An Abbreviated C++ Code Inspection Checklist

1. VARIABLE DECLARATIONS

1.1 Arrays

1.1.1 Is an array dimensioned to a hard-coded constant?

```
int intarray[13];
```

should be

```
int intarray[TOT_MONTHS+1];
```

1.1.2 Is the array dimensioned to the total number of items?

```
char entry[TOTAL_ENTRIES];
```

should be

```
char entry[LAST_ENTRY+1];
```

The first example is extremely error-prone and often gives rise to off-by-one errors in the code. The preferred (second) method permits the writer to use the LAST_ENTRY identifier to refer to the last item in the array. Instances that require a buffer of a certain size are rarely rendered invalid by this practice, which results in the buffer being one element bigger than absolutely necessary.

1.2 Constants

1.2.1 Does the value of the variable never change?

```
int months_in_year = 12;
```

should be

```
const  unsigned months_in_year = 12;
```

1.2.2 Are constants declared with the preprocessor #define mechanism?

```
#define MAX_FILES 20
```

should be

```
const unsigned MAX_FILES = 20;
```

1.2.3 Is the usage of the constant limited to only a few (or perhaps only one) class? If so, is the constant global?

```
const unsigned MAX_FOOS = 1000;

const unsigned MAX_FOO_BUFFERS = 40;
```

should be

```
class foo {
    public:
            enum { MAX_INSTANCES = 1000; }
            ...
    private:
            enum { MAX_FOO_BUFFERS = 40; }
            ...
    };
```

If the size of the constant exceeds int, another mechanism is available:

```
class bar {
public:
    static const long MAX_INSTS;
    ...
    };
const long bar::MAX_INSTS = 70000L;
```

The keyword static ensures there is only one instance of the variable for the entire class. Static data items are not permitted to be initialized within the class declaration, so the initialization line must be included in the implementation file for class bar.

Static constant members have one drawback: you cannot use them to declare member data arrays of a certain size. This is because the value is not available to the compiler at the point which the array is declared in the class.

1.3 Scalar Variables

1.3.1 Does a negative value of the variable make no sense? If so, is the variable signed?

```
int age;
```

should be

```
unsigned  int age;
```

This is an easy error to make, since the default types are usually signed.

1.3.2 Does the code assume char is either signed or unsigned?

```
typedef char SmallInt;

SmallInt mumble = 280; // WRONG on Borland C++ 3.1 // or
MSC/C++ 7.0!
```

The typedefs should be

```
typedef  unsigned  char SmallUInt;
typedef  signed  char SmallInt;
```

1.3.3 Does the program unnecessarily use float or double?

```
double acct_balance;
```

should be

```
unsigned long acct_balance;
```

In general, the only time floating point arithmetic is necessary is in scientific or navigational calculations. It is slow, and subject to more complex overflow and underflow behavior than integer math is. Monetary calculations, as above, can often be handled in counts of cents, and formatted properly on output. Thus, acct_balance might equal 103446, and print out as $1,034.46.

1.4 Classes

1.4.1 Does the class have any virtual functions? If so, is the destructor non-virtual?

Classes having virtual functions should always have a virtual destructor. This is necessary since it is likely that you will hold an object of a class with a pointer of a less derived type. Making the destructor virtual ensures that the right code will be run if you delete the object via the pointer.

1.4.2 Does the class have any of the following:

Copy-constructor
Assignment operator
Destructor

If so, it generally will need all three. (Exceptions may occasionally be found for some classes having a destructor with neither of the other two.

2. DATA USAGE

2.1 Strings

2.1.1 Can the string ever not be null-terminated?

2.1.2 Is the code attempting to use a strxxx() function on a non-terminated char array, as if it were a string?

2.2 Buffers

2.2.1 Are there always size checks when copying into the buffer?

2.2.2 Can the buffer ever be too small to hold its contents?

For example, one program had no size checks when reading data into a buffer because the correct data would always fit. But when the file it read was accidentally overwritten with incorrect data, the program crashed mysteriously.

2.3 Bitfields

2.3.1 Is a bitfield really required for this application?

2.3.2 Are there possible ordering problems (portability)?

3. INITIALIZATION

3.1 Local Variables

3.1.1 Are local variables initialized before being used?

3.1.2 Are C++ locals created, then assigned later?

This practice has been shown to incur up to 350% overhead, compared to the practice of declaring the variable later in the code, when an initialization variable is known. It is the simple matter of putting a value in once, instead of assigning some default value, then later throwing it away and assigning the real value.

3.2 Missing Reinitialization

3.2.1 Can a variable carry an old value forward from one loop iteration to the next?

Suppose the processing of a data element in a sequence causes a variable to be set. For example, a file might be read, and some globals initialized for that file. Can those globals be used for the next file in the sequence without being re-initialized?

4. MACROS

4.1 If a macro's formal parameter is evaluated more than once, is the macro ever expanded with a actual parameter having side effects?

For example, what happens in this code:

```
#define max(a,b) ( (a) > (b) ? (a) : (b) )

max(i++, j);
```

4.2 If a macro is not completely parenthesized, is it ever invoked in a way that will cause unexpected results?

```
#define max(a, b) (a) > (b) ? (a) : (b) result = max(i, j) + 3;
```

This expands into:

```
result = (i) > (j) ? (i) : (j)+3;
```

See the example in 4.1 for the correct parenthesization.

4.3 If the macro's arguments are not parenthesized, will this ever cause unexpected results?

```
#define IsXBitSet(var)  (var && bitmask) result = IsXBitSet( i || j );
```

This expands into:

```
result = (i || j && bitmask);  // not what expected!
```

The correct form is:

```
#define IsXBitSet(var) ((var) && (bitmask))
```

5. SIZING OF DATA

5.1 In a function call with arguments for a buffer and its size, is the argument to sizeof different from the buffer argument?

For example:

```
memset(buffer1, 0, sizeof(buffer2));  // danger!
```

This is not always an error, but it is a dangerous practice. Each instance should be verified as (a) necessary, and (b) correct, and then commented as such.

5.2 Is the argument to sizeof an incorrect type?

Common errors:

sizeof(ptr) instead of sizeof(*ptr)
sizeof(*array) instead of sizeof(array)
sizeof(array) instead of sizeof(array[0]) (when the user wanted the size of an element)

6. DYNAMIC ALLOCATION

6.1 Allocating Data

6.1.1 Is too little space being allocated?

6.1.2 Does the code allocate memory and then assume someone else will delete it?

This is not always an error, but should always be prominently documented, along with the reason for implementing in this manner. Constructors which allocate, paired with destructors which deallocate, are an obvious exception, since a single object has control of its class data.

6.1.3 Is malloc(), calloc(), or realloc() used in lieu of new?

C standard library allocation functions should never be used in C++ programs, since C++ provides an allocation operator.

If you find you must mix C allocation with C++ allocation:

6.2.2 Is malloc , calloc , or realloc invoked for an object which has a constructor?

Program behavior is undefined if this is done.

6.2 Deallocating Data

6.2.1 Are arrays being deleted as if they were scalars?

```
delete  myCharArray;
```

should be

```
delete [] myCharArray;
```

6.2.2 Does the deleted storage still have pointers to it?

It is recommended that pointers are set to NULL following deletion, or to another safe value meaning "uninitialized." This is neither necessary nor recommended within destructors, since the pointer variable itself will cease to exist upon exiting.

6.2.3 Are you deleting already-deleted storage?

This is not possible if the code conforms to 6.2.2. The draft C++ standard specifies that it is always safe to delete a NULL pointer, so it is not necessary to check for that value.

If C standard library allocators are used in a C++ program (not recommended):

6.2.4 Is delete invoked on a pointer obtained via malloc , calloc , or realloc?

6.2.5 Is free invoked on a pointer obtained via new?

Both of these practices are dangerous. Program behavior is undefined if you do them, and such usage is specifically deprecated by the ANSI draft C++ standard.

7. POINTERS

7.1 When de-referenced, can the pointer ever be NULL?

7.2 When copying the value of a pointer, should it instead allocate a copy of what the first pointer points to?

8. CASTING

8.1 Is NULL cast to the correct type when passed as a function argument?

8.2 Does the code rely on an implicit type conversion?

C++ is somewhat charitable when arguments are passed to functions: if no function is found which exactly matches the types of the arguments supplied, it attempts to apply certain type conversion rules to find a match. While this saves unnecessary casting, if more than one function fits the conversion rules, it will result in a compilation error. Worse, it can cause additions to the type system (either from adding a related class, or from adding an overloaded function) to cause previously working code to break!

See the Section (A) for an example.

9. COMPUTATION

9.1 When testing the value of an assignment or computation, is the parenthesization incorrect?

if (a = function() == 0)

should be

if ((a = function()) == 0)

9.2 Can any synchronized values not get updated?

Sometimes, a group of variables must be modified as a group to complete a single conceptual "transaction." If this does not occur all in one place, is it guaranteed that all variables get updated if a single value changes? Do all updates occur before any of the values are tested or used?

10. CONDITIONALS

10.1 Are exact equality tests used on floating point numbers?

if (someVar == 0.1)

might never be evaluated as true. The constant 0.1 is not exactly representable by any finite binary mantissa and exponent, thus the compiler must round it to some other number. Calculations involving someVar may never result in it taking on that value.

Solution: use >, >=, <, or <=, depending on which direction you wish the variable bound.

10.2 Are unsigned values tested greater than or equal to zero?

if (myUnsignedVar >= 0)

will always evaluate true.

10.3 Are signed variables tested for equality to zero or another constant?

if (mySignedVar) // not always good

if (mySignedVar >= 0) // better!

if (mySignedVar <= 0) // opposite case

If the variable is updated by any means other than ++ or --, it may miss the value of the test constant entirely. This can cause subtle and frightening bugs when code executes under conditions that weren't planned for.

10.4 If the test is an error check, could the "error condition" actually be legitimate in some cases?

11. FLOW CONTROL

11.1 Control Variables

11.1.1 Is the lower limit an exclusive limit?

11.1.2 Is the upper limit an inclusive limit?

By always using inclusive lower limits and exclusive upper limits, a whole class of off by-one errors is eliminated. Furthermore, the following assumptions always apply:

the size of the interval equals the difference of the two limits
the limits are equal if the interval is empty
the upper limit is never less than the lower limit

Examples: instead of saying x>=23 and x<=42 , use x>=23 and x<43.

11.2 Branching

11.2.1 In a switch statement, is any case not terminated with a break statement?

When the same block of code follows several cases, they may be "stacked" together and the code terminated with a single break.

Cases may also be exited via return.

All other circumstances requiring "drop through" cases should be clearly documented in a strategic comment before the switch. This should only be used when it makes the code simpler and clearer.

11.2.2 Does the switch statement lack a default branch?

There should always be a default branch to handle unexpected cases, even when it appears that the code can never get there.

11.2.3 Does a loop set a boolean flag in order to effect an exit?

Consider using break instead. It is likely to simplify the code.

11.2.4 Does the loop contain a continue?

If the continue occurs in the body of an if conditional, consider replacing it with an else clause if it will simplify the code.

12. ASSIGNMENT

12.1 Assignment operator

12.1.1 Does "a += b" mean something different than "a = a + b"?

The programmer should never change the semantics of relationships between operators. For the example here, the two statements above are semantically identical for intrinsic types (even though the code generated might be different), so for a user defined class, they should be semantically identical, too. They may, in fact, be implemented differently (+= should be more efficient).

12.1.2 Is the argument for a copy constructor or assignment operator non const ?

12.1.3 Does the assignment operator fail to test for self-assignment?

The code for operator=() should always start out with:

```
if (this == &right_hand_arg )
return *this;
```

12.1.4 Does the assignment operator return anything other than a const reference to this?

Failure to return a reference to this prevents the user from writing (legal C++):

```
a = b = c;
```

Failure to make the return reference const allows the user to write (illegal C++):

```
(a = b) = c;
```

12.2 Use of assignment

12.2.1 Can this assignment be replaced with an initialization?

(See question 3.1.2 and commentary.)

12.2.2 Is there a mismatch between the units of the expression and those of the variable?

For example, you might be calculating the number of bytes for an array when the number of elements was requested. If the elements are big (say, a long, or a struct!), you'd be using way too much memory.

13. ARGUMENT PASSING

13.1 Are non-intrinsic type arguments passed by value?

```
Foo& do_something( Foo anotherFoo, Bar someThing );
```

should be

```
Foo&  do_something( const Foo& anotherFoo, const Bar& someThing );
```

While it is cheaper to pass ints, longs, and such by value, passing objects this way incurs significant expense due to the construction of temporary objects. The problem becomes more severe when inheritance is involved. Simulate pass-by-value by passing const references.

14. RETURN VALUES

14.1 Is the return value of a function call being stored in a type that is too narrow?

(See Section (B))

14.2 Does a public member function return a non const reference or pointer to member data?

14.3 Does a public member function return a non const reference or pointer to data outside the object?

This is permissible provided the data was intended to be shared, and this fact is documented in the source code.

14.4 Does an operator return a reference when it should return an object?

14.5 Are objects returned by value instead of const references?

(See question 13.1 and commentary.)

15. FUNCTION CALLS

15.1 Varargs functions (printf, and other functions with ellipsis ...)

15.1.1 Is the FILE argument of fprintf missing? (This happens all the time.)

15.1.2 Are there extra arguments?

15.1.3 Do the argument types explicitly match the conversion specifications in the format string? (printf and friends.)

Type checking cannot occur for functions with variable length argument lists.

For example, a user was surprised to see nonsensical values when the following code was executed:

```
printf(" %d %ld \n", a_long_int, another_long_int);
```

On that particular system, int s and long s were different sizes (2 and 4 bytes, respectively). printf() is responsible for manually accessing the stack; thus, it saw "%d" and grabbed 2 bytes (an int).

It then saw "%ld" and grabbed 4 bytes (a long). The two values printed were the MSW of a_long_int , and the combination of a_long_int 's LSW and another_long_int 's MSW.

Solution: ensure types explicitly match. If necessary, arguments may be cast to smaller sizes (long to int) if the author knows for certain that the smaller type can hold all possible values of the variable.

15.2 General functions

15.2.1 Is this function call correct? That is, should it be a different function with a similar name? (e.g. strchr instead of strrchr?)

15.2.2 Can this function violate the preconditions of a called function?

16. FILES

16.1 Can a temporary file name not be unique?

(This is, surprisingly enough, a common design bug.)

16.2 Is a file pointer reused without closing the previous file?

```
fp = fopen(...);

fp = fopen(...);
```

16.3 Is a file not closed in case of an error return?

Section A. Errors due to implicit type conversions.

Code which relies upon implicit type conversions may become broken when new classes or functions are added. For example:

```
class String {
      public:
              String( char *arg ); // copy constructor operator const char* ()
const;
      // ...
};

void foo( const String& aString );
void bar( const char *anArray );

// Now, we added the following class

class Word {
      public:
              Word( char *arg ); // copy constructor
      // ...
};

// need another foo that works with "Words"

void foo( const Word& aWord );

int gorp()
{
      foo("hello"); // This used to work!
      // Now it breaks! What gives?

      String baz = "quux";
      bar(baz); // but this still works.
}
```

The code worked before class Word and the second foo() were added. Even though there was no foo() accepting an argument of type const char * (i.e. a constant string like "hello"), there is a foo() which takes a constant String argument by reference. And (un)fortunately, there is also a way to convert Strings to char * 's and vice-versa. So the compiler performed the implicit conversion.

Now, with the addition of class Word , and another foo() which works with it, there is a problem. The line which calls foo("hello") matches both:

```
void foo( const String& );

void foo( const Word& );
```

Since the mechanisms of the failure may be distributed among two or more header files in addition to the implementation file, along with a lot of other code, it may be difficult to find the real problem.

The easiest solution is to recognize while coding or inspecting that a function call results in implicit type conversion, and either (a) overload the function to provide an explicitly typed variant, or (b) explicitly cast the argument.

Option (a) is preferred over (b), since (b) defeats automatic type checking. Option (a) can still be implemented very efficiently, simply by writing the new function as a forwarding function and making it inline.

Section B. Errors due to loss of "precision" in return values

Functions which can return EOF should not have their return values stored in a char variable.
For example:

```
int getchar(void);

char chr;

while ( (chr = getchar()) != EOF )  { ... };
```

should be:

```
int tmpchar;

while ( (tmpchar = getchar()) != EOF )  {
      chr = (char) tmpchar; // or use casted tmpchar ...
               // throughout...
};
```

The practice in the top example is unsafe because functions like getchar() may return 257 different values: valid characters with indexes 0 -255, plus EOF (-1). If sizeof(int) > sizeof(char) , then information will be lost when the high-order byte(s) are scraped off prior to the test for EOF. This can cause the test to fail. Worse yet, depending on whether char is signed or unsigned by default on the particular compiler and machine being used, sign extension can wreak havoc and cause some of these loops never to terminate.

Section C. Loop Checklist

The following loops are indexed correctly, and are handy for comparisons when doing inspections. If the actual code doesn't look like one of these, chances are that something is wrong—or at least that something could be clearer.

Acceptable forms of for loops which avoid off-by-one errors.

```
for ( i = 0; i <= max_index; ++i )

for ( i = 0; i < sizeof(array); ++i )

for ( i = max_index; i >= 0; --i )

for ( i = max_index; i ; --i )
```

Copyright Notices