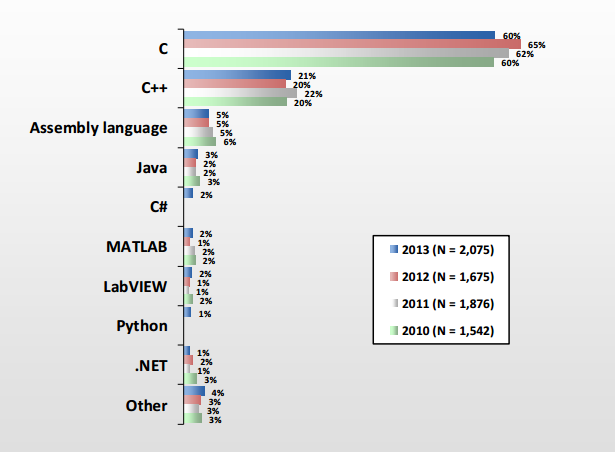
**Embedded Systems and C++ Programming Language**

C++ is an object-oriented/procedural programming language created by Bjarne Stroustrup in 1979. It is a superset of C programming language. That is in 99% cases a C program should compile fine with a C++ compiler. In this paper, I am referencing C++11 standard as few compilers have yet to support the C++14 standard. How does this relate to the embedded system world? Below is a picture showing the usage of programming languages in embedded systems.



\* See references section [1]

C++ is typically used in Automobile, Military Aircraft, and Medical Device industries. It is typically used on 16-bit and 32-bit microcontrollers, but can be used in 8-bit microcontrollers. However care must be taken not to exhaust memory or speed constraints. More care must be taken when using language features such as the STL (Standard Template Library), exceptions, and dynamic memory allocation. In real-time and/or safety-critical applications, how one uses language features of C++ is extremely important. It typically requires the knowledge of behind-the-scenes implementation of certain constructs such as virtual functions for example.

The advantages of C++ over C include object-oriented principles such as encapsulation, inheritance, and polymorphism. These principles allow code-reuse, more readability, maintainability, and modularity. Templates in C++ can also reduce code-reuse and increase readability and extendibility. However, with these additions can come pitfalls such as decrease in run-time performance and increase in code size which can break an embedded system. Let’s examine in more details the additional constructs that make programming easier and some of the pitfalls related to those constructs within the context of embedded systems.

Function overloading allows one to use the same name as an existing function or method but use a different signature. For example, “void temp(int a, int b)” and “void temp(int a)” are both legal in C++. This is handled at compile-time so as far as I know there are no disadvantages of use in an embedded system. Another construct is creating method or function as ‘inline.’ What this means is that one can define a method in a class declaration or use the inline keyword before a method or function definition. This tells the compiler to replace every invocation with the corresponding code in the method or function. This eliminates the call and return instructions which increases run-time performance. However, if not used wisely, inline routines can increase code size. In addition, it is not mandatory that the compiler “inline” a routine. Another construct is polymorphism. This allows an object to change behavior at run-time. This allows code-reuse and dynamic behavior through inheritance. By declaring a method as “virtual” and then overriding the method in the derived class, one can call the method through the base class and get the derived class’s behavior. However, underneath a table is created and lookup is needed at runtime in order to select the correct method to call at run-time. This increases code size and decreases run-time performance. However, it is deterministic. A construct that is not deterministic is dynamic allocation using the “new” and “delete” keywords because memory can become fragmented overtime and underlying implementation details can vary between compiler suites. Exceptions are also not deterministic because underlying implementations can vary and run-time performance and code-size can take a big hit. The std::bad\_alloc exception is the only exception that is allocated on the stack since if heap memory fills up, the exception must be allocated somewhere. The only location left is the stack. The programmer can choose not to use exceptions at all or use compile-time options to disable exceptions. When it comes to dynamic memory allocations, fortunately C++ allows the ‘new’ and ‘delete’ operators to be overridden globally and at the class level. This feature allows programmers to define their own allocation routines that are deterministic and non-fragmenting. Examples of known allocation algorithms are Pools and LIFO(Last in First Out). Also, most of the standard template library containers take an additional allocator object which allows the programmer to pass in a custom allocator for the container to use. This makes the STL more able to be used. However, care must be taken because some routines in the STL can throw exceptions. The last construct I will discuss is TMP(Template Meta-Programming) which provides a mechanism whereby operations are computed at compile time incurring little additional code or run-time performance hit. An example of using TMP is found in Finite State Machine algorithms. Once such implementation is Boost::statechart from the Boost Library.[2]

In conclusion, used correctly for a particular embedded systems application, C++ can sometimes be faster than C with cleaner code. As C++ has become more mature, we have seen it expand more into the embedded system market since the 1998 standard.

By:

Kerry Murphy

**References**

[1] [**http://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming.php**](http://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming.php)