C++ in Safety Critical Systems

David W. Binkley *

Technology Administration
National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899
NIST IR 5769
November 1995

Abstract

The safety 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 constructing 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 [Hat94], has lead to research projects such as the safe C compiler [ABG94] and Les Hatton's recent book *Safer C* [Hat94].

*Email: binkley@sst.ncsl.nist.gov

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 [SoH95] to the C++ programming language. Second, it considers a series of techniques and examples for creating safer C++ programs.

2 GUIDELINES

This section considers the guidelines for creating safe software discussed by SoHaR [SoH95], in the context of the C++ programming language. The SoHaR report provides general programming guidelines for the assessment of software used in safety systems. It then 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.

This paper partitions the SoHaR guidelines into three groups:

1. **Outside.** Those guidelines outside the scope of C++. For example, control of memory paging is outside the control of a C++ programmer or the C++ compiler.
2. **General.** Those guidelines that represent general advice not directly applicable to particular C++ language features. For example, minimizing the nesting level of statements.
3. **Specific.** Those guidelines directly applicable to specific C++ language features. For example, minimizing dynamic memory allocation applies directly to C++'s built-in functions `new` and `delete`.

The first column of the following table indicates the group for each guideline. A further discussion of those guidelines in the general and specific categories appears after the table. Discussion of outside guidelines can be found in [SoH95].

Group	Number	Guideline
	1	Reliability
	1.1	Predictability of Memory Utilization
Specific	1.1.1	Minimizing Dynamic Memory Allocation
Outside	1.1.2	Minimizing Memory Paging and Swapping
	1.2	Predictability of Control Flow

Specific	1.2.1	Maximizing Structure
Specific	1.2.2	Minimizing Control Flow Complexity
Specific	1.2.3	Initialization of Variables before Use
Specific	1.2.4	Single Entry and Exit Points in Subprograms
Specific	1.2.5	Minimizing Interface Ambiguities
Specific	1.2.6	Use of Data Typing
General	1.2.7	Precision and Accuracy
Specific	1.2.8	Use of Parentheses rather than Default Order of Precedence
Specific	1.2.9	Separating Assignment from Evaluation
Outside	1.2.10	Proper Handling of Program Instrumentation
General	1.2.11	Control of Class Library Size
General	1.2.12	Minimizing Dynamic Binding
General	1.2.13	Control of Operator Overloading
	1.3	Predictability of Timing
Outside	1.3.1	Minimizing the Use of Tasking
Outside	1.3.2	Minimizing the Use of Interrupt Driven Processing
	2	Robustness
	2.1	Controlled Use of Software Diversity
General	2.1.1	Control of Internal Diversity
Outside	2.1.2	Control of External Diversity
	2.2	Controlled Use of Exception Handling
Outside	2.2.1	Local Handling of Exceptions
Outside	2.2.2	Preservation of External Control Flow
Outside	2.2.3	Uniformity of Exception Handling
	2.3	Input and Output Checking
General	2.3.1	Input Data Checking
General	2.3.2	Output Data Checking
	3	Traceability
	3.1	Use of Built-In Functions
General	3.1.1	Controlled Use of Built-in Functions
	3.2	Use of Compiled Libraries
General	3.2.1	Controlled Use of Compiled Libraries
	4	Maintainability
	4.1	Readability
General	4.1.1	Conformance to Indentation Guidelines
General	4.1.2	Descriptive Identifier Names

General	4.1.3	Comments and Internal Documentation
General	4.1.4	Limitations on Subprogram Size
Outside	4.1.5	Minimizing Mixed Language Programming
Specific	4.1.6	Minimizing Obscure or Subtle Programming Constructs
Specific	4.1.7	Minimizing Dispersion of Related Elements
Specific	4.1.8	Minimizing Use of Literals
	4.2	Data Abstraction
Specific	4.2.1	Minimizing the Use of Global Variables
Specific	4.2.2	Minimizing the Complexity of Class and Function Interfaces
	4.3	Functional Cohesiveness
General	4.3.1	Single Purpose Function and Procedure
	4.4	Malleability
General	4.4.1	Isolation of Alterable Functions
	4.5	Portability
General	4.5.1	Isolation of Non-Standard Constructs

2.1 Discussion of General Guidelines Relation to C++

The following terms 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.

Method. Methods are the functions (services) provided by the objects of a class.

These definitions are used in the following discussion of the general guidelines. Section 2.2 discusses the specific guidelines. Note that 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.

G1.2.7 Precision and Accuracy. Regression test the floating point data type 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 size has a particular number of bits is made. Use regression 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. (For additional safety and control, use classes such as `SafeInt`, described in example 3.1 in Section 3).

G1.2.11 Control of Class Library Size. Failure to break the problem down into the correct classes can lead to too few or too many classes. Both tend to obscure the relationship between the code and the problem domain and thus increase the possibility of errors. In other words, the problem and not some outside guideline should dictate the number of classes.

G1.2.12 Minimizing Dynamic Binding. In the SoHaR report [SoH95] this guideline confuses two unrelated topics: dynamic function binding and dynamic link libraries. Dynamic binding occurs when a virtual function is called. The actual function that gets called is determined dynamically at run time. Dynamic binding poses no more risk than static binding in well constructed classes. Dynamic link libraries are linked in with an executable at load time rather than link time. This produces smaller executables, but uses the library code on the target system and not that of 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, regression 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 (this operator is reasonable, but not obvious).

G2.1.1 Control of Internal Diversity. To support internal diversity, create multiple classes that respond to the same messages, but implement the corresponding methods differently. This allows the classes to be used interchangeably. These classes should also have different internal data representation *i.e.*, different attributes.

G2.3.1 Input Data Checking. Besides checking program input for validity, input data checking includes input parameter checking. For a class, this must include the current state of the object receiving the message. In this case, each class can include a (`private`) method responsible for normalizing its internal state. This method is called at the beginning of other methods of the class (see G2.3.2).

G2.3.2 Output Data Checking. In addition to checking program input for validity, each method could check its output. For methods that modify the internal state of an object (especially class constructors) this includes normalizing the current state of the object. For example, class `polar_point` would contain a function `normalize()`, which ensures a positive radius and an angle between 0 and 360 degrees. Checking validity of both input (G2.3.1) and output (G2.3.2) may be redundant.

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

Regression testing could be performed as part of program initialization to protect against the program being ported to a different machine. If a separate compliance test program is used, it must be run on both the development system and the target system if dynamic libraries are used (see G1.2.12). A better solution is to code for portability as in the example in Section 3.1.

G3.2.1 Controlled Use of Compiled Libraries. Use regression testing to verify compliance of library functions with standard and expected behavior (see G3.1.1). Class libraries that provide smart (safe) pointers and array bounds checking should be used.

G4.1.1 Conformance to Indentation Guidelines. Consistent code appearance helps reduce oversights made by human code inspectors.

G4.1.2 Descriptive Identifier Names. Appropriate names help the code be "self documenting" which reduces the need for comments. Self documenting code also avoids out-of-date comments that fail to reflect the current code. The style encouraged in object-oriented programs is to have lots of small functions [CGZ94]; this provides opportunities for including good internal documentation (see G4.1.3) through the use of good function names.

G4.1.3 Comments and Internal Documentation. Minimize the use of comments: If something can be stated in the language itself, it should be, and not just mentioned in a comment. Comments should identify the major data structures and the purpose, inputs, and output of each function. Well written code should be self documenting. Considering that object-oriented programs typically have short single purpose functions and each function should have a descriptive header comment, individual line comments may impede rather than help program understanding.

G4.1.4 Limitations on Subprogram Size. This is encouraged by the object-oriented programming style. A well written object-oriented program will naturally have many short functions.

G4.3.1 Single Purpose Function. Each function should implement a single thought. As mentioned in G1.2.9, functions should be divided into evaluation functions and update functions. No function should do both.

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 methods in `displayable_playing_card`. Those methods 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.

G4.5.1 Isolation of Non-Standard Constructs. As in G4.4.1, isolate these in separate subclasses or their own class.

2.2 Discussion of Specific Guidelines Relation to C++

This section discusses those guidelines applicable to specific C++ language features. 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. Avoid the use of C's `malloc` and `free`. Instead use (sparingly) C++'s `new` and `delete` operators. Overloading these operators changed dramatically with the new C++ standard [ANS95]; overloading them is dangerous until the definitions and compiler implementations stabilize.

To avoid memory leaks, a clear understanding of resource acquisition (allocation and release) is important. To avoid leaks, all classes should include a destructor that releases any memory allocated by the class' constructor. To ensure that destructors are called, class constructors should be declared

```
foo::foo(a, b):  a(...), b(...)  {...}
```

and not

```
foo::foo(a, b) {a = ...; b = ...; ...}
```

because failure in the constructor after the initialization of `b` will call the destructors for `a` and `b` in the first definition, but not in the second; thus, in the second there is a potential memory leak. A related example, which ensures that once a separate file is successfully opened it is closed, is presented in the example in Section 3.5.

Finally, 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.

G1.2.1 Maximizing Structure. Beyond obvious control-flow structure, this guideline includes structuring the data (primarily through classes and subclasses). Many of the precautions and guidelines herein deal with controlling problems using classes and class hierarchies; for example see G1.2.6.

G1.2.2 Minimizing Control Flow Complexity. The use of `break` and `continue` in loops should be avoided while `break` should always be used in switch statements. In general, the use of many small functions in object-oriented programs helps minimize control flow complexity.

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. They provide (restricted) interprocedural control transfers, which violate single exit. (Implementation of these features is patchy from compiler to compiler, which opens concerns about the correctness of their semantics and implementation). 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 in safety critical systems.

G1.2.6 Use of Data Typing. Use `class` in place of `struct` as it provides better access control. Also use a class hierarchy with virtual functions in place of a `union` as it provides type checking of the data stored in the "union" (see the example in Section 3.7). (This technique applies to any code that contains a discrete *type* or *kind* field.)

Since the fundamental types `int, float,` and `char` are not true classes, their use is restricted in certain contexts (function overloading for example). Creating classes for the fundamental types provides access control and also increases uniformity. For example, use the classes `SafeInt` (see the example in Section 3.1) in place of `int` and `SafeFloat` (see the example in Section 3.2) in place of `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.

G1.2.8 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.

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

- 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 data structures.

- 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 `v++` where `v` is a simple variable or `*p` where `p` 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 be explicitly defined in every class (see the example in Section 3.3). To avoid unwanted implicit calls to these constructors and operators, declare them `private`. A technique for providing a replacement "default" constructor is given in the example in Section 3.10.
- The scope resolution operator `::` should be used to explicitly indicate which of a collection of functions or variables with the same name is being used. This includes globals accessed as `::global_variable`.
- Avoid pointers to members. They unnecessarily complicate the code. Use virtual functions or redesign.
- 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++`."

G4.1.7 Minimizing Dispersion of Related Elements. Classes provide excellent containers for related elements. Their use should avoid dispersion. However, avoid the use of friends. Their use often indicates an oversight in the analysis or design. Particularly bad are declarations such as "`class x {friend class y; ...}`," which gives all methods of `class y` assess to the internal `private` attributes of `class x`. The example in Section 3.4 illustrates how most `friend` declarations can be removed without loss of efficiency.

G4.1.8 Minimizing Use of Literals. Literals (and `#define`d constants) should be replaced by identifiers declared `const` (or enumerated types for a group of related constants). Also replace `#define`d functions with inline functions. This allows the compiler to type check expressions and parameters and helps self document the code.

For literal 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 absolutely 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 [ANS95]). If sensible, the two functions can be placed in a class and the "global" variable made an attribute of the class. Alternatively, 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 `public` attributes as erroneous values to be assigned to the attribute from anywhere in the code. See G2.3.2 and the example in Section 3.8.

G4.2.2 Minimizing the Complexity Class and Function Interfaces. In function calls, avoid complicated actual parameter expressions. In classes, include only necessary functionality. All attributes and functions should be declared `private` if possible. If not, then they should be declared `protected` if possible and, if not, then they should be declared `public`. Private base classes can be used to hide implementation details of a derived class. Finally, use `const` for member functions that do not modify attributes. One common error in calling functions is to interchange parameters. The example in Section 3.6 provides an examples of how to simulate *named* parameter passing (not call-by-name) in C++ which avoids this problem.

In addition to the guidelines from [SoH95], the following guidelines are applicable to C++ programming.

- Don't return "`&local`" from the function "`int *f()`," or "`local`" from the functions "`int &f()`."
- Avoid nested classes.
- Don't use exit() in a destructor; it may cause an 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 `int`s, `float`s, `char`s, and all classes that correctly define the operator `operator<`, but fails to work for `char *` because it compares the pointer values and not the strings.
- The use of class conversion operators in place of constructors and friends can help reduce the need and use of friends. This is illustrated in the example in Section 3.9.
- Minimize the use of the C preprocessor. In particular, the use of `#define`.

3 TECHNIQUES AND EXAMPLES

This section illustrates techniques that can be used to improve the safety of C++ programs through a series of examples.

3.1 SafeInt

The following simple code illustrates many of C++'s features for controlling access to data. 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]
[ 7]   public:
[ 8]       SafeInt(const SafeInt other) { i = other.i;}
[ 9]       SafeInt() { i = 0;}
[10]       SafeInt(const int value) { i = value;}
[11]       ~SafeInt() {}
[12]
[13]       SafeInt operator=  (const SafeInt value)   { i = value.i; return(*this);}
[14]       SafeInt operator=  (const int     value)   { i = value;   return(*this);}
[15]       SafeInt operator+  (const int     b) const { return(SafeInt(i+b));}
[16]       SafeInt operator+  (const SafeInt b) const { return(SafeInt(i+b.i));}
[17]       SafeInt operator/  (const SafeInt b) const { if (b.i == 0) ... else ...}
[18]       SafeInt operator%  (const SafeInt b) const { ... }
[19]       int     operator!= (const SafeInt b) const { return(i != b.i);}
[20]       SafeInt operator++ ()      { i++; return(*this);}
[21]       SafeInt operator++ (int _) { SafeInt t = *this; i++; return(t);}
[22]       int     value()    {return i;}
[23]   };
```

Notes

- Line [4] declares `i` the attribute that holds the actual value of the `SafeInt`.
- 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 R4.1.6). For example, if `SafeInt`'s were larger, the addition operator would have been declared

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

Declaring a parameter (`const &`) preserve the semantics of values parameter passing, without the cost of copying large data structures.

- Every class should include a copy constructor (Line [8]), a default constructor (Line [9]), and an assignment operator (Line [13]). Failure to do so causes C++ to include default definitions that may have undesirable effects. If any of these operations is not desired its definition should be declared in the `private` part of the class. In particular, 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 [8]) is moved to the `private` section of the class, then definitions of the form

    ```
    SafeInt ss = si;
    ```

    are flagged as errors by the compiler.
  - If the default constructor (Line [9]) 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 `SafeInt`s constructed from other `SafeInt`s (Line [8]) or `int`s (Line [10]) are allowed.
  - Finally, if the assignment operator (Line [13]) is moved to the `private` section of the class, then assignment to `SafeInt`s is not permitted; thus, statements such as

    ```
    si1 = si2;
    ```

    are flagged as errors by the compiler.

- 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 (unwanted) use of `SafeInt`s. 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 [22]). This function should return a copy of the object; thus, preventing the caller from modifying the object. (Returning an `int` returns a copy of the `int` so no explicit copying is shown in the code.) Thus, `SafeInt` explicitly allows the programmer to control or avoid unwanted type conversions.

- The assignment operator on Line [14] 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 [10] and then assigned using the assignment operator on Line [13]. (If both are present the assignment operator on Line [14] is used.)

- Similarly the operator on Line [15] 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 [10] then added using the addition operator on Line [16].

- There are two reasons having `SafeInt` as a **class** is an advantage: first, the built-in types `char`, `int`, and `float` (including modified versions `unsigned char`, `long int`, etc.) are not classes. This makes the build-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.

- 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.

```
{
    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`. (Missing definitions parallel those of `SafeInt` except for the inclusion of `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. For example comparing `SafeFloats` `a` and `b` as in

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

is equivalent to

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

The operators `operator=`, `operator&` (address-of), `operator,` (sequencing) and the default and copy constructor all have default meaning. The `operator=` and the two constructors are discussed in the example of Section 3.1. This example shows making all the predefined operators "`private`". In particular, it discusses the operators `operator&` and `operator,` (the comma operator). Making `operator& private` prevents taking the address of an object (but allows objects to be passed to a function as a reference parameter). Making `operator, private` prevents an element of the class from appearing as the left operand of the comma operator.

```
class NoPredefines
{
private:
   NoPredefines(const NoPredefines &other);
   NoPredefines();
   NoPredefines* operator& ();
   void          operator, (void *);
   NoPredefines  operator= (const NoPredefines &value);

public:
   ...
};
```

Each line of the following function generates a compiler error because the above constructors and operators are `private`.

```
f(NoPredefines np, OtherClass oc)
{
    NoPredefines *p = &np;
    np, np;
    np, 1;
    np, 'c';
    np, oc;
}
```

An excellent example where the predefined default operator `operator=` has the wrong semantics is in the class `String`. The expected output of the following program is "`bye bye bYe bye`" (the assignment on Line [24] should affect `s1` but not `s2`); however, using the default definition of `operator=` the output is "`bye bye bYe bYe`" because Line [24] affects both `s1` and `s2`. The reason for this is that the default assignment operator `operator=`, which does a field by field assignment, causes `s2.s` to point to the same memory location as `s1.s` when the assignment on Line [21] is executed. This means that the update to `s1` on Line [24] affects the value of `s2`. In contrast, using the assignment operator on Line [12], which copies the characters of the string not just the pointer to the string, the update to `s1` on Line [24] does not affect the values of `s2`. (To simplify the example, `operator=` assumes there is enough space in the target string and the constructor assumes `new` does not return `0`.)

```
class String
{
private:
    char *s;

public:
    String(char *initial_value)
    {
        s = new char [sizeof(initial_value)];
        strcpy(s, initial_value);
    };
    String &operator=(String & rhs) {strcpy(s, rhs.s);return *this;};
    void print() {printf(" %s ", s);};
    char &operator[](int i) {return s[i];};
};

main()
{
    String s1 ("bye");
    String s2 ("hello");
    s2 = s1;
    s1.print();
    s2.print();
    s1[1] = 'Y';
    s1.print();
    s2.print();
}
```

3.4 Avoiding Friends

Friends are commonly used in C++ to allow global operators access to attributes of a class. Consider the following example.

```
class ComplexNumber
{
private:
    float real, img;
public:
    ComplexNumber(float r) {real = r; img = 0.0}
    ComplexNumber operator+ (ComplexNumber b) { ... }
};

ComplexNumber operator+(ComplexNumber a, ComplexNumber b)
{
    return (a.operator+(b));
}

```

```
[15]   main()
[16]   {
[17]       ComplexNumber a(5);
[18]       a = a + 3.0;
[19]       a = 4.0 + a;
[20]   }
```

Without the operator on Line [10], the addition on Line [18] is allowed, but the addition on Line [19] cannot be resolved by the compiler and therefore produces an error: On Line [18], the `3.0` is passed to the constructor on Line [6] resulting in the `ComplexNumber`, $3.0 + 0.0i$, which is then passed to `Complex::operator+`. On the other hand, for Line [19] there is no function in class `float` that takes a `ComplexNumber`[1]. The lack of symmetry is both annoying and problematic.

In contrast, the global operator `::operator+` (Line [10]) works with both additions: the `float` parameter is first converted to a `ComplexNumber` using the constructor on Line [6] before the addition. Normally, the function on Line [10] is declared as a friend of class `ComplexNumber` to allow it access to the `private` attributes of `class ComplexNumber`. Friend declarations violate the data abstraction and hiding and should be avoided as illustrated by this example (function inlining removes any run-time overhead).

3.5 Safe File Pointer

The class constructor and destructor semantics can be used to provide safe files. C++ semantics guarantee that once the constructor for a variable completes (such as for local `file` on Line [15] below) any control transfer out of the variable's scope will cause a call to its destructor. This is used in the following code to close the file; thus, the file use is safe as the function properly releases acquired resources.

[1]In fact, `float` is not even a class in C++, which further complicates the problem (See the examples in Sections 3.1 and 3.2).

```
[ 1]   #include "stdio.h"
[ 2]   class SafeFile
[ 3]   {
[ 4]   private:
[ 5]       FILE *f;
[ 6]
[ 7]   public:
[ 8]       SafeFile(const char *name, const char *mode)
               {f = fopen(name, mode);}
[ 9]       ~SafeFile() {fclose (f);}
[10]       operator FILE*() {return f;}
[11]   };
[12]
[13]   f(char *buf, int size)
[14]   {
[15]       SafeFile file("data", "r");
[16]
[17]       if (fread(buf, 1, size, file) != size)
[18]           return(-1);
[19]       ...
[20]       fclose(file);
[21]   }
```

- The `return` on Line [18] forces a call to `SafeFile`'s destructor (Line [9]), which closes the file. If `FILE*` replaces `SafeFile`, then the file would remain open and function `f` would fail to release the file descriptor, whenever Line [18] is executed.
- The operator `operator FILE` defined on Line [10] is used to convert an instance of class `SafeFile` to type `FILE*`. This allows a variable of class `SafeFile`, such as `file`, to be used in any context requiring a `FILE*`.

3.6 Named Formal Parameters

Many errors occur because of changed or misunderstood function interfaces. Beyond good documentation, parameter validity checking, and parameter type alternating,[2] not much can be done to ensure actuals are passed to the correct formals. This example considers an alternative that emulates passing parameters by name (not Algol 60's *call-by-name*). For example, the FORTRAN open statement

```
open(UNIT=in, file='data.text', status='old')
```

[2] Parameter type alternating attempts to avoid having adjacent parameters with the same type. This, unfortunately, makes the code harder to read as it forces unnatural parameter ordering and also requires sufficient parameters of differing types.

passes the actual '`in`' to the formal '`unit`', the actual `'data.text'` to the formal '`file`' and the actual `'old'` to the formal '`status`'. Although `file` and `status` both have type string, they are harder to confuse when passed using names.

The following examples emulate matching actuals with formals by name. Two versions are shown: the first simply uses a C struct, the second uses a C++ class. With the first, parameter validity checking must be done in the called function, while the second allows checking to be done by the class constructor; thus, separating it from the actual computation of the called function.

```
typedef struct
{
    int height;
    int length;
    int width;
} volume_parameters;

int compute_volume(const volume_parameters & parameters)
{
    int volume;
    volume = parameters.height * parameters.length * parameters.width;
    return (volume);
}

example_call()
{
    volume_parameters v;

    // replace the call "int answer = compute_volume(4,3,5)" with
    v.height = 4;
    v.width = 3;
    v.length = 5;
    int answer = compute_volume(v);
}
```

In this example, using `struct` field names (*e.g*, `height`) makes it harder to confuse the parameters even though they all have the same type.

The second example uses a C++ `class` in place of the `struct` in the parameters to the `fread` library call. (In this example, underscores are used in the attribute names because C++ uses the same name space for attributes and methods.)

```
class FreadParameters
{
private:
    void *_buffer;
    int   _item_size;
    int   _number_of_items;
    FILE *_stream;
public:
    FreadParameters() {_buffer = 0, _item_size = 0,
                       _number_of_items = 1, _stream = 0;}
    set_buffer(void *b)
    {
        if (b == 0) error()
        else _buffer = b;
    }
    set_item_size(int is) {_item_size = is;}
    set_number_of_items(int ni)
    {
        if (ni < 1) error()
        else _number_of_items = ni;
    }
    set_stream(FILE *stream);

    void *buffer()          {return(_buffer);}
    int   item_size()       {return(_item_size);}
    int   number_of_items() {return(_number_of_items);}
    FILE *stream()          {return(_stream);}
};

example_call()
{
    FreadParameters f;

    // replace the call "fread(buf, 1, 60, file)" with
    f.set_buffer(buf);
    f.set_item_size(1);
    f.set_number_of_items(60);
    f.set_stream(file);
    my_fread(f);
}

my_fread(FreadParameters &p)
{
 ...    // no parameter validity checking necessary here
}
```

Notes

- Matching actuals and formals by name avoids confusion between `number_of_items` and `item_size`, which are adjacent parameters of the same type.
- The functions for setting the `buffer` and `number_of_items` illustrate parameter validity checking: Line [13] ensures `buffer` is non-zero while Line [19] ensures that `number_of_items` is at least one.
- Placing parameter validity checking in the class constructor, separates it from the implementation of the called function and consequently, improves code clarity.

3.7 Union Removal

C++'s `union` type is untagged and therefore unsafe. Unions require the programmer to include a separate tag field indicating which field of the `union` is "current." Unfortunately, in large projects it is increasingly likely that this tag is incorrect or is left out in a particular function either unintentionally or because "that case can't possibly happen here."

The following example shows a simple `union` and then the class hierarchy that replaces it. One advantage of the class hierarchy is that is provides automated tag checking. The particular `union` represents a literal pool entry for a compiler symbol table. A literal is assumed to be either an `int`, a `char`, or a `float`. Adding new literal kinds is discussed below. First the original `union` and an example function that operates on it.

```
enum literal_kind{INT, CHAR, FLOAT};
typedef struct
{
    literal_kind kind;
    union
    {
        int   int_value;
        char  char_value;
        float float_value;
    };
} literal_union;

print_literal_union(literal_union *l)
{
    switch(l->kind)
    {
        case INT:   printf("%s = %d\n", l->name, l->int_value); break;
        case CHAR:  printf("%s = %c\n", l->name, l->char_value); break;
        case FLOAT: printf("%s = %f\n", l->name, l->float_value); break;
    }
}
```

The class hierarchy replacing this `union` includes the *pure virtual* class "`literal`." Such a class can have no instances; rather, it provides an interface. In this case, all classes derived from `literal` must override all pure virtual functions (those whose declarations end with `= 0`). This gives different types of literals the same interface. (Constructors are not shown in the code.)

```
class literal
{
public:
    virtual void print() = 0;
};

class int_literal : public literal
{
private:
    int value;
public:
    void print() {printf("%s = %d\n", name, value);};
};

class char_literal : public literal
{
private:
    char value;
public:
    void print() {printf("%s = %c\n", name, value);};
};

class float_literal : public literal
{
private:
    float value;
public:
    void print() {printf("%s = %f\n", name, value);};
};
```

The following code illustrate the violation possible with the `union` and how the C++ class hierarchy avoids it.

```
[ 1]   main()
[ 2]   {
[ 3]       literal_union lu;
[ 4]
[ 5]       lu.kind = INT;                // should be lu.kind = FLOAT;
[ 6]       lu.float_value = 5.6;
[ 7]       print_literal_union(&lu);
[ 8]
[ 9]       int_literal il(5);
[10]       il.print();
[11]
[12]       int_literal fl(5.6);          // compile time error
[13]       fl.print();
[14]   }
```

Notes

- In Line [5] the wrong type is assigned to `lu.kind`, but this is not (and cannot be) trapped by the compiler. In contrast, Line [12] generates a compiler error message because there is no constructor in class `int_literal` that takes a `float` as its argument.
- There is no possibility of trying to print an int or a float as a character. Such a statement simply cannot be stated (without abusive casting).
- Adding new kinds of literals requires deriving a new class from class `literal`. This new class must override all the virtual functions and thus is guaranteed to provide the necessary functionality required in other parts of the program.

3.8 Polar Point

The polar point class illustrates the use of a `private` method to maintain an invariant on the internal state of an object. The class enforces the assertion that the radius `rho` is positive and that the angle `theta` is between `0` and `360`. This simplifies writing methods that manipulate points, for example consider writing the method `quadrant()` with and without this assertion. All methods that manipulate a point call `normalize()` before returning; this maintains the invariant. Unlike `structs`, the visibility rules for C++ guarantee no outside code can violate the assertion by modifying `rho` or `theta`.

```
class point
{
private:
    SafeFloat rho, theta;
    void normalize()
    {
        if (rho < 0.0)
        {
            theta += 180.0;
            rho = -rho;
        }

        while (theta > 360.0)
            theta -= 360.0;

        while (theta < 0.0)
            theta += 360.0;
    }

public:
    point(SafeFloat r, SafeFloat t) : rho(r), theta(t) {normalize();}
    . . .
};
```

3.9 Class Conversions

Two incomplete classes are shown below to illustrate the construction of an object of one class from an object of another.

```
class bar
{
public:
    operator foo();
};

class foo
{
public:
    foo(bar b);
};

void f_of_foo(foo f);

main()
{
    bar b;
    f_of_foo(b);
}
```

The function call on Line [18] requires converting the `bar b` to an object of class `foo`. There are two ways of doing this: using the constructor on Line [10], which provides a method for constructing a `foo` from a `bar`, or using the operator on Line [4] to produce an object `foo` from a `bar` object. However, the operator on Line [4] and the constructor on Line [10] cannot coexist because it is ambiguous which to use on Line [18].

In choosing between them, the following guidelines are suggested: favor the operator (Line [4]) because it has access to the internal attributes of `class bar`. This reduces the temptation to use `friends` and the need for access functions, which the constructor in class `foo` would need.

The constructor version is only necessary when an object is constructed from two or more parts. For example, water (H_2O) is constructed from two instances of class `hydrogen` (H) and one instance of class `oxygen` (O). It is not possible for class `hydrogen` or class `oxygen` to provide an `operator water()` because an instance of class water is composed of both `hydrogen` and `oxygen`. Thus, class `water` should include the constructor

```
water::water(hydrogen h1, hydrogen h2, oxygen o);
```

3.10 Explicit Default

This section demonstrates how to provide a default constructor that is not implicitly called. This provides explicit control over when an object is constructed (see also the examples in Sections 3.1 and 3.3). Similar to exceptions, the technique uses a new class, `Default`, to indicate the desire to use the default constructor.

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

class person
{
private:
    char *name;
    int age;
    person() {error("private implicit default person created");}
public:
    const int default_age = 5;
    person(Default)       {name = "default_name", age = default_age;}
    person(char *n, int a) {name = n, age = a;}
};
```

```
examples()
{
    // person p1;
    // compiler error: constructor ‘person::person()’ is private
    person p2(Default());
    person p3("Judy", 29);
}
```

This technique is useful in preventing misinterpretations such as that in the following code:

```
main()
{
    person chris();
    . . .
}
```

Even if the default constructor `person::person()` is public, this code does not call it to create person object `chris`. Instead it declares `chris` to be a function of zero arguments that returns an object of class `person`. This confusion can be avoided by using the explicit default technique.

3.11 Replace Structure Initialization with Class Constructor

Universally, `struct` should be replaced by `class`. One reason for this is that structure initialization provides no error checking as illustrated in the following code.

```
struct worker
{
    char *name;
    int age;      // must be 18 years old to work
};

wrong()
{
    struct worker child = {"Erin", 7};
}
```

Replacing `struct worker` with a class and the initialization with the appropriate constructor allows for such error checking.

```
class Worker
{
    char *name;
    int age;

public:
    Worker(char *initial_name, int initial_age)
    {
        if (initial_age < 18)
            labor_law_violation();
        else
        {
            age = initial_age;
            name = initial_name;
        }
    }
};

right()
{
    Worker child("Erin", 7);    // flagged as a labor_law_violation
}
```

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. Adhering to these guidelines can lead to safer, more maintainable, C++ programs. This is true even for non-safety critical software.

The paper also considers a collection of techniques that can be incorporated into the development of C++ programs. These classes 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 (*e.g.*, for performance reasons), its use during development restricts the use of integers. This has the effect of making the resulting code more predictable and safer.

References

[ABG94] T.M. Austin, S.E. Breach, and Sohi G.S. "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, June 1994. Association for Computing Machinery.

[ANS95] ANSI. *C++ Programming Languages working paper.* American National Standards Institute, 1250 Eye Street NW, Suite 200, Washington DC 20005, 1995. Document number X3J16/95-0087/WG21/N0687.

[CGZ94] B Calder, D. Grunwald, and B. Zorn. "Quantifying behavioral differences between C and C++ programs". *Journal of Programming Languages*, 2(4):313–353, December 1994.

[Hat94] L. Hatton. *Safer C: Developing software for high-integrity and safety-critical systems.* McGram-HIll International, Maidenhead, Berkshire, England, 1994.

[SoH95] SoHaR. *Assessment of software languages for use in nuclear power plant safety systems.* SoHaR Incorporated, Beverly Hills, CA 90211, 1995. Report Number J1030-2.