EC++ for Mission Critical Applications. A Big Mistake.

Marcel-Titus Marginean (mtm@mezonix.com)

April 4, 2013

Abstract

It is a common practice for certain industries to steer away from modern C++ for mission critical application; this is done out of a fear that "advanced techniques are unsafe". Many projects doing DO-178 mission critical software, using C++, are using a "watered down" version of the language, avoiding advanced techniques like templates and the standard library. Even worse, some groups promote Embedded C++ (EC++) as the only safe way to use C++ for mission critical applications.

In this paper, I show that relying on this "safe subset" of the language does not improve safety in any way, and, by promoting dangerous coding practices, these subsets produce deceptively safe-looking code. Using modern C++ we get important tools that, when used correctly, significantly improve the safeness and robustness of code. Also, I will explore how advanced techniques combined with good design can make modern C++ one of the safest languages available for developing mission critical applications. Furthermore, I will examine the roles of the software designer and team leader, and how corporate culture's influence over them may be a leading cause of unsafe code.

In conclusion, adopting a full featured C++ will improve the safety, and reliability, of code used for mission critical applications, and the practice of stripping down C++ has no discernible benefit.

Frequently, mission critical applications in C++ are developed without some of the most powerful tools in the standard language, including templates, multiple inheritance, the mutable specifier, name-spaces and more. The EC++ specifications eliminated them completely from the start. Led by some Japanese chip manufacturers, EC++ has been an effort to correct some of the perceived "shortcomings" in C++ related to embedded applications.

The goal has been to minimize the footprint of C++ generated code[EC++.1]. Unfortunately, instead of focusing on the places where a stripped down version of C++ could have been useful, mainly 8 or 16 bit micro-controllers, they discarded that market up-front by focusing on 32 bit CPU's. The group has been joined by a few software makers that wrote proprietary compilers, and other tools, for EC++ development; later they started to market EC++ as a "mission critical" software development method.

From the beginning EC++ received a healthy share of criticism from many C++ experts, including Bjarne Stroustrup (The creator of C++) who said "To the best of my knowledge EC++ is dead, and if it isn't it ought to be"[Stroustrup]. Despite all of this criticism, EC++ is still said by some groups to be a better choice than standard C++ for mission critical applications, even though it is possible for companies to disallow any feature of C++ they deem dangerous. Let's take a look at the implications of this approach, feature by feature.

1 Templates.

Contrary to some popular beliefs, templates do not introduce run-time overhead. Templates are resolved at compile time; they are "interpreted" by the C++ compiler and the code inserted is the result of template expansion[Todd.1]. They do not introduce more overhead, in terms of CPU usage and memory, than the same code written by an average programmer implementing the same algorithm for each of the data types involved. Using templates we can take advantage of people with advanced skills who have already written highly optimized generic algorithms in the form of template libraries and chances are that the generated code will be much better, and more efficient, than what the average programmer can write under typical project deadline pressure[Todd.1, Scott.1].

Now, lets look at the drawbacks of their absence, and how removal of the templates undermines efforts to produce safe code.

1.1 Copy/Paste programming

In the absence of a template library for often used containers and algorithms, most programmers will try to "templatize by hand," by copying a piece of code written by somebody else for a different data type and changing it to handle his own data type. There are quite a few problems with this programming style.

First, with "templatization by hand," it is easy to introduce errors by forgetting to replace a statement in a larger piece of code. A small example will highlight the problem. Let say that programmer A writes:

```
void saveInt(byte *&buff, byte *buffEnd, int val){
                if(buff+sizeof(int)<=buffEnd){
                        *(int *)buff=val;
                        buff+=sizeof(int);
                }
        }
```

Programmer B needs to load a double into the buffer, he copies/pastes A's code and adapts it for use with doubles. Unfortunately, this introduces an error:

```
void saveDouble(byte *&buff, byte *buffEnd, double val){
                if(buff+sizeof(int)<=buffEnd){ // easy spot to forget to repla
                        *(double *)buff=val;
                        buff+=sizeof(double);
```

```
            }
        }
```

On a 32 bit system, the code above checks to have at least 4 bytes of available space then writes 8 bytes at that address. A system crash is likely to happen, and that can bring down a plane. This error is hard to spot using a static analysis tool. By contrast, the code using templates:

```
template <typename T>
        void save(byte *&buff, byte *buffEnd, T val){
                if(buff+sizeof(T)<=buffEnd){
                        *(T *)buff=val;
                        buff+=sizeof(T);
                }
        }
```

The code above will work correctly for any primitive data types (and any data type that avoids indirection).

Second, if code is copied to multiple places by various team members, when the original author discovers and fixes a bug he eliminates it only from the original fragment of code. The bug remains active in all copies of the code. To make matters worse, if the original author is unaware that his code has been copied, he will not know to inform others of the update. By contrast, if the algorithm had been implemented as a generic template, once the author fixes the bug, it will be fixed for everyone.

Third, the simple fact that a generic style is used forces a programmer to use correct generic techniques. If a programmer writes an algorithm he believes will be used exclusively on 32 bit CPU's he may take shortcuts from correctness. For example, assuming the size of the data-type (integer) is the same as the size of pointers and taking "advantage" of this to write more "optimized" algorithms can cause problems when a junior programmer copies the code to customize it for a different data type, because he may not understand the assumption made by the original author. This would introduce a very hard to find bug in the new version of the code. Simply having to think generic will eliminate these issues from the beginning.

Finally, when under pressure beginners using the copy/paste/customize templatization technique can easily make mistakes, lulled into a false sense of security by this rote method of programming. By contrast, when a programmer writes a generic algorithm, or class, he is forced to pay more attention to details due to the inherent generic nature of templates. More often, the job of writing the generic core library of the project is taken by a senior member of the team with lot of experience, while copy/paste programming, due to its repetitive nature, is often left to less experienced programmers.

1.2 Opaque containers undermines type safety

In the absence of generic containers, programmers will be tempted to try to avoid copy/paste programming by writing general purpose containers based on

the old style of "casting pointers to ***void ****". For example a vector of ***void*** pointers can be used to store the pointers to the objects needed.

```
Plane p1,p2,p3;
        Vector lst;
        lst.push_back(&p1);
        lst.push_back(&p2);
        lst.push_back(&p3);
        ......
        for (int i=0; i<lst.size();i++){
                Plane *plane=(Plane *)lst[i];
                do_plane_work(plane);
        }
```

However, later on, if in the segment of code marked with ***...*** somebody mistakenly did the following:

```
Payload pd1;
        Datalink dl1;
        lst.push_back(&pd1);
        lst.push_back(&dl1);
```

the ***do_plane_work()*** function will erronious believe that the Datalink object is a plane, and a crash may not be avoidable in this situation. By contrast, using templates, if ***lst*** it is of the type ***std::vector<Plane ****> the two lines above will trigger a compile time error forcing the programmer to fix his mistake and avoiding a run-time crash.

1.3 Unfamiliarity with libraries introduces bugs, many eyes improve safety

In recent years a lot of the C++ community's effort has been focused on the advancement of generic algorithms and containers. The fruit of this effort is a well known, well documented, royalty free set of class libraries that most C++ programmers are familiar with. The C++ committee standardized many of them into the C++ Standard Library[C++.2]. It is a requirement that a C++ compiler comply with the standard, as a result, C++ programmers know very well how to use these libraries and avoid pitfalls.

Moreover, due to their widespread usage, the standard libraries are very well scrutinized, analyzed, debugged and corrected. If one is interested in mission critical code safety, he or she will expect to be able to use the safest and most documented libraries on the planet.

Without templates these libraries do not work. As a result most of the standard libraries are incompatible with EC++. As a workaround, some authors implemented libraries that exhibit an interface "as close as possible" to the standard library for some of the components like ***string*** and ***iostream*** classes. While this familiarity is a very good thing we must notice that most of these libraries have a very small audience and are proprietary. Inherently, they lack

the scrutiny the standard, or other open source, libraries enjoy. As a result, the probability of latent defects hiding in these replacement libraries is higher than in the standard or open source ones. Linux, the leading Open Source project, is known to have a lower bug count than the industry average[LINUX.1].

Since there is no way to implement generic containers, or algorithms, these libraries provide only a small fraction of equivalent functionality; the rest is up to the developers of each project to write by themselves. And, as Scott Meyers points out [Scott.1], it is always better and safer to use standard library algorithms rather than writing your own.

1.4 Metaprogramming and modern C++ language increase efficiency and safety

Contrary to popular belief, using C++ will not result in slower code and bloating. Sometimes, a good C++ compiler can generate code that is more efficient than the equivalent C program [Scott.2]. Also, using modern C++ techniques it is possible to greatly improve safety by implementing "unit aware" programs that are capable of doing dimensional analysis so that compilers will give you an error if you attempt, for example, to assign velocities to mass [C++.3, Brown.1, Scott.2]. This is a safety feature unmatched of any other programming language.

A relatively novel technique to improve both efficiency and safety of the code is "Language Oriented Programming", which means to create a programming language that is specific for the class of the problems you want to solve. That way the team is going to focus on the problem to solve, and solve it in a very high level language designed purposely for that the specific problem[Ward.1]. With modern C++ it is possible to use C++ as a domain specific language, using the tools described above, creating very fast pseudo "domain specific languages" for your problem.

While it is true that metaprogramming and advanced techniques are very hard to master, the majority of C++ users do not need to handle these kinds of advanced features. These features are going to be used by experts to write libraries, components, and frameworks, and be utilized by the rest of the team.

2 Multiple inheritance

In C++, multiple inheritance is required to provide a class the ability to exhibit distinct behavioral interfaces. Even in Java, a class belonging to a tree hierarchy can implement a second, third, etc..., interface in order to allow the class to expose a certain behavior not part of the main hierarchy.

The biggest problem with total elimination of multiple inheritance is that it impairs the ability of a software designer to implement a coherent design. Public inheritance, in C++, is used to express the concept of "***IS A***". Let's consider an unmanned flying boat. With multiple inheritance we can just write:

```
class OurDrone: public PlaneI, public BoatI, public RobotI{ };
```

```
...
logBoatStatus(ourDrone);
logPlaneStatus(ourDrone);
logRobotStatus(ourDrone);
```

This works because ourDrone ***IS*** **A** Plane, a Boat, and a Robot. The log functions will just make use of the appropriate interface to log the parameters specific to each entity. By contrast, in EC++, ourDrone can be either a Plane, a Boat, or a Robot. The ***logBoatStatus*** function has no way to accept a Plane, or a Robot, since if our drone inherits from ***BoatI*** it only inherits from ***BoatI***. This implies that we have to implement a complete set of functions (like: ***logBoat(), logPlane(), logRobot(), logBoatPlane(), logBoatRobot(), logPlaneRobot(), logBoatPlaneRobot()***) that will operate on each possible inheritance hierarchy of ***OurDrone***. Now consider the issues we have adding a second class, ***OurBetterDrone***, unrelated to OurDrone, or a third; this quickly gets bloated, inefficient, and hard to maintain. Frustrated programmers will engage in more copy and paste, and as the code becomes more complex there will be more room for errors, bugs, crashes and other potential disasters. While design patterns like Adapter can help deal with the complexity, while avoiding multiple inheritance, multiple inheritance is a more natural way to express the multiple "***IS A***" relationship[Luca.1]. It is also efficient [Jack.1].

Note, I am not necessarily advocating the use of heavyweight multiple inheritance. Inheriting from a single heavy class and "implementing" a few interface may be the best approach for most of the places where multiple inheritance is used. But in the absence of Java style interfaces, multiple inheritance cannot be eliminated from the language without seriously handicapping the ability to express coherent designs.

Multiple inheritance does not introduce any significant run-time overhead, generally there is a need to adjust the ***this*** pointer for each call, but usually this is achieved in a few machine instructions, practically insignificant for real-time performance[C++.1].

3 Namespace and using directive

Namespaces do not introduce any run-time overhead. All required lookups happen at compile-time so there is absolutely no need to worry about performance for real-time systems[C++.1]. Namespaces, however, are essential to allow programmers to deal with the complexity of large projects.

Today's targets for unmanned vehicles embedded projects are computers that are more powerful than the average server a decade ago. The complexity of the code for these systems is significant and the possibility of name clashes it is real. Namespaces exist to give developers an easy way to prevent these clashes from occuring.

4 static_cast, reinterpret_cast, const_cast

Now, lets talk about undermining safety! New casts were introduced in the C++ language for code safety reasons. The old C style cast concentrated way to much power in a single statement, it practically quieted the compiler, because there was no way for it to tell if the programer wanted to perform a particularly strange conversion, or had just made a humongous mistake. Programmers should always prefer the new, safer, C++ casts[Scott.3].

The new group of C++ cast operators attempt to explicitly express what the programmer intended to achieve with that cast and enable safety checks to catch potential errors. For example a ***static_cast*** will not allow you to remove the const away from a pointer, nor will a ***const_cast*** allow you to assign a base pointer to a pointer of a derived type.

More than that, the new style casts are very easy to detect by text searching, therefore safety checks, during code reviews, can quickly find out where any cast is used in order to be "blessed" by the reviewer. Since all casts are simply a way to side-step compiler safety checks it is very important to have them scrutinized carefully. The special syntax of new style cast operators draws attention to themselves, unlike the old style C cast.

By removing, or denying, the use of the new style casts operators, one removes a very important tool, designed to alert the programmer when a mistake has been made, and a method that allows the reviewer to find any segment of code that needs special attention. Promoting the removal of safety checks from a language is unwise, not to mention dangerous, in the mission critical environment for which 178C was designed.

These new cast operators do not introduce any run-time overhead[C++.1]. The only type of cast that introduces run-time overhead is ***dynamic_cast,*** if RTTI is enabled. Not having any penalty for their usage, the elimination of ***static_cast***, ***reinterpret_cast***, and ***const_cast*** was a senseless decision with an extremely detrimental effect on safety. This decision, by itself, should automatically disqualify EC++ for mission critical applications.

5 Mutable

Here is another example where safety is undermined by stripping down the language.

Before I start explaining the issues, I want first to make clear the design implications of the ***mutable*** keyword because I find it to be misunderstood very often, even by programmers with years of experience.

C++ introduced the concept of const correctness. A const object shall not be modified. Practically, it is a contract between the programmer and the compiler that guarantees that nobody shall be allowed to modify that object within the given const scope. For example, if we want to implement a logger function ***void log(const Object &o);*** any attempt inside the logger function to modify the object, ***o***, will be prevented by the compiler with an error message. This

contract it is often used by software designers to make clear the intent that inside a particular function an object is not supposed to be modified, and by this safety is improved. For example lets consider an object that connects the software with the aileron of a plane:

```
class Aileron{
                    volatile float currentAngle;
                    volatile float commandedAngle;
            public:
                    bool setCommandedAngle(float angle);
                    float getRealAngle() const;
                    float getCommandedAngle() const;
        };

...
        bool sendDataToOperator(const Aileron &left, const Aileron &right){
                MsgXYZ m;
                m.leftAileron=left.getRealAngle();
                m.rightAileron=right.getRealAngle();
                send(m);
        }
```

If some other programmer makes a mistake and tries to set the commanded angle inside the ***sendDataToOperator()*** function, the compiler will prevent this mistake with an error message. The reason is that the system architect envisioned that this function will never change the plane flight parameters but only provide existing status to the operator.

Now, lets consider that the plane does not bank as snappy as expected and the engineers want to log the evolution, in time, of the difference between the prescribed and actual aileron angle. For that the programmer will have to add a L***ogObject*** on the ***Aileron*** class, and write to it the values of those 2 angles whenever that data is read (get methods) by the algorithms. However, since get methods are constant and the new ***log*** member variable is written into (by a write method, not const) the programmer can not do that without violating the const contract. In regular C++, the programmer has the ability to use the ***mutable*** keyword to acknowledge that the logger is not a state of the aileron and therefore it is OK to modify it from within the scope of a ***const*** method. That is, ***mutable*** is used to discriminate between a state of an object that shall be maintained const during a const scope of that object and a non-state member variable that can be allowed to be modified during const scope without undermining the safety of the design.

In summary:

"mutable is usually used when an object might be logically constant, i.e., no outside observable behavior changes, but not bitwise const, i.e. some internal member might change state." - [C++ Programming (Wikibooks.org)]

In the EC++ the ***mutable*** keyword is not supported at all and some "C++ coding practice documents" declare it as "to be avoided, dangerous or unnec-

essary". As a result the programer is forced to change the get methods to not be const anymore. In exchange, this will trigger an error in ***sendDataToOperator()*** function, and the programmer will have to remove the const specifier from ailerons parameters. By doing this he violated the policy enforcement of the software architect, because this function is not supposed to change the state of the ***Aileron***. The architect, knowing that it is safe to play within ***sendDataToOperator()*** function, may assign a task to modify something there to the most junior team member, who decide to use the ***setCommandedAngle()*** in order to round the values to make them "look prettier". Guess how that will play out in flight?

The ***mutable*** keyword is an essential part of const correctness and being removed from the language introduces safety problems, however, the removal of the mutable keyword does provide the ability to easily find ROM-able objects[Bruce.1]. The reason ***mutable*** has been excluded from EC++ was because a const object having a mutable variable can not be stored into a Read-Only Flash or ROM. Imutability is a necessary but insufficient condition for storing an const object in ROM. Any time some "clever" programmer takes advantage of an old C style cast operator, to force a const ***this*** pointer to a non const one, will invalidate the ability to store the object in ROM. For example, let us consider a const array of initialization parameters with a const conversion method that provide convenient access to the data.

```
struct InitValue{
        float altitudeMeters;
        float getFeet() const;
};

const InitValue knownAltitudes[]={{1.0},{2.0},{3.0}};
```

When an "ingenious" programmer wants to count the accesses to the conversion method, and he is unaware of the architect's intent to ROM the initialization parameters, and wants to avoid removing the const signature, he decides to use the "clever trick" below:

```
struct InitValue{
        float altitudeMeters;
        float getFeet() const;
        int counter; // newly added counter
};

float InitValue::getFeet() const{
        ((InitValue *)this)->counter++; // BOOOM !!!
        return altitudeMeters*METERS_PER_FOOT;
}
```

If the ***knownAltitude***s is set up in Flash or ROM the safety of the whole program is totally compromised by a potential SEG_FAULT or equivalent. The problem is made worse by the lack of ***const_cast<>*** operator. Without

const_ cast<> the ability to search for the unsafe use of ROM components is diminished, and such a fragment of code may avoid detection until it brings down the system. By contrast, if the ***mutable*** modifier were available (or at worst, ***const_ cast<>***), a simple grep for keywords would point immediately at the ***InitValue*** struct as a non ROM-able component, bringing the issue to the attention of architect or chief designer. At least this would expose the "clever trick" as a bomb waiting to explode. It is always better to educate your programmers about project specific restrictions instead of providing them with a stripped down language in the hope that your programmers are not "smart enough" to find workarounds or hacks. Eventually, they will find them, and the results will not be something to look forward to.

6 Exceptions and RTTI

Unlike the rest, exceptions and RTTI do introduce real run-time overhead[C++.1]. However, all major compilers have flags like ***–no-exceptions*** or ***–no-rtti*** that can be used to disable them as needed. The decision to disable these features should be left to the software architect, or other technical project leader, rather than being imposed by the compiler vendor, or some coding standards document written years ago outside of a given project. The technical leader knows very well the true details of each project and can independently decide if he can afford the memory/CPU costs these extra safety features consume. If the target is a microcontroller with under 64kB of RAM it is likely that the overhead of exceptions and RTTI is not a cost that can be incurred, and will be eliminated. I do this on all of my microcontroller projects. However, if the target system is a mission critical computer with 512MB RAM, or more, the ability to catch a write to a NULL buffer, and avoid a system crash, may be well worth the tens or hundreds of kilobytes "wasted" with exceptions.

Stack unwinding is usually a non-deterministic sequence of code, but this can be eliminated by conservative using of exceptions.

6.1 About exception usage

In C++ there are 2 schools of thought about the usage of exceptions.

The camp coming to C++ from plain old C advocate the usage of exceptions as a "solution of last resort", use them if everything else fails. They advocate that most of error checking be done within the regular flow of the program, by returning error codes. This implies that each function checks the preconditions and guarantee the post conditions. If the preconditions are not right, return error code instead of doing any calculation. An exception should only be thrown in case of a catastrophic type of error where the code is way overwhelmed and incapable of dealing with it in any meaningful way within the work-flow. The exception will, most of the time, be caught at the thread level (or a few calls away at most) where there is code to do "damage control", "clean up the mess" and restart the functionality.

On the other side, the camp coming to C++ from Java or Python advocate extra liberal use of exceptions, using them whenever something is not absolutely perfect. The main flow assumes that everything goes one way and if the inputs are not as expected, throw an exception. The exception is caught most of the time at the same function level where it is thrown (or at most one or two call-levels below), and the exception may even be used to change the flow of control as an **if** replacement. It is not uncommon to see Python programmers use the following pattern to count the occurrences of an element into a list:

```
counters={}
for e in lst:
        try:
                counters[e]=counters[e]+1
        except:
                counters[e]=1;
```

This pattern assumes that it is "normal" for a counter to already exist in the map and if it is not there (aka it is first time when we see an e) that is an "exceptional" event "worthy of" raising an exception.

6.2 Exception overhead

While there are still active public debates about which style is "better," when efficiency is considered the balance goes, without any doubt, towards the conservative camp. While C++ has been designed to not add any overhead to the C language, while doing the same work, this is not true when using exceptions. The fact that exceptions break, what is commonly called, "Stroustrups promise: With C++ you don't pay for what you don't use", may be the reason most C++ compilers have flags to disable exceptions.

However, the overhead introduced during normal function calls is relatively small and a constant for each call level[C++.1]. Therefore even in a real-time system it is possible to account for it in a deterministic way. That is, overhead added by enabling exceptions is deterministic as long no exceptions are thrown.

However, when you use them, (i.e. throw an exception), that is another story. Stack unwinding may take a lot more CPU than setting it up and it is regarded as non-deterministic because it usually involves the same run-time checking as RTTI and it is also compiler specific [C++.1]. While it may be possible to measure a given program and say with a higher level of confidence after testing that "the stack unwinding on this program will not take longer than XX milliseconds" the fact that it is a big unknown prompts us to consider it non-deterministic and therefore not for normal real time operations.

6.3 How to use exceptions in Real-Time systems

Many Real-Time professionals will cringe at what I am going to say next, but bear with me for a while. It is time to talk about this "taboo" subject once for all.

Yes, I am going to make the controversial claim that there may be places to use exceptions in Real-Time systems. The catch: throw it only when otherwise the program will terminate with an abnormal termination code; that is, no fragment of code is perfect, once in a while something absolutely unexpected happens and the best written piece of code may fail. Whether a memory block had just burned out, or a gamma-ray-burst changed the value of a bistable circuit on the microprocessor, the hard-drive just got demagnetized or yes, that perfect piece of code you wrote 5 years ago had a hidden bug in it that manifested only with an unlikely set of parameters that is impossible to happen but ... just happened. If in such a condition the program crashes, it is the job of the OS to restart it, and then maybe throwing an exception can help the real-time behaviour. And here is why.

Consider this, the ***catch(...){}*** statement from ***main*** gets the control. In that statement you launch a procedure that dumps into a shared memory segment the state of the system. That ***DoomsDay()*** function has been written with the intent that it gets called exclusively when a catastrophe happens and therefore any states it dumps out are checked, double-checked and cross-referenced for consistency, and only states that are 100% valid are written. When the program is restarted, after the crash, it is able to restore most of the state in milliseconds and then wait to gather only the data that has been compromised. This may be way faster than waiting to gather all the data from scratch. The program may be able to start faster than otherwise possible, and this may be the difference between the plane smashing into the mountain or launching an avoidance maneuver in time.

An alternative use is at the thread level. Without exceptions an error in a thread will force a program to terminate and restart. If the designers figure out that it is possible to intercept a thread level fatal error, clean-up the state of that particular thread and restart it, it is going to be more efficient and closer to real-time requirements than program termination and restart.

That is, properly used, even the non-deterministic "anti-real-time" exceptions can improve real-time performance while improving safeness. The catch here is "properly used"; and this implies that real technical expertise must be involved in managing the project.

6.4 Conclusion about exceptions

Let's summarize: the overhead introduced by exceptions, when you don't throw any, is a small percent of every function call. You can deterministically account for that and consider it part of the cost of any function call and return. Therefore, if you don't throw any exceptions you don't pay any extra penalty (beside what we already accounted for) nor do you have any non-deterministic behavior for having code that uses exceptions. If you throw an exception when the system is about to crash the extra penalty is small compared with the time it takes for the operating system to terminate, clean up, and restart the program. If there is any advantage in throwing an exception, then you may want to consider using them. If there is no advantage you can always compile your

standard C++ program program with ***–no-exceptions*** or the equivalent flag and you will not pay a penalty anymore.

6.5 RTTI

Like exceptions, there is a performance impact from using RTTI. It mainly comes in the form of extra checking done when using a ***dynamic_ cast<>()*** (which involves run-time look-ups)[C++.1]. Like exceptions, the overhead introduced is compiler specific and therefore can not exactly be accounted for unless you do your own exhaustive measurements on your project. This is the reason RTTI it is also considered non-deterministic and not to be used for real-time systems.

However, exactly like exceptions, it can be compiled out of the project with something like ***-no-rtti*** or similar, so if you don't need it you don't have to pay for it.

Also, there is still public controversy about RTTI. Some C++ experts actually recommend to not rely on RTTI for your designs regardless if you do real-time or not[Scott.4].

Like exceptions, you can use any standard C++ compiler with ***-no-rtti*** or equivalent compilation flag if you want to remove any trace of it from your programs.

7 Considerations about the importance of design and good designers

7.1 Focus effect

Unlike multicore microprocessors, our brains were not really designed for multitasking. Various studies found that when one multitasks, they are more prone to commit errors, fail to perform as expected, and experience a decrease in productivity[Multi.1, Multi.2]. Due to a higher rate of errors while multitasking, it is preferred for safety critical systems to provide a development environment that reduces the attention switching a programmer must endure while solving a problem. Let us look at a simple example:

```
int problemsDetected=0;
for (int i=0; i < noOfPlanes; i++){
        if(plane[i] != NULL){
                if(plane[i]->state == BAD_STATE ||
                   plane[i]->state == WORST_STATE)
                        problemsDetected++;
                }
}
if(problemsDetected >0){
        do something here
};
```

The programmer writing this fragment of code has to constantly switch his attention from "what states are important to handle in this fragment" to implementation details like "Pointer dereference, Null checking, array access, index increment, and correct range for the loop". If the company has a very strict coding style, in the hope that this will somehow improve the safety, other distractions arise (such as "are these braces right"). The constant switch of attention from one aspect to another is a distraction and it has the potential to increase the error rate in code. Because of this, some large projects go through great lengths to design and build "problem oriented languages" for their developers to express the problem [Ward.1]. Designing a problem specific language has the advantage of separating the "problem solvers" from "low level details" and allows each group to focus on their task better. In such an imaginary language the developers can express the problem above, something like:

```
if any(planes).state in (BAD_STATE, WORST_STATE) then
        do something here
endif
```

However, the lack of familiarity with the new language may be a problem. Since the language is designed for one project only, the contractors will not be very enthusiastic to learn it in great detail, since they know that once the project finishes they can not make use of it anymore. Therefore, it makes sense to create the pseudo-language as close as possible to a popular general purpose, well known language.

Another aspect is that writing a compiler for a new language is a highly specialized task, and it is not easy to find experts to do it. And here is where the true power of the modern C++ cames into the picture.

C++ with all the features discussed above like templates, multiple inheritance, exceptions etc. is a perfect tool for creating "pseudo problem oriented languages" that look and behave exactly like C++ because they are C++. That is, we can use the C++ language to create a high level framework for solving the problem at hand, totally separating the scope of framework building (involving low level C++ details and advanced techniques) from the high level business logic. Now the high level programmer can focus exclusively on the business logic they have to solve. For example, designing an appropriate StateChecker unary predicate, the problem above can be solved in pure C++ as easy as:

```
if( count_if(planes.begin(), planes.end(), StateChecker(BAD_STATE, WORST_STAT
        do something here
}
```

This way, using the standard C++ language, the framework allows the programmer to focus exclusively on the problem to solve without low level distractions. This is pure C++, a general purpose language that they already know, and have many incentives to keep learning - the best of both worlds.

It is exactly because a system architect can create a "pseudo problem specific language" out of C++ that has the potential to makes C++ one of the safest languages that can be employed for mission critical applications. That

is, with clever template meta-programming, good design, good tools, and a well informed team, C++ can become one of the safest language for mission critical applications. Can anyone argue that the following for-loop in the Ada language (modified from one in Wikipedia):

```
Array_Loop :
        for I in X'Range loop
                Do_Something (X(I))
    end loop Array_Loop;
```

it is any safer or easier to understand than the one in modern C++ 11?

```
for (const Type value : X)
        doSomething(value);
```

There is no mistake. I purposely wrote: ***const Type value***, not ***Type value*** despite the fact that it is an iteration by value not by reference. What extra safety check over the Ada example above we can enforce here by using ***const Type value*** as opposed to more common ***Type value***?

By specifying the value to be const, the caller can enforce the fact that the ***doSomething()*** function must not alter it parameters in any way (think an Ada ***In*** not an ***In Out***). If later another programmer wants to modify that function to have an ***In Out*** parameter the ***const*** specifier will alert him that certain assumptions have been made about that function and it will bring the issue to the attention of the designer. This is the power C++ puts at the fingertips of the designer to enforce safety.

7.2 Contracts are best enforced with C++

Unlike Eiffel, C++ lacks built in constructs for programming by contract. This is the place where coding standards are useful. Unfortunately, for each coding standard manual that talked about checking of preconditions, I have seen quite a few that were silent in this respect while putting emphasis on trivial stuff like where to put braces, or to prefer underscore_identifiers instead of camelCaseing. It is the role of team leader and software designers to properly enforce this programming style, both by specifying in their designs the pre-conditions and post-conditions to be checked, by example, and by code reviews.

However, beside making the contracts explicit in the designs he gives to programmers, a good software designer can do much more in C++. Consider the following Eiffel example (modified from one in Wikipedia):

```
set_hour (a_hour: INTEGER)
            -- Set 'hour' to 'a_hour'
require
        valid_argument: a_hour >= 0 and a_hour <= 23
do
    hour := a_hour
end
```

The naive C++ implementation of Eiffel contract may be to just do nothing if the parameter is invalid or raise an assertion:

```
// do nothing option
void setHour(int h){
        if(h<0 || h>23) return;
        hour=h;
}

// assertion option
void setHour(int h){
        assert(h >=0 && h <=23);
        hour=h;
}
```

The problem with both of these approaches (Eiffel and C++ above) is that if the parameter passed is junk we still have to either throw a run-time assertion or simply ignore the operation like it never happened. Ignoring introduces possible inconsistencies since the caller expects something to change or a value to be stored in order to be accessible later. The main problem here is that the pre-condition failure is handled at the function level, not at the place of call. The creator of the function does not know the intent of the caller, and what a safe fall back strategy is at the point when the function was called.

A good C++ designer however may write inside the framework he develops for the project an Hour class:

```
class Hour{
public:
        enum Hours{INVALID=-1, H0=0, H1, H2,.... ,H23};
        Hour(Hours h);
        Hour(int value, Hours fallBackValue);
        bool valid() const;
....
};

void setHour(Hour h){
        if(! h.valid()) return;
        hour=h;
}

int someValue=45;
setHour(someValue); // Compiler ERROR !
// programmer is forced by the compiler to write instead:
setHour(Hour(someValue,Hour::H0));
```

With the two constructors above, the programmer who calls the function can pass either a valid H0...H23 value or a regular integers but the designer has forced the programmer to provid a valid fallback value just in case the

integer is wrong. The caller of the function is better equipped to select the fallback startegy at the point of call, the function designer just need to make sure that the programmer writting the call is going to make that choice. This C++ function it is safer for mission critical applications than the original Eiffel approach.

7.3 Safety by unit testing

As I explained before, a cleaner design, one that separates the implementation details and business logic, has the potential to highly improve the correctness of code by taking away distractions. The separation of business logic and framework design allows the framework components to be tested and validated independently. Having a general purpose component makes it easier, and thus more justifiable and more cost efficient, to go through full unit testing procedures. By this, the method of iteration will be stressed with fringe conditions and unlikely cases along with a full set of automated tests. When the programmer uses that component in his business logic, the component has been tested and validated. However, if the programmer writes his own check, using a loop, unit testing may be impossible, because the code can not be taken out of context and isolated in such a way that it can be unit tested independently.

Separating component implementation from business logic may allow for some components that implement known computer science algorithms to be formally proven correct[Dijkstra.1]. This goes a long way to prove code is safe, by contrast once you insert a copy/pasted algorithm into business logic the chances that your algorithm may be formally proven are very poor.

A good designer can optimize the set of functionality exposed by the framework with the set of dependencies injected into a module employing design patterns, such as "Dependency Inversion"[INVDEP]. Having all the modules relying exclusively on a standard set of functions/objects in a framework and having all the rest injected as described above, enables all these modules to be independently testable. Instead of having to wait for the whole system to be assembled for the tests to begin, now individual components can be tested, and validated, in isolation allowing the tester to focus on that particular module. The test harness can simulate a whole range of fringe values that may appear extremely rare in real life and may not be achievable in the limited setting of the lab when the end system is assembled. That way, defects can be detected early and corrected in parallel with other modules being developed. It is always better to have your component crash in the lab's test harness rather than in the field.

However, to achieve this level of safety with C++, the teams must have skilled technical leaders with a deep knowledge of the best design patterns, mastery of the C++ advanced techniques, talent for analyzing architectures and making coherent designs, along with many years of experience in customizing the language to be fit for a particular purpose and scope.

7.4 Experience, Enthusiasm and Talent matter.

The designer's task of decomposing a system into subsystems that "makes sense" is not an exact science. Figuring out what "makes sense" is an art. A designer can make, or break, a project. This is a task that we do not know how to automate, there aren't rules that "everybody can follow" to do a good job. That is, designing good software architectures is a job that requires creativity, vision, talent and expertise. It is as much an art as it is a science or discipline.

Experience plays a major role in decomposing a system. An experienced designer is able to understand pitfalls junior programmers are prone to fall into, and design the framework architecture to disallow it.

It is also a talent to be able to imagine ways in which the framework can handle run-time errors, recover the state and be able to cope with unforeseen situations. This is creativity at work; creativity can not be generated "on a conveyor belt" by anyone reading "Sam teach yourself software engineering in 21 days".

Some aspects of advanced engineering are still an art and those who disregard this aspect do it at their own peril. Elegance matters. Professor Dijkstra said: "Elegance is not a dispensable luxury but a quality that decides between success and failure." This is a fundamental truth in software engineering.

An elegant design nicely guides the user (application programmer) into accomplishing his task without too much effort and allows him to focus on the problem. The result of ***D=A+B*C;*** is identical with ***D=MatAdd(A, MatMul(B,C))***;, but the first is more natural to write and intuitive to the user, thus it is more elegant. For the second, the programmer may have to go through library documentation to find the correct names of functions after he gets a compiler error while writing ***D=MatrixAdd(A, MatrixMul(B,C));.*** Elegance matters. The first one is natural, intuitive and beautiful, while the second one just delivered a dose of frustration. It is the job of the library designer to understand what is natural, intuitive and helpfull to users. Again, there is talent involved here, if awesome music is the art of creating beauty out of sounds, and awesome painting is the art of creating beauty out of colors, then awesome engineering is the art of creating beauty out of function.

Andrew Koenig in Dr. Dobbs [Koenig.1] points out that when floating points are involved even the well known and mathematically proved addition of 3 numbers can render surprises. This kind of problem highlights why there is always a need for an organization to hire, and strive to keep, highly talented individuals in house. When such a peculiar problem arises, during final tests or right before a deadline, in order to find it and come up with a solution in time, you need either an engineer with experience, who happens to have already encounter similar issues, or is passionate about the trade and enjoys constantly reading article in professional journals and encountering "interesting problems" in professional discussion groups.

What is so pleasant about good engineers is their spirit of "Maker/Creator", what ancient Romans called "Homo Faber". The enthusiasm of dreaming to see their effort being productive, rise to the stars, or helping change our world for

the better. However, this enthusiasm can be worn out by long periods of routine work without an end in sight due to the lack of work saving tools, or the absence of a modular design that allows them to see and validate their progress. Losing focus is a big step toward unsafe code.

7.5 Is corporate culture to blame?

As highlighted in all the previous arguments, a common denominator seems to arise. In order to use C++ to create the safest mission critical applications you need ***true technical leaders***. You need people willing to sacrifice their leisure time to advance and sharpen their technical skills. Yet, they are an increasingly rare appearance at decision levels. Unfortunately, taking the time to keep knowledge current hinders the ability of individuals to engage in corporate politics. In an organization where conformity is appreciated more than outstanding talents, the talents will always be kept at the bottom[Corp.1]. If the corporate culture forces people to either stop doing engineering or leave the company in order to advance in their career, chances are the company will be left with unskilled people doing the tasks that require skilled talents.

Without true technical leadership in a team, chances are somebody will attempt to use a tool they do not understand and it will be used the wrong way. The smart answer would be to hire technical leaders. The wrong answer will be to forbid any advanced techniques in the hope of improving safety. The later is a false and dangerous hope, as highlighted in this article.

If the code safety problem resides mainly in the corporate culture, then it is the corporate culture that needs bug-fixing, not the programming language. The promotion of "diluted C++ dialects" for mission critical projects, specifically the usage of EC++, is not going to do any good.

Software engineering is not an assembly line factory[Corp.3]. The strive for a "repeatable process" does not necessarily mean that the company must always generate mediocre results just to avoid "dependence on one superstar". This is the wrong understanding of what a software engineering process means[Corp.2] and is likely to end up delivering unsafe products, and exceed deadline and budget.

8 Conclusion

The C++ language is extremely powerful, and with great power comes great responsibility. The responsibility comes in the form of the advanced knowledge one must acquire in order to become an expert in the language. It also comes in the form that every safety conscious team must be lead by experts, and true professionals must have a say in decisions. A highly skilled designer can use advanced C++ techniques to create a "problem oriented pseudo-language" insulating the rest of the team from advanced language details while providing them with safe to use, and efficient, tools to solve the problem.

Junior members need to be mentored and helped to master the very powerful but equaly complicated language. With a language as powerful as C++ there are ways to do harm if you don't understand it properly and have no guidance to avoid pitfalls. You need leading experts able to spot the mistakes of the beginners and spread the wisdom of how to do it the right, and safe, way. You need good designers able to create frameworks to minimize mistakes and to make the system tolerant of them.

Using a "dumbed-down" language is not going to help create safer programs. Quite the opposite, it will force people to: write many more lines of code increassing the bug-count; lose focus on the problem in order to handle implementation details; use unsafe coding practices to workaround the artificial limitations; and deprive programers of the tools and frameworks that improve safety. That is, EC++, and similar efforts, just do more harm than good. They make our programs unsafe, buggier and more likely to crash. This dangerous and counterproductive tactic must be abandoned by the safety conscious industry. The mission critical industry needs to start using modern C++ in teams lead by technical leaders.

References

[EC++.1] Rationale for the Embedded C++ specification: http://www.caravan.net/ec2plus/rationale.html

[Stroustrup] "To the best of my knowledge EC++ is dead (2004), and if it isn't it ought to be" http://www.stroustrup.com/bs_faq.html

[EC++.2] EC++ Questions and Answers: http://www.caravan.net/ec2plus/question.html

[C++.1] Technical Report on C++ Performance: http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf

[Todd.1] Template Metaprograms, Todd Veldhutzen. C++ Report 1995

[Scott.1] STL Algorithms vs. Hand-Written Loops, Scott Meyers, October 2001

[C++.2] Standard C++ Library. http://www.cplusplus.com/reference/

[LINUX.1] Linux, less bugs. http://www.wired.com/software/-coolapps/news/2004/12/66022

[Scott.2] Scott Meyers C++ in Embedded Applications (video): http://www.informit.com/podcasts/episode.aspx?e=2214a9d9-aacc

[C++.3] User-defined Literals, C++ comitee. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2378.pdf

[Ward.1] Language Oriented Programming, M. P. Ward. January 17, 2003

[Jack.1] Multiple Inheritance Considered Useful,Jack Reeves. Dr.Dobbs February 2006

[Scott.3] More Efective C++, Scott Meyers

[Brown.1] Introduction to the SI Library of Unit-Based Computation,Walter E. Brown @Fermilab

[Luca.1] A Semantics of Multiple Inheritance, Luca Cardelli 1988

[Bruce.1] Thinking in C++, 2nd ed, Bruce Eckel 2000

[Scott.4] Efective C++, Scott Meyers

[C++ Programming (Wikibooks.org)] C++ Programming http://upload.wikimedia.org/wikipedia/commons/e/e9/CPlusPlusProgramming.pdf

[Multi.1] Impact of Simultaneous Collaborative Multitasking on Communication Performance and Experience, Xu, LingBei 2008

[Multi.2] The multitasking clinician: Decision-making and cognitive demand during and after team handoffs in emergency care, 2006 Archana Laxmisana, Forogh Hakimzadaa, Osman R. Sayanb, Robert A. Greenb, Jiajie Zhangc, Vimla L. Patel

[Koenig.1] Are You Sure That Your Program Works? Andrew Koenig September 2012 Dr.Dobbs

[CMMI.1] http://www.sei.cmu.edu/cmmi/

[INVDEP] http://www.objectmentor.com/resources/articles/dip.pdf

[Corp.1] http://gmwilliams.hubpages.com/hub/People-Fear-Becoming-Authentic-and-Independent-Thinking-Individuals-It-Is-Easier-to-Conform

[Corp.2] High-Performance Teams: Critical for your Software Projects, Leonardo Mattiazzi. http://sandhill.com/article/high-performance-teams-critical-for-you

[Corp.3] The Fallacy of Software Factories and the Importance of Talent,Glenn Gruber. http://connect.phocuswright.com/2010/05/the-fallacy-of-software-f

[Dijkstra.1] http://www.cs.auckland.ac.nz/~jmor159/PLDS210/dij-proof.html

Appreciation to David Derenberger and George Bachman for reviewing this article.