C++ in Safety Critical Systems

David W. Binkley[1)]

National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899

[1)] Email: binkley@sst.ncsl.nist.gov. Also at Loyola College in Maryland

Abstract

The safety and reliability of software is influenced by the choice of implementation language and the choice of programming idioms. C++ is gaining popularity as the implementation language of choice for large software projects because of its promise to reduce the complexity and cost of their construction. But is C++ an appropriate choice for such projects? An assessment of how well C++ fits into recent software guidelines for safety critical systems is presented along with a collection of techniques and idioms for the construction of safer C++ code.

1 INTRODUCTION

The last few years have seen a dramatic increase in the use of software, in particular, the use of embedded software. As this software is given greater responsibility for flying airplanes, driving cars, and operating power plants, safety concerns increase. The growing need for high-integrity software and the concern over the use of C, the most common implementation language [Hatton, 1994], has lead to research projects such as the safe C compiler [Austin et al., 1994] and Les Hatton's recent book *Safer C* [Hatton, 1994].

The growing popularity of object-oriented programming and the explosive growth of the use of C++, has created a need to consider the implication of using C++ in high-integrity and safety-critical systems. This paper addresses creating such software in two parts. First, it applies the guidelines from a recent assessment of software languages for use in safety-critical systems [Hecht et al., 1996] to the C++ programming language. Second, it considers a series of techniques and examples for creating safer C++ programs.

2 GUIDELINES

This section considers guidelines for creating safe reliable software in the context of the C++ programming language. These guidelines were developed by SoHaR Inc. to access the characteristics of high level languages used in safety and high integrity systems [Hecht et al., 1996]. They "address generic characteristics and features that may affect safety" [Hecht et al., 1996]. The SoHaR report considers these guidelines in the context of several programming languages (including C++); however, SoHaR's discussion treats C and C++ together, and was written with a strong C bias. There is a need for a true C++ assessment.

Roughly one third of the SoHaR guidelines deal with concerns not directly related to any programming language, for example, the control of memory paging. These guidelines are not discussed herein. The remainder are applicable to specific C++ language features (*e.g.*, ensuring overloaded operators maintain the equivalence of "`p->m`", "`(*p).m`", and "`p[0].m`"). This paper focuses on guidelines dealing with reliability and safety. A complete discussion of all guidelines related to C++ appears in [Binkley, 1996].

The SoHaR report divides guidelines into four categories *Reliability*, *Robustness*, *Traceability*, and *Maintainability*. Guidelines numbered 1.x.x, 2.x.x, 3.x.x, 4.x.x come from each category respectively. The following terms and notation are used in the discussion below.

Attribute. Attributes hold the state information of an object from a class.

Declaration. A declaration declares the type of an identifier, but does not allocate storage for it (*e.g.*, `void update(Stack s);` or `class Stack {...};`).

The declaration of a function is often called its *prototype*.

Definition. A definition declares the type of an identifier and allocates storage (*e.g.*, `Stack s1, s2;` or `int increment(int x) { return x+1; }`). The definition of a function, which contains the body of the function, allocates storage for the body of the function.

`TeleType` **font.** The `teletype` font is used for C++ code that appear in this paper. In addition, C++ keywords and code fragments that might otherwise be overlooked are underlined.

Before discussing the guidelines, three things are worth noting. First the guideline numbers have been kept consistent with those used by SoHaR, so they are not contiguous. Second, the number for each guideline is prefixed by a "**G**" to distinguish references to guidelines from references to sections of the text. For example G3.1, refers to Guideline 3.1, while 2.2 refers to Section 2.2. Finally, it is worth noting that the length of "G4.1.6: Minimizing Obscure and Subtle Programming Constructs" is an indication of the dangers of using C++.

G1.1.1 Minimizing Dynamic Memory Allocation. A clear understanding of resource acquisition (allocation and release) is important.

Avoid the use of C's <u>`malloc`</u> and <u>`free`</u>. Instead use (sparingly) C++'s <u>`new`</u> and <u>`delete`</u> operators. Overloading these operators changed dramatically in the new C++ standard [ANSI, 1995]; overloading them is dangerous until the definitions and compiler implementations stabilize.

The semantics of class constructors and destructors can be used to ensure the proper release of acquired resources. Thus, to avoid memory leaks all classes should include a destructor that releases any memory allocated by the class' constructor or member functions. To ensure that destructors for attributes are called, class constructors should be declared

```
foo::foo() : class_attribute(...)  { sl }
```

and not

```
foo::foo() {class_attribute = ...; sl }
```

because failure in statement list `sl` causes a call to the destructor for `class_attribute` in the first definition, but not in the second; thus, in the second there is a potential memory leak.

Also, always set a "`new_handler`" using the built-in function `set_new_handler`. The default `new_handler` terminates the program when it cannot satisfy a memory request. Program termination at a critical time may be disastrous.

Proper resource acquisition extends to other resources. For example, a class representing an open file, can ensure the file is closed when it is no longer accessible, by having the destructor close the file. C++ semantics guarantee that once the constructor for a variable completes, any control transfer out of the variable's scope will cause a call to its destructor.

G1.2.3 Initialization of Variables before Use. Initialize all variables at their point of definition. An array class makes this possible for arrays. In the absence of such a class, initialize an array in a loop immediately after its definition. Finally, class constructors and `operator=` should initialize *all* class attributes. For constructors, "call" attribute's constructors before the body of the constructor (as in G1.1.1). It is important to note that C++ defines the order of these calls as the order in which the attributes are declared in the class and **not** the order they appear in the constructor definition.

G1.2.4 Single Entry and Exit Points in Subprograms. Avoid the use of the `catch` and `throw` exception handling mechanism. It provides (restricted) interprocedural control transfers, which violate single exit. (Implementation of these features is patchy from compiler to compiler, which opens correctness concerns). The use of multiple `return`s should be avoided in long functions (functions with 100 or more lines of code); in small functions, the opposite is often true.

G1.2.5 Minimizing Interface Ambiguities. Avoid the use of `varargs` and `extern "C"` as neither the type nor number of parameters can be verified by the compiler. Have all functions check their parameters for range correctness. Alternatively make parameters with range restrictions a separate class that includes the range check. This requires a single copy of the range check (in the class constructor) and helps avoid forgotten checks. Finally, the use of multiple inheritance should be tightly controlled if not eliminated. Confusion over which member functions are included in the deriving class and the use of virtual base classes should be considered before multiple inheritance is used.

G1.2.6 Use of Data Typing. Since the fundamental types `int`, `float`, and `char` are not true classes, their use is restricted in certain contexts (operator overloading for example). Creating classes for the fundamental types provides access control and also increases uniformity. For example, use classes `SafeInt` (see Section 3.1) and `SafeFloat` (see Section 3.2) in place of `int` and `float`. If necessary for execution speed, these classes can be removed, but only if profile data indicates such a substitution is warranted. For example, `SafeInt` can be replaced with `int` using "`typedef int SafeInt;`." Finally, C++ has a much stronger type system than C; however, it still includes the ability to type-cast pointers. This usage should be avoided.

G1.2.7 Precision and Accuracy. Test the floating point data type with a test suite to assure compliance with standards (*e.g.*, the floating point standard ANSI/IEEE Std 754-1985). For integer types, C++ guarantees only that `sizeof(short)` <= `sizeof(int)` <= `sizeof(long)`; no guarantee that a particular integral type has a particular number of bits is made. Use similar testing to ensure the size of `short`, `int`, and `long` are sufficient. Such testing should be made part of program startup, to protect against improper porting of the software. If a separate compliance test program is used, it must be run on both the development system and the target system when dynamic libraries are used (see G1.2.12). (For additional safety and control, use classes such as `SafeInt`, described in Section 3.1).

G1.2.8 Use of Parentheses rather than Default Precedence. In C++, this becomes a particular problem when operators are overloaded with definitions that do not correspond to the normal definition (see G1.2.13). Thus, it is not a problem when the complex number class defines `operator+` and `operator*` as *add* and *multiply* because they have the expected precedence in C++. However, overloading becomes a problem when a real number class defines `operator^` (bitwise exclusive or) as *exponentiation* because it has an unexpected precedence: the expression 6.23^2.0+3.0 is 6.23^(2.0+3.0) and not the desired (6.23^2.0) + 3.0. Always use parentheses; do not rely on precedence, especially in the presence of operator overloading.

G1.2.9 Separating Assignment from Evaluation. Functions should be divided into *evaluation functions*, which compute results based on their parameters without modifying them, and *update functions*, which may modify their parameters. Evaluation functions should have all constant (`const`) parameters; thus, preventing parameter modification. Update functions should either update the receiving object or produce a new object. Those updating the receiving object should return `void`; those producing a new object should return a new value of the same class (or type) and leave their parameters unchanged.

G1.2.12 Minimizing Dynamic Binding. Dynamic link libraries are linked in with an executable at load time rather than at link time. This produces smaller executables, but uses the library code on the target system not that on the development system. Thus, the developer and tester may not uncover problems that arise when a dynamic link library on a different machine is used. At a minimum, testing of the libraries on both systems should be performed (see G3.1.1 below). The use of dynamic link libraries is not recommended in safety critical code.

G1.2.13 Control of Operator Overloading. Overloading an operator to other than its obvious meaning may lead to errors in the code. For example, the operator `Complex::operator+` that adds two complex numbers has the obvious meaning, while the operator `List::operator+` that adds an element to a list does not. It is reasonable, but not obvious. Another reasonable meaning would be list concatenation.

G3.1.1 Controlled Use of Built-in Functions. Use testing with a standard test suite to verify compliance of built-in functions with standard and expected behavior. An example of a built-in operator with different behavior on different systems when applied to negative numbers is *integer remainder* (`%` in C++).

G4.1.6 Minimizing Obscure or Subtle Programming Constructs. The following "laundry list" addresses common C++ error prone idioms.

- The C++ reference type should be avoided because it allows implicit modification of referenced variables. Explicit modification through pointers is preferred since it avoids hidden implicit changes. This is especially true when a called function modifies an actual parameter through a reference formal parameter. The exception to this is parameters passed by reference to avoid the cost of copying a large data objects. Such parameters should be passed as constant references (*e.g.*, `f(const large_type &x)`). Declaring a parameter "`const &`" preserves the semantics of values parameter passing, without the cost of copying large objects.
- Avoid using default parameters to combine functions. For example, do not use the single function `lookup(char *name, code=-1)`, where the value of `code` determines whether `lookup` should fail or add `name` if it is not found. Such combinations violate G1.2.9.
- Avoid complex expressions in a condition. For example, the expression "`if (i & mask == 0)`" is evaluated as "`if (i & (mask == 0))`" and not as "`if ((i & mask) == 0)`." Replace it with "`long masked_i = i & mask; if (masked_i == 0)`."
- Avoid using `operator++` except for <u>`v++`</u> where <u>`v`</u> is a simple variable or <u>`*p`</u> where <u>`p`</u> is an identifier. In particular, expressions such "`v[i] = i++`" are undefined.
- Since the default constructor, copy constructor, destructor, and the operators `operator=`, `operator&`, and `operator,` (*i.e.*, `operator<comma>`) all have default meanings; they should all be explicitly defined in every class (see the example in Section 3.1). To avoid unwanted implicit calls to these constructors and operators, declare them <u>`private`</u>. A technique for providing a replacement "default" constructor that cannot be implicitly called is given in Section 3.3.

- For a C++ member function declared virtual in a base class the keyword virtual should be used in the definition of the function and all declarations and definitions of the function in each derived class even though it is optional.

- For a class that defines the operators `operator->`, `operator*`, and `operator[]`, ensure the equivalences between "`p->m`", "`(*p).m`", and "`p[0].m`". This will avoid unexpected errors when programmers assume the equivalence for classes that do not provide it. Also for a class that defines the operators `operator+`, `operator+=`, `operator++()`, and `operator++(int)`, ensure the equivalence of "`x = x + 1`", "`x += 1`", and "++x" and their relationship to "x++."

- Don't use `exit()` in a destructor; it may cause infinite recursion.

- Avoid templates. They may lead to unexpected code. For example, the template for the function `sort`

```
template <class T> void sort(T a[], int size)
{
    ⋮
    if (a[i] < a[j])
    ⋮
}
```

  works correctly for ints, floats, chars, and all classes that correctly define the operator `operator<`, but fails to work for strings declared `char*` because it compares the pointer values and not the strings.

G4.1.8 Minimizing Use of Literals. For strings, avoid "`char *p="string"`" as the literal `"string"` can be changed through the pointer (on some system this causes abnormal program termination). Instead use "`const char *p="string"`" or, if necessary, "`p = new char[sizeof("string")]; strcpy(p, "string")`."

G4.2.1 Minimizing the Use of Global Variables. Limit the visibility of variables and functions. One way of doing this is to declare local variables only where needed. The ability in C++ to declare variables anywhere within a block (rather than just the beginning) allows declarations to be made at their point of use. Variables local to a loop or branch of a conditional should be declared within the loop or branch and not be visible to the entire function.

If two functions must share a variable they should be placed in a separate file and the variables declared static in the file. This limits the visibility to the two functions only. The two could also occupy a common *Namespace* (part of the new C++ standard [ANSI, 1995]). If sensible, the two functions can be placed in a class and the "global" variable made an attribute of the class. Finally, static class attributes are shared by all instances of the class. These can be used in place of some global variables.

In class declarations, declare all attributes `private` where possible and `protected` where not. This limits the functions-that-can-change-an-attribute to class members for `private` attributes and class or sub-class members for `protected` attributes. Never have non-constant `public` attributes as erroneous values can be assigned to the attribute from anywhere in the code.

G4.4.1 Isolation of Alterable Functions. Isolation of alterable functions can be obtained in two ways. First, placing such functions in subclasses reduces the need to change the superclass when an alterable function in the subclass must be altered. For example, consider the class `displayable_playing_card`, a subclass of class `playing_card`. While playing cards seldom change, display technology does. However, all the changes for a new display technology should be localized to attributes and functions in `displayable_playing_card`. Those functions and attributes inherited from `playing_card` should remain unchanged. The second alternative is to group alterable functions together in their own class. This should be done with alterable functions not directly related to any other class.

3 TECHNIQUES AND EXAMPLES

This section illustrates techniques that can be used to improve the safety of C++ programs through a series of examples. Additional examples describing how to avoid excess use of friends, safe file pointers, named formal parameters, removing unions, maintaining code invariants, and controlling class conversions can be found in [Binkley, 1996].

3.1 SafeInt

The following class illustrates many of C++'s features for controlling access to data. Program reliability is improved by an understanding of these features. Following the class definition, the general use of some C++ features in safety critical code and some comments specific to class `SafeInt` are discussed. Note that in practice this class would occupy two files: `SafeInt.h` would include the declaration of the class, its attributes and functions, while `SafeInt.c++` would include the definitions (bodies) of the functions. The two are combined below for exposition purposes.

```
[ 1]   class SafeInt
[ 2]   {
[ 3]   private:
[ 4]       long int i;    // the actual value of the safe integer
[ 5]       operator int() const { return i;}
[ 6]       void     operator, (const SafeInt &);
[ 7]       SafeInt *operator& ();
[ 8]
[ 9]   public:
[10]       explicit SafeInt(const SafeInt other) { i = other.i;}
[11]       explicit SafeInt() { i = 0;}
[12]       explicit SafeInt(const int value) { i = value;}
[13]       ~SafeInt() {}
[14]
[15]       SafeInt operator=  (const SafeInt value)   { i = value.i; return(*this);}
[16]       SafeInt operator=  (const int     value)   { i = value;   return(*this);}
[17]       SafeInt operator+  (const int     b) const { return(SafeInt(i+b));}
[18]       SafeInt operator+  (const SafeInt b) const { return(SafeInt(i+b.i));}
[19]       SafeInt operator/  (const SafeInt b) const { if (b.i == 0) ... else ...}
[20]       SafeInt operator%  (const SafeInt b) const { ... }
[21]       int     operator!= (const SafeInt b) const { return(i != b.i);}
[22]       SafeInt operator++ ()      { i++; return(*this);}
[23]       SafeInt operator++ (int _) { SafeInt t = *this; i++; return(t);}
[24]       int     value()    {return i;}
[25]   };
```

Notes

- The keyword `explicit` (Lines [10]-[12]) prevents the compiler from implicitly calling these constructors.
- Because there is only one attribute, objects of class `SafeInt` are small and thus efficiently passed as call-by-value parameters. To pass larger objects, use constant reference parameters (see G4.1.6). For example, if `SafeInt`'s were larger, the addition operator would have been declared

```
SafeInt operator+ (const SafeInt &b);
```

- Every class should include a copy constructor (Line [10]), a default constructor (Line [11]), and an assignment operator (Line [15]), address-of operator (Line [7]), and a sequencing operator (Line [6]). If an operator is not desired, its definition should be declared in the `private`:
  - Line [7] makes `operator&` `private`, which prevents taking the address of an object (but does allow objects to be passed to functions as reference parameters).
  - If the default constructor (Line [11]) is moved to the `private` section of the class, then declarations of the form

```
SafeInt si;
```

are flagged as errors by the compiler. In this situation, no uninitialized `SafeInt`'s can be constructed. Only `SafeInts` constructed from other `SafeInts` (Line [10]) or `int`s (Line [12]) are allowed.

–Control of the copy constructor and assignment operator can be used to limit the number of objects of a class that are created. If the copy constructor (Line [10]) is moved to the `private` section of the class, then definitions of the form

```
SafeInt ss = si;
```

are flagged as errors by the compiler.

–Finally, if the assignment operator (Line [15]) is moved to the `private` section of the class, then assignment to `SafeInts` is not permitted; thus, statements such as

```
si1 = si2;
```

are flagged as errors by the compiler.

Failure to include these operators causes C++ to include default definitions that may have the wrong semantics. An excellent example of this failure is a class `String` that has a single attribute declared "`char *s`." For the assignment `string1 = string2` the default assignment operator, which does an attribute by attribute assignment, causes `string2.s` to point to the same memory as `string1.s`. This means that an update of `string1` will also affect the value of `string2`. The correct semantics are obtained by an assignment operator that copies the characters of the string.

- Line [5] provides a "use as an integer" operator, which is implicitly called in any context where an `int` is required, but a `SafeInt` is given. If `public`, it would allow unwanted implicit use of `SafeInts`. By making it `private`, the compiler issues an error message when a `SafeInt` is used in such a context. Many compilers will issue such messages if the definition is simply omitted, but providing such a definition in the `private` section makes explicit that a `SafeInt` should not be implicitly converted to an `int`. Explicit access can be provided via another access function, such as the function `value()` (Line [24]). This function should return a copy of the object; thus, preventing the caller from modifying the object. (The function `value()` (Line [24]) does not explicitly create a copy because returning an `int` implicitly returns a copy.)

- The assignment operator on Line [16] assigns an `int` to a `SafeInt`. Code checking the validity of the `int` could be placed in this function. For example, if `SafeInts` had to be less than 1000, that check could be made here. If this operator is omitted, then the int would be converted to a SafeInt using the constructor on Line [12] and then assigned using the assignment operator on Line [15]. (If both are present the assignment operator on Line [16] is used.)

- Similarly the operator on Line [17] performs the addition of a SafeInt and an int. Without this operator the int would first be converted into a SafeInt using the constructor on Line [12] then added using the addition operator on Line [18].

- There are two advantages to having SafeInt be a class: first, the built-in types char, int, and float (including modified versions unsigned char, long int, etc.) are not classes. This makes the built-in types unusable in certain contexts (*e.g.*, at least one of the parameters of an overloaded operator must be of class type). It also prohibits their being used as base classes (*e.g.*, to declare a subrange class).

- The second reason for having a class such as SafeInt is that it allows operators like division and remainder to have consistent predictable behavior: C++ leaves the definition of integer remainder up to the compiler writer. Most compiler writers use the hardware divide instruction for computing integer remainders. Unfortunately, some hardware divide instructions ensure that the remainder is positive while others do not. In class SafeInt, operator% can provide consistent results (unlike C++'s default % operator). As an added bonus, operator/ can also check for division by zero.

- Finally, use of the classes SafeInt and SafeUnsignedInt (not shown) prevents the mixing of signed and unsigned numbers. For example, many C++ compilers accept the following code without issuing a diagnostic.

```
{
    unsigned a = 1;
    int i = -5;
    a = i;
}
```

3.2 SafeFloat

Class SafeFloat, which is similar to SafeInt, allows control over floating point numbers. Beyond the concerns with integers, floating point numbers are subject to rounding errors. For example, tests such as "0.4 == 0.004 * 100" incorrectly return false on many systems. The definitions of operator==, operator<, and operator> in class SafeFloat account for this by including a tolerance for equality testing; they also preserve the relation that at most one of "a == b," "a < b," and "a > b is true." For example, comparing SafeFloats a and b as in

```
if (a == b)
```

is equivalent to

```
if (((b - TOLERANCE) <= a) && (a <= (b + TOLERANCE)))
```

In this example, *absolute* tolerance is used because it is easier to understand. A production version would use *relative* tolerance, where `TOLERANCE` is expressed as a fraction of the numbers involved and thus depends on the magnitude of those numbers. Only excerpts relating to `TOLERANCE` are presented below; missing definitions are still important and parallel those of `SafeInt` except for the inclusion of `TOLERANCE`.

```
class SafeFloat
{
private:
    double d;
    const float TOLERANCE = 0.00001;

public:
    SafeFloat(double initial_value);
    SafeFloat(SafeFloat &initial_value);
    SafeFloat() { d = 0;}

    int operator==(const SafeFloat value)
        { return ((d <= value.d + TOLERANCE) &&
                  (d >= value.d - TOLERANCE));}
    int operator< (const SafeFloat other)
        { return (d < other.d - TOLERANCE);}
    int operator> (const SafeFloat other)
        { return (d > other.d + TOLERANCE);}
};
```

3.3 Controlled Initialization

`Class` declarations are preferred over `struct` declarations because structure initialization provides no error checking. Thus, a declaration such as

```
struct worker child = {"Erin", 9};
```

is legal while

```
Worker child("Erin", 9);
```

can be flagged as a "labor law violation" by the constructor for class `Worker`.

Initialization includes the construction of default objects. Such objects are often implicitly constructed. Control over this construction can prevent unwanted object creation. It is also useful in preventing misinterpretations such as that in the following code:

```
person chris();
```

This code does not call the default constructor `person::person()` (even if it is public), to create `person` object `chris`. Instead it declares `chris` to be a function of zero arguments that returns an object of class `person`.

Similar to exception handling, this confusion can be avoided by using a constructor that takes an object of a "special" default class:

```
class Default { public:  Default() {}; };
```

The construction of default objects is accomplished as follows

```
person chris(Default());
```

4 SUMMARY

The dramatic increase in the use of software in safety critical applications such as flying airplanes, driving cars, and operating nuclear power plants, has increased the need for creating high-integrity software. This paper discusses the use of the C++ language in creating such software. It first considers C++ language features from the perspective of guidelines for use in constructing safety-critical systems. Overall, following these guidelines can improve the quality and reliability of programs; thus, adhering to these guidelines can lead to safer, more maintainable, C++ programs.

This paper also considers several techniques that can be incorporated into C++ programs. These techniques are meant as examples to illustrate some of the pitfalls of using C++ for high-integrity software. They are also intended to illustrate how some of the features of C++ can be used to produce high-integrity software. Even if a class such as `SafeInt` is not used in the production version of software (perhaps for performance reasons), its use during development restricts the use of integers. This has the effect of making the resulting code more reliable and safer.

REFERENCES

ANSI (1995). *C++ Programming Languages working paper.* American National Standards Institute, 1250 Eye Street NW, Suite 200, Washington DC 20005. Document number X3J16/95-0087/WG21/N0687.

Austin, T., Breach, S., and Sohi, G. (1994). "Efficient detection of all pointer and array access errors". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, Orlando, FL, USA. Association for Computing Machinery.

Binkley, D. (1996). C++ in safety critical systems. Technical report, National Institute of Standards and Technology, Gaithersburg, MD 20899. Report Number IR-5769.

Hatton, L. (1994). *Safer C: Developing software for high-integrity and safety-critical systems.* McGram-Hill International, Maidenhead, Berkshire, England.

Hecht, H., Hecht, M., Graff, S., Green, W., Lin, D., Koch, S., Tai, A., and Wendelboe, D. (1996). Review guidelines on software languages for use in nuclear power plant safety systems. Technical report, U. S. Nuclear Regulatory Commission, Washington, DC 20555. Report Number Nureg-CR-6463.