

# Contents

## Introduction

## Part I The Techniques

### Chapter 1 Start with a Good Design

- Technique 1: Define the Problem
- Technique 2: Use the 1-3-5 Rule
- Technique 3: Solution in Search of a Problem
- Technique 4: Problem in Search of a Solution
- Technique 5: In Search of a Solution and Problem
- Technique 6: KISS - Keep It Simple and Stupid
- Technique 7: The Importance of Good Design
- Technique 8: The Customer Is Always Right
- Technique 9: Design Is an Iterative Process
- Technique 10: Coding Is Also an Iterative Process
- Technique 11: Work Smarter, Not Longer
- Technique 12: Four Eyes Are Better than Two
- Technique 13: Measure Twice, Cut Once
- Technique 14: Know When to Say When
- Technique 15: Pick the Right Algorithm and Data Structures for the Problem

### Chapter 2 Darn Reasonable Practices

- Technique 16: Be Careful When Cutting and Pasting Code
- Technique 17: Initialize Local Variables
- Technique 18: Validate Arguments
- Technique 19: Check Return Codes
- Technique 20: Check for 0 before Dividing
- Technique 21: Check for 0 before Mod'ing
- Technique 22: Write the Loop Increment/Decrement First
- Technique 23: 0 == a versus a == 0
- Technique 24: Use Header Sentinels
- Technique 25: Intrinsic Use Byte Counts
- Technique 26: memcpy Uses Byte Counts
- Technique 27: Use const instead of Numeric Literals
- Technique 28: const Precomputes Sizes
- Technique 29: Sometimes You Need to Use #define instead of const
- Technique 30: Using an Assert instead of, or in addition to, Commenting Code
- Technique 31: Handle the Exception Even If You Assert
- Technique 32: Carefully Choose Signed or Unsigned Types
- Technique 33: Be Careful When You Mix Signed and Unsigned Values
- Technique 34: Use Parentheses If You Need Them
- Technique 35: Use Parentheses If You Need Them
- Technique 36: Always Treat Warnings as Errors

- Technique 37: Always Use at Least Warning Level 3
- Technique 38: Be Careful When Using \ in Strings
- Technique 39: Be Careful When You Mix C and C++
- Technique 40: Determining If a Constant Is Already Defined
- Technique 41: Know Exactly What Functions Do  
(At Least If You Use Them)
- Technique 42: Make Sure Strings Are Null Terminated
- Technique 43: Check the Actual Size
- Technique 44: Use Inline Functions instead of Macros

## **Chapter 3 Dealing with Compiler-Generated Code**

- Technique 45: Constructors and Destructors Are Automatically Created for You
- Technique 46: Wrap new to Save Space
- Technique 47: If You Overload new, Overload Delete
- Technique 48: The + Operator Generates Considerable Overhead

## **Chapter 4 Pointers and Memories**

- Technique 49: Check for Successful Allocation
- Technique 50: Free Once, Not Often
- Technique 51: Make Sure You Can Allocate before You Replace
- Technique 52: Be Prepared for Multiple Calls to Buffer Allocation Methods
- Technique 53: Don't Return Pointers to Automatic Variables
- Technique 54: Always Initialize to a Known State
- Technique 55: Avoid Self-Referential Copies
- Technique 56: Use const\*
- Technique 57: Avoiding Errors When Using const const
- Technique 58: Use Smart Pointers
- Technique 59: Use Smart Pointers for Objects
- Technique 60: Custom Memory Allocators and Smart Pointers
- Technique 61: Don't Blow Your Buffers
- Technique 62: Create a Memory Manager That Detects Leaks
- Technique 63: Global versus Class-Specific Memory Allocators

## **Chapter 5 Arrays**

- Technique 64: Use delete [ ] with Arrays
- Technique 65: Avoid Index Underflow
- Technique 66: Avoid Index Overflow
- Technique 67: foo[K] Is the Same as foo.operator[ ] (K)
- Technique 68: An Array Is a Pointer

## **Chapter 6 Classes**

- Technique 69: Initialize Member Variables
- Technique 70: Use Initialization Lists
- Technique 71: Don't Initialize consts inside the Body of the Constructor
- Technique 72: const Member Variables Allocate Memory
- Technique 73: Nonstatic consts Are Allocated in the Code Area
- Technique 74: Wrap Member Variables to Protect from Outside Access
- Technique 75: Keeping Your News Private
- Technique 76: Using Runtime Type Information
- Technique 77: If You Allocate Memory, Override the = Operator
- Technique 78: Guard against Self-Referential Operators
- Technique 79: Too Much Parameter Checking Is Still Too Much
- Technique 80: Creating const Operators
- Technique 81: Private Derivation
- Technique 82: Making Sure a Class or Structure Size Stays within Bounds
- Technique 83: Use Inheritance instead of Unions
- Technique 84: Be Careful with Initialization When You Use Unions
- Technique 85: Don't Cast to a Derived Class
- Technique 86: Be Very Careful when Using Member Variables after Deleting this

## **Chapter 7 Abstract Base Classes**

- Technique 87: The Compiler Checks for Instantiation of Abstract Base Classes
- Technique 88: Base Classes Can Be Designed to Assume That Derived Classes Provide Implementation
- Technique 89: vtables Use Space
- Technique 90: Multiple Inheritance from Abstract Base Classes Does Not Cause Ambiguities

## **Chapter 8 Constructors**

- Technique 91: Initialize Your Member Variables
- Technique 92: Default Arguments Let You Reduce the Number of Constructors
- Technique 93: Making Constructors Private
- Technique 94: If You Allocate Memory, Create a Copy Constructor
- Technique 95: If You Create a Copy Constructor, Remember to Copy Over Allocated Memory
- Technique 96: If You Expect to Inherit, Make Destructors Virtual
- Technique 97: If You Have Multiple Constructors, Be Explicit about Which One to Use

## **Chapter 9 Inheritance**

- Technique 98: IsA versus HasA
- Technique 99: Scoping Rules
- Technique 100: Multiple Inheritance Can Cause Ambiguities

## **Chapter 10 Operator Overloading**

- Technique 101: The Difference between Pre- and Postfix Operators
- Technique 102: Be Careful When You Overload Operators
- Technique 103: Don't Change the Meaning of Operators
- Technique 104: Overload = If You Expect to Use Assignment with Classes
- Technique 105: Operators Can Do More Work than You Think
- Technique 106: Don't Return References to Local Variables

## **Chapter 11 Templates**

- Technique 107: Keep the Template Implementation in a Different Class

## **Chapter 12 Miscellaneous Goop**

- Technique 108: Inserting Graphic File Resources
- Technique 109: Make Sure Your Paths Are Correct
- Technique 110: Keep Reference-Counted Objects off the Stack
- Technique 111: Passing Preallocated Memory to new
- Technique 112: Aliasing with References

## **Chapter 13 Performance**

- Technique 113: Design Is More Important than Tuning
- Technique 114: Know What to Improve
- Technique 115: Instrument Your Code
- Technique 116: Reduce Your Working Set
- Technique 117: Use the optimize for space Flag
- Technique 118: Delay Loading
- Technique 119: Invest in Good Tools
- Technique 120: Templates Usually Mean Bloat
- Technique 121: Floating Point Is Faster than Integer Math on a Pentium.
- Technique 122: Look-Up Tables Can Increase Performance
- Technique 123: Be Cautious When Using Inline Functions
- Technique 124: Know What Code Is Generated
- Technique 125: Shifts Are Faster than Divides
- Technique 126: Pointer Arithmetic Is Not Faster than Array Look Ups
- Technique 127: Memory Allocations Are Expensive
- Technique 128: Be Careful When Using String Functions
- Technique 129: Avoid the CRT If You Can
- Technique 130: Intrinsics Are Faster than the CRT

## **Chapter 14 Using Assembly**

- Technique 131: C++ Variables Can Be Directly Accessed from Assembly
- Technique 132: Use Inline Assembly Only If You Are Writing for a Specific Processor
- Technique 133: You Don't Need to Use `return` If `eax` Is Already Set
- Technique 134: If You Set `eax` from Assembly, Disable Warning 4035
- Technique 135: Always Restore Warning Defaults after You Disable Them

## **Chapter 15 General Debugging Stuff**

- Technique 136: What's a Bug?
- Technique 137: Debug Your Design before You Code
- Technique 138: Always Single Step through New Code
- Technique 139: Debug If You Can, Build If You Must
- Technique 140: Debug Data, Not Code
- Technique 141: Know the Code That Is Generated
- Technique 142: Plan for Testing
- Technique 143: Test Early, Test Often
- Technique 144: Test under Stress Conditions
- Technique 145: Test with Edge Conditions
- Technique 146: Test from the User's Perspective
- Technique 147: Test with Other Applications
- Technique 148: Test on All the Target Platforms
- Technique 149: Test Retail and Debug Versions

## **Chapter 16 Specific Debugging Stuff**

- Technique 150: Loading DLLs for Debugging
- Technique 151: Loading Executables for Debugging
- Technique 152: Casting Data
- Technique 153: Dumping Registers and What They Mean
- Technique 154: Switching to Hex Display
- Technique 155: Handling First-Chance Exceptions
- Technique 156: Break on Data Changed
- Technique 157: Skipping Execution to a Particular IP
- Technique 158: Creating and Using a MAP File
- Technique 159: Walking the Call Stack
- Technique 160: Fixing the Call Stack If You Can't See Everything

## **Part II Sample Code**

**Chapter 17 Smart Pointers**

**Chapter 18 Reference Counting**

**Chapter 19 Dynamic Arrays**

**Chapter 20 Strings**

**Chapter 21 Bit Manipulation**

**Chapter 22 Sorting**

**Chapter 23 Regular Expression Matching**

**Appendix About the CD-ROM**



<http://www.springer.com/978-1-893115-04-0>

Mike and Phani's Essential C++ Techniques

Hyman, M.; Vaddadi, P.

1999, X, 240 p. With online files/update., Softcover

ISBN: 978-1-893115-04-0

A product of Apress