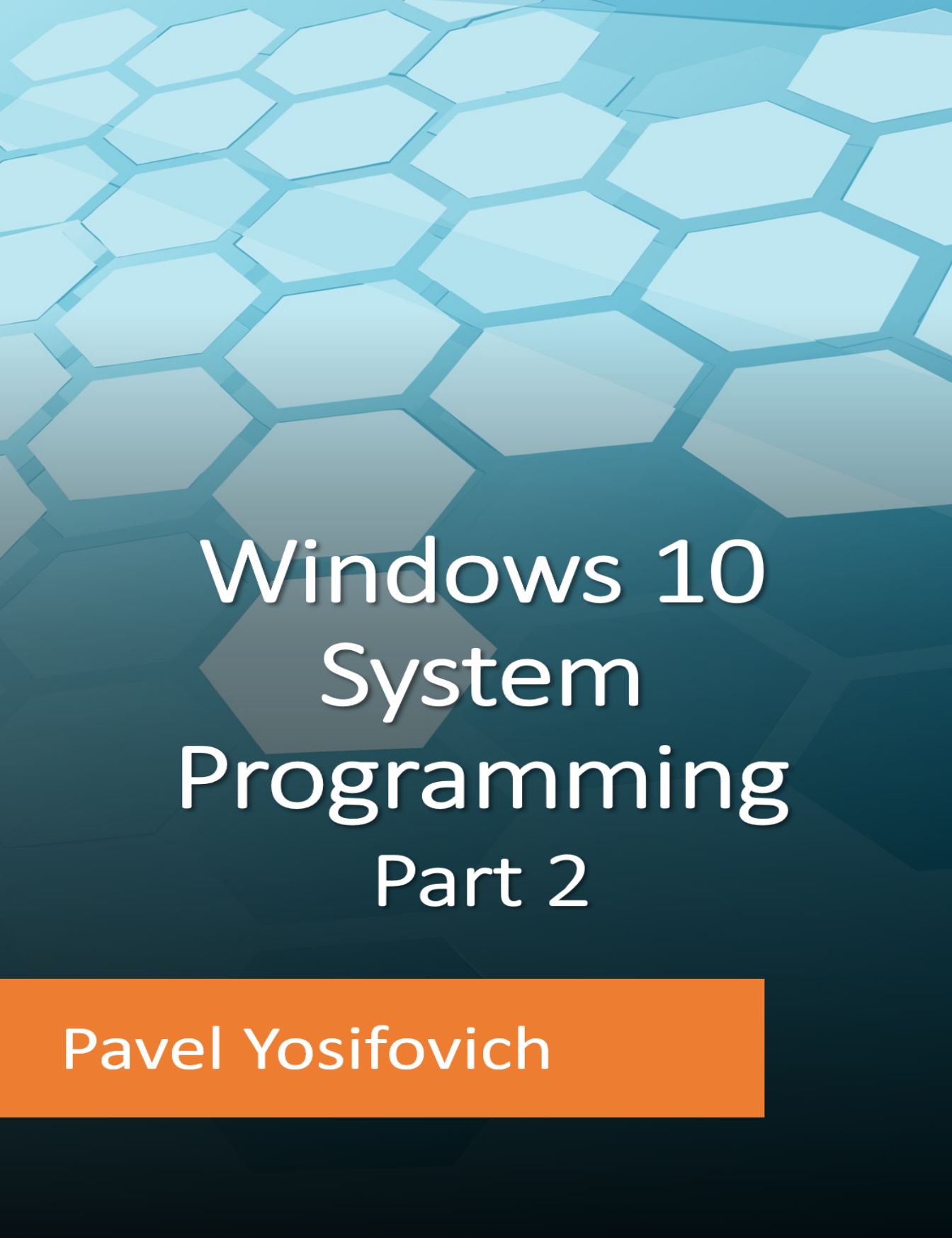
Windows 10
System
Programming
Part 2
Pavel Yosifovich

Windows 10 System Programming, Part 2

Pavel Yosifovich

This book is available at https://leanpub.com/windows10systemprogrammingpart2

This version was published on 2025-10-19

Contents

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Who Should Read This Book

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

What You Should Know to Use This Book

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Sample Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 13: Working With Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Memory APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The `VirtualAlloc*` Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Decommitting / Releasing Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Reserving and Committing Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *Micro Excel* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Working Sets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *Working Sets* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Heaps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Private Heaps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Heap Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Heap Debugging Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The C/C++ Runtime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Local/Global APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Other Heap Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Other `Virtual` Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Memory Protection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Locking Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Memory Block Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Memory Hint Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Writing and Reading to/from Other Processes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Large Pages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Address Windowing Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

NUMA

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The `VirtualAlloc2` Function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 14: Memory Mapped Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Mapping Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *filehist* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Sharing Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Sharing Memory with File Backing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *Micro Excel 2* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Other Memory Mapping Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Data Coherence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 15: Dynamic Link Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Building a DLL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Implicit and Explicit Linking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Implicit Linking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Explicit Linking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Calling Conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

DLL Search and Redirection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The `DllMain` Function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

DLL Injection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Injection with Remote Thread

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Windows Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

DLL Injecting and Hooking with `SetWindowsHookEx`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

API Hooking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

IAT Hooking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

“Detours” Style Hooking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

DLL Base Address

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Delay-Load DLLs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The `LoadLibraryEx` Function

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Miscellaneous Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 16: Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

WinLogon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

LogonUI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

LSASS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

LsaIso

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Security Reference Monitor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Event Logger

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

SIDs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Tokens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Secondary Logon Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Impersonation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Impersonation in Client/Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Privileges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Super Privileges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Take Ownership

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Backup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Restore

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Debug

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

TCB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Create Token

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Access Masks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Security Descriptors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Default Security Descriptor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Building Security Descriptors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

User Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Elevation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Running As Admin Required

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

UAC Virtualization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Integrity Levels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

UIPI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Specialized Security Mechanisms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Control Flow Guard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Process Mitigations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 17: The Registry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Hives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

HKEY_LOCAL_MACHINE

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

HKEY_USERS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

HKEY_CURRENT_USER (HKCU)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

HKEY_CLASSES_ROOT (HKCR)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

HKEY_CURRENT_CONFIG (HKCC)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

HKEY_PERFORMANCE_DATA

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

32-bit Specific Hives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Working with Keys and Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Reading Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Writing Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Deleting Keys and Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Creating Registry Links

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Enumerating Keys and Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Registry Notifications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Transactional Registry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Registry and Impersonation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Remote Registry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Miscellaneous Registry Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 18: Pipes and Mailslots

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Mailslots

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Mailslot Clients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Multi-Mailslot Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Anonymous Pipes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Command Redirect Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Named Pipes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Pipe Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Pipe Calculator Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Other Pipe Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 19: Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Services Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Service Process Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

A Simple Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Installing the Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

A Service Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Controlling Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Installing a Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Starting a Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Stopping a Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Uninstalling the Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Service Status and Enumeration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *enumsvc* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Service Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Service Description

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Failure Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Pre-Shutdown Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Delayed Auto-Start

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Trigger Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Preferred NUMA Node

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Launch as PPL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Debugging Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Interactive Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Service Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Service SID

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Service Security Descriptor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Per-User Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Miscellaenous Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 20: Debugging and Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Debugger Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *DebugPrint* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Performance Counters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Working with Counters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *QSlice* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Process Snapshots

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Querying a Snapshot

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The *snapproc* Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Event Tracing for Windows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Creating ETW Sessions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Processing Traces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Extended Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Real-Time Event Processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Kernel Provider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

More ETW

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

ETW Filters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Session Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Active Sessions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Trace Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Publishing Events with Trace Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Debuggers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

A Simple Debugger

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

More Debugging APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Writing a Real Debugger

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 21: The Component Object Model

The *Component Object Model* (COM) technology made its official debut around 1992/93, initially called *OLE 2.0* and later renamed to *COM*, to better underline the fact that its use cases go much beyond an enhanced implementation for *Object Linking and Embedding* (OLE), most often used within the Microsoft Office suite of applications.

COM became the de-facto standard for object communication based on a binary protocol, which means COM servers and clients could be written in any programming language that adheres to the COM specification. COM is one of the most influential technology in the Microsoft technology stack, but it is also one of the most misunderstood.

In this chapter, we'll take a look at the fundamentals of COM, hopefully demystifying it, and see how to use it from a client and server perspective. At the end of the chapter, you should have the knowledge required to work with existing COM components and create your own when needed.

> This chapter is by no means exhaustive, as complete books have been written about COM.

In this chapter:

What is COM?

Whenever a new technology is created, it's because there are certain problems it aims to fix. COM is no different; what was COM trying to fix? Here are a few scenarios encountered while developing applications that are not easy to handle.

A developer wants to write a library, packaged as a DLL, to provide some functionality. What language should the library be written in? If it's written in C, it can be (relatively) easily consumed by a C or C++ client for sure, but what about other languages/platforms? C is considered the lowest common denominator, and facilities to consume it are available in most languages/platforms, such as .NET, Java, Python, Rust, and many others. But what if the developer writes the library in C++ that exposes a set of C++ classes? It's easy enough to consume in C++, but what about other languages/platforms? It's virtually impossible because C++ works on the source level rather than the binary level.

Even with C++, life is not ideal. The way to expose a C++ class in a DLL is to provide a LIB file that would be linked by the client. That forces the DLL to be loaded when the client process launches, rather than when it's needed. And if the DLL cannot be located, the process terminates, rather than having the opportunity to handle the failure gracefully. As we saw in chapter 15, it's practically impossible to dynamically link to a C++ DLL with calls to `LoadLibrary` and `GetProcAddress`.

it's possible to circumvent the dynamic loading problem by using Delay-Load DLLs, as described in chapter 15.

Here is another example: a developer wants to implement some C++ classes that are hosted in their own process. In this case, the client and server are different processes. How would the client gain access to the server's library functionality? Clearly, some form of inter-process communication (IPC) is required here, but how is that to be implemented so that it's easy for the client to consume the library, and easy for the server to expose that functionality?

Another issue is related to component evolution. Suppose that a C++ class is exposed by some library implemented in a DLL. At some point, a developer wants to extend the class with new functionality or even fix a bug or enhance a feature. Standard C++ rules dictate that if you don't change the public methods of a class, then you're good to go, as clients of your class do not need to make any source code changes. Here is a simple C++ class to serve as an example:

```
// RPNCalculator.h

class RPNCalculator {
public:
    RPNCalculator();
    void Push(double value);
    double Pop();
    bool Add();
    bool Subtract();
```

```
private:
    std::stack<double> _stack;
};
```

This class implements a *Reverse Polish Notation* (RPN) calculator, where values are pushed onto a stack, and calculations are performed by calling the relevant method (e.g. `Add`). The method pops two values off the stack, performs the calculation, and then pushes the result onto the stack. The stack is implemented in this example using the C++ standard library `std::stack<>` adapter class.

Suppose this class is properly implemented and packaged in a DLL. A client working with the calculator can do so with code like the following:

```
#include "RPNCalculator.h"

void SimpleCalc() {
    RPNCalculator calc;
    calc.Push(10);
    calc.Push(20);
    calc.Add();

    // should output 30 (and the stack is empty)
    printf("Result: %lf\n", calc.Pop());
}
```

So far, so good. Now suppose the developer of the `RPNCalculator` class realizes that working with any instance is not thread-safe, as the `std::stack<>` class is not thread-safe. The developer decides to add synchronization support by adding a `CRITICAL_SECTION` object as a private member, like so:

```
class RPNCalculator {
public:
    RPNCalculator();
    void Push(double value);
    double Pop();
    bool Add();
    bool Subtract();

private:
    std::stack<double> _stack;
    CRITICAL_SECTION _cs;
};
```

The relevant method implementations are updated to use the critical section internally. The developer recompiles the DLL and hands it off to the client developer. The client developer replaces his copy of the DLL (the old one) with the new updated version. From a C++ perspective, everything should be fine, since

the public methods in the class have not changed. Now the client application runs (without recompiling the code, as the public stuff has not changed). Can you guess what happens when the client app runs and the `SimpleCalc` function executes?

If you guessed "something bad" or "memory corruption", then you're absolutely correct. Can you spot the problem? Suppose the constructor is implemented in this way:

```
RPNCalculator::RPNCalculator() {
    ::InitializeCriticalSection(&_cs);
}
```

Fairly simple, but this causes a memory corruption. This is because the client code was **not** recompiled, so the size of an `RPNCalculator` object is the size of `std::stack<double>` - the client does not know of the `CRITICAL_SECTION` member, which causes the updated class code to trample stack memory. (if the allocation was dynamic, this would cause a heap corruption). This all happens because the C++ language has no binary compatibility, only source compatibility. We cannot use a "plug & play" approach, where a DLL can simply be replaced by a newer version, with the new functionality available without a hitch if the public surface of the class remains intact.

> There are other issues when using C++ for evolving components, but the above list should suffice as motivation for developing something better.

COM aims to solve these problems, by providing a binary standard of communication. This means the communication protocol between client and server is not based on the semantics of any specific programming language or platform; instead, communication is facilitated by defining some binary object layout that can be implemented by (theoretically) any language or platform, by adhering to the COM specification.

COM has many features and aspects, but it's built on top of two fundamental principles:

- Clients communicate with server objects using (abstract) interfaces, not concrete classes.
- Location transparency - the client does not need to know where a COM class implementation resides. Once the client obtains an interface pointer, it just makes calls and that's it. The server object may be in the same process, a different process on the same machine, or even a process on another machine (this is known as *Distributed COM* or DCOM, although it's not really different from standard COM).

Some of the terms used above need a more precise definition:

- *COM Interface* - a binary contract consisting of a set of methods with a well-defined binary layout.
- *COM Class* - an implementation of one or more COM interfaces.
- *COM Object* - an instance of a COM class.
- *COM Component/Server* - a deployment binary, consisting of one or more COM classes.

A COM Server can be *in-process* or *out-of-process*, depending on whether it's a DLL (loaded into the client's process) or an executable (launched in a separate process). Clients should be able to communicate with COM objects in either way. The in-process scenario is easier to understand, and is depicted in figure 21-1.

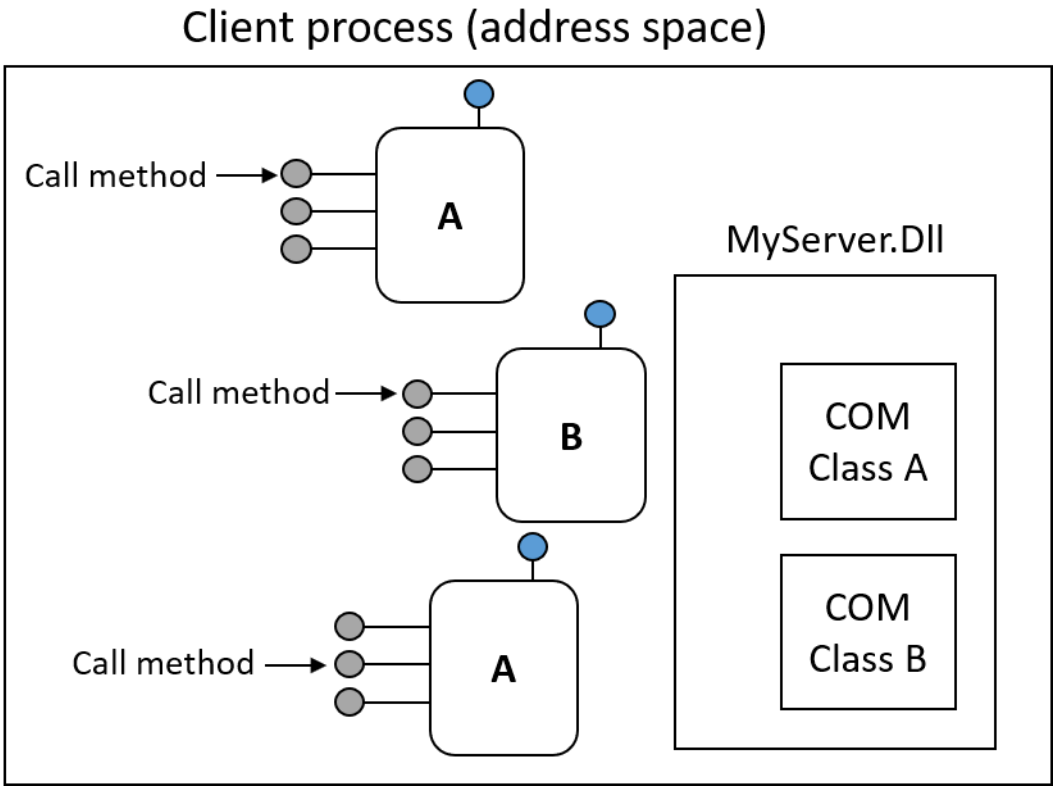

Figure 21-1: DLL (in process) COM Server

The *MyServer.Dll* implements two COM classes, A and B. From these implementations, three objects are created by the client: two instances of A and one instance of B. Then the client calls methods (invoking functionality) on one or more interfaces exposed by these objects. We have yet to discuss how the DLL is loaded, how objects are created, etc. We'll explain all these details as the chapter progresses.

In the out-of-process scenario, the COM infrastructure uses a *proxy* and *stub* object pair to facilitate transparent communication between client and server. The proxy is the server object's representation in the client's address space, while the stub is listening on the object's (server) side for method invocation requests. This layout is depicted in figure 21-2.

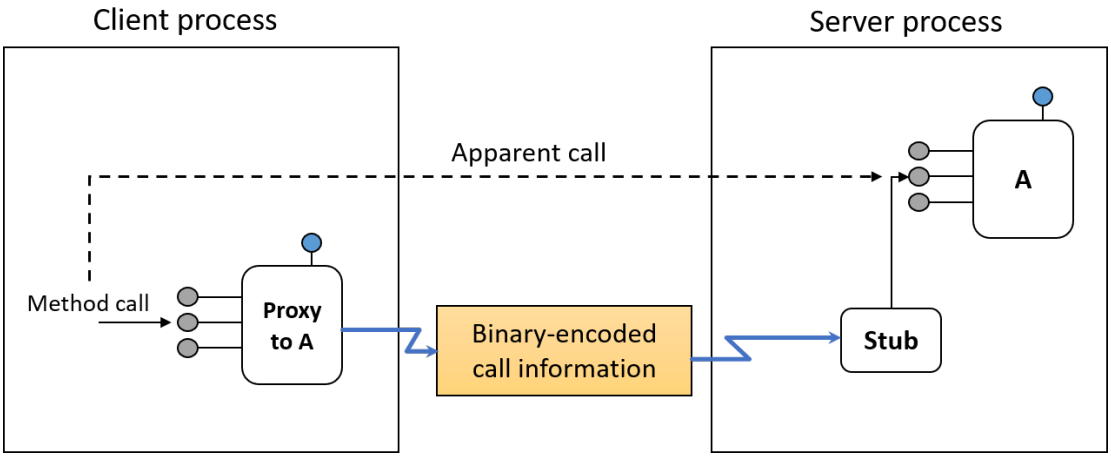

Figure 21-2: EXE (out of process) COM Server

Figure 21-2 demonstrates the location transparency property of COM. The client gets an interface pointer that looks like the real object - the proxy implements all the interfaces implemented by the real object - the client cannot (easily) and need not distinguish between a proxy and a real object. The client invokes methods normally (on the proxy). The proxy's job is to *marshal* the arguments of the method to the other side, so that the receiving end (the stub) can *unmarshal* the arguments and call the real object. This *marshaling* procedure is also performed in reverse for values that need to be returned back to the client. All this marshaling behavior is provided by the "magic" of COM out-of-process invocation.

Interfaces and Implementations

The most fundamental entity in COM is the interface. A COM interface is a binary contract, meaning that once defined and exposed to the clients, an interface should be immutable. This means it should never again change, as existing clients depend on the binary layout and semantics of the interface. If a change or extension is to be introduced, it must be done by defining a new interface (it may inherit all the methods from the original interface or by creating a completely new one).

In C++, a COM interface definition consists of a class with all functions defined as pure virtual. Here is an example of an interface that could be used with an RPN calculator implementation:

```
struct IRPNCalculator {
    virtual void Push(double value) = 0;
    virtual double Pop() = 0;
    virtual bool Add() = 0;
    virtual bool Subtract() = 0;
};
```

It's customary to name interfaces starting with a capital `I`.

With the above definition, the class implementation can be changed in terms of the interface:

```
class RPNCalculator : public IRPNCalculator {
public:
    void Push(double value) override;
    double Pop() override;
    bool Add() override;
    bool Subtract() override;

private:
    std::stack<double> _stack;
    CRITICAL_SECTION _cs;
};
```

Does the above change solve the crashing problem we observed earlier? Not yet. The next step is to hide the `RPNCalculator` implementation from the client entirely. The header exposed to clients should contain two pieces: the interface definition, and a factory function to create an instance. Here's what that may look like:

```
// RPNCalculatorClient.h

struct IRPNCalculator {
    virtual void Push(double value) = 0;
    virtual double Pop() = 0;
    virtual bool Add() = 0;
    virtual bool Subtract() = 0;
};

extern "C" IRPNCalculator* CreateCalculator();
```

The client has no idea where the interface is implemented or how. Creating an instance is now turned over to the server. This is ideal, since the server knows which implementation to create, and it's always going to have the correct size. Here is a revised client code:

```
void SimpleCalc() {
    IRPNCalculator* calc = CreateCalculator();
    if (calc) {
        calc->Push(10);
        calc->Push(20);
        calc->Add();
        printf("Result: %f\n", calc->Pop());

        // not ideal (see later)
```

```
        delete calc;
    }
}
```

The factory is defined with C linkage, because as established earlier, C is the lowest common denominator that is supported for exporting functions by literally all languages/platforms.

This is almost perfect. If the implementation changes, for example by adding or changing data members in the behind-the-scenes implementation, the client should not be affected. This is because the binary layout of the interface remains the same. With a C++ implementation, the virtual table mechanism is used to implement virtual functions, which provide the exact layout defined by a COM interface, making C++ a natural choice for COM class implementations. This layout is depicted in figure 21-3.

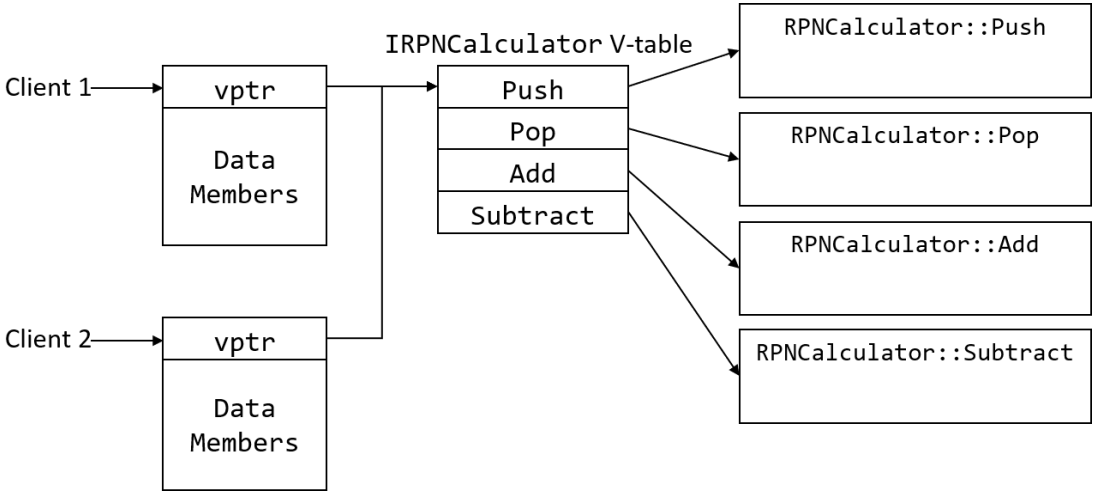

Figure 21-3: Virtual dispatch mechanism

All the client sees is the virtual table pointer (`vptr`) that is always the first member of any instance with virtual functions. The data members of the implementation are unknown (hidden) to the client. In fact, there is no way for the client to query any implementation details, such as the size of the object. This is great, as the implementation can be changed freely without any need for the client to recompile anything, so long as the interface remains the same - the same function order and the same parameters (the function names themselves mean nothing as the invocation is based on the offset of the function pointer within the v-table).

The above `SimpleCalc` still has one snag. To free the object, it calls the C++ `delete` operator. This assumes the object's memory was allocated with the C++ `new` operator.

> Using `delete` in this way has even more hidden assumptions: the client and server must use the same compiler. Both use the same C++ runtime library (dynamic vs.static), and that there is no operator overloading for `new` and `delete` in the server's class implementation).

This assumption (or assumptions) are too problematic. The solution is to transfer the responsibility of freeing the object back to the server (just as was done with object creation). This could be done by adding another method to every COM interface like so:

```
struct IRPNCalculator {
    virtual void Release() = 0;
    virtual void Push(double value) = 0;
    //...
};
```

All the client needs to do is call the `Release` method and not have to know anything about how the object's memory was allocated.

The `IUnknown` Interface

The previous section looked at some problematic details of the C++ language, and how separating interface from implementation could solve these issues. Now it's time to introduce the "official" COM definitions that are based on the principles outlined in the previous section.

COM defines a base interface, from which all interfaces must derive (extend). This ensures certain functionality is always available given any COM interface. This interface is called `IUnknown` and defined like so:

```
struct IUnknown {
    virtual HRESULT __stdcall
        QueryInterface(const IID& riid, void** ppv) = 0;
    virtual ULONG   __stdcall AddRef() = 0;
    virtual ULONG   __stdcall Release() = 0;
};
```

COM defines semantics for managing an object's lifetime. Instead of just providing a way to create and destroy an object, two methods are defined on `IUnknown` - `AddRef` and `Release` to manage the object's reference count (and indirectly - its lifetime). Whether the class implementation actually uses reference counting or not is no concern of COM, but the rules are clear: if the client receives an interface pointer, it must eventually call `Release`. `AddRef` may be used to artificially increment the object's reference count before passing the interface pointer to an independent entity (such as a separate thread of execution). In this way, each client works with the object safely until the pointer is no longer needed. Then the client calls `Release` on its pointer, and from that point on the pointer should be considered poison. The object may or may not be destroyed, but that should not matter to the particular client.

> All COM interface methods must use the standard calling convention (`__stdcall`). This is required as part of the binary interface - the choice seems arbitrary, but nevertheless some choice must be made so that clients and servers are in sync.

The first function in `IUnknown`, `QueryInterface` is concerned with querying the object for another interface that may or may not be supported by the object. Identifying interfaces, like most other COM entities, is done with Globally Unique Identifiers (GUIDs, also called Universally Unique Identifiers - UUIDs). These 128-bit numbers are generated by an algorithm that statistically guarantees uniqueness across time and space. Since 128-bit numbers cannot yet be represented in C/C++ as simple types, the `GUID` structure is defined to hold such a value. It has several alternative `typedef`s such as `IID` and `CLSID`, that have slightly different semantic meanings, but are otherwise identical from the binary perspective:

```
typedef struct _GUID {
    unsigned long  Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char  Data4[ 8 ];
} GUID;
```

GUIDs can be generated programmatically as needed by calling `CoCreateGuid`:

```
HRESULT CoCreateGuid(_Out_ GUID* pguid);
```

You might encounter some strange macros in the Windows header files, such as `FAR` that expand to nothing. This is a relic from 16-bit Windows, where there were near and far pointers.

Visual Studio provides a tool called *Create GUID*, normally accessible from the *Tools* menu, that calls `CoCreateGuid` and provides several formatting options (figure 21-4).
Each time you click *New GUID*, a new GUID is generated anf formatted. Clicking *Copy* copies the selected format to the clipboard. We'll use this tool later to generate GUIDs for COM components we'll author.

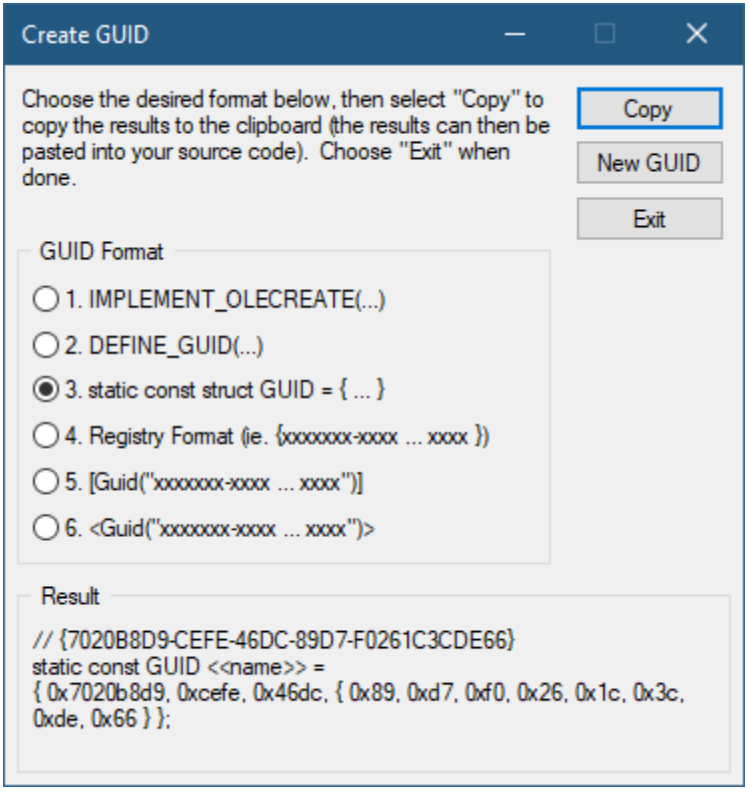

Figure 21-4: The Create GUID tool

GUIDs are used everywhere in COM. For example, interfaces are identified by GUIDs, as using strings such as "IRPNCalculator" are not globally unique. This means that the `IUnknown` interface has its own GUID, defined in the earliest days of COM. It's identified in the Windows headers with the variable `IID_IUnknown`.

This brings us back to the `QueryInterface` method from `IUnknown`. Its first argument is the interface ID to query for (interface names don't mean anything and can be projected differently by other languages/platforms). The result of the query is stored in the second argument (a pointer to a void pointer). The method returns an `HRESULT` - the standard COM return value, indicating success or failure of the operation.

HRESULTs

Most COM methods encountered return an `HRESULT` type. An `HRESULT` is just a signed 32-bit integer that conveys an error or success code. If the most significant bit (MSB) is set, this indicates an error. Looking back at the `IUnknown` interface, it's clear `AddRef` and `Release` do not return `HRESULT`s. These methods should be the only exceptions.

The COM standard specifies that `AddRef` and `Release` should return zero if the object is destroyed, and non-zero value otherwise. The return value is not required to be the actual reference count of the object, but often is. It follows that `AddRef` should never return zero, as it's only possible call `AddRef` on a live object.

The standard `HRESULT` success code is `S_OK` (0). `QueryInterface` returns `S_OK` if the requested interface is supported (and its pointer returned in `*ppv`, otherwise it returns `E_NOINTERFACE`, indicating the interface is not supported. Some of the common error codes are shown in table 21-1.

Table 21-1: Common failure `HRESULT`s

`HRESULT` symbol	Description
`E_NOINTERFACE`	Interface is not supported
`E_POINTER`	Pointer is not valid (typically `NULL` when it shouldn't be)
`E_UNEXPECTED`	Unexpected call to this method at this time
`E_NOTIMPL`	Functionality is not implemented
`E_OUTOFMEMORY`	Not enough memory to complete the operation
`E_INVALIDARG`	One or more invalid arguments to the call
`E_FAIL`	Generic failure

The general layout of an `HRESULT` contains three parts: the *facility* that identifies the category of result, the error or success, and the actual code. The `HRESULT` macros are built based on these three parts in the following way: `FACILITY_S/E_CODE`. The facility for all the values in table 21-1 is `FACILITY_NULL`, that does not show up in the named constants.

Checking for an `HRESULT` is typically done using the `FAILED` or `SUCCEEDED` macros. These return `true` on a failed or successful `HRESULT`, respectively.

More specific `HRESULT` values are defined in the SDK headers, and new ones may be defined by components as needed. If a new `HRESULT` is to be defined, bit 29 should be set for custom `HRESULT` values. Here is a ficticious example for querying for some custom interface and handling an error:

```
void DoWork(IUnknown* p) {
    ICalculator* pCalc;
    HRESULT hr = p->QueryInterface(IID_ICalculator,
        reinterpret_cast<void**>(&pCalc));
    if(SUCCEEDED(hr)) {
        // successful call to QueryInterface increments
        // the object's reference count
        int result;
        hr = pCalc->Add(4, 6, &result);
        if(FAILED(hr)) {
            printf("Failed in call to Add (0x%08X)\n", hr);
```

```
        }
        // done working with the interface
        pCalc->Release();
    }
    else {
        printf("Interface not supported\n");
    }
}
```

You may wonder why `reinterpret_cast` is needed in the above code. Can't we cast `ICalculator**` to `void**` implicitly? The answer is no as these types have no inheritance relationship, so the cast is necessary. If you feel lazy, you can use a C-style cast `(void**)` instead of `reinterpret_cast`.

COM Rules (pun intended)

There are several rules concerning reference counting and `IUnknown` that are worth listing explicitly:

- Whenever an interface pointer is returned via `QueryInterface`, the client should assume the reference count of the object was incremented, meaning the client must eventually call `Release` on that interface pointer, or else that object will leak.
- `QueryInterface` implementations must be symmetric: if it's possible to query for interface `IX` from `IY`, it should be possible to go the other way around.
- It should always be possible to query for the same interface calling `QueryInterface`.
- If one can get from `IX` to `IY` and from `IY` to `IZ`, then it should also be possible to directly request `IZ` from an `IX` interface pointer.
- The `IUnknown` pointer serves as the object's identity. This means an object should always return the same `IUnknown` pointer, no matter from which interface it's requested.
- All COM interfaces must inherit from `IUnknown`, directly or indirectly. This means that the first 3 methods of every interface are `QueryInterface`, `AddRef`, and `Release`, in that order.

Our early `IRPNCalculator` interface was not COM compliant. Let's turn it into a proper COM interface:

```
#include <Unknwn.h>  // including <Windows.h> also includes this one

struct IRPNCalculator : IUnknown {
    virtual HRESULT __stdcall Push(double value) = 0;
    virtual HRESULT __stdcall Pop(double* value) = 0;
    virtual HRESULT __stdcall Add() = 0;
    virtual HRESULT __stdcall Subtract() = 0;
};
```

The interface has seven methods, including those inherited from `IUnknown`. Ordinary return values (such as from `Pop`) turned into parameters with an extra level of indirection (`double` turned into `double*`) because of the requirement to return `HRESULT` from every method.

There is still something missing from the interface - its GUID. We'll get to that in the section "COM Servers", later in this chapter.

COM Clients

We now have enough information to start working with COM as clients, which is considerably easier than creating a COM server. The following example will use the *Background Intelligent Transfer Service* (BITS) API to download a file in the background. BITS is implemented as a Windows Service (see chapter 20 for more on services), but that fact is not really relevant to the way a BITS client invokes BITS functionality thanks to the location transparency property of COM, as we shall see.

We'll start with a standard C++ console application (named *BitsDemo* in the accompanying source code). First, we'll add the usual includes and an extra one for the BITS API:

```
#include <Windows.h>
#include <stdio.h>
#include <Bits.h>       // BITS API
```

The steps we need to take to download a file using the BITS service are outlined below:

1. Initialize COM for this thread.
2. Create an instance of the BITS manager.
3. Create a BITS job for download.
4. Add a URL to download from and local file to store the resulting file.
5. Initiate the transfer.
6. Wait for the transfer to complete.
7. Display final results.
8. Release the various objects created in the previous steps.

Most of the above steps are specific to the way the BITS API is to be used, but some of the steps are generic and applicable to all COM clients.

Step 1: Initialize COM

Before making any COM-related API call from a thread (there are very few exceptions to this rule), COM must be initialized for that thread. This can be done with a call to one of the following functions: `CoInitialize` or the extended `CoInitializeEx`:

```
HRESULT CoInitialize(_In_opt_ LPVOID pvReserved);
HRESULT CoInitializeEx(
    _In_opt_ LPVOID pvReserved,
    _In_ DWORD dwCoInit);
```

`CoInitialize` can only accept `NULL`, and in fact is just special case of `CoInitializeEx`. It's equivalent to calling `CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED)`.

Although the term "initializing COM" may be good enough to proceed to the next step, it would be more accurate to say that the function puts the calling thread into an apartment whose type depends on the `dwCoInit` parameter (and is always a single-threaded apartment in the `Coinitialize` case). A comprehensive discussion of apartments is reserved for the section "Threads and Apartments". For now, we'll just call one of the functions as the first line inside `main` and move on to step 2:

```
::CoInitialize(nullptr);
```

Step 2: Create the BITS Manager

As with any API, proper usage requires reading the API's documentation. The next steps are based on the BITS API official documentation. Accessing BITS functionality requires creating an instance of a COM class called the *Background Copy Manager*. In general, creating instances of COM classes is achieved (in most cases) by calling `CoCreateInstance`:

```
HRESULT CoCreateInstance(
    _In_ REFCLSID rclsid,
    _In_opt_ LPUNKNOWN pUnkOuter,
    _In_ DWORD dwClsContext,
    _In_ REFIID riid,
    _COM_Outptr_ LPVOID* ppv);
```

> COM object creation is also referred to as *Activation*.

The prefix `Co` that most COM APIs use stands for "Component Object".

The purpose of `CoCreateInstance` is to create an instance of a given class and return a requested interface pointer to the new object. The first parameter identifies the class itself with a GUID, referred to here as class ID (CLSID), to make it easier to understand, but it's a GUID like any other. COM classes - implementations of COM interfaces - are identified by GUIDs, giving them unique names. The CLSID is looked up in the Windows Registry, as we shall see in the next section. The interface ID (supplied as the fourth parameter) is not good enough, as one interface can have any number of implementations.

The second parameter to `CoCreateInstance` is an `IUnknown` pointer called "outer IUnknown". This parameter is related to a COM extensibility mechanism called *Aggregation*. Aggregation is beyond the scope of this chapter, and if not used (the typical case), `NULL` is specified.

The next parameter (`dwClsContext`) indicates (mostly) which context the server should be allowed to run in. The most common value is `CLSCTX_ALL`, which means the client doesn't care, and would accept any implementation. Here are some Common alternatives:

- `CLSCTX_INPROC_SERVER` - in-process (DLL) implementation.
- `CLSCTX_LOCAL_SERVER` - out-of-process (EXE) implementation on the same machine as the client.
- `CLSCTX_REMOTE_SERVER` - out-of-process (EXE) implementation on another machine (used with `CoCreateInstanceEx`).

Multiple flags can be specified by using the or (`|`) operator. The function will attempt to get the "closest" implementation (i.e. DLL preferred over EXE, local EXE preferred over a remote server).

The next parameter is the interface ID requested from the new object. This can be `IUnknown` (`IID_IUknown`), which must always be supported, or a more specific interface that is known to be implemented by the class. If `IUnknown` is requested, another interface can later be obtained by calling `QueryInterface`.

The last parameter is the resulting pointer, if the call succeeds. As usual, the return value is an `HRESULT`, that is `S_OK` with a successful call.

Given this information and the BITS docs, we need to make the following call:

```
IBackgroundCopyManager* mgr;
HRESULT hr = ::CoCreateInstance(CLSID_BackgroundCopyManager,
    nullptr, CLSCTX_ALL,
    IID_IBackgroundCopyManager, reinterpret_cast<void**>(&mgr));
if (FAILED(hr))
    return Error(hr);
```

CLSIDs are traditionally prefixed with `CLSID_` and interface IDs with `IID_`. The above GUIDs are defined in the *bits.h* header.

> The last two arguments in the above `CoCreateInstance` call can be shortened by using the `IID_PPV_ARGS` macro like so:
>
> ```
> HRESULT hr = ::CoCreateInstance(CLSID_BackgroundCopyManager,
> nullptr, CLSCTX_ALL, IID_PPV_ARGS(&mgr));
> ```

```
```

`Error` is just a simple function that displays the `HRESULT`:

```
int Error(HRESULT hr) {
    printf("COM error (hr=%08X)\n", hr);
    return 1;
}
```

The process of locating the server is described in the next section. For the purposes of this demo, if we receive back a proper interface, we can move on to step 3.

Step 3: Create a BITS Job

The `IBackgroundCopyManager` interface is defined like so (copied from *bits.h*):

```
MIDL_INTERFACE("5ce34c0d-0dc9-4c1f-897c-daa1b78cee7c")
IBackgroundCopyManager : public IUnknown {
public:
 virtual HRESULT STDMETHODCALLTYPE CreateJob(
  /* [in] */ __RPC__in LPCWSTR DisplayName,
  /* [in] */ BG_JOB_TYPE Type,
  /* [out] */ __RPC__out GUID *pJobId,
  /* [out] */ __RPC__deref_out_opt IBackgroundCopyJob **ppJob) = 0;
 virtual HRESULT STDMETHODCALLTYPE GetJob(
  /* [in] */ __RPC__in REFGUID jobID,
  /* [out] */ __RPC__deref_out_opt IBackgroundCopyJob **ppJob) = 0;
 virtual HRESULT STDMETHODCALLTYPE EnumJobs(
  /* [in] */ DWORD dwFlags,
  /* [out] */ __RPC__deref_out_opt IEnumBackgroundCopyJobs **ppEnum)
    = 0;
 virtual HRESULT STDMETHODCALLTYPE GetErrorDescription(
  /* [in] */ HRESULT hResult,
  /* [in] */ DWORD LanguageId,
  /* [out] */ __RPC__deref_out_opt LPWSTR *pErrorDescription) = 0;
};
```

At first glance, this doesn't appear to be human-written code - it looks machine-generated. And indeed it is, as can be seen from reading the first line in *bits.h*:

```
/* this ALWAYS GENERATED file contains the definitions for the interface\
s */
```

The file was generated from an *Interface Definition Language* (IDL) file that was compiled by the Microsoft IDL (MIDL) compiler. The reason for using yet another file format for generating interfaces is discussed in the section "The Interface Definition Language", later in this chapter. For now, let's make sure we understand the above interface definition.

The `STDMETHODCALLTYPE` macro expands to `__stdcall`, as is required by all COM methods. All the `__RPC`-something macros are SAL annotations. The `MIDL_INTERFACE` is defined like so:

```
#define MIDL_INTERFACE(x)   struct DECLSPEC_UUID(x) DECLSPEC_NOVTABLE
```

Continuing with the remaining macros:

```
#define DECLSPEC_UUID(x)    __declspec(uuid(x))
#define DECLSPEC_NOVTABLE   __declspec(novtable)
```

`MIDL_INTERFACE` defines an interface by using the `struct` keyword (as C++ has no special keyword for interfaces), and uses two Visual C++ specific attributes. `uuid(x)` associates a GUID with the definition, which helps to simplify code by not requiring usage of an `IID_IX` variable, instead opting for the more elegant `__uuidof(IX)` operator that uses the attached GUID. This can also be used by classes, by the way, as it is with the Background Copy Manager:

```
class DECLSPEC_UUID("4991d34b-80a1-4291-83b6-3328366b9097") BackgroundCo\
pyManager;
```

All this means that the initial `CoCreateInstance` code can be simplified like so:

```
HRESULT hr = ::CoCreateInstance(__uuidof(BackgroundCopyManager),
    nullptr, CLSCTX_ALL,
    __uuidof(IBackgroundCopyManager), reinterpret_cast<void**>(&mgr));
```

All this is just compiler trickery and does not have any effect at runtime.

The macro `IID_PPV_ARGS` uses the `__uuidof` operator, without which it cannot "get" the interface ID.

A new BITS job is created by calling `IBackgroundCopyManager::CreateJob`, rather than requiring its own `CoCreateInstance` call. This is a very common pattern, as it allows the creation function to accept any extra parameters needed and provides control over instance creation (and does not require any Registry settings).

```
GUID jobId;
IBackgroundCopyJob* pJob;
hr = mgr->CreateJob(L"My job", BG_JOB_TYPE_DOWNLOAD, &jobId, &pJob);
if (FAILED(hr))
    return Error(hr);
```

The call to `CreateJob` returns a new interface pointer on a BITS job object. It also returns a GUID identifying this job. This GUID has nothing to do with COM, and is used internally by BITS to uniquely identify transfers. The inputs are a display name, which can be anything, and whether the job is a download or upload.

We can now technically release the interface pointer to `IBackgroundCopyManager` - we won't be needing it again:

```
mgr->Release();
```

Remember, this doesn't necessarily mean the object is destroyed. It's possible (and in fact probable) that the job object holds another interface pointer to the BITS manager. Regardless, as clients, we don't care. Once we don't need an interface pointer we received earlier, we release it.

Step 4: Add a Download

The next step is to add at least one file to download:

```
hr = pJob->AddFile(
    L"https://www.fnordware.com/superpng/pnggrad16rgb.png",
    L"c:\\temp\\image.png");
if (FAILED(hr))
    return Error(hr);
```

The first parameter is the remote URL to download from (or upload to if this was an upload job). The second parameter is the local file to which the remote is to be downloaded.

I've selected some fairly random image URL, that may or may not continue to work by the time you read this. If this does not work, find some other file to download from the web.

We can add more files if we want for the same job, but in this example we'll continue with just one.

Step 5: Initiate the Transfer

Starting the transfer is fairly easy with a call to `IBackgroundCopyJob::Resume`:

```
hr = pJob->Resume();
```

BITS works asynchronously, so `Resume` should start the download and return immediately. We need to know when the transfer is complete (or there is some error). BITS provides two ways to do this: synchronously and asynchronously. We'll use the synchronous option here, and discuss the asynchronous option later in this chapter, as it requires us to implement a callback interface.

Step 6: Wait for Transfer to Complete

We'll have a loop that polls the job object every some interval and query the state of the transfer, exiting the loop when it's done:

```
if (SUCCEEDED(hr)) {
    printf("Downloading... ");
    BG_JOB_STATE state;
    for (;;) {
        pJob->GetState(&state);    // assume it cannot fail
        if (state == BG_JOB_STATE_ERROR
            || state == BG_JOB_STATE_TRANSFERRED)
            break;
        printf(".");
        ::Sleep(300);
    }
```

`IBackgroundCopyJob::GetState` returns the state of the transfer. The code waits for a successful completion or some error to break out of the loop. It uses a `Sleep` call to prevent CPU hogging, as this is a network transfer.

Step 7: Display Results

Once out of the loop, we can show the results:

```
    if (state == BG_JOB_STATE_ERROR) {
        printf("\nError in transfer!\n");
    }
    else {
        pJob->Complete();
        printf("\nTransfer successful!\n");
    }
}
pJob->Release();
```

The call to `Complete` is required, as BITS stores the downloaded file with a temporary name generated by BITS. The call to `Complete` flushes any remaining bytes and renames the file to the client-provided name. Finally, the job interface pointer is released.

Step 8: Clean Up

We already released the manager and job interfaces. There is very little left to do - just uninitialize COM before the thread exits:

```
    ::CoUninitialize();
    return 0;
}
```

Run the application and you should see an output like the following:

```
Downloading... ..............
Transfer successful!
```

If some error occurs, you should get an error output (one simple way to get this is to modify the URL to a non-existing one):

```
Downloading... ...
Error in transfer!
```

If an error occurs, BITS provides extended error information by calling `IBackgroundCopyJob::GetError`. Add code to display rich error information in case of a failure with the transfer. Remember to follow COM rules regarding interface pointers.

COM Smart Pointers

The rules concerning `AddRef` and `Release` calls are not complicated when viewed in isolation, but in practice it's difficult to keep track of all interface pointers going around. As a consequence, it's easy to miss calling `Release` especially when an interface pointer is no longer needed in all code paths using that pointer. This is where COM smart pointers come in, automating calls to `AddRef` and `Release` so that explicit calls to these methods is rarely needed.

As a bonus, these smart pointers also provide easier access to `QueryInterface` by overloading constructors and the assignment operators. Several COM smart pointers are available in the Windows SDK today:

- *<comdef.h>* has definitions for smart pointers based on a class named `_com_ptr_t,.`. These throw C++ exceptions for failed `QueryInterface` calls.
- ATL provides two smart pointer classes that don't ever throw exceptions.
- The newer *Windows Runtime Library* (WRL) provide its own version of smart pointers (`ComPtr<>`), that are more verbose than the ATL ones.

Which smart pointer class you use is mostly a matter of taste. As this book uses ATL and WTL, I will demonstrate the usage of the ATL smart pointers, which I also personally prefer for their convenience and simplicity.

The two classes most often used are `CComPtr<>` and `CComQIPtr<>`, the latter having extra constructors that call `QueryInterface` when faced with a different typed interface.

What makes a pointer "smart"? In the COM case, it's about convenient constructors and destructor, that call `AddRef`, `QueryInterface` and `Release` as appropriate, and operator overloading for dereferencing (`*`, `->`) and address-of (`&`), as we shall see in the next code snippet.

The previous section used raw pointers (sometimes dubbed "stupid pointers") to access BITS functionality. Here is the equivalent code using the ATL smart pointers:

```
#include <atlcomcli.h>

HRESULT DoBITSWork() {
    // assume CoInitialize has already been called for this thread

    CComPtr<IBackgroundCopyManager> spMgr;
    HRESULT hr = spMgr.CoCreateInstance(
        __uuidof(BackgroundCopyManager));
    if (FAILED(hr))
        return hr;

    CComPtr<IBackgroundCopyJob> spJob;
    GUID guid;
    hr = spMgr->CreateJob(L"My Job", BG_JOB_TYPE_DOWNLOAD,
        &guid, &spJob);
    if (FAILED(hr))
        return hr;

    hr = spJob->AddFile(
        L"https://www.fnordware.com/superpng/pnggrad16rgb.png",
        L"c:\\temp\\image.png");
    if (FAILED(hr))
        return hr;

    hr = spJob->Resume();
    if (SUCCEEDED(hr)) {
        printf("Downloading... ");
        BG_JOB_STATE state;
        for (;;) {
            spJob->GetState(&state);
            if (state == BG_JOB_STATE_ERROR
```

```
                || state == BG_JOB_STATE_TRANSFERRED)
                break;
            printf(".");
            ::Sleep(300);
        }
        if (state == BG_JOB_STATE_ERROR) {
            printf("\nError in transfer!\n");
        }
        else {
            spJob->Complete();
            printf("\nTransfer successful!\n");
        }
    }
    return hr;
}
```

The function starts by including *<atlcomcli.h>* where the ATL smart pointers are defined. Alternatively, you can include *<atlbase.h>*, which has some useful extras and also includes *<atlcomcli.h>*.

The various interface methods are exposed directly by `CComPtr<>` because the `->` operator is overloaded and returns the internal interface pointer. Notice there are no `Release` calls in the above code. In fact, trying to call `Release` as the following example shows fails to compile:

```
spMgr->Release();
```

This is intentional, as allowing this call to `Release` will likely cause a crash, since the destructor, unaware that `Release` has been called, will attempt another `Release` call, which is one too many. The class *does* provide a `Release` method that can be called so that the interface is released early (before the destructor runs):

```
spMgr.Release();
```

This call invokes the internal interface's `Release` and sets the interface pointer to `NULL` so that the destructor does not call `Release` again (seeing the interface pointer is `NULL`). An equivalent call is setting the object to `NULL` (operator overloading at work):

```
spMgr = nullptr;
```

The original call to `CoCreateInstance` is replaced in the above code by the `CoCreateInstance` method exposed by the `CComPtr<>` class. This is just a helpful shortcut that calls `CoCreateInstance`, but provides the defaults of `NULL` for the outer `IUnknown` and `CLSCTX_ALL` for the class context.

Write a console application that lists all currently active BITS jobs by calling `IBackgroundCopyManager::EnumJobs`. Display each job's display name, state, description, priority, GUID, creation time, and progress. Use smart pointers.

My *BITSManager* tool shows how to accomplish this (https://github.com/zodiacon/BITSManager).

Querying for Interfaces

The BITS job object implements more than the `IBackgroundCopyJob` interface - it also supports the extended interfaces `IBackgroundCopyJob2` and `IBackgroundCopyJob3`. However, these interfaces may or may not be supported based on the BITS version on the machine. This extended functionality was added after the first BITS version was out, so new interfaces had to be defined. The new interfaces may inherit (extend) an existing interface, or be unrelated (inheriting directly from `IUnknown`). This is left to the discretion of those defining the new interface(s).

A client that wants to gain access to the new functionality must query for it and be ready to handle failure gracefully. Here is an example for working with one of the extended job interfaces:

```
// after the job is created (in spJob)

// leading space for SetNotifyCmdLine
WCHAR localPath[] = L" c:\\temp\\image.png";

CComPtr<IBackgroundCopyJob2> spJob2;
hr = spJob->QueryInterface(__uuidof(IBackgroundCopyJob2),
    reinterpret_cast<void**>(&spJob2));
if (spJob2) {    // checking HR is ok too
    hr = spJob2->SetNotifyCmdLine(
        L"c:\\windows\\system32\\mspaint.exe", localPath);
    // interface pointer released here
}

hr = spJob->AddFile(
    L"https://www.fnordware.com/superpng/pnggrad16rgb.png",
    localPath + 1);
//  rest of code is unchanged
```

If you try this out, *mspaint* should come up automatically when the file is downloaded successfully.

The caller must be prepared for the interface not being implemented and handle that appropriately. In some cases, the caller just won't use the new functionality. In other cases, it may be more appropriate to report an error and notify the user that a newer library is required for proper functionality.

The above `QueryInterface` (QI) call uses the real `QueryInterface` method to make the call, providing the interface ID and the address of a pointer to fill in on success. The ATL smart pointers, however, provide simplified ways to achieve the same thing. Here are more options for making the same QI call:

```
// leverages the __uuidof operator
hr = spJob.QueryInterface(&spJob2);

// uses CComQIPtr<>
CComQIPtr<IBackgroundCopyJob2> spJob2(spJob);
if(spJob2) {    // no HRESULT to examine
    // interface available
}
```

Make sure you release an interface pointer if you plan to reuse it for a future create or QI, otherwise you'll get an assertion failure from the ATL smart pointers.

`CoCreateInstance` Under the Hood

`CoGetClassObject`

DLL Activation

EXE Activation

Service Activation

Implementing COM Interfaces

COM Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Implementing the COM Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Implementing the Class Object (Factory)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Implementing `DllGetClassObject`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Implementing Self Registration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Registering the Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Debugging Registration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Testing the Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Testing with non C/C++ Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Proxies and Stubs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

IDL and Type Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Threads and Apartments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The Free Threaded Marshalar (FTM)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Odds and Ends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 22: The Windows Runtime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Working with WinRT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

The `IInspectable` interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Language Projections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

C++/WinRT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Asynchronous Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Other Projections

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Chapter 23: Structured Exception Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Termination Handlers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Replacing Termination Handlers with RAII

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Exception Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Simple Exception Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Using `EXCEPTION_CONTINUE_EXECUTION`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Exception Information

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Unhandled Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Just in Time Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Windows Error Reporting (WER)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Vectored Exception Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Software Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

High-Level Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Visual Studio Exception Settings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.

Book Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/windows10systemprogrammingpart2.