

## C++ Best Practices and Design Patterns: Hands-On - 4 Days

### Course 397 Overview

- You Will Learn How To**
- Design and implement efficient object-oriented solutions using C++
  - Improve C++ code quality and reusability with design patterns and proven idioms
  - Build robust, efficient libraries using namespaces, templates and exceptions
  - Use the standard C++ library, including the Standard Template Library (STL)
  - Leverage third-party tools, class libraries and application frameworks
  - Avoid the subtle traps and pitfalls of C++ programming
- Course Benefits** The ability to leverage the work of others, avoid pitfalls, and apply proven idioms and patterns can greatly improve the effectiveness of programming efforts. In this course, you learn to increase productivity by combining tools, idioms, syntax and libraries to produce industrial-strength C++ code. Numerous hands-on exercises provide real-world experience in developing high-quality C++.
- Who Should Attend** Programmers, software engineers, analysts and designers wishing to develop advanced C++ skills. Previous C++ programming experience is assumed.
- Hands-On Training** Our expert instructors guide you through practical hands-on exercises that reinforce your skills in advanced C++ programming techniques. Learn by doing as you develop complete programs from architectural design to refining the implementation via a series of exercises, including:
- Forwards- and reverse-engineering C++ and UML
  - Improving code quality using design patterns
  - Modifying a working, but poorly structured, application to increase flexibility, robustness and efficiency
  - Applying all the major components of the STL
  - Using namespaces, exceptions and templates to build reusable libraries
  - Debugging and correcting subtle errors

## C++ Best Practices and Design Patterns: Hands-On - 4 Days

### Course 397 Outline

#### Introduction to Object-Oriented Development

##### OO fundamentals

- Inheritance, encapsulation and polymorphism
- Classes, objects and attributes
- Associations, messages and methods
- Interfaces and abstract classes

#### Using the Unified Modeling Language

- Characteristics of UML
- Mapping UML into C++

#### Exploiting development tools

- Automating the life cycle with CASE tools
- Code generation and reverse engineering
- Debuggers and browsers

#### Idioms and Design Patterns

##### C++ idioms

- Handle/body and related idioms
- Functors: functions coded as objects

#### Introducing design patterns

- The motivation for design patterns
- Categories of patterns: creational, behavioural and structural
- Describing design patterns

#### Putting patterns to work

- Synchronising multiple views with the Observer pattern
- Handling recursive data structures with the Composite pattern
- Minimising code duplication with the Template Method pattern
- Managing object creation with the Singleton pattern

#### Using the ISO Standard C++ Library

##### The Standard Template Library (STL)

- The structure of the STL
- Declaring and populating sequence and associative containers
- Accessing containers using iterators
- Applying standard and user-supplied algorithms
- Customising behaviour using function objects and adapters
- Extending the STL

#### The new iostream library

- Basic input/output
- Formatting textual output
- Handling errors in input data
- Wide character types and internationalisation

#### Storage Management

##### Managing memory

- Recognising and reducing memory overhead
- Preventing memory leaks with the **auto\_ptr** template
- Overloading **operator new** and **operator delete**
- Writing and using smart pointers

##### File storage

- Preparing classes for simple file storage and retrieval
- Storing and retrieving objects using Boost serialisation libraries

#### Writing Better C++

##### Increasing reusability

- Avoiding name clashes using namespaces
- Using templates for type-safe reusability

##### Improving robustness

- Strengthening encapsulation by consistent and appropriate use of **const**
- Implementing a coherent exception strategy
- Decoupling algorithms from data structures with the Visitor pattern

##### Enhancing efficiency

- Saving processing and memory with reference counting
- Sharing state between lightweight objects

#### Avoiding C++ Traps and Pitfalls

##### Things you need to do—and why

- Virtual destructors
- Assignment operators and copy constructors

##### Features to handle with care

- Friends vs. public members
- Runtime-type information vs. virtual member functions
- Multiple and virtual inheritance