FEABHAS

Modern C++ for Embedded Systems (C++11/14/17)

Course category	C++ Training Courses
Training area	Programming Languages
Course code	C++11-501
Duration	5 days
Price exc VAT	£3000.00

C++ is a remarkably powerful systems-programming language, combining multiple programming paradigms – Procedural, Object Oriented and Generic – with a small, highly-efficient run-time environment. This makes it a strong candidate for building complex high-performance embedded systems.

The C++11 standard marked a fundamental change to the C++ language, introducing new idioms and more effective ways to build systems. This new style of programming is referred to as 'Modern C++'.

This practical, hands-on course introduces the C++ language for use on resource-constrained, real-time embedded applications.

The course highlights areas of concern for real-time and embedded development. The focus is on developing effective, maintainable and efficient C++ programs.

The course covers C++11, C++14 and C++17 and where relevant refers to C++20.

Overview:

A five-day course that provides a practical overview of C++ focusing on developing object-oriented programs in an embedded, real-time environment.

Course objectives:

- To provide a solid understanding of the essentials of the C++ programming language.
- To give you practical experience of writing Modern C++ for resource-constrained real-time and embedded systems.
- To give you the confidence to apply these new concepts to your next project.

Delegates will learn:

- Modern C++ syntax and semantics and idioms

- Using C++ for hardware manipulation
- The Application Binary Interface (ABI) and memory model of C++
- Idioms and patterns for building effective C++ programs

Pre-requisites:

- A strong working knowledge of C
- Embedded development skills are useful, but not essential

Who should attend:

This course is aimed at C programmers who are moving to C++ for their embedded development.

Duration:

- Five days

Course materials:

- Delegate manual
- Delegate workbook

Course workshop:

Attendees perform hands-on embedded programming, during course practicals. Approximately 50% of the course is given over to practical work.

The board targeted is an ARM Cortex-M based MCU which gives attendees a real sense of embedded application development.

Program structure

- "Hello World!"
- The build process
- Object files
- Linking Activities

Stream I/O

- C++ stream I/O objects
- Stream modifiers

The C++ object model

- Objects and types
- Brace initialisation
- Object visibility scope
- Object lifetime

Constants

- Literals
- Compile-time constant expressions
- Enum classes

Pointers

- Dynamic objects
- The value of an ‘empty’ pointer
- Pointers and const

Hardware manipulation

- Memory-mapped registers are accessed via pointers
- volatile objects
- Bitwise Operators
- General Purpose Input / Output (GPIO)

Structures

- User-defined types
- Packing
- Performance implications of packing

Arrays

- Containers
- The iterator model
- Range-for statement
- Applying algorithms to arrays – filling and sorting

Functions

- The 'One Declaration Rule'

- How function arguments are passed
- The overheads of pass-by-value
- Pass-by-reference
- std::tuple represents a general n-tuple
- Function overloading
- Function inlining

Structuring code

- Separating Interface and Implementation
- Compilation Dependencies
- Using the __cplusplus macro

Namespaces

- Defining functions within a namespace
- Ambiguity when accessing namespace members

Principles of Object Oriented Design

- Coupling
- Encapsulation
- Cohesion
- The Single Responsibility Principle
- Abstraction

User-defined types

- Creating new types
- Access specifiers
- 'this' pointer

Initialising objects

- Non-Static Data Member Initializers (NSDMIs)
- The compiler-supplied default constructor
- The class destructor

Objects and functions

- Pass-by-value
- Explicit constructors

- Disabling pass-by-value
- Disabling copying
- Return Value Optimisation (RVO)
- 'Copy elision'

Static

- The static storage specifier
- Static member variables
- Static member functions

Object-based I/O

- An Object-Oriented approach to I/O
- Using a struct for I/O device access
- Nesting a structure overlay in a class

Operator overloading

- Problem domain types
- Overloading the stream operator
- Conversions to other user-defined types

Building composite objects

- Composite aggregation
- Overriding default initialisers
- The composite object on the stack...

Connecting objects

- Connected objects form a system
- Forming the Association (Client-Server)
- Bi-directional associations…
- Friend functions
- Forward references to namespace elements

Creating Substitutable Objects

- Specialisation
- Inheritance
- Overriding base class behaviour

- The 'protected interface'
- The Liskov Substitution Principle
- Late binding (of polymorphic operations)
- Dynamic binding

Abstract Base Classes

- An abstract class
- Extending derived classes
- Safely accessing the extended interface

Realising Interfaces

- The Dependency Inversion Principle
- Provided and Required Interfaces
- The Interface Segregation Principle
- Cross-casting

Dynamic objects

- Memory model
- std::unique_ptr allows single ownership
- std::shared_ptr is reference-counted
- std::weak_ptr
- Resolving circular dependency issues

Dynamic containers

- Sequence containers
- std::vector is a dynamically-resizable array
- std::list is a doubly-linked list
- Key-value containers – std::pair
- Sorted containers
- Containers and memory allocation

Callable objects

- Function objects
- Lambdas
- Under the hood
- Using std::function for call-back
- Containers of callable objects

Initializer lists

- std::initializer_list
- initializer_list overload rules

Template functions

- Generic programming
- Function templates
- Overloading with template functions

Template classes

- Class templates
- Lazy instantiation
- Member Functions of Class Templates