FPCL Coding Standards
(or, "How to Zoom in C++")

Walter E. Brown
Fermilab Physics Class Library Task Force

Draft 0.7.13; December 3, 1996

Contents

1 Approach

We believe, of course, that software must be correct to be useful. Studies have shown that well-formatted code is typically also well-structured: it is a strong symptom of a sound design and so tends to exhibit far fewer defects. Further, such well-presented code is generally convenient to read as well as clearly persuasive of its correctness.

We further believe that essentially all software evolves throughout its useful lifetime. This evolution may result in fewer defects, better performance, added functionality, or other improvement. Independent of motive, to carry out such evolution requires that a literate, knowledgeable software specialist be able to read and quickly comprehend the intent of the software at all levels.

We believe that software is a form of literature. Accordingly, we seek to apply reasonable rules of literacy to the FPCL software, and to apply these rules in a manner consistent with good taste. While there are many possible valid approaches to achieving such literacy, we believe it is useful to recommend one approach to be used consistently throughout FPCL.

It is, we believe, less that consistency is an inherent virtue than that lack of consistency in coding style is an inherent vice. Therefore, to the extent possible, we require consistent adoption of the following recommendations in all FPCL code. While recognizing that coding styles are often a matter of personal preference rather than technical merit, we also suggest these rules to other FNAL experiments as well.

2 General appearance and layout

2.1 Whitespace

Use whitespace to promote readability. Judicious and stylized use of blank lines, indentation, and spacing between words serve as aids to the reader in conventional literature; so ought they be consistently used in program code, and for the same purpose.

1. Use only blank, tab, and newline characters to insert whitespace into source code. Unprintable characters may not be introduced for any purpose.

2. Place two blank lines between major sections of code (e.g., functions). Use one blank line to separate code subsections.

3. When coding a statement, space

   (a) after each comma, keyword, colon, and semicolon (except at the end of a line);

   (b) before and after each binary infix operator (but not around `.` or `->`);

   (c) before each unary operator; and

   (d) around `?:`.

   Do not space before the open parenthesis denoting function call. Otherwise, use judgment as to spacing before and after parentheses and brackets; always make expressions as easy to read as possible.

4. Indent all code constituting the body of each of the following control structures:

   (a) iteration statements including `do`, `for`, `while`;

   (b) selection statements including `if`, `else`, `case` and `default` (within `switch`); and

   (c) compound statements including functions, `try`, `catch`, other `{`blocks`}`.

   (Braces, if present, are not considered part of the body they surround; see below.)

5. Two, three, four, or five spaces are acceptable indentations, but use the same amount consistently throughout the code. Nested constructs call for indentation in multiples of the basic indentation amount.
6. Short lines of code are easier to read than long lines. (That's why newspapers have multiple narrow columns of text, rather than a single full-width column.) Thus, lines should very rarely exceed 70 to 75 characters in length.
7. If a line of code is too long or too complicated to fit conveniently on a single line, break the line before an operator, if possible. Resume the continued statement after indenting the next line; the goal is to maximize readability.
8. If a parenthesized (or bracketed) expression spans multiple lines, consider aligning the closing parenthesis (bracket) with the matching opening parenthesis (bracket).
9. Always start a preprocessor directive with the leading `#` in column one. A continued preprocessor directive should be indented if syntactically possible.

2.2 Braces and control structures

Placement of braces is sometimes a controversial issue. The controversy is, however, stylistic rather than technical. We therefore set forth the following, first promulgated in [7], in the interest of having a common standard.

1. The opening brace that begins a block of code is placed on the same line as the structure that governs the block. Only a comment may appear following an opening brace.
2. The closing brace that ends a block of code is placed on a line by itself, except that a semicolon must follow on the same line if closing a `class` or `struct` definition. (Semicolons following closing braces in other contexts, notably at the end of a function, are illegal and must be removed.) Position the brace so as to align vertically with the start of the keyword introducing the control structure.
3. Align the keyword `else` with its matching `if`. (But see next item for a special case.)
4. When using nested `if`s to implement a multiple-alternative decision structure, write `else if` on a single line. Align each `else if` (and the trailing `else`) with the original `if` that began the decision structure.
5. Align the keywords `case` and `default` with the matching `switch`.
6. Align the keywords `public`, `protected`, and `private` with the matching `class` (or `struct`).
7. Align the keyword `catch` with the matching `try`.
8. Wherever possible, phrase predicates (conditions) affirmatively rather than negatively. Use `==` in preference to `!=`, and avoid the use of the `!` operator. Avoid especially the use of double negatives. Apply DeMorgan's rules, if needed, to achieve desired phrasing that is logically equivalent. (But see also the next item.)
9. If an `if` ... `else` construct has one clause much shorter than the other, phrase the predicate so as to place the shorter clause first, with the longer clause as the target of the `else`. This makes the short clause more obvious and less easy to overlook.
10. Consider using exceptions (`try`, `throw`, `catch`) as an alternative to deeply nested control structures, especially if the logic is testing for unusual circumstances.
11. If a label is used, enhance its visibility by outdenting it one level from the prevailing code.

3 Naming styles and conventions

Several categories of program entities require naming. In C++, these include:

- types (`class`es, `struct`s, `enum`s, `union`s, `typedef`s),
- data objects (`const`ants, variables, objects, data members, function arguments), and
- program units (functions, labels, `namespace`s, preprocessor macros, source files).

We promote the following rules for the selection and use of *identifiers* (names) denoting such entities.

1. Select names that accurately describe (in English) the intended usage, yet that are not overly burdensome to type and read.
2. Use abbreviations only if they are reasonably standard (among non-programmers, preferably). Don't just drp vwls; as a rule of thumb, a name that can't be pronounced was badly chosen.
3. Avoid names that are resemble or are otherwise similar to other names. For example, avoid using both singular and plural forms of a word. Also avoid using both abbreviated and unabbreviated forms of a word, as well as forms that differ only in capitalization.
4. The length of a name should be roughly proportional to its scope (larger scope, longer name) and inversely proportional to its frequency of use: in particular, use very short names only for heavily-used local entities.
5. Lower case letters (and occasional digits) are strongly preferred for most names of variables, arguments, functions, and members. If a name is composed of more than a single word, concatenate the words (without intervening underscores) and capitalize (the first letter of) each word after the first. If an acronym is part of a name, capitalize the acronym.
6. Type names (introduced by the keywords `class`, `struct`, `typedef`, `enum`, or `union`) begin with a capital letter but are otherwise consistent with the previous rule.
7. Use only capital letters (with perhaps an occasional underscore) for names of constants. This includes `enum` values, variables and objects declared `const`, and `#define`d preprocessor macro names.
8. Use an underscore as the leading character of a name that denotes a `protected` or `private` data member or member function.
9. Names should never contain a double underscore (`__`); such names are reserved for use by compilers, libraries, etc.
10. Each global name that is visible to the linker must begin with a prefix that is unique to that library or package in which it is declared. This practice helps to avoid name collisions, and is a good idea even when `namespace`s are available (you may want packages to have finer granularity than `namespace`s).
11. Verbs are often useful as names for functions; names beginning with `is` are recommended for `bool`ean functions, variables, arguments, and members. Similarly, names beginning with `create` are good choices for functions that yield `new` objects that must later be `delete`d by the calling function. Accessor functions returning some quantity (whether calculated or from a data member) should be named for that quantity (without any prefix such as `get`). Mutator functions that store a value into data members should be named starting with `set`. Naming `const`ants with spelled-out versions of their values is a worthless practice.

12. The name of a source file declaring or defining a `class` should be named to match (including case) that `class`' name. Use the file name suffix `.hh` if it contains only declarations, the suffix `.icc` if the file contains definitions of `inline` functions, and the suffix `.cc` if the file contains `static` data members' and/or non-`inline` member functions definitions. Avoid, however, lengthy file names, as most system software (e.g., linkers, loaders, file systems) does not allow file names of arbitrary length. Use the file name suffix `.h` only if the file contents are acceptable to both C and C++ compilers.

4 Code documentation rules and patterns

Code documentation takes several forms. External to the code, we have user tutorial materials (intended to introduce novice users to the intent and proper use of the code), user reference materials (intended for use by more experienced users on an as-needed basis), and technical reference documents typically describing the overall design of the software (intended for use by the code's maintainers). Standards for external documentation are described in a separate document. Note, however, that each package must be accompanied by a `README` file. This file must contain:

1. the package's name and the name of the person responsible for the package,
2. a description of the package's purpose and intended usage,
3. a list of the `class`es and functionality provided by the package, and
4. a description of any dependencies external to the package.

Internally, code documentation is typically understood to mean *commentary* or *remarks*. While such information does, of course, serve to annotate the code, clarifying its intent, we must always keep in mind that the code itself is the ultimate documentation. Therefore, it is always paramount to "say what you mean, simply and clearly" [8, citing [10]]. This benefits not only a human reader, but also tends to allow optimum compilation efficiencies.

1. Follow C++ comment style by using `//` wherever possible. Reserve C-style comments (`/* ... */`) to place a remark when other code must follow on the same line.
2. In general, keep each remark reasonably short, yet informative. Do not duplicate information inherent in the code; instead, provide either high-level summary or additional indication of intent, depending on the context. (See below for details.) Note that difficulty in providing such remarks is typically a symptom that some aspect of the code is not yet fully understood.
3. Each source file must begin with a substantial comment that provides (in English) at least the following information:
   (a) the file's name;
   (b) a (one-line) description of the file's purpose;
   (c) a (multi-line) overview/abstract of the file's contents, including any references to relevant published literature;
   (d) copyright and/or related legal information, if appropriate;
   (e) a description, as needed, of such technical issues as file formats, storage structures, and usage conventions;
   (f) a table of contents of the functions to be found in the file; and

(g) a provenance (pedigree or history) giving the file's original author and date of authorship together with a list giving the date, editor's name, and brief statement as to the nature of each revision made to the file.

This is akin to the title page (including abstract) of a document.

4. Follow the instantiation of each variable or object with at least a brief remark as to that entity's intended purpose. Similarly annotate each `class` member, whether a data member or a member function. Such documentation is, however, no substitute for selecting appropriate names for these entities.

5. Each function declaration (whether member or non-member) should include a descriptive name for each parameter. Further, use the `const` attribute in function declarations:

   (a) for each parameter not subject to change (*side effect*) by the function,

   (b) for the function's result type when the resulting value may not be changed by the calling function, and

   (c) for all accessors and other member functions that do not change (any part of) their invoking object.

   In addition to their technical merit (e.g., hints for optimization of compiled code), these practices aid in documenting a function's usage and behavior: a reader must assume anything not declared `const` is `mutable`.

6. Precede each function definition and `class` declaration with one or more lines of comment that describe its intended purpose, usage, behavior, assumptions, etc. Provide similar commentary ahead of other logical units of code (lines of code that, as a whole, perform a single identifiable task). This narrative should have the flavor of an extended section or subsection heading. In the case of a function, it should include any necessary preconditions and postconditions, even if these are also coded via, say, the `assert` macro.

7. Annotate each individual line of code that has a non-obvious intent. Such a comment is much like a sidebar explanation or marginal note, and thus is best placed following (on the same line as) the statement. Precede the remark by one or more tabs, if space permits.

8. Make a practice of commenting:

   (a) when a `case` of a `switch` has no `break` and is thus allowed to *fall through* to the next `case`,

   (b) following the final `}` of a function definition (give the function's name),

   (c) following the final `};` of a `class` (or `struct`) definition (give the `class`' name),

   (d) following the `#endif` matching an `#ifdef` (give the defined symbol's name), and

   (e) whenever you find a need to deviate from any of the rules prescribed herein.

5 Code management rules, tools, and procedures

5.1 Code management and management tools

As recommended by the Run II Configuration Management Working Group (Don Petravick, chair) at Fermilab, we will use CVS as our primary code management tool, and will organize files according to the directory structures following the model of BaBar. Distribution of

code will be handled by the UPS/UPD software developed at Fermilab. These practices are described in detail in [4].

Use only compilers from the list recommended by the C++ Languages and Tools Working Group (Mark Fischler and Marc Paterno, chairs. Always compile with all possible warnings enabled, and always use the linker and libraries appropriate to your compiler, platform, and environment.

5.2 Code development procedures

1. The primary source of performance efficiency lies in the selection of an appropriate algorithm. Therefore, design before you code.
2. Hand-optimize code only if you **know** that you have a performance problem, and after you have instrumented your code and **know** where the problem lies.
3. An individual file should contain a group of related routines (e.g., `class`es and functions) such that if a program needs one of them, it normally needs all of them. Many compilers treat one file as one compilation unit and so linkers often pull in the code for everything in that file if anything in the file is used (*all-or-none linking*, a common cause of *code bloat*). "One `class` per file" should be the goal except for `class`es that are heavily interdependent. (These are often `friend class`es; see below.)
4. Avoid cyclic dependencies between components. Either break the dependency (preferred), or else put both into a single component.
5. Put a class together with all of its `friend`s and functions into the same component.
6. A large component can often be broken up into several files to improve compiler and linker efficiency, and to reduce code bloat.
7. Within a source file, organize code in the following order:
   (a) Introductory comments.
   (b) `#include` directives.
   (c) Global `const`s.
   (d) Global types (`class`, `struct`, `typedef`, `enum`, `union`).
   (e) Global variables, preferably declared `static`.
   (f) Functions, then member functions, in order of declaration.
   (g) Test code (see below).
8. Each component must have a test harness that exercises that component's functionality. This program is to reside in the component's `.cc` file, and is to be surrounded by `#ifdef TEST` ... `#endif // TEST`. This test program is to be kept up to date to enable both regression testing (to assure changed or added code has not broken existing code) as well as confidence testing (to assure that the new or revised code works properly). This requirement implies that each component must be fully compilable and executable.
9. Object composition is preferable to inheritance.

6 Explicit coding rules and guidelines

Many of the coding practices described below are not confined to a single application. For the sake of exposition, however, they are placed in the section to which they most readily apply.

6.1 Preprocessor use

1. Every `.cc` file should `#include` the file `"fpcl.hh"`. This provides a means of handling compiler differences, etc. as the first substantive (non-comment, non-blank) line of code.

2. The `.cc` file of every component should `#include` the corresponding `.hh` file as the second substantive line of code.

3. An `.icc` file should `#include` no other files.

4. Always place a unique and predictable internal include guard around the contents of each `.hh` file. We recommend the following sequence:

```
// introductory comments here

#ifndef FILENAME_HH
#define FILENAME_HH

// complete contents of "filename.hh" here

#endif  // FILENAME_HH
```

   This will protect against including the same file multiple times.

5. If preprocessor efficiency (and hence time needed to compile) is a concern, place a redundant external include guard around each preprocessor `#include` directive in each header file:

```
#ifndef FILENAME_HH
#include "filename.hh"
#endif
```

   This practice will prevent the preprocessor from even opening a file if the file has already been previously included.

6. Put `inline` functions into a separate `.icc` file which is then `#include`d into the `.hh` file.

7. Do not use the `#define` directive to name constants. Use `enum` (preferred) or `const` instead; both alternatives are type-safe, while preprocessor use in this regard is not.

8. Avoid preprocessor macros. Wherever possible, use (`inline`) functions instead.

6.2 Scope and linkage

1. Avoid data with external linkage at file scope. A definition placed outside any blocks or classes in a file is at *file scope*, and is therefore visible to the linker unless it is declared `static`. Such *global* data tends to reduce modularity and *locality of reference*. This, in turn, often inhibits or reduces compiler code optimization.

2. Similarly, avoid (global) functions, `enum`s, and `typedef`s at file scope; use `class` scope instead (see next rule, for example).

3. Global data is often best implemented as `static` data members in a `class`' `private` section. This `class` will make these data members' values available via `public` `static` accessor functions.

6.3 Declarations and definitions

1. If an entity is defined in a `.cc` file, it should be declared in the corresponding `.hh` file. This rule (in conjunction with the next, related, rule) makes it possible for compilation dependencies to be obtained from the list of `#include`d header files.

2. Obtain the declaration for an entity (e.g., a function) by `#include`-ing the `.hh` file of the component in which the entity is declared.

3. Use the same (meaningful!) argument names in a function's prototype (declaration) as in the function's definition. (It is perhaps surprising that this is not required by the C++ language.)

4. When using a `class`, declare the `class` locally, e.g.,

   ```
   class myClass;
   ```

   (This is a *forward declaration*.) Only `#include` the `class`' definition if required by the compiler. Often, the compiler does not need to know the details of a particular class[1], and so `#include`-ing the header file would introduce an unnecessary compilation dependency between components. (As a rule of thumb, try to compile with only the forward declaration, and `#include` the definition only if the compiler complains.)

5. Do not define functions or member functions in `.hh` files. Instead, place `inline` definitions in `.icc` files and other function definitions in `.cc` files.

6. Group related declarations via an appropriate `class` or `struct`.

7. Avoid nested `class` declarations; most compilers do not yet correctly support this language feature.

8. Avoid global `enum`erations; wherever possible, make `enum`erations local to a `class`. This helps to avoid cluttering the global `namespace`.

6.4 Statements and expressions

1. Use `break`, `continue`, `return`, or `throw` in preference to `goto`. Further, use these in such as way as to produce the simplest control structures that reflect your logic.

2. Select a looping construct (`do`, `for`, or `while`) based on your specific use of the loop.

3. Use the `bool` type for truth values and for expressions yielding truth values.

4. Always test pointer values against `0`; avoid such statements as:

   ```
   if ( ptr )  {
     ...
   }
   ```

5. Use casts only of the form `Type(variable)`.

6. Avoid using the functions from `<stdio.h>`; use `<iostream.h>` functionality instead.

7. Do not define variables in a `for` statement; some compilers do not produce the correct scoping. Define a loop counter, for example, immediately before the `for` statement.

8. Each `switch` statement must have a `default` case that handles the unexpected.

[1] Use of pointers and references, for example, do not need the full definition, while creating or destroying an object, invoking a member function, or inheriting do require the full `class` definition.

6.5 Functions and parameters

1. Be sure that `main` is declared with an `int` result type, and that `main` actually `return`s an appropriate value. The most common result values are known as `EXIT_SUCCESS` and `EXIT_FAILURE`; these are made available via `#include <cstdlib>`.

2. Several relatively short functions are preferable to a long and complicated function. Not only are long functions harder to design, code, debug, and maintain (a 10-way decision structure may need 210 test cases), it may be very difficult to recover from an exceptional circumstance detected near the end of the function.

3. Only functions with a `void` return type may use the simple `return;` statement. All other functions must `return` an appropriate value of the matching type. (This problem often arises in connection with a *stub*: a function whose body is still empty, waiting to be written.)

4. Use `inline` functions and member functions only when their bodies consist of three or fewer statements. Such functions are ideal when used as accessor methods.

5. When side effects are not desired, pass parameters by value if they are small variables or objects, and by constant reference otherwise. This is especially important for copy constructors and assignment operators, where side effects are never appropriate. Pointer parameters are occasionally useful in conjunction with heap (dynamic) objects, but should be avoided for stack (`auto`matic) objects.

6. Do not write functions requiring many parameters. Such functions are hard to read, use, document, and maintain. They are often symptomatic of poor design.

7. Be careful in declaring default parameters. If you accidentally omit a parameter when the function is called, the default value will usually be supplied instead. This can be hard to debug.

8. Avoid passing pointers as parameters. Use references instead, as these are more convenient (e.g., dereferencing is implicit). In addition, the possibility of a null pointer will typically require that the function perform an existence test; no such test is needed when a reference is passed, as the referenced object is guaranteed to exist.

9. Avoid the use of `...` (ellipsis notation for unspecified parameters). Unspecified parameters are not *type-safe*, for the compiler can't check what wasn't declared.

10. A function's result should be returned via the `return` statement, not via a side effect on a parameter. In fact, avoid designing functions that operate via side effects.

11. A `public` function may never `return` a pointer (or reference) to a local variable. Because all local data is destroyed upon exit from a function, this practice guarantees an instantaneous dangling pointer (or reference).

12. Wherever possible, functions should `return` references to previously-instantiated objects, rather than `return`ing newly-instantiated temporary objects that will merely be copied back to the calling function and then immediately destroyed.

13. Do not give a member function both the attributes `inline` and `virtual`; compiler technology to date seldom correctly implements this combination. Similarly, avoid `virtual` member functions in —verb—template—s.

14. Always check the error codes resulting from calling a library function.

6.6 Constants, objects, pointers, types, ...

1. Avoid the use of magic numbers amid the code; use `enum` (preferred) or `const` instead.
2. Integer variables which never take on negative values should be declared `unsigned`.
3. Define each automatic variable in its own statement.
4. Define each variable (object) with the smallest scope it needs. (See also next item.)
5. The scope of a pointer to an `auto`matic object may not be wider than the scope of the object it points to. Otherwise, it is possible for the object to be destroyed while the pointer still points to it, a *dangling pointer*.
6. Initialize each variable (object) when it is instantiated so that it is in a valid state, ready to be used. Avoid functions (other than constructors) that must be called to initialize a variable. Similarly, avoid functions (other than destructors) that must be called to clean up a variable after its use.
7. Refer to the null pointer as `0`.
8. Consider polymorphism as an alternative to use of `union`.

6.7 Memory management

1. Whenever possible, use automatic storage (i.e., on the run-time stack) rather than dynamic (heap) storage.
2. Avoid calling `malloc`, `realloc` or `free`; use C++ `new` and —delete— instead.
3. Every call of `new` must be matched by exactly one call of `delete`, preferably in the same function. Don't expect some other function to do this for you. (See also the next rule.)
4. A function must not use a pointer parameter to `delete` an object as this will likely leave a dangling pointer in the calling function.
5. Immediately assign a new value to a pointer as soon as the object it points to has been `delete`d.

6.8 `classes`

1. Avoid `public` and `protected` data members. Supply `protected` accessor member functions for use by derived classes; otherwise furnish `public` member functions exclusively. This practice implements the important *information-hiding* principle.
2. Within a `class`, declare `friend`s first, then `public` entities, then `protected` entities, and finally `private` entities. Within each section, place any `enum`s ahead of data members, and data members before member functions.
3. Declare and define member functions in the following order whenever possible: constructors, destructor, accessors, mutators, overloaded operators, other member functions.
4. Use `friend`s sparingly. Large numbers of `friend`s suggest that the design's modularity is poor.
5. Use `public` inheritance whenever possible; `protected` and `private` inheritance are discouraged in the absence of strong justification for their use.

6. Avoid exposing any pointer to a `class`' internal representation. For this reason, a `public` member function may never return a non-`const` pointer or reference to member data.

7. Do not initialize a `const` data member within a `class`' definition; this is a syntax error. A separate definition, placed within the `.cc` file, is required and should be used for initialization.

6.9 Constructors and destructors

1. Each constructor must build a fully functional object, not a partially initialized object that needs further work (e.g., calling `open()`) before it becomes useable.

2. Provide a null constructor and a copy constructor, but only if they are consistent with the intent of the `class`.

3. Whenever possible, use an initializer list in a constructor rather than assignment operations. Objects that are declared `const` can't appear on the left hand side of an assignment. Further, assignment often causes instantiation of an ephemeral object on the right hand side in order that the assignment operator may copy it. Use of initializer list syntax overcomes both.

4. Within an initializer list, arrange the individual initializers in the order in which they will be carried out. C++ specifies a particular sequence of initialization; all base classes followed by all data members, both in the order of their appearance in the `class`' definition. Since this initialization sequence does not depend on the initialization list's physical ordering, it is best to arrange the initializers in the order they will be executed.

5. A `class` that has a data member of pointer type must declare both a copy constructor and an assignment operator. Without these, C++ will provide defaults that will do bitwise copies. These result in duplicate pointers to the same object, almost never what is wanted, as this nearly guarantees at least one of them will be a dangling pointer when the object is destroyed.

   These operations can be declared `private` and need not be defined, if so desired, in order to prevent their use.

6. Declare constructors `explicit` to prevent their implicit use as conversion (type case) operators.

7. A non-trivial destructor is required for every class that has one or more pointers as data members

8. Every `class` that is to be used as a base class must have a `virtual` destructor. This allows an object to be `delete`d properly even via a pointer to its base class. (See also next rule.)

9. Any non-trivial destructor must be declared `virtual`, unless derivation from its `class` is made impossible.

6.10 Overloaded operators and functions

1. An assignment operator must properly handle *self-assignment* (e.g., `x = x;`). Self-assignment is usually detected by comparing `this` to the address of the right hand side operand. If this circumstance is discovered, it is typical for the assignment operator to do nothing further but `return`.

2. An assignment operator must return a reference to its left operand, typically via `return *this;`. This will ensure that multiple assignment (such as `x = y = z;`) will work on `class` objects just as it does on built-in objects.

3. Overloaded operators should have reasonably conventional semantics. For example, operator+() should always resemble addition, albeit in some form appropriate to the `class`.

4. When overloading an operator, make sure all of that operator's usual and customary properties are preserved. For example, commutativity should always hold for all `+` and `*` operators, even when the types of its operands are not identical.

5. When overloading an operator, ensure that all variations of that operator are provided and are semantically consistent with each other. For example, overloading `+` strongly suggests that `+=` and both forms of `++` be overloaded to match so that a programmer can write any of the following:

```
a = a + 1;
a += 1;
++a;
a++;
```

   Similarly, if overloading `==`, it is also appropriate to overload `!=`.

6. Keep the number of user-defined conversions to a minimum.

7. An overloaded (non-destructor) function in a derived class is implicitly `virtual` whenever the corresponding function in the base `class` was declared `virtual`. However, this may be non-obvious to a reader who has not yet studied the corresponding base `class`, Therefore, explicitly declare an overloaded virtual function as `virtual`.

8. All overloaded functions sharing a common name should also share common semantics.

6.11 Exceptions

1. Use the exception mechanism (, ,) to handle unusual situations.

6.12 Portability considerations

1. Place machine-dependent code into a separate file for later ease of transition to a new computing environment.

2. Don't use pre-defined data types; define your own (perhaps via `typedef`) and use those instead. This makes it easier to change the types to accommodate the needs of a new platform.

3. Keep in mind the size guarantees for the various data types:

   short 8 bits, giving a range of -127 to +127;

   int 16 bits, giving -32768 to +32767; and

   long 32 bits, enough precision for all 9-digit (and some 10-digit) numbers.

   Also avoid the assumption that a `char` is either `signed` or `unsigned`, or that a pointer and an `int` are the same size.

4. Do not convert from a shorter type to a longer type.

5. Make no assumptions that objects may begin at arbitrary addresses.

6. Do not assume you know the order in which:
   - operands in an expression are evaluated,
   - static objects are initialized,
   - parameters are evaluated,
   - an object is initialized by its constructor.
7. Do not assume you know the lifetime of a temporary object.
8. Do not use shift operators in place of arithmetic operators.
9. Avoid pointer arithmetic.

6.13 Miscellaneous

1. When iterating, use inclusive lower limits but exclusive upper limits:

```
\begin{quote}
for ( k = 0;  k < N;  ++k )  {
   ...
}
\end{quote}
```

   This convention will avoid several difficult coding errors, since
   - the limits will be equal if and only if the number of iterations is zero, and
   - the number of iterations is the difference of the limits (assuming the step size is one).
2. Use `size_t` as the type for array indices, not `int` or `unsigned int`.
3. Never cast a `const` to a non-`const`.

Glossary

code bloat A phenomenon in which the compiled size of a program increases beyond logically necessary bounds. Often caused by indiscriminate use of `inline` or `templates`, or by poor linking strategy.

component A `.hh` file together with the corresponding `.cc` and `.icc` files.

declaration A means of introducing a name into a program.[2] Provides a description of the named entity, but does not create it. Does not result in any memory allocation. A declaration may be repeated many times.

[2] A *declaration* is also a *definition* unless:
- it declares a function without specifying its body, or
- it specifies `extern` and has no initializer or function body, or
- it declares a `static` class data member within a `class` definition, or
- it is a `class` name declaration, or
- it is a `typedef` declaration.

definition A unique description of an entity.[3] Often results in memory allocation for an instance of the entity. There must be exactly one definition of an entity.

external linkage A name that is known to the linker is said to have *external linkage* because it can interact with other translation units at link time.

file scope The scope outside of all blocks and classes in a file.

flow of control The order in which functions of a program, statements within a function, and operators within a statement are carried out.

global identifier A name (a) with external linkage or (b) at file scope or (c) both. Such a name is potentially useable anywhere in the program.

internal linkage A name local to its translation unit is said to have *internal linkage* because it is not known to the linker and so can not collide with names in other translation units.

package A collection of components organized as a physically cohesive unit.

translation unit A `.cc` file after all `.hh` files have been included by the preprocessor.

side effect A non-local effect, typically a change to a parameter or to a global variable, committed by a function.

References

[1] C++ Languages and Tools Working Group (Mark Fischler and Marc Paterno, chairs). "Physical Design Rules." Fermilab, 1996.

[2] Fisher, S. M. and L. Tuura. "C++ Coding Standards for ATLAS." Revision 1.4, Aug. 28, 1996.

[3] Greenlee, Herb, *et al.* "Draft for the D0 C++ Language Coding Guidelines." Fermilab, undated.

[4] Harris, Robert M., ed. and the Run II Configuration Management Working Group (Don Petravick, chair). "A Unix Based Software Management System." Fermilab, Computing Note GU0013, 1996. Also available as D0 Note 3118 and as CDF Note CDF/DOC/COMP_UPG/PUBLIC/3891.

[5] Henricson, Mats and Erik Nyquist. "Programming in C++, Rules and Recommendations." Ellemtel Telecommunication Systems Laboratories (Älvsjö, Sweden), Document No. M 90 0118 Uen, 1992.

[6] Jacobsen, Bob. "Reconstruction Design Checklist." BaBar, Nov. 7, 1996.

[7] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[8] Kernighan, Brian W. and P. J. Plauger. *The Elements of Programming Style*. McGraw Hill, 1974.

[9] Lakos, John. *Large-scale C++ Software Design*. Addison-Wesley, 1996.

[10] Strunk, Jr., William and E. B. White. *The Elements of Style*. Macmillan, 1959.

[3] A *definition* is also a *declaration* unless:

- it defines a `static` data member; or
- it defines a member function of a class, and is not inside the body of code that defines that class.