

CODE COMPLETE, 2D ED, REALLY DETAILED CONTENTS**FRONT MATTER**

Preface [Preface]

Who Should Read This Book?

Experienced Programmers

Self-Taught Programmers

Students

Where Else Can You Find This Information?

Key Benefits of This Handbook

Complete software-construction reference

Ready-to-use checklists

State-of-the-art information

Larger perspective on software development

Absence of hype

Concepts applicable to most common languages

Numerous code examples

Access to other sources of information

Why This Handbook Was Written

The Topic of Construction Has Been Neglected

Construction Is Important

No Comparable Book Is Available

Book Website

Author Note

Notes about the Second Edition [new]

Why did you write a second edition? Weren't the principles in the first edition supposed to be timeless?

Does the second edition discuss object-oriented programming?

What about extreme programming and agile development? Do you talk about those approaches?

What size project will benefit from Code Complete, Second Edition?

Have there been any improvements in programming in the past 10 years?

Has anything moved backwards?

Which of the first edition's ideas are you most protective of?

Will there be a third edition 10 years from now?

Acknowledgments [n/a]

Acknowledgments for the Second Edition [New]

Acknowledgments for the First Edition

LAYING THE FOUNDATION

Welcome to Software Construction [1]

1.1 What Is Software Construction?

1.2 Why Is Software Construction Important?

Construction is a large part of software development

Construction is the central activity in software development

With a focus on construction, the individual programmer's productivity can improve enormously

Construction's product, the source code, is often the only accurate description of the software

Construction is the only activity that's guaranteed to be done

1.3 How to Read This Book

Metaphors for a Richer Understanding of Software Development [2]

2.1 The Importance of Metaphors

2.2 How to Use Software Metaphors

2.3 Common Software Metaphors

Software Penmanship: Writing Code

Software Farming: Growing a System

Software Oyster Farming: System Accretion

Software Construction: Building Software

Applying Software Techniques: The Intellectual Toolbox

Combining Metaphors

Measure Twice, Cut Once: Upstream Prerequisites [3]

3.1 Importance of Prerequisites

Do Prerequisites Apply to Modern Software Projects?

Causes of Incomplete Preparation

Utterly Compelling and Foolproof Argument for Doing Prerequisites Before Construction

Appeal to Logic

Appeal to Analogy

Appeal to Data

Boss-Readiness Test

3.2 Determine the Kind of Software You're Working On

3.3 Problem-Definition Prerequisite

3.4 Requirements Prerequisite

Why Have Official Requirements?

The Myth of Stable Requirements

Handling Requirements Changes During Construction

Use the requirements checklist at the end of the section to assess the quality of your requirements

Make sure everyone knows the cost of requirements changes

Set up a change-control procedure

Use development approaches that accommodate changes

Dump the project

3.5 Architecture Prerequisite

Typical Architectural Components

Program Organization

Major Classes

Data Design

Business Rules

Security

Performance

Scalability

Interoperability

Internationalization/Localization

Error Processing

Fault Tolerance

Architectural Feasibility

Overengineering

Buy-vs.-Build Decisions

- Reuse Decisions
- Change Strategy
- General Architectural Quality
- 3.6 Amount of Time to Spend on Upstream Prerequisites
 - Requirements
 - Software Architecture
 - General Software Development Approaches

Key Construction Decisions [3+new material]

- 4.1 Choice of Programming Language
 - Language Descriptions
 - Ada
 - Assembly Language
 - C
 - C++
 - Cobol
 - Fortran
 - Java
 - JavaScript
 - Perl
 - PHP
 - Python
 - SQL
 - Visual Basic
 - Language-Selection Quick Reference
- 4.2 Programming Conventions
- 4.3 Your Location on the Technology Wave
- 4.4 Selection of Major Construction Practices

CREATING HIGH QUALITY CODE

Design in Construction [mostly new material, some from 7]

- 5.1 Design Challenges
 - Design is a Wicked Problem
 - Design is a Sloppy Process
 - Design is About Trade-Offs and Priorities
 - Design Involves Restrictions
 - Design is Non-Deterministic
 - Design is a Heuristic Process
 - Design is Emergent
- 5.2 Key Design Concepts
 - Software's Primary Technical Imperative: Managing Complexity
 - Accidental and Essential Difficulties
 - Importance of Managing Complexity
 - How to Attack Complexity
 - Desirable Characteristics of a Design
 - Minimal complexity
 - Ease of maintenance
 - Minimal connectedness
 - Extensibility
 - Reusability
 - High fan-in
 - Low-to-medium fan-out
 - Portability

- Leanness
- Stratification
- Standard techniques
- Levels of Design
 - Level 1: Software System
 - Level 2: Division into Subsystems or Packages
 - Common Subsystems
 - Business logic
 - User interface
 - Database access
 - System dependencies
 - Level 3: Division into Classes
 - Classes vs. Objects
 - Level 4: Division into Routines
 - Level 5: Internal Routine Design
- 5.3 Design Building Blocks: Heuristics
 - Find Real-World Objects
 - Identify the objects and their attributes
 - Determine what can be done to each object
 - Determine what each object can do to other objects
 - Determine the parts of each object that will be visible to other objects
 - Define each object's interface
 - Form Consistent Abstractions
 - Encapsulate Implementation Details
 - Inherit When Inheritance Simplifies the Design
 - Hide Secrets (Information Hiding)
 - Secrets and the Right to Privacy
 - An Example of Information Hiding
 - Two Categories of Secrets
 - Barriers to Information Hiding
 - Excessive Distribution Of Information
 - Circular Dependencies
 - Class Data Mistaken For Global Data
 - Perceived Performance Penalties
 - Value of Information Hiding
 - Identify Areas Likely to Change
 - Business logic
 - Hardware dependencies
 - Input and output
 - Nonstandard language features
 - Difficult design and construction areas
 - Status variables
 - Data-size constraints
 - Anticipating Different Degrees of Change
 - Keep Coupling Loose
 - Coupling Criteria
 - Size
 - Visibility
 - Flexibility
 - Kinds of Coupling
 - Simple-data-parameter coupling
 - Simple-object coupling
 - Object-parameter coupling
 - Semantic coupling
 - Look for Common Design Patterns
 - Patterns reduce complexity by providing ready-made abstractions

- Patterns reduce errors by institutionalizing details of common solutions
- Patterns provide heuristic value by suggesting design alternatives
- Patterns streamline communication by moving the design dialog to a higher level

Other Heuristics

- Aim for Strong Cohesion
- Build Hierarchies
- Formalize Class Contracts
- Assign Responsibilities
- Design for Test
- Avoid Failure
- Choose Binding Time Consciously
- Make Central Points of Control
- Consider Using Brute Force
- Draw a Diagram
- Keep Your Design Modular

- Summary of Design Heuristics
- Guidelines for Using Heuristics

5.4 Design Practices

- Iterate
- Divide and Conquer
- Top-Down and Bottom-Up Design Approaches
 - Argument for Top Down
 - Argument for Bottom Up
 - No Argument, Really
- Experimental Prototyping
- Collaborative Design
- How Much Design is Enough?
- Capturing Your Design Work
 - Insert design documentation into the code itself
 - Capture design discussions and decisions on a Wiki
 - Write email summaries
 - Use a digital camera
 - Save design flipcharts
 - Use CRC cards
 - Create UML diagrams at appropriate levels of detail

5.5 Comments on Popular Methodologies

- Software Design, General
- Software Design Theory
- Design Patterns
- Design in General
- Standards

Working Classes [mostly new material, a little from 6]

6.1 Class Foundations: Abstract Data Types (ADTs)

- Example of the Need for an ADT

- Benefits of Using ADTs

- You can hide implementation details
- Changes don't affect the whole program
- You can make the interface more informative
- It's easier to improve performance
- The program is more obviously correct
- The program becomes more self-documenting
- You don't have to pass data all over your program
- You're able to work with real-world entities rather than with low-level implementation structures

- More Examples of ADTs
 - Build or use typical low-level data types as ADTs, not as low-level data types
 - Treat common objects such as files as ADTs
 - Treat even simple items as ADTs
 - Refer to an ADT independently of the medium it's stored on
- Handling Multiple Instances of Data with ADTs in Non-OO Environments
 - Option 1: Use implicit instances (with great care)
 - Option 2: Explicitly identify instances each time you use ADT services
 - Option 3: Explicitly provide the data used by the ADT services
- ADTs and Classes
- 6.2 Good Class Interfaces
 - Good Abstraction
 - Present a consistent level of abstraction in the class interface
 - Be sure you understand what abstraction the class is implementing
 - Provide services in pairs with their opposites
 - Move unrelated information to another class
 - Beware of erosion of the interface's abstraction under modification
 - Don't add public members that are inconsistent with the interface abstraction
 - Consider abstraction and cohesion together
 - Good Encapsulation
 - Minimize accessibility of classes and members
 - Don't expose member data in public
 - Don't put private implementation details in a class's interface
 - Don't make assumptions about the class's users
 - Avoid friend classes
 - Don't put a routine into the public interface just because it uses only public routines
 - Favor read-time convenience to write-time convenience
 - Be very, very wary of *semantic* violations of encapsulation
 - Watch for coupling that's too tight
- 6.3 Design and Implementation Issues
 - Containment ("has a" relationships)
 - Implement "has a" through containment
 - Implement "has a" through private inheritance as a last resort
 - Be critical of classes that contain more than about seven members
 - Inheritance ("is a" relationships)
 - Implement "is a" through public inheritance
 - Design and document for inheritance or prohibit it
 - Adhere to the Liskov Substitution Principle
 - Be sure to inherit only what you want to inherit
 - Don't "override" a non-overrideable member function
 - Move common interfaces, data, and behavior as high as possible in the inheritance tree
 - Be suspicious of classes of which there is only one instance
 - Be suspicious of base classes of which there is only one derived class
 - Be suspicious of classes that override a routine and do nothing inside the derived routine
 - Avoid deep inheritance trees
 - Prefer inheritance to extensive type checking
 - Avoid using a base class's protected data in a derived class (or make that data private instead of protected in the first place)
 - Multiple Inheritance
 - Why Are There So Many Rules for Inheritance?
 - Member Functions and Data
 - Keep the number of routines in a class as small as possible
 - Disallow implicitly generated member functions and operators you don't want
 - Minimize direct routine calls to other classes
 - Minimize indirect routine calls to other classes

- In general, minimize the extent to which a class collaborates with other classes
- Constructors
 - Initialize all member data in all constructors, if possible
 - Initialize data members in the order in which they're declared
 - Enforce the singleton property by using a private constructor
 - Enforce the singleton property by using all static member data and reference counting
 - Prefer deep copies to shallow copies until proven otherwise
- 6.4 Reasons to Create a Class
 - Model real-world objects
 - Model abstract objects
 - Reduce complexity
 - Isolate complexity
 - Hide implementation details
 - Limit effects of changes
 - Hide global data
 - Streamline parameter passing
 - Make central points of control
 - Facilitate reusable code
 - Plan for a family of programs
 - Package related operations
 - To accomplish a specific refactoring
- Classes to Avoid
 - Avoid creating god classes
 - Eliminate irrelevant classes
 - Avoid classes named after verbs
- Summary of Reasons to Create a Class
- 6.5 Language-Specific Issues
- 6.6 Beyond Classes: Packages
 - Classes in General
 - C++
 - Java
 - Visual Basic
- High-Quality Routines [5]
 - 7.1 Valid Reasons to Create a Routine
 - Reduce complexity
 - Make a section of code readable
 - Avoid duplicate code
 - Hide sequences
 - Hide pointer operations
 - Improve portability
 - Simplify complicated boolean tests
 - Improve performance
 - To ensure all routines are small?
 - Operations That Seem Too Simple to Put Into Routines
 - Summary of Reasons to Create a Routine
 - 7.2 Design at the Routine Level
 - 7.3 Good Routine Names
 - Describe everything the routine does
 - Avoid meaningless or wishy-washy verbs
 - Make names of routines as long as necessary
 - To name a function, use a description of the return value
 - To name a procedure, use a strong verb followed by an object
 - Use opposites precisely

- Establish conventions for common operations
- 7.4 How Long Can a Routine Be?
- 7.5 How to Use Routine Parameters
 - Put parameters in input-modify-output order
 - Create your own *in* and *out* keywords
 - If several routines use similar parameters, put the similar parameters in a consistent order
 - Use all the parameters
 - Put status or error variables last
 - Don't use routine parameters as working variables
 - Document interface assumptions about parameters
 - Limit the number of a routine's parameters to about seven
 - Consider an input, modify, and output naming convention for parameters
 - Pass the variables or objects that the routine needs to maintain its interface abstraction
 - Used named parameters
 - Don't assume anything about the parameter-passing mechanism
 - Make sure actual parameters match formal parameters
- 7.6 Special Considerations in the Use of Functions
 - When to Use a Function and When to Use a Procedure
 - Setting the Function's Return Value
 - Check all possible return paths
 - Don't return references or pointers to local data
- 7.7 Macro Routines and Inline Routines
 - Fully parenthesize macro expressions
 - Surround multiple-statement macros with curly braces
 - Name macros that expand to code like routines so that they can be replaced by routines if necessary
 - Limitations on the Use of Macro Routines
 - Inline Routines
 - Use inline routines sparingly
- Defensive Programming [5.6 + new material]
 - 8.1 Protecting Your Program From Invalid Inputs
 - Check the values of all data from external sources
 - Check the values of all routine input parameters
 - Decide how to handle bad inputs
 - 8.2 Assertions
 - Building Your Own Assertion Mechanism
 - Guidelines for Using Assertions
 - Use error handling code for conditions you expect to occur; use assertions for conditions that should *never* occur
 - Avoid putting executable code in assertions
 - Use assertions to document preconditions and postconditions
 - For highly robust code, assert, and then handle the error anyway
 - 8.3 Error Handling Techniques
 - Return a neutral value
 - Substitute the next piece of valid data
 - Return the same answer as the previous time
 - Substitute the closest legal value
 - Log a warning message to a file
 - Return an error code
 - Call an error processing routine/object
 - Display an error message wherever the error is encountered
 - Handle the error in whatever way works best locally
 - Shutdown

- Robustness vs. Correctness
- High-Level Design Implications of Error Processing
- 8.4 Exceptions
 - Use exceptions to notify other parts of the program about errors that should not be ignored
 - Don't use an exception to pass the buck
 - Avoid throwing exceptions in constructors and destructors unless you catch them in the same place
 - Throw exceptions at the right level of abstraction
 - Include all information that led to the exception in the exception message
 - Avoid empty *catch* blocks
 - Know the exceptions your library code throws
 - Consider building a centralized exception reporter
 - Standardize your project's use of exceptions
 - Consider alternatives to exceptions
- 8.5 Barricade Your Program to Contain the Damage Caused by Errors
 - Convert input data to the proper type at input time
 - Relationship between Barricades and Assertions
- 8.6 Debugging Aids
 - Don't Automatically Apply Production Constraints to the Development Version
 - Introduce Debugging Aids Early
 - Use Offensive Programming
 - Plan to Remove Debugging Aids
 - Use version control and build tools like make
 - Use a built-in preprocessor
 - Write your own preprocessor
 - Use debugging stubs
- 8.7 Determining How Much Defensive Programming to Leave in Production Code
 - Leave in code that checks for important errors
 - Remove code that checks for trivial errors
 - Remove code that results in hard crashes
 - Leave in code that helps the program crash gracefully
 - Log errors for your technical support personnel
 - See that the error messages you leave in are friendly
- 8.8 Being Defensive About Defensive Programming
 - Assertions
 - Exceptions
- The Pseudocode Programming Process [4+new material]
- 9.1 Summary of Steps in Building Classes and Routines
 - Steps in Creating a Class
 - Create a general design for the class
 - Construct each routine within the class
 - Review and test the class as a whole
 - Steps in Building a Routine
- 9.2 Pseudocode for Pros
- 9.3 Constructing Routines Using the PPP
 - Design the Routine
 - Check the prerequisites
 - Define the problem the routine will solve
 - Name the routine
 - Decide how to test the routine
 - Think about error handling
 - Think about efficiency

- Research functionality available in the standard libraries
- Research the algorithms and data types
- Write the pseudocode
- Think about the data
- Check the pseudocode
- Try a few ideas in pseudocode, and keep the best (iterate)
- Code the Routine
 - Write the routine declaration
 - Turn the pseudocode into high-level comments
 - Fill in the code below each comment
 - Check whether code should be further factored
- Check the Code
 - Mentally check the routine for errors
 - Compile the routine
 - Step through the code in the debugger
 - Test the code
 - Remove errors from the routine
- Clean Up Leftovers
- Repeat Steps as Needed
- 9.4 Alternatives to the PPP
 - Test-first development
 - Design by contract
 - Hacking?

VARIABLES

General Issues in Using Variables [10]

- 10.1 Data Literacy
 - The Data Literacy Test
 - Additional Resources on Data Types
- 10.2 Making Variable Declarations Easy
 - Implicit Declarations
 - Turn off implicit declarations
 - Declare all variables
 - Use naming conventions
 - Check variable names
- 10.3 Guidelines for Initializing Variables
 - Initialize each variable as it's declared
 - Initialize each variable close to where it's first used
 - Ideally, declare and define each variable close to where it's used
 - Pay special attention to counters and accumulators
 - Initialize a class's member data in its constructor
 - Check the need for reinitialization
 - Initialize named constants once; initialize variables with executable code
 - Use the compiler setting that automatically initializes all variables
 - Take advantage of your compiler's warning messages
 - Check input parameters for validity
 - Use a memory-access checker to check for bad pointers
 - Initialize working memory at the beginning of your program
- 10.4 Scope
 - Localize References to Variables
 - Keep Variables Live for As Short a Time As Possible
 - Measuring the Live Time of a Variable

- General Guidelines for Minimizing Scope
 - Initialize variables used in a loop immediately before the loop rather than back at the beginning of the routine containing the loop
 - Don't assign a value to a variable until just before the value is used
 - Group related statements
 - Begin with most restricted visibility, and expand the variable's scope only if necessary
- Comments on Minimizing Scope
- 10.5 Persistence
- 10.6 Binding Time
- 10.7 Relationship Between Data Types and Control Structures
 - Sequential data translates to sequential statements in a program
 - Selective data translates to if and case statements in a program
 - Iterative data translates to for, repeat, and while looping structures in a program
- 10.8 Using Each Variable for Exactly One Purpose
 - Use each variable for one purpose only
 - Avoid variables with hidden meanings
 - Make sure that all declared variables are used
- The Power of Variable Names [9]
 - 11.1 Considerations in Choosing Good Names
 - The Most Important Naming Consideration
 - Problem-Orientation
 - Optimum Name Length
 - The Effect of Scope on Variable Names
 - Use qualifiers on names that are in the global name space
 - Computed-Value Qualifiers in Variable Names
 - Common Opposites in Variable Names
 - 11.2 Naming Specific Types of Data
 - Naming Loop Indexes
 - Naming Status Variables
 - Think of a better name than *flag* for status variables
 - Naming Temporary Variables
 - Be leery of "temporary" variables
 - Naming Boolean Variables
 - Keep typical boolean names in mind
 - Give boolean variables names that imply *True* or *False*
 - Use positive boolean variable names
 - Naming Enumerated Types
 - Naming Constants
 - 11.3 The Power of Naming Conventions
 - Why Have Conventions?
 - When You Should Have a Naming Convention
 - Degrees of Formality
 - 11.4 Informal Naming Conventions
 - Guidelines for a Language-Independent Convention
 - Differentiate between variable names and routine names
 - Differentiate between classes and objects
 - Identify global variables
 - Identify member variables
 - Identify type definitions
 - Identify named constants
 - Identify elements of enumerated types
 - Identify input-only parameters in languages that don't enforce them
 - Format names to enhance readability

- Guidelines for Language-Specific Conventions
 - Java Conventions
 - C++ Conventions
 - C Conventions
 - Visual Basic Conventions
 - Mixed-Language Programming Considerations
 - Sample Naming Conventions
- 11.5 Standardized Prefixes
 - User-Defined-Type (UDT) Abbreviation
 - Semantic Prefix
 - Advantages of Standardized Prefixes
- 11.6 Creating Short Names That Are Readable
 - General Abbreviation Guidelines
 - Phonetic Abbreviations
 - Comments on Abbreviations
 - Don't abbreviate by removing one character from a word
 - Abbreviate consistently
 - Create names that you can pronounce
 - Avoid combinations that result in mispronunciation
 - Use a thesaurus to resolve naming collisions
 - Document extremely short names with translation tables in the code
 - Document *all* abbreviations in a project-level "Standard Abbreviations" document
 - Remember that names matter more to the reader of the code than to the writer
- 11.7 Kinds of Names to Avoid
 - Avoid misleading names or abbreviations
 - Avoid names with similar meanings
 - Avoid variables with different meanings but similar names
 - Avoid names that sound similar, such as *wrap* and *rap*
 - Avoid numerals in names
 - Avoid misspelled words in names
 - Avoid words that are commonly misspelled in English
 - Don't differentiate variable names solely by capitalization
 - Avoid multiple natural languages
 - Avoid the names of standard types, variables, and routines
 - Don't use names that are totally unrelated to what the variables represent
 - Avoid names containing hard-to-read characters
- Fundamental Data Types [11]
 - 12.1 Numbers in General
 - Avoid "magic numbers."
 - Use hard-coded 0s and 1s if you need to
 - Anticipate divide-by-zero errors
 - Make type conversions obvious
 - Avoid mixed-type comparisons
 - Heed your compiler's warnings
 - 12.2 Integers
 - Check for integer division
 - Check for integer overflow
 - Check for overflow in intermediate results
 - 12.3 Floating-Point Numbers
 - Avoid additions and subtractions on numbers that have greatly different magnitudes
 - Avoid equality comparisons
 - Anticipate rounding errors
 - Check language and library support for specific data types

12.4 Characters and Strings

- Avoid magic characters and strings
- Watch for off-by-one errors
- Know how your language and environment support Unicode
- Decide on an internationalization/localization strategy early in the lifetime of a program
- If you know you only need to support a single alphabetic language, consider using an ISO 8859 character set
- If you need to support multiple languages, use Unicode
- Decide on a consistent conversion strategy among string types

Strings in C

- Be aware of the difference between string pointers and character arrays
- Declare C-style strings to have length *CONSTANT+1*
- Initialize strings to null to avoid endless strings
- Use arrays of characters instead of pointers in C
- Use *strncpy()* instead of *strcpy()* to avoid endless strings

12.5 Boolean Variables

- Use boolean variables to document your program
- Use boolean variables to simplify complicated tests
- Create your own boolean type, if necessary

12.6 Enumerated Types

- Use enumerated types for readability
- Use enumerated types for reliability
- Use enumerated types for modifiability
- Use enumerated types as an alternative to boolean variables
- Check for invalid values
- Define the first and last entries of an enumeration for use as loop limits
- Reserve the first entry in the enumerated type as invalid
- Define precisely how *First* and *Last* elements are to be used in the project coding standard, and use them consistently
- Beware of pitfalls of assigning explicit values to elements of an enumeration

If Your Language Doesn't Have Enumerated Types

12.7 Named Constants

- Use named constants in data declarations
- Avoid literals, even "safe" ones
- Simulate named constants with appropriately scoped variables or classes
- Use named constants consistently

12.8 Arrays

- Make sure that all array indexes are within the bounds of the array
- Think of arrays as sequential structures
- Check the end points of arrays
- If an array is multidimensional, make sure its subscripts are used in the correct order
- Watch out for index cross talk
- Throw in an extra element at the end of an array
- In C, use the *ARRAY_LENGTH()* macro to work with arrays

12.9 Creating Your Own Types

Why Are the Examples of Creating Your Own Types in Pascal and Ada?

Guidelines for Creating Your Own Types

- Create types with functionally oriented names
- Avoid predefined types
- Don't redefine a predefined type
- Define substitute types for portability
- Consider creating a class rather than using a *typedef*

Unusual Data Types [11.9, 10.6]

13.1 Structures

- Use structures to clarify data relationships
- Use structures to simplify operations on blocks of data
- Use structures to simplify parameter lists
- Use structures to reduce maintenance

13.2 Pointers

Paradigm for Understanding Pointers

- Location in Memory

- Knowledge of How to Interpret the Contents

General Tips on Pointers

- Isolate pointer operations in routines or classes
- Declare and define pointers at the same time
- Check pointers before using them
- Check the variable referenced by the pointer before using it
- Use dog-tag fields to check for corrupted memory
- Add explicit redundancies
- Use extra pointer variables for clarity
- Simplify complicated pointer expressions
- Draw a picture
- Free pointers in linked lists in the right order
- Allocate a reserve parachute of memory
- Free pointers at the same scoping level as they were allocated
- Shred your garbage
- Set pointers to NULL after deleting or freeing them
- Check for bad pointers before deleting a variable
- Keep track of pointer allocations
- Write cover routines to centralize your strategy to avoiding pointer problems
- Use a nonpointer technique

C++ Pointer Pointers

- Understand the difference between pointers and references
- Use pointers for “pass by reference” parameters and *const* references for “pass by value” parameters
- Use *auto_ptr*
- Get smart about smart pointers

C-Pointer Pointers

- Use explicit pointer types rather than the default type
- Avoid type casting
- Follow the asterisk rule for parameter passing
- Use *sizeof()* to determine the size of a variable in a memory allocation

13.3 Global Data

Common Problems with Global Data

- Inadvertent changes to global data
- Bizarre and exciting aliasing problems with global data
- Re-entrant code problems with global data
- Code reuse hindered by global data
- Uncertain initialization-order issues with global data
- Modularity and intellectual manageability damaged by global data

Reasons to Use Global Data

- Preservation of global values
- Emulation of named constants
- Emulation of enumerated types
- Streamlining use of extremely common data
- Eliminating tramp data

Use Global Data Only as a Last Resort

- Begin by making each variable local and make variables global only as you need to

- Distinguish between global and class variables
- Use access routines
- Using Access Routines Instead of Global Data
 - Advantages of Access Routines
 - How to Use Access Routines
 - Require all code to go through the access routines for the data
 - Don't just throw all your global data into the same barrel
 - Use locking to control access to global variables
 - Build a level of abstraction into your access routines
 - Keep all accesses to the data at the same level of abstraction
 - How to Reduce the Risks of Using Global Data
 - Develop a naming convention that makes global variables obvious
 - Create a well-annotated list of all your global variables
 - Don't use global variables to contain intermediate results
 - Don't pretend you're not using global data by putting all your data into a monster object and passing it everywhere

STATEMENTS

Organizing Straight-Line Code [13]

- 14.1 Statements That Must Be in a Specific Order
 - Organize code so that dependencies are obvious
 - Name routines so that dependencies are obvious
 - Use routine parameters to make dependencies obvious
 - Document unclear dependencies with comments
 - Check for dependencies with assertions or error-handling code
- 14.2 Statements Whose Order Doesn't Matter
 - Making Code Read from Top to Bottom
 - Grouping Related Statements

Using Conditionals [14]

- 15.1 *if* Statements
 - Plain *if-then* Statements
 - Write the nominal path through the code first; then write the unusual cases
 - Make sure that you branch correctly on equality
 - Put the normal case after the *if* rather than after the *else*
 - Follow the *if* clause with a meaningful statement
 - Consider the *else* clause
 - Test the *else* clause for correctness
 - Check for reversal of the *if* and *else* clauses
 - Chains of *if-then-else* Statements
 - Put the most common cases first
 - Make sure that all cases are covered
 - Replace *if-then-else* chains with other constructs if your language supports them
- 15.2 *case* Statements
 - Choosing the Most Effective Ordering of Cases
 - Order cases alphabetically or numerically
 - Put the normal case first
 - Order cases by frequency
 - Tips for Using *case* Statements
 - Keep the actions of each case simple
 - Don't make up phony variables in order to be able to use the case statement
 - Use the default clause only to detect legitimate defaults
 - Use the default clause to detect errors
 - In C++ and Java, avoid dropping through the end of a *case* statement

In C++, clearly and unmistakably identify flow-throughs at the end of a *case* statement

Controlling Loops [15]

16.1 Selecting the Kind of Loop

When to Use a *while* Loop

Loop with Test at the Beginning

Loop with Test at the End

When to Use a loop-with-exit Loop

Normal loop-with-exit Loops

Abnormal loop-with-exit Loops

When to Use a *for* Loop

When to Use a *foreach* Loop

16.2 Controlling the Loop

Entering the Loop

Enter the loop from one location only

Put initialization code directly before the loop

In C++, use the *FOREVER* macro for infinite loops and event loops

In C++ and Java, use *for(;;)* or *while(true)* for infinite loops

In C++, prefer *for* loops when they're appropriate

Don't use a *for* loop when a *while* loop is more appropriate

Processing the Middle of the Loop

Use *{* and *}* to enclose the statements in a loop

Avoid empty loops

Keep loop-housekeeping chores at either the beginning or the end of the loop

Make each loop perform only one function

Exiting the Loop

Assure yourself that the loop ends

Make loop-termination conditions obvious

Don't monkey with the loop index of a *for* loop to make the loop terminate

Avoid code that depends on the loop index's final value

Consider using safety counters

Exiting Loops Early

Consider using *break* statements rather than boolean flags in a *while* loop

Be wary of a loop with a lot of *breaks* scattered through it

Use *continue* for tests at the top of a loop

Use labeled *break* if your language supports it

Use *break* and *continue* only with caution

Checking Endpoints

Using Loop Variables

Use ordinal or enumerated types for limits on both arrays and loops

Use meaningful variable names to make nested loops readable

Use meaningful names to avoid loop-index cross talk

Limit the scope of loop-index variables to the loop itself

How Long Should a Loop Be?

Make your loops short enough to view all at once

Limit nesting to three levels

Move loop innards of long loops into routines

Make long loops especially clear

16.3 Creating Loops Easily—from the Inside Out

16.4 Correspondence Between Loops and Arrays

Unusual Control Structures [16]

17.1 Multiple Returns from a Routine

Use a return when it enhances readability

Use guard clauses (early returns or exits) to simplify complex error processing
Minimize the number of returns in each routine

17.2 Recursion

Example of Recursion

Tips for Using Recursion

Make sure the recursion stops

Use safety counters to prevent infinite recursion

Limit recursion to one routine

Keep an eye on the stack

Don't use recursion for factorials or Fibonacci numbers

17.3 *goto*

The Argument Against *gotos*

The Argument for *gotos*

The Phony *goto* Debate

Error Processing and *gotos*

Rewrite with nested *if* statements

Rewrite with a status variable

Rewrite with *try-finally*

Comparison of the Approaches

Summary of Guidelines for Using *gotos*

17.4 Perspective on Unusual Control Structures

Returns

Table-Driven Methods [12.2]

18.1 General Considerations in Using Table-Driven Methods

Two Issues in Using Table-Driven Methods

18.2 Direct Access Tables

Days-in-Month Example

Insurance-Rates Example

Flexible-Message-Format Example

Logic-Based Approach

Object-Oriented Approach

The Table-Driven Approach

Fudging Lookup Keys

Duplicate information to make the key work directly

Transform the key to make it work directly

Isolate the key-transformation in its own routine

18.3 Indexed Access Tables

18.4 Stair-Step Access Tables

Watch the endpoints

Consider using a binary search rather than a sequential search

Consider using indexed access instead of the stair-step technique

Put the stair-step table lookup into its own routine

18.5 Other Examples of Table Lookups

General Control Issues [17]

19.1 Boolean Expressions

Using *True* and *False* for Boolean Tests

Compare boolean values to *True* and *False* implicitly

In C, use the *I==I* trick to define *TRUE* and *FALSE*

Making Complicated Expressions Simple

Break complicated tests into partial tests with new boolean variables

Move complicated expressions into boolean functions

- Use decision tables to replace complicated conditions
- Forming Boolean Expressions Positively
 - In *if* statements, convert negatives to positives and flip-flop the code in the *if* and *else* clauses
 - Apply DeMorgan's Theorems to simplify boolean tests with negatives
- Using Parentheses to Clarify Boolean Expressions
 - Use a simple counting technique to balance parentheses
 - Fully parenthesize logical expressions
- Knowing How Boolean Expressions Are Evaluated
- Writing Numeric Expressions in Number-Line Order
- Guidelines for Comparisons to 0
 - Compare logical variables implicitly
 - Compare numbers to 0
 - Compare characters to the null terminator (<;\$QS>\0<;\$QS>) explicitly
 - Compare pointers to NULL
- Common Problems with Boolean Expressions
 - In C and C++, put constants on the left side of comparisons
 - In C++, consider creating preprocessor macro substitutions for &&, <;\$LB><;\$LB>, and == (but only as a last resort)
 - In Java, know the difference between *a==b* and *a.equals(b)*
- 19.2 Compound Statements (Blocks)
 - Write pairs of braces together
 - Use braces to clarify conditionals
- 19.3 Null Statements
 - Call attention to null statements
 - Create a preprocessor *null()* macro or inline function for null statements
 - Consider whether the code would be clearer with a non-null loop body
- 19.4 Taming Dangerously Deep Nesting
 - Simplify a nested *if* by retesting part of the condition
 - Simplify a nested *if* by using a *break* block
 - Convert a nested *if* to a set of *if-then-elses*
 - Convert a nested *if* to a *case* statement
 - Factor deeply nested code into its own routine
 - Use a more object-oriented approach
 - Redesign deeply nested code
- Summary of Techniques for Reducing Deep Nesting
- 19.5 A Programming Foundation: Structured Programming
 - The Three Components of Structured Programming
 - Sequence
 - Selection
 - Iteration
- 19.6 Control Structures and Complexity
 - How Important Is Complexity?
 - General Guidelines for Reducing Complexity
 - How to Measure Complexity
 - What to Do with Your Complexity Measurement
 - Other Kinds of Complexity

CODE IMPROVEMENTS

The Software-Quality Landscape [23]

- 20.1 Characteristics of Software Quality
- 20.2 Techniques for Improving Software Quality
 - Software-quality objectives

- Explicit quality-assurance activity
- Testing strategy
- Software-engineering guidelines
- Informal technical reviews
- Formal technical reviews
- External audits
- Development process
- Change-control procedures
- Measurement of results
- Prototyping
- Setting Objectives
- 20.3 Relative Effectiveness of Quality Techniques
 - Percentage of Defects Detected
 - Cost of Finding Defects
 - Cost of Fixing Defects
- 20.4 When to Do Quality Assurance
- 20.5 The General Principle of Software Quality
 - Relevant Standards
- Collaborative Construction [24]
 - 21.1 Overview of Collaborative Development Practices
 - Collaborative Construction Complements Other Quality-Assurance Techniques
 - Collaborative Construction Provides Mentoring in Corporate Culture and Programming Expertise
 - Collective Ownership Applies to All Forms of Collaborative Construction
 - Collaboration Applies As Much Before Construction As After
 - 21.2 Pair Programming
 - Keys to Success with Pair Programming
 - Support pair programming with coding standards
 - Don't let pair programming turn into watching
 - Don't force pair programming of the easy stuff
 - Rotate pairs and work assignments regularly
 - Encourage pairs to match each other's pace
 - Make sure both partners can see the monitor
 - Don't force people who don't like each other to pair
 - Avoid pairing all newbies
 - Assign a team leader
 - Benefits of Pair Programming
 - 21.3 Formal Inspections
 - What Results Can You Expect from Inspections?
 - Roles During an Inspection
 - Moderator
 - Author
 - Reviewer
 - Scribe
 - Management
 - General Procedure for an Inspection
 - Planning
 - Overview
 - Preparation
 - Perspectives
 - Scenarios
 - Inspection Meeting
 - Inspection Report
 - Rework

- Follow-Up
- Third-Hour Meeting
- Fine-Tuning the Inspection
- Egos in Inspections
- Inspections and *Code Complete*
- Inspection Summary
- 21.4 Other Kinds of Collaborative Development Practices
 - Walkthroughs
 - What Results Can You Expect From A Walkthrough?
 - Code Reading
 - Dog-and-Pony Shows
 - Pair Programming
 - Inspections
 - Relevant Standards
- Developer Testing [25]
 - 22.1 Role of Developer Testing in Software Quality
 - Testing During Construction
 - 22.2 Recommended Approach to Developer Testing
 - Test First or Test Last?
 - Limitations of Developer Testing
 - Developer tests tend to be “clean tests”
 - Developer testing tends to have an optimistic view of test coverage
 - Developer testing tends to skip more sophisticated kinds of test coverage
 - 22.3 Bag of Testing Tricks
 - Incomplete Testing
 - Structured Basis Testing
 - Data-Flow Testing
 - Defined
 - Used
 - Killed
 - Entered
 - Exited
 - Combinations of Data States
 - Defined-Defined
 - Defined-Exited
 - Defined-Killed
 - Entered-Killed
 - Entered-Used
 - Killed-Killed
 - Killed-Used
 - Used-Defined
 - Equivalence Partitioning
 - Error Guessing
 - Boundary Analysis
 - Compound Boundaries
 - Classes of Bad Data
 - Classes of Good Data
 - Use Test Cases That Make Hand-Checks Convenient
 - 22.4 Typical Errors
 - Which Classes Contain the Most Errors?
 - Errors by Classification
 - The scope of most errors is fairly limited
 - Many errors are outside the domain of construction
 - Most construction errors are the programmers’ fault

- Clerical errors (typos) are a surprisingly common source of problems
- Misunderstanding the design is a recurring theme in studies of programmer errors
- Most errors are easy to fix
- It's a good idea to measure your own organization's experiences with errors
- Proportion of Errors Resulting from Faulty Construction
- How Many Errors Should You Expect to Find?
- Errors in Testing Itself
 - Check your work
 - Plan test cases as you develop your software
 - Keep your test cases
 - Plug unit tests into a test framework
- 22.5 Test-Support Tools
 - Building Scaffolding to Test Individual Classes
 - Diff Tools
 - Test-Data Generators
 - Coverage Monitors
 - Data Recorder
 - Symbolic Debuggers
 - System Perturbers
 - Error Databases
- 22.6 Improving Your Testing
 - Planning to Test
 - Retesting (Regression Testing)
 - Automated Testing
- 22.7 Keeping Test Records
 - Personal Test Records
 - Testing
 - Test Scaffolding
 - Test First Development
 - Relevant Standards
- Debugging [26]
 - 23.1 Overview of Debugging Issues
 - Role of Debugging in Software Quality
 - Variations in Debugging Performance
 - Defects as Opportunities
 - Learn about the program you're working on
 - Learn about the kind of mistakes you make
 - Learn about the quality of your code from the point of view of someone who has to read it
 - Learn about how you solve problems
 - Learn about how you fix defects
 - An Ineffective Approach
 - The Devil's Guide to Debugging
 - Find the defect by guessing
 - Don't waste time trying to understand the problem
 - Fix the error with the most obvious fix
 - Debugging by Superstition
 - 23.2 Finding a Defect
 - The Scientific Method of Debugging
 - Stabilize the Error
 - Locate the Source of the Error
 - Tips for Finding Defects
 - Use all the data available to make your hypothesis
 - Refine the test cases that produce the error

- Exercise the code in your unit test suite
- Use available tools
- Reproduce the error several different ways
- Generate more data to generate more hypotheses
- Use the results of negative tests
- Brainstorm for possible hypotheses
- Narrow the suspicious region of the code
- Be suspicious of classes and routines that have had defects before
- Check code that's changed recently
- Expand the suspicious region of the code
- Integrate incrementally
- Check for common defects
- Talk to someone else about the problem
- Take a break from the problem
- Brute Force Debugging
 - Set a maximum time for quick and dirty debugging
 - Make a list of brute force techniques
- Syntax Errors
 - Don't trust line numbers in compiler messages
 - Don't trust compiler messages
 - Don't trust the compiler's second message
 - Divide and conquer
 - Find extra comments and quotation marks
- 23.3 Fixing a Defect
 - Understand the problem before you fix it
 - Understand the program, not just the problem
 - Confirm the defect diagnosis
 - Relax
 - Save the original source code
 - Fix the problem, not the symptom
 - Change the code only for good reason
 - Make one change at a time
 - Check your fix
 - Look for similar defects
- 23.4 Psychological Considerations in Debugging
 - How "Psychological Set" Contributes to Debugging Blindness
 - How "Psychological Distance" Can Help
- 23.5 Debugging Tools—Obvious and Not-So-Obvious
 - Diff
 - Compiler Warning Messages
 - Set your compiler's warning level to the highest, pickiest level possible and fix the code so that it doesn't produce any compiler warnings
 - Treat warnings as errors
 - Initiate project wide standards for compile-time settings
 - Extended Syntax and Logic Checking
 - Execution Profiler
 - Test Frameworks/Scaffolding
 - Debugger
- Refactoring [Mostly new material; some from 30]
 - 24.1 Kinds of Software Evolution
 - Philosophy of Software Evolution
 - 24.2 Introduction to Refactoring
 - 24.3 Reasons to Refactor

- Code is duplicated
- A routine is too long
- A loop is too long or too deeply nested
- A class has poor cohesion
- A class interface does not provide a consistent level of abstraction
- A parameter list has too many parameters
- Changes within a class tend to be compartmentalized
- Changes require parallel modifications to multiple classes
- Inheritance hierarchies have to be modified in parallel
- Related data items that are used together are not organized into classes
- A routine uses more features of another class than of its own class
- A primitive data type is overloaded
- A class doesn't do very much
- A chain of routines passes tramp data
- A middle man object isn't doing anything
- One class is overly intimate with another
- A routine has a poor name
- Data members are public
- A subclass uses only a small percentage of its parents' routines
- Comments are used to explain difficult code
- Global variables are used
- A routine uses setup code before a routine call or takedown code after a routine call
- A program contains code that seems like it might be needed someday

Reasons Not To Refactor

24.4 Specific Refactorings

Data Level Refactorings

- Replace a magic number with a named constant
- Rename a variable with a clearer or more informative name
- Move an expression inline
- Replace an expression with a routine
- Introduce an intermediate variable
- Convert a multi-use variable to multiple single-use variables
- Use a local variable for local purposes rather than a parameter
- Convert a data primitive to a class
- Convert a set of type codes to a class
- Convert a set of type codes to a class with subclasses
- Change an array to an object
- Encapsulate a collection
- Replace a traditional record with a data class

Statement Level Refactorings

- Decompose a boolean expression
- Move a complex boolean expression into a well-named boolean function
- Consolidate fragments that are duplicated within different parts of a conditional
- Use *break* or *return* instead of a loop control variable
- Return as soon as you know the answer instead of assigning a return value within nested *if-then-else* statements
- Replace conditionals with polymorphism (especially repeated *case* statements)
- Create and use null objects instead of testing for null values

Routine Level Refactorings

- Extract a routine
- Move a routine's code inline
- Convert a long routine to a class
- Substitute a simple algorithm for a complex algorithm
- Add a parameter
- Remove a parameter
- Separate query operations from modification operations

- Combine similar routines by parameterizing them
- Separate routines whose behavior depends on parameters passed in
- Pass a whole object rather than specific fields
- Pass specific fields rather than a whole object
- Encapsulate downcasting
- Class Implementation Refactorings
 - Change value objects to reference objects
 - Change reference objects to value objects
 - Replace virtual routines with data initialization
 - Change member routine or data placement
 - Extract specialized code into a subclass
 - Combine similar code into a superclass
- Class Interface Refactorings
 - Move a routine to another class
 - Convert one class to two
 - Eliminate a class
 - Hide a delegate
 - Replace inheritance with delegation
 - Replace delegation with inheritance
 - Remove a middle man
 - Introduce a foreign routine
 - Introduce an extension class
 - Encapsulate an exposed member variable
 - Remove *Set()* routines for fields that cannot be changed
 - Hide routines that are not intended to be used outside the class
 - Encapsulate unused routines
 - Collapse a superclass and subclass if their implementations are very similar
- System Level Refactorings
 - Create a definitive reference source for data you can't control
 - Change unidirectional class association to bidirectional class association
 - Change bidirectional class association to unidirectional class association
 - Provide a factory method rather than a simple constructor
 - Replace error codes with exceptions or vice versa
- 24.5 Refactoring Safely
 - Keys to Refactoring Safely
 - Save the code you start with
 - Keep refactorings small
 - Do refactorings one at a time
 - Make a list of steps you intend to take
 - Make a parking lot
 - Make frequent checkpoints
 - Use your compiler warnings
 - Retest
 - Add test cases
 - Review the changes
 - Adjust your approach depending on the risk level of the refactoring
 - Bad Times to Refactor
 - Don't use refactoring as a cover for code and fix
 - Avoid refactoring instead of rewriting
- 24.6 Refactoring Strategies
 - Refactor when you add a routine
 - Refactor when you add a class
 - Refactor when you fix a defect
 - Target error-prone modules
 - Target high complexity modules

In a maintenance environment, improve the parts you touch
Define an interface between clean code and ugly code, and then move code across the interface

Code-Tuning Strategies [28]

25.1 Performance Overview

Quality Characteristics and Performance

Performance and Code Tuning

Program Requirements

Program Design

Class and Routine Design

Operating-System Interactions

Code Compilation

Hardware

Code Tuning

25.2 Introduction to Code Tuning

The Pareto Principle

Old Wives' Tales

Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code—false!

Certain operations are probably faster or smaller than others—false!

You should optimize as you go—false!

A fast program is just as important as a correct one—false!

When to Tune

Compiler Optimizations

25.3 Kinds of Fat and Molasses

Common Sources of Inefficiency

Input/output operations

Paging

System calls

Interpreted languages

Errors

Relative Performance Costs of Common Operations

25.4 Measurement

Measurements Need to be Precise

25.5 Iteration

25.6 Summary of the Approach to Code Tuning

Performance

Algorithms and Data Types

Code-Tuning Techniques [29]

26.1 Logic

Stop Testing When You Know the Answer

Order Tests by Frequency

Compare Performance of Similar Logic Structures

Substitute Table Lookups for Complicated Expressions

Use Lazy Evaluation

26.2 Loops

- Unswitching
- Jamming
- Unrolling
- Minimizing the Work Inside Loops
- Sentinel Values
- Putting the Busiest Loop on the Inside
- Strength Reduction
- 26.3 Data Transformations
 - Use Integers Rather Than Floating-Point Numbers
 - Use the Fewest Array Dimensions Possible
 - Minimize Array References
 - Use Supplementary Indexes
 - String-Length Index
 - Independent, Parallel Index Structure
 - Use Caching
- 26.4 Expressions
 - Exploit Algebraic Identities
 - Use Strength Reduction
 - Initialize at Compile Time
 - Be Wary of System Routines
 - Use the Correct Type of Constants
 - Precompute Results
 - Eliminate Common Subexpressions
- 26.5 Routines
 - Rewrite Routines In Line
- 26.6 Recoding in Assembler
- 26.7 The More Things Change, the More They Stay the Same

SYSTEM CONSIDERATIONS

How Program Size Affects Construction [21]

- 27.1 Communication and Size
- 27.2 Range of Project Sizes
- 27.3 Effect of Project Size on Errors
- 27.4 Effect of Project Size on Productivity
- 27.5 Effect of Project Size on Development Activities
 - Activity Proportions and Size
 - Programs, Products, Systems, and System Products
 - Methodology and Size

Managing Construction [22, and some from 27.4]

- 28.1 Encouraging Good Coding
 - Considerations in Setting Standards
 - Techniques
 - Assign two people to every part of the project
 - Review every line of code
 - Require code sign-offs
 - Route good code examples for review
 - Emphasize that code listings are public assets
 - Reward good code
 - One easy standard
 - The Role of This Book
- 28.2 Configuration Management

- What Is Configuration Management?
 - Requirements and Design Changes
 - Follow a systematic change-control procedure
 - Handle change requests in groups
 - Estimate the cost of each change
 - Be wary of high change volumes
 - Establish a change-control board or its equivalent in a way that makes sense for your project
 - Watch for bureaucracy, but don't let the fear of bureaucracy preclude effective change control
 - Software Code Changes
 - Version-control software
 - Tool Versions
 - Machine Configurations
 - Backup Plan
 - Additional Resources on Configuration Management
 - 28.3 Estimating a Construction Schedule
 - Estimation Approaches
 - Establish objectives
 - Allow time for the estimate, and plan it
 - Spell out software requirements
 - Estimate at a low level of detail
 - Use several different estimation techniques, and compare the results
 - Re-estimate periodically
 - Estimating the Amount of Construction
 - Influences on Schedule
 - Estimation vs. Control
 - What to do If You're Behind
 - Hope that you'll catch up
 - Expand the team
 - Reduce the scope of the project
 - Additional Resources on Software Estimation
 - 28.4 Measurement
 - For any project attribute, it's possible to measure that attribute in a way that's superior to not measuring it at all
 - To argue against measurement is to argue that it's better not to know what's really happening on your project
 - Additional Resources on Software Measurement
 - 28.5 Treating Programmers as People
 - How do Programmers Spend Their Time?
 - Variation in Performance and Quality
 - Individual Variation
 - Team Variation
 - Religious Issues
 - be aware that you're dealing with a sensitive area
 - Use "suggestions" or "guidelines" with respect to the area
 - Finesse the issues you can by sidestepping explicit mandates
 - Have your programmers develop their own standards
 - Physical Environment
 - Additional Resources on Programmers as Human Beings
 - 28.6 Managing Your Manager
 - Relevant Standards
- Integration [27]
- 29.1 Importance of the Integration Approach

- 29.2 Integration Frequency—Phased or Incremental?
 - Phased Integration
 - Incremental Integration
 - Benefits of Incremental Integration
 - Errors are easy to locate
 - The system succeeds early in the project
 - You get improved progress monitoring
 - You'll improve customer relations
 - The units of the system are tested more fully
 - You can build the system with a shorter development schedule
 - 29.3 Incremental Integration Strategies
 - Top-Down Integration
 - Bottom-Up Integration
 - Sandwich Integration
 - Risk-Oriented Integration
 - Feature-Oriented Integration
 - T-Shaped Integration
 - Summary of Integration Approaches
 - 29.4 Daily Build and Smoke Test
 - Build daily
 - Check for broken builds
 - Smoke test daily
 - Automate the daily build and smoke test
 - Establish a build group
 - Add revisions to the build only when it makes sense to do so
 - ... but don't wait too long to add a set of revisions
 - Require developers to smoke test their code before adding it to the system
 - Create a holding area for code that's to be added to the build
 - Create a penalty for breaking the build
 - Release builds in the morning
 - Build and smoke test even under pressure
 - What Kinds of Projects Can Use the Daily Build Process?
 - Continuous Integration
 - Integration
 - Incrementalism
- Programming Tools [20]
- 30.1 Design Tools
 - 30.2 Source-Code Tools
 - Editing
 - Integrated Development Environments (IDEs)
 - Multiple-File String Searching and Replacing
 - Diff Tools
 - Merge Tools
 - Source-Code Beautifiers
 - Interface Documentation Tools
 - Templates
 - Cross-Reference Tools
 - Class Hierarchy Generators
 - Analyzing Code Quality
 - Picky Syntax and Semantics Checkers
 - Metrics Reporters
 - Refactoring Source Code
 - Refactorers
 - Restructurers

- Code Translators
- Version Control
- Data Dictionaries
- 30.3 Executable-Code Tools
 - Code Creation
 - Compilers and Linkers
 - Make
 - Code Libraries
 - Code Generation Wizards
 - Setup and Installation
 - Macro Preprocessors
 - Debugging
 - Testing
 - Code Tuning
 - Execution Profilers
 - Assembler Listings and Disassemblers
- 30.4 Tool-Oriented Environments
 - UNIX
- 30.5 Building Your Own Programming Tools
 - Project-Specific Tools
 - Scripts
- 30.6 Tool Fantasyland

SOFTWARE CRAFTSMANSHIP

Layout and Style [18]

- 31.1 Layout Fundamentals
 - Layout Extremes
 - The Fundamental Theorem of Formatting
 - Human and Computer Interpretations of a Program
 - How Much Is Good Layout Worth?
 - Layout as Religion
 - Objectives of Good Layout
 - Accurately represent the logical structure of the code
 - Consistently represent the logical structure of the code
 - Improve readability
 - Withstand modifications
 - How to Put the Layout Objectives to Use
- 31.2 Layout Techniques
 - White Space
 - Grouping
 - Blank lines
 - Indentation
 - Parentheses
- 31.3 Layout Styles
 - Pure Blocks
 - Emulating Pure Blocks
 - Using *begin-end* pairs (braces) to Designate Block Boundaries
 - Endline Layout
 - Which Style Is Best?
- 31.4 Laying Out Control Structures
 - Fine Points of Formatting Control-Structure Blocks
 - Avoid unindented *begin-end* pairs
 - Avoid double indentation with *begin* and *end*

- Other Considerations
 - Use blank lines between paragraphs
 - Format single-statement blocks consistently
 - For complicated expressions, put separate conditions on separate lines
 - Avoid *gotos*
 - No newline exception for *case* statements
- 31.5 Laying Out Individual Statements
 - Statement Length
 - Using Spaces for Clarity
 - Use spaces to make logical expressions readable
 - Use spaces to make array references readable
 - Use spaces to make routine arguments readable
 - Formatting Continuation Lines
 - Make the incompleteness of a statement obvious
 - Keep closely related elements together
 - Indent routine-call continuation lines the standard amount
 - Make it easy to find the end of a continuation line
 - Indent control-statement continuation lines the standard amount
 - Do not align right sides of assignment statements
 - Indent assignment-statement continuation lines the standard amount
 - Using Only One Statement per Line
 - In C++, avoid using multiple operations per line (side effects)
 - Laying Out Data Declarations
 - Use only one data declaration per line
 - Declare variables close to where they're first used
 - Order declarations sensibly
 - In C++, put the asterisk next to the variable name in pointer declarations or declare pointer types
- 31.6 Laying Out Comments
 - Indent a comment with its corresponding code
 - Set off each comment with at least one blank line
- 31.7 Laying Out Routines
 - Use blank lines to separate parts of a routine
 - Use standard indentation for routine arguments
- 31.8 Laying Out Classes
 - Laying Out Class Interfaces
 - Laying Out Class Implementations
 - If you have more than one class in a file, identify each class clearly
 - Laying Out Files and Programs
 - Put one class in one file
 - Give the file a name related to the class name
 - Separate routines within a file clearly
 - Sequence routines alphabetically
 - In C++, order the source file carefully
- Self-Documenting Code [19]
 - 32.1 External Documentation
 - Unit development folders
 - Detailed-design document
 - 32.2 Programming Style as Documentation
 - 32.3 To Comment or Not to Comment
 - 32.4 Keys to Effective Comments

- Kinds of Comments
 - Repeat of the Code
 - Explanation of the Code
 - Marker in the Code
 - Summary of the Code
 - Description of the Code's Intent
- Commenting Efficiently
 - Use styles that don't break down or discourage modification
 - Use the Pseudocode Programming Process to reduce commenting time
 - Integrate commenting into your development style
 - Performance is not a good reason to avoid commenting
- Optimum Number of Comments
- 32.5 Commenting Techniques
 - Commenting Individual Lines
 - Avoid self-indulgent comments
 - Endline Comments and Their Problems
 - Avoid endline comments on single lines
 - Avoid endline comments for multiple lines of code
 - When to Use Endline Comments
 - Use endline comments to annotate data declarations
 - Avoid using endline comments for maintenance notes
 - Use endline comments to mark ends of blocks
 - Commenting Paragraphs of Code
 - Write comments at the level of the code's intent
 - Focus your documentation efforts on the code itself
 - Focus paragraph comments on the *why* rather than the *how*
 - Use comments to prepare the reader for what is to follow
 - Make every comment count
 - Document surprises
 - Avoid abbreviations
 - Differentiate between major and minor comments
 - Comment anything that gets around an error or an undocumented feature in a language or an environment
 - Justify violations of good programming style
 - Don't comment tricky code
 - Commenting Data Declarations
 - Comment the units of numeric data
 - Comment the range of allowable numeric values
 - Comment coded meanings
 - Comment limitations on input data
 - Document flags to the bit level
 - Stamp comments related to a variable with the variable's name
 - Document global data
 - Commenting Control Structures
 - Put a comment before each block of statements, *if*, *case*, or loop
 - Comment the end of each control structure
 - Treat end-of-loop comments as a warning indicating complicated code
 - Commenting Routines
 - Keep comments close to the code they describe
 - Describe each routine in one or two sentences at the top of the routine
 - Document parameters where they are declared
 - Differentiate between input and output data
 - Document interface assumptions
 - Comment on the routine's limitations
 - Document the routine's global effects
 - Document the source of algorithms that are used

- Use comments to mark parts of your program
- Commenting Classes, Files, and Programs
 - General Guidelines for Class Documentation
 - Describe the design approach to the class
 - Describe limitations, usage assumptions, and so on
 - Comment the class interface
 - Don't document implementation details in the class interface
 - General Guidelines for File Documentation
 - Describe the purpose and contents of each file
 - Put your name, email address, and phone number in the block comment
 - Include a copyright statement in the block comment
 - Give the file a name related to its contents
 - The Book Paradigm for Program Documentation
 - Software Development Standards
 - Software Quality Assurance Standards
 - Management Standards
 - Overview of Standards
- Personal Character [31]
 - 33.1 Isn't Personal Character Off the Topic?
 - 33.2 Intelligence and Humility
 - 33.3 Curiosity
 - Build your awareness of the development process
 - Experiment
 - Read about problem solving
 - Analyze and plan before you act
 - Learn about successful projects
 - Read!
 - Read other books and periodicals
 - Make a commitment to professional development
 - 33.4 Intellectual Honesty
 - 33.5 Communication and Cooperation
 - 33.6 Creativity and Discipline
 - 33.7 Laziness
 - 33.8 Characteristics That Don't Matter As Much As You Might Think
 - Persistence
 - Experience
 - Gonzo Programming
 - 33.9 Habits
- Themes in Software Craftsmanship [32]
 - 34.1 Conquer Complexity
 - 34.2 Pick Your Process
 - 34.3 Write Programs for People First, Computers Second
 - 34.4 Program Into Your Language, Not In It
 - 34.5 Focus Your Attention with the Help of Conventions
 - 34.6 Program in Terms of the Problem Domain
 - Separating a Program into Levels of Abstraction
 - Level 0: Operating System Operations and Machine Instructions
 - Level 1: Programming-Language Structures and Tools
 - Level 2: Low-Level Implementation Structures
 - Level 3: Low-Level Problem-Domain Terms

- Level 4: High-Level Problem-Domain Terms
- Low-Level Techniques for Working in the Problem Domain
- 34.7 Watch for Falling Rocks
- 34.8 Iterate, Repeatedly, Again and Again
- 34.9 Thou Shalt Rend Software and Religion Asunder
 - Software Oracles
 - Eclecticism
 - Experimentation
- Where to Find More Information [33]
- 35.1 Information About Software Construction
- 35.2 Topics Beyond Construction
 - Overview Material
 - Software-Engineering Overviews
 - Other Annotated Bibliographies
- 35.3 Periodicals
 - Lowbrow Programmer Magazines
 - Highbrow Programmer Journals
 - Special-Interest Publications
 - Professional Publications
 - Popular-Market Publications
- 35.4 A Software Developer's Reading Plan
 - Introductory Level
 - Practitioner Level
 - Professional Level
- 35.5 Joining a Professional Organization