



# Datasim's Course guide for 2011

Shipluidenlaan 4  
1062 HE Amsterdam  
[www.datasim.nl](http://www.datasim.nl)  
[info@datasim.nl](mailto:info@datasim.nl)

### **About DatasimEducation BV**

Datasim BV was founded in 1987. The main reason for forming the company was to promote the new object-oriented technology that started to find its way into mainstream software projects. In 1989 Datasim organised one of the first C++ courses in the Netherlands and has since the early pioneering days grown to be a market leader in object-oriented and component training. In particular, we offer the full range of training services from the initial requirements phase right up to implementation.

Due to the acceptance of object-oriented technology (OOT) in the latter part of the 1990's and the huge demand for strong support in all areas of OOT we have found it desirable to diversify. To this end, the organisation has been partitioned into specialised companies:

- Datasim Education BV (The Netherlands).
- Datasim Education Ltd. (Ireland).

Datasim Education BV is responsible for all training and mentoring activities in both generic object-oriented and component training as well as the Microsoft range of development products.

Datasim Education Ltd. is responsible for training, consultancy and software activities in the UK and Ireland.

### **The Vision**

Datasim is one of the few companies with in-depth knowledge of both OOT training and software development. Our trainers have practical software experience and there is a close relationship between each of the companies. In this sense we are a real learning organisation.

### **Applications**

Datasim focuses on a number of related domains. This focus can be seen both in our training programmes and the types of applications that we produce for customers. The main attention areas are:

- Engineering (production and process control).
- Computer graphics, visualisation and design.
- Risk management applications.
- Advanced modelling techniques for industry and business.
- Components and reusable software.

The software techniques are based on OO techniques and we

have a proven track record for quality, robust software and on-time project development schedules.

### **Our Customers**

Datasim has trained many software professionals throughout Europe and our customer base consists of the following groups:

- Telecommunication companies.
- Companies in electronics.
- Large, medium and small software houses and development companies.
- Banks and insurance companies.
- Software developers.

### **More Information?**

If you wish to receive more information on any of our services please see our Web-site: <http://www.datasim.nl>

For a personal conversation, please contact:

Dr. Daniel Duffy or Ms. Ilona Hooft Graafland

Datasim Education B.V.

Schipluidenlaan 4

1062 HE Amsterdam

The Netherlands

tel: +31-(0)20-6240055

fax: +31-(0)20-4200075

email: [info@datasim.nl](mailto:info@datasim.nl)

# Contents

## **C and C++ Programming**

- Fundamentals of ANSI C++ 4
- Advanced Object-Oriented and Generic Programming in C++ 5

## **C# Programming**

- C# Programming in .NET 2.0 & 3.x 6
- Advanced C# Programming in NET 2.0 & 3.x 8
- Generic Programming in C# and .NET 2.0 9

## **Design Patterns and Libraries**

- C# and Design Patterns 10
- Standard Template Library 11
- The Boost C++ Library 12
- Advanced Boost Interprocess Communication and Networking Libraries 14

## **Parallel Programming**

- Parallel Design and Programming in C++ and OpenMP 15
- Multi-threaded and Parallel Programming in C# 17
- Programming Generic and Parallel Design Patterns in C++ 18

## **Object-Oriented and Component Technology**

- Requirements Determination 19
- Component and Service Oriented Analysis 20
- Component and Service Oriented Design 22

## **Architecture and Design**

- Advanced Design and System Patterns 23
- Parallel and Multithreaded Design Patterns: Next generation GOF and POSA 24

## **Distance Learning**

- Advanced C++ - Programming Models, Libraries and Parallel Computation 26

# Fundamentals of ANSI C++

This 4-day course is a thorough introduction to the C++ programming language that was invented by Dr. Bjarne Stroustrup. C++ is an extension of the popular C language and it has support for the object-oriented paradigm and generics (in the form of template classes). We introduce the most important programming techniques to help you write flexible, reliable and understandable applications in C++. The approach is to build up your skills in an incremental fashion. This ensures that you learn all the important features in the language and programmer productivity will be improved. When you have completed this course, you will be in a position to write object-oriented C++ code that is reliable and flexible.

This course takes you step-step through all the elements of the C++ language that are the foundations for applications and advanced techniques. The percentage theory/practice is 50/50.

## What do you learn?

- Comprehensive treatment of C++
- Hands-on experience and practice
- Step-by-step learning experience
- Using C++ to create robust and maintainable code

## Prerequisites

The student should have a good knowledge of C. We assume that you know C syntax. In particular, we expect that you have experience programming with pointers, structs, function pointers and other advanced topics in C.

## Who should attend?

Software developers and programmers who wish to get a thorough introduction to C++. C++ is easy if it is done right!

## Course form

The percentage theory/practice is about 50:50. This course is independent of any operating system and the student can choose between Windows and Linux as the system of choice.

What previous Delegates have said about the Course

- "Good, positive. I am now prepared for more!"
- "First class course (prima cursus)"
- "Clear and streamlined"
- "Very good/solid/thorough/complex"
- "Very satisfied"

Course Contents updated January 2010

## Course contents

### The Class Concept

- Abstract Data Types
- Classes
- Objects
- C++ members
- Constructors and destructors
- The C++ header file
- Access specifiers
- The C++ source file
- Creating objects
- Some differences between C and C++

### Improving your Classes

- Function name overloading
- Call by reference vs. call by value
- The 'const' specifier
- The copy constructor

### Basic Operator Overloading

- Introduction to operator overloading
- Binary operators
- Unary operators
- Assignment operators
- The 'this' pointer
- Returning references
- The Canonical Header File

### Some C++ Features

- Inline functions
- Friend classes and friend functions
- Friend operators
- Static member data and functions
- Colon syntax

- Default values

### Memory Management, Fundamentals

- Stack, heap and static memory
- Who takes care of memory?
- The new and delete operators
- Dynamic arrays
- Clases with dynamic memory

### Memory Management, Advanced Topics

- Problems with raw C++ pointers
- When you should not use auto\_ptr
- The boost library and pointers
- Scoped pointers
- Shared pointers and reference counting

### Advanced Overloading

- Pre and postfix operators
- Overloading index operators [] and ()
- The assignment operator and memory management
- Overloading the ostream operator <<
- Functors and function objects
- Comparing functors with function pointers
- Conversions (explicit conversions and conversion operator)

### Simple Inheritance

- Inheritance and ISA Relationship
- Specialisation scenarios
- Inheritance and object creation

- Using base class constructors
- Accessibility of base members
- Overriding functions

### Polymorphism

- Pointers to the base class
- Function visibility
- Polymorphism
- Defining an interface
- Abstract base classes
- Virtual destructors
- Operator overloading and inheritance

### Template Classes in C++

- An introduction to generic programming
- Parameterization
- Template classes and function templates
- Creating templates (source & header files)
- Type template parameter & value template parameter
- Operations on the generic argument
- Explicit template instantiation
- Static data & templates
- Massive reusability: specializing templates
- Combining inheritance and templates
- Composition and delegation
- Default template arguments
- Function templates and overloading
- Explicit template specialisation

- Partial template specialisation

### Exception Handling

- Error handling and exceptions
- Throwing and catching exceptions
- Exception hierarchy
- Polymorphic and explicit casts
- Order of handlers
- Exception specification
- Layering Exceptions

### Namespaces

- Why using namespaces
- Using classes in a namespace
- Placing classes in a namespace
- Nested namespaces
- Aliases
- Name lookups

### An Introduction to STL

- The 6 STL components
- Sequential and associate data containers
- Iterators
- Adaptors
- Algorithms
- Function objects
- Using STL in your applications

### Run Time Type Information

- What is RTTI
- Type\_info class and operator typeid
- Typeid and inheritance
- Casting operators and type-safe casting using RTTI

### C++ 0x: The upcoming C++ standard

- Features already supported in Visual C++ 2010
- auto variables & decltype
- nullptr
- rvalue references and move semantics
- Lambda functions

### Designing Robust Code

- Delegation and composition
- Combining delegation and inheritance
- The pitfalls of multiple inheritance
- Combining the procedural and object-oriented programming styles

## Advanced Object-Oriented and Generic Programming in C++

This advanced 3-day course brings the application of C++ to a new level by describing how object-oriented and generic programming principles are applied to real software systems. We build on the skills developed in the C++ Fundamentals course and we extend and improve our knowledge of C++ syntax. We apply this knowledge to writing robust and sophisticated C++ code. We also pay special attention to template classes and why they are so important in C++ applications.

This unique course has been developed by experienced developers and trainers and the topics reflect modern C++ technology combined with hands-on application experience. In this way you are assured of quality training. The percentage theory/practice is 70/30.

Course Contents updated May 2008

### What do you learn?

- State-of-the art C++ languages features
- Unique combination of techniques
- Standard Template (STL) and boost libraries
- Combining object-oriented and template programming techniques
- Advanced tuning and performance improvement

### What previous Delegates have said about the Course

- "Technically very deep and gave me a good overview"
- "A lot of different topics discussed; enough to think about"
- "I liked the integration of design and C++"
- "Trainer knows his stuff and is open to discussion"

### Prerequisites

You should be an experienced C++ programmers (about 2 years practical experience on real project). Not for beginners.

### Who should attend?

Software developers and programmers who wish to avail of the functionality that C++ has to offer to help them develop efficient and robust applications.

### Course form

The percentage theory/practice is about 60:40. This course is independent of any operating system and the student can choose between Windows and Linux as the system of choice.

Remark: we shall use the book "C++ Templates - The Complete Guide" by David Vandevoorde & Nicolai M. Josuttis (Addison-Wesley ISBN 978-0-201-73484-3) as one of the sources for this course.

### Follow-up courses

- Standard Template Library
- The Boost C++ Library
- Parallel Design and Programming in C++ and OpenMP

## Course contents

### Part I: Advanced C++ Syntax

#### Quick Review of C++

- Namespaces
- Function pointers
- Run-time information (RTTI)
- Exception Handling

### Programming with Templates I

- Multiple parameters
- Nested template class
- Inheritance and composition
- Compile-time and fixed-sized array classes

### Programming with Templates II

- Default parameter values
- Template template parameters
- Some templated design patterns
- Template specialization; partial specialisation

### Templated Software Components

- Traits classes
- Services and policy-based design
- 'Provides' and 'requires' interfaces
- Implementing policies in C++

### Strings

- The String class
- Basic operation
- Strings in detail
- Using strings in applications

### Part II: STL and boost Libraries

#### Review of STL

- Containers and Complexity Analysis
- Sequential and associate containers
- Const and non-const iterators
- Algorithms

### Function Objects (Functors)

- Creating function objects
- Function objects with internal state
- Predefined function objects
- Composing function objects

### Creating Adaptors for STL Components

- Information hiding and Composition
- The three ways to extend STL
- Examples
- Advantages of adaptor classes

### Overview of boost Library

- Background and history
- Overview of the specialized libraries in boost
- Relationship between STL and boost
- Advantages of boost

### Containers and Data Structures

- MultiArray
- Range
- Tuple
- Variant and Any

### Function Objects and Higher-Order Programming

- Bind
- Functions
- Signals and slots

### Smart Pointers and Memory

- The problems with 'raw' pointers and auto\_ptr
- Scoped and shared pointers; reference counting
- Single objects and arrays
- Shared pointers with STL containers

### Part III: Design, Optimisation and Deployment

#### Designing C++ Applications

- Choice of programming models
- Complexity Analysis and data structures
- Which libraries to use
- Design patterns

### Performance of C++ I

- Classifying and discovering performance bottlenecks
- Virtual versus non-virtual functions
- Preventing unnecessary object creation
- Exceptional handling

### Performance of C++ II

- Templates versus inheritance
- Using the appropriate data structures from STL
- Loop optimizing techniques
- Loop fission, fusion, unrolling and tiling

### Multi-threaded C++ Applications

- Boost thread library
- OpenMP library
- Speedup and performance improvement

## C# Programming in .NET 3.x & 4.0

This 3-day course is the entry-level course for a range of C# courses. The C# language is part of the Microsoft .NET platform. The .NET platform enables developers to use different programming languages to create one application. C# compiles to a mid level language (MSIL) that is rich enough to support a wide variety of programming languages.

The beginning of the course covers the basic C# as well as more advanced concepts in C#. Also additions to C# in .NET 3.x and .NET 4.0 will be discussed. Finally we discuss some parts of the Framework Class Library.

This course has been created for developers. The emphasis is on the appropriate use of the C# language to help them create real-life applications.

Course contents updated January 2010

### What do you learn?

- .NET platform and framework
- Basic C# syntax and object-oriented programming
- Advanced C# syntax
- C# 3 and C# 4 additions
- Important Framework Class Library classes
- Using C# effectively

### What previous Delegates have said about the Course

- "First class course, thanks"
- "Good course and good examples"
- "Very good course"
- "From 1 to 10, a 8.5"
- "Course build up very good, good teacher, useful course"

### Prerequisites

The student should have a basic knowledge of at least one procedural/structural programming language. Knowledge of object-oriented principles is an advantage but not essential.

### Who should attend?

Software engineers who wish to learn C# syntax. This course covers the fundamentals of C# which is enough to give you a head start with this new technology.

### Course form

This is a hands-on course where the percentage theory/practice is 40/60. The objective of the course is to acquaint the student with the principles of C#. The exercises in the course give the student a good idea of how to use the C# syntax.

### Follow-up courses

- ADO.NET Programming
- Creating, securing and Deploying .NET Assemblies
- Advanced C# Programming in NET 3.x & 4.0
- Generic Programming in C#
- C# and Design Patterns

# Course contents

## C# and the .NET framework

- The .NET Framework
- Common Language Runtime (CLR)
- What is MSIL
- .NET Framework Class Library (FCL)
- Where does C# fit in
- C# as an object-oriented language
- Assemblies
- Language interop

## C# Language

- The start of the application
- Variables and types
- Value types and reference types
- Copying and comparing reference types
- Strings and arrays
- Operators and their precedence
- The Console class
- String formatting
- Statements and flows
- Command-line arguments

## Classes & Objects

- Abstract Data Types
- Objects and classes
- Creating and using your own classes
- Data members and member methods
- Accessibility levels
- Constructors
- Method overloading
- This keyword

## More on classes

- Properties
- Static variables, methods & classes
- Extension Methods
- Object destruction & finalizers
- ref and out parameters
- Variable length argument lists
- Named and optional arguments
- Constant values
- Enumerations
- Nullable types & coalescing operator
- var variables

## Inheritance and Polymorphism

- The root of all classes
- Creating derived classes
- Method overriding and hiding
- Polymorphism and virtual functions
- Casting objects
- Abstract classes
- Sealed classes & methods
- More access specifiers

## Namespaces, Nested Classes and Conversions

- Why using namespaces
- Using classes in a namespace
- Placing classes in a namespace
- Nested namespaces
- Aliases
- Using assemblies
- Nested classes
- Partial classes
- Implicit conversions and member lookup
- Explicit conversions
- Checked conversions

## Interfaces

- What is an interface?
- Creating, implementing and using interfaces
- Interfaces and properties
- The is and as operators
- Interfaces versus abstract classes
- Cloning objects using the ICloneable interface
- Comparing objects using the Equals method
- IDisposable interface and using statement
- Explicit interface implementation
- Implementing ICloneable as explicit interface

## Structs

- User defined value types
- Structs versus classes
- Boxing and unboxing
- Object Initializers

## Operator Overloading

- What is operator overloading?
- Overloading binary operators
- Comparing objects using overloaded == and != operators
- Overloading unary operators
- Prefix and postfix operators
- true and false operators
- User defined conversions
- Indexers
- Guidelines

## Exception Handling

- What are exceptions
- Exceptions in C#
- Build-in exception classes and their members
- Catching exceptions: try ... catch
- finally
- Nesting try blocks
- Throwing exceptions
- Creating your own exception classes
- Chaining exceptions

## Delegates and Events

- Loose coupling using interfaces: Strategy pattern
- Delegates: safe function pointers
- Loose coupling using delegates: Strategy pattern
- Multicast delegates
- Events
- Publisher-subscribe idiom
- Model-view / observer pattern
- Defining and raising events
- Create event handlers and subscribe
- Anonymous methods

## Introduction to the .NET Framework Class Library

- Framework namespaces
- Basic framework functionality and interfaces
- Array sorting and searching
- Mathematics

- Collections: ArrayList and Hashtable
- Enumerators and foreach

## Introduction to Windows Forms

- Windows forms library
- Forms and controls
- Creating simple GUI by hand
- Event handling
- Common Dialog Boxes
- GDI+

## Introduction to Windows Presentation Foundation

- What is WPF
- XAML
- Code behind files
- Controls
- Graphics

## Introduction to Generic Programming

- Traditional .NET object data structures
- Concrete type wrapper classes
- Generic .NET datastructures
- Collection initializers
- Creating generic classes
- Templates versus generics
- Default values
- Multiple generic types
- Generic type alias & var variables
- Generic derivation- and constructor constraints
- Generic methods
- Generic delegates and events
- Strategy pattern with generics

## Deployment

- Deploying your application to the end user
- CAB setup
- Microsoft Installer (MSI) setup
- Merge modules
- Web setup

# Advanced C# Programming in NET 3.x & 4.0

This intensive 3-day course deals with a number of advanced programming topics in C# and Framework Class Library. The main objective is to develop skills in order to create robust and flexible (multi-threaded) applications in C# with .NET 3.x and the Framework Class Library. The percentage theory/practice is around 60/40.

## What do you learn?

- Break up your application in separate assemblies and share them with other applications
- Use attributes and reflection for more dynamic applications
- Use advanced IO and network components for efficient data exchange
- Use multiple threads to take advantage of modern multi-core processors
- Use generics to make more robust and efficient code and at the same time reducing your code size
- Use existing .dlls and COM components in your C# application

## What previous Delegates have said about the Course

"First class course and course documentation"

## Perequisites

We assume that the student has attended the Datasim "C# Programming" course or has similar C# experience.

## Who should attend?

C# developers and programmers who wish to take advantage of the advanced programming features in C#.

## Follow-up courses

C# and Design Patterns

Course contents updated January 2010

## Course contents

### Assemblies

- What are assemblies?
- Modules
- Assemblies versus regular .DLLs
- Assemblies for reuse, versioning, deployment and security
- Assembly contents: Metadata, Resources, code and manifest
- Private and shared assemblies
- Creating and using assembly DLLs
- Assemblies and namespaces

### Shared Assemblies and Versioning

- Shared assemblies
- Global Assembly Cache (GAC)
- Side by side versions and cultures
- Strong names
- Public and private key encryption
- Signing assemblies
- Installing assemblies in the GAC
- Delay signing
- Versioning
- Other assembly attributes

### Internationalisation & Globalisation

- Cultures
- Satellite assemblies
- Resource fallback
- Creating and using resources

### Settings

- Setting files
- Global and user settings
- Setting file versioning

### Deployment

- Installing applications at end users
- CAB setup
- Microsoft Installer (MSI) setup
- Merge modules
- Web setup
- ClickOnce deployment
- Updates

- Application and Deployment Manifests
- Signatures and code access security

### Delegates and Events

- Loose coupling using interfaces: Strategy pattern
- Delegates: safe function pointers
- Loose coupling using delegates: Strategy pattern
- Multicast delegates
- Events
- Publisher-subscribe idiom
- Model-view / observer pattern
- Defining and raising events
- Create event handlers and subscribe
- Anonymous methods
- Custom event storage

### Reflection

- What is reflection?
- Metadata, data about data
- Reflection API: Assembly, Module, Type and MemberInfo classes
- Dynamic object creation
- Dynamic method invocation
- Dynamic assembly loading
- Dynamic programming in .NET 4

### Attributes

- What are attributes?
- Some intrinsic (build-in) attributes
- Creating custom attributes
- Reading attributes

### Generics

- Traditional .NET object data structures
- Concrete type wrapper classes
- Generic .NET datastructures
- Collection initializers
- Creating generic classes
- Templates versus generics
- Default values

- Multiple generic types
- Generic type alias & var variables
- Generic derivation- and constructor constraints
- Generic methods
- Generic delegates and events

### Advanced Generics

- Generics and reflection
- Generics and serialization
- Integrating Object Oriented Programming & Generic Programming
- Covariance & contravariance
- Strategy pattern with generics

### Streams

- What are streams?
- FileStream and MemoryStream
- Files and directories
- Stream adapters
- StreamReader and StreamWriter
- BinaryReader and BinaryWriter
- Stream decorators
- BufferedStream and CryptoStream
- Creating your own stream decorators
- Object serialisation
- BinaryFormatter and SoapFormatter
- IDeserializationCallback interface

### Processes and Threads

- What are processes?
- Starting external processes
- Redirecting standard IO
- What are threads?
- Thread class and ThreadStart and ParameterizedThreadStart delegates
- Volatile data
- Thread life cycle
- Controlling thread execution: Abort, Suspend, Resume and Sleep
- Joining threads
- Thread synchronisation

- Monitor class and lock keyword
- Synchronising collections
- Thread notification: Wait, Pulse, PulseAll
- Producer-Consumer pattern
- Windows GUI and Multi-Threading
- A-synchronous delegates

#### Networking

- URIs, URLs, IP addresses and DNS
- Simple networking with WebRequest & WebResponse
- Sockets
- User Datagram Protocol (UDP)
- Sending and receiving datagrams
- Transmission Control Protocol (TCP)
- NetworkStream
- Servers (TcpListener) and clients (TcpClient)

#### Introduction to ADO.NET

- History and background
- The ADO.NET architecture and its components
- Connected and disconnected environment
- Data Sources and .NET Data

- Providers
- Connections and connection strings
- ADO.NET exceptions
- Commands
- Data reader
- Single result queries
- Parameterised queries
- Adding, editing and deleting data
- Disconnected Data
- DataSet and DataAdapter features

#### Introduction to LINQ

- var variables
- Anonymous classes
- Lambda expressions
- Extension Methods
- LINQ query expressions
- Query operators
- LINQ to SQL
- LINQ to Entities
- LINQ to XML

#### Interoperability with Legacy Code

- Using legacy DLLs
- DLL Callback functions
- Using existing COM components,

- ActiveX controls and automation servers
- Runtime Callable Wrapper
- Error handling: HRESULT and expetions
- Using the Microsoft Word object model
- Primary Interop Assemblies
- Using .NET components in non .NET software
- COM Callable Wrapper
- Registering .NET components for COM
- Early and late binding
- Creating type libraries
- Using .NET components from VB script
- Using .NET components from Visual Basic for Applications

#### Mixing C# with C++ using C++/CLI

- What is C++/CLI
- C# application using native C++
- Exposing C++ class to .NET using C++/CLI wrapper
- Native C++ application using C#
- Using a C# GUI in C++

## Generic Programming in C#

C# is an object-oriented language and since version 2.0 it supports the generic programming model. This models Abstract Data Types (ADTs) which are defined as generic data and operations on that data. We see that the generic programming model is complementary to the object-oriented (OOP) model and in some cases it is more appropriate to use it.

This 2-day course has three goals. First, we introduce the foundations of generic programming (GP) and policy-based design and we show how to design a system using generic components. Second, we describe how C# supports the GP model by introducing syntax and giving appropriate examples. Finally, we discuss how to integrate GP with OOP to produce highly robust and reusable code and applications.

This course has been developed by Datasim based on real-life application development experience with the generic programming model in C# and other languages. This knowledge has been transferred to this course. The percentage theory/practice is this course is 65/35.

Course contents updated May 2008

#### What do you learn?

- How to use C# generics in applications
- Integrating generics with OOP
- Modelling reusable components with generics
- Generic design patterns

#### Prerequisites

We assume that the reader has a reasonable knowledge of the C# language, for example classes, objects and inheritance. We take the student to the stage where he or she can write reusable and generic code and classes in C#.

#### Who should attend?

Developers who wish to learn how to implement the generic programming paradigm in C#. This course is also for those developers who extend the features in object-oriented technology to help them create reusable and generic code and classes. In fact, this course shows how to integrate these two paradigms into your applications. All source code based on the course will be provided.

## Course contents

### Part I: Policy-Based Designs and Generics

#### Introduction

- What are Generics?
- Generics versus traditional OO
- Combining Generics and OO
- Advantages of Generics
- What Generics can not do

#### Policy-based Design in C#

- Implementing contracts with interfaces
- Combining behaviour and structure: interfaces and inheritance
- Moving GOF patterns to their generic forms

#### Interfacing in C#

- Interfaces and abstract classes

- Provides and requires interfaces
- Contracts and service-oriented programming
- Contracts in C# (the where clause)

#### Defining Structure

- Inheritance
- Aggregation and Composition
- Delegation: stateful and stateless

variants

### Generics in C#

- Creating generic classes
- Generic parameters
- My first class
- Using constraints between client and generic class
- Using generic classes in your code

### Inheritance and Generics

- Deriving a generic class from another one
- Deriving concrete classes (specific data type)
- Generic abstract base classes
- Generic interfaces

### Part II: Core Syntax and Features

#### Generic Methods in C#

- Why use generic methods?
- Generic static methods
- Generic operators and operator overloading
- Where to apply generic methods

### Generics and the .NET Framework

- Impact on the .NET Framework
- System libraries
- Serialisation and generics
- Generics and Remoting
- GenericActivator

### The Generic Collections

- Improving on Object-based collections
- Performance and type-safety
- List <T>
- Dictionary<K,T>
- IEnumerable<T> and IEnumerator<T>
- Stack <T>and Queue<T>Generic

### Delegates

- What is a delegate?
- What is a generic delegate?
- Using generic delegates for event handling
- Generic Event Handlers

### Part III: Intergration and Applications

#### Generics and Reflection

- Introduction to reflection in C#
- Extending reflection to support generics

- The reflection API
- Attributes and generics

### Integrating Generics and OOP

- Generic Interfaces and Inheritance
- When to use GP and when to use OOP
- Generic Design Patterns
- Composition and Delegation

### Generic Constraints

- Ensuring type safety
- Derivation, default constructor and reference/value constraints
- Constraints and policy-based design
- Provides and requires interfaces

### Generic Serialisation and Deserialisation

- Serialisation engines
- Client-side serialisation
- Explicit and implicit serialization

## C# and Design Patterns

The goal of this 3-day course is to introduce the well-known system and design patterns (POSA, GOF) and then implement them using C#. We discuss the most important patterns that help you create flexible applications and we use the object-oriented (OOP) and generic programming (GP) to implement robust designs, quickly and efficiently.

The GOF patterns were published in 1995 and they used OOP techniques as the basis for constructing the patterns. These techniques are still valid but in some cases it is more advantageous to create these using GP techniques. Furthermore, C# has a number of language extensions and extra libraries that directly implement a number of design patterns, thus increasing developer productivity even more.

This unique course has been specially developed by Datasim for designers and developers who create practical applications. Based on years of real application software and training experience, this course discusses all the essential features in C# in combination with proven design patterns to allow developers to create robust and flexible applications.

Course contents updated May 2008

#### What do you learn?

- Learn the GOF (Gamma) patterns in C#
- Critical POSA (Buschmann) patterns to structure your applications
- Meta Design Patterns and Reflection
- Integrating OOP and GP in pattern design
- Implementing GOF patterns using generics (developed by Datasim)

#### What previous Delegates have said about the Course

"Very good course"

#### Prerequisites

The student should have a good working knowledge of C#. We assume that the student has attended the Datasim "C# Programming" course. This course is not for beginners. It is not enough to have syntax knowledge of C# in order to gain benefit from this course. If you have any queries, please contact Datasim .

#### Who should attend?

Developers and programmers who wish to apply the Gamma patterns to the creation of flexible, reusable and maintainable software systems in C#.

In short, this is the course that applies modern design patterns in combination with object-oriented and generic programming models to speed up development of real-life applications.

#### Course form

This is a hands-on course. The percentage theory/practice is 40/60. Simple examples are given as exercises while the full power of patterns in C# will be explained by extended real-life examples. The objective of this course is to acquaint the student with design patterns and how to program them using C#.

# Course contents

## Part I: Structural Patterns and Infrastructure

### Structuring a C# Application

- Whole Part Pattern
- Model View Controller (MVC)
- Presentation Abstraction Control (PAC)

### Interfacing in C#

- Interfaces and abstract classes
- Provides and requires interfaces
- Contracts and service-oriented programming
- Contracts in C# (the where clause)

### Defining Structure

- Inheritance
- Aggregation and Composition
- Delegation: stateful and stateless variants

### Short Overview of UML 2.0 Diagrams

- Class diagrams
- Component diagrams
- Assembly and connector components
- Sequence diagrams and statecharts

## Part II: GOF Patterns in C#

### Overview

- Background to design patterns
- GOF pattern classification
- 20-80% rule: essential and non-essential patterns
- Advantages of using design patterns

### Creational Design patterns

- Applicability of creational patterns to C#
- The Builder pattern

- Factory Method and Abstract Factory patterns
- Prototype pattern
- Using .NET Reflection for factories: Activator

### Structural Design Patterns

- Composite pattern and nested objects
- Proxy pattern
- Decorator pattern
- Class and object Adapter
- Bridge pattern

### Behavioural Patterns I

- Command pattern
- Strategy pattern
- Using Delegates for creating algorithms
- Using interfaces and inheritance for Template Method pattern
- Visitor pattern and service extension

### Behavioural Patterns II

- Observer (Publisher-Subscriber) pattern
- Mediator pattern and change control
- Using events and Delegates as an alternative to Observer

### Other Behavioural Patterns

- State pattern
- Role pattern
- Propagator pattern

## Part III: Patterns using C# Generics

### Policy-based Design in C#

- Implementing contracts with interfaces
- Combining behaviour and structure:

inter-faces and inheritance

- Moving GOF patterns to their generic forms

### Generic Patterns

- Strategy pattern
- Bridge pattern
- Command pattern

## Part IV: Patterns and C# Extensions

### Regular Expressions

- What is a regular expression?
- Matching
- Compiled regular expressions
- Regex options

### Lambda Expressions

- Using lambda expressions instead of delegates
- Expression trees
- Generic lambda expressions
- Anonymous methods

### Serialisation

- Serialisation engines
- Explicit and implicit serialization
- Binary and XML serialization
- Integration with Builder and Visitor patterns

### Multi-threaded patterns

- Master-slave pattern
- Producer-Consumer pattern
- Thread pooling

# Standard Template Library

The ANSI Standard Template Library (STL) is a library of parametrised classes for several entities such as containers, algorithms, iterators and allocators.

### Prerequisites

Knowledge of C++ and some months working experience with C++.  
Knowledge of templates and exception handling is re-commended.

### Follow-up courses

The Boost C++ Library

### Who should attend?

C++-programmers who wish to develop reliable, and efficient software using STL standardised classes.

Course duration 2 days

# Course contents

## Overview of Standard Template Library (STL)

- What is STL?
- STL Components
- Containers
- Main Container Types
- Algorithms
- Main Algorithm Categories
- Set-like Operations
- Iterators

## Function Objects

- Adaptors
- Allocators
- Strengths and Limitations of STL
- Student Prerequisite Knowledge

## STL Containers

- Sequence Containers
- Vector

- Deque
- List

## Sorted Associative Containers

- Multisets (Bags)
- Sets
- Set\_like Operations on Sorted Structures
- Multimaps

- Maps

#### Iterators in STL

- What is an Iterator?
- Iterator Categories
- Iterator functions
- Iterator functions: Input Iterators
- Output Iterator Types
- Forward Iterators
- Bi-directional Iterators
- Random Access Iterators
- Qualifying Iterators: Mutable and Constant Iterators

#### Algorithms in STL

- Overview of STL Algorithms
- Algorithm Categories
- Algorithms with Function Parameters
- Non-mutating Sequence Algorithms
- Mutating Sequence Algorithms

#### Function Objects

- Overview of Function Objects
- Categories of Function Objects
- Advantages/Features of Function Objects

#### Adaptors in STL

- What is an Adaptor?
- Container Adaptors
- Stack Container Adaptor
- Queue Container Adaptor
- Iterator Adaptors
- Function Adaptors

#### Extending the STL

- Creating new containers
- User-defined sequence and associative containers
- Creating reusable template libraries
- Domain-specific classes
- 

#### Generalised Functors

- The Command design pattern
- Functor template classes
- Handling functors
- Handling pointers to member functions
- Binding
- Chaining requests

#### Casus: An Application using STL

- Description of problem
- Why we use STL
- Extending STL to support dynamic properties
- Using Functors
- Nested objects

## The Boost C++ Library

The goal of this unique 3-day hands on course is to discuss the most important libraries in the Boost library suite. Boost is a de-facto standard and a number of its libraries will be part of the new C++ standard. The main advantage for developers is that it contains functionality that can be used directly in applications without the need to reinvent the wheel. The course discusses each library in detail, from the fundamental syntax, numerous examples to show how to use the library and how to integrate the library in applications. We classify the libraries in the course as follows:

- Higher order functions
- Data types
- Text and string processing
- Data structures
- Mathematics and Statistics
- Utilities and miscellaneous libraries

The percentage theory/practice is approximately 70/30. We also provide the students with source code and extended library documentation.

You also receive a copy of the book "Introduction to the C++ Class Libraries - Volume I" by Robert Demming and Daniel J. Duffy, published by Datasim Press. This is the first textbook that discusses 30 of the most important libraries in Boost and we also provide full working source code for all the examples in the book.

Course contents updated October 2010

#### What do you learn in this course?

In this course you get a detailed overview of the most important Boost libraries and how to use them in practical projects:

- Essential libraries and 'nice to have' libraries
- Integrating Boost libraries with your applications
- Hands-on experience with library code
- Reengineering legacy code to Boost

The benefits of attending this course for the student are:

- Learn and use essential syntax of 30 Boost libraries
- Kick start software development using Boost
- Knowing which libraries to use in a given context
- Use the book and full source code as resource after course end
- Learn from experienced trainers

#### Your trainer

Dr. Daniel Duffy has been working in software application development since 1979 in projects for heat flow and fluid dynamics,

computer graphics and Computer Aided Design (CAD), simulation and optimisation and computational finance. He has been working with C++ since 1989 and is the author of 7 books on C++ and its applications. Daniel Duffy has a PhD in numerical analysis from Trinity College, Dublin.

#### Prerequisites

Knowledge of C++ and its syntax, in particular templates as discussed in Datasim's C++ Fundamentals course for example. We also assume that the students have a reasonable working knowledge of STL.

#### Who should attend?

C++ developers and designers who wish to get a fast understanding of the Boost libraries. This course is also of interest to software designers and architects who wish to gain an appreciation of how their designs can be implemented in C++ using Boost.

# Course contents

## Category: Function Objects and Higher-Order Programming

### Function

- Generalised callback mechanisms
- Storage and invocation of functors, (member) function pointers
- Applications to notification patterns (Observer, Signals and Slots)
- Example: separating GUIs from business logic

### Bind

- Generalising and improving the STL Bind
- Uniform syntax for functors, (member) function pointers
- Functional composition and nested binders
- Bind and in Boost.Function combined

### Signals and Slots

- Implementation of Observer (Publisher-Subscriber) pattern
- Event management with minimal inter-object dependencies
- Signals == Subject, Slots == Subscriber
- Slot groups

### Lambda

- Unnamed functions
- Using Lambda in STL algorithms
- Avoiding creation of many small function objects
- Less code: write function at location where it is needed

### Phoenix

- Phoenix architecture
- Actors and arguments
- Function operator
- Other components in Phoenix

## Category: Data Types

### Tuple

- Modelling n-tuples (pair is a 2-tuple)
- Using tuples as function arguments and return types
- Accessing the elements of a tuple
- Advantages and applications of tuples

### Variant

- Creating discriminated unions with heterogeneous types
- Manipulating several distinct types in a uniform manner
- Type-safe visitation
- Using static\_visitor
- Avoiding type-switching for variant data

### Any

- Value-based variant types
- Discriminated types
- Typesafe storage and retrieval
- Applications of Any

### Interval

- Interval arithmetic
- Modelling round-off errors explicitly
- Applications to numerical analysis
- Quantifying the propagation of rounding errors

### Other Data Types

- Integer
- Rational
- Parameter

## Category: String and Text Processing

### Regex

- Regular expressions and pattern matching
- Processing large and inexact strings
- Regex and callbacks
- Exception handling
- Emulating functionality as in Perl, awk, sed

### String Algorithm

- Trimming and conversion
- Find and replace
- Find iterator
- Join and split

### Tokenizer

- Separating character sequences into tokens
- Finding data in delimited text streams
- User-defined delimiters

### Xpressive

- Dynamic regular expressions
- Mixing static and dynamic regular expressions
- Semantic actions and user assertions
- Applications

## Category: Data Structures

### Date and Time

- Gregorian, Posix and local time
- Temporal types
- Calculating with dates and times
- Applications of DateTime

### Graph Library (BGL)

- Overview of library: data structures and algorithms
- Graph's generic components
- Graph algorithms

### MultiArray

- MultiArray components: construction and assignment
- Array view type generators
- Specifying array dimensions
- Accessing elements
- Creating views
- Storage ordering

## Category: Mathematics and Statistics

### Math/Special Functions

- Gamma, beta, error functions
- Bessel functions

- Factorials and binomial coefficients
- Other functions

### Math/Statistical Distributions

- Continuous and discrete univariate statistical distributions
- Density functions
- Properties (skewness, quantile, mode, mean etc.)

### Random Number Library

- Concepts
- Random number variate generators
- Random number library distributions
- variate\_generator
- Applications

### uBLAS (Basic Linear Algebra System)

- Other Math
- Brent's method
- Math Common Factor
- Quaternion and Octonion

## Category: Utilities and other Libraries

### Memory and Smart Pointers

- Conversions
- Polymorphic casts and downcast
- Numeric casts
- Lexical casts

### Filesystem

- Portable manipulation of paths, directories and files
- Defining functionality as in scripting languages
- Platform portability

### Serialisation

- Saving arbitrary data to an archive (e.g. XML)
- Restoring data from an archive
- Separate load and save actions
- Versioning

### Other Libraries

- Program options
- Test
- Flyweight
- Lexical Cast
- Program Options
- Concept Check
- Functional Hash

## Category: Multi-threading and Networking

### Thread

- Modelling portable lightweight processes
- Threading primitives
- Higher-level abstractions
- Thread groups and thread arrays

### Statecharts

- C++ implementation of UML Statechart method
- Hierarchical and orthogonal states
- Transitions and actions

- Applications
- Introduction to Meta State Machine

#### Introduction to Asio

- Core Concepts
- Concurrency and patterns
- Socket IOstreams
- TCP, UDP, ICMP
- BSD Socket API

#### Timer

- Measuring elapsed time
- Reporting progress for programmer tests
- Asio timers

# Advanced Boost Interprocess Communication and Networking Libraries

The goal of this intensive 3-day course is to learn and apply a number of libraries in Boost that are concerned with interprocess communication, multi-threading, networking and advanced memory management. Using these libraries allows developers to build reliable, efficient and portable applications.

One of the advantages of Boost libraries is that they form an integrated and portable set of modules and classes without having to use proprietary libraries developed for the same purpose. These libraries support much if not all of the functionality that is present in other C++ libraries but in our opinion Boost is easier to learn and to use in applications.

#### Overview of Course

This comprehensive course introduces the techniques needed in order to write robust and efficient code. The percentage theory/practice is 80/20. The following topics are discussed in detail:

- Thread essentials and multi-threaded design
- Advanced thread synchronization, notification and thread patterns
- Process communication and shared memory
- Advanced memory allocation
- Synchronous and a-synchronous process and network I/O
- TCP, UDP and SSL protocols
- Event-driven signals and slots

#### What do you learn?

This course introduces and examines in detail all the relevant boost libraries that are needed when developing networked and multi-(process, thread) applications. We start with model problems, then we move on to more advanced and extended example and we also discuss the application of the libraries to event-driven, real-time data flow and other related problems. The course includes documentation, book and full source code of all examples. The percentage theory/practice is 80/20. The objective is to learn the libraries by using a small framework and modifying it for maximum effectiveness.

#### Prerequisites

A working knowledge of C++. Ideally, you develop applications in which these boost libraries can be used to maximal effect.

## Course contents

#### Timer

- Timers overview
- Measuring elapsed time
- Automatically display elapsed time
- Displaying the progress

#### Dynamic Bitset

- Dynamic bitset overview
- Setting and reading bits
- Bitwise operations

#### Thread Basics

- Thread overview
- Creating threads
- Thread function as global function or static function
- Thread function as callable object
- Volatile variables
- Thread lifecycle and states
- Controlling thread execution (yield,

sleep, join, interrupt)

#### Advanced Threading

- Thread synchronization
- Locking objects & classes (mutex & lock\_guard)
- Thread notification
- Condition variables
- Producer/Consumer via synchronized queue
- Thread groups
- Thread local storage

#### Interprocess Communication

- Process overview
- Interprocess communication
- Persistence of IPC mechanisms
- Shared memory
- Memory mapped files
- Mapped regions

- Pointers in mapped regions
- Fixed address mapping
- Static members in mapped region

#### Interprocess Memory Manager

- Managed memory segments
- Managed shared memory
- Managed mapped file
- Managed heap memory
- Managed external buffer
- Segment manager
- Allocating memory fragments

#### Advanced Memory Manager

- Allocating simple objects
- Allocating composite objects
- Synchronising composite object creation
- Boost STL compatible containers
- Using allocators to create your own

classes that can be stored in managed memory

#### Interprocess Synchronisation

- Synchronization mechanisms
- Named and anonymous synchronization mechanisms
- Mutexes & scoped locks
- Condition variables
- File locks
- Message queues

#### System

- System overview
- Portable errors
- Error codes, error conditions and error categories

#### Asio Concepts

- Asio internals - The reactor design pattern
- Threads
- Strands
- Streams

#### Asio Networking Basics

- Synchronous IO and a-synchronous IO
- IP address & hostnames
- Resolving IP addresses using DNS
- Sockets and end points

#### Asio UDP communication

- User Datagram Protocol (UDP)
- Sending datagrams
- Receiving datagrams

#### Asio TCP communication

- Transmission Control Protocol (TCP)
- Servers and clients
- Network streams
- Secure Sockets Layer (SSL)
- Timers

#### Pool

- Pool overview
- Simple segregated storage
- Pool types (simple, object, singleton)
- Allocators

#### Signals & Slots

- Signals overview
- Publishers and subscribers
- Single-cast
- Multi-cast and defining calling order
- Signals with input arguments
- Signal return arguments and combiners

#### Advanced Signals & Slots

- Connection management (disconnecting slots, blocking slots)
- Automatic connection management
- Signals2 (thread-safe signals & slots)

#### UML State charts in Boost

- Hierarchical (composite, nested) states
- Orthogonal states
- Transitions and Guards
- Event delay

## Parallel Design and Programming in C++ and OpenMP

Many C++ applications are written for single-core processors but with the emergence of affordable and powerful multi-core processors we can create applications that use the full power of these processors using parallel programming techniques. Present-day commodity computers already contain duo-core and quad-core processors and this opens up new possibilities to create even faster applications that would have been unthinkable even a few years ago.

This unique three-day hands-on course is a thorough introduction to the design and implementation of multi-threaded C++ applications using the OpenMP library. OpenMP is a collection of compiler directives, library routines and environment variables that developers can use to specify shared-memory parallelism in C and C++. It is a de-facto industry standard and it has been accepted by several major software and hardware vendors.

The course has three main parts; first, we introduce the most important background concepts and methods that you will need when developing OpenMP applications and we discuss the major design methods for parallel applications. Second, we discuss all the features in OpenMP, how to use them in practical situations and why they are useful. Finally, we implement applications using OpenMP in conjunction with the design techniques that we describe in the earlier sections in the course.

After having completed this course you will be in a position to develop efficient multi-threaded applications using C++ and OpenMP by incremental addition of features to your sequential programs or by developing new applications from scratch.

The percentage theory/practice is 60/40. The examples and applications will reflect cases from actual software systems.

#### What do you learn in this course?

- Implement high-performance applications in C++
- Learn all about OpenMP
- "thinking parallel", design parallel from the start

#### Who should attend?

C++ developers and designers who wish to implement fast applications using multi-processor computers. The focus is on building and running well-designed and scalable applications with OpenMP.

#### Prerequisites

A working knowledge of C or C++. It is an advantage if you work with software applications in which performance is important and where advantage can be taken of multi-processor computers.

# Course contents

## Part 1: An Introduction to Multi-Threading Concepts

In this section we discuss the most important issues and concepts that relate to multi-threaded and parallel programming environments. The results are of general applicability to both shared memory and distributed memory applications as well as to well-known programming languages such as C++, Java and C#.

### Memory Systems

- Shared memory parallel computers (SMPs)
- Shared and cache memory
- Shared memory consistency models
- Distributed memory and shared distributed memory

### Threads

- What is a thread?
- Thread attributes
- Thread execution lifecycle
- User threads and kernel threads

### Data Access in Threads

- Fork-join (master/slave) model
- Shared and private data
- Thread synchronization

### Synchronisation in Detail

- Mutual exclusion (mutex) and condition variables
- Critical sections
- Memory synchronization and fences
- Barriers

### Troubleshooting

- Sequential consistency
- Removing data dependencies
- Race conditions
- Deadlock and livelock

## Part 2: Parallel Design Techniques

This section discusses how to design software systems that will run in a multi-processor environment. In this case the traditional system development methods (for example, object-oriented design) must give way to design methods that support the parallel nature of the problem. To this end, we introduce data and task decomposition techniques to help the designer to partition the problem into independent subsystems and assign them to appropriate processors.

### Introduction

- Parallel architecture types
- Flynn's taxonomy
- SIMD (SPMD) and MIMD architectures
- Amdahl's and Gustafson's laws
- Speedup

### Decomposition Techniques

- Task and data decomposition
- Grouping and ordering tasks
- Data sharing among tasks

- Evaluation

### Algorithm Structure

- Task and data parallelism
- Divide and conquer
- Geometric decomposition
- Other decomposition techniques

### Parallel Design Patterns

- SPMD pattern
- Master/Worker pattern
- Loop parallelism pattern
- Shared data and shared queues patterns

## Part 3: OpenMP Core Techniques

This section discusses the OpenMP library and what it has to offer when we wish to implement parallel software systems. We introduce the most important pragmas, library functions and environment variables that are the building blocks for multi-threaded and parallel applications.

### Overview

- Compiler directives
- Library routines
- Environment variables

### My First OpenMP Program

- Writing the serial program
- Determining parallel code
- Adding OpenMP directives
- Debugging and performance measurement

### Data Clauses in OpenMP

- Shared and private
- Lastprivate, firstprivate
- Default and nowait clause

### OpenMP Synchronisation Constructs

- Barrier
- Ordered
- Critical and Atomic
- Locks, Master construct

### Work Sharing in OpenMP

- Loop construct
- Sections and section
- Single construct
- Combined parallel work-sharing constructs

### Other Clauses

- Reduction clause
- Copyin clause
- Copyprivate clause
- Ordered clause

### Configuration and Run-Time Information

- Setting environment variables' values
- Library functions for thread information
- Scheduling functions
- Lock functions
- Timing functions

## Part 4: Applications and Performance Measuring

In this section we show how to integrate the techniques from the first three parts of the course in order to implement robust, correct and efficient parallel applications. We discuss loop optimization, troubleshooting OpenMP and developing applications.

### Troubleshooting in OpenMP

- Common problems
- Race, shared and private variables
- Work scheduling assumptions
- Side-effects; the need for thread safety

### Advanced Problems

- Memory consistency problems
- Using flush
- Deadlock and livelock situations

### Debugging

- Verification of the serial program version
- Verification of the parallel program version
- Using tools

### Applications, Demos and Discussion

- Monte Carlo simulation
- Matrix algebra and solving linear systems
- Sorting Finite Difference method

# Multi-threaded and Parallel Programming in C#

Many applications have been written for single-core processors but with the emergence of affordable and powerful multi-core processors we can create applications that use the full power of these processors using parallel programming techniques. Present-day commodity computers already contain dual-core and quad-core processors and this opens up new possibilities to create even faster applications that would have been unthinkable even a few years ago.

This two-day hands-on course is a thorough introduction to the design and implementation of multi-threaded C# in the .NET framework. The course has three main parts; first, we introduce the most important background concepts and methods that you will need when developing multi-threaded C# applications and we discuss the major design methods for parallel applications. Second, we discuss all the features in C# Threading Library, how to use them in practical situations and why they are useful.

After having completed this course you will be in a position to develop efficient multi-threaded applications using C# by incremental addition of features to your sequential programs or by developing new applications from scratch. The percentage theory/practice is 70/30. The examples and applications will reflect cases from actual software systems. The focus in the programming part of the course is on applying multi-threaded code in existing sequential code, for example, thus allowing you to concentrate on essential issues.

Course contents updated in January 2010

## What do you learn in this course?

- Implement high-performance applications in C#
- Learn all about C# multithreading in the .NET framework
- "thinking parallel", design parallel from the start

## Prerequisites

A working knowledge of C#. It is an advantage if you work with software applications in which performance is important and where advantage can be taken of multi-core/multi-processor computers.

## Who should attend?

C# developers and designers who wish to implement fast applications using multi-core/multi-processor computers. The focus is on building and running well-designed and scalable applications with C#.

## Course contents

### Part 1: An Introduction to Multi-Threading Techniques

In this section we discuss the most important issues and concepts that relate to multi-threaded and parallel programming environments. The results are of general applicability to both shared memory and distributed memory applications as well as to well-known programming languages such as C++, Java and C#.

#### Memory Systems

- Shared memory parallel computers (SMPs)
- Shared and cache memory
- Shared memory consistency models
- Distributed memory and shared distributed memory

#### Threads

- What is a thread?
- Thread attributes
- Thread execution lifecycle
- User threads and kernel threads

#### Data Access in Threads

- Fork-join (master/slave) model
- Shared and private data
- Thread synchronization
- Synchronisation in detail

#### Mutual exclusion (mutex) and condition

#### variables

- Critical sections
- Memory synchronization and fences
- Barriers

#### Troubleshooting

- Sequential consistency
- Removing data dependencies
- Race conditions
- Deadlock and livelock

### Part 2: C# Multi-threading, Core Functionality

This section discusses the C# threading library and what it has to offer when we wish to implement parallel software systems. We introduce the most important library functions and environment variables that are the building block for multi-threaded and parallel applications.

#### Getting Started

- Writing a serial program
- A simple '101' threaded program
- Debugging; discovering race conditions
- Thread pre-emption
- Performance

#### Threads and Data

- Implementing the shared data metaphor

- Passing data to a thread
- Mutex and Semaphore in C#
- Thread safety

#### Synchronisation I

- Blocking methods
- Sleep and EndInvoke
- Join
- Spinning

#### Synchronisation II

- Locking
- Mutex and Semaphore, again
- Nested locking
- Locking and atomicity

#### Synchronisation III

- Nonblocking synchronization
- Atomicity and interlocked
- Memory barriers
- Signalling with event wait handles

#### Synchronisation IV

- Two-way signaling
- Pooling wait handles
- Signalling with wait and pulse
- Producer-consumer queues

#### .NET 4.0 Features

- Cancellation tokens
- Concurrent collections
- Task Parallel Library (TPL)
- Parallel LINQ (PLINQ)

## Part 3: Advanced Multi-threading and Applications

### Decomposition Techniques

- Task and data decomposition
- Grouping and ordering tasks
- Data sharing among tasks
- Evaluation

### Algorithm Structure

- Task and data parallelism
- Divide and conquer
- Geometric decomposition
- Other decomposition techniques

### Parallel design patterns

- SPMD pattern
- Master/Worker pattern

- Loop parallelism pattern
- Shared data and shared queues patterns

# Programming Generic and Parallel Design Patterns in C++

## Using Boost and OpenMP

The goal of this 4-day hands-on course is to design and program computationally intensive applications using parallel design patterns and the multi-threading libraries OpenMP and Boost.Thread. We apply task and data decomposition techniques to help discover parallel design patterns and we then implement them in C++. Another goal is to improve and update the GOF (Gamma) design patterns and to implement them using templates and the functionality in Boost.

Course contents made 20 June 2010

### What do you learn?

This programming course (percentage theory/practice is approximately 60%-40%) for software developers and designers who wish to design and implement parallel software systems using the OpenMP and Boost.Thread. The focus is on applying the new generation of generic and parallel design patterns to software development. In particular, you learn the following core techniques:

- Upgrading GOF patterns: which patterns are more suitable using templates?
- Foundations of multi-threading programming in Boost.Thread and OpenMP
- Using Boost libraries that directly support design patterns
- The catalog of multi-threaded patterns (Mattson, POSA) and how to apply them
- Writing parallel software using patterns and multi-threading libraries

In short, this course answers the following questions:

- Which patterns are suitable/optimal for my problem?

- What is the applicability of a given pattern? What is it good for?
- Should I use Boost.Thread or OpenMP (or even combine them)?
- Debugging and troubleshooting parallel software

This course replaces the course C++ and Design Patterns that we supported until 2010.

### Prerequisites

We assume that you have a good working knowledge of C++.

### Who should attend?

This course is for software developers and designers who write C++ code for computationally intensive applications and who wish to avail of modern multi-core and multi-processor CPUs to improve the speedup and responsiveness of these applications. The course uses the latest developments in parallel and generic programming models in C++.

## Course contents

### Part 1: Generic Design Patterns

#### A Review of the GOF (Gamma) Patterns

- Foundations (polymorphism and delegation)
- Creational, structural and behavioural patterns
- Consequences of using the object-oriented model
- Action points for using new design models

#### Designing Classes: Fundamental Techniques

- Object Connection architecture (OCA)
- Interface Connection architecture (ICA)
- Plug and socket architecture (PSA)
- Which architecture to choose and when?

#### Introduction to generic Design Patterns

- What is a generic design pattern?
- Relationship with connection

architectures

- Modelling structure and behaviour
- Combing object and generic architectures

#### Generic Design Building Blocks

- Class traits
- Policy-based design and C++ templates
- Template Template Parameters (TTP)
- Curiously Recurring Template Pattern (CRTP)
- Using Design Building Blocks
- TTP for aggregation and composition
- Implementation of static polymorphism with CRTP
- UML object structures using OOP and GP
- Review: efficiency, reusability, reliability and maintainability

### Part 2: Design Patterns in Boost

#### Overview of Boost Libraries

- Boost library categories
- Boost libraries and design patterns
- Boost libraries that directly support design patterns
- Designing with Boost libraries

#### Flyweight (Boost) Pattern

- Creating and managing shared objects (intrinsic/extrinsic data)
- Key-value flyweights
- Flyweight and composites
- Flyweight factories
- Multi-threading

#### Boost Signals and Slots Library

- Creating multicast callbacks for functions and function objects
- Triggering and handling events: loose coupling
- Connecting slots and slot grouping
- Signals with arguments
- Multi-threaded Signals library

### The GOF Observer Pattern

- What is not right with the GOF Observer?
- Using Signals and Slots to implement the Observer
- Multiple publishers, multiple subscribers
- Application: Generic Model View Controller

### The generic Command Pattern

- A review of the GOF Command pattern: what are the issues?
- Using Boost.Function to create object-level Command patterns
- Boost.Regex as an interpreter for user requests
- Composite commands

### Generic Strategy Pattern

- Template member function solution
- Using generic function objects
- Where does Boost.Function fit in?
- Advantages of using generic Strategy
- 

### Part 3: Multi-Threaded Patterns and their Implementation

#### Overview of Parallel Software Design Process

- Finding Concurrency: task and data decomposition
- Algorithm structure design
- Supporting parallel design patterns
- Implementation mechanisms

#### Multithreading Basics

- Thread class and lifecycle
- Shared and private data
- Mutual exclusion
- Barriers and fences
- Thread synchronization and notification

#### OpenMP

- Core concepts and runtime library
- Structured blocks and directive formats
- Worksharing Synchronisation; scheduling

### Boost.Thread

- Thread class and thread start options
- Mutex and locking
- Wait and notify
- Condition variables
- Thread groups

### Algorithm Structure Patterns

- Task parallelism
- Divide and Conquer
- Geometric Decomposition
- Producer-Consumer

### Supporting Structures Patterns

- Master/Worker
- Single Program Multiple Data (SPMD)
- Fork/Join
- Loop Parallelism

### Troubleshooting

- Sequential consistency
- Removing data dependencies
- Race conditions
- Deadlock and livelock

## Requirements Determination

The goal of this 3-day course is to discover, analyse and document the functional and non-functional requirements that a software system should satisfy. The focus is on determining those features that a system should offer to its actors and stakeholders. The core process is to create an unambiguous requirements document that contains all the technical details that are needed by analysts and designers.

The course consists of three main stages; the first stage entails scoping the problem and finding an initial set of requirements and features. The second stage involves analyzing these requirements for inconsistencies, conflicts and missing requirements. The main objective is to resolve these problems before starting with stage 3, which is concerned with describing and documenting requirements that other stakeholders (such as analysts, designers and users) can understand and use. The percentage theory/practice is 70/30.

This course is based on a number of years of practical experience with requirements determination in real projects combined with use of modern and improved requirements capture techniques. In particular, we incorporate use case technology into the course as a special case. For complex systems we must ensure that they do not become unmaintainable, which is what we have experienced in projects.

Course Contents updated May 2008

### What do you learn?

- Scoping large complex systems
- A structured and defined approach to requirements discovery
- Interviewing techniques and improved stakeholder communication
- How to incorporate other approaches into the model, for example use cases

### What previous Delegates have said about the Course

- "Good, step-by step approach"
- "Modern and highly useful methods"
- "Exceeded my expectations"

### Prerequisites

We assume that the student is a senior analyst or designer and has extensive knowledge of at least one systems methodology (for example, OMT, Yourdon, Jackson, UML). Furthermore, the student must have experience with real-life application development.

### Who should attend?

Analysts, novice requirements analysts, product managers and other professionals who are involved in 'upstream' system development activities. This course is not suitable for junior designers or developers.

### Course form

The percentage theory/exercises is 70:30. The student learns a number of interviewing techniques in order to learn how to elicit requirements during the role playing sessions.

# Course contents

## Part I: Skills and Techniques

### Interviewing Techniques

- The Inquiry-based Model
- Storyboards
- Other sources for Requirements Capture

### The Inquiry-based Model in Detail

- Introduction to the inquiry-based model
- Starting an interview; the body of an interview
- Closing an interview
- Information capturing

### Viewpoints and Stakeholders

- Kinds of stakeholders
- Stakeholders as sources of requirements
- Viewpoint-oriented requirements determination
- Viewpoints and business concerns
- Mapping viewpoints to requirements

### Requirements

- Functional requirements
- Non-functional requirements
- Usability requirements
- Goal modelling

### Requirements and Use Cases

- Why a use case is not a requirement
- Stakeholders and UML Actors
- Functional and non-functional requirements

### Service Oriented Architectures

- What is a service?
- Provides and requires services
- Discovering requirements from services
- Documenting services
- 

## Part II: Requirements Elicitation

### The Customer Requirements Document

- Initial features and requirements
- Interviewing key stakeholders
- Is the project feasible?
- Creating a Proof-Of-Concept (POC) system

### Identifying Stakeholders

- Preparing for interviews
- Applying the Inquiry-based model
- Hidden and real goals
- Other sources of requirements

### Risk-Reduction during Elicitation

- The major business concerns
- Business processes
- Creating a context diagram
- Major functional (FRs) and non-functional requirements (NFRs)

## Part III: Requirements Analysis and Negotiation

### Requirements Analysis

- Sharpening our understanding of the initial requirements
- Requirements' priorities and risks
- Checklist questions and requirements classification
- Inter-requirement correlation (negative, positive)

### Requirements Negotiation

- Customer Importance Levels
- Conflicts and Overlaps
- Describing requirements quantitatively
- Classification of requirements for IT stakeholders

### System Structure and Decomposition

- Defining system boundaries; updated context diagram

- Removing superfluous requirements
- The Single Responsibility Principle (SRP) and unique requirements
- Aligning behaviour and structure

### Domain Architectures

- What is a DA?
- The 5 categories: MAN, RAT, MIS, PCS, ACS
- Behavioural and structural aspects of DA
- The role of DAs in requirements engineering
- DAs and analogical reasoning

### Scenarios: DAs to Requirements

#### Determination

- Finding the DA corresponding to my system
- What are the important requirements in a DA?
- Finding the important stakeholders in a DA

## Part IV: Requirements Description and Documentation

### General Rules

- Standard templates
- Use of diagrams (UML class and component diagrams)
- Creating the requirements document; structure
- Requirements validation

### Documentation using Services

- Determining 'provides' and 'requires' services
- Services and requirements
- Aligning services and systems

### Requirements Management

- What is requirements management?
- Defining management policies
- Change management policies
- Management and requirements

## Component and Service Oriented Analysis

The goal of this course is to analyse a software system (called the System Under Discussion, or SUD) and its requirements by decomposing them into loosely coupled subsystems (logical components). We analyse and design each subsystem independently of the other subsystems by ensuring that each one has well-defined responsibilities and its functionality is orthogonal to that of the other subsystems. We integrate these subsystems to realise all system requirements.

The process has a step-by-step character and it entails both behavioural and structural aspects that will be integrated into one set of products, namely a collection of logical components, their internal structure and their dependencies on other components:

A1: Create context diagram (SUD and its external systems)

A2: Find and describe system services

A3: System decomposition

A4: Realising functional and non-functional requirements

A5: System and sub-system interfaces

A6: Documenting logical components in UML 2.0

This unique course defines a defined and proven process to map system requirements to independent logical components. As a special case, we can use the process to model less complex systems using OOA (object-oriented analysis). The percentage theory/practice is 70/30.

### What do you learn?

- Create logical components for input to design
- Analyse requirements documents to define system boundaries
- Decompose a system into simpler subsystems
- A defined and proven process to analyse systems
- Integrate UML artefacts into the analysis process

### What previous Delegates have said about the Course

- "Very clear and educational"
- "I liked the steps in the COA process"
- "Great, very interesting and useful"

This course is based on application and training experience that your trainer Dr. Daniel Duffy.

### Prerequisites

A good knowledge of at least one object-oriented language and some project experience with technical or business systems.

### Who should attend?

Software analysts, designers and developers who wish to take system and user requirements and map them into a network of loosely coupled generic components that communicate via 'provided' and 'required' interfaces. The products of COA will now be used as input to the Component Oriented Design (COD) course.

## Course contents

### Part 1: System Scoping

#### Review of Requirements Determination

- Stakeholders, actors and viewpoints
- Functional and non-functional requirements
- An Introduction to Use Cases
- Documenting Use Cases

#### System Scoping and Discovery

- Goals and core processes in system
- Visualising flow in the system using UML activity diagrams
- Object and data discovery

#### System Context

- Key and satellite systems
- System responsibilities
- Inter-system interaction (client-server relationships)
- Initial top-level component diagram

#### Where does OOA fit in?

- Components are not objects
- When OOA works and when it does not work
- Integration of OOA into COA
- Class and component diagrams

### Part 2: Service Identification and Elaboration

#### Service-based Architectures

- What is a service?
- Discovering services from system actors and their viewpoints
- Service types and categories
- The relationship between services and system

#### Documenting Services

- Choosing a standard template
- Preconditions and postconditions
- Describing a service: its capability
- Dependencies on other services

### Interfaces

- What is an interface?
- Using services to find interfaces
- Documenting interfaces
- Interfaces, reusability and standardisation

### Part 3: Defining and Structuring System Components

#### High-level Components

- What is a component?
- Component decomposition and aggregation
- Component specialization
- Top-level component diagram

#### Standardised Components

- Introduction to Domain Architectures (DA)
- Using DA to "generate" subsystems and interfaces
- How does my system relate to DA?
- Examples and guidelines

#### Documenting Components in UML 2.0

- Basic and composite components
- Packaging components
- UML stereotypes
- Connectors

#### Component Decomposition Techniques

- 'Divide and conquer' (whole-part pattern)
- Mediators and loosely coupled systems
- Layered systems Model-View-Controller (MVC)
- Presentation-Abstraction-Control (PAC)

#### Discovering Internal Components and Classes

- Internal communication
- Sequence diagrams in UML 2.0 and their application
- Classes, data, operations and components
- Ports and connectors

### An Introduction to Architectural Description Languages (ADL)

- Architectural styles
- Event-based systems
- Repositories
- Pipes and Filters
- OO systems

### Part 4: Appendix: UML 2.0 Essentials for COA

The topics in this part are used as support for the COA topics. We introduce them as needed in the course.

#### Use Cases

- Actors and roles
- Discovering use cases
- Template structure for use case documentation
- Advantages and disadvantages of use cases

#### Activity Modelling

- What is an activity?
- Data flow and control flow
- Input and output pins
- Activity nesting and invocation hierarchies
- Activity partitioning and swimlanes

#### Interactions: Overview

- Why do we need interaction diagrams in COA?
- Level of detail Interaction building blocks
- Kinds of diagrams

#### Deployment in UML 2.0

- Component diagrams
- Interfaces
- Composite structure diagrams
- Deployment diagrams

# Component Oriented Design Generic and Multi-threaded Implementations

The goal of this updated and revised 3-day course is to design complex software system using modern object-oriented, generic and multi-threaded (parallel) design techniques. Starting from a high-level specification of the problem (from analysis) we then design it while taking account of the functional and non-functional requirements in the system. We take this specification and design in it in ever-increasing detail so that it can be implemented in a language such as C#, C++ or Java.

In particular, we design a system using object, generic and interface architectures and we support both parallel/multi-threaded and sequential programming models. Furthermore, we apply parallel design patterns (such as Producer-Consumer) as well as the GOF and POSA patterns (and their improved versions) to speed up the development process and improve design quality.

## Major Topics in the Course

The following topics are discussed as a step-by-step process and they allow developers to design flexible and maintainable software systems using modern design techniques:

- System decomposition into loosely-coupled components
- The ability to design large, complex systems
- Integration with existing systems
- Interface, object and plug-and-socket architectures (OCA, ICA, PSA)
- Extending Gamma (GOF) and POSA patterns for ICA and PSA
- Multi-threaded and parallel design patterns
- How to do COD in C++ and C#: implementation mechanisms

New course since March 2010

## Prerequisites

The student should have a number of years experience in at least one modern OO language (such as C#, Java, C++) and project experience in a production environment is an asset.

## Who should attend?

Developers and designers who wish to design robust and flexible software systems using modern OOP. GP and parallel design methods. The percentage theory/practice is 65:35 and the exercises range from examples to show understanding of the topics up to component and application level.

## Course contents

### Day 1: High-level Design

#### Overview of High-level Design

- Mapping analysis to components and interfaces
- Specifying the architectural type and style
- Using system patterns to design systems

#### Choosing an Architectural Style

- Pipes and filters
- Blackboard and repository systems
- Event-based, implicit invocation
- Data abstraction and object-oriented organization
- State transition systems

#### System Decomposition

- Data and task decomposition
- Grouping and ordering tasks
- Data sharing
- Design evaluation

#### Components and Interfaces

- Component and interface definitions
- 'Provides' and 'requires' interfaces
- Conformance

### Day 2: Core Design

#### Overview of Core Design

- Choosing a connection architecture

- Detailed component decomposition
- Modelling component lifecycle

#### Object Connection Architecture (OCA)

- Properties of object-oriented systems
- ISA and HASA relationships
- Disadvantages of OCA approach

#### Interface Connection Architecture (ICA)

- Predicting system behaviour before it is built
- Varieties of component/module specifications
- Provided and required features
- Data type compatibility

#### Plug and Socket Architecture (PSA)

- Modelling complex topologies
- Interface grouping
- Services (plugs) and dual services (socket)
- Refining communication detail without increasing complexity

#### Choosing an Architecture

- Functional and non-functional requirements
- Checklist of questions to determine OCA, ICA, PSA
- Substitutivity of conforming components
- Performance issues

### Detailed Component Decomposition

- Whole-part decomposition
- Layered decomposition
- Separation of data and UI: PAC and MVC models

### Data Decomposition

- Divide and conquer
- Geometric decomposition
- Recursive data pattern

### Review of Core Design

- Creating a Proof-of-Concept in an OO language
- Conformance and Compliance
- Component and class diagrams
- Physical diagrams

### Day 3: Implementation Mechanisms

#### Overview of this Stage

- Gamma (GOF) patterns using OCA, ICA and PSA
- Multi-threaded and parallel design patterns
- Specific examples in C++ and C#
- Test cases and applications

#### Gamma Pattern Overview: which patterns are most useful?

- Object factories and Builder patterns
- Flyweight pattern with reference counting
- Bridge, Composite, Proxy

- Command, Strategy, Observer, Visitor

#### Generic Gamma Patterns

- Implementing GOF in C++ and C#
- Using boost libraries (Function, Signals)
- GP approach versus OOP: which one to use and when
- Test case: a generic Command pattern

#### Performance Tuning

- Avoiding unnecessary object creation
- Loop fission and fusion

- Dynamic polymorphism versus static polymorphism (CRTP pattern)
- Shared versus private data

#### Multi-threaded Patterns

- Master/worker
- Fork and join
- Shared queue/producer consumer
- Loop parallelism

#### Languages and Libraries

- Threads, synchronization and notification
- OpenMP and boost Thread
- C# Thread and parallel libraries
- Speedup, performance and troubleshooting

## Advanced Design and System Patterns

The main goal of this course is to learn the popular system patterns (also known as POSA and Buschmann patterns) and to apply them to structuring complex software systems. These patterns are distillations of best practices in software design and development and we apply them to systems using object-oriented and component-based models. We discuss each pattern in detail and we give guidelines on how to design and implement them. Furthermore, we integrate them into Domain Architectures and the design blueprints from the COA and COD courses. We also discuss how the class-level GOF patterns can be included into the POSA framework.

This unique course discusses how to design robust and flexible software systems using modern object oriented patterns.

Course contents updated May 2008

#### What do you learn?

- The complete POSA patterns (and more)
- Integrating GOF patterns into system patterns
- Multi-threaded and parallel patterns
- Robust, efficient and maintainable software designs

#### What previous Delegates have said about the Course

- "Very many practical tips, good combination theory and practice"
- "Good/very good"
- "Useful and motivating"

The percentage theory:practice is approximately 65:35.

#### Prerequisites

This course is for those software designers who have real hands-on experience of Gamma patterns and who wish to gain experience of those patterns that are needed and used for large, distributed and concurrent systems. It is not enough just to have read the Gamma book; you should have applied patterns such as Bridge, Factory, Visitor and State in applications. **This is NOT a beginner's course!**

#### Who should attend?

Software architects, senior designers and others who wish to gain insight into creating reusable software architectures for large systems.

## Course contents

#### A quick Review of the GOF Patterns

- Pattern categories
- Most important patterns in real-life applications
- Precise documentation of patterns using UML 2.0
- Extensions and adaptations of GOF patterns
- What are the most important patterns?

#### Pattern Languages using GOF patterns

- Combining inheritance and composition
- Discovering and inventing new pattern combinations
- Combining Creational, Structural and Behavioural patterns
- Reusing pattern languages

#### Generic Patterns: Fundamentals

- What is genericity, exactly?
- Generic programming versus

object-oriented programming

- Subtype and parametric polymorphism
- Creating generic versions of GOF patterns

#### Generic Patterns: Design and Applications

- Services and interfaces (provides and requires)
- Traits and traits classes
- Policies and their relationship with contracts
- Policy-based design
- Enriched policies

#### The POSA Patterns: Overview

- What is POSA?
- Architectural patterns in POSA
- Design patterns in POSA
- Relationship with GOF patterns

#### Structural Decomposition

- Whole-Part pattern

- Examples from application areas
- Comparing Whole-Part with Composite
- Combinations

#### Management

- Command
- Command Processor
- Generic Command
- Combining Command and Visitor
- View Handler pattern

#### Communication

- Mediator and its role in large systems
- The 6 Proxy types
- The Propagator pattern
- Forward-Receiver pattern
- Client-Dispatcher-Server pattern
- Why Observer is not always the best solution

### Organisation of Work

- The Builder pattern
- Applying Builder to create arbitrary object networks
- Master-Slave pattern
- Master-Slave: threads and parallel versions
- Facet (Role) pattern

### Interactive Systems

- Model-View-Controller (MVC)
- Presentation-Abstraction-Control (PAC)
- PAC versus MVC; which one (or both)?
- Creating large applications with PAC

### Parallel Decomposition Techniques

- Task and data decomposition
- Grouping and ordering tasks
- Data sharing among tasks

### Algorithm Structure

- Task and data parallelism
- Divide and conquer
- Geometric decomposition
- Other decomposition techniques

### Multithreading

- An introduction to multithreading
- Threads and thread lifecycle
- Mutexes, lock and condition variables

### Designing Threaded Applications

- Thread models
- Boss-Worker
- Crew model
- Assembly line
- 

### High-level Structures

- Pipes and Filters
- Broker
- Blackboard
- The steps in designing a Blackboard application

### An Introduction to Domain Architectures

- What is a domain architecture?
- The five DA categories
- Using domain architectures to structure POS and GOF patterns
- Embedding POSA and GOF patterns in a DA setting
- Examples and test cases

## Parallel and Multi-Threaded Design Patterns: Next generation GOF and POSA

The goal of this course is to apply data/task decomposition and parallel design principles to creating software systems in shared memory models and for multi-core CPUs. Many applications' performance can be improved by using data and task decomposition methods to produce a system consisting of loosely-coupled components. We then map these components to processors and threads to improve the speedup of the application.

### Overview of the course

This three-day hands-on courses analyses, designs and implements multi-threaded software applications using modern parallel design methods First, we analyse systems by partitioning them into semi-independent tasks. Second, after having identified exploitable concurrency, we then refine the design by deciding how to choose the most appropriate algorithms which strike a balance between abstraction and portability on the one hand and suitability for a particular target architecture, on the other hand. Third, we discuss commonly-used shared data structures to create source code that best supports the algorithms that we are working on. Finally, we introduce implementation mechanisms for thread/process management, synchronization and communication and our examples will be based on OpenMP, boost C++ threads, OpenCL, and C# threads.

### What do you learn?

This is one of the first courses to analyse, design and implement computationally-intensive applications in parallel and multi-threaded hardware environments. We discuss all important issues such as:

- Analyse and design problems with 'parallelism' in mind
- Efficient and flexible applications by using the most appropriate parallel patterns
- How to implement multi-threaded patterns in C++ and C#
- Introduction to OpenMP, boost threads, C# threads and OpenCL
- 'Thinking parallel'

### Prerequisites

A working knowledge of some high-level programming language such as C++, C# or Java is necessary. Some basic knowledge of multi-threading is advantageous.

### Who should attend?

Software developers, designers and architects who wish to obtain an understanding of the full software process using parallel and multi-threaded programming techniques. We deal with data and task decomposition, algorithmic structure, detailed design and implementations in C++, C# and OpenCL.

This course is ideally suitable for computationally-intensive applications, for example. The percentage theory/practice is 75/25.

# Course contents

## Part I: Background and Terminology

### Introduction to Parallel Architectures

- Communication architecture
- Shared address space
- Message passing
- Data flow architectures
- Flynn's taxonomy

### Performance Analysis Formulae

- Amdahl's law
- Gustafson-Barsis
- Karp-Flatt
- Isoefficiency relation
- Parallel applications and serial fraction threshold

### Some Parallel Environments and Libraries

- OpenMP
- Boost Thread and Interprocess
- C# Threads
- Concurrent Java
- Message Passing Interface (MPI)
- OpenCL and GPU Technology

## Part II: Finding Concurrency: Data and Task Decomposition

### Introduction to Graph Theory

- Nodes and edges
- Directed and undirected graphs
- Weighted graphs
- Shortest paths in graphs
- Applications to parallel programming

### System Development and Dependency Graphs

- Decomposition patterns
- Data and task decomposition
- Array-based and recursive data structures
- Choosing between data and task decomposition

### Dependency Analysis

- Grouping and ordering tasks
- Temporal dependencies and other requirements
- Data sharing among tasks
- Kinds of shared data

### Dependency Graph Evaluation

- Is it suitable for target platform?
- Design quality
- Synchronous and a-synchronous interaction
- Regular or irregular dependencies?

## Part III: Parallel Design and Algorithm Structure

### Architectural Styles

- Pipes and filters; dataflow
- Hierarchical repository and Blackboard
- Event-driven systems and observers
- Layered systems

### Algorithm Structure Decision Tree

- How to organize (by task, data decomposition or flow of data)
- Linear and recursive task organization
- Geometric decomposition and recursive data structures
- Data flow using pipelines and event-based coordination

### Task Parallelism

- Suitability and applicability
- Dependencies and schedule
- Removable and separable dependencies
- Load balancing

### Divide and Conquer Pattern

- Problems, sub-problems, sub-solutions and solutions
- Mapping to processors and threads
- Communication costs and dealing with dependencies
- Special cases: Master-Worker, Fork-and-Join

### Geometric Decomposition Pattern

- Dividing data into chunks
- Inter-chunk dependencies
- Data distribution and task scheduling
- Examples

## Part IV: Supporting Structures and Multi-threaded Design Patterns

### Introduction and Forces

- Program structure and data structure
- Relationship with Algorithms Structure patterns
- Quality issues (scalability, efficiency, ...)

### Program Structure

- SPMD (Single Program, Multiple Data)
- Master/Worker
- Loop parallelism
- Fork/join
- Applicability issues

### Data Structures

- Shared data
- Shared queue
- Distributed array
- Replication and reduction
- Locking and data races

## Part V: Implementation Mechanisms in C++ and C#

### Introduction

- What is an implementation mechanism?
- Mapping instructions to processes and threads
- Thread lifecycle
- Synchronisation and communication

### Threads and Processes

- Creation and destruction
- Master and child threads

- Memory synchronization: fences and barriers

### Communication Operations

- Broadcast
- Barriers
- Reduction

### Debugging and Troubleshooting

- Serial equivalence
- Race conditions
- Memory consistency problems
- Speedup problems and their resolution

### Introduction to OpenMP

- Overview of OpenMP
- API Directives, runtime functions, environment variables
- Shared and private variables
- Loop-level parallelization
- Work-share constructs
- Nested parallelism

### Introduction to Boost Thread

- The thread class and its lifecycle
- Creating threads: static function, thread function and callable object
- Thread synchronization and notification
- Mutex and locking classes

### Threads in C#

- Implementing the shared data metaphor
- Passing data to a thread
- Mutex and Semaphore in C#
- Thread safety

### Synchronisation in C#

- Blocking methods
- Sleep and EndInvoke
- Join
- Spinning
- Locking and atomicity

### An Introduction to OpenCL

- OpenCL Models
- Platform model
- Memory model
- Execution model
- Programming model

### Applications and Test Cases

- Multi-threaded C# for optical technology (graphics)
- Optimisation problems and matrix algebra
- Monte Carlo simulation
- Financial option pricing

# Advanced C++ - Programming Models, Libraries and Parallel Computation

The goal of this hands-on distance learning course is to learn the most advanced features of object-oriented and generic programming in C++, the STL and boost libraries and modern software design methods. We have developed this course for those professionals working in business, engineering and other areas who are involved in software development. The nine modules in the course represent state of the start developments in C++.

## What do you learn?

- Modern object-oriented and generic programming techniques
- STL and boost libraries from A to Z
- Multi-threaded and parallel programming in C++
- Design techniques and creating applications

## Prerequisites

A reasonably good knowledge of the C++ syntax. We have a review module in the course to bring those students who need to refresh their C++ knowledge.

## Who should attend?

For all developers and designers who use C++ as development language.

## How does Distance Learning work?

The current distance learning formula is based on our other distance learning programs. The main steps in the process are:

- The student registers for the Advanced C++ distance learning course by sending us a signed registration form (which can be printed from our website) after which we will send an invoice. Once we have received the full payment in our bank account we

will enable you as a user on the Advanced C++ distance learning forum.

- We send you 3 textbooks on C++, as well as source code, manuscripts and exercises (per module) in the post.
- For each module, we have a special forum section where you can find screen capture/audios, videos, user area and ongoing developments
- We include a step-by-step account on how to learn each module, and how to test your knowledge after you have finished the module (including sending your finished exercises to the mentor and other feedback)
- Optionally, it is possible to do an examination and do a project. This is included in the price of the course.
- On successful completion, you will be presented with a certificate.

Typically, the duration of the course is one year. This estimate is based on the fact that most students have a full-time job and will do the course in their free time.

Your mentor is Dr. Daniel J. Duffy

## Course contents

### Module 1: Quick Review of C++ Essentials

#### General Considerations

- The canonical class definition
- Why const is important
- Raw and smart pointers
- Robust C++ code: guidelines

#### Advanced Overloading

- Overloading index operators [] and ()
- The assignment operator and memory management
- Overloading the ostream operator <<
- Functors and function objects
- Comparing functors with function pointers

#### Simple Inheritance

- Inheritance and ISA Relationship
- Specialisation Scenarios
- Inheritance and Object Creation
- Using Base Class Constructors
- Accessibility of Base Members
- Overriding Functions

#### Polymorphism

- Pointers to the Base Class
- Function Visibility
- Polymorphism
- Defining an Interface
- Abstract Base Classes
- Virtual Destructors
- Operator Overloading and Inheritance
- 

### Module 2: Generic Programming and Policy-based Design

#### Programming with Templates I

- Multiple parameters
- Nested template class
- Inheritance and composition
- Compile-time and fixed-sized array classes

#### Programming with Templates II

- Default parameter values
- Template template parameters
- Some templated design patterns
- Template specialization; partial specialisation

#### Templated Software Components

- Traits classes
- Services and policy-based design
- 'Provides' and 'requires' interfaces
- Implementing policies in C++

#### Advanced GOF: Combining Components into larger Components

- Creating pattern languages
- Creating networks of inter-related patterns
- Using GOF patterns in larger architectures
- Finding the right patterns
- Contracts and where clauses

#### Generic Patterns and Generic Programming

- An introduction to generic programming

- Comparing OOP with GP
- Designing components in GP framework
- 'Provides' and 'requires' interfaces

#### The Design of Generic Components

- Traits and their applications
- Policy classes
- Combining policies and traits
- Test Case: a policy-based templated Command and Proxy patterns
- Examples

### Module 3: Standard Template Library (STL)

#### Overview of Standard Template Library (STL)

- What is STL?
- STL Components
- Containers
- Main Container Types
- Algorithms
- Main Algorithm Categories
- Set-like Operations
- Iterators
- Function Objects
- Adaptors
- Allocators
- Strengths and Limitations of STL
- Student Prerequisite Knowledge

#### STL Containers

- Sequence Containers
- Vector
- Deque
- List

## Sorted Associative Containers

- Multisets (Bags)
- Sets
- Set\_like Operations on Sorted Structures
- Multimaps
- Maps

## Iterators in STL

- What is an Iterator?
- Iterator Categories
- Iterator functions
- Iterator functions: Input Iterators
- Output Iterator Types
- Forward Iterators
- Bi-directional Iterators
- Random Access Iterators
- Qualifying Iterators: Mutable and Constant Iterators

## Algorithms in STL

- Overview of STL Algorithms
- Algorithm Categories
- Algorithms with Function Parameters
- Non-mutating Sequence Algorithms
- Mutating Sequence Algorithms
- Sorting and searching

## Module 4: Boost Containers, Data Structures and Higher-Order Programming

### MultiArray

- Creating n-dimensional data structures
- Performance issues compared to STL
- Slicing and views
- Resize, reshape and storage
- Multi-index and sub-object searching

### Range

- Modelling pairs of iterators
- Using ranges with generic algorithms and STL containers
- Raising the abstraction level
- Using metafunctions

### Tuple

- Modelling n-tuples (pair is a 2-tuple)
- Using tuples as function arguments and return types
- Accessing the elements of a tuple
- Advantages and applications of tuples

### Variant

- Creating discriminated unions with heterogeneous types
- Manipulating several distinct types in a uniform manner
- Type-safe visitation
- Avoiding type-switching for variant data

### Any

- Value-based variant types
- Discriminated types
- Typesafe storage and retrieval Applications of Any

### Multi-Index Containers

- Bidirectional maps
- Sets with several iteration orders
- Emulation of standard containers
- MRU lists
- Category: Function Objects and

## Higher-Order Programming

### Bind

- Generalising and improving the STL Bind
- Uniform syntax for functors, (member) function pointers
- Functional composition and nested binders
- Bind as used in Boost.Function

### Function

- Generalised callback mechanisms
- Storage and invocation of functors, (member) function pointers
- Useful in notification patterns (Observer, Signals and Slots)
- Example: separating GUIs from business logic

### Signals and Slots

- Implementation of Observer (Publisher-Subscriber) pattern
- Event management with minimal inter-object dependencies
- Signals == Subject, Slots == Subscriber
- Application to Mediator and Observer patterns

### Lambda

- Unnamed functions
- Useful for STL algorithms
- Avoiding creation of many small function objects
- Less code: write function at location where it is needed

## Module 5: Boost I/O and other Utilities

### Filesystem

- Portable manipulation of paths, directories and files
- Defining functionality as in scripting languages
- Platform portability

### Serialisation

- Saving arbitrary data to an archive (e.g. XML)
- Restoring data from an archive
- Versioning

### Regex

- Regular expressions and pattern matching
- Processing large and inexact strings
- Emulating functionality as in Perl, awk, sed

### Spirit

- Functional, recursive descent parser generator framework
- Command-line parsers
- Specifying grammar rules in C++ (EBNF syntax)
- Performance issues

### Tokenizer

- Separate character sequences into tokens
- Finding data in delimited text streams
- User-defined delimiters

## Module 6: Boost Interprocess and Network Communication

## Asynchronous Communication

- Network and low-level I/O
- Proactor design patterns
- Strands
- Custom memory allocation

## Networking

- TCP, UDP, ICMP
- Socket I/O streams
- SSL support
- Serial ports

## Interprocess

- Shared memory
- Memory-mapped files
- Semaphores and mutexes
- File locking
- Message queues

## UML Statecharts in Boost

- Hierarchical (composite, nested) states
- Orthogonal states
- Transitions and Guards
- Event delay

## Module 7: Multi-threaded and Parallel Programming

### Boost Thread

- Threads and thread groups
- Synchronisation (mutex, locks, barriers etc.)
- Thread local storage
- Date and time requirements

### Message Passing Interface (MPI) in Boost

- An Introduction to MPI
- What is distributed computing?
- Message passing metaphor
- Fundamental MPI concepts
- MPI library functions

### MPI Core

- Communicators Point-to-point communication
- Collective communication
- MPI data type
- Separating form and content

## Module 8: OpenMP, an Introduction

### Overview

- Compiler directives
- Library routines
- Environment variables

### My First OpenMP Program

- Writing the serial program
- Determining parallel code
- Adding OpenMP directives
- Debugging and performance measurement

### Data Clauses in OpenMP

- Shared and private
- Lastprivate, firstprivate
- Default and nowait clause

### OpenMP Synchronisation Constructs

- Barrier
- Ordered
- Critical and Atomic

- Locks, Master construct

### Work Sharing in OpenMP

- Loop construct
- Sections and section
- Single construct
- Combined parallel work-sharing constructs
- Other Clauses
- Reduction clause
- Copyin clause
- Copyprivate clause
- Ordered clause

### Configuration and Run-Time Information

- Setting environment variables' values
- Library functions for thread information
- Scheduling functions
- Lock functions
- Timing functions

### Module 9: Applications: Design and Implementation

#### Detailed Software Requirements for Components

- Throwaway, non-throwaway and production software
- What are the top software requirements?
- Functional and non-Functional requirements (FRs and NFRs)
- How FRs and NFRs affect component design

### Combining Component and Object Technologies

- Comparing Component and Object Design
- The differences between OOD and COD
- Combining components and objects
- Assemblies and namespaces
- Developing components from objects
- Component loading and the object instantiation process

### Using Components and Objects for GOF Patterns

- When to use interfaces and when to use abstract classes
- Using classes and objects in combination with components
- Stateless and Stateful GOF patterns
- Delegation and Composition

### Designing C++ Applications

- Choice of programming models
- Complexity Analysis and data structures
- Which STL and boost libraries to use
- Design patterns

### Performance of C++ I

- Classifying and discovering performance bottlenecks
- Virtual versus non-virtual functions
- Preventing unnecessary object creation
- Exceptional handling

### Performance of C++ II

- Templates versus inheritance
- Using the appropriate data structures from STL
- Loop optimizing techniques
- Loop fission, fusion, unrolling and tiling

