FEABHAS

Advanced Modern C++ for Embedded Developers (C++11/14/17)

Course category	C++ Training Courses
Training area	Programming Languages
Course code	AC++11-501
Duration	5 days
Additional information	Available for on-site delivery only. Can be delivered remotely or Face-to-Face.

The term 'Modern C++' is used to describe the current best practices for the use of C++. In some cases, this may mean new capabilities of the language; in other cases it means more effective ways of performing familiar programming tasks.

This practical, hands-on course expands on the Modern 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.

Course objectives:

- To provide a deep understanding of the Modern 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
- Real-time and concurrency design issues

Pre-requisites:

- A good working knowledge of Modern C++
- Embedded development skills are useful, but not essential

Who should attend:

This course is aimed at experienced Modern C++ programmers who want to apply Modern C++ to a low-level/deeply-embedded target environment.

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.

The C++ object model

- Declaration and definition
- Brace initialisation syntax
- ODR-use and 'The One Declaration Rule'
- Object scope and lifetime
- The C++ object (memory) model

The C++ build process

- The seven stages of compilation
- Object files
- Symbol tables
- Linkage
- ELF files
- Object conversion for embedded systems

The C++ object model

- Declaration and definition
- Brace initialisation syntax
- ODR-use and 'The One Declaration Rule'
- Object scope and lifetime

- The C++ object (memory) model

Expressions

- Expressions
- l-values and r-values
- statements
- Sequence points

User defined types

- Aggregate types – structs
- Brace elision
- Classes
- Non-Static Data Member Initialisers
- Delegating constructors
- std::initialize

Functions

- Function call ABIs
- Input, Output and Input-Output parameters
- const correctness
- Copy elision
- Attributes

Type deduction

- Automatic type deduction
- Automatic function return-type deduction
- Structured bindings
- Using aliases

Constants

- Literals
- Const qualification
- Constexpr
- constexpr functions
- enum class
- enum underlying type

Hardware manipulation

- Using pointers for I/O access
- Bit manipulation
- The volatile qualifier

Object-based I/O

- Nested pointer approaches
- Pointer-offset approaches
- Structure overlay approaches

Composition

- Nested object construction

Connecting objects

- Unidirectional Associations
- Bidirectional association
- Forward declarations

Bit fields and unions

- Bit field structures
- Bit field structure overlay for hardware
- The size of a bit field structure
- Alignment issues
- Unions
- Using unions and bit field structures together

Creating substitutable types

- Specialisation vs inheritance
- Substitution
- The Liskov Substitution principle
- The virtual function ABI

Abstract Base Classes

- The Single Responsibility principle
- Pure virtual functions

- Abstract types
- Dynamic cast

Realising interfaces

- The Dependency Inversion principle
- The Interface concept
- Pure virtual classes
- The Interface Segregation principle

STL containers

- The problems with C-style arrays
- std::array
- Dynamic sequence containers
- Sets and maps
- Hash-maps - std::unordered_map
- Emplacement

Algorithms

- The iterator model
- Range-for
- Algorithms

Callable objects

- Lambda expressions
- The 'block-scoped function' concept
- Generic lambdas
- std::function

Resource management

- The resource lifetime problem
- Overloading the copy constructor
- Overloading the assignment operator
- The 'Rule of the Big Three'
- The copy-swap idiom

Move semantics

- The cost of copying
- 'Resource pilfering'
- Move constructors
- The Rule of Four and A Half
- Move assignment
- std::move
- Compiler overload provision for copy / swap

Smart pointers

- The problem with raw pointers for memory management
- std::unique_ptr
- std::shared_ptr
- std_weak_ptr

Template functions

- The problems with function-like macros
- Template functions
- Template parameter type deduction
- The forwarding reference idiom

Template classes

- Generic classes
- Template type deduction
- Template deduction guidelines

Templates and polymorphism

- The cost of virtual interfaces
- Policy patterns

Perfect forwarding

- Variadic templates
- std::forward
- std::forward vs std::move

Interrupts

- The interrupt mechanism

- Encapsulating an interrupt within a class
- Race conditions

Trait classes

- Making generic code more specific
- Compile-time lookup
- Trait classes
- Trait classes vs auto

Appendices

Appendix - User defined literals

- 'Rommable' types
- operator ""

Appendix - Associative containers

- std::set
- std::map
- std::unordered_map

Appendix - Variable types

- std::optional
- std::any
- std::variant
- The Visitor pattern

Appendix - STL allocators

- Replacing the STL allocator
- Memory resources
- Standard library memory resources
- Writing your own memory resources
- Polymorphic allocators

Appendix - Exception handling

- Error handling strategies
- Throwing/catching exceptions

- Building an exception hierarchy
- Standard library exceptions
- Specifying your exception contract

Appendix - Conditional coding

- Using the pre-processor for conditional inclusion
- Inline namespaces
- Tag dispatch
- SFINAE
- std::enable_if
- constexpr-if