LESS Software Engineering

Mike Dahlin (editor)

May 15, 2000

> There are two ways of constructing a software design: one way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. *C.A.R. Hoare, "The Emperor's Old Clothes," CACM Feb 1981.*

This document describes the detailed programming style for all code produced by the LESS group. Much of this document has been lifted verbatim from the *Glunix Programming Style Document* by Doug Ghormley and Amin Vahdat, John Outsterhout's *Sprite Engineering Manual*, Eric Brewer's *Tools and Conventions for CS169: Software Engineering*, and Tom Anderson's *A Quick Introduction to C++*.

Most of this document deals with low-level coding standards. However, we first survey some high-level strategies needed for effective software engineering.

1 Write Solid Code

1.1 It's about attitude

Being a productive software engineer has more to do with attitude than anything else. Demand of yourself that your code be bug free. Be *embarrassed* when someone else finds a bug in your code — you've wasted their time. Don't work on a new feature until the old ones are done (done means bug-free).

Perhaps by cutting corners, you can quickly slap together some code that is "90% done." However, you will then invariably spend many days (or weeks or months) tracking down the many bugs scattered throughout your program. Not only does this approach actually take longer from beginning to end than a more disciplined strategy, it is decidedly less pleasant. After spending a few fun weeks of apparent progress as you add new feature after new feature, you spend months banging your head against bug after bug with little evidence that you are making progress.

Writing Solid Code by Steve Maguire is required reading for all LESS software developers. You should read it within 6 months of joining the team; it is short and well written — you can read it in a few afternoons.

The rest of this section touches on some key strategies to support a solid code attitude.

1.2 Find bugs as soon after you write them as possible

The longer you wait to find a bug, the longer it will take to diagnose it and fix it. The following list suggests a range of opportunities to find bugs starting from when the bug is in introduced and going forward in time (and upward in cost).

1. Design the system so that it prevents you from introducing bugs.

   The cheapest way to fix a bug is to not write it.

2. Make full use of compiler warnings (or lint).

   These tools allow you to detect bugs before you run the program. Furthermore these tools work automatically, requiring very little thought or effort on your part.

3. Read your code.

   Just as you wouldn't turn in a prose conference paper or English paper without proofreading it, you shouldn't run "first draft" code that you have not read. You will spot a surprising number of bugs this way.

If you have an IDE, consider making it a habit to step through every line of code you write right after you write it.

4. Make use of run-time checks to automatically find bugs.

   Use `assert()`*s liberally.* `Assert()`s are a terrific way to document your assumptions about the code and to document your understanding of how the code is supposed to behave. Put as many `assert()`s into your code as you can. The benefit of adding an `assert()` is that it might automatically find a bug that you would otherwise have to spend effort tracking down. There is almost no down side; in the unlikely event that your `assert()`s significantly slow you down, they can always be turned off when you compile the production code.

   When someone asked one of the developers of the Microsoft Tiger multimedia server how fast it was, he replied "It executes 42,000 asserts a second." This is an example of a good attitude.

   Use purify or some other run-time memory checker. Purify can automatically detect many common pointer errors, including crossing array bounds, accessing freed memory, and memory leaks. These bugs can be a nightmare to track down by hand, but purify finds them automatically. This is a no brainer.

   Use run-time sanity checks. This rule is simply an extension of the `assert()` idea. Guard your assumptions and data structure integrity by testing them as the program runs. Even if this slows down your program a little, computer time is much cheaper than programmer time.

5. Use module tests and regression tests.

   Module tests test the behavior of a module of your system as a black box; regression tests do the same for the whole system. Black-box testing is not as nice as the earlier options — it can tell you that something is wrong, but it doesn't point you to the line of the code that is causing problems. It is still necessary so that you can systematically find bugs that span modules or that your internal tests don't catch.

   You should use module tests to test as you go. This is a hard discipline to follow when a deadline looms; you will often feel you barely have time to write the functional code let alone the testing code. This is a false laziness. Testing as you go almost always takes less time than trying to debug a big mass of code.

6. Other programmers on your team find your bugs.

   This is a huge waste of time for everyone. Think of when this happens from their point of view: they add a feature, test it, and find that there is a bug. They then spend hours or days banging their head into the wall trying to figure out what they just broke. Finally, they decide that the bug was already there and have to search through all of the systems code to figure out what was wrong. This takes much longer in total time than if the person that put in the bug had found it. Futhermore it is really unfair – they pay for your mistakes.

   As more programmers get involved, the inefficiencies mount. Now, instead of finding each bug once (when it is written), we have to find the bug as many as n times (as each developer hits it.) If developers can't trust their colleagues to do regression tests before checking in code, then the tests need to be done n times for each update (by each person checking out the update) instead of once (by the person checking in the update.

7. *Worst:* Find the bug during production use of the system.

   There are many variations, all of them bad. At best, you might notice an anomalous output and spend time tracking it down. It gets worse from there. In a multi-developer project, your bugs can waste the other developers' time because they can't know whether the bugs they observe start in their code or yours. In a technology demonstration that you ship to users, users might discount your ideas because your implementation is bad. In a simulation study, your bugs could cause you to publish incorrect results at a conference.

1.3 Simplicity

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it. *Alan J. Perlis "Epigrams in Programming"*

If you are going to build a complex system, your only hope for success is to make it as simple as possible. There is a bias against simplicity; some people seem to like to show off their intelligence by building complex systems that only they can understand. If, however, you want to build solid, bug-free, maintainable code with minimal effort, simplicity should be your guiding principle.

1. Use creativity to simplify, not to show off.
2. Avoid complexity. Design then code.
3. Be extremist. Maximize simplicity. Systems always turn out to be more complex than you expect. Guard against this by pushing for the simplest possible design.

1.3.1 Simplicity vs. Performance

Usually, design decisions that encourage simplicity and correctness are superior to design decisions that emphasize program execution speed. Computer cycles are cheap and getting cheaper; programmer hours are not. Unfortunately, most introductory computer science courses emphasize speed over simplicity: "What is the fastest algorithm to do X?" rather than "What is the simplest way to do X?" and this bias can be hard for programmers to overcome. Keep in mind Ousterhout's rule of performance tuning:

> The biggest speedup in a system is when it goes from "not working" to "working." That is an infinite speedup. *John Ousterhout*

Finally, remember that the 90/10 rule applies to performance tuning. It is probably better to get a system running and then study it with performance tools to focus your effort on the 10% of the code that matters to performance than to try to squeeze every cycle out of every line of code you write. Furthermore, a simple system will probably be easier to systematically attack with performance tools than a complex system rife with premature "peephole" optimizations.

1.4 Go from working system to working system

You should use an incremental strategy in which you add a feature and debug it and only move to the next feature when the previous features are tested and bug-free.

One of the most common causes of disasters in both software and hardware projects is building all of the features first and then trying to pull them all together and debug them. In addition to probably taking longer than an incremental approach (because bug repair happens long after bug introduction), this approach is also very high risk. Whereas schedule slippage under the incremental system means that you will have a working system with not quite the full feature set when the deadline arrives, slippage under the other strategy means that you will have nothing that works when the deadline arrives.

Another advantage of the incremental approach is that it makes it easier to track progress. For example, if I am working on a project that has 10 subtasks of equal complexity, and I finished 2 last week and 3 this week, I can guess that I'll be done in 2 or 3 more weeks. In contrast, with most projects it is hard to predict how long "tracking down the last few bugs" will take.

1.4.1 When you observe a bug, that is the right time to dive in and fix it.

> When you discover a fluke, first determine if a whale is attached to it. *Dennis O'Connor (Intel i960(R) Microprocessor Division)*

It is always tempting when you see a little anomaly on the output of a run to convince yourself that it is a small problem that you can come back and fix later. There are three problems with that attitude. First, it might not be just a small problem (there may be a whale attached to the fluke). Second, the accumulation of these small problems eventually makes your system incomprehensible as bugs interfere with other bugs. It is hard to tell when you have successfully fixed a bug, and conversely, it is hard to tell when you have introduced a new bug. Third, as you add other stuff to the system, the anomaly may disappear. Bugs, however, don't fix themselves. In such a case you have traded a relatively easy problem ("What is causing behavior X?") for a hard one ("There's a bug in there somewhere, I just have to find it.")

1.5 Reflect

When you find a bug that was not detected automatically by your tools and tests, something went wrong with your development methodology. Think about the bug and how to avoid or automatically detect such bugs in the future.

2 File Structure

There are a number of regions that normally appear at the top of source files before any procedures are declared. Those regions are:

- Including `.h` files.
- Declaring `extern` variables.
- Declaring macros or `#define` constants. *However, macros and* `#define` *should almost never be used. See below "The C Preprocessor is Evil."*
- Declaring types, structs, unions, enums, etc.
- Declaring global and static variables.

These regions should (ideally) appear in the above order, though any readable ordering is fine. When in doubt, though, use this ordering. Between each of these regions, and at the end of the file, there should be the following single comment to help give visual clues as to the structure of the file:

Header files (`.h` files) should generally follow a consistent order. System and library files (e.g. <`stdio.h`>) should precede local header files (e.g. ``` ``FiniteCache.h'' ```. Within each group of header files, files should be listed alphabetically unless a dependency precludes that order.

```
/***********************************************************************/
```

Following these regions should be:

1. Initialization routines for this module
2. Cleanup routines for this module
3. All other procedures for the module

2.1 Nested include files

It is legal and desirable for one include file to `#include` another. For example, the exported include file for a module should `#include` any additional files needed to use the module. With this approach, clients need only `#include` the exported include files of the modules they use.

Use `#ifdefs` to make sure that each include file is processed only once, even if it is included multiple times. For example, the file `FiniteCache.h` should begin:

```
#ifndef _FiniteCache_h_
#define _FiniteCache_h_

... body of FiniteCache.h ...

#endif _FiniteCache_h_
```

3 Code Standards

Every programmer should strive to write code whose behavior is immediately obvious to the reader. If you find yourself writing code that would require someone reading through it to thumb through a manual in order to understand it, you are almost certainly being way too subtle. There is probably a much simpler and more obvious way to accomplish the same end. Maybe the code will be a little longer that way, but in the real world, it's whether the code works and how simple it is for someone to modify that matters a whole lot more than how many characters you had to type.

3.1 Minimize scope

One of the primary tools for producing maintainable code is minimization of scope. All variables should have scopes consistent with their lifetime and intended use.

There should be almost no global functions or constants.

3.2 Types

Use classes or typedefs instead of raw types like `int`. If you are using an `int` (or some other primitive type) to represent some concept, such as dollars, hide the representation via a class or typedef:

```
typedef int Dollars;
```

This allows you to change the representation later on. Without this typedef, you would have to determine which ints are actually Dollars and convert them by hand. Using the typedefs not only adds flexibility, it also improves the documentation.

When should you use a class for Dollars rather than a typedef? Typedefs are just synonyms: if you also typedef Yen to be an `int`, then Dollars and Yen are indistinguishable to the compiler. If you use classes, then you can create constructors that *convert* Dollars to Yen (and vice versa).

3.3 Classes

This section deals with using classes as abstract data types. Conventions for inheritance are discussed later.

1. **Only functions should be public members.** With few exceptions, all public members of a class should be functions. Public data

members (like `int x;`) prevent the implementor from changing the representation down the road.

2. **Never return a pointer or reference to private data.** This is a corollary to the previous rule. Returning pointers to private data provides full-time access to that data, even though the invariants might only hold during function invocation. This is worse than explicit public data members because the access is indirect.

3. **All of the types and members of a class should be explicitly declared `public`, `protected`, or `private`.** They should be grouped together and declared in that order. Note that `protected` only makes sense for base classes that support inheritance.

4. **All classes must explicitly have a copy constructor, an assignment operator, at least one non-copy constructor, and a destructor.**

   The copy constructor constructs a T by copying its argument, which is also a T. In general, any class with private pointers should not use the default copy constructor or assignment operator. The default versions only copy the value of the pointer rather than the contents to which it points. Needless to say, this can be a painful source of bugs. *Requiring* all classes to have an explicit copy constructor and assignment operator reminds the writer to check for pointers and explicitly copy their contents.

   If you know the default version of any of these is sufficient, then indicate this by explicit comments for each one:

   ```
   // operator=  use the default assignment operator
   ```

   It is also acceptable to prohibit assignment or copying. This is done by declaring the function private and creating a stub function that calls `assert(0);` (or otherwise exits with an error) if it is called. Making the function private allows the compiler to prevent clients from calling the routine. Since the compiler can't prevent other member functions from making the call, we must also include the `assert`-ing stub function to fully deactivate the call.

5. **The copy constructor for class `T` should take an argument of type `const T&`.**

6. **The assignment operator for class `T` should have the following signature:**

   ```
   T& T::operator=(const T& rhs);
   ```

   It is important to return `T&` so that assignments can be chained (`a = b = c`). Declaring the argument as `const` allows the compiler to verify that you don't update the right-hand side and may also lead to better optimization. There may be additional (overloaded) versions of the assignment operator that have a different argument type.

7. **The assignment operator must return *this.** This rule ensures that assignments can be chained together correctly.

8. **The first step of every assignment operator should be:**

   ```
   if(this == &rhs){
     return *this;
   }
   ```

   This simple rule ensures that `x = x` works correctly. Leaving it out can be painful. For example, if the class has a dynamically allocated array, the operator will normally delete the storage for the left hand side, allocate storage for a copy of the right hand side, and then copy the array. If the two sides are the same and this case is not detected, the array for the right hand side is deleted before it is copied.

9. **All fields of a class must be assigned (or copied) in the order of declaration.** This is tricky to ensure since it is relatively easy to add fields to a class and then forget to check the assignment operator and the copy constructor. The only thing that works is discipline: *if you add a field, check all of the methods.*

   For methods that must touch all fields of the class (assignments, comparisons, etc.), I find it is useful to begin the method by `assert`-ing that the `sizeof(TheClass)` equals the sum of the `sizeof()` values for all of the fields. If I add a field without updating the method, the assertion will fail the first time I call it.

3.4 Procedures and methods

1. **Prototype all functions.** All functions exported from a module or exported to other files within the same module should be prototyped in a header file. Functions only used within a single file should be prototyped at the top of the file (do not rely on definition-before-use). All prototypes should list the function arguments in the prototype. If there are no arguments to the function, then the argument list should be identified as void in both the prototype and the procedure definition:

   ```
   void waitForSignal(void);

   void
   waitForSignal(void)
   {
   }
   ```

   Each procedure declaration will start with the return value on its own line, the prototype for the function on one (or more lines), and then an opening brace on its own line. If the procedure does not return a value, you must explicitly specify `void`:

   ```
   int
   main(int argc, char **argv)
   {
   }
   ```

2. **Declare returns explicitly.** Procedures should always have a return type explicitly declared. They should also always have explicit `return` statements, even if the return type is `void`.

3. **Declare arguments, methods, and procedures `const` whenever possible.** Declaring an argument or function as constant allows better optimization, documents the code, and allows the compiler to detect unintended modifications. Member functions that do not change an object should be declared constant by adding `const` after the closing function parenthesis but before the semicolon or left brace. Note that only `const` members can be invoked on `const` objects of the corresponding class.

4. **When passing an array, use array notation, not pointer notation.** E.g. `int foo[]` rather than `int *foo`. The latter hides the fact that there are multiple ints.

5. **Don't return pointers or references unless they are 1) `this`, 2) a passed in pointer or reference, or 3) a newly allocated object.**

3.5 Statements

1. **Individual C statements should be simple and straightforward.** The following is an example of what *not* to do:

```
      numEmptyBuffers =
        (curBufPtr->currentData + curBufPtr->incomingData) ? 0 :
         MAX_BUFFER_SIZE - (curBufPtr->waitingData +
         curBufPtr->incomingData + (curBufPtr->numBytesRequested -
curBufPtr->currentData));
```

   This could be improved by assigning the "else" portion of the `?` operator to a temporary variable with a meaningful name and using that variable in the above expression.

2. **`goto` statements are illegal.**

3. **Spurious sub-scopes (i.e., not the body of an `if`, `for`, etc.) should not generally be used.**

4. **The operators ++ and --- should not appear in complex statements, only as a command by itself.** The statement

```
    buf[ctr++] = x;                   // Illegal!
```

   should be written as

```
    buf[ctr] = x;
    ctr++;
```

5. **The body of a control statement is always enclosed in braces.** The following statements should have braces, even if there is only a single statement within the body: `if`, `else`, `do`, `while`, `for`, `switch`.

```
if(x < min){
  min = x;
}
```

For an empty `while` or `for` statement, do not use braces and put the semi-colon on the next line, indented by 2 spaces. There should never be a do-while loop with an empty body; in such a case, use a while loop.

```
for(x = 0; x < y; x++)
  ;
```

6. **Use parenthesis rather than rely on order of evaluations.** In complex comparisons involving OR's (`||`) and AND's (`&&`) together or using arithmetic or logical operations, you must use parens to make the groupings explicitly. This rule helps both correctness and readability (even if you know the order will come out okay, assume you're writing the code for someone who isn't as smart as you are.)

3.6 Constants

- **Use `const` as often as you can.** Be a `const` zealot. Declaring a variable or function as constant allows better optimization, documents the code, and allows the compiler to detect unintended modifications. Member functions that do not change an object should be declared constant by adding `const` after the closing function parenthesis but before the semicolon or left brace.

- **Prefer `const` to `#define`.**

  In C++ don't declare constant values by `#defining` them (see below: "The C Preprocessor is Evil".) Instead, declare `const` variables of the appropriate type so that the compiler can do type checking.

  ```
  #define BUF_SIZE 1024    // ERROR!
  ```

  Should be written:

  ```
  const int BUF_SIZE = 1024;
  ```

Due to compiler constraints, the exception to this rule is when the constant is to be used to specify the size of a static array.

- In C, you must use `#define` to specify a symbolic name to use to indicate the size of an array.
- In C++, if you are not using the Gnu compiler, you cannot set the value of a class's member variables outside of their constructors. To achieve the same effect, use `enum`s whose values are set explicitly.

  ```
  class AnArray{
  private:
    enum maxSize {MAX_SIZE = 1024};
    static char buffer[MAX_SIZE];
  };
  ```

  This syntax is more portable but less readable than the `const` syntax. If portability is a concern, you may wish to use this syntax even if you have no immediate plans to use a compiler other than gcc. If you are certain that you will not need to port the code, the `const` syntax may be better.

- **Prefer `enum` to `const`.** When creating a list of possible values, it is traditional to define a series of integer values for the different possibilities. It is better to use an enumeration so that the type system can help you.

```
//
// Compiler can't distinguish different ints.
// XXX Dont do this. XXX
//
const int DISK_STATE_SEEKING = 0;
const int DISK_STATE_READING = 1;
const int DISK_STATE_IDLE    = 2;
const int CPU_STATE_ACTIVE   = 0;
const int CPU_STATE_IDLE     = 1;
int state;
state = DISK_STATE_SEEKING;
if(state == CPU_STATE_IDLE){    // Error but no compiler warning
```

```
  ...
//
// Better.
//
enum disk_state{DISK_STATE_SEEKING,
                DISK_STATE_READING,
                DISK_STATE_IDLE};
enum cpu_state{ CPU_STATE_ACTIVE,
                CPU_STATE_IDLE};
enum disk_state state;
state = DISK_STATE_SEEKING;
if(state == CPU_STATE_IDLE){  // Compiler detects this error
  ...
```

Note that you can explicitly set the values for enumerated types if needed.

3.7 Dangerous language features

> If programming in Pascal is like being put in a straightjacket, then programming in C is like playing with knives, and programming in C++ is like juggling chainsaws. *Anonymous.*

Tom Anderson has written a document specifying a subset of C++ that can do most of what one wants to do while avoiding features that are dangerous or difficult to understand. See `http://http.cs.berkeley.edu/˜tea/c++example/` for his take.

3.8 C and C++ language features to use with caution

3.8.1 Inheritance

The downside of inheritance is that it tends to spread implementation details across multiple files; if you have a deep inheritance tree, it can take some serious digging to figure out what code actually executes when a member function is invoked.

The question to ask yourself before using inheritance is: what's your goal? Is it to write your programs with the fewest number of characters possible? If so, inheritance is really useful, but so is changing all of your function and variable names to be one letter long ("a", "b", ...).

When is it a good idea to use inheritance and when should it be avoided? A rule of thumb is to use inheritance only for representing *shared behavior* between objects and not to use it for representing *shared implementation*. In C++, you can use inheritance for both concepts, but only the first will lead to truly simpler implementations.

One way to control inheritance is to restrict yourself to using inheritance from only *abstract classes*. Abstract classes define interfaces to *pure virtual functions*, but they explicitly do not define implementations of those functions.

```
class CacheMissHandler{
public:
  virtual void add(Event *event) = 0;
  virtual Time predictMissTime(Id id) = 0;
};
```

Note that the pure virtual functions are defined to be 0.

We can now define subtypes of CacheMissHandler that simulate a local disk, an NFS server, and an HTTP server. These file systems would share a common interface but not share any implementation.

If there is a piece of functionality shared by a set of sibling classes, try defining that functionality in a separate class and instantiating that class in the siblings.

3.9 Templates

The principle problem with templates is that they can be difficult to debug. Templates are easy to use when they work, but finding a bug in them can be difficult. In part this is because current generation C++ debuggers don't really understand templates very well. Nevertheless, it is easier to debug a template than two nearly identical implementations that differ only in their types.

The best advice is — don't make a class into a template unless there really is a near term use for the template. And if you do need to implement a template, implement and debug a non-template version first. Once that is working, it won't be hard to convert it to a template.

3.10 C and C++ language features to avoid like the plague

C++ is a wildly over-complicated language, with a host of features that only very, very rarely find a legitimate use. It's not too far off the mark to say that C++ includes every programming language feature ever imagined and more. I'm sure that each feature has its advocates, but there are compelling reasons to avoid these features — they are easy to misuse, resulting in programs that are harder to understand. In most cases, the features are also redundant — there are other ways of accomplishing the same end. Why have two ways of accomplishing the same thing? Why not stick with the simpler one?

If, despite these warnings, you find yourself tempted to use these features, a careful code review is in order. Talking through the design with someone else might bring to light a better strategy.

3.10.1 The C preprocessor is evil

> It has been observed that almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer. *Bjarne Stroustrup* The C++ Programming Language *2nd Edition p. 138*

Don't use it.

1. **Macro functions are better done with inline functions.** The latter are type checked and evaluate their arguments exactly once.
2. **Use `const` variables rather than using `#define`.** Again — we get type checking from the compiler.

3. **Bit manipulation.** If you need bit operations —- usually a sign of premature optimization —- use the C++ bit field constructs rather than carefully constructed constants combined with bitwise operators (like &).

4. **Use templates rather than macros for polymorphic functions.** If you must use polymorphic functions, templates are better than macros.

There exist two permitted uses of the C preprocessor:

1. Conditional compilation to avoid reprocessing a `.h` file that is `#include`-ed multiple times.

2. `#define`-ing a value to indicate the size of a statically-allocated array in a C (not C++) program.

3.10.2 Multiple inheritance

It is possible in C++ to define a class as inheriting behavior from multiple classes (for instance, a dog is both an animal and a furry thing). But if programs using single inheritance can be difficult to untangle, programs with multiple inheritance can get really confusing.

3.10.3 Operator overloading

C++ lets you redefine the meanings of operators (such as + and >>) for class objects. This is dangerous at best, and when used in non-intuitive ways, a source of profound confusion, made worse by the fact that C++ does implicit type conversion, which can affect which operator is invoked.

Unfortunately, C++'s I/O facilities make heavy use of operator overloading and references, so you can't completely escape them, but think twice before you define '+' to mean "concatenate two strings."

3.10.4 Pointer arithmetic

Runaway pointers are a principle source of hard-to-find bugs in C programs, because the symptom of this happening can be mangled data structures in a completely different part of the program. Depending on exactly which objects are allocated on the heap in which order, pointer bugs can appear and disappear, seemingly at random. For example, `printf` sometimes allocates memory on the heap, which can change the addresses returned by all future calls to `new`. Thus, adding a `printf` can change things so that a pointer that used to (by happenstance) mangle a critical data structure now overwrites memory that may not even be used.

The best way to avoid runaway pointers is (no surprise) to be *very* careful when using pointers. Instead of iterating through an array with pointer arithmetic, use a separate index variable, and assert that the index is never larger than the size of the array. Optimizing compilers have gotten very good, so that the generated machine code is likely to be the same in either case.

3.10.5 Casts from integers to pointers and back

Don't do this.

3.10.6 Using bit shift in place of multiply or divide

This is a clarity issue. If you are doing arithmetic, use arithmetic operators; if you are doing bit manipulation, use bitwise operators. If I am trying to multiply by 8, which is easier to understand, `x << 3` or `x * 8`? In the 70's, when C was being developed, the former would yield more efficient machine code, but today's compilers generate the same code in both cases, so readability should be your primary concern.

4 Naming

Choosing names is one of the most important aspects of programming. Good names clarify the function of a program and reduce the need for other documentation. Poor names result in ambiguity, confusion, and error. This section gives some general principles to follow when choosing names, then lists specific rules for name syntax such as capitalization, and finally describes how we use prefixes to highlight the structure of the system.

1. **Names should have meaning.**

   When choosing names, play devil's advocate with yourself to see if there are ways that a name might be misinterpreted or confused. Here are some things to consider:

   (a) **Is someone who sees the name out of context likely to realize what it stands for, or could they easily confuse it for something else?** Ousterhout provides the following example: "Our procedure for doing byte-swapping and other reformatting was originally called `Swap_Buffer`. When I first saw that name, I assumed it had something to do with I/O buffer management, not reformatting. We subsequently changed the name to `Fmt_Convert`."

   (b) **Does the name look a lot like another name that is used for a different purpose in the same context?** For example, it is probably a mistake to have two variables named `proc` and `process`, both referring to processes in the same piece of code: it will be difficult for readers to remember which is which. Instead, add a bit more information to the names to distinguish them; for example: `masterProc` and `slaveProc`.

   (c) **Is the name so generic that it doesn't convey any information?** What can I expect to find in the variable `tmp`?

   (d) **Use the same name to refer to the same thing everywhere.** For example, you might use the name `filePtr` each time you have a pointer to an open file.

2. **Prefix globally-visible names with module abbreviations.** Anything globally-visible (procedures, variables, classes, structures, etc.)

must be prefixed by the module abbreviation, and an underscore.

`Idle_processList`	global variable
`Idle_getProcessList()`	global procedure
`struct Idle_ProcessEntry {...}`	global structure

Variables that are intra-module but which are visible to multiple files within that module should be treated as though they are global.

3. **Variables which are only visible within one file or procedure should omit the prefix.** Such variables should also be defined as `static` if they are at the global scoping level.

`static int localVariable;`	local variable
`static int localProc(void);`	local procedure

4. **Method names omit the prefix.** Their module is documented by looking at the class of the variable they are associated with.

5. **Variables, methods, and procedures begin with lower-case letters.**

6. **Classes, Modules, Types, and Structures begin with upper-case letters.**

7. **CONSTANT variables should be named with all caps.**

8. **Capitalize trailing words in multi-word names.** All multi-word names should have trailing words capitalized, but with no internal underscores except the one immediately following the module abbreviation. The exception to this rule is multi-word constants; separate their words with underscores.

5 Low-level Formatting Details

1. **Indentation is 2 spaces.** All editors we know of can be set to this setting.

2. **Each C statement will end with a semi-colon with *no* preceding space.**

3. **There should be *no* space separating the following keywords from the following open parenthesis or brace: `if`, `for`, `while`, `do`, `switch`, `case`.**

4. **Curly-braces don't normally stand alone.** Opening braces appear on the same line as the statement that is opening the scope. The closing brace appears on a line by itself, lined up with the statement which opened the scope.

   ```
   if(filePtr->length == 0){
     return -1;
   }
   ```

   The exception to this rule is procedure declarations: both the opening and closing brace should be on their own line flush left. For example:

   ```
   static void
   countToTen(void)
   {
     int  i;

     for(i = 1; i <= 10; i++){
       printf("%d\n", i);
     }
   }
   ```

5. **All comma-separated argument lists should have a single space after each comma in the list.**

6. **All assignment statements and comparisons should have a single space before and after the symbol(s).** Examples:

   ```
       x = 4;
       a |= b << 2;
       if(y < x){
   ```

7. **No line should be longer than 80 characters.**

6 Comments

The most important thing to remember in documenting your code is "quality, not quantity." A few carefully chosen words in the right place may be more helpful than a page of drivel. This section lists a few things to consider in order to improve the quality of your in-line documentation and then provides standard formats for different types of comments.

6.1 Document things with wide impact

The most important things to document are those that affect many different pieces of a program. Thus, it is essential that every procedure interface, every structure declaration, and every global variable be documented clearly. If you haven't documented one of these things, it will be necessary to look at all the uses of the thing in order to figure out how it's supposed to work. This will be tedious and error-prone.

On the other hand, things with only local impact may not need much documentation. For example, in short procedures I don't usually have comments explaining the local variables. If the overall function of the procedure has been explained, and if there isn't too much code in the procedure, and if the variables have meaningful names, then it will be easy to figure out how they are used.

6.2 Don't just repeat what's in the code.

The biggest mistake made in documentation is simply to repeat what's already obvious from the code, such as the trivial (but exasperatingly common) example:

```
//
// Increment i.
//
i ++;
```

Documentation should provide higher-level information about the overall function of the code — what a complex collection of statements really means. For example, the comment

```
//
```

```
// Probe into the hash table to see if the symbol exists.
//
```

Is likely to be more helpful than

```
//
// Mask off all but the lower 8 bits of x, then index into
// table t, then traverse the list looking for a character
// string identical to s.
//
```

6.3 Be creative.

Draw a picture of what's going on. Give an example of how the module/procedure is supposed to be used.

6.4 Document things in exactly one place.

Systems evolve over time. If something is documented in several places, it will be hard to keep the documentation up-to-date as the system changes.

For example, I put the documentation of each structure right next to the declaration for the structure, including the general rules for how it is to be used. I don't explain the fields of the structure again in the code that accesses the structure; people can always refer back to the structure declaration for this.

The other side of the coin is that every major decision needs to be documented *at least* once.

6.5 Comment formats

1. **Figure 1 shows the header comment that should be placed at the top of each file.**
2. **Figure 2 shows the trailer comment that should be placed at the end of each file.**

```
/*************************************************************************
 *
 * LESS Group
 *
 * filename.cc|h ---
 *
 *     Description of the purpose and function of this file.
 *
 * $Date: 1997/07/29 14:59:57 $ $Id: eng.tex,v 1.2 1997/07/29 14:59:57 dahlin Exp dahlin $
 *
 * Copyright (c) 1997 by <author list.>
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice appear in
 * all copies of this software.
 *
 *************************************************************************/
```

Figure 1: Standard file header.

```
/*************************************************************************
 *
 * Change History
 *
 * $Log: eng.tex,v $
 * Revision 1.2  1997/07/29 14:59:57  dahlin
 * Added a few more quotes.
 *
 *
 *
 *************************************************************************/
```

Figure 2: Standard file trailer.

```
/***************************************************************************
 * ProcedureName ---
 *     Description of purpose and function of the procedure (e.g. policy
 *     rather than mechanism).
 *
 * Arguments:
 *     type1 arg1 - a description of argument 1
 *     type2 arg2 - a description of argument 2
 *
 * Results:
 *     A description of the results returned by the function, either
 *     in the return value or in pass-by-reference arguments.
 *
 * Side effects:
 *     Any changes in the state of the program or its environment
 *     that may be visible to the program or the user.
 ***************************************************************************/
```

Figure 3: Comment format for procedures, functions, and method declarations.

3. **The preface comment format for all procedures, functions, and method declarations is shown in figure 3.**

4. **Code comments occupy full lines.** Comments used to document code (as opposed to declarations) should occupy full lines, rather than being tacked onto the ends of lines containing code. Tacked-on comments are hard to see. Use proper English in your comments (e.g., capitalize the first word of the comment, and structure your comments in sentences.)

   Full-line comments should be indented to the same level as the surrounding code. The comment lines should begin with lined up double slashes with a blank comment line before and after the comment text and a blank separator line before the comment.

```
headPtr = getHead();
```

```
//
// Move the hint out of the way so it won't get
// invalidated by the deletion.
//
if(first.lineIndex > 0){
     ...
```

If comments are indicated with the C syntax, the comment lines should begin with lined up stars, the open and close comment symbols appear on separate lines from the text, and a blank separator line appears before the comment.

```
/*
 * This is also a legal code comment.
 */
```

5. **Declarations comments are side-by-side.** When you document declarations for procedure arguments and structure members, place comments on the same lines as the declarations. Place the comments to the right of the declarations with all left edges of the comments lined up. When a comment requires more than one line, indent the additional lines to the same level as the first line, with close-comment characters (if using the C syntax) on the same line as the end of the text.

```
typedef struct Floater{
  File *filePtr;                  // File to which floater
                                  // belongs.
  Position *point;                /* Where it is located. */
} Floater;
```

7 Tools

Use tools to make your life easier.

7.1 Compiler

If you use g++, then all code should compile without warnings using the -Wall flag.

If you use cc, then all code should compile without warnings under lint.

7.2 Purify

Purify is a run-time tool to detect memory bugs — uninitialized memory, array out-of-bounds, reading freed memory, memory leaks, etc. All code should run without warnings under purify.

To compile with purify, add the following to your makefile:

```
###
### PURIFY
###
PURIFY=/lusr/bin/purify -cache_dir=./purify
```

Then, for the main compilation line (that links the objects into an executable), prepend `$(PURIFY)` to the command (before the `$(CC)` command name.

```
$(TARGET): $(OBJS)
        $(PURIFY) $(C++) $(C++FLAGS) $(LIBDIRS) $(OBJS) $(C++LIBS) -o $(TARGET)
```

You can deactivate purify by commenting out the `PURIFY = ...` line while leaving the compilation command unchanged.

7.2.1 Quantify

The Quantify tools is also shipped with Purify. You invoke it the same way you do with purify (except substitute "quantify" for "purify"). It provides a simple way to get performance data. Use this tool before you spend time optimizing your code for speed.

7.3 cvs

CVS, concurrent versions system, is a powerful tool for managing source code. It can integrate changes from many developers, can keep track of all old versions, and can give symbolic names to specific versions (e.g. "beta release"). All code developed for LESS should be checked into a CVS repository on a regular basis. CVS's documentation is relatively complete, but it is sometimes difficult for beginners. We will develop a tutorial in the near future.

7.4 Other recommended tools

If you're not familiar with these, it is worth your time to use them.

etags	Emacs support for navigating large source trees.
gdb	Debugger

At the current time we are also examining a promising gnu tool for automating regression testing.

We need a tool for tracking bugs; Calvin has one that might be good.

8 Acknowledgements and References

Much of this document has been lifted verbatim from the *Glunix Programming Style Document* by Doug Ghormley and Amin Vahdat, John Outsterhout's *Sprite Engineering Manual*, Eric Brewer's *Tools and Conventions for CS169: Software Engineering*, and Tom Anderson's *A Quick Introduction to C++*.

The following references provide more information:

C.A.R. Hoare, "The Emperor's Old Clothes." Communications of the ACM, Vol 24, No 2, Feb 1981, pp. 75–83.

Steve Maguire, *Writing Solid Code*, Microsoft Press.

Steve Maguire, *Debugging the Development Process*, Microsoft Press.

Steve McConnell, *Rapdi Development: Taming Wild Software Schedules*, Microsoft Press.

Scott Meyers, *Effective C++* and *More Effective C++*, Addison Wesley.

Bjarne Stroustrup, *The C++ Programming Language*