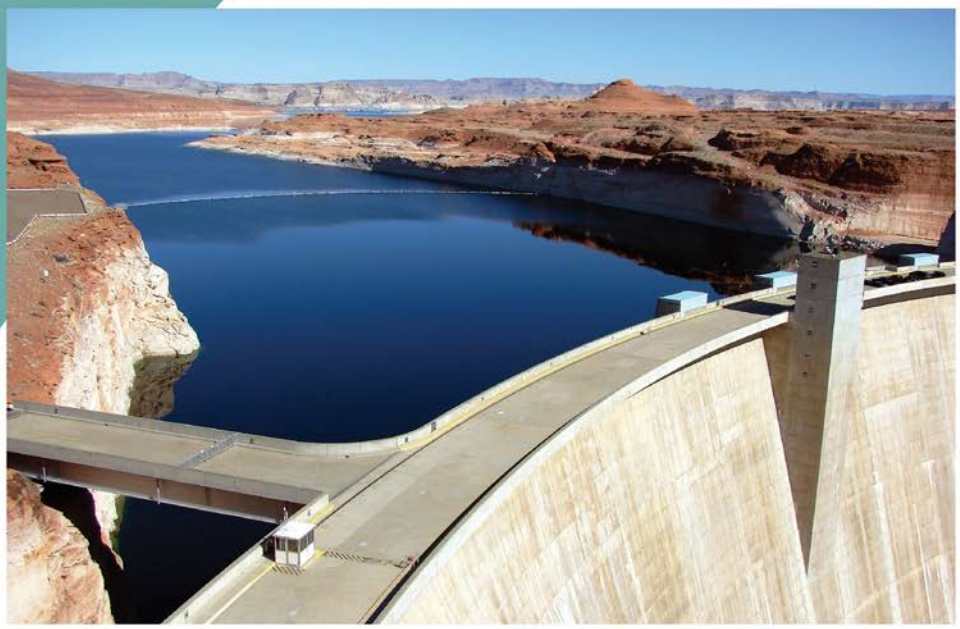
Large-Scale C++

Volume I

Process and Architecture

John Lakos



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Large-Scale C++

Large-Scale C++ Volume I: Process and Architecture

Table of Contents

Cover
Half Title
Series Page
Title Page
Copyright Page
Dedication
Contents
Preface
Acknowledgments
Chapter 0: Motivation
- 0.1 The Goal: Faster, Better, Cheaper!
- 0.2 Application vs. Library Software
- 0.3 Collaborative vs. Reusable Software
- 0.4 Hierarchically Reusable Software
- 0.5 Malleable vs. Stable Software
- 0.6 The Key Role of Physical Design
- 0.7 Physically Uniform Software: The Component
- 0.8 Quantifying Hierarchical Reuse: An Analogy
- 0.9 Software Capital
- 0.10 Growing the Investment
- 0.11 The Need for Vigilance

Table of Contents

0.12 Summary

Chapter 1: Compilers, Linkers, and Components

1.1 Knowledge Is Power: The Devil Is in the Details

1.1.1 Hello World!

1.1.2 Creating C++ Programs

1.1.3 The Role of Header Files

1.2 Compiling and Linking C++

1.2.1 The Build Process: Using Compilers and Linkers

1.2.2 Classical Atomicity of Object (.o) Files

1.2.3 Sections and Weak Symbols in .o Files

1.2.4 Library Archives

1.2.5 The Singleton Registry Example

1.2.6 Library Dependencies

1.2.7 Link Order and Build-Time Behavior

1.2.8 Link Order and Runtime Behavior

1.2.9 Shared (Dynamically Linked) Libraries

1.3 Declarations, Definitions, and Linkage

1.3.1 Declaration vs. Definition

1.3.2 (Logical) Linkage vs. (Physical) Linking

1.3.3 The Need for Understanding Linking Tools

1.3.4 Alternate Definition of Physical Linkage: Bindage

1.3.5 More on How Linkers Work

1.3.6 A Tour of Entities Requiring Program-Wide Unique Addresses

1.3.7 Constructs Where the Callers Compiler Needs the Definitions Source Code

1.3.8 Not All Declarations Require a Definition to Be Useful

1.3.9 The Clients Compiler Typically Needs to See Class Definitions

1.3.10 Other Entities Where Users Compilers Must See the Definition

1.3.11 Enumerations Have External Linkage, but So What?!

Table of Contents

1.3.12 Inline Functions Are a Somewhat Special Case

1.3.13 Function and Class Templates

1.3.14 Function Templates and Explicit Specializations

1.3.15 Class Templates and Their Partial Specializations

1.3.16 extern Templates

1.3.17 Understanding the ODR (and Bindage) in Terms of Tools

1.3.18 Namespaces

1.3.19 Explanation of the Default Linkage of const Entities

1.3.20 Summary of Declarations, Definitions, Linkage, and Bindage

1.4 Header Files

1.5 Include Directives and Include Guards

1.5.1 Include Directives

1.5.2 Internal Include Guards

1.5.3 (Deprecated) External Include Guards

1.6 From .h /.cpp Pairs to Components

1.6.1 Component Property 1

1.6.2 Component Property 2

1.6.3 Component Property 3

1.7 Notation and Terminology

1.7.1 Overview

1.7.2 The Is-A Logical Relationship

1.7.3 The Uses-In-The-Interface Logical Relationship

1.7.4 The Uses-In-The-Implementation Logical Relationship

1.7.5 The Uses-In-Name-Only Logical Relationship and the Protocol Class

1.7.6 In-Structure-Only (ISO) Collaborative Logical Relationships

1.7.7 How Constrained Templates and Interface Inheritance Are Similar

1.7.8 How Constrained Templates and Interface Inheritance Differ

1.7.8.1 Constrained Templates, but Not Interface Inheritance

1.7.8.2 Interface Inheritance, but Not Constrained Templates

Table of Contents

1.7.9 All Three Inheriting Relationships Add Unique Value

1.7.10 Documenting Type Constraints for Templates

1.7.11 Summary of Notation and Terminology

1.8 The Depends-On Relation

1.9 Implied Dependency

1.10 Level Numbers

1.11 Extracting Actual Dependencies

1.11.1 Component Property 4

1.12 Summary

Chapter 2: Packaging and Design Rules

2.1 The Big Picture

2.2 Physical Aggregation

2.2.1 General Definition of Physical Aggregate

2.2.2 Small End of Physical-Aggregation Spectrum

2.2.3 Large End of Physical-Aggregation Spectrum

2.2.4 Conceptual Atomicity of Aggregates

2.2.5 Generalized Definition of Dependencies for Aggregates

2.2.6 Architectural Significance

2.2.7 Architectural Significance for General UORs

2.2.8 Parts of a UOR That Are Architecturally Significant

2.2.9 What Parts of a UOR Are Not Architecturally Significant?

2.2.10 A Component Is Naturally Architecturally Significant

2.2.11 Does a Component Really Have to Be a .h /.cpp Pair?

2.2.12 When, If Ever, Is a .h /.cpp Pair Not Good Enough?

2.2.13 Partitioning a .cpp File Is an Organizational-Only Change

2.2.14 Entity Manifest and Allowed Dependencies

2.2.15 Need for Expressing Envelope of Allowed Dependencies

2.2.16 Need for Balance in Physical Hierarchy

Table of Contents

2.2.17 Not Just Hierarchy, but Also Balance

2.2.18 Having More Than Three Levels of Physical Aggregation Is Too Many

2.2.19 Three Levels Are Enough Even for Larger Systems

2.2.20 UORs Always Have Two or Three Levels of Physical Aggregation

2.2.21 Three Balanced Levels of Aggregation Are Sufficient. Trust Me!

2.2.22 There Should Be Nothing Architecturally Significant Larger Than a UOR

2.2.23 Architecturally Significant Names Must Be Unique

2.2.24 No Cyclic Physical Dependencies!

2.2.25 Section Summary

2.3 Logical/Physical Coherence

2.4 Logical and Physical Name Cohesion

2.4.1 History of Addressing Namespace Pollution

2.4.2 Unique Naming Is Required; Cohesive Naming Is Good for Humans

2.4.3 Absurd Extreme of Neither Cohesive nor Mnemonic Naming

2.4.4 Things to Make Cohesive

2.4.5 Past/Current Definition of Package

2.4.6 The Point of Use Should Be Sufficient to Identify Location

2.4.7 Proprietary Software Requires an Enterprise Namespace

2.4.8 Logical Constructs Should Be Nominally Anchored to Their Component

2.4.9 Only Classes, structs, and Free Operators at Package-Namespace Scope

2.4.10 Package Prefixes Are Not Just Style

2.4.11 Package Prefixes Are How We Name Package Groups

2.4.12 using Directives and Declarations Are Generally a BAD IDEA

2.4.13 Section Summary

2.5 Component Source-Code Organization

2.6 Component Design Rules

2.7 Component-Private Classes and Subordinate Components

2.7.1 Component-Private Classes

2.7.2 There Are Several Competing Implementation Alternatives

Table of Contents

2.7.3 Conventional Use of Underscore

2.7.4 Classic Example of Using Component-Private Classes

2.7.5 Subordinate Components

2.7.6 Section Summary

2.8 The Package

2.8.1 Using Packages to Factor Subsystems

2.8.2 Cycles Among Packages Are BAD

2.8.3 Placement, Scope, and Scale Are an Important First Consideration

2.8.4 The Inestimable Communicative Value of (Unique) Package Prefixes

2.8.5 Section Summary

2.9 The Package Group

2.9.1 The Third Level of Physical Aggregation

2.9.2 Organizing Package Groups During Deployment

2.9.3 How Do We Use Package Groups in Practice?

2.9.4 Decentralized (Autonomous) Package Creation

2.9.5 Section Summary

2.10 Naming Packages and Package Groups

2.10.1 Intuitively Descriptive Package Names Are Overrated

2.10.2 Package-Group Names

2.10.3 Package Names

2.10.4 Section Summary

2.11 Subpackages

2.12 Legacy, Open-Source, and Third-Party Software

2.13 Applications

2.14 The Hierarchical Testability Requirement

2.14.1 Leveraging Our Methodology for Fine-Grained Unit Testing

2.14.2 Plan for This Section (Plus Plug for Volume II and Especially Volume III)

2.14.3 Testing Hierarchically Needs to Be Possible

2.14.4 Relative Import of Local Component Dependencies with Respect to

Table of Contents

Testing
2.14.5 Allowed Test-Driver Dependencies Across Packages
2.14.6 Minimize Test-Driver Dependencies on the External Environment
2.14.7 Insist on a Uniform (Standalone) Test-Driver Invocation Interface
2.14.8 Section Summary
2.15 From Development to Deployment
2.15.1 The Flexible Deployment of Software Should Not Be Compromised
2.15.2 Having Unique .h and .o Names Are Key
2.15.3 Software Organization Will Vary During Development
2.15.4 Enterprise-Wide Unique Names Facilitate Refactoring
2.15.5 Software Organization May Vary During Just the Build Process
2.15.6 Flexibility in Deployment Is Needed Even Under Normal Circumstances
2.15.7 Flexibility Is Also Important to Make Custom Deployments Possible
2.15.8 Flexibility in Stylistic Rendering Within Header Files
2.15.9 How Libraries Are Deployed Is Never Architecturally Significant
2.15.10 Partitioning Deployed Software for Engineering Reasons
2.15.11 Partitioning Deployed Software for Business Reasons
2.15.12 Section Summary
2.16 Metadata
2.16.1 Metadata Is By Decree
2.16.2 Types of Metadata
2.16.2.1 Dependency Metadata
2.16.2.2 Build Requirements Metadata
2.16.2.3 Membership Metadata
2.16.2.4 Enterprise-Specific Policy Metadata
2.16.3 Metadata Rendering
2.16.4 Metadata Summary
2.17 Summary
Chapter 3: Physical Design and Factoring

Table of Contents

3.1 Thinking Physically

3.1.1 Pure Classical (Logical) Software Design Is Naive

3.1.2 Components Serve as Our Fine-Grained Modules

3.1.3 The Software Design Space Has Direction

3.1.3.1 Example of Relative Physical Position: Abstract Interfaces

3.1.4 Software Has Absolute Location

3.1.4.1 Asking the Right Questions Helps Us Determine Optimal Location

3.1.4.2 See What Exists to Avoid Reinventing the Wheel

3.1.4.3 Good Citizenship: Identifying Proper Physical Location

3.1.5 The Criteria for Colocation Should Be Substantial, Not Superficial

3.1.6 Discovery of Nonprimitive Functionality Absent Regularity Is Problematic

3.1.7 Package Scope Is an Important Design Consideration

3.1.7.1 Package Charter Must Be Delineated in Package-Level Documentation

3.1.7.2 Package Prefixes Are at Best Mnemonic Tags, Not Descriptive Names

3.1.7.3 Package Prefixes Force Us to Consider Design More Globally Early

3.1.7.4 Package Prefixes Force Us to Consider Package Dependencies from the Start

3.1.7.5 Even Opaque Package Prefixes Grow to Take On Important Meaning

3.1.7.6 Effective (e.g., Associative) Use of Package Names Within Groups

3.1.8 Limitations Due to Prohibition on Cyclic Physical Dependencies

3.1.9 Constraints on Friendship Intentionally Preclude Some Logical Designs

3.1.10 Introducing an Example That Justifiably Requires Wrapping

3.1.10.1 Wrapping Just the Time Series and Its Iterator in a Single Component

3.1.10.2 Private Access Within a Single Component Is an Implementation Detail

3.1.10.3 An Iterator Helps to Realize the Open-Closed Principle

3.1.10.4 Private Access Within a Wrapper Component Is Typically Essential

3.1.10.5 Since This Is Just a Single-Component Wrapper, We Have Several Options

3.1.10.6 Multicomponent Wrappers, Not Having Private Access, Are Problematic

3.1.10.7 Example Why Multicomponent Wrappers Typically Need Special Access

3.1.10.8 Wrapping Interoperating Components Separately Generally Doesnt Work

3.1.10.9 What Should We Do When Faced with a Multicomponent Wrapper?

3.1.11 Section Summary

3.2 Avoiding Poor Physical Modularity

Table of Contents

3.2.1 There Are Many Poor Modularization Criteria; Syntax Is One of Them

3.2.2 Factoring Out Generally Useful Software into Libraries Is Critical

3.2.3 Failing to Maintain Application/Library Modularity Due to Pressure

3.2.4 Continuous Demotion of Reusable Components Is Essential

3.2.4.1 Otherwise, in Time, Our Software Might Devolve into a Big Ball of Mud!

3.2.5 Physical Dependency Is Not an Implementation Detail to an App Developer

3.2.6 Iterators Can Help Reduce What Would Otherwise Be Primitive Functionality

3.2.7 Not Just Minimal, Primitive: The Utility struct

3.2.8 Concluding Example: An Encapsulating Polygon Interface

3.2.8.1 What Other UDTs Are Used in the Interface?

3.2.8.2 What Invariants Should our::Polygon Impose?

3.2.8.3 What Are the Important Use Cases?

3.2.8.4 What Are the Specific Requirements?

3.2.8.5 Which Required Behaviors Are Primitive and Which Arent?

3.2.8.6 Weighing the Implementation Alternatives

3.2.8.7 Achieving Two Out of Three Aint Bad

3.2.8.8 Primitiveness vs. Flexibility of Implementation

3.2.8.9 Flexibility of Implementation Extends Primitive Functionality

3.2.8.10 Primitiveness Is Not a Draconian Requirement

3.2.8.11 What About Familiar Functionality Such as Perimeter and Area?

3.2.8.12 Providing Iterator Support for Generic Algorithms

3.2.8.13 Focus on Generally Useful Primitive Functionality

3.2.8.14 Suppress Any Urge to Colocate Nonprimitive Functionality

3.2.8.15 Supporting Unusual Functionality

3.2.9 Semantics vs. Syntax as Modularization Criteria

3.2.9.1 Poor Use of uas a Package Suffix

3.2.9.2 Good Use of util as a Component Suffix

3.2.10 Section Summary

3.3 Grouping Things Physically That Belong Together Logically

3.3.1 Four Explicit Criteria for Class Colocation

Table of Contents

3.3.1.1 First Reason: Friendship

3.3.1.2 Second Reason: Cyclic Dependency

3.3.1.3 Third Reason: Single Solution

3.3.1.4 Fourth Reason: Flea on an Elephant

3.3.2 Colocation Beyond Components

3.3.3 When to Make Helper Classes Private to a Component

3.3.4 Colocation of Template Specializations

3.3.5 Use of Subordinate Components

3.3.6 Colocate Tight Mutual Collaboration within a Single UOR

3.3.7 Day-Count Example

3.3.8 Final Example: Single-Threaded Reference-Counted Functors

3.3.8.1 Brief Review of Event-Driven Programming

3.3.8.2 Aggregating Components into Packages

3.3.8.3 The Final Result

3.3.9 Section Summary

3.4 Avoiding Cyclic Link-Time Dependencies

3.5 Levelization Techniques

3.5.1 Classic Levelization

3.5.2 Escalation

3.5.3 Demotion

3.5.4 Opaque Pointers

3.5.4.1 Manager/Employee Example

3.5.4.2 Event/EventQueue Example

3.5.4.3 Graph/Node/Edge Example

3.5.5 Dumb Data

3.5.6 Redundancy

3.5.7 Callbacks

3.5.7.1 Data Callbacks

3.5.7.2 Function Callbacks

3.5.7.3 Functor Callbacks

3.5.7.4 Protocol Callbacks

Table of Contents

3.5.7.5 Concept Callbacks

3.5.8 Manager Class

3.5.9 Factoring

3.5.10 Escalating Encapsulation

3.5.10.1 A More General Solution to Our Graph Subsystem

3.5.10.2 Encapsulating the Use of Implementation Components

3.5.10.3 Single-Component Wrapper

3.5.10.4 Overhead Due to Wrapping

3.5.10.5 Realizing Multicomponent Wrappers

3.5.10.6 Applying This New, Heretical Technique to Our Graph Example

3.5.10.7 Why Use This Magic reinterpret_cast Technique?

3.5.10.8 Wrapping a Package-Sized System

3.5.10.9 Benefits of This Multicomponent-Wrapper Technique

3.5.10.10 Misuse of This Escalating-Encapsulation Technique

3.5.10.11 Simulating a Highly Restricted Form of Package-Wide Friendship

3.5.11 Section Summary

3.6 Avoiding Excessive Link-Time Dependencies

3.6.1 An Initially Well-Factored Date Class That Degrades Over Time

3.6.2 Adding Business-Day Functionality to a Date Class (BAD IDEA)

3.6.3 Providing a Physically Monolithic Platform Adapter (BAD IDEA)

3.6.4 Section Summary

3.7 Lateral vs. Layered Architectures

3.7.1 Yet Another Analogy to the Construction Industry

3.7.2 (Classical) Layered Architectures

3.7.3 Improving Purely Compositional Designs

3.7.4 Minimizing Cumulative Component Dependency (CCD)

3.7.4.1 Cumulative Component Dependency (CCD) Defined

3.7.4.2 Cumulative Component Dependency: A Concrete Example

3.7.5 Inheritance-Based Lateral Architectures

3.7.6 Testing Lateral vs. Layered Architectures

3.7.7 Section Summary

Table of Contents

3.8 Avoiding Inappropriate Link-Time Dependencies

3.8.1 Inappropriate Physical Dependencies

3.8.2 Betting on a Single Technology (BAD IDEA)

3.8.3 Section Summary

3.9 Ensuring Physical Interoperability

3.9.1 Impeding Hierarchical Reuse Is a BAD IDEA

3.9.2 Domain-Specific Use of Conditional Compilation Is a BAD IDEA

3.9.3 Application-Specific Dependencies in Library Components Is a BAD IDEA

3.9.4 Constraining Side-by-Side Reuse Is a BAD IDEA

3.9.5 Guarding Against Deliberate Misuse Is Not a Goal

3.9.6 Usurping Global Resources from a Library Component Is a BAD IDEA

3.9.7 Hiding Header Files to Achieve Logical Encapsulation Is a BAD IDEA

3.9.8 Depending on Nonportable Software in Reusable Libraries Is a BAD IDEA

3.9.9 Hiding Potentially Reusable Software Is a BAD IDEA

3.9.10 Section Summary

3.10 Avoiding Unnecessary Compile-Time Dependencies

3.10.1 Encapsulation Does Not Preclude Compile-Time Coupling

3.10.2 Shared Enumerations and Compile-Time Coupling

3.10.3 Compile-Time Coupling in C++ Is Far More Pervasive Than in C

3.10.4 Avoiding Unnecessary Compile-Time Coupling

3.10.5 Real-World Example of Benefits of Avoiding Compile-Time Coupling

3.10.6 Section Summary

3.11 Architectural Insulation Techniques

3.11.1 Formal Definitions of Encapsulation vs. Insulation

3.11.2 Illustrating Encapsulation vs. Insulation in Terms of Components

3.11.3 Total vs. Partial Insulation

3.11.4 Architecturally Significant Total-Insulation Techniques

3.11.5 The Pure Abstract Interface (Protocol) Class

3.11.5.1 Extracting a Protocol

Table of Contents

3.11.5.2 Equivalent Bridge Pattern

3.11.5.3 Effectiveness of Protocols as Insulators

3.11.5.4 Implementation-Specific Interfaces

3.11.5.5 Static Link-Time Dependencies

3.11.5.6 Runtime Overhead for Total Insulation

3.11.6 The Fully Insulating Concrete Wrapper Component

3.11.6.1 Poor Candidates for Insulating Wrappers

3.11.7 The Procedural Interface

3.11.7.1 What Is a Procedural Interface?

3.11.7.2 When Is a Procedural Interface Indicated?

3.11.7.3 Essential Properties and Architecture of a Procedural Interface

3.11.7.4 Physical Separation of PI Functions from Underlying C++ Components

3.11.7.5 Mutual Independence of PI Functions

3.11.7.6 Absence of Physical Dependencies Within the PI Layer

3.11.7.7 Absence of Supplemental Functionality in the PI Layer

3.11.7.8 1-1 Mapping from PI Components to Lower-Level Components (Using the z_ Prefix)

3.11.7.9 Example: Simple (Concrete) Value Type

3.11.7.10 Regularity/Predictability of PI Names

3.11.7.11 PI Functions Callable from C++ as Well as C

3.11.7.12 Actual Underlying C++ Types Exposed Opaquely for C++ Clients

3.11.7.13 Summary of Essential Properties of the PI Layer

3.11.7.14 Procedural Interfaces and Return-by-Value

3.11.7.15 Procedural Interfaces and Inheritance

3.11.7.16 Procedural Interfaces and Templates

3.11.7.17 Mitigating Procedural-Interface Costs

3.11.7.18 Procedural Interfaces and Exceptions

3.11.8 Insulation and DLLs

3.11.9 Service-Oriented Architectures

3.11.10 Section Summary

3.12 Designing with Components

3.12.1 The Requirements as Originally Stated

3.12.2 The Actual (Extrapolated) Requirements

Table of Contents

3.12.3 Representing a Date Value in Terms of a C++ Type

3.12.4 Determining What Date V alue Today Is

3.12.5 Determining If a Date Value Is a Business Day

3.12.5.1 Calendar Requirements

3.12.5.2 Multiple Locale Lookups

3.12.5.3 Calendar Cache

3.12.5.4 Application-Level Use of Calendar Library

3.12.6 Parsing and Formatting Functionality

3.12.7 Transmitting and Persisting Values

3.12.8 Day-Count Conventions

3.12.9 Date Math

3.12.9.1 Auxiliary Date-Math Types

3.12.10 Date and Calendar Utilities

3.12.11 Fleshing Out a Fully Factored Implementation

3.12.11.1 Implementing a Hierarchically Reusable Date Class

3.12.11.2 Representing Value in the Date Class

3.12.11.3 Implementing a Hierarchically Reusable Calendar Class

3.12.11.4 Implementing a Hierarchically Reusable PackedCalendar Class

3.12.11.5 Distribution Across Existing Aggregates

3.12.12 Section Summary

3.13 Summary

Conclusion

Appendix: Quick Reference

Bibliography

Index