Embedded C++

Embedded C++ (**EC++**) is a dialect of the C++ programming language for embedded systems. It was defined by an industry group led by major Japanese central processing unit (CPU) manufacturers, including NEC, Hitachi, Fujitsu, and Toshiba, to address the shortcomings of C++ for embedded applications. The goal of the effort is to preserve the most useful object-oriented features of the C++ language yet minimize code size while maximizing execution efficiency and making compiler construction simpler. The official website states the goal as to provide embedded systems programmers with a subset of C++ that is easy for the average C programmer to understand and use.

Object Oriented Language Characteristics

A large program in objected oriented language C++ or Java, splits into the logical groups (also known as classes).

• Each class defines the data and functions (methods) of using data.

• Each class can inherit another class elements

A set of these groups (classes) then gives an application program of the Embedded System

• Each group has internal user-level fields for data and has methods of processing that data at these fields

• Each group can then create many objects by copying the group and making it functional

Each object is functional. Each object can interact with other objects to process the user's data.

• The language provides for formation of classes by the definition of a group of objects having similar attributes and common behavior. A class creates the objects. An object is an instance of a class.

Embedded Programming in C++

• C++ is an object oriented Program (OOP) language, which in addition, supports the procedureoriented codes of C.

Program coding in C++ codes provides the advantage of objected oriented programming as well as the advantage of C and in-line assembly.

C++

struct that binds all the member functions together in C. But a C++ class has object features. It can be extended and child classes can be derived from it. A number of child classes can be derived from a common class. This feature is called polymorphism. A class can be declared as public or private. The data and methods access is restricted when a class is declared private. Struct does not have these features.

A class binds all the member functionsλ together for creating objects. The objects will have memory allocation as well as default assignments to its variables that are not declared static

A class can derive (inherit) from another class also. Creating a child class from RTCSWT as a parent class creates a new application of the RTCSWT.

Methods (C functions) can have sameλ name in the inherited class. This is called method overloading

Methods can have the same name as well as the same number and type of arguments in the inherited class. This is called method overriding. These are the two significant features that are extremely useful in a large program.

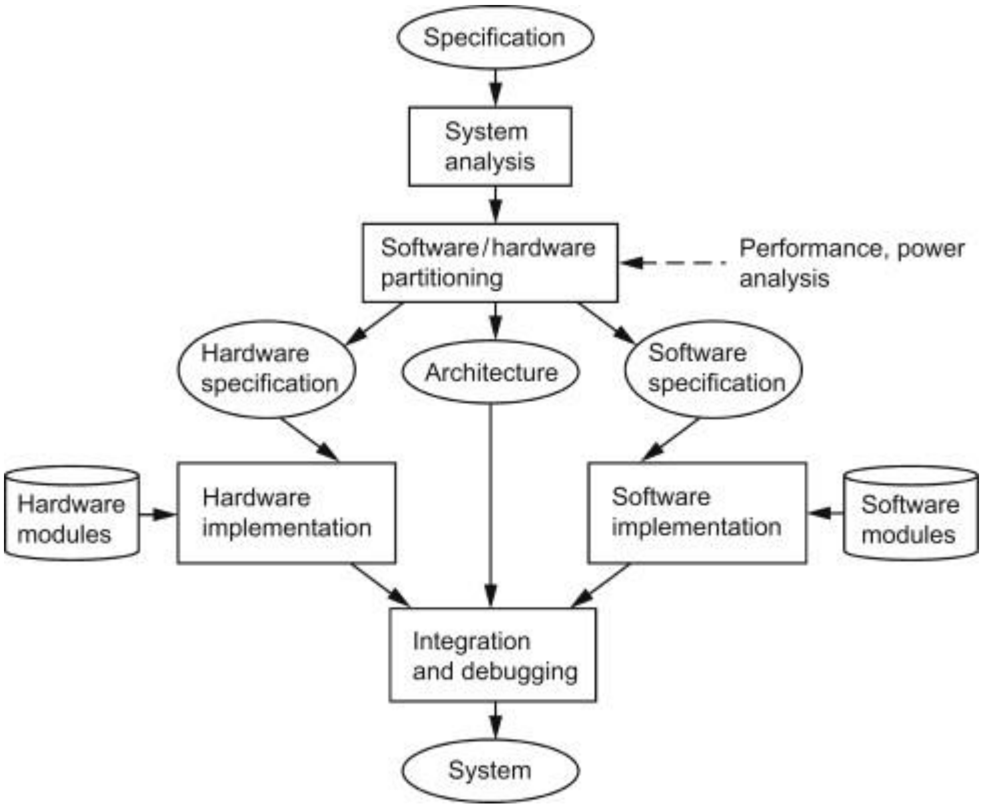
Operators in C++ can be overloaded like in method overloading.

For example, operators ++ and ! areλ overloaded to perform a set of operations.

Some disadvantages

• Lengthier Code when using Template, Multiple Inheritance (Deriving a class from many parents), Exceptional handling, Virtual base classes and classes for IO Streams

Software Implementation



Software implementation advantages

(i) Easier to change when new hardware versions become available

(ii) Programmability for complex operations (

(iii) Faster development time

(iv) Modularity and portability

(v) Use of standard software engineering, modeling and RTOS tools.

(vi) Faster speed of operation of complex functions with high-speed microprocessors.

(vii) Less cost for simple systems

Software testing

Embedded systems software testing shares much in common with application software testing. In general, you test for four reasons:

• To find bugs in software (testing is the only way to do this)

• To reduce risk to both users and the company

• To reduce development and maintenance costs

• To improve performance

Unit Testing. Individual developers test at the module level by writing stub code to substitute for the rest of the system hardware and software. At this point in the development cycle, the tests focus on the logical performance of the code.

Typically, developers test with some average values, some high or low values, and some out-of-range values (to exercise the code's exception processing functionality). Unfortunately, these "black-box" derived test cases are seldom adequate to exercise more than a fraction of the total code in the module.

Regression Testing. It isn't enough to pass a test once. Every time the program is modified, it should be retested to assure that the changes didn't unintentionally "break" some unrelated behavior.

Called regression testing , these tests are usually automated through a test script. For example, if you design a set of 100 input/output (I/O) tests, the regression test script would automatically execute the 100 tests and compare the output against a "gold standard" output suite. Every time a change is made to any part of the code, the full regression suite runs on the modified code base to insure that something else wasn't broken in the process.

Different approaches: **functional testing** and **coverage testing** .

Functional testing (also known as black-box testing) selects tests that assess how well the implementation meets the requirements specification. Coverage testing (also known as white-box testing) selects cases that cause certain portions of the code to be executed.

Functional Tests. Functional testing is often called black-box testing because the test cases for functional tests are devised without reference to the actual code—that is, without looking "inside the box."

An embedded system has inputs and outputs and implements some algorithm between them. Black-box tests are based on what is known about which inputs should be acceptable and how they should relate to the outputs. Black-box tests know nothing about how the algorithm in between is implemented. Example black-box tests include:

• **Stress tests** : Tests that intentionally overload input channels, memory buffers, disk controllers, memory management systems, and so on.

• **Boundary value tests** : Inputs that represent "boundaries" within a particular range (for example, largest and smallest integers together with − 1, 0, + 1, for an integer input) and input values that should cause the output to transition across a similar boundary in the output range.

• **Exception tests** : Tests that should trigger a failure mode or exception mode.

• **Error guessing** : Tests based on prior experience with testing software or from testing similar programs.

• **Random tests** : Generally, the least productive form of testing but still widely used to evaluate the robustness of user-interface code.

• **Performance tests** : Because performance expectations are part of the product requirement, performance analysis falls within the sphere of functional testing.

Coverage Tests

The weakness of functional testing is that it rarely exercises all the code. Coverage tests attempt to avoid this weakness by (ideally) ensuring that each code statement, decision point, or decision path is exercised at least once. (Coverage testing also can show how much of your data space has been accessed.)

Also known as white-box tests or glass-box tests, coverage tests are devised with full knowledge of how the software is implemented, that is, with permission to "look inside the box." White-box tests are designed with the source code handy.

Example white-box tests include:

• **Statement coverage** : Test cases selected because they execute every Statement in the program at least once.

• **Decision or branch coverage** : Test cases chosen because they cause every

branch (both the true and false path) to be executed at least once.

• **Condition coverage** : Test cases chosen to force each condition (term) in a

decision to take on all possible logic values.

Debugging

The important technique to find and remove the number of errors or bugs or defects in a program is called Debugging. It is a multistep process in software development. It involves identifying the bug, finding the source of the bug and correcting the problem to make the program error-free. In software development, the developer can locate the code error in the program and remove it using this process. Hence, it plays a vital role in the entire software development lifecycle.

To perform the debugging process easily and efficiently, it is necessary to follow some techniques. The most commonly used debugging strategies are,

- Debugging by brute force
- Induction strategy
- Deduction strategy
- Backtracking strategy and
- Debugging by testing.

Debugging by brute force is the most commonly used technique. This is done by taking memory dumps of the program which contains a large amount of information with intermediate values and analyzing them, but analyzing the information and finding the bugs leads to a waste of time and effort.

Induction strategy includes the Location of relevant data, the Organization of data, the Devising hypothesis (provides possible causes of errors), and the Proving hypothesis.

Deduction strategy includes Identification of possible causes of bugs or hypothesis Elimination of possible causes using the information Refining of the hypothesis(analyzing one-by-one)

The backtracking strategy is used to locate errors in small programs. When an error occurs, the program is traced one step backward during the evaluation of values to find the cause of bug or error.

Debugging by testing is the conjunction with debugging by induction and debugging by deduction technique. The test cases used in debugging are different from the test cases used in the testing process.

System on chip

An embedded system is any computer which is embedded in some piece of equipment and effectively runs a single program for the whole of its life. It may be quite a powerful computer - a high end PC - or even a network of such devices.

A System on Chip (SoC) is a single chip which contains a CPU, a number of peripherals, and sometimes the memory for a complete system. It does not have a general purpose bus to allow it to access random additional devices. Instead, all its pins are dedicated to built in peripherals (and possibly external memory). The set of devices any single SoC can drive is defined at the time of manufacture.

Most embedded systems are built round an SoC. But the more capable SoCs could run a low end tablet, which would not be called an embedded system, and as mentioned, much larger computers might be embedded.