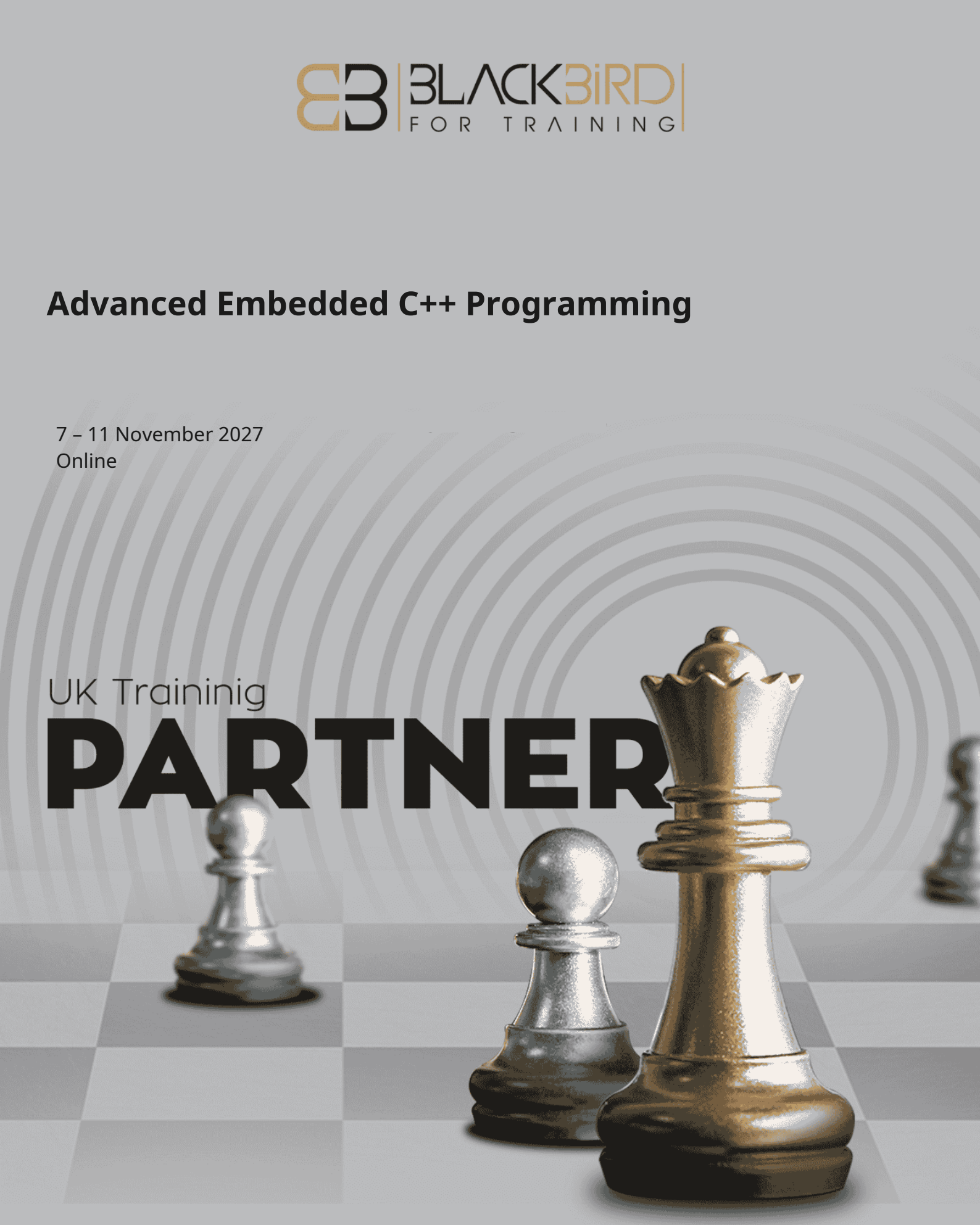
BLACKBIRD FOR TRAINING

Advanced Embedded C++ Programming

7 – 11 November 2027
Online



Advanced Embedded C++ Programming

Ref: 3255961_357233 **Date:** 7 – 11 November 2027 **Location:** Online **Fees:** 2700 GBP

Course Introduction

Embedded systems are at the heart of modern products and critical infrastructure, including automotive control units, industrial automation, aerospace platforms, medical devices, telecommunications equipment, robotics, consumer electronics, energy systems, and connected devices. As these systems become more complex, developers must produce software that is efficient, predictable, reliable, maintainable, and capable of operating within strict memory, processing, power, and timing constraints.

C++ provides powerful abstraction, type safety, modularity, and reusable design capabilities that can significantly improve embedded software development. However, using the language effectively in resource-constrained and real-time environments requires more than general programming knowledge. Developers must understand how language features affect memory allocation, execution time, code size, hardware access, compiler behavior, and system reliability.

Advanced embedded development presents several challenges. Software may need to interact directly with registers and peripherals, respond to interrupts within fixed deadlines, coordinate concurrent tasks, and operate continuously without failure. Dynamic memory allocation, uncontrolled object lifetimes, hidden copies, excessive template expansion, race conditions, and poorly designed abstractions can produce unpredictable behavior or unnecessary resource consumption. These issues are especially important in safety-critical and high-integrity applications.

The Advanced Embedded C++ Programming course develops the practical skills required to design and implement robust embedded software using modern C++ techniques. It focuses on applying object-oriented and generic programming without sacrificing deterministic behavior, runtime performance, memory efficiency, or visibility into hardware operations. Participants learn how to select appropriate language features and avoid practices that introduce unacceptable overhead or risk.

The course explores advanced type systems, resource management, compile-time programming, templates, move semantics, hardware abstraction, interrupt-safe design, concurrency, real-time considerations, and error-handling strategies. It also addresses software architecture, testability, debugging, performance measurement, and optimization for embedded targets.

Participants will examine how Resource Acquisition Is Initialization supports safe management of hardware resources, locks, communication channels, and peripheral states. They will also learn how smart pointers, ownership models, fixed-size containers, and controlled allocation policies can reduce memory errors while preserving predictable operation. Particular attention is given to determining when standard library features are suitable for embedded use and when specialized alternatives are necessary.



The program connects language-level techniques with practical hardware development. Participants will work with memory-mapped input and output, peripheral interfaces, interrupt service routines, device drivers, real-time operating system tasks, and communication mechanisms. They will learn how to create clear hardware abstraction layers that improve portability and testing without hiding important performance characteristics.

This course is suitable for embedded software engineers, firmware developers, systems programmers, electronics engineers, automotive developers, robotics specialists, technical leads, software architects, and professionals responsible for high-performance or safety-related embedded applications. It supports participants in building software that is not only technically functional, but also structured for long-term maintenance, verification, reuse, and controlled evolution.

Course Objectives

By the end of this course, participants will be able to:

- Apply advanced C++ features in resource-constrained embedded systems.
- Evaluate the runtime and memory impact of language features.
- Design deterministic and maintainable embedded software architectures.
- Implement safe object lifetime and resource ownership models.
- Use Resource Acquisition Is Initialization for hardware resource control.
- Apply templates and compile-time programming to reduce runtime overhead.
- Use move semantics and perfect forwarding appropriately.
- Develop type-safe interfaces for hardware registers and peripherals.
- Design reusable hardware abstraction layers and device drivers.
- Manage interrupts, shared resources, and concurrent execution safely.
- Apply real-time programming principles to embedded applications.
- Select suitable error-handling strategies without uncontrolled overhead.
- Reduce memory fragmentation and unsafe dynamic allocation.
- Use standard library components selectively in embedded environments.
- Measure, debug, profile, and optimize embedded C++ software.
- Develop a practical improvement plan for an embedded software project.



Course Modules

Day 1: Advanced C++ for Embedded Systems

- Embedded C++ design constraints.
- Language standards and compiler support.
- Zero-cost abstraction principles.
- Object layout and memory representation.
- Storage duration and object lifetime.
- Stack, static, and dynamic memory.
- Constructors and destructors.
- Copy and move operations.
- References, pointers, and ownership.
- Const correctness.
- Strong types and scoped enumerations.
- Undefined and implementation-defined behavior.

Day 2: Resource Management and Compile-Time Techniques

- Resource Acquisition Is Initialization.
- Deterministic resource cleanup.
- Ownership and borrowing models.
- Smart pointer suitability.
- Custom allocators and memory pools.
- Fixed-capacity containers.
- Template functions and classes.
- Template specialization.
- Compile-time constants.
- Constant expressions and evaluation.
- Type traits and type deduction.
- Compile-time interface validation.

Day 3: Hardware Abstraction and Driver Development

- Memory-mapped input and output.
- Volatile access and its limitations.
- Type-safe register representation.
- Bit manipulation techniques.
- Peripheral abstraction patterns.
- Device driver architecture.
- Static and dynamic polymorphism.
- Curiously recurring template pattern.
- Policy-based design.
- Dependency injection for embedded systems.
- Hardware-independent test interfaces.
- Portable board support layers.



Day 4: Concurrency, Interrupts, and Real-Time Behavior

- Interrupt service routine design.
- Interrupt latency and execution time.
- Shared data and race conditions.
- Atomic operations.
- Critical sections and synchronization.
- Lock-free programming considerations.
- Real-time operating system integration.
- Tasks, threads, and scheduling.
- Inter-task communication.
- Priority inversion and mitigation.
- Timing determinism and deadlines.
- Watchdogs and fault recovery.

Day 5: Reliability, Testing, and Optimization

- Embedded error-handling strategies.
- Exceptions and exception-free designs.
- Assertions and defensive programming.
- Static analysis and coding standards.
- Unit testing embedded software.
- Mocking hardware dependencies.
- Integration and target testing.
- Debugging optimized code.
- Execution-time measurement.
- Memory and code-size analysis.
- Practical embedded C++ refactoring exercise.
- Software quality improvement roadmap.

Why Should You Attend This Course?

- Use modern C++ without sacrificing embedded-system performance.
- Develop safer approaches to memory and resource management.
- Reduce defects caused by unclear ownership and object lifetimes.
- Build reusable and type-safe hardware interfaces.
- Improve interrupt, concurrency, and real-time software design.
- Control memory allocation and reduce fragmentation risks.
- Apply compile-time techniques to eliminate runtime overhead.
- Improve software portability across processors and platforms.
- Strengthen testing without depending entirely on physical hardware.
- Identify performance, timing, and code-size bottlenecks.
- Improve the maintainability of complex firmware projects.
- Apply advanced techniques directly to embedded development work.



Course Conclusion

Advanced embedded C++ development requires a careful balance between abstraction and control. The language offers powerful tools for building modular, reusable, and type-safe software, but each feature must be evaluated in relation to timing, memory, code size, hardware behavior, and system reliability.

This course equips participants with practical techniques for managing resources, developing hardware abstractions, controlling object lifetimes, applying compile-time programming, and designing concurrent and real-time software. It also strengthens their understanding of how compiler behavior and language choices influence the final embedded application.

Participants can apply the course outcomes by improving existing firmware architectures, replacing unsafe resource-handling practices, creating clearer peripheral interfaces, and introducing more reliable testing and measurement processes. They will also be able to make informed decisions about templates, dynamic memory, exceptions, standard library components, and concurrency mechanisms.

Over the long term, these capabilities support the development of embedded products that are more reliable, efficient, portable, and maintainable. They also help organizations reduce debugging effort, control technical debt, improve software quality, and manage increasingly complex embedded applications with greater confidence.

