: binary decision diagrams (BDDs), AND-inverter graphs (AIGs), and their associated algorithms for optimization, traversal, and other operations (such as graph matching). Common computational engines for digital systems would include: optimization and verification engines for deterministic and nondeterministic finite state machines; decision procedures for the boolean satisfiability problem (SAT solvers) and satisfiability modulo theories (SMT solvers); quantified boolean formula (QBF) solvers; and SAT and SMT solvers for maximum satisfiability (i.e., Max-SAT and Max-SMT solvers).

Regarding EDA problems that require numerical computation (in digital, analog, or mixed-signal VLSI design), the data structures and algorithms for circuit simulation based on sparse graph would be required. In addition, techniques for model order reduction shall be implemented.

Computational engines for statistical and probabilistic analyses or stochastic modeling can include data structures and algorithms for partially observable Markov decision processes (POMDPs) and Markov chains. Tools for analyses of queueing systems (based on queueing theory) should be included.

Regarding cyber-physical systems and mixed-signal circuits, hybrid automata can be used to represent these circuits and systems.

Optimization engines for EDA include: solvers for different types of mathematical programming, such as linear programming (LP), integer linear programming (ILP), mixed-integer linear programming (MILP), quadratic programming (QP), convex programming (CP), geometric programming (GP), and second-order conic programming (SOCP); solvers for pseudo-boolean optimization (PBO solvers) and weighted-boolean optimization (WBO); and meta-heuristics (e.g., evolutionary algorithms, simulated annealing, and ant colony optimization).

Algorithms shall be implemented using parallel programming, in a scalable style. In addition, considerations shall be given to the use of constraint programming.

More stuff to be included...

Revision History

Revision History:

1. Version 0.1, December 23, 2014. Initial copy of the report.
2. Version 0.1.1, September 16, 2015. Added sections for mathematics and statistics, and the abstract.

Contents

Chapter 1

Algorithms

This section documents algorithms that I have implemented for my C++ -based boilerplate code repository.

A template for typesetting algorithms is shown in PROCEDURE 1.

```
NAME OF THE ALGORITHM(ARGUMENTS)
    // Input ARGUMENT #1: Definition1
    // Input ARGUMENT #2: Definition2
    BODY OF THE PROCEDURE
    // A while loop.
    while [condition]
        [Something]
    // A for loop.
    for Var = [initial value] to [final value]
        [Something]
    // An if-elseif-else block.
    if [Condition1]
        Blah...
    elseif [Condition2]
        Blah...
    elseif [Condition3]
        Blah...
    else
        Blah...
    // A variable assignment.
    blah = A[j]
        // This is indented with a tab.
    // What is the output of this procedure?
    return
```

Chapter 2

Data Structures

2.1 Graphs

2.1.1 Directed Graphs

2.1.1.1 Functions that need to be implemented

2.1.1.2 Binary Decision Diagrams (BDDs)

2.1.1.3 AND-Inverter Graphs (AIGs)

2.1.2 Undirected Graphs

Chapter 3

Optimization

3.1 Benchmarks for Optimization

A collection of "optimization solvers" and benchmarks are available at [25].

Benchmarks for optimization problems:

1. MIPLIB 2010 – Mixed Integer Programming Library version 5 [53]. See [2] for publications associated with this set of benchmarks (or benchmark set).
2.

3.2 Notes on Using Optimization Tools

Optimization problems in EDA can be solved via optimization engines that I implement or external (i.e., third-party) optimization solvers.

Regarding external optimization solvers, some of them use *Algebraic Modeling Languages (AML)* [91] to model the optimization problem computationally. These optimization solvers can solve optimization problems that are formulated as computational models in a specific AML representation.

I am avoiding the use of external optimization solvers that require paid licenses. Hence, any external optimization solvers that I would use are either open-source software (or rather, free/libre/open-source software, FLOSS) or software that have free academic licenses.

Solvers that use an AML, or several AMLs, in their software interface are:

1.

For a list of optimization solvers/tools, see §3.5.

3.3 Robust Linear Programming

During the "lab meeting" on Friday, December 4, 2015, Prof. Jiang Hu told me that I can transform a robust linear programming into a standard/"standard" linear programming problem. He told me to look at [9] and its references.

3.4 Discrete Optimization

Discrete optimization is classified into the following categories [36, 56, 92]:

1. combinatorial optimization
2. integer programming

3.5 Optimization Solvers

A (brief) description of optimization solvers (not restricted to solvers for mathematical programming), including linear programming solvers, is provided as follows in §3.5.1 and §3.5.2.

3.5.1 Accessible Optimization Solvers

External optimization solvers that are open-source software or provide free academic licenses:

1. *LocalSolver* [43]:
   (a) Hybrid solver for optimization problems
   (b) Properties of the solver [43, Product: Overview]:
      i. "next-generation, hybrid mathematical programming solver"
      ii. solve "ultra-large real-life nonlinear problems"
      iii. solve problems in a "model-and-run fashion without any tuning"
      iv. reliable and robust solver: **Define reliability and robustness for solvers of optimization problems.**
      v. dynamically combines solutions from various optimization approaches and resolves them via a hybrid neighborhood search approach
      vi. solver engines:
         A. "local search techniques"
         B. "constraint propagation techniques"
         C. "inference techniques"
         D. linear programming solver/techniques
         E. mixed-integer programming solver/techniques, including mixed-integer linear programming (MILP) solver/techniques; check performance comparisons on MIPLIB benchmarks (`http://www.localsolver.com/news.html?id=32`)
         F. nonlinear programming solver/techniques
         G. combined pure and direct local search techniques
      vii. is based on the *LocalSolver Programming language* (LSP) for mathematical modeling
      viii. has lightweight object-oriented APIs
   (c) From [43, Support Center: Example tour], *LocalSolver* can solve continuous and discrete/-combinatorial optimization problems:
      i. continuous optimization problems:
         A. minimization of the Branin function: find the minimal point of the Branin function, within a specified domain
         B. optimal bucket design: minimization of a bucket encapsulating/covering the rod/-cylinder
         C. Steel mill slab design: mathematical programming
      ii. discrete optimization problems:
         A. car sequencing: scheduling problem, or assignment problem.
         B. Flowshop: scheduling problem
         C. knapsack problem

D. max-cut problem
E. Quadratic Assignment Problem (QAP)
F. Steel mill slab design: integer programming
G. Travelling salesman problem
H. Vehicule routing problem

(d) Its technical documentation can be found at: http://www.localsolver.com/documentation/index.html [42].

2. Stanford University:
   (a) Systems Optimization Laboratory researchers, "SOL Optimization Software," from *Stanford University: School of Engineering: Department of Management Science and Engineering: Systems Optimization Laboratory*, Stanford, CA, 2015. Available online at: http://web.stanford.edu/group/SOL/download.html; last accessed on December 14, 2015.
      i. "Iterative solvers for sparse $Ax = b$: SYMMLQ, MINRES, MINRES-QLP, cgLanczos, CRAIG"
      ii. "Iterative solvers for sparse least-squares problems: LSQR, LSMR, CGLS, covLSQR, LSRN"
      iii. "Sparse and dense LU factorization (direct methods): LUSOL, LUMOD"
      iv. "Sparse optimization: ASP"
      v. "Optimization with convex objective and linear constraints: PDCO (including sparse optimization)"
      vi. "Convex optimization in composite form: PNOPT"
      vii. "Fortran 90 quad-precision dotproduct of double-precision vectors: qdotdd"
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.

3.5.2 Not Accessible Optimization Solvers

External optimization solvers that require paid licenses:

1. Stanford University:
   (a) Systems Optimization Laboratory researchers, "SOL Optimization Software," from *Stanford University: School of Engineering: Department of Management Science and Engineering: Systems Optimization Laboratory*, Stanford, CA, 2015. Available online at: http://web.stanford.edu/group/SOL/download.html; last accessed on December 14, 2015.
      i. LSSOL
      ii. MINOS

    iii. NPSOL
    iv. QPOPT
    v. SNOPT
    vi. SQOPT

2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.

Chapter 4

Mathematics

Math symbols that I use frequently:

1. $\mathbb{N}$
2. $\sum_{n}^{i=1}$
3. $f(x) = \lim_{n \to \infty} \frac{f(x)}{g(x)}$
4. $\varnothing$
5. q

A 3×3 matrix: $\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix}$

Here is an equation:

$$\iint_{\Sigma} \nabla \times \mathbf{F} \cdot \mathrm{d}\mathbf{\Sigma} = \oint_{\partial\Sigma} \mathbf{F} \cdot \mathrm{d}\mathbf{r}. \tag{4.1}$$

Here is an equation that is not numbered.

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

Here is the set of Maxwell's equations that is numbered.

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0} \tag{4.2}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{4.3}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{4.4}$$

$$\nabla \times \mathbf{B} = \mu_0 \left(\mathbf{J} + \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t} \right) \tag{4.5}$$

$$\begin{gathered}
\text{minimize} \sum_{\mathrm{i}=1}^{\mathrm{c}} \mathrm{c_i} \cdot \mathrm{x_i} \\
\underline{x} \in S \\
\text{subject to :} \\
x_1 + x_4 = 0 \\
x_3 + 7 \cdot x_4 + 2 \cdot x_9 = 0
\end{gathered}$$

$$f(n) = \begin{cases} case - 1 & : \text{n is odd} \\ case - 2 & : \text{n is even} \end{cases} \tag{4.6}$$

Proof. This is a proof for BLAH ... □

Theorem 4.1. *TITLE of theorem. My theorem is...*

Axiom 4.1. *TITLE of axiom. Blah...*

Cases of putting a bracket/parenthesis on the right side of the equation.

$$\left.\begin{aligned} B' &= -\partial \times E, \\ E' &= \partial \times B - 4\pi j, \end{aligned}\right\} \qquad \text{Maxwell's equations}$$

Labeling an arrow: $\xrightarrow{ewq}$

Chapter 5

Statistics

Chapter 6

C++ Resources

Quick advice: `Learn how to use` *C++1y* `features, including those of C++11, C++14, and C++17.` Older *C++* versions include *C++98* and *C++03*.

Reference: Free Software Foundation contributors, "C++1y/C++14 Support in GCC," from *GCC, the GNU Compiler Collection: GCC Projects*, Free Software Foundation, Boston, MA, November 14, 2015. Available online at: `https://gcc.gnu.org/projects/cxx1y.html`; last accessed on January 25, 2016.

6.1 Resources for C++ and Notes About C++

Some *C++* resources are:

1. Online *C++* tutorial reference: [15, 16, 18, 86]: `http://www.cplusplus.com/reference/stl/`
2. Pointers to functions [16]: `http://www.cplusplus.com/doc/tutorial/pointers/`
3. An online *C++* reference: [19].
4. From an online *Marshall Cline* resource: [12–14]
5. Books on *C++*: [54, 69]; processed for *C++* templates
6. Embedded *C++*: [50]
7. GUI and other graphics with *C++*: [69]
8. Summaries of *C++*: [46]

Some *C++* STL resources are:

1. Online *C++* STL reference: [17]: `http://www.cplusplus.com/reference/stl/`
2. [19]: `http://en.cppreference.com/w/cpp/container`
3. From an online *C++* STL resource from SGI (formerly, Silicon Graphics, Inc.): [38, 39].
4. Another online C++ STL reference: [61]: `http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm`
5. Other online *C++* STL references: [73, 74].
6. `http://www.cs.wustl.edu/~schmidt/PDF/stl4.pdf`

Books to classify:

1. C++ programming: [70, 76, 79, 82, 83]
2. C++ templates: [40,]

References for *C++* libraries of interest:

1. *Boost C++*: [48, 63, 68, 80]

2.
3.

C++ topics:

1. Function objects:
   (a) https://en.wikipedia.org/wiki/Functional_(C%2B%2B)
   (b) http://stackoverflow.com/questions/356950/c-functors-and-their-uses
   (c) http://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html
   (d) [47, pp. 233–243]
   (e) [70, pp. 327–332, 885, 922–929, 931, 947]
2. Strings:
   (a) [88], Chp 23
   (b) [87], Chp 23
   (c) [35], Chp 18
   (d) [5], Chp 19
   (e) [40, pp. 56–60, string data types, variable vs. literal; strings and string class, 13, 82–87, 320–324, 363–365, 496]
   (f) [47, 655–716]
   (g) [79, pp. 64–67, 320–325, 465–482]
   (h) [81, §14.2]
   (i) [70, pp. 114–131]
   (j) [27], Chp 1
   (k) [37]:
      i. The `put` pointer points "to the next free byte in the `stringstream`." That is, it "holds the address of the next byte in the output area of the" `stringstream`. When the `stringstream` is empty, the `put` pointer points to the beginning of the `stringstream` buffer. [37, §9.8].
      ii. "The type of the `put` pointer" does not matter to the software developer(s), since they "cannot access it directly" [37, §9.8].
      iii. "The `get` pointer holds the address of the next byte in the input area of the stream, or the next byte we get if we use >> to read data from the `stringstream`" [37, §9.9].
      iv. "The `end` pointer indicates the end of the `stringstream`. Attempting to read anything at or after this position will cause the read to fail because there is nothing else to read" [37, §9.9].
      v. Developers only have to know about how `put`, `get`, and `end` pointers work. They do not have to know the actual representation of these pointers [37, §9.9].
      vi. The `stringstream` object acts as a buffer, and is "an area of allocated memory" ("by the stringstream member functions") [37, §9.9].
   (l) [59]:
      i. C strings (or C-strings, or C-style strings) are null-terminated strings (arrays of characters that each end with a terminating "null character" with ASCII value 0) and are arrays of characters; the "null character" is usually represented by the literal character '\0'. "However, an array of `char` is NOT by itself a C string."
      ii. "Since `char` is a built-in data type, no header file is required to create a C string. The C library header file <`cstring`> contains a number of utility functions that operate on C strings."

iii. "It is also possible to declare a C string as a pointer to a `char`: char* s3 = "hello"; " It creates a character array with just enough memory space (in the heap) to store the null-terminated string. The address of the string's first character is placed in the `char` pointer *s3*. When this improperly used, it can corrupt program memory or cause run-time errors.
iv. "[Use] the `C` library function `strlen()`" to determine "the length of a `C` string." It returns an unsigned integer representing the number of characters in the string, excluding the terminating null character.
v. Relational operators (such as $==, !=, >, <, >=, <=$) compare the addresses of the first characters in the two string operands (as the array names are treated as pointers), instead of the contents of these strings.
vi. "Use the `C` library function `strcmp()`" "to compare the contents of two `C` strings." The input arguments of this function are two pointers to `C` strings.
vii. "Use the `C` library function `strcpy()`" to assign a string to a C string or change its contents. The `strcpy()` function accepts a pointer to the C string as the first input argument, and a pointer to the contents of a valid C string or string literal (i.e., a character) as the second input argument. The `C` library function `strcat()` has the same input arguments as `strcpy()`, and is used for concatenating two strings.
viii. C strings can be used as input parameters or the return type. They are specified as `char[]` or `char`*.
ix. "A `C++` string is an object of the class string, which is defined in the header file <string> and which is in the standard namespace." The variable name of a `C++` string is a pointer to the first character of the string; the variable name contains the address of the string's first character. The `C++` string is a dynamically-allocated array of characters.
x. "[Use] the string class methods `length()` or `size()`" to determine "the length of the `C++` strings."
xi. To improve memory efficiency and reduce memory usage, explicitly *pass a string object*. Else, the `C++` string objects are pass and returned by value, which involves making a copy of the string object.
xii. Concatenate C++ strings, C strings, and string literals in any order using the "+" operator.
xiii. Convert a C++ string into a C string via the `c_str()` function of the `string` class. The `c_str()` function returns a pointer to the array of characters representing the string. If the C++ string is not null-terminated, a null character is appended to the new C string. The returned C string "can be used, printed, copied, etc." but not be modified.
xiv. Since programming with arrays can enbug the code more easily, the use of C++ string is (strongly) recommended for use. This is because the properties of a2
xv. When a C string is required by a function, convert the C++ string into a C string (as aforementioned). Instances in which a C string have to be converted into a C++ string are:
   A. Strings passed into `main()` as C strings from the command line argument.
   B. Functions for file input/output operations require filenames to be specified as C strings.
   C. The C++ string class does not have the equivalent functions of certain C string library functions.
   D. Unlike C++ strings, C strings can be serialized in binary format without requiring a bunch of extra code to be written.
xvi. The function `atoi` converts a string to an integer. Similar functions for converting strings into numbers are: `atol` and `atof`. The C++ STL does not have a `itoa` function to convert a number to an integer. However, some compilers supports this function in

the *C Standard General Utilities Library*.

(m) The function `strtol` converts a string into a long integer:
  i. See http://www.cplusplus.com/reference/cstdlib/strtol/.
  ii. [17, <cstdlib> (stdlib.h) – C Standard General Utilities Library: `strtol` function]

(n) Danny Kalev, "String Streams," in *InformIT: The Trusted Technology Learning Source: Articles: Programming: C/C++ Articles: InformIT C++ Reference Guide*, Pearson Education, Indianapolis, IN, January 1, 2003. Available online at: http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=72; last accessed on November 13, 2015.
  i. Static buffers (via *atoi()*, *sprintf()*, or *sscanf()* from the *<stdio.h>*) for type conversions can cause buffer overflow and do not provide adequate type safety (i.e., adequate type checking mechanism). This can be mitigated via *stringstreams*.

3. IO Streams:
   (a) [35], Chp 12
   (b) [88], Chp 10-11
   (c) [57], Chp 8
   (d) [5], Chp 28
   (e) [33, chp. 12].
   (f) [47, pp. 743–847]
   (g) [71], Chp 17q
   (h) [34, Chp. 13].
   (i) [32], Chp 12.
   (j) [79, §2.2; Chp. 6]
   (k) [87], Chp 10-11
   (l) [81, §14.3-14.8]
   (m) [27], Chp 2
   (n) [64], Chp 16
   (o) [84], Chp 21
   (p) [90], Chp 10
   (q) [40, pp. 4, 12–15, 49–51, 150–154, 352, 361–365, 497–498]

4. Templates:
   (a) [35], Chp 11,21
   (b) [88], Chp 19
   (c) [57], Chp 16
   (d) [5], Chp 29
   (e) [33, §16.2–§16.4]
   (f) [47, pp. 13, 26–27, 33–34, 36, 62, 68, 1024]
   (g) [34, §16.2–§16.4]
   (h) [79, Chp. 17]
   (i) [87], Chp 19
   (j) [69, §6.16, pp. 193–195]
   (k) [1], book
   (l) [27], Chp 3
   (m) [64], Chp 24
   (n) [84], Chp 18
   (o) [89], book
   (p) [4], book; typelist - Chp 3

   (q) [90], Chp 6
   (r) [26], Chp 16

5. Debugging:
   (a) [27], Chp 11 (especially memory management problems, pp. 533)

6. STL, books on *C++* STL:
   (a) [47]
   (b) [79, Chp. 18, pp. 943–998]
   (c) [72]
   (d) [75]
   (e) [45]

7. STL containers:
   (a) [35], Chp 15-16
   (b) [57], Chp 9,11
   (c) [5], Chp 18
   (d) [71], Chp 16
   (e) [28]:
      i. `vector<int> v(10);` *// Create an int vector of size 10.*
      ii. `v[5] = 10;` *// Target of this assignment is the return value of operator[].*
   (f) [79, §18.2, pp. 960–977]
   (g) [72], book
   (h) [81, Chp. 8]
   (i) [85], Chp 8
   (j) [27], Chp 4
   (k) [64], Chp 25
   (l) [90], Chp 7

8. STL algorithms:
   (a) [35], Chp 15,17
   (b) [57], Chp 10
   (c) [5], Chp 18
   (d) [71], Chp 16
   (e) [79, §18.3, pp. 977-991]
   (f) [72], book
   (g) [27], Chp 5
   (h) [64], Chp 25
   (i) [90], Chp 7

9. Function addresses:
   (a) [88], Chp 8
   (b) [87], Chp 8
   (c) [26], Chp 3, pp. 213

10. Dynamic memory management problems:
   (a) [35], Chp 10,22
   (b) [57], Chp 12,13
   (c) [5], Chp 14
   (d) [33, §13.9, 750–754].

   (e) [71], Chp 9,12
   (f) [32], Chp 13.
   (g) [60], Chp 2-4
   (h) [70, Chp. 9, pp. 393–423]
   (i) [84], Chp 29
   (j) [26], Chp 6,13

11. Function overloading:
    (a) [88], Chp 8
    (b) [32], Chp 6. See all of [32–34].
    (c) [79, §4.4, pp. 230–243]
    (d) [87], Chp 8
    (e) [84], Chp 14
    (f) [26], Chp 7

12. Operator overloading:
    (a) [57], Chp 14
    (b) [79, §11.2, pp. 633–651]
    (c) [64], Chp 18
    (d) [84], Chp 15
    (e) [26], Chp 12

13. Constants:
    (a) [26], Chp 8

14. Functions and pointers:
    (a) [88], Chp 8:
        i. Pass-by-reference:
           A. e.g., void init(vector<double> &v)
           B. "It is not possible to refer directly to a reference variable after it is defined; any occurrence of its name refers directly to the variable it references. "
           C. "Once a reference is created, it cannot be later made to reference another variable. This is something that is often done with pointers."
           D. "References cannot be null, whereas pointers can; every reference refers to some variable, although it may or may not be valid."
           E. "References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created. In particular, local and global variables must be initialized where they are defined, and references which are data members of a class must be initialized in the initializer list of the class's constructor."
           F. Avoid mixing references and pointers in a block of code to avoid confusion, and make it easier for the *C++* code to be read and debug.
           G. The required syntax for pointers make them prominent in comparison to that of references.
           H. The number of operations on references is less than that on pointers. Hence, usage of references is easier to understand than that of pointers. Consequently, it is easier to use references than pointers without enbugging the code.
           I. Pointers can be invalidated as follows:
              - "Carrying a null value"
              - "Out-of-bounds [pointer] arithmetic"

        - Illegal casts on pointers
        - Produce pointers from random integers
      J. References can be invalidated as follows:
        - "[Refer] to a variable with automatic allocation which goes out of scope"
        - "[Refer] to an object inside a block of dynamic memory which has been freed"
      K. "Arrays are always passed by address. This includes C strings."
      L. "Dynamic storage is allocated using pointers."
      M. Reference: Kurt McMahon, "Passing Variables by Address," in *Northern Illinois University: College of Engineering and Engineering Technology: Department of Computer Science: CSCI 241 Intermediate Programming in C++ (Fall 2015): Notes*, Northern Illinois University, DeKalb, IL, October 28, 2015. Available online at: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/pass_by_address.html; last accessed on November 3, 2015.
    ii. Pass-by-const-reference: e.g., void print(const vector<double> &v)
    iii. Pass-by-value: e.g., void fn(int x)
    iv. Pass-by-address: e.g., void print(int * ptr)
      A. Reference: Kurt McMahon, "Passing Variables by Address," in *Northern Illinois University: College of Engineering and Engineering Technology: Department of Computer Science: CSCI 241 Intermediate Programming in C++ (Fall 2015): Notes*, Northern Illinois University, DeKalb, IL, October 28, 2015. Available online at: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/pass_by_address.html; last accessed on November 3, 2015.

(b) [57], Chp 6
(c) [5], Chp 12-13
(d) [71], Chp 7-8
(e) [79, Chp. 4–5; §11.1; and Chp. 14]
(f) [87], Chp 8
(g) [70, Chp. 7–8, pp. 279–391]
(h) [64], Chp 15,20
(i) [26], Chp 11:
    i. use const at the end of accessor functions
    ii. Do not use pointers as instance variables
(j) Elsewhere:
    i. http://stackoverflow.com/questions/1143262/what-is-the-difference-between-const- [62]:
      A. Read it backwards; the first *const* can be on either side of the type.
      B. "Read pointer declarations right-to-left."
      C. From the answer of Ted Dennison, July 17, 2009. **Rule: The "const" goes after the thing it applies to. Putting const at the very front (e.g., const int *) is an exception to the rule.**
      D. int* – pointer to int
      E. int const * == const int * – pointer to const int
      F. int * const – const pointer to int
      G. int const * const == const int * const – const pointer to const int
      H. int ** – pointer to pointer to int
      I. int ** const – A const pointer to a pointer to an int
      J. int * const * – A pointer to a const pointer to an int
      K. int const ** – A pointer to a pointer to a const int
      L. int * const * const – A const pointer to a const pointer to an int

ii. For the following [62], let: *int var0 = 0;*
   A. *const int &ptr1 = var0;* // Constant reference
   B. *int * const ptr2 = &var0;* // Constant pointer
   C. *int const * ptr3 = &var0;* // Pointer to const
   D. *const int * const ptr4 = &var0;* // Const pointer to a const

iii. [65]:
   A. "A reference is a variable that refers to something else and can be used as an alias for that something else. A pointer is a variable that stores a memory address, for the purpose of acting as an alias to what is stored at that address. So, a pointer is a reference, but a reference is not necessarily a pointer. Pointers are a particular implementation of the concept of a reference, and the term tends to be used only for languages that give you direct access to the memory address. References can be implemented internally in a language using pointers, or using some other mechanism." Answer from dan1111.
   B. "Passing an object by value means making a copy of it. You can modify that copy without affecting the original. Making that copy can cost a lot of memory access though. Passing an object by reference means passing a handle to that object. This is cheaper because you don't need to make a copy. It also means that any changes you make will affect the original." Answer from Steve Rowe.
   C. "There is no such thing as a null reference. A reference must always refer to some object. As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable a pointer, because then you can set it to null. On the other hand, if the variable must always refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference." Answer from Harssh S. Shrivastava.

iv. A pointer is dereferenced via the explicit * operator. The * operator should not be used to dereference a reference (variable) [77].

v. [77]:
   A. int *pi = &i; // Indirect expression to dereference *pi* to *i*. "Declare *pi* as an object of type 'pointer to int' whose initial value is the address of object *i*" [78].
   B. int &ri = i; // *ri* is dereferenced to refer to *i*. "Declares *ri* as an object of type 'reference to int' referring to *i*" [78].
   C. The *C++* standard does not dictate how compilers shall implement references. However, popular compilers tend to implement references as pointers. Therefore, there are no significant advantages of using references or pointers.

vi. [78]:
   A. "A valid reference must refer to an object; a pointer need not. A pointer, even a const pointer, can have a null value. A null pointer doesn't point to anything."
   B. I can bind a reference to a null pointer, but I cannot dereference a null pointer since it can "produce undefined behavior".

vii. You cannot call a non-const method from a const method. That would 'discard' the const qualifier.:
   A. http://stackoverflow.com/questions/2382834/discards-qualifiers-error

viii. Pointer to constant data: *const type* variable;* and *type const * variable;*
   A. http://www.cprogramming.com/reference/pointers/const_pointers.html

ix. Pointer with constant memory address: *type * const variable = some-memory-address;*
   A. http://www.cprogramming.com/reference/pointers/const_pointers.html

x. Constant data with a constant pointer: *const type * const variable = some-memory-address;* and *type const * const variable = some-memory-address;*
A. http://www.cprogramming.com/reference/pointers/const_pointers.html

(k) With shallow copying, I would only copy the memory references or pointers. The copy and the original reference the same object. On the other hand, with deep copying, I would copy the values; this is also known as cloning. The copy and the original reference do not share objects; each of them references its own object. The default copy constructor carries out shallow copy.

15. Extern function:
    (a) :
        i.
    (b) :
        i.
    (c) :
        i.
    (d) :
        i.
    (e) :
        i.
    (f) :
        i.
    (g) :
        i.

16. `typename`:
    (a) Evan Driscoll, "A Description of the C++ `typename` keyword," from the Department of Computer Sciences, University of Wisconsin-Madison College of Engineering, University of Wisconsin-Madison, Madison, WI. Available online at: http://pages.cs.wisc.edu/~driscoll/typename.html; last accessed on February 15, 2016.
    (b) From [6, API documentation for Eigen3: The template and typename keywords in C++], or http://eigen.tuxfamily.org/dox/TopicTemplateKeyword.html.
    (c) Wikipedia contributors, "typename," in *Wikipedia, The Free Encyclopedia: C++*, Wikimedia Foundation, San Francisco, CA, April 13, 2015. Available online at: https://en.wikipedia.org/wiki/Typename; last accessed on February 15, 2016.:
        i. Usage #1: "A synonym for "class" in template parameters"
        ii. Usage #2: "A method for indicating that a dependent name is a type"
    (d) [79, pp. 916]
    (e)
    (f)
    (g)
    (h)
    (i)
    (j)
    (k)

17. OOD and inheritance:
    (a) [35], Chp 4-9

(b) [88], Chp 9
(c) [57], Chp 7,15,18,19
(d) [5], Chp 24-26
(e) [33, Chp. 13–15; Appendices E and J].
(f) [71], Chp 10-11,13,14,15
(g) [34, Chp. 7, 11, 15; Appendices A, D, J, and K].
(h) [32], Chp 13,14,15.
(i) [79, Chp. 10; §12.1, 696–711; Chp. 15; Chp. 17]
(j) [87], Chp 9
(k) [64], Chp 13-14,21
(l) [90], Chp 3-4,8
(m) [26], Chp 14,15

18. SW engineering issues:
    (a) [35], Chp 24-26
    (b) [5], Chp 21
    (c) Debugging:
        i. [81, §14.2]
19. multi-threading:
    (a) [85], Chp 3
20. graphs:
    (a) [85], Chp 7
21. typedef:
    (a) In the sandbox, use the *Make* target *make typedef* to study an example of how *typedef* can be used. When the *header file* defines/specifies the *typedef*, and is included in the *C++ implementation file* and other *C++ implementation file*s that instantiates those objects, it can be used subsequently without additional definition/specification. October 6, 2015.
    (b) [79, pp. 510-512]

6.2 Computational Complexity of C++ Containers

Table 6.1 shows a tabulated summary of containers in the *C++* Standard Template Library (STL) and the computational complexity for each of their common operations: add(element e), remove(element e), search(element e), size(), empty(), begin(), and end().

To conclude, we can get some facts about each data structure:

1. `std::list` is very very slow to iterate through the collection due to its very poor spatial locality.
2. `std::vector` and `std::deque` perform always faster than `std::list` with very small data
3. `std::list` handles very well large elements
4. `std::deque` performs better than a `std::vector` for inserting at random positions (especially at the front, which is constant time)
5. `std::deque` and `std::vector` do not support very well data types with high cost of copy/assignment

This draws simple conclusions on the usage of each data structure [11, 45]:

1. Number crunching: use std::vector or std::deque

Table 6.1: Computational Complexity of Basic Operations of Containers from the *C++ STL*.

Container \ Complexity	add	remove	search	size	empty	begin	end
vector	O(1)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
list	O(1)	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
queue	O(1) amortized	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)
priority queue	O(log n)	O(log n)	O(n)	O(1)	O(1)	O(1)	???
set	O(log n)	O(log n)	O(log n)	O(1)	O(1)	O(1)	O(1)
multi-set	O(log n)	???	O(log n)	O(1)	O(1)	O(1)	O(1)
map	O(log n)	O(log n)	O(log n)	O(1)	O(1)	O(1)	O(1)
multi-map	O(log n)	???	O(log n)	O(1)	O(1)	O(1)	O(1)
stack	O(1)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)

2. Linear search: use std::vector or std::deque
3. Random Insert/Remove:
4. Small data size: use std::vector
5. Large element size: use std::list (unless if intended principally for searching)
6. Non-trivial data type: use std::list unless you need the container especially for searching. But for multiple modifications of the container, it will be very slow.
7. Push to front: use std::deque or std::list

Notes about asymptotic notations:

1. Comparison of big O notations, and other asymptotic notations, in general – based on "running time (T(n))" [Wikipedia 2015a][Wikipedia 2015]:
   (a) $O(1)$: constant time
   (b) $O(\log^* n)$, log star: iterated logarithmic time.
   Log star n is a recursive function; $\log^* n := \begin{cases} 0 & : \text{if } n \leq 1 \\ 1 + \log^*(\log n) & : \text{if } n > 1 \end{cases}$ [Wikipedia 2015b]
   (c) $O(\log \log n)$: log-logarithmic, double logarithmic
   (d) $O(\log n)$: logarithmic time, computational time complexity class DLOGTIME. E.g., $\log n^2$.
   (e) poly($\log n$) or $O((\log n)^c), c > 1$: polylogarithmic time. E.g., $(\log n)^2$.
   (f) $O(n^c)$, where $0 < c < 1$: fractional power. E.g., $n^{\frac{2}{3}}$.
   (g) $o(n)$: sub-linear time (or sublinear time)
   (h) $O(n)$: linear time
   (i) $O(n \log^* n)$: "n log star n" time, or "n log-star n"
   (j) $O(n \log n) = O(\log n!)$: linearithmic time, including $\log n!$. Or, loglinear, or quasilinear.
   (k) $O(n^2)$: quadratic time
   (l) $O(n^3)$: cubic time
   (m) poly(n), or $2^{O(\log n)}$. Or, $O(n^c), c > 1$: polynomial time, including $n, n \log n, n^{10}$. Computational time complexity class P. Or, algebraic.
   (n) $2^{\text{poly}(\log n)}$: quasi-polynomial time, including $n^{\log \log n}, n^{\log n}$. Computational time complexity class QP.
   (o) $O(2^{n^\varepsilon}), \forall \varepsilon > 0$: sub-exponential time, including $O(2^{\log n^{\log \log n}})$. Computational time complexity class SUBEXP.
   (p) $2^{o(n)}$: sub-exponential time, including $2^{n^{\frac{1}{3}}}$. Computational time complexity class SUBEXP. Or, L-notation.

(q) $2^{O(n)}$: exponential time (with linear exponent), including $1.1^n, 10^n$. Computational time complexity class E.
(r) $2^{\text{poly}(n)}$. Or, $O(c^n), c > 1$: exponential time, including $2^n, 2^{n^2}$. Computational time complexity class EXPTIME.
(s) $O(n!)$: factorial time, including $n!$.
(t) $2^{2^{\text{poly}(n)}}$: double exponential time, including 2^{2^n}. Computational time complexity class 2-EXPTIME.
(u) $n! > n^n$

2. Types of asymptotic notations [Wikipedia 2015]:
   (a) $f(n) = O(g(n))$: Big O notation, or Big Oh notation
   (b) $f(n) = \Omega(g(n))$: Big Omega notation
   (c) $f(n) = \Theta(g(n))$: Big Theta notation
   (d) $f(n) = o(g(n))$: Small O notation, or Small Oh notation
   (e) $f(n) = \omega(g(n))$: Small Omega notation
   (f) $f(n) \sim g(n)$: "On the order of"
3. References:
   (a) [Wikipedia 2015] Wikipedia contributors, "Big O notation," sections *Orders of common functions* and *Related asymptotic notations: Family of Bachmann?Landau notations*, in *Wikipedia, The Free Encyclopedia: Analysis of algorithms, or Asymptotic analysis*, Wikimedia Foundation, San Francisco, CA, November 29, 2015. Available online at: https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions; last accessed on December 1, 2015.
   (b) [Wikipedia 2015a] Wikipedia contributors, "Time complexity," section *Table of common time complexities*, in *Wikipedia, The Free Encyclopedia: Computational complexity theory*, Wikimedia Foundation, San Francisco, CA, November 16, 2015. Available online at: https://en.wikipedia.org/wiki/Time_complexity#Table_of_common_time_complexities; last accessed on December 1, 2015.
   (c) [Wikipedia 2015b] Wikipedia contributors, "Iterated logarithm," in *Wikipedia, The Free Encyclopedia: Asymptotic analysis*, Wikimedia Foundation, San Francisco, CA, November 6, 2015. Available online at: https://en.wikipedia.org/wiki/Iterated_logarithm; last accessed on December 1, 2015.
4. Note that I denote "is defined as" as: $\equiv, \triangleq, \stackrel{\text{def}}{=}, :=$
5. Note that $\log n$ is faster than $(\log n)^2$, although initially the latter is slightly faster than the former (for negligibly small n).

Books on computational complexity:

1. [47, pp.10–11]

6.3 Additional Notes About C++

Static variables:

1. K. Hong, "Static Variables and Static Class Members - 2015," San Francisco, CA. Available online from *Open Source . . . : Java/C++/Python/Android/Design Patterns: C++ Tutorial Home – 2015* at: http://www.bogotobogo.com/cplusplus/statics.php; last accessed on October 23, 2015.

Formatting data:

1. Synesis Software Pty Ltd staff, "Synesis Software Training Courses: FastFormat, Beginner's (part 1 of 2)," Synesis Software Pty Ltd, Sydney, Australia, 2015. Available online at: http://www.synesis.com.au/training-beginners-fastformat.html; December 1, 2015 was the last accessed date.
   (a) "Formatting APIs":
      i. "Replacement-based APIs":
         A. "Streams (printf()-family)"
         B. "Boost.Format"
         C. "FastFormat.Format"
      ii. "Concatenation-based APIs":
         A. "IOStreams"
         B. "Loki.SafeFormat"
         C. "FastFormat.Write"
   (b) "struct tm"
   (c) "struct in_addr"
   (d) "ATL types"
   (e) "ACE types"
2. Synesis Software Pty Ltd staff, "Synesis Software Training Courses: FastFormat, Advanced (part 2 of 2)," Synesis Software Pty Ltd, Sydney, Australia, 2015. Available online at: http://www.synesis.com.au/training-advanced-fastformat.html; December 1, 2015 was the last accessed date.
   (a) "Format-specification Defect Handling": "Scoping" and "Disgnostic Logging"

6.3.1 Alternate Computer Number System for Representing Fractions in C++

An alternate computer number system for representing fractions in C++ is the fixed-point number system. For a detailed classification of computer number systems, see [58].

In *C++*, the numerical data types are based on cardinal numbers (e.g., one, two, three, ...), instead of ordinal numbers/integers (e.g., first, second, third, ...); see [24,66,94] for the definitions of "cardinal number" and "ordinal number." From [66], "a Nominal Number is a number used only as a name, or to identify something (not as an actual value or position)." E.g., "the number on the back of a footballer ("8")," "a postal code ("91210")," and "a model number ("380")."

Resources to help me implement the fixed-point "data type" as a class in *C++*, and fixed-point arithmetic:

1.

6.4 Software Development in C++

Notes about software development in *C++*:

1. Notes from Synesis Software Pty Ltd:
   (a) Synesis Software Pty Ltd staff, "Synesis Software Training Courses," Synesis Software Pty Ltd, Sydney, Australia, 2015. Available online at: http://www.synesis.com.au/training.html; December 1, 2015 was the last accessed date.
      i. Use `FastFormat` as a "C++ diagnostic logging API library"

    ii. `STLSoft libraries`. "Apply the concepts, principles and techniques of Extended STL to enhance the expressiveness, flexibility, and performance of your C++ software." See [93] for more details.
    iii. "Building Bullet-Proof Software in C++ - no system built by Synesis Software has ever failed in production. This course takes you through the principles and practices of how we develop software, providing you with practical, applicable strategies and tactics for achieving the same outcome in your software developments."
    iv. "Guerilla Testing C++ - or, **'How to discover the Gold Nuggets in your Big Ball of Mud'**. No matter how badly a C++ codebase is enmeshed, you can get it under test if you know how to master its coupling."

(b) Synesis Software Pty Ltd staff, "Resources," Synesis Software Pty Ltd, Sydney, Australia. Available online at: `http://www.synesis.com.au/resources.html`; December 1, 2015 was the last accessed date.
    i. 100% type-safe *C++* API
    ii. C++ diagnostic logging API library (or, diagnostic logging libraries):
        A. Pantheios: `http://pantheios.org/`
        B. ACE
        C. log4cxx
    iii. C++ formatting library: FastFormat `http://fastformat.org/`
    iv. "The STLSoft libraries provide STL extensions and facades over operating-system and third-party-library APIs. The libraries are 100% header-only." See `http://stlsoft.org/`.
    v. "UNIXem is a simple library that emulates a useful subset of the UNIX system APIs on Windows... UNIXem is the only library provided by Synesis Software that is not production-quality. It is appropriate for research, such as when developing tests for cross-platform software." See `http://synesis.com.au/software/unixem.html`.

(c) Synesis Software Pty Ltd staff, "Guerilla Testing C++ or, 'How to discover the Gold Nuggets in your Big Ball of Mud'," Synesis Software Pty Ltd, Sydney, Australia. Available online at: `http://www.synesis.com.au/training-guerilla-testing-cplusplus.html`; December 1, 2015 was the last accessed date.:
    i. "Change is the most expensive part of the cost of a software project. The biggest impediments to change are lack of clarity on what to alter to effect the change, and uncertainty about unintended side-effects of the change."
    ii. "No matter how badly a C++ codebase is enmeshed, you can get it under test if you know how to master its coupling."
    iii. "Many long-lived codebases have evolved to a point where some, perhaps most, aspects of its functionality are no longer precisely known / codified / automatically tested. This course will teach you, using practical examples, how to wrest control from any codebase, no matter how badly enmeshed, isolate known pieces of good functionality, get them under test, and eventually to isolate and separate them into a new context, while, where required, maintaining compatibility with their original context."
    iv. "This course will teach you how to refactor any codebase with confidence, rather than poking at the edges of its functionality in fear."
    v. "Release costs" serve as an indicator to the existence of "a Big Ball of Mud."
    vi. "Factors that inhibit testing":
        A. "Coupling, coupling, coupling"
        B. "The inconstant environment"
        C. "Trust"
        D. "Defensive code"
        E. "Fuzzy (or no!) abstraction borders"

vii. "Key characteristics" identified in situ: "diagnostics, contracts, code coverage, and testing."
viii. Remember the following "when testing mud-balls": "automation; minimalism, incrementality, unit testing vs component testing; coverage (in realistic time); only change what you can test (and are testing!) – [there are] exceptions to this rule; beyond salvation – sometimes it's just mud."
ix. Islands of "known Functionality" are created as follows:
    A. "Decomposition – Identifying Units, Identifying Components, and Identifying Modules"
    B. "Triage"
    C. "Isolation"
    D. "Striding two worlds"
    E. "Transplantation"
    F. "Separation"
    G. "Versioning – Static and Dynamic"
    H. "When to 'throw it out'."
x. "Inconstant Environment" handling:
    A. "File system"
    B. "Memory"
    C. "User-interface"
    D. "Time"
    E. "Data storage"
xi. Techniques to address/mitigate coupling:
    A. "Pre-processor":
        - #ifdef
        - #define
        - #include
    B. "linkage":
        - "interpositioning"
        - "dynamic library redirection"
    C. "object-oriented techniques":
        - "overloading"
        - "overriding"
        - "inheritance"
        - "interfaces"
    D. "patterns":
        - "class adaptor"
        - "instance adaptor"
        - "decorator"
        - "visitor"
    E. "generic programming":
        - "policies"
        - "shims"
        - "traits"
    F. "Testing":
        - "Stubbing"
        - "Mocking"
        - "Versioned testing"
xii.
xiii.

xiv.
xv.
xvi.
xvii.
xviii.
xix.

6.4.1 Using "Design By Contract"

The "Design By Contract" approach shall be used in software development. This approach is also known as: "contract programming, programming by contract, and design-by-contract programming." Adhere strongly to Hoare logic.

References:
Wikipedia contributors, "Design by contract," in *Wikipedia, The Free Encyclopedia: Software design*, Wikimedia Foundation, San Francisco, CA, January 20, 2016. Available online at: https://en.wikipedia.org/wiki/Design_by_contract; last accessed on February 9, 2016.

Wikipedia contributors, "Hoare logic," in *Wikipedia, The Free Encyclopedia: Static program analysis*, Wikimedia Foundation, San Francisco, CA, November 8, 2015. Available online at: https://en.wikipedia.org/wiki/Hoare_logic; last accessed on February 10, 2016.

That is, at the start of an implementation of a (*C++*) function, check that its precondition(s) is (/are) met; preconditions shall be chosen to be as weak as possible. Within the implementation of the function, check if the assertions (properties that must be true during execution of the function) hold. Lastly, at the end of the implementation, check that its postcondition(s) is (/are) met; postconditions shall be chosen to be as strong as possible.

Reference:
Wikipedia contributors, "Predicate transformer semantics," in *Wikipedia, The Free Encyclopedia: Formal methods*, Wikimedia Foundation, San Francisco, CA, November 25, 2015. Available online at: https://en.wikipedia.org/wiki/Predicate_transformer_semantics; last accessed on February 10, 2016.

6.4.1.1 Hoare Logic for Computer Arithmetic

Check if arithmetic and logical operations cause overflows or underflows [20, 21]. When faced with a constrained range for representing real numbers with bits in computer hardware, a constrained resolution such that a representation smaller than single-precision floating-point numbers is required, or low-cost and/or low-power electronic/computer systems that do not have floating-point arithmetic circuits, use a circuit-based implementation of fixed-point arithmetic.

6.4.2 Debugging *C++* Software

This section covers debugging syntax errors (in §6.4.2.1) that are reported by *C++* compilers. It also covers semantic errors that can be discovered via software testing and formal verification tools; see [23, §4.4, pp. 119]. In addition, it includes performance debugging [23, §4.5, pp. 119–120], using software profilers [29, Figure 7.1, pp. 292; and §7.2.10, pp. 302] [51, pp. 148] [55, pp. 35-9 – 35-10] [30, §3.2, pp. 16–18] [7, §3.3.2, pp. 21–22; pp. 23; Figure 40, pp. 102; Appendix C, §C.1.7, pp. 104] and static analysis software [7, §5.4, 65-66; and Figure 40, pp. 102] [3] [8, §7.1, 176–183] [22, §5.2.4, pp. 82–83; and §5.4.2, pp. 90–92] [31, Chapter 7, pp. 109–117] [55, §35.5, pp. 35-10 – 35-14] [10]. These phases cannot be highly overlapped consider by more than a significant amount.

To functionally debug software with success, these four steps should be carried out: "test input generation, error detection, error diagnosis, and error correction" [52].

6.4.2.1 Interpreting *C++* Compilation Errors

A missing semiconlon ";", or lack of matching curly braces "}" (or parentheses ")", square brackets "]", or angle brackets ">"), can cause a lot of compilation errors [41].

6.4.3 Parser Development

Parser development via *Lex/Yacc*, *Flex/Bison*, *ANTLR*, *Parsec*, *Ragel*, *Spirit Parser Framework*, *Jet-PAG (Jet Parser Auto-Generator)*, *Monkey*, *MyParser*, *SableCC*, ...

Reference: Wikipedia contributors, "Comparison of parser generators," in *Wikipedia, The Free Encyclopedia: Parser generators*, Wikimedia Foundation, San Francisco, CA, February 18, 2016. Available online at: `https://en.wikipedia.org/wiki/Comparison_of_parser_generators`; last accessed on February 18, 2016.

Determine if `LLVM` *can be used for parser development.*

6.5 Parallel Programming in C++

Resources for parallel programming in C++:

1. C++ -based MPI programming: [49]

6.6 Numerical Computing in C++

Numerical computing resources for *C++*:

1. Scientific computing: [67]
2.

Chapter 7

Questions

7.1 Unresolved C++ Questions

Questions about `C++`:

1.

7.2 Resolved C++ Questions

Difference between pointers and references:

1. Yusuf Kemal Özcan ("BFaceCoder"), "Is there any difference between pointers and references? [duplicate]," Stack Exchange Inc., New York, NY, April 18, 2013. Available online from *Stack Exchange Inc.: Programmers Stack Exchange: Questions* at: http://programmers.stackexchange.com/questions/195337/is-there-any-difference-between-pointers-and-references; October 6, 2015 was the last accessed date.
   (a) Answer from *dan1111*, April 18, 2013: http://programmers.stackexchange.com/a/195343 and http://programmers.stackexchange.com/questions/195337/is-there-any-difference-be 195343#195343.
   (b)

2. Macneil Shonle and Programmers Stack Exchange contributors, "What's a nice explanation for pointers? [closed]," Stack Exchange Inc., New York, NY, July 30, 2015. Available online from *Stack Exchange Inc.: Programmers Stack Exchange: Questions* at: http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanation-for-pointers; October 6, 2015 was the last accessed date.
   (a) Answer from Kevin, November 10, 2010: http://programmers.stackexchange.com/a/17919 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanation-fo 17919#17919. "A pointer is a variable that contains an address to a variable. A pointer is both defined and dereferenced (yielding the value stored at the memory location that it points to) with the '*' operator; the expression is mnemonic." ... `char (*(x())[])()`
   (b) Answer from Barfield, November 10, 2010: http://programmers.stackexchange.com/a/18087 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 18087#18087. "Pointer[s] are a bit like the application shortcuts on your desktop."
   (c) Answer from Gulshan, November 10, 2010: http://programmers.stackexchange.com/a/17915 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 17915#17915. Pointers point to instance and static variables. A pointer can point to different variables during the execution of the program, but must point to one variable at

any instance (i.e., point in time) during execution. Also, the pointer must point to variables of the same type. Associate a pointer with a variable via the reference to the variable; e.g., `int *pointer; pointer = & variable;` ... According to *Ptolemy*, December 2, 2010: http://programmers.stackexchange.com/a/23016. `int *pointer = & variable;` creates a pointer to the variable. ... Dereference the pointer (add * as a prefix) to store the value of an expression (based on variables, strings, or constants). According to *Ptolemy*, `& variable` is the "address of the variable" and it "represents the literal value for" the pointer. "The pointer" refers to the data that the pointer points to, or something "pointed to by" the pointer.

(d) Answer from Sridhar Iyer, November 11, 2010: http://programmers.stackexchange.com/a/18529 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explana 18529#18529. A "pointer is a variable that store[s] the address of another variable (or just any variable). * is used to get the value at the memory location that is stored in the pointer variable. & operator gives the address of a memory location."

(e) Answer from *rwong*, November 2, 2010: http://programmers.stackexchange.com/a/18054 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanation-fo 18054#18054. Each pointer, which is a special type of variable, must point to only one variable. Variables that are not pointers must not point to anything; however, such variables can be pointed to by any number of pointers.

(f) Answer from *back2dos*, November 10, 2010: http://programmers.stackexchange.com/a/18092 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 18092#18092. The pointer [variable] interprets the value of the pointer [variable] as the address of another variable that it points to. Hence, the value of the pointer [variable] refers to a specific location in memory (specified by the address), and is called the reference. Dereferencing is the process of accessing the value of the memory location that it points/refers to. That is, $*v$ dereferences the value of v, and provides the value at the memory location referred to by the address in v. $\&v$ provides a reference (or the address of the memory location for v) to the variable v.

(g) Answer from *Ptolemy*, December 2, 2010: http://programmers.stackexchange.com/a/23016 and http://programmers.stackexchange.com/questions/17898/whats-a-nice-explanati 23016#23016. At a low level, the concept of memory can be viewed as a massive array. "Any position in the array" can be accessed "by its index location." "Passing the index location rather than copying the entire memory" is more efficient in terms of performance and memory usage. Hence, "pointers are useful." "For [a] method to store the index location [of] where all the data [in the array] is stored," "a memory index location" can be passed in as a parameter. Pointers can be chained indefinitely; "keep track of how many times [I] need to look at the addresses to find the actual data object." While pointers to heap memory are safe, "pointers to stack memory are dangerous when passed outside the method."

(h) Also, see http://www.udel.edu/CIS/105/pconrad/03F/2003.fall.doc by "P. Conrad."

3. [44, pp. 15, second last paragraph]
   (a) "The value of a pointer is the address to which it points"; or, the "the value of a pointer is the address."

4. [28]
   (a) "pointers use the * and $->$ operators, references use ."
   (b) "Both pointers and references let you refer to other objects indirectly."
   (c) "there is no such thing as a null reference"
   (d) "A reference must always refer to some object."

   (e) **"As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable a pointer, because then you can set it to null."**
   (f) *"On the other hand, if the variable must always refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference."*
   (g) "Because a reference must refer to an object, C++ requires that references be initialized." ... Pointers do not have to be initialized; i.e., pointers can be uninitialized. However, "uninitialized pointers" are "valid but risky."
   (h) Since null references do not exist, references can be used more efficiently than pointers. This is because the validity of a reference does not have to be tested prior to usage.
   (i) Before using pointers, they should be tested against null (i.e., check the validity of a reference prior to usage).
   (j) "Pointers may be reassigned to refer to different objects." "A reference ... always refer to the object with which it is initialized."
   (k) "You should use a pointer whenever you need to take into account the possibility that there's nothing to refer to (in which case you can set the pointer to null) or whenever you need to be able to refer to different things at different times (in which case you can change where the pointer points)."
   (l) "You should use a reference whenever you know there will always be an object to refer to and you also know that once you're referring to that object, you'll never want to refer to anything else."
   (m) "There is one other situation in which you should use a reference, and that's when you're implementing certain operators. The most common example is operator[]. This operator typically needs to return something that can be used as the target of an assignment."
   (n) "References, then, are the feature of choice when you know you have something to refer to, when you'll never want to refer to anything else, and when implementing operators whose syntactic requirements make the use of pointers undesirable. In all other cases, stick with pointers."

5. Prakash Rajendran, Theodore Logan (Commodore Jaeger), Josh Lee, sbi, Rob$_{\varphi}$, Sudhanshu Aggarwal, lpapp, Alf, Deduplicator, Sam, and Siddhant Saraf, "What are the differences between a pointer variable and a reference variable in C++?," Stack Exchange Inc., New York, NY, March 2, 2015. Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: http://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-a October 8, 2015 was the last accessed date.
   (a) A pointer can be re-assigned any number of times while a reference can not be re-seated after binding.
   (b) Pointers can point nowhere (NULL), whereas reference always refer to an object.
   (c) You can't take the address of a reference like you can with pointers.
   (d) There's no "reference arithmetics" (but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).
   (e) Use references in function parameters and return types to define useful and self-documenting interfaces.
   (f) Use pointers to implement algorithms and data structures.
6. (a)
7. (a)
8. (a)
9. (a)
10. (a)
11. (a)

12. (a)
13. (a)
14. (a)
15. (a)
16. (a)

Chapter 8

Miscellaneous

8.1 Setting Up Software Development Environment

Setting up software development environment for `C++`:

1. Platform-independent environments and software:
   (a) *Linux*, *Mac OS X*, and *Microsoft Windows*:
      i.
   (b) Truly platform independent:
      i.
2. *Mac OS X*:
   (a) Integrated development environments (IDEs):
      i. *Xcode*:
         A. *Preferences* $\Longrightarrow$ *Text Editing* $\Longrightarrow$ *Editing* $\Longrightarrow$ *Code folding ribbon*
         B. *Preferences* $\Longrightarrow$ *Text Editing* $\Longrightarrow$ *Indentation* $\Longrightarrow$ *Syntax-aware indenting: Automatically indent based on syntax* + *Indent "//" comments one level deeper* + *Align consecutive "//" comments*
3. *Linux*:
   (a) Text editors:
      i. *gedit*:
         A.
      ii. *NEdit*:
         A.
   (b)

Acknowledgments

I would like to thank Ms. Deepika Panchalingam for motivating me to revise basic data structures and algorithms for internship and job interviews.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. C++ In-Depth Series. Pearson Education, Boston, MA, 2005.

[2] Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, E. A. Boyd, Sebastian Ceria, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, R. R. Indovina, Thorsten Koch, Andrea Lodi, Alexander Martin, Cassandra M. McZeal, Hans Mittelmann, Ted Ralphs, Daniel Rehfeldt, Domenico Salvagnin, Martin W. P. Savelsbergh, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010 bibliography. Available online from *Konrad-Zuse-Zentrum für Informationstechnik Berlin: MIPLIB – Mixed Integer Problem Library* at: http://miplib.zib.de/biblio.html; December 9, 2015 was the last accessed date, December 1 2015.

[3] Sarita V. Adve, Doug Burger, Rudolf Eigenmann, Alasdair Rawsthorne, Michael D. Smith, Catherine H. Gebotys, Mahmut T. Kandemir, David J. Lilja, Alok N. Choudbary, Jesse Z. Fang, and Pen-Chung Yew. Changing interaction of compiler and architecture. IEEE Computer, 30(12):51–58, December 1997.

[4] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. C++ In-Depth Series. Addison-Wesley, Indianapolis, IN, 2001.

[5] Alex Allain. Jumping into C++. Cprogramming.com, San Francisco, CA, 2012.

[6] Philip Avery, Abraham Bachrach, Sebastien Barthelemy, Carlos Becker, David Benjamin, Cyrille Berger, Armin Berres, Jose Luis Blanco, Mark Borgerding, Romain Bossart, Kolja Brix, Gauthier Brun, Philipp Büttgenbach, Thomas Capricelli, Nicolas Carre, Jean Ceccato, Vladimir Chalupecky, Benjamin Chrétien, Andrew Coles, Jeff "complexzeros", Marton Danoczy, Jeff Dean, Georg Drenkhahn, Christian Ehrlicher, Martinho Fernandes, Daniel Gomez Ferro, Rohit Garg, Mathieu Gautier, Anton Gladky, Stuart Glaser, Marc Glisse, Frederic Gosselin, Gaël Guennebaud, Philippe Hamelin, Marcus D. Hanwell, David Harmon, Chen-Pang He, Hauke Heibel, Christoph Hertzberg, Pavel Holoborodko, Tim Holy, Intel staff, Trevor Irons, Benoît Jacob, Bram de Jong, Kibeom Kim, Moritz Klammler, Claas Köhler, Alexey Korepanov, Igor Krivenko, Marijn Kruisselbrink, Abhijit Kundu, Moritz Lenz, Bo Li, Sebastian Lipponer, Daniel Lowenberg, David J. Luitz, Naumov Maks, Angelos Mantzaflaris, D. J. Marcin, Konstantinos A. Margaritis, Roger Martin, Ricard Marxer, Vincenzo Di Massa, Christian Mayer, Frank Meier-Dörnberg, Keir Mierle, Laurent Montel, Eamon Nerbonne, Alexander Neundorf, Jason Newton, Jitse Niesen, Desire Nuentsa, Jan Oberländer, Jos van den Oever, Michael Olbrich, Simon Pilgrim, Bjorn Piltz, Benjamin Piwowarski, Zach Ploskey, Giacomo Po, Sergey Popov, Manoj Rajagopalan, Stjepan Rajko, Jure Repinc, Kenneth Frank Riddile, Richard Roberts, Adolfo Rodriguez, Peter Román, Oliver Ruepp, Radu Bogdan Rusu, Guillaume Saupin, Olivier Saut, Benjamin Schindler, Michael Schmidt, Dennis Schridde, Jakob Schwendner, Christian Seiler, Martin Senst, Sameer Sheorey, Andy Somerville,

Alex Stapleton, Benoit Steiner, Sven Strothoff, Leszek Swirski, Adam Szalkowski, Silvio Traversaro, Piotr Trojanek, Anthony Truchet, Adolfo Rodriguez Tsourouksdissian, James Richard Tyrer, Rhys Ulerich, Henry de Valence, Ingmar Vanhassel, Michiel Van Dyck, Scott Wheeler, Freddie Witherden, Urs Wolfer, Manuel Yguel, and Pierre Zoppitelli. Eigen. Available online at: http://eigen.tuxfamily.org/index.php?title=Main_Page; February 15, 2016 was the last accessed date, February 12 2016.

[7] Hyunki Baik, François Bodin, Ross Dickson, Max Domeika, Scott A. Hissam, Skip Hovsmith, James Ivers, Ian Lintault, Stephen Olsen, and David Stewart. Multicore programming practices guide. Technical report, The Multicore Association, El Dorado Hills, CA, 2013.

[8] Brian Bailey, Grant Martin, and Andrew Piziali. ESL Design and Verification: A Prescription for Electronic System-Level Methodology. The Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann, San Francisco, CA, 2007.

[9] Dimitris Bertsimas and Melvyn Sim. The price of robustness. Operations Research, 52(1):35–53, January–February 2004.

[10] Jean-Louis Boulanger. Static Analysis of Software: The Abstract Interpretation. John Wiley & Sons, Hoboken, NJ, 2013.

[11] Dov Bulka and David Mayhew. Efficient C++: Performance Programming Techniques. Addison Wesley Longman, Inc., Indianapolis, IN, 2000.

[12] Marshall Cline. C++ FAQ Lite: Frequently asked questions. Available online from the course web page of *CS210 Data Structures and Abstractions Lab*(Spring 2015), Department of Computer Science, Faculty of Science, University of Regina at: http://www.cs.uregina.ca/Links/class-info/210/C++FAQ/; July 10, 2015 was the last accessed date, July 10 2000.

[13] Marshall Cline. C++ FAQ Lite: Frequently asked questions. Available online from the Computer Science Department, B. Thomas Golisano College of Computing and Information Sciences, Rochester Institute of Technology at: http://www.cs.rit.edu/~mjh/docs/c++-faq/; July 10, 2015 was the last accessed date, May 2 2003.

[14] Marshall Cline. C++ FAQ Lite: Frequently asked questions. Available online from the web page of Laura Mensi and Paolo Copello, *Tiscali Italia S.p.A.: Tiscali Webspace: Fanelia Italy – Computer Programming, Psychiatry, Escaflowne, and much more* at: http://web.tiscali.it/fanelia/cpp-faq-en/; July 10, 2015 was the last accessed date, July 28 2011.

[15] cplusplus.com. The C++ resources network. Available online at: http://www.cplusplus.com/; April 2, 2014 was the last accessed date, 2014.

[16] cplusplus.com. C++ language. Available online from *The C++ Resources Network: Tutorials* at: http://www.cplusplus.com/doc/tutorial/; February 5, 2016 was the last accessed date, 2015.

[17] cplusplus.com. Reference: C++ reference. Available online at: http://www.cplusplus.com/reference/; November 2, 2015 was the last accessed date, 2015.

[18] cplusplus.com. Tutorials. Available online from *The C++ Resources Network* at: http://www.cplusplus.com/doc/; February 5, 2016 was the last accessed date, 2015.

[19] cppreference.com contributors. C++ reference. Available online at: http://en.cppreference.com/w/cpp; September 17, 2015 was the last accessed date, June 16 2015.

[20] Lawrence Crowl. C++ binary Fixed-Point arithmetic. Technical Report N3352, International Organization for Standardization and International Electrotechnical Commission, Genèva, Switzerland, January 15 2012.

[21] Lawrence Crowl. C++ binary Fixed-Point arithmetic. Technical Report P0106R0, International Organization for Standardization and International Electrotechnical Commission, Genèva, Switzerland, September 27 2015.

[22] Mourad Debbabi, Fawzi Hassaïne, Yosr Jarraya, Andrei Soeanu, and Luay Alawneh. Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models. Springer-Verlag Berlin Heidelberg, Heidelberg, Germany, 2010.

[23] Ian G. Harris Dhiraj K. Pradhan. Practical Design Verification. Cambridge University Press, New York, NY, 2009.

[24] Dictionary.com staff. Dictionary.com: Find the meanings and definitions of words at dictionary.com. Available online at: http://dictionary.reference.com/; February 5, 2016 was the last accessed date, 2016.

[25] Jack Dongarra and Eric Grosse. Netlib repository. Available online at: http://www.netlib.org/; February 3, 2016 was the last accessed date, 2016.

[26] Bruce Eckel. Thinking in C++: Introduction to Standard C++, volume 1. Prentice Hall, Upper Saddle River, NJ, second edition, 2000.

[27] Bruce Eckel and Chuck Allison. Thinking in C++: Practical Programming, volume 2. Prentice Hall, Upper Saddle River, NJ, 2003.

[28] EliteHussar. Distinguish between pointers and references in C++. Available online from *cplusplus.com – The C++ Resources Network* at: http://www.cplusplus.com/articles/ENywvCM9/; October 8, 2015 was the last accessed date, August 20 2010.

[29] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools. Morgan Kaufmann, San Francisco, CA, 2005.

[30] Agner Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. Available online at: http://www.agner.org/optimize/optimizing_cpp.pdf; July 1, 2015 was the last accessed date, August 7 2014.

[31] Neal Ford. The Productive Programmer: Theory in Practice. O'Reilly Media, Sebastopol, CA, 2008.

[32] Tony Gaddis. Starting Out With C++: From Control Structures Through Objects. Pearson Education, Boston, MA, sixth (brief) edition, 2010.

[33] Tony Gaddis. Starting Out With C++: From Control Structures Through Objects. Addison-Wesley, Boston, MA, seventh edition, 2012.

[34] Tony Gaddis, Judy Walters, and Godfrey Muganda. Starting Out With C++: Early Objects. Addison-Wesley, Boston, MA, seventh edition, 2011.

[35] Marc Gregoire. Professional C++. John Wiley & Sons, Indianapolis, IN, third edition, 2014.

[36] P. L. Hammer, E. L. Johnson, and B. H. Korte. Conclusive remarks. In Discrete Optimization II, Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium, volume 5 of Annals of Discrete Mathematics, pages 427–453. North-Holland, Banff, Alberta, Canada and Vancouver, B.C., Canada, August 1979.

[37] Steve Heller. C++: A Dialog: Programming with the C++ Standard Library. Pearson Education, Upper Saddle River, NJ, 2003.

[38] Hewlett-Packard Company staff. Standard template library programmer's guide. Available online in *SGI – The Trusted Leader in High Performance Computing: Tech Archive: Standard xTemplate Library Programmer's Guide* at: http://www.sgi.com/tech/stl/; September 30, 2015 was the last accessed date, 1994.

[39] Hewlett-Packard Company staff. STL complexity specifications. Available online in *SGI – The Trusted Leader in High Performance Computing: Tech Archive: Standard Template Library Programmer's Guide: Design documents: STL Complexity Specifications* at: http://www.sgi.com/tech/stl/complexity.html; September 30, 2015 was the last accessed date, http://www.sgi.com/tech/stl/complexity.html 2014.

[40] Cay S. Horstmann. C++ for Everyone. John Wiley & Sons, Hoboken, NJ, second edition, 2012.

[41] Justin Husted. Fixing some common compiler problems. Available online from the *UW ACM's (University of Washington's chapter of the Association for Computing Machinery) web page: Tutorials: [Development] in UNIX* at: http://www.cs.washington.edu/acm/tutorials/dev-in-unix/compiler-problems.html; February 17, 2016 was the last accessed date, May 19 2000.

[42] Innovation 24 staff. LocalSolver 5.5 documentation: Overview. Innovation 24, Paris, France, 2015.

[43] Innovation 24 staff. Localsolver: Mathematical optimization solver. Available online at: http://www.localsolver.com/; December 7, 2015 was the last accessed date, 2015.

[44] Ted Jensen. A tutorial on pointers and arrays in C. Available online as Version 1.2 at: http://pweb.netcom.com/~tjensen/ptr/cpoint.htm, http://pweb.netcom.com/~tjensen/ptr/pointers.htm, and http://home.earthlink.net/~momotuk/pointers.pdf; October 8, 2015 was the last accessed date, September 2003.

[45] Nicolai M. Josuttis. The C++ Standard Library: A Tutorial and Reference. Addison-Wesley, Reading, MA, 1999.

[46] Nicolai M. Josuttis. Josuttis' summary of STL algorithms. Available online at: http://www.josuttis.com/libbook/algolist.pdf; July 1, 2015 was the last accessed date, 1999.

[47] Nicolai M. Josuttis. The C++ Standard Library: A Tutorial and Reference. Pearson Education, Upper Saddle River, NJ, second edition, 2012.

[48] Björn Karlsson. Beyond the C++ Standard Library: An Introduction to Boost. Pearson Education, Boston, MA, 2006.

[49] George Em Karniadakis and Robert M. Kirby II. Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation. Cambridge University Press, Cambridge, U.K., 2003.

[50] Jayantha Katupitiya and Kim Bentley. Interfacing with C++: Programming Real-World Applications. Springer-Verlag Berlin Heidelberg, Heidelberg, Germany, 2006.

[51] Brian W. Kernighan and Rob Pike. The Practice of Programming. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1999.

[52] Darko Kirovski and Miodrag Potkonjak. A quantitative approach to functional debugging. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1997), pages 170–173, San Jose, CA, November 9–13 1997. IEEE Computer Society, IEEE Press.

[53] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010: Mixed integer programming library version 5. Mathematical Programming Computation, 3(2):103–163, June 2011.

[54] Andrew Koenig and Barbara Moo. Accelerated C++: Practical Programming by Example. C++ In-Depth Series. Addison-Wesley, Boston, MA, 2000.

[55] Insup Lee, Joseph Y-T. Leung, and Sang H. Son. Handbook of Real-Time and Embedded Systems. Chapman & Hall/CRC Computer & Information Science. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[56] Jon Lee. A First Course in Combinatorial Optimization, volume 36 of Cambridge Texts in Applied Mathematics. Cambridge University Press, New York, NY, 2004.

[57] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. C++ Primer. Addison-Wesley, Upper Saddle River, NJ, fifth edition, 2013.

[58] Mi Lu. Arithmetic and Logic in Computer Systems. John Wiley & Sons, Hoboken, NJ, 2004.

[59] Kurt McMahon. C strings and C++ strings. Available online from *Kurt McMahon's web page: Notes* at: `https://www.prismnet.com/~mcmahon/Notes/strings.html`; November 3, 2015 was the last accessed date.

[60] Scott Meyers. Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley Professional Computing Series. Pearson Education, Upper Saddle River, NJ, third edition, 2005.

[61] Mohtashim. C++ STL tutorial. Available online at *Tutorials Point: C++ Tutorial: C++ STL Tutorial*: `http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm`; September 17, 2015 was the last accessed date, 2015.

[62] Peter Mortensen. What is the difference between const int*, const int * const, and int const *? Available online from *Stack Exchange Inc.: Stack Overflow: Questions* at: `http://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const`; October 1, 2015 was the last accessed date, March 13 2015.

[63] Arindam Mukherjee. Learning Boost C++ Libraries: Solve Practical Programming Problems Using Powerful, Portable, and Expressive Libraries from Boost. Packt Publishing, Birmingham, West Midlands, England, U.K., July 2015.

[64] Steve Oualline. Practical C++ Programming. Programming Style Guidelines. O'Reilly Media, Sebastopol, CA, second edition, 2003.

[65] Yusuf Kemal Özcan. Is there any difference between pointers and references? Available online from *Stack Exchange Inc.: Programmers Stack Exchange: Questions* at: http://programmers.stackexchange.com/questions/195337/is-there-any-difference-between-pointers-and-references; October 28, 2015 was the last accessed date, April 18 2013.

[66] Rod Pierce. Maths is fun: Maths resources. Available online at: https://www.mathsisfun.com/; February 5, 2016 was the last accessed date, 2016.

[67] Joe Pitt-Francis and Jonathan Whiteley. Guide to Scientific Computing in C++. Undergraduate Topics in Computer Science. Springer-Verlag London, London, U.K., 2012.

[68] Antony Polukhin. Boost C++ Application Development Cookbook: Over 80 practical, task-based recipes to create applications using Boost libraries. Packt Publishing, Birmingham, West Midlands, England, U.K., 2013.

[69] Constantine Pozrikidis. Introduction to C++ Programming and Graphics. Springer Science+Business Media, LCC, New York, NY, 2007.

[70] Stephen Prata. C++ Primer Plus. Sams Publishing, Indianapolis, IN, fifth edition, 2005.

[71] Stephen Prata. C++ Primer Plus: Developer's Library. Pearson Education, Upper Saddle River, NJ, sixth edition, 2012.

[72] Greg Reese. C++ Standard Library Practical Tips. Charles River Media Programming Series. Charles River Media, Hingham, MA, 2006.

[73] Chris Riesbeck. Standard C++ containers. Available online from *Prof. Chris Riesbeck's web page: Programming: Useful C++ / Unix Resources*, Computer Science Division, Department of Electrical Engineering and Computer Science, Robert R. McCormick School of Engineering and Applied Science, Northwestern University at: http://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-summary.html; September 30, 2015 was the last accessed date, July 3 2009.

[74] Chris Riesbeck. Useful C++ / Unix resources. Available online from *Prof. Chris Riesbeck's web page: Programming*, Computer Science Division, Department of Electrical Engineering and Computer Science, Robert R. McCormick School of Engineering and Applied Science, Northwestern University at: http://www.cs.northwestern.edu/~riesbeck/programming/c++/; September 30, 2015 was the last accessed date, July 2 2009.

[75] Robert Robson. Using the STL: The C++ Standard Template Library. Springer-Verlag Berlin Heidelberg New York, Heidelberg, Germany, second edition, 2000.

[76] Philip Romanik and Amy Muntz. Applied C++: Practical Techniques for Building Better Software. C++ In-Depth Series. Addison-Wesley, Boston, MA, 2003.

[77] Dan Saks. An introduction to references. Available online from *UBM Electronics: UBM Canon Electronics Engineering Communities: Embedded – Cracking the Code to Systems Development* at: http://www.embedded.com/print/4024641; October 8, 2015 was the last accessed date, February 26 2001.

[78] Dan Saks. References vs. pointers. Available online from *UBM Electronics: UBM Canon Electronics Engineering Communities: Embedded – Cracking the Code to Systems Development* at: http://www.embedded.com/electronics-blogs/programming-pointers/4023307/References-vs-Pointers and http://www.embedded.com/print/4023307; October 28, 2015 was the last accessed date, March 15 2001.

[79] Walter Savitch. Problem Solving with C++. Pearson Education, Boston, MA, seventh edition, 2009.

[80] Boris Schäling. The Boost C++ Libraries. Self-published, 2012.

[81] Edward Scheinerman. C++ for Mathematicians: An Introduction for Students and Professionals. Chapman & Hall/CRC, Boca Raton, FL, 2006.

[82] Herbert Schildt. C++: The Complete Reference. McGraw-Hill, Berkeley, CA, third edition, 1998.

[83] Herbert Schildt. C++ from the Ground Up. McGraw-Hill/Osborne, Berkeley, CA, third edition, 2003.

[84] Herbert Schildt. C++: The Complete Reference. Osborne Complete Reference Series. McGraw-Hill/Osborne, Berkeley, CA, fourth edition, 2003.

[85] Herbert Schildt. The Art of C++. McGraw-Hill/Osborne, Emeryville, CA, 2004.

[86] Juan Soulié. C++ Language Tutorial. cplusplus.com, June 2007.

[87] Bjarne Stroustrup. Programming: Principles and Practice Using C++. Pearson Education, Boston, MA, 2009.

[88] Bjarne Stroustrup. Programming: Principles and Practice Using C++. Pearson Education, Upper Saddle River, NJ, second edition, 2014.

[89] David Vandevoorde and Nicolai M. Josuttis. C++ Templates: The Complete Guide. Pearson Education, Boston, MA, 2003.

[90] Dirk Vermeir. Multi-Paradigm Programming using C++. Springer-Verlag London Berlin Heidelberg, London, U.K., 2001.

[91] Wikipedia contributors. Algebraic modeling language. Available online in *Wikipedia, The Free Encyclopedia: Specification languages* at: https://en.wikipedia.org/wiki/Algebraic_modeling_language; December 11, 2015 was the last accessed date, May 3 2015.

[92] Wikipedia contributors. Discrete optimization. Available online in *Wikipedia, The Free Encyclopedia: Mathematical optimization* at: https://en.wikipedia.org/wiki/Discrete_optimization; December 9, 2015 was the last accessed date, May 17 2015.

[93] Matthew Wilson. Extended STL: Collections and Iterators, volume 1. Addison-Wesley, Upper Saddle River, NJ, 2007.

[94] Wolfram Research staff. Wolfram MathWorld: The web's most extensive mathematics resource. Available online at: http://mathworld.wolfram.com/; February 5, 2016 was the last accessed date, January 29 2016.