

## Advanced Real-Time Modern C++ (C++11/14/17)

---

Course category	C++ Training Courses
Training area	Programming Languages
Course code	AC++11-502
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 advanced course is designed to transition experienced C++ programmers to the latest incarnation of the C++ language. The focus is to teach good programming practice using Modern C++ and to put the latest features of the language into context.

The course covers C++11, C++14 and C++17.

### Course objectives:

- To provide a deep understanding of the C++ programming language.
- To give you practical experience of writing Modern C++ on hosted embedded systems
- To give you the confidence to apply these new concepts to your next project.

### Delegates will learn:

- Modern C++ syntax, semantics and library features
- 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 C++ (knowledge of C++11 onwards is useful, but not essential)
- Some experience of development multi-threaded applications is useful.
- A working understanding of machine architectures is helpful.

**Who should attend:**

- This course is aimed at C++ programmers who are using earlier standards of C++, and experienced C++ programmers who want to extend and expand their C++ skills.

**Duration:**

- Five days.

**Course materials:**

- Delegate manual

**Course workshop:**

- Attendees perform hands-on exercises during course practicals. Approximately 50% of the course is given over to practical work. The tools used are indicative of current modern working practices in the embedded arena.

**Part 1 - Core concepts****Build process**

- The C++ build process
- The preprocessor stage
- Compiler and linker stages
- Linking embedded and hosted applications

**The C++ object model**

- Declaration and definition
- Brace initialisation syntax
- Pointers and references
- Empty pointer validation
- Temporary object materialization

**User defined types**

- The Single Responsibility Principle (SRP)
- Member variable initialisation
- Constructors

- Re-enabling the default constructor
- Delegating constructors
- Explicit constructors
- Inline initialisation of static member variables

## **Type deduction**

- Automatic type deduction
- Qualifying auto-deduced types
- The decltype operator
- Auto-deduced function return types
- Trailing return-type syntax

## **Constants**

- Numeric and character literals
- Enum classes
- constexpr objects
- constexpr functions and classes
- User-defined literals
- `std::string_view`

## **Functions call ABI**

- Function declaration and definition
- Procedure Activation Record
- Member functions

## **Functions**

- Function parameters
- Pass-by-value, pointer and reference
- const member functions
- Returning by value
- Named Return Value Optimisation (NRVO)
- Return Value Optimisation (RVO)
- Copy elision
- Factory functions
- Auto-deduced function return types
- Compiler diagnostics

## **Arrays and iterators**

- Arrays of objects
- `std::array`
- Arrays as parameters
- Iterators
- range-for statement

## **Vocabulary types**

- Structured bindings
- `std::pair` and `std::tuple`
- `std::optional` and `std::expected`
- `std::variant` and `std::visit`
- `std::any` and small buffer optimisation

## **Part 2 – Object Oriented Design**

### **Composite objects**

- Composition
- Aggregation
- Composite object initialisation
- Optional composite objects

### **Connecting objects**

- Unidirectional Associations
- Bidirectional association
- Forward declarations

### **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 functions
- The Interface Segregation principle
- Cross casting

## Part 3 - Standard Library

### Sequence containers

- `std::vector`
- `std::bitset`
- `std::list` and `std::forward_list`
- Container classes and `std::initializer_list`

### Associative containers

- `std::set`
- `std::pair` and `std::map`
- `std::unordered map`

### Algorithms

- The Standard Library model
- `std::fill` and `std::sort`
- `std::find`, `std::count` and `std::accumulate`
- The Remove-Erase idiom
- `std::transform` and `std::bind`
- `std::bind` placeholders

### Callable objects

- Functors
- Lambda functor syntax
- Lambdas as a block-scoped function
- Capture context
- Capture initialisers
- `std::function` and `std::invoke`

## Part 4 - Resource management

### Resource management

- Disabling copy constructor and assignment
- Deep copy constructor and assignment
- The Rule of the Three
- Copy-swap idiom
- Virtual copy constructor

## **Move semantics**

- Compiler generated copy operations
- 'Resource pilfering'
- L-value, R-value and PR-value expressions
- X-value objects
- Move constructor and assignment
- `std::move`
- Compiler generated move operations

## **Smart pointers**

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

## **C++ Strings**

- C-Strings (NTBS)
- Raw String Literals
- `std::string`
- Short String Optimization (SSO)
- `std::string_view`

## **Part 5 - Templates**

### **Template functions**

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

### **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`

## **Part 6 - Concurrency**

### **Threading**

- Concurrency vs parallelism
- `std::thread`
- Run policies
- Polymorphic threads
- Waiting for threads to finish
- Detaching threads

### **Mutual exclusion**

- Race conditions
- Mutual exclusion
- The scope-locked idiom
- `std::lock_guard` and `std::unique_lock`

### **Condition variables**

- Thread synchronisation
- The Guarded Suspension pattern

### **Atomic types**

- Shared objects in multiprocessor environments
- Why volatile is inappropriate

- Atomic types
- Load-acquire / Store-release barriers

### **Asynchronous tasks**

- The Asynchronous Message pattern
- The Promise / Future pattern
- `std::async` and packaged tasks
- Launch policies

**Feabhas Ltd** - [www.feabhas.com](http://www.feabhas.com)