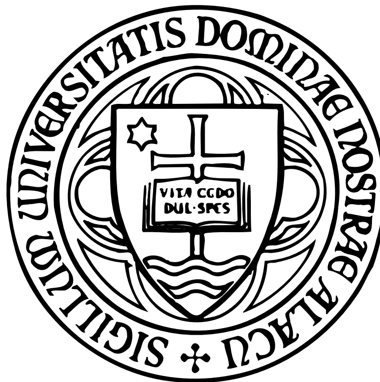
# The Laboratory for Scientific Computing (LSC):

# Coding Standards

http://www.lsc.nd.edu/

April 16, 2000




Jeremy G. Siek (jsiek@lsc.nd.edu)
Jeffrey M. Squyres (squyres@cse.nd.edu)
Andrew Lumsdaine (lums@lsc.nd.edu)

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556

Copyright ©1998, University of Notre Dame.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to process this file through TeX and/or LaTeX and print the results, provided the printed document carries copying permission notice identical to this one except for the removal of this paragraph (this paragraph not being relevant to the printed manual).

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "The GNU Manifesto", "Distribution" and "GNU General Public License" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the sections entitled "The GNU Manifesto", "Distribution" and "GNU General Public License" may be included in a translation approved by the Free Software Foundation instead of in the original English.

# Contents


**1 Introduction** **1**

**2 Application Organization** **2**
2.1 Development vs. Release . . . 2
2.2 Directory Structure . . . 2
2.3 File Name Conventions . . . 4
2.4 Program Files . . . 4
2.5 Module Files . . . 5
2.6 Header Files . . . 5
2.7 Documentation . . . 5
2.8 Configuration and Compilation . . . 5
2.8.1 Autoconf . . . 6
2.8.2 Makefiles . . . 6
2.8.3 Optimization . . . 6
2.9 Version Control . . . 7
2.10 Library Archive . . . 7
2.11 Releasing Your Application . . . 7

**3 Development Tools** **8**
3.1 Emacs . . . 8
3.2 Workshop Tools . . . 8
3.3 Version Control . . . 8
3.4 Unix Tools . . . 9

**4 File Organization** **10**
4.1 Header Files . . . 10
4.2 C++ Header Files . . . 14
4.3 Source Files . . . 17
4.4 C++ Source Files . . . 19
4.5 Some Comments About Comments . . . 19
4.6 Makefiles . . . 20
4.6.1 Makefile Style . . . 20
4.6.2 Makefile Template . . . 20
4.7 Documentation . . . 22
4.7.1 LaTeX . . . 22
4.7.2 Man pages . . . 23

**5 Function and Class Organization** **25**
5.1 C++ Functions . . . 26

**6 Program Statements** **27**
6.1 Variable Declarations . . . 27
6.2 Control Structures . . . 27
6.3 Conditional Expressions . . . 28
6.4 Precedence and Order of Evaluation . . . 28
6.5 Gotos . . . 28

6.6 Indentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
6.7 Whitespace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
6.8 Pointer Declarations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**7 Software Quality** **32**

**8 Defensive Programming** **33**
8.1 Safe `configure.in` Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
8.2 Conditional Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
8.2.1 Using `#else` Directives . . . . . . . . . . . . . . . . . . . . . . . . . . 33
8.2.2 Using Individual Names . . . . . . . . . . . . . . . . . . . . . . . . . . 34
8.3 Assertions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
8.4 The `DEBUG` Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
8.5 Freeing Memory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
8.6 Default `case` Statements . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
8.7 Miscellaneous C++ Advice . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

**9 Version Control** **36**
9.1 Introduction to CVS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
9.2 Integration with AFS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
9.2.1 Setting AFS Permissions Attributes . . . . . . . . . . . . . . . . . . . . 37
9.2.2 AFS Groups . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
9.2.3 Using AFS for the CVS Repository . . . . . . . . . . . . . . . . . . . . 38
9.3 Using CVS in a Group Project . . . . . . . . . . . . . . . . . . . . . . . . . . 39
9.4 For More Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**10 Releasing Software** **40**
10.1 Software Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
10.2 First Step . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
10.3 What to Include . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
10.3.1 The `configure` Script . . . . . . . . . . . . . . . . . . . . . . . . . . 41
10.3.2 Some Notes About `Makefiles` . . . . . . . . . . . . . . . . . . . . . . 45
10.3.3 Generated `.h` Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
10.3.4 Using `AC_` Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
10.3.5 Things to Not Include . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
10.4 Creating a Distribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
10.5 Verifying the Distribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
10.6 Version Numbers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
10.7 Once It's Out the Door . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**11 Project Web Pages / HTML and PHP3** **49**

**12 Where to Learn More** **52**
12.1 Resources in Print . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
12.2 Resources on the Web . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

**A The Ten Commandments for C Programmers (Annotated Edition) 54**
A.1 Lint . . . . . 54
A.2 NULL . . . . . 54
A.3 Type-Casting . . . . . 54
A.4 Header Files . . . . . 55
A.5 String Bounds . . . . . 55
A.6 Error Codes . . . . . 55
A.7 Libraries . . . . . 55
A.8 Braces . . . . . 56
A.9 Identifiers . . . . . 56
A.10 Portability . . . . . 56

# List of Tables

1 Standard filename formats. . . . . . 4
2 Examples of good filenames, and corresponding “bad” versions of the same filename. Notice that associated .h files should have the name basename as the .c or .cc file that contains their definitions. . . . . . 4
3 AFS permission flags and their meanings. . . . . . 38
4 Set of standardizes tests for autoconf/automake scripts. Additions to this list are welcome! . . . . . . 43

# List of Figures

1 Canonical sub-directory structure for an application under development. The directory tree contains CVS subdirectories that are created and maintained by the CVS software (i.e., they should not be modified). . . . . . 3
2 Canonical sub-directory structure for a released application. . . . . . 3
3 Typical Makefile that includes a top-level file (config.mk) with values substitued in from the configure script, as well as all the standardized targets. . . . . . 45

# 1 Introduction

*I really hate this damned machine*
*I wish that they would sell it.*
*It never does quite what I want*
*But only what I tell it.*
- Anonymous (`/bin/fortune`)

This document attempts to describe various aspects of software Quality[1] and particular mechanisms by which your software can attain Quality. Particular aspects of software Quality include:

- Robustness
- Reliability
- Efficiency
- Clarity
- Maintainability

Instilling your software with Quality requires discipline on your part as well as a proper appreciation for what Software Quality is. Some aspects of the discipline of Software Quality are attention to proper software style, defensive programming, and the proper use of programming tools.

In this document we use a top-down approach, discussing issues of application organization, file organization, function and class organization, statements and control flow, general issues of readability, and defensive programming.

[1]The definitive treatise on Quality (with a capital "Q") is *Zen and the Art of Motorcycle Maintenance* by Robert Pirsig. If you don't understand the difference between quality and Quality, you should read this book.

# 2 Application Organization

At the highest level, you need to organize the files that together comprise your application. We use the following definitions for the various categories of files that typically make up an application:

**Program Files:** The source code files for your application containing a `main()` function.

**Module Files:** The source code files for your application not containing a `main()` function. Groups of related module object files may be combined together into a library archive file.

**Header Files:** Contain interface information for functions, data types, and classes, as well as other semi-global information.

**Documentation:** This may include all of your software process documents, but should typically include a user's guide suitable for printing and on-line viewing, as well as `man` pages and `README` files.

**Configuration and Compilation:** These include gnu *autoconf* files (e.g., `configure`) for automated architecture specific configuration of your application as well as `Makefiles` for automated compilation.

**Version Control Subdirectories:** Each version control system typically requires some meta-information in the source tree itself. RCS, for example, requires links to the master RCS database directories. CVS automatically inserts "`CVS`" directories into each directory in your tree. See Section 2.9.

## 2.1 Development vs. Release

You should view your application in two different ways: as a development project (the code that you work on) and a releasable product (what you give to the end user). The released product will typically just be a subset of files and directories from your development project (still organized into the same directory structure).

In the following subsections, we will discuss the organization of your application from the development point of view. Differences that apply to the released product are discussed in Section 2.11.

## 2.2 Directory Structure

The files in your application should be organized into subdirectories. The canonical sub-directory organization is shown in Figures 1 and 2 (for development and release versions, respectively).

The basic contents of the subdirectories are as follows:

**doc:** Contains documentation for the application, excluding man pages.

**include:** Contains the header files for the application.

**lib:** Contains library archive files. In general, this should only be the repository location for the application executables *as they are being built*. If a library archive is also a deliverable of your application, it should be installed in a final (user specified) destination location.

**man:** Contains man pages for the application. This directory is further subdivided into `man1`, `man3`, etc., as required.

**src:** Contains program and module source files for the application. If the application contains a large number of different applications, this directory may be divided into subdirectories to facilitate organization.

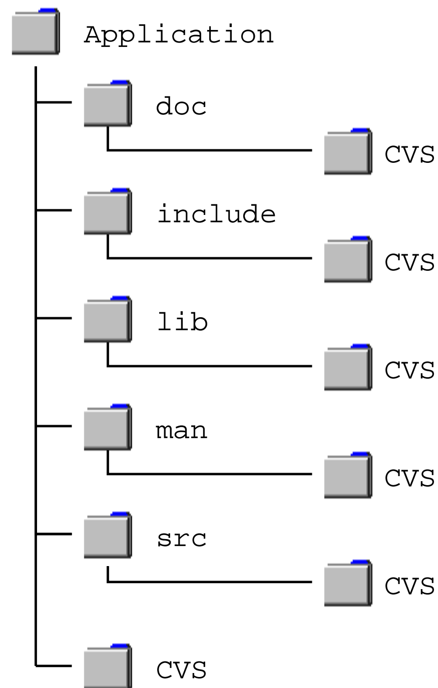



Figure 1: Canonical sub-directory structure for an application under development. The directory tree contains `CVS` subdirectories that are created and maintained by the CVS software (i.e., they should not be modified).

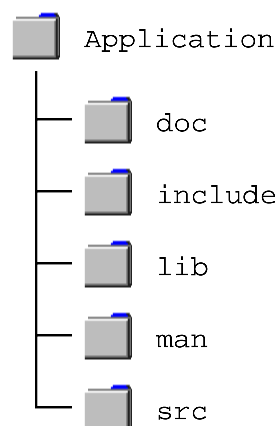



Figure 2: Canonical sub-directory structure for a released application.

| Type of file | Filename format |
|---|---|
| C source code | *.c |
| C++ source code | *.cc |
| C++ templates | *.cct |
| C/C++ header files | *.h |
| Lex/Flex source code | *.l |
| Yacc/Bison source code | *.y |
| Library archive | *.a |
| man page | *.[1-9][a-z] |
| PDF document file | *.pdf |
| Postscript document file | *.ps |
| LaTeX source code | *.tex |
| BibTeX source code | *.bib |
| Makefiles | Makefile |

Table 1: Standard filename formats.

| Good filenames | Bad filenames |
|---|---|
| mpi_send.c | ms.c |
| graphics_engine.c | geng.c |
| gradient_solver.c | dsolv.c |
| student_record.cc | rec015.cc |
| student_record.h | records.h |

Table 2: Examples of good filenames, and corresponding "bad" versions of the same filename. Notice that associated .h files should have the name basename as the .c or .cc file that contains their definitions.

## 2.3 File Name Conventions

Table 1 shows the required suffix convention for application files. In addition to the suffix requirements, you should indicate groups of related modules with a suitable prefix as well as a meaningful basename. Although at one time some compilers and operating systems limited filenames to 8 characters plus a three character suffix, those dark days are long behind us (we hope). Therefore, choose file basenames to be as meaningful as possible. There is absolutely nothing to be gained by the use of short cryptic filenames (just as there is nothing to be gained by the use of short cryptic variable names). Table 2 shows some examples of "good" and "bad" filenames.

## 2.4 Program Files

Your software project may consist of more than one executable. For instance, you may have a separate test program, or several example programs. Each of these executables will have a main() function as well as some other high-level functions. These functions all go in the program files. You may want to have a separate directory for each set of program files. For the most part, the code in the program files will not be very long since the majority of the code will be contained in the module files, which are shared by the programs.

## 2.5 Module Files

The highest level of organization within your software is the grouping of functionality into modules. Once compiled, these modules will be archived into one or more libraries, which can be conveniently linked into any executables that require their functionality. Each module contains functions that combine to perform a particular task, or functions that work on a particular type of data.

## 2.6 Header Files

Header files define the communication possible between files and modules. It also defines the communication between modules and the programs that use them. The larger the "communication channel" between different parts of your code, the more complicated your program can become. This is why programmers strive to create "clean" interfaces for modules of code. The buzz-words for this include *encapsulation* and *information hiding*. In practical terms, keep the number of function declarations in header files small. Also carefully design these functions to be easy to understand, with clear arguments and comments as to their purpose. In addition, header files are the main form of communication between group members. If Joe is working on the trigonometry module of a math program, I should not have to ask Joe what the name and argument list of a particular function I need to call. Instead, I should just look it up in `trig.h`, and not waste Joe's time.

## 2.7 Documentation

Documentation serves many purposes during the lifecycle of software development. In the beginning stages, it becomes a focal point for brainstorming and initial design decisions. It also enables group members to ensure that they are "on the same page". A major part of this is agreeing on names for entities in the problem domain of the software project. All of this is embodied in the *Requirements Definition and Specification* document.

Once the groundwork is laid, more detailed decisions can be made as to how the software will meet the requirements. In a sense, the *design* document is a log of these decisions. It is important that *nothing* is left out in this document. Typically, the worst bugs spring from difficulties that were unforeseen in the design. You may ask, "how can you possibly foresee everything"? The answer is, you should foresee as much as possible by rigorously following up every loose end in the design. The design will be your guiding light during the implementation stage. It will sit right next to your keyboard and prevent you from straying from the plan.

Once your software is complete, the user manual must be created. This can take several forms, and should include individual `man` pages for each executable program provided by your application (`man` section 1). If your software product is a library of functions to be used by others, you should include a `man` page for each function in your application's API (`man` section 3). See the `intro` manual page for each section to get a description of what that section is for, and all the subsections in that section.[2]

## 2.8 Configuration and Compilation

In the ever-changing world of computers, your software will need to adapt to many types of changes. It should be able to compile and link under a number of different types of operating systems (i.e., different versions, or "flavors", of Unix) and different hardware architectures (e.g., different types of workstations, such as Sun SPARC stations, IBM RS/6000s, SGI Indigos, etc.). You may even need to make sure that

[2] For example, `man -s 1 intro` and `man -s 3 intro` show the `intro` pages for sections 1 and 3, respectively.

your software can compile with multiple compilers for each architecture (e.g., for Solaris, software must be developed for both the Workshop and GNU[3] compilers).[4]

While at first it may not seem so, but `configure` is your friend. `configure` allows your `Makefiles` and header files to automatically change according to the system onto which your software is being installed.

### 2.8.1 Autoconf

`autoconf` is a GNU program that takes a `configure.in` shell script file that you created, and turns it into the `configure` script that a user will invoke in the first step of the installation of your software. RTFM[5] to learn how to use `autoconf`; there are a large number of macros available that help in creating cross-platform software.

As an example, typical `configure` scripts determine the specific flavor of Unix that it is being run on, locates specific libraries needed for compilation, determines placement of resulting binaries, libraries, and man pages, etc.

`autoconf`, and the `configure` file that it generates, is discussed more in Section 10.3.1.

### 2.8.2 Makefiles

`Makefiles` are invaluable to a programmer. When used properly, they will save you time by ensuring that only modified code is recompiled. A detailed template `Makefile` are described in Section 4.6. Typically, there is a `Makefile` in each directory of your project. The top-most `Makefile` should recursively invoke the `Makefile` in each sub-directory so that a single `make` command at the top level of the hierarchy will cause the whole project to be compiled, linked, etc.

### 2.8.3 Optimization

When compiling software for production use or timing results, you should *always* use at least "-O" level optimization (or RTFM on the compiler that you are using) to turn on the compiler's optimizer. The compiler can optimize at the machine code level; it can optimize much more than you can do in C code.

For the Solaris Workshop compilers, the following compiler flags can be used for aggressive optimization in C and C++:

```
-xtarget=ultra -xarch=v8plusa  \
-fast -xdepend -xO4 \
-xsafe=mem -xrestrict=%all -fsimple=2
```

For AIX, the following compiler flags can be used for aggressive optimization in C and C++:

```
-O3 -qarch=pwr2 -qassert=allp -Q
```

[3]GNU is a recursive definition that stands for "GNU is Not Unix". The GNU Free Software Foundation are huge advocates of providing software suites free of charge. They provide a large number of popular Unix tools for download on the Internet, to include `emacs`, `gcc/g++`, `autoconf`, `gzip`, `ghostview`, `flex/bison`, etc.

[4]While the GNU compilers (`gcc` and `g++`) are nice, they never produce optimal code for a given architecture. Native compilers should *always* be used whenever possible.

[5]RTFM stands for "Read The Frickin' Manual". It usually carries the connotation that the user should be able to reference the documentation (often available online, in the form of man pages, web pages, etc.) to find the details of a particular command or tool.

You should read the `man` page on your compiler;[6] the Solaris WorkShop compilers offer several different optimization choices, and even offer some auto-parallelization features for multi-processor machines. Moreover, different languages (e.g., Fortran, C++) may have different optimization flags available. All of these features can *dramatically* reduce the run time of your program.

## 2.9 Version Control

One of the biggest headaches of working in a group of programmers is sharing files. For example, one developer must not overwrite changes made by someone else who is simultaneously editing the same file. There are several tools that help solve this problem by either "locking" files for individual use (i.e., so that two people will not be editing the same file at the same time), or by determining which changes belong to which developer, and intelligently merging them into a single resultant file. The tool that you will use is CVS. It is briefly described in Section 9; you will need to RTFM on CVS to get specific information on how to use the latest version of CVS.

## 2.10 Library Archive

A library archive contains compiled source code for a collection of functions. Library archive files are always named with a `lib` prefix and a `.a` or `.so` suffix. These libraries can be linked into executables with the use of the `-l<name>` flag (leave off the suffix and `lib` prefix) for the compiler at the link stage of the `Makefile`.

One can create library archives with the use of the archive program, `ar` (RTFM). Some flavors of Unix require a second command, `ranlib` (RTFM on this as well), to create library archives. Solaris 2.5.1 does not require the use of `ranlib`, but provides it for hysterical raisins.[7]

## 2.11 Releasing Your Application

Since one of the goals of this lab is to create software that is used by people outside of our group (to include both users at Notre Dame and elsewhere), it is necessary to release software that we have created on the internet. However, releasing software is an incredibly complicated and detailed process. Creating a software release must be done *just right*; there is nothing worse that publishing software with our names on it that just doesn't work.

As such, *extraordinary* care must be taken when releasing software; the software that you publish becomes a reflection upon our lab and contributes to the public opinion about our group. Section 10 discusses releasing software in more detail.

---

[6] Author's node: the compiler's `man` page is similar to the Owner's Manual of an automobile: there's a ton of great stuff in there, but no one ever reads it.

[7] The term "hysterical raisins" is slang for "historical reasons", meaning that something is the way that it is solely for the reason of continuity with previous systems. The term comes from the Hacker's Dictionary, which is available in emacs (C-h i, and select "Jargon"), or at `http://locke.ccil.org/jargon/`.

# 3 Development Tools

Using the right tools is just as important as the actual code that you write, and frequently has a direct impact on your resulting software. Using a few tools wisely can prevent hours of wasted time in the lab looking for bugs.[8]

## 3.1 Emacs

Emacs is the editor of choice for all right-thinking programmers. It provides many tools that aid in the development of Quality software.

The use of `textedit` is strictly forbidden.

- If you have not already done so, you should go through the emacs tutorial, available with C-h t.
- It is highly recommended that you both edit and compile in emacs. Emacs has the capability to scroll through the compiler messages and jump to the corresponding line in the source code.
- Emacs can also be used for browsing source code when used in conjunction with a "tags" file. It is recommended that you build and use a tags file for all project source repositories.
- The `font-lock` mode should be used in Emacs so that the highlighting in different modes is consistent, and happens on the fly (as opposed to only happening when hitting C-l). Do not use the `hilite` mode – it is outdated.
- Never open more than one emacs process on the same computer – use C-x 5 2 to open a second window frame of the same emacs. This prevents you from editing the same file in two different instances of emacs.
- Use the emacs info mode to find out all kinds of information that usually supplements what you can find in `man` pages, particularly relating to GNU products. M-x info.
- As with all other aspects of your computing environment, you should customize your emacs environment to make you maximally productive.

## 3.2 Workshop Tools

The SunSoft Workshop tools are vital to software development. Type `workshop` at the command line, and the Workshop status bar will appear. If nothing else, you should use the debugger (you can use `debugger` to launch the debugger from the command line) to aid in debugging your software. The debugger is almost always preferable to "`printf()` debugging".

Additionally, the analyzer, LoopTool, and Visual Workshop tools are all very cool (and useful!). RTFM to find out more information.

## 3.3 Version Control

Version control is extremely important in software that will be released, especially if multiple people are working on the same set of source files. Use CVS (RTFM) for version control and/or when multiple people are working on one project. Section 9 gives an introduction to CVS.

**NEVER** check in source code that doesn't compile!

[8] The authors of this paper cannot stress this point enough; all of use have spent many many hours looking for silly mistakes in code that a good debugger would have found in minutes.

## 3.4 Unix Tools

There are several Unix commands that can make your life much easier and avert a lot of frustration when used properly. It is highly recommended that you RTFM on the following common Unix commands:

- `nm`. This command can look through libraries and see what functions and variables are defined in `.a` library files, `.o` object files, and non-stripped executables. It can even de-mangle the C++ names (if you used the Solaris WorkShop C++ compiler). This is very useful if you are:
  1. Trying to find our which library to use so that you can compile your program.
  2. Trying to find a bad function reference in a library/object file/executable.
- `grep`. `grep` and its variants (`egrep` and `fgrep`) can be used to either find a specific instance of something, or just to verify the existence of something (e.g., you can `nm` a library and pipe it through `grep` to see if a specific function exists in that library). `grep` can probably be best described with: “Some people have dogs. The rest of us have `grep`.”
- `man`. As stated before, RTFM.
- `make`. `Makefiles` are perhaps one of the greatest computer inventions since sliced bread. See Section 4.6 for more information on makefiles.
- `nohup`. The `nohup` command allows you to continue to run programs after you logout. That is, if you have a program that will take a long time to run, you can `nohup` it and go home. If you redirect the input and output, you can see the results of the run when you return.
- `makedepend`. This utility generates lists of dependencies for compilation targets. It is very useful in automating Makefile maintenance.

# 4 File Organization

As your program becomes larger, the organization of functions and type definitions into files becomes increasingly important. If this organization is done properly, each new function created by one of your group members will have a “logical” file to go into. When another group member needs to modify or look at this new function, they should be able to deduce which file it should be in. The group member that created the new function may not be around to help find the function, so it is important for this organization to make sense to everyone in the group. Group members should talk with each other about this organization scheme before any code is written to make sure it is clear to everyone.

Typically, functions that operate on similar data, or functions that work together for a common purpose should be grouped into the same file. For instance, all the functions necessary for a particular GUI window should be grouped together in a single file.

*All functions must be prototyped.*

The next step is to figure out where the functions will be used. If a function is only used in its file, then declare the function `static` at the top of its `.cc` file. If the function will be used from multiple files, it needs to be placed in the declaration of a header file.

Finally, you need to decide which structures, constants, etc are used in which files. If a definition is only used in one file, place it at the top of the source file. If it is used in multiple files, place it in a header file. Structure definitions and constants should be organized in header files in much the same way as functions in source files. Structures and constants should be grouped together and placed in separate header files based on their purpose and role within the program.

## 4.1 Header Files

Header files should be organized in the following way:

1. Prolog
2. Include loop protection
3. Includes
4. Macros
5. Extern Constant Declarations
6. Extern Global Variables
7. Structure Definitions
8. Functions Declarations

**Prolog.** This comment should explain how the header file fits into the file organization for your program. Your prolog should contain the following sections:

An example C header prolog is shown below. Note the comment syntax is specific to C. C++ header prologs may use the C comment syntax or C++ comment syntax (C++ is preferred unless the file may also be used by C program). Whichever syntax you choose, the prolog comment syntax used in the headers should be consistent throughout your project. Note the inclusion the `$Id$` and `$Log$` identifiers for use by CVS.

```
/*-----------------------------------------------------------------------
 *
 * File Name:   example.h
 *
 * Author:      A. Lumsdaine
 *
 * Revision:    $ Id $
 *
 *-----------------------------------------------------------------------
 *
 * NAME
 *     example.h
 *
 * SYNOPSIS
 *     #include "example.h"
 *
 * DESCRIPTION
 *     Example header file prolog.
 *
 * DIAGNOSTICS
 *
 *-----------------------------------------------------------------------
 *
 * REVISION HISTORY
 *
 * $ Log $
 *
 *---------------------------------------------------------------------*/
```

**Include loop protection.** Included files (header and template) should be wrapped up with suitable preprocessor statements to prevent re-interpretation:

```
#ifndef _EXAMPLE_H_
#define _EXAMPLE_H_

// Contents of the example.h file

#endif // _EXAMPLE_H_
```

The name of the wrapper macro should be the name of the file, with all letters in uppercase, pre- and post-pending underscores, and underscore substituted for dot. Note the final comment that denotes the end of the preprocessor block; since `#endif` does not indicate its matching start statement, a comment should be provided for clarity.

The only thing that you should be using the preprocessor for is for selective compilation of code – either for protection from re-interpretation, architecture dependent code, or debugging code. It will be helpful, nonetheless, if you comment your `#endif`'s as shown above. It makes it much easier to match up `#if`, `#else`, and `#endif` statements. The comment should just contain the keyword associated with the `#if` regardless of whether the `#if` is a `#ifdef` or `#ifndef`.

**Includes.** Include only the necessary files. When writing C++ code, enclose includes of C headers in the following:

```
extern "C" {
#include <c_header_file.h>
#include <other_c_header_file.h>
/* etc. */
}
```

**Macros.** Preprocessor macros, such as `#define DEBUG` go here.

In general, aside from include file protection, the `#if` constructs should be used in favor of `#ifdef`/`#ifndef`. While this is probably somewhat of a religious argument, the `#if` construct is slightly more general than `#ifdef`/`#ifndef`. For example:

```
#define TEST_FOO 0
#if TEST_FOO
  // some code...
#endif
```

has exactly the same effect as

```
#undef TEST_FOO
#if TEST_FOO
  // some code...
#endif
```

That is, `#if` will return true *only* if the macro is defined to have a non-zero value. It will return false if the macro is undefined, *or* if the macro has a zero value. `#ifdef` returns true if the macro is defined, just as `#ifndef` returns true if the macro is not defined – these constructs do not derive their result from the value of the macro.

Hence, it is safer to use `#if` rather than `#ifdef`/`#ifndef`, because it can be used to distinguish either a zero value, or a non-defined macro. These are the most common two ways of setting compile-time options – the defining (and/or strategically undefining) of preprocessor macros.

Recall that preprocessor names usually go in header files, and effectively make them globally scoped. As such, preprocessor macro names must be chosen with care. Especially if you are writing a user library, it is critical to add some unique (but common) prefix to *all* of your preprocessor macro names so as not to conflict with any user macros. The LAM/MPI library uses the `LAM_` prefix in all of its preprocessor macros, for example: `LAM_HAVE_SNPRINTF`.

**Constant Declarations.** The use of the `#define` macro for constants should be avoided. Use `extern` constant variables. The term *extern* means the variable is really defined elsewhere (in a `.cc` file). For example, if the file `trig.h` contained:

```
extern const float PI;
```

you would still have to instantiate this variable in a source file, such as `trig.c`:

```
const float PI = 3.14159;
```

Typically, it is a bad idea to have information duplication like the above definition/declaration of the `PI` constant. This increases the amount of work to keep things consistent when changing the code. The above duplication can be avoided with some fancy macro work. Here is `trig.h` again:

```
#ifndef _MAIN_C_
#define EXTERN extern
#define ASSIGN(x) /* */
#else
#define EXTERN /* */
#define ASSIGN(x)  = (x)
#endif

EXTERN const float PI ASSIGN(3.14159);
```

and the corresponding code in `trig.c`:

```
#define _MAIN_C_
#include "trig.h"

int
main(int argc, char* argv[])
{
  /* program */
}
```

**Global Variables.** These scourges of programming can be almost completely avoided in a well written program. If your program has more than a half dozen global variables, you should redesign your functions. Typically, you can replace globals by adding arguments to functions. This has several advantages:

- This makes clear what variables are affected and what variables are used in the function.
- This tends to make functions more "general". You are more likely able to reuse a function if it does not use globals.
- This makes dependencies with other parts of the program more visible, which means you will have fewer bugs due to unforeseen side effects.
- Global variables are inherently not thread-safe.[9]

**Structure Definitions.** Use the following format for `struct` definitions in C. Make sure to use the `typedef` so that you do not need to use the keyword `struct` when declaring variables of this structure type.

```
typedef struct somename_ {
  /*
   * members of the struct
   */
} somename;

somename array[ARRAY_LEN];
```

[9] If you have not had an operating systems class, you may not know what a thread is. Suffice it to say that being thread-unsafe can be a bad thing in terms of what types of programs you should want to write.

**Function Declarations.** For each source file, there should be just one header file that contains the function declarations for that source file. These files should have the same name to show that they are related. Only place function declarations in the header file if the function needs to be used outside of its source file. Otherwise, just put the function declaration at the top of the source file and mark those functions as `static`. For example, `trig.h` is as follows:

```
#ifndef _TRIG_H_
#define _TRIG_H_

double compute_pythag(double a, double b);

#endif // _TRIG_H_
```

Corresponding code in `trig.cc` includes the `compute_pythag()` function as well as a `static` local function, `local_pythag_helper()`:

```
#include <iostream.h>
#include "trig.h"

static double local_pythag_helper(double alpha, double beta);

double
compute_pythag(double a, double b)
{
  double ret = local_pythag_helper(a, b);
  // rest of function
  return ret;
}

static double
local_pythag_helper(double alpha, double beta)
{
  // pythagorean helper function
}
```

## 4.2 C++ Header Files

Except for very small classes, limit each header file to the declaration of a single class. Use the class name as the base name of the file. If a class is only to be used by one other class, embed the class within the `private` section of the class that uses it.

Before the class definition, there should be a comment that explains the role of the class plays within the program, and how it interacts with other classes. You should also explain the typical usage of instances of this class. Pretend that you are writing the `man` page for this class. Particularly important to describe is the “life cycle” of the object. This should cover the how, when, and why of the object's creation, destruction, and short but useful life.

The class definition itself should follow this order:

1. Friends
2. Public members

3. Protected members
4. Private members

Within each of the protection levels, organize the members in the following order:

1. Type definitions
2. Static methods
3. Methods
4. Data

The following is an example is of an a sparse matrix class that uses the STL `vector` class.

```
class ExtendedSparseMatrix {
public:
  class Entry1;
  typedef vector<Entry1> EntryVector;
  typedef int index;
  typedef double real;

  ExtendedSparseMatrix(index m, index n, bool rowMajor);

  void set(index i, index j, real v);
  real get(index i, index j) const;

  EntryVector& vec(index i) { return matrix[i]; }

  bool isRowMajor() const { return rowMajor; }
  index numRows() const { return nrows; }
  index numCols() const { return ncols; }

  // linear algebra methods

  void matVecMult_dot_product(vector<real>& x, vector<real>& b) const;
  void matVecMult_linear_comb(vector<real>& x, vector<real>& b) const;
  static void* multiThreaded_MatVecMult_dot(void* info);

  void matMatMult_dot(const ExtendedSparseMatrix& B,
                      ExtendedSparseMatrix& C);
  void matMatMult_linear(const ExtendedSparseMatrix& B,
                         ExtendedSparseMatrix& C);
  void clear();

protected:

  class Entry1 { // 1 index stored
    friend class ExtendedSparseMatrix;
```

```
    union { index col; index row; };
    real value;
    Entry1() : col(-1), value(0.0) { }
    Entry1(index c) : col(c), value(0.0) { }
    Entry1(index c, real v) : col(c), value(v) { }

    bool operator < (const Entry1& e) const { return col < e.col; }
    bool operator == (const Entry1& e) const { return col == e.col; }
  };

  index nrows, ncols;
  vector<EntryVector> matrix;
  bool rowMajor;

private:

  // Empty
};
```

The following rules apply to all classes:

- The order of the functions in the header file should match that of the source file.
- Data members should **never** be `public`.
- Design your classes for inheritance; use `protected` members for all members that should be inherited. Only use `private` members for members that are truly "private"; a good example of truly private functions are "utility" functions that should only be accessed from within that class itself.
- In C++, you should never use global variables. If you must, at least use `static` class data members instead. For example:

```
class GraphicsContext {
public:
  // This data member is ok to be public, because it is not
  // really an internal member of an object; it is a static
  // member, and hence is a class variable, meaning that there
  // is only one for all the instances of this class.
  static GraphicsContext theGraphicsContext;
protected:
private:
};
```

- Constants should also be made `static` class `protected` or `private` data members.
- Initialization of data members within constructors should use the initializer syntax when possible. For example:

```
class ComplexNumber {
public:
  ComplexNumber(double r, double i) : real(r), imag(i) { }
private:
  double real, imag;
};
```

## 4.3 Source Files

Source files should be organized into the following sections:

1. Prolog
2. Include directives
3. Constants and enumerated types
4. Type declarations, i.e., `typedefs`
5. Function macros (in C++, use `inline` functions)
6. `extern` global variable declarations
7. `static` global variables
8. `static` function declarations
9. Function definitions

The following rules apply when organizing source files.

- All source files should have a prolog. An example C source prolog is shown below. Note the comment syntax is specific to C. C++ header prologs must use the C++ comment syntax. Note the inclusion the `$Id$` and `$Log$` identifiers for use by RCS.

```
/*-----------------------------------------------------------------------
 *
 * File Name:   example.c
 *
 * Author:      A. Lumsdaine
 *
 * Revision:    $ Id $
 *
 *-----------------------------------------------------------------------
 *
 * NAME
 *     demo
 *
 * SYNOPSIS
 *     demo -n [size] -v <inputfile>
 *
 * DESCRIPTION
 *     Example source file prolog.
```

```
 *
 * DIAGNOSTICS
 *
 *-------------------------------------------------------------------------
 *
 * REVISION HISTORY
 *
 * $ Log $
 *
 *-------------------------------------------------------------------------*/
```

- `#include` statements are placed *only* at the beginning of files.
- Your code should not have any number literals embedded within function bodies. Instead create a constant (i.e., a `const` variable) for the number, and use the constant instead. If the constant is only used within the file, declare the constant at the top of the source file. Otherwise, create an `extern` declaration in a header file. The use of named constants greatly improves the readability of your code. Also, if there is a need to change the constant, you only need to make the change in one place, instead of wherever it was used.

  Note that the use of 0 and 1 to initialize loop indices is an obvious exception to this rule.
- Use the proper format for numbers. Floating point numbers have a decimal with at least one digit on each side; hexadecimal numbers start with `0x`; long integers end in an `L`.
- All macro arguments should be enclosed in parentheses when they are used (to prevent problems with operator precedence). Remember, macros just expand the *text* of their arguments for their parameters. For this reason, you should also never use an expression that has side effects as an argument to a macro.

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

  In general, you should avoid the use of macros altogether in C++ and use `inline` functions and const variables instead. In C, you should minimize the use of macros.
- Multi-statement functional macros (macros that are invoked like function calls) should be wrapped up in the following way (so that they can be used like functions in all cases):

  ```
  #define LONG_MACRO(x, y) do { \
    /* body of macro */  \
  } while(0)
  ```
- `extern` globals should have a comment next to them which states the file they are declared in.
- Declare all functions; if a function is to be used in more than one source file, it should be declared in a header file. Otherwise, it should be declared `static` at the top of the source file that it is to be used in. Variables that are not exported (i.e., used outside of the `.c` or `.cc` file in which they are defined) should also be declared `static`.
- Remember that for functions and global variables the keyword `static` means the scope (or use) of the variable or function is restricted to the file.
- Group function definition by type and sort in a breadth-first order based on abstraction level. Put a in comment labeling each section of function types. If `main()` is in the file, it should come first.

## 4.4 C++ Source Files

- Put all of the function definitions for a class in a single `.cc` file. Use the class name as the base name of the source file.
- Place the definition of any `static` data members for the class at the top of the source file, after the `#include` directives.

## 4.5 Some Comments About Comments

Your comments should be in clear English (complete sentences), and should be at a conceptual level above that of the code itself. Don't just restate the code. Use comments to point out parts of the code that are not obvious.

```
int
foo(list& bar)
{
   // Re-sort the bar list in place in descending order
   // using a quicksort, carefully maintaining internal
   // pointers in an auxiliary array

   // ...implementation here...
}
```

Do not use `/*` and `*/` to comment out code for debugging purposes; this is ineffective if there are comments in the section that you are attempting to remove. Use `#if 0` and `#endif`. For example:

```
int
foo(int i)
{
  int j;

#if 0
  /* Section that you want to eliminate */

  j = 42;

  /* It can even be fairly large, and include comments. */
#endif

  return i;
}
```

This way, you can change the 0 to a 1 and “turn on” the entire section of code – you do *not* have to remove the `#if` and `#endif`. This is a very useful way to selectively compile sections of code for debugging purposes.

In C++ use the `//` style comments; do not use `/* */`.

## 4.6 Makefiles

`make` can be used to create one type of file from another. For example, it can create executables from C/C++ source files. It can also be used to generate Postscript files from their respective LaTeX source files. RTFM on `make` to get more information.

### 4.6.1 Makefile Style

Here is a list of items that should be followed when writing `Makefiles`:

- The default target should the main program(s) that will be built in that directory. That may include recursive `makes` in subdirectories.
- C and C++ programs should be compiled and linked separately. That is, they should be compiled from `*.cc` to `*.o` in the `.cc.o` rule, and linked separately into the final executable/library.
- Always include the following targets:
  - `depend`. This target which will generate a list of dependencies in Makefiles. This is vital for program development because `*.h` dependencies change over the course of development. Use either with the source compiler (with appropriate option) or `makedepend`.
  - `clean`. This target should conform to the GNU Makefile standard (see the GNU Coding Standards) for `make clean`. Additionally, it should remove template depository directories (using the Solaris WorkShop C++ compiler, this directory is named `Templates.DB`).
- Read the GNU Coding Standard and include any of the standard targets that are necessary for your software. For example, if you are making distributable software, you need to have the `dist` and `distclean` targets.

See also Section 10.3.2 for information on releasing software with `Makefiles`, particularly in conjunction with `configure` scripts.

### 4.6.2 Makefile Template

The following is an Makefile template that you can use as a starting point for your makefiles. Explanations of each part of the `Makefile` have been included, but RTFM on `make` to provide much more insight than is provided here.

```
# Solaris -*-Makefile-*-
# Assumes Solaris make and /usr/share/lib/make/make.rules

# User portion of Makefile
# List all your source files here
SOURCES         = main.cc reader.cc mmio.cc
OBJECTS         = $(SOURCES:.cc=.o)
TARGETS         = $(SOURCES:.cc=)

default: main
```

```
main:  $(OBJECTS)
        $(LINK.cc) $(OBJECT) -o main $(LDLIBS)

# Compiler macros to define which compilers to use
CC               = cc
CCC              = CC
FC               = f77

# Optimization macros to give specific optimization options
# to the compilers
OPTFLAGS         = -fast -xO4 -xtarget=ultra -xarch=v8plus \
                   -xsafe=mem -pto

# Include paths, so that the compiler will know where to look
# for #include files
INCDIRS          = -I/myhome/include

# Libraries and library paths.  The -L flag tells the compiler which
# directory to search for libraries in; the -l flag tells the compiler
# which libraries are used for linking the final executable.  In
# this example, we are linking to libmylib.a and libm.a (the math
# library).
LIBDIRS          = -L/myhome/lib
LDLIBS           = -lmylib -lm

# CPP flags -- assumed by COMPILE.c, COMPILE.cc, COMPILE.F and by
#                  LINK.c, LINK.cc, LINK.F
# This is where defines should go, for instance -DDEBUG
CPPFLAGS         =

# Compiler flags -- assumed by COMPILE.c, COMPILE.cc, COMPILE.f,
#     COMPILE.F and by LINK.c, LINK.cc, LINK.f, LINK.F
CFLAGS           = $(OPTFLAGS) $(DEBUGFLAGS) $(PROFILE) $(INCDIRS)
CCFLAGS          = $(OPTFLAGS) $(DEBUGFLAGS) $(PROFILE) $(INCDIRS)
FFLAGS           = $(OPTFLAGS) $(DEBUGFLAGS) $(PROFILE) $(INCDIRS)

# Linker flags -- assumed by LINK.c, LINK.cc, LINK.f, LINK.F
# Here, we combine the previous -L and -l flags into a single macro
LDFLAGS          = $(LIBDIRS) $(LIBS)

# Flags for makedepend
DEPFLAGS         = $(INCDIRS)

# Rules to make assembly code from .c, .cc, and .f files
.SUFFIXES: .c .cc .f .s

.c.s:
```

```
        $(COMPILE.c) -S $<

.cc.s:
        $(COMPILE.cc) -S $<

.f.s:
        $(COMPILE.f) -S $<

# Remove cruft from your directory, "make clean"
clean:
        /bin/rm -rf Templates.DB; /bin/rm -f *.o *

# Generate a dependency list here in the makefile.  Running "make
# depend" will make a list of all .h files that your .c, .cc, and .f
# files depend on (i.e., use).  When one of the .h files changes,
# make will recompile all .c, .cc, and .f files that #include that
# .h file.
depend:
        makedepend -- -xM -DCCC=CC $(DEPFLAGS) -- $(SOURCES)
```

## 4.7 Documentation

Documentation is vital to any software project. Most programmers hate *writing* documentation, but know that documentation is essential to any successful program. The RTFM acronym is a perfect example of how necessary documentation is; this document uses "RTFM" frequently to refer to external Unix and third-party documentation.

### 4.7.1 LaTeX

LaTeX is pronounced "lā - tek". It is *not* pronounced "lā - tex".

LaTeX is a *text formatting language*. It has a strange history behind it,[10] but is immensely useful for writing papers. While the LaTeX language takes getting used to, it is almost universal in Computer Science circles (indeed, many Engineering and Science disciplines also use LaTeX, not just Computer Science); many papers, theses, dissertations, and textbooks are written using TeX and LaTeX.

LaTeX in itself is a language; while it can be quite sophisticated, the basics are fairly easy to comprehend. As a language, you write your paper in a text editor (such as `emacs`) and then compile it with the `latex` program.[11] A good `Makefile` is extremly helpful to take the tediousness out of compiling your paper. Standard LaTeX `Makefiles` are readily available.

Once the text compiles successfully, you can generate any number of output formats from it. Most often, a postscript file is generated, which can be sent directly to a postscript printer (which most printers at Notre Dame are) or viewed on screen with the `ghostview` program.

[10]The short story: Donald Knuth, a revered God in Computer Science, and a professor at the University of Stanford, decided that he wanted to write [yet another] textbook, but decided that there were no good word processors available for such a purpose. So he decided to write a text formatting language for it. But he further decided that there were no good languages to write a new text formatting language. So he wrote the language web so that he could write TeX so that he could write his textbook. LaTeX is a package of macros on top of TeX that make your life much easier.

[11]Author's note: While LaTeX is a good system, and does really nice things for mathematical formulas, I will never get used to the concept of debugging a paper.

Why go to so much trouble? Why is LaTeX so widely used? LaTeX does wonderful things for mathematical equations; it has a simple interface for creating complex mathematical formulae, both for inline text equations and separate, itemized formulae. LaTeX also automatically generates the the following:

- Table of Contents (see page ii)
- Lists of Figures and Lists of Tables (see page v)
- Numbering for tables, figures, sections, footnotes, etc.
- Numbering, formatting, and inline citations of bibliographies
- In-text cross references (for example, Section 2.8.1 discusses the GNU `autoconf`, Appendix A.4 discusses C header files, while Table 2 shows "good" and "bad" filename examples, and is on page 4)

In general, LaTeX takes care of all the menial tasks of writing papers. Even if you insert a new section in your paper, and perhaps add a few more footnotes, LaTeX will automatically renumber everything and update all cross references.

A few points to remember when using LaTeX:

- Be sure to use the LSC `latex` binary – do not use the one provided by the OIT! We have special paths setup in our binary. We also use several important features of LaTeX that are not provided by OIT (most notably, PSNFSS).
- Set your environment variables that control the behavior of TeX software appropriately. In general, it is probably not necessary for you to explicitly set any TeX related environment variables as the defaults are properly set for our current directory structure.
- Order your bibliographical citations so they print in numerical order. That is, after you generate Postscript for your paper, examine the inline citations and change the order of citations in the `\cite` command if necessary. For example, these citations are in the wrong order: [7, 4]. These citations are in the right order: [4, 7].
- Always use `Makefiles` with the proper dependencies to generate the Postscript.
- At most, include one (1) chapter per `.tex` file.
- Spell check your documents before printing them out. Use `ispell` (RTFM – it's accessible in emacs, M-x ispell-buffer).
- Always use the central copy of style files (e.g., the `ndthesis` class file) so that you utilize the latest copy and get the most up-to-date changes.
- Use and contribute to the central BibTeX database.

### 4.7.2 Man pages

Your application should typically include `man` pages as a means for providing on-line help. You should include individual `man` pages for each executable program provided by your application (`man` section 1). Also, your software product is a library of functions to be used by others, you should include a `man` page for each function in your application's API (`man` section 3). See the `intro man` page for each section for a complete description of what that section covers, and a list of all the sub-sections in that section (e.g., `man -s 1 intro` will give the `intro man` page for section 1).

The sections that a standard `man` page should include are the following:

**NAME** The name(s) of the program(s) or API(s) described on the page

**SYNOPSIS** For executables, a concise description of usage (including all command line arguments). For APIs, a declaration of the function or variable as well as necessary `#include` directives to access the declaration.

**DESCRIPTION** A description of the functionality of the program or API. This will probably be the largest section of the `man` page, as it is likely to contain the majority of information about the program or function.

**RETURN VALUES** A description of return codes or return values. Exceptional values should be specifically discussed.

**SEE ALSO** Cross references. These are valuable references to additional information; if the `man` page that you are reading does not make much sense, read some of the **SEE ALSO** pages to get more background information.

**DIAGNOSTICS** A description of error messages or error codes that may be produced.

**BUGS** Any known problems.

View some standard `man` pages for examples: `printf`, `ar`, `ranlib`, `gzip`, `cvs`, etc.

# 5 Function and Class Organization

Function definitions should be in the following order:

1. Function description comment
2. Return type
3. Function name and argument list
4. Automatic variable declarations
5. Static variable definitions
6. Function body

**Function Description Comments.** Use paragraph long comments before each of the important functions as a description of the purpose of the function, what functions are called, and what variables are used or affected by the function. If you use an algorithm from literature, cite the reference.

**Return Type.** Function definitions should have the return type of the function left justified at column 0 on the line preceding the function name. This format allows `etags` (RTFM) to find the function name (and allows `emacs` to properly highlight the function name). The open curly brace should be on the next line, in column 0. ANSI-style argument lists should be specified; pre-ANSI argument lists will not be used. There should be no space between the function name and the "(" in either function declarations, function definitions, or function calls. For example:

```
void
foo(int i, char *name)
{
  // body of function
}
```

In addition, you should *always* explicitly state the return type. Do not rely on the default (which is `int`). The following is *strictly* disallowed:

```
main()
{
  // body of function
  return 0;
}
```

**Function Name and Argument List.** The correct declaration for `main` is:

```
int main(int argc, char* argv[]);
```

`main()` does *not* return `void`. Always use the prototype listed above. *Always* return an error code from `main`; the operating system expects one. If you don't return one, a random value is used (this is not a Good Thing!). See the C Commandments in Appendix A for a more comprehensive list of do's and don't's with regards to C (and, by extension, C++) programming.

Function names should be descriptive and easy to understand. Use complete words, not abbreviations. In order to convey a complete thought, you should at least use a two word combination of a noun and a verb.

**Automatic Variables.** Declare all variables used in the function at the top of the function body. As frequently mentioned, use descriptive, meaningful names for your variables. This can be more helpful than good comments for people reading your code.

**Static Variable Definitions.** Note the the use of the word `static` for variables has a special meaning when used inside (internal to) a function, and is a good thing to lookup in K&R (*The C Programming Language* by Kernighan and Ritchie [4], p. 83).

**Function Body.** Within a function, group statements into paragraphs, using white space to delineate each paragraph. Each "paragraph" of statements should carry out a specific logical task, which you should explicitly write down in the form of a comment at the beginning of the paragraph. Make sure that your comment adds information for the reader, and does not just restate the code.

Avoid having more than one return in a function. Functions with multiple returns are more difficult to understand and change. The following code snippets demonstrate how multiple returns may be eliminated. Example: multiple returns.

```
for (i = 0; i < max; i++)
  if (vec[i] == key)
    return TRUE;
return FALSE;
```

Example: single return.

```
found = FALSE;
for (i = 0 ; i < max && found == 0; i++)
  if (vec[i] == key)
     found = TRUE;
  return found;
```

Strive to keep functions less than a page long. In fact, under a half a page is ideal. If you find a function is becoming too long, take several of the statement paragraphs and turn them into a function. One rule of thumb is that you should generally never have more than one level of looping in any function, and not more than two levels of control structures such as `if` statements. Of course, this is only a loose guideline can be disregarded if you are trying to optimize a bottleneck in your program (which you will probably not need to do).

## 5.1 C++ Functions

- If the function does not change any of the data members of the called object, declare the function `const`. Figure this out before you code up the function, do not wait to do it at a later time.
- Declare pointer and reference arguments `const` if they are not changed within the function.
- Function names may be shorter in C++ because the noun is already given by the object. In most cases, a verb will do for the function name.

# 6 Program Statements

## 6.1 Variable Declarations

Automatic variables (variables local to a function) are declared at the top of the function body. The following are rules to apply in declaring these variables.

- Group related variables. Do not put unrelated variables on the same line.
- Variable and argument names, like function names, should be made of whole words and should clearly convey to the reader the purpose of the variable. In the function body, do not use the variable for a purpose that does not fit its name. Instead, create another variable with a different name.
- For very short functions (5 lines or less), arguments and variables names may be shorter, a letter or two, since the meaning should already be clear. This is also true for variables used in only short blocks of code.
- Use `const` where appropriate. Do not wait to put in `const` if you can figure out right away whether the variable will be `const`.

## 6.2 Control Structures

- Do not use side-effects within control structures. For example, the use of the ++ operator in the following example is a bad idea.

```
if ((a < b) && (c == d++))
```

- Split a string of conditional operators that will not fit on one line onto separate lines, breaking after the logical operators:

```
if (p->next == NULL &&
     (total_count < needed) &&
     (needed <= MAX_ALLOT) &&
     (server_active(current_input))) {
  // body of if statement
}
```

- The same is true for `for` statements. If a for loop will not fit on one line, split it among three lines rather than two:

```
for (vector<int>::iterator i = array.begin();
      i != array.end();
      ++i) {
  // body of for statement
}
```

## 6.3 Conditional Expressions

In C, conditional expressions allow you to evaluate expressions and assign results in a shorthand way. For example, the following if then else statement

```
if (a > b)
  z = a;
else
  z = b;
```

could be expressed using a conditional expression as follows:

```
// z = max(a, b)
z = (a > b) ? a : b;
```

While some conditional expressions seem very natural, others do not, and we generally recommend against using them. The following expression, for example, is not as readable as the one above and would not be as easy to maintain:

```
c = (a == b) ? d + f(a) : f(b) - d;
```

Do not use conditional expressions if you can easily express the algorithm in a more clear, understandable manner. If you do use conditional expressions, use comments to aid the reader's understanding.

## 6.4 Precedence and Order of Evaluation

There are 21 precedence rules. Rather than trying to memorize the rules or look them up every time you need them, remember these simple guidelines from Steve Oualline's *C Elements of Style*:

- * % / come before + and -
- Put ( ) around everything else

## 6.5 Gotos

You should not use `gotos` except for disastrous error processing. One proper use of `gotos` in C is when you need to handle a disastrous situation and jump out of multiple levels of logic. In C++, throw an exception instead of using `goto`.

```
// C code
for (...)
  for (...) {
    ...
    if (disaster)
      goto error;
  }
...
error:
  // error processing

// C++ code
try {
```

```
  for (...)
    for (...) {
      ...
      if (disaster)
        throw Exception;
    }
  ...
}
catch(Exception e) {
  // error processing
}
```

## 6.6 Indentation

C and C++ indentation styles are hotly debated among programmers. In CSE 532 we basically use K&R, with some modifications. If someone gives you C or C++ source code that is not properly indented, fear not. The `indent` command will reformat the file so that it is readable to all right-thinking people (RTFM on `indent`). The following sequence of options for `indent` will (more or less) properly format C source code files.

```
-bap -bacc -bad -br -nbs -ncdb -ce -di1 -nfc1 -i2 -ip2 -sc -c40 -l75 -npcs
```

You should put these options in your own `.indent.pro` file (in your `$HOME` directory) so that `indent` will automatically use them. You will find that if you are properly using `emacs`, you will only need to use `indent` to fix other people's source code.

The items below apply to both C and C++:

- Two (2) space indents should be used. No more, no less. Placing the following expressions in your `.emacs` file will cause emacs to properly use two space indents. It is especially important to conform to this because the indentation will become garbled if other programmers modify/extend the code you have written using a different number of spaces.

  ```
  (setq c-argdecl-indent 2)
  (setq c-indent-level 2)
  (setq c-continued-statement-offset 2)
  (setq c-label-offset -2)
  ```

- When writing `if`, `for`, `while`, and `do` statements, if there is only one statement within the control structure, do not use curly braces. If there is more than one statement, use curly braces. In either case indent two (2) spaces:

  ```
  for (i = 0; i < max; i++)
    for (j = 0; j < max; j++) {
      // Body of loop that is more than one (1)
      // statement long
    }
  ```

  There is an exception to this rule to watch out for in more complicated logic. If you are using Emacs, you will find that the last `else` is aligned with the inner `if`, which will alert you to put in the extra curly braces.[12]

[12]This ambiguity is responsible for the single shift/reduce conflict in the BNF description of the C language.

Example: Absence of braces produces undesired result.

```
if (n > 0)
  for (i = 0; i < n; i++)
    if (s[i] > 0) {
      (void) printf("...");
      return i;
    }
else // WRONG -- the compiler will match to closest
     // else-less if
  (void) printf("error - n is zero\n");
```

- The open curly brace ("{") goes on the *same* line as the `if`, `for`, `while`, or `do` statement. Do *not* put the open curly brace on the next line. The close curly brace ("}") goes on a line by itself, left justified with the keyword that opened the block statement. This also applies to any other block-type statement, except for function definitions. `else` should be "cuddled" with the closing brace of its `if`.

  For example:

```
if (i == 0) {
  // Put body of if here
} else {
  // Put body of else here
}
```

## 6.7 Whitespace

The following rules apply to the use of whitespace in your code.

- Include whitespace on the right of all keywords (`if`, `for`, `while`, `do`, etc.), and on the left if appropriate.

```
if ((a / 3) == 2 || (q + 1) == w)
  // Body of if statement
```

  The exception is `sizeof`, it should be spaced as if it were a function call.

- Include whitespace on both sides of all operators except `(`, `->`, `.` (the period), `[`, `!`, `,` (the comma), `)`, and `;`.

```
for (i = 1; i <= n; i++)
  product += 1 / (5 * (i + 4));
```

- There should be one space between a declaration keyword and the following identifier.

```
double x, y, z;
```

- There should be no whitespace surrounding the `->`, `.` (dot), `[`, and `!` operators.

```
rptr = newRes();
rptr->next = resList;
resList = rptr;
```

- Whitespace should be included only on the right of: `,` (the comma), `)`, and `;`. It may be omitted when used in multiple-level algebraic formulas, as shown for `(`, above.

```
double x, y, z;
```

- There should be a blank line after every procedure body.
- There should be a blank line around every conditional compilation block.
- There should be a blank line after every block of declarations.

## 6.8 Pointer Declarations

In C, the `*` operator, when used in argument lists or variable declarations, should be "on the right". For example:

```
void foo(int *int_pointer);
```

In C++, the `*` and `&` operators should be "on the left". For example:

```
void foo(int& int_reference);
```

# 7 Software Quality

Among other things, Quality software must be bug free. Although there has not yet been invented a tool that will automatically debug your code for you, there are several tools that will help you locate some obvious problems in your software.

**The compiler.** Most compilers have flags that allow you to adjust the level of pickiness for the warning messages that are emitted. You should *always compile with the highest level of pickiness*. You should eliminate all warning and error messages from your code – delivered code should compile completely with no warnings and no errors. If you do not know what an error message means, you need to 1) try to figure out what each word of the message means, 2) try to lookup what it means, 3) ask a smart friend, and as a last resort 4) ask your TA.

For Solaris CC, use `+w2`, for Solaris `cc`, use `-v`, for `gcc` and `g++`, use `-pedantic -Wall`. It should be noted that the recent IRIX compilers are good at finding warnings that the Solaris and GNU compilers missed – it will be worth your while to try them as well.

On a related issue – do not use the compiler as a syntax checker. You should completely understand everything that you enter into the computer and be able to explain why you did what you did. Only compile your code when you are satisfied that what you have entered is really what you wanted (that it is *right*). This doesn't mean that you should type in your whole program and then try to compile. If you do this, you are asking for a migrane. Instead, try to choose small pieces of code that you can code up, compile, run and test independently of the rest of your program.

**lint.** Lint is a wonderful, under-appreciated, and under-utilized tool for source-code checking. All of your C code should be lint clean – i.e., you should be able to run `lint` on your entire project with no warnings or errors.

**bcheck.** The Solaris Workshop debugger has utilities for checking the memory use patterns of your programs. Among other things, it can check for reading/writing to unallocated memory, reading from uninitialized memory, and memory leaks. The `bcheck` program allows you to make these checks from the command line (i.e., without using the debugger GUI). Your code must not generate any warnings or errors when `bcheck` is run with all checks enabled.

Memory problems are particularly troublesome and difficult to find with the debugger. Typically, memory problems will manifest themselves by some unexplainably corrupted data in one part of your program. What makes this problem so difficult to rectify is that the corruption was caused inadvertently in a complete separate part of your code. The only way to systematically find these types of errors is to use tools like `bcheck` (or its manifestation within the Workshop debugger).

All of your code written for must conform to the style specified in this document. In addition, your the code must:

1. Compile with maximal pickiness and no warnings.
2. Be lint clean (C code).
3. Have no memory related errors as reported by `bcheck`.

Verifying that your software software meets these minimum quality metrics is your responsibility. If you perform these checks periodically you will be able to keep problems to a minimum.

# 8 Defensive Programming

## 8.1 Safe `configure.in` Tests

It is safer to *always* define a given C/C++ preprocessor macro (particularly a macro which will hold a true or false value) – either define it to be 0 or 1. These preprocessor macro values are typically determined in the `configure` script (generated either with `autoconf` or any of its cousins). Note that it is *not* sufficient to simply define one case (e.g., true) and ignore the other (false). For example, the following snipit from a `configure.in` file is not safe:

```
# ...some test that puts its result in $HAVE_FOO
if test "$HAVE_FOO" = "yes"; then
  AC_DEFINE(HAVE_FOO, 1)
fi
```

This code will define the C preprocessor macro "`HAVE_FOO`" to be 1 if the test turned out to be true. However, it does not define anything if the test was false – it `assumes` the the preprocessor macro is already set to 0 by default. This is not a good assumption. Rather, the following code is safer because it always sets the `HAVE_FOO` macro to either 1 or 0.

```
# ...some test that puts its result in $HAVE_FOO
if test "$HAVE_FOO" = "yes"; then
  AC_DEFINE(HAVE_FOO, 1)
else
  AC_DEFINE(HAVE_FOO, 0)
fi
```

## 8.2 Conditional Compilation

### 8.2.1 Using `#else` Directives

If you have preprocessor directives that indicate options that should be compiled in (or out) of your software, be sure to be all-inclusive. It is not sufficient to just program for just the "true" cases – frequently, it is necessary to program for the "false" cases as well. For example, the following C code is not necessarily safe:

```
char fname[80];
#if HAVE_SNPRINTF
snprintf(fname, 80, "/tmp/baz-%s.%d", name, getpid());
#endif
FILE *fp = fopen(fname, "r");
```

It is almost always a bad idea to have a single `#if/#endif` statement with no `#else` statements in the middle – only do this if this is absolutely what you are sure that you want to do. The code snipit shown above is obviously going to fail if `HAVE_SNPRINTF` is 0, because `fname` will not get set properly, and the `fopen()` call will get a random string. The following code is safer:

```
char fname[80];
#if HAVE_SNPRINTF
snprintf(fname, 80, "/tmp/baz-%s.%d", name, getpid());
#else
```

```
my_snprintf(fname, 80, "/tmp/baz-%s.%d", name, getpid());
#endif
FILE *fp = fopen(fname, "r");
```

In this way, `fname` is guaranteed to be set to the right value.

### 8.2.2 Using Individual Names

When creating these preprocessor directives to create portable code, it is always better to have names that reflect specific conditions rather than trying to lump a group of conditions into a single name (such as an operating system name). Using the name `USING_SOLARIS`, for example, instead of `HAVE_SNPRINTF` (and others) is not a good idea because Solaris will change over time, perhaps changing the values which `USING_SOLARIS` means. Indeed, even different installations or versions of Solaris may require different values.

While it is a bit more painful, it is always more safe to test each non-portable item separately in your `configure` script (e.g., whether the underlying operating system – regardless of what it is – has `snprintf()` or not) and define a preprocessor macro for each rather than trying to glump many pre-determined conditions into a single name.

## 8.3 Assertions

The `assert()` macro is an important tool for putting diagnostics (sanity checks) into programs. When executed, a failed assertion will produce diagnostic messages on the standard error output device and abort the program.

The synopsis for `assert()` is the following:

```
#include <assert.h>

void assert(int expression);
```

`expression` can be any valid boolean (in the sense of the C or C++ languages) expression. A false value of the expression will result in a failed assertion.

The macro `NDEBUG` will prevent assertions from being compiled. Use the preprocessor directive `-DNDEBUG` to exclude assertion compilations from an entire application. This is a simple way to strip out all `assert()` instances from production-level code.

The `assert()` macro is particularly useful for doing sanity checks in your program — it should be used to check for things that are drastically wrong *in your program*. It should not be used for checking for such things as user errors which should be handled gracefully by the program. That is, when properly functioning, your program should function the same with or without the presence of the `assert()` macros.

The following assertion checks that an array index is within its proper bounds:

```
assert(i >= 0 && i < n);
```

Whether to leave assertions in your released code is up to you. Leaving them in may impact performance but it will give you the user opportunity to provide you with more useful information if (heaven forbid) a bug occurs. The best option is to leave the assertions in your code but to have the `Makefile` compile your application with `-DNDEBUG`. Then, if a problem with your application occurs, you can ask the user to recompile without `-DNDEBUG` and report any failed assertions.

In general, you should be very liberal in your use of `assert()`.

## 8.4 The `DEBUG` Directive

All code that is used only during the development and debugging phase of your project should exist only within the context of a `DEBUG` directive. The code is thus easily excluded. The format is as follows. The particular value of `DEBUG` is checked determines whether the debugging code exists (or not); if `DEBUG` is 1, the code is compiled. If `DEBUG` is 0, the code is stripped out before it reaches the compiler.

```
#if DEBUG
  // debugging code in here
#endif
```

## 8.5 Freeing Memory

Destructors typically free memory associated with an object. *Debugging destructors* not only free memory, but also wipe the values of all class data before freeing the memory. contents of the RAM before freeing it. This simple step will help identify potential memory bugs and mis-matched reference counting schemes.

The body of the code that wipes the class data should be enclosed in an `#if DEBUG` preprocessor directive; once the code has been proven to work properly and there are no memory problems, wiping the class data should no longer be necessary.

## 8.6 Default `case` Statements

You should never assume the argument to a `case` statement will always have a valid value. You should always have a `default` for your case statements to catch invalid values.

## 8.7 Miscellaneous C++ Advice

The following points have proven to be good advice:

- Multiple and/or virtual inheritance is bad! Don't use it!
- Use references, not pointers, wherever possible.
- Beware of unnoticed copy constructors, or other implicitly-invoked functions. They can cost heavily in performance.
- Always override the default “Big 4” (default constructor, copy constructor, assignment operator, and destructor), unless you are *absolutely sure* that you know what you are doing.
- Use `friends` as little as possible. You probably will not need to use them at all if you associate functions with the proper classes as well as provide the correct accessor functions to private data members.
- Use `[]` to denote arrays in argument lists, not `*`. This is more semantically precise.
- Do not write your own list data structures and algorithms. Use the Standard Template Library.
- Do not use `char*` for strings. Instead, use the new C++ Standard's `string` class.

# 9 Version Control

Version control software is vital when multiple programmers are simultaneously working on one software project. The version control software that you will be using is CVS.

## 9.1 Introduction to CVS

CVS is a set of tools for managing the development of large software systems over a long period of time and with multiple developers.[13]

The basic idea behind CVS is to have a single (global) repository for your development code, from which all developers in the project can check out their own working copies. Individual developers work on their copies, add functionality in a modular fashion, and then propagate their changes to the central repository. Once changes have been propagated to the repository, they are visible to everyone.

To ensure consistency in the central repository, CVS can operate in two modes: developers can *lock* individual files so that only one person can work on a file at a time, or developers can *share* files, and CVS resolves most conflicts (but sometimes human intervention is required; surprisingly, CVS is able to correctly figure out conflicts in most situations).

Full documentation of CVS is not included here (RTFM) because it is far too long. However, some key concepts are listed below:

1. Choose a directory location to be the global project repository. Everyone in the group should have full access to this directory (see Section 9.2).

2. Set the environment variable `CVSROOT` to the top level of this central repository (it can be specified on the command line as well, but using `$CVSROOT` is more convenient).

3. CVS will maintain your whole development tree (as in Figure 1 on page 3). CVS will place some extra directories named `CVS` throughout the tree; these directories contain meta-state about your changes and should not be edited.

4. Each developer can decide which mode to operate in: locking or shared. While Emacs can be used to check out and check in individual files in locking mode, the command line mode is also important for updating the whole source tree, commiting large numbers of files, etc.

It is *highly* recommended that you RTFM on CVS, especially if you have never used a version control software system before. You may also want to setup a “dummy” project tree with a few source code files, header files, and a Makefile or two. Experiment with other members in your group with this “dummy” tree to learn how to use CVS properly.

## 9.2 Integration with AFS

Since your central CVS repository resides in AFS, it is necessary to set permissions such that every member in your group has read and write access to it.

AFS allows a higher degree of security than standard Unix. While standard Unix provides *read*, *write*, and *execute* permissions for three different groupings of security (*owner*, *group*, and *other*), AFS allows setting (among others) *read*, *list*, *insert*, *delete*, *write*, for individual or groups of AFS users. This higher level of flexibility greatly enhances the security possibilities available for your files.

[13]The Netscape Navigator development teams use CVS.

### 9.2.1 Setting AFS Permissions Attributes

You are probably already familiar with the `fs` command. With `fs`, you can set and reset AFS permissions flags. The `fs` command can also perform many other AFS functions which are beyond the scope of this tutorial; we will only discuss the permission-setting functions here. For your own personal edification, type `fs help` to see the full list of `fs`'s options. To examine the permissions list for a given directory, type the following command:

```
unix% fs la dir_name
```

This will display a list of the AFS users and groups who are explicitly mentioned in the permissions list for the AFS directory `dir_name`, as well as their associated permission rights. To set and reset AFS security flags on a given directory, the format of the `fs` command is:

```
unix% fs sa dir_name AFS_username permissions
```

Where `dir_name` is the target AFS directory, `AFS_username` is the target AFS user that you with to explicitly specify in `dir_name`'s permissions list, and `permissions` are the actual flags that you want to set. Note that you reset flags by not setting them.

`AFS_username` can not only take the value of any AFS user ID, it can also take the value of the special AFS group `system:anyuser`. This group represents all AFS users. That is, whatever permissions that you assign to `system:anyuser` for a given directory will be used for all users not otherwise explicitly stated in the permissions list. For example:

```
unix% fs sa $HOME system:anyuser rl
```

will give read and list privileges to any user not otherwise listed in the AFS permissions for your home directory. You can also take away someone's access with the special permission `none`:

```
unix% fs sa $HOME system:anyuser none
```

This will remove the AFS group `system:anyuser` from the permissions lists of your home directory. That is, unless an AFS user is explicitly stated in the permissions list of your home directory, they will have no access rights. **BE CAREFUL!** AFS does not protect against stupidity – it is legal to give yourself the `none` permissions setting on your own directory!

The settable flags that the `fs` command will accept are listed in Table 3

The `administrate` flag enables a user or group to set or reset the AFS permissions flags on a given directory. It should also be noted that AFS permissions, like most things in Unix, are inheritable. That is, if you make a subdirectory, it will inherit the same permissions as its parent (which of course, are changeable).

### 9.2.2 AFS Groups

AFS provides for the ability to specify multiple AFS users under one logical name. This ability is called grouping, and is slightly different than the standard Unix implementation of grouping. These groups are most useful when working on projects that require several AFS users to have access to the same files.

AFS groups are accessed with the `pts` command. As a normal AFS user, you can create, edit, and destroy groups that begin with your AFS id. For example, the AFS user `rplant` could create a group for his software project with the following command:

```
unix% pts creategroup rplant:cse532_team
group rplant:cse532_team has id -612
```

| Flag | Meaning |
|---|---|
| r | Read |
| l | List |
| i | Insert |
| d | Delete |
| w | Write |
| k | Lock |
| a | Administrate |
| read | Combo: rl |
| write | Combo: rlidkw |
| all | Combo: rlidkwa |
| none | None |

Table 3: AFS permission flags and their meanings.

AFS will return some non-important numeric identifier for this group. Once the group has been created, several operations are possible. These include: adding AFS users to the group, listing the AFS users in the group, and deleting AFS users from the group.

```
unix% pts adduser jpage rplant:cse532_team
unix% pts membership rplant:cse532_team
Members of rplant:cse532_team (id: -612) are:
  jpage
unix% pts removeuser jpage rplant:cse532_team
```

The main purpose for defining AFS groups is that they can be used as the AFS_username argument in the fs command. So if rplant executed the following commands:

```
unix% pts adduser jpage rplant:cse532_team
unix% fs sa $HOME rplant:cse532_team rl
unix% fs sa $HOME/CVS-repository rplant:cse532_team write
```

then he and jpage would be able to access the CVS repository. Note how it was not necessary to set the administrate flag to the group's access; this flag should only be set for the owner and the special AFS group system:administrators (select OIT personnel). As more users are placed into the rplant:cse532_team group, they will have all the access rights that have been defined for rplant:cse532_team.

### 9.2.3 Using AFS for the CVS Repository

With AFS groups, you can dynamically define, associate, and change a group of users with a set of AFS permissions. The team leader should create an AFS group with the other members of the team. Then, using the fs command, set permissions for that AFS group to read and list (rl) in the directory hierarchy all the way down to the destination top-level CVS repository directory. In the top-level repository directory, it is necessary to give write permissions to the rplant:cse532_team group so that the members of this group and read and write to the CVS databases.

As described in the preceding paragraph, it is only necessary for rplant to give read and list privileges to rplant:cse532_team in his home directory. This group does not need full writing privileges in rplant's home directory; it only needs read and list in order to reach the $HOME/CVS-repository

subdirectory. This can also be taken a step further; `rplant` only has to grant `write` privileges in the top-level CVS repository directory and sub directories for those projects for which `rplant:cse532_team` need to work on. If `rplant` has other project trees in CVS, he does not need to give `rplant:cse532_team` access to them at all. For example, `rplant` could define `rplant:netscape_team` and give that group access to selected schematics.

## 9.3 Using CVS in a Group Project

The fundamental rule when using CVS in the context of a group project is: **Only check in working files. Never** check in code that does not compile or that is otherwise broken. The other members of your group will be updating their local repositories with the files that you check in. If you check in a file with a bug, or worse yet, a file that will not compile, you will disable your entire group.

On the other hand, you should attempt to be brief with your checkouts, as this will reduce the chance for conflicts in CVS that require human intervention.

In general, you should make sets of well thought out changes that do not require the files to be checked out for long periods of time.

## 9.4 For More Information

For more information on the actual use of CVS, read the `man` pages for CVS, and the Emacs information pages for "Version Control".

Other version control systems include `SCCS`, `RCS`,[14] and **TrueChange**.

[14] RCS is actually the predecessor of CVS; CVS uses RCS internally to manage its database of files. CVS is known as "RCS on steroids".

# 10 Releasing Software

Since the LSC is an applied lab group, one of our goal is to release usable software as the result of our research efforts. That is, we go all the way from theory to application. Most of our research efforts result in some kind of software, which is then published in some form (usually on the internet).

It is imperative that all the software that this lab group publishes contains a high level of Quality. Since the only way that most people around the world will recognize and build an opinion of our group is through the software that we publish, the software must be created in a logical, functional, and correct manner.

Of course, through the development life of a software package, there will be bugs – especially with large and complex software packages, it is difficult (if not impossible) to release without some number of bugs – but there are steps that can (and will) be taken to minimize the number of bugs in a release.

## 10.1 Software Packages

There are two fundamental kinds of software packages: applications and libraries. Applications are generally easier to release, because you are delivering the entire product in a self-contained form. Libraries are a bit more difficult because your software must account for how the user will use it in their application, which tends to make design and documentation more complex. That is, even though library packages are likely to include some "helper" applications, the main product is the library, not the applications that come with it.

These two types of packages are slightly different in their build and install mechanisms; particularly for C++ template libraries since there are likely no binaries to install (everything will most likely go into `$prefix/include` somewhere). Applications will likely go into `$prefix/bin`, and have other support files that must be installed under `$prefix` somewhere. Libraries typically go under `$prefix/lib`, and have associated header files as well.

## 10.2 First Step

Before *any* package is published by the LSC, it must meet all of the items discussed in Section 7 – Software Quality. You should also use all of the techniques described in Section 8 – Defensive Programming. Using the techniques listed in these two sections will pre-empt many bugs (or at least greatly assist in finding bugs).

As a first step towards releasing, your software must be:

1. Compiler error free. This is an obvious one (because your software wouldn't compile otherwise), but is listed here anyway.

2. Compiler warning free. Compilers print warnings for a reason. Your software must compile under all reasonable compilers with maximum pickyiness enabled with *no* warnings. Fix/eliminate all warnings that any compiler issues.

3. Functional. Your software should do what it advertises that it does. Exceptions should be noted in documentation.

4. Documented. At least *some* form of documentation is required for all released software. The majority of people who use your software won't care how it works – they will want to follow a set of instructions to configre, compile, and install it. Then they just want to *use* it – they won't want to discect the inner code to figure out the subtle details. Documentation should provide instructions on how to build and use your software.

5. `bcheck` and/or `purify` clean. Your software must run (particularly if you are supplying a library) under a variety of situations and not have any memory errors or warnings. For example, all memory that you `malloc()` or `new` must be freed with `free()` or `delete`/`delete[]` (as appropriate).

   There are some cases where software cannot be `bcheck` clean. For example, it seems that Solaris 2.6's `gethostbyname` allocates some internal memory that it never frees. This cannot be helped. But for every unavoiable case, documentation must be provided stating *and proving* each case where you cannot fix the `bcheck` problems.

   There are other cases where being `bcheck` clean will cause degredation in overall performance. For example, the LAM/MPI library frequently sends a standardized message structure across sockets. However, certain types of messages do not require that all of the elements in the structure, so LAM does not bother to initialize them. This is known to be safe, because *in each case*, the LAM authors verified that the receiver of the message does not used the unitialized data members. In cases like this, documentation must also be provided. Additionally, some compile-time directives must be provided to force the code to be `bcheck` clean (albeit at the cost of performance). The LAM/MPI package supports a switch to its `configure` script – `--with-purify` that enables LAM/MPI to be entirely `bcheck` clean.

## 10.3 What to Include

The released version of your software should be delivered to the customer in such a form that she can readily use it. Typically, this will mean delivering a compressed tape archive file containing the source code for your application such that the application can be easily compiled and run by the end user. Tape archives are created using the `tar` command, and compressed with either the `compress`, `gzip`, or `bzip2` commands (RTFM on all of these).

We also publish software in RedHat Package Management (RPM) format for the various flavors of Linux. Creating RPMs are a bit more complicated, however, and unless your software is targeted specifically at Linux, it is not necessary to publish and RPM for it.

Your release should include, at the least:

- All necessary source code files
- A `configure` script for setting up architecture, OS, and compiler specific options in `Makefiles` and other project files
- All documentation
- Relevant application-level test programs
- Interesting examples

### 10.3.1 The `configure` Script

A `configure` script must be included with your package that determines all relevant facts from the underlying OS, compiler, and any other necessary applications and/or libraries. This script should use as many of the predefined `autoconf` `AC_` and `LSC_` tests as much as possible.

We have a standard set of `autoconf` tests that have evolved from writing many `configure` scripts – we found that we kept copying the relevant `configure.in` code from one project to another, and there was no central control. Hence, we have an LSC version of the `aclocal.m4` file that contains several standardized tests that you can readily import into your `configure.in` script. This `aclocal.m4` file is

in the same location as the LSC sample Makefiles, etc. See Table 4 for a list of the LSC tests. Descriptions of each of these tests are included below.

If you are using aclocal/automake instead of autoconf, this file needs to be named aclocal.am instead of aclocal.m4.

- LSC_CXX_TEMPLATE_REPOSITORY

  Finds the name of the template repository directory. The resulting name will be in the TEMPLATE_-REP shell variable.

- LSC_MPI_ERR_PENDING

  Checks to see if the underlying MPI uses MPI_PENDING or MPI_ERR_PENDING. At least one old implementation of MPI incorrectly used MPI_ERR_PENDING. The shell variable LSC_HAVE_-PENDING will be 1 if the underlying MPI uses MPI_PENDING, 0 if it uses MPI_ERR_PENDING.

- LSC_CHECK_MPI_H

  Checks to see if <mpi.h> can be found. If it is not found, AC_MSG_ERROR will be invoked to display an error message, and configure will abort.

- LSC_CHECK_LMPI

  Checks to see if libmpi.* can be found. If it is not found, AC_MSG_ERROR will be invoked to display an error message, and configure will abort.

- LSC_CHECK_CXX_BOOL

  Checks to see if the C++ compiler has the bool type. If it does, defines LSC_HAVE_BOOL to be 1, and sets the environment variable LSC_HAVE_BOOL to be 1. If the compiler does not have bool, both the macro and the environment variable will be set to 0.

- LSC_CXX_CHECK_DEPDIRS

  Checks to see what -I flags are necessary for makedepend (so that files like <string> and the like are found) such that dependencies can be generated for C++ source files. This macro will invoke LSC_CXX_HAS_STL first if it has not already been invoked.

  This macro tries a bunch of heuristics to find the location of the C++ header files. If you find a compiler that this macro does not work for, you are encouraged to update this macro. :-)

  The shell variable LSC_CXX_DEPDIRS will contain one or more space-delimited directories (each prefixed with -I) that need to be included on the makedepend command line in order for makedepend to find all the proper C++ header files. AC_SUBST is invoked on the same name.

- LSC_CHECK_CXX_EXCEPTION_FLAGS

  Checks to see what flags are necessary to compile exceptions support with the C++ compiler. The relevant flags are put into the shell variables LSC_EXCEPTION_CXXFLAGS and LSC_EXCEPTION_-LDFLAGS, and AC_SUBST is called on both.

- LSC_CHECK_CXX_EXCEPTIONS

  Checks to see if the C++ compiler supports exceptions. This really checks to see if the compiler supports the throw and catch keywords. If it does, the shell variable LSC_CXX_HAS_EXCEPTIONS is set to 1, otherwise it is set to 0.

| Test name | Description |
|---|---|
| LSC_CXX_TEMPLATE_REPOSITORY | Finds the name of the template repository directory. |
| LSC_MPI_ERR_PENDING | Checks to see if the underlying MPI uses MPI_PENDING or MPI_ERR_PENDING. At least one old implementation of MPI incorrectly used MPI_ERR_PENDING. |
| LSC_CHECK_MPI_H | Checks to see if <mpi.h> can be found. |
| LSC_CHECK_LMPI | Checks to see if libmpi.* can be found. |
| LSC_CHECK_CXX_BOOL | Checks to see if the C++ compiler has the bool type. |
| LSC_CHECK_CXX_DEPDIRS | Checks to see what -I flags are necessary for makedepend (so that files like <string> and the like are found) such that dependencies can be generated for C++ source files. |
| LSC_CHECK_CXX_EXCEPTION_FLAGS | Checks to see what flags are necessary to compile exceptions support with the C++ compiler. |
| LSC_CHECK_CXX_EXCEPTIONS | Checks to see if the C++ compiler supports exceptions. |
| LSC_CHECK_CXX_TRUE_FLASE | Checks to see if the C++ compiler has the true and false constants. |
| LSC_CHECK_SIGNAL_TYPE | Checks to see if the OS supports SYSV or BSD style signal handler declarations. |
| LSC_CXX_CHECK_NOTHORW_NEW | Checks to see if the C++ compiler supports new(std::nothrow). |
| LSC_CXX_HAS_STL | Checks to see if the C++ compiler has STL built in. |
| LSC_CXX_HAS_NAMESPACE | Checks to see if the C++ compiler has support for the namespace keyword. |
| LSC_CXX_HAS_STD_STL | Checks to see if the C++ compiler puts STL in the std namespace. |
| LSC_CXX_HAVE_STD_ALLOCATOR | Checks to see if the C++ compiler has the standard allocator. |
| LSC_CXX_HAVE_LIMITS | Checks to see if the C++ compiler has the <limits> header file. |

Table 4: Set of standardizes tests for autoconf/automake scripts. Additions to this list are welcome!

- LSC_CHECK_CXX_TRUE_FLASE

  Checks to see if the C++ compiler has the true and false constants. If it does, defines LSC_HAVE_TRUE_FALSE to be 1, otherwise it is defined to 0.

- LSC_CHECK_SIGNAL_TYPE

  Checks to see if the OS supports SYSV or BSD style signal handler declarations. The shell variable LSC_SIGNAL_TYPE will be set to either BSD or SYSV.

- LSC_CXX_CHECK_NOTHORW_NEW

  Checks to see if the C++ compiler supports new(std::nothrow). If it does, define LSC_NOTHROW_NEW to (std::nothrow), and defines LSC_HAVE_NOTHORW_NEW to be 1. If not, LSC_NOTHROW_NEW is defined to be empty, and LSC_HAVE_NOTHROW_NEW is defined to be 0.

- LSC_CXX_HAS_STL

  Checks to see if the C++ compiler has STL built in. The shell variable LSC_CXX_STL will be set to 1 if STL is found, or set to 0 otherwise. Additionally, AC_DEFINE is invoked on LSC_CXX_STL.

- LSC_CXX_HAS_NAMESPACE

  Checks to see if the C++ compiler has support for the namespace keyword. This also enables the "--with-namespace" option to configure to allow users to force the use of the namespace keyword (or not). Hence, you can specify --without-namespace to configure in order to force the test to result in saying that the compiler does not, in fact, have the namespace keyword.

  This option is specifically for the GNU compilers; while they *do* have the namespace keyword, its implementation is so broken that it's not worth using.

  If the compiler has namespaces, the shell variable LSC_CXX_NAMESPACE will be set to 1, otherwise it will be set to 0. AC_DEFINE will be invoked on the same name.

- LSC_CXX_HAS_STD_STL

  Checks to see if the C++ compiler puts STL in the std namespace. This function will first invoke LSC_CXX_STL and/or LSC_CXX_HAS_NAMESPACE if they haven't already been invoked, and only pass if the compiler has both the STL and support for namespaces.

  The shell variable LSC_CXX_STD_STL will be set to 1 if the STL is in the std namespace (and the compiler has STL and namespace support), otherwise it will be set to 0. AC_DEFINE will be invoked on the same name.

- LSC_CXX_HAVE_STD_ALLOCATOR

  Checks to see if the C++ compiler has the standard allocator. The Makefile macro LSC_CXX_HAVE_STD_ALLOCATOR is defined to 1 if the standard allocator is found, otherwise it is defined to 0.

  Additionally, the macro __STL_USE_OLD_SGI_ALLOCATOR is defined if the allocator is found. That is, corresponding .h.in files should contain the following line for this macro:

  ```
  #undef __STL_USE_OLD_SGI_ALLOCATOR
  ```

  The name __STL_USE_OLD_SGI_ALLOCATOR is used in SGI's implementation of the STL, which is why this test may define it. However, SGI's implementation does not use #if – it uses #ifdef. So

the macro must be defined or undefined – setting its value to 0 or 1 will not have the desired effect. Hence, the `.h.in` files should `#undef` this macro. If the test passes and `AC_DEFINE` is invoked on `__STL_USE_OLD_SGI_ALLOCATOR`, `configure` will change the `#undef` to `#define`.

- `LSC_CXX_HAVE_LIMITS`

  Checks to see if the C++ compiler has the `<limits>` header file. The Makefile macro `LSC_CXX_HAVE_LIMITS` will be set to 1 if the header file is found, otherwise it is set to 0.

#### 10.3.2 Some Notes About `Makefile`s

It is frequently necessary to set a number of compilation values that can be passed to the compilation process. These are usually set in the `Makefiles` of the project via the `AC_SUBST` macro in the `configure.in` file. A common mistake for programmers is to use their `configure` script to generate a `Makefile` in every subdirectory in their project. This is not a good strategy, because it does not allow for good code reuse.

A better idea is to have your `configure` script generate a top-level "include" file, named `make-defs`, or something similar. This top-level file can be included in every `Makefile` in the project.

Use the `make` directive `include` to include the top-level file in each of the project's `Makefiles`. This is usually paired with a `TOPDIR` macro that is hard-coded in each file. The top-level file should also include all the standard targets that are common to every directory. As described in other sections in this document, your `Makefiles` should have all the common targets ("`all`", "`install`", "`examples`", "`clean`", and "`distclean`", at the very least).

See Figure 3 for an example.

```
# Sample Makefile
# This is in the src subdirectory, so the top-level directory is ..

TOPDIR = ..

include ${TOPDIR}/config.mk
```

Figure 3: Typical `Makefile` that includes a top-level file (`config.mk`) with values substitued in from the `configure` script, as well as all the standardized targets.

#### 10.3.3 Generated `.h` Files

It is typical for the `configure` script to generate one or more `.h` files. It is usually sufficient to put all `configure`-generated values in a single file (for uniformity, and ease of locating that information), but there are times when splitting it into multiple files is acceptable.

It is, however, a *Very Bad Idea* to name your generated files `config.h` (for both libraries and applications, alike). This is because other programmer are likely to have followed this bad convention, such that "`#include <config.h>`" may not actually include the file that you think you are including.

Instead, name your file more descriptively – *project*`_config.h`, for example – is usually sufficient to uniquely name your file, and will guarantee that when you `#include` it, you are actually getting the file that you intend to include.

#### 10.3.4 Using AC_ Tests

If your software package is a library, you need to be careful about using the pre-defined AC_ tests that come with autoconf (etc.). Some of these tests will AC_DEFINE standard preprocessor macros that may be used in other packages. If your package and another package both #define the same preprocessor macro, you will get a warning when compiling.[15]

There are two solutions to this problem:

1. In the .h file that is generated, be sure to check with #ifndef before #defineing a macro:

   ```
   #ifndef HAVE_LIMITS_H
   #define HAVE_LIMITS_H 1
   #endif
   ```

   This subscribes to the theory that if you are using your package in conjunction with some other package that uses the same AC_ macro as you use, the other configure script will get the same result that you will, so you don't need to re-define the macro.

2. If, however, you are not comfortable assuming this, you can do the following:

   ```
   #ifdef HAVE_LIMITS_H
   #undef HAVE_LIMITS_H_
   #endif
   #define HAVE_LIMITS_H 1
   ```

Both of these methods, however, assume that your .h files will be included last. If some other .h file is included after yours that redefined the macros, you'll get warnings (unless those files are careful, too – which they generally are not). Unfortunately, there isn't too much that you can do about this. :-(

#### 10.3.5 Things to Not Include

What you should *not* include in the release tarball:

- Version control files and directories
- Module test programs
- Generated files — to include object files and library archives, Makefiles for which a Makefile.in is provided, .aux files, etc.
- Any development tool droppings, e.g., emacs backup or autosave files, template repositories, profiling output, etc.
- Internal notes files, such as TO-DO, NOTES, etc.

[15] Recall that all released LSC software must be warning-free.

## 10.4 Creating a Distribution

Creating a release should be an automated process (e.g., using `make` and/or a shell script) that creates a clean release directory structure and then creates the distributable file (the compressed `tar` file).

The general steps that should be followed to make a distribution tarball are:

1. Extract the latest sources from the CVS repository (use `cvs export`).
2. Obtain the version number from the source tree (you'll use it later).
3. Insert license/copyright headers in all files in the project.
4. Set social Unix permissions on all the files (0755 for directories, 0644 for files). Ensure that any executable scripts have the `x` bit set.
5. Run `autoconf` or `aclocal`/`automake` to generate a `configure` file.
6. Remove any excess files that should not be in the distribution tarball.
7. Rename the directory where the project was extracted to be of the form *projectname-versionnumber*.
8. Create a `tar` file of the entire tree; the `tar` filename should be of the form *projectname-versionnumber*`.tar`.
9. Create three files: one that is compressed, one that is `gzip`'ed, and one that is `bzip2`'ed. Copy these three files back to the directory where the distribution process was invoked from.
10. Remove any temporary directories and files that were used in this process; the only files that should be left over are the three compressed tarballs.

## 10.5 Verifying the Distribution

Before you actually release your application, you should go through all the steps the end user will go through to verify that everything unpacks and builds properly. This should also be an automated and thoroughly exhaustive process. That is, for each supported configuration of your application (architecture, OS, compiler), you should go through the entire unpack and build process and verify that your application can be built with no warnings or errors.

There is nothing more annoying (and more damaging to your credibility as a developer) than to have your distribution not build and run "out of the box." Tools such as `rsh`, `csh`, and `perl` are helpful for release verification.[16]

To prevent against this, your software should be run in every possible configruation, and should be tested such that every code path is actually exercised. This usually entails writing test programs to run a multitude of test cases, and individually report on all the failures (if there are a large number of test cases, you only want to see the failures, not the successes).

This reporting can take the form of many individual programs that return 0 (or not) and have a controlling shell script monitor each of the return values, or a smaller number of larger programs that report test failures themselves. Test case failures should probably be reported to `stderr`, so that they can be funneled into a separate file.

[16] We have some automated distributed testing tools for this, but nothing standardized yet.

## 10.6 Version Numbers

Version numbers are typically of the form *major.minor.release*. For the first major release of a version, *minor* is typically zero, and *release* is is typically omitted. For example, `1.0`, `1.0.5`, `6.3.12`, etc.

It is also helpful to have temporary version numbers when going through the final stages of creating a release. Running all the test compilations and tests usually will turn out some bugs. This means going and fixing the bugs, re-creating a distribution tarball, and re-running all the tests. A distribution is only releasable when when it passes *all* the tests.[17]

But it may get confusing if you keep generating and verifying a tarball by the same name, especially on multiple architectures. Hence, temporary version numbers are very helpful. They are typically of the form *major.minor.release*-b*X*. So when you make the first candidate tarball for release, it can be `project-1.0.3-b1.tar.gz`. When errors are found and corrected, bump up the beta number, and create `project-1.0.3-b2.tar.gz`.

Making new versions numbers will ensure that you actually test the new version instead of mistakenly testing an old tarball. When the final beta tarball passes all the tests, you can drop the -b*X* and re-create the distribution tarball.

## 10.7 Once It's Out the Door

In order for people to use your software, it has to be published. This is typically on the web. See Section 11 for more information on project web pages. However, once the software is published, users will tend to find bugs. While the procedures outlined in this section can certainly help to reduce the number of bugs, some will inevitably turn up when real users start using the software.

As such, it is important to use the `cvs tag` command to mark the current position in the CVS repository at the current version number. It should be marked with a tag name of the form v*major_minor_release* (CVS does not allow "`.`" in tag names). For example, `v1_0_3`. It is *not* necessary to put the project name in the tag name at all – this would be redundant, since the CVS repository already has the project name in it.

Using CVS tags will allow you to recall previous versions of the project if necessary. It can also be used to branch prior versions of the project to make bug fixes. RTFM on CVS to see how to do this.[18]

---

[17] Even the most minor change in the project's code mandates re-creating the distribution and re-running all the tests (it's the only way to guarantee that one change doesn't affect anything else – sorry).

[18] One important fact to know about creating branches (one that I learned the hard way) is that the sticky tag will stay with the branch for the *entire* branch. Be sure to pick a tag name that is sufficient for the whole branch, not just the first release version number.

# 11 Project Web Pages / HTML and PHP3

To be written. Got a bunch to say about the use of PHP3.

Topics to include:

- Online vs. offline pages
  - Why do we do online vs. offline
  - How to access offline pages (must use our server to get proper PHP access)
  - CVS access to LSC pages (some specials, such as `tlc` project)
  - AFS permissions of CVS pages
  - Distribution tarball location – not within HTML tree
  - MailArchive trees. Migrate to `listserv.nd.edu`
- HTML
  - Never forget to close tags
  - Do not use SHTML
  - Style sheets
  - Indent / use proper whitespace for readability
  - Make HTML tags be UPPERCASE
  - Use full `HTML`, `HEAD`, and `BODY` tags
  - Use `META` tags if possible
  - `HREF`, `SRC`, etc. values should be in quotes
  - *Always* use relative links when referring to pages on the same site. *Only* use absolute links when referring to pages that are on another site (to include using a different server name for our server, e.g., a page on `www.lsc.nd.edu` referring to something on `www.mpi.nd.edu`). *NEVER NEVER NEVER* use absolute links to refer to a page on the same site.
  - Use of tables
    - Only way to ensure alignment
    - Basic stlye (`TABLE, TR, TD, /TD, /TR, /TABLE` each on separate lines)
    - Don't forget `align` and `valign`
  - HTTP redirects – use the right web server name (use PHP for this, see below). Only valid names are `www.lsc.nd.edu` and `www.mpi.nd.edu`.
  - Basic scheme of most LSC pages: title bar, left hand navigation, right hand content, footer
- PHP
  - What is / why to use PHP3
  - Use of `version.php3` and < `print($ver);` > to hold/print project version numbers
  - Show MD5 sum for all downloadable files
  - Use perl or C++ mode in emacs for editing PHP files
  - Use of PHP `global` keyword

- Use of `$topdir` – hard coded in each file, and only used to go "backwards" in links. Use normal relative links for going "forwards".
- Never invoke printable code by including, must call a function
- Never end a file with "`?>`"
- It's a programming language, indent and use proper whitespace
- Use of common headers and footers
  - Basic table layout of pages
  - Basic scheme – include header, have content, include footer.
  - Header should do all table things, to include: top title bar, left hand navigation, and open cell for content of page.
  - Footer should close content cell and have a contact e-mail address (or URL) and a dynamic copyright notice.
  - If a mirrored page, footer should have a flag showing where the site is located.
- Use of `force_server()` LSC PHP function
- Searching LSC web pages using `search.nd.edu`
- Showing source code via PHP – `show_src.php3`

- CGI
  - Tend not to use CGI – use PHP as much as possible.
  - When need to process a form, have the ÂCTION be a `.php3` file that does the action, and then does an HTTP redirect to the appropriate result page (or prints results itself).
  - If an HTTP redirect is the result, must be careful not to output anything before the HTTP redirect.

- Database access
  - Perfect way to access large quantities of data in a standardized way (e.g., test results, FAQ questions, etc.)
  - mSQL running on `lsc.nd.edu`
  - May be quicker to use PHP generate static pages for online viewing; dynamic database access should probably be restricted to data maintainers only
  - Some projects have been implemented in this way (MPI list, LAM FAQ), but nothing standardized yet
  - Mini-databases for small amounts of standard information: text files. Parse these to show tables of information. e.g., the LAM OS/hardware lists, the LAM mirror sites, etc. Much easier to edit the text field files than HTML, and using a standard PHP routine to print out the data guarantees consistency.

- HTTP mirroring
  - Not as easy as it sounds
  - Use of GNU `wget` tool.
  - *POSITIVELY CANNOT USE ABSOLUTE URLs* to refer within a site.

- Apache web server does not understand how to output HTTP header `last-modified` for PHP3 files – must manually generate it and output it as part of the standardized header. Only necessary to calculate this date/time when a mirror is coming through – not for every user (takes too long).
- Can always output US flag unless a mirror is coming through. Be sure to output "unauthorized mirror" if not in mirror list
- Use HTTP agent field to identify the mirror to the PHP3 mirroring code – the correct generation of the HTTP `last-modified` and correct flag at bottom.

# 12 Where to Learn More

## 12.1 Resources in Print

The following books are software development and language references that we have found to be particularly useful.

Frederick P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, 1995.

James O. Coplien. *Advanced C++ Styles and Idioms*. Addison-Wesley, 1991.

Bill Cureton. *Software Engineering on Sun Workstations*. Springer-Verlag, 1993.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2 edition, 1988.

Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.

Steve McConnell. *Code Complete*. Microsoft Press, 1993.

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

## 12.2 Resources on the Web

NASA's Goddard Space Flight Center provides a number of software engineering resources on their Software Engineering Laboratory web site at the following URL:

`http://fdd.gsfc.nasa.gov/seltext.html`

Included at this site are guides for the software development process (*Recommended Approach to Software Development*, SEL-81-305) and for C style (*C Style Guide, SEL-94-003*).

The Software Engineering Institute has a web site at

`http://www.sei.cmu.edu/`

The ANSI C++ Standard can be found at

`http://www.cygnus.com/misc/wp/`

# References

[1] Frederick P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, 1995.

[2] James O. Coplien. *Advanced C++ Styles and Idioms*. Addison-Wesley, 1991.

[3] Bill Cureton. *Software Engineering on Sun Workstations*. Springer-Verlag, 1993.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2 edition, 1988.

[5] Steve Maguire. *Writing Solid Code*. Microsoft Press, 1993.

[6] Steve McConnell. *Code Complete*. Microsoft Press, 1993.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

# A The Ten Commandments for C Programmers (Annotated Edition)

by Henry Spencer

## A.1 Lint

*Thou shalt run lint frequently and study its pronouncements with care, for verily its perception and judgment oft exceed thine.*

This is still wise counsel, although many modern compilers search out many of the same sins, and there are often problems with lint being aged and infirm, or unavailable in strange lands. There are other tools, such as Saber C, useful to similar ends.

"Frequently" means thou shouldst draw thy daily guidance from it, rather than hoping thy code will achieve lint's blessing by a sudden act of repentance at the last minute. De-linting a program which has never been linted before is often a cleaning of the stables such as thou wouldst not wish on thy worst enemies. Some observe, also, that careful heed to the words of lint can be quite helpful in debugging.

"Study" doth not mean mindless zeal to eradicate every byte of lint output – if for no other reason, because thou just canst not shut it up about some things – but that thou should know the cause of its unhappiness and understand what worrisome sign it tries to speak of.

## A.2 NULL

*Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.*

Clearly the holy scriptures were mis-transcribed here, as the words should have been "null pointer", to minimize confusion between the concept of null pointers and the macro NULL (of which more anon). Otherwise, the meaning is plain. A null pointer points to regions filled with dragons, demons, core dumps, and numberless other foul creatures, all of which delight in frolicing in thy program if thou disturb their sleep. A null pointer doth not point to a 0 of any type, despite some blasphemous old code which impiously assumes this.

## A.3 Type-Casting

*Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.*

A programmer should understand the type structure of his language, lest great misfortune befall him. Contrary to the heresies espoused by some of the dwellers on the Western Shore, `int' and `long' are not the same type. The moment of their equivalence in size and representation is short, and the agony that awaits believers in their interchangeability shall last forever and ever once 64-bit machines become common.

Also, contrary to the beliefs common among the more backward inhabitants of the Polluted Eastern Marshes, `NULL' does not have a pointer type, and must be cast to the correct type whenever it is used as a function argument.

(The words of the prophet Ansi, which permit NULL to be defined as having the type `void *', are oft taken out of context and misunderstood. The prophet was granting a special dispensation for use in cases of great hardship in wild lands. Verily, a righteous program must make its own way through the Thicket Of Types without lazily relying on this rarely-available dispensation to solve all its problems. In any event, the great deity Dmr who created C hath wisely endowed it with many types of pointers, not just one, and thus it would still be necessary to convert the prophet's NULL to the desired type.)

It may be thought that the radical new blessing of "prototypes" might eliminate the need for caution about argument types. Not so, brethren. Firstly, when confronted with the twisted strangeness of variable numbers of arguments, the problem returns... and he who has not kept his faith strong by repeated practice shall surely fall to this subtle trap. Secondly, the wise men have observed that reliance on prototypes doth open many doors to strange errors, and some indeed had hoped that prototypes would be decreed for purposes of error checking but would not cause implicit conversions. Lastly, reliance on prototypes causeth great difficulty in the Real World today, when many cling to the old ways and the old compilers out of desire or necessity, and no man knoweth what machine his code may be asked to run on tomorrow.

## A.4 Header Files

*If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.*

The prophet Ansi, in her wisdom, hath added that thou shouldst also scourge thy Suppliers, and demand on pain of excommunication that they produce header files that declare their library functions. For truly, only they know the precise form of the incantation appropriate to invoking their magic in the optimal way.

The prophet hath also commented that it is unwise, and leads one into the pits of damnation and subtle bugs, to attempt to declare such functions thyself when thy header files do the job right.

## A.5 String Bounds

*Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest "foo" someone someday shall type "supercalifragilisticexpialidocious".*

As demonstrated by the deeds of the Great Worm, a consequence of this commandment is that robust production software should never make use of gets(), for it is truly a tool of the Devil. Thy interfaces should always inform thy servants of the bounds of thy arrays, and servants who spurn such advice or quietly fail to follow it should be dispatched forthwith to the Land Of Rm, where they can do no further harm to thee.

## A.6 Error Codes

*If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.*

All true believers doth wish for a better error-handling mechanism, for explicit checks of return codes are tiresome in the extreme and the temptation to omit them is great. But until the far-off day of deliverance cometh, one must walk the long and winding road with patience and care, for thy Vendor, thy Machine, and thy Software delight in surprises and think nothing of producing subtly meaningless results on the day before thy Thesis Oral or thy Big Pitch To The Client.

Occasionally, as with the ferror() feature of stdio, it is possible to defer error checking until the end when a cumulative result can be tested, and this often produceth code which is shorter and clearer. Also, even the most zealous believer should exercise some judgment when dealing with functions whose failure is totally uninteresting... but beware, for the cast to void is a two-edged sword that sheddeth thine own blood without remorse.

## A.7 Libraries

*Thou shalt study thy libraries and strive not to reinvent them without cause, that thy code may be short and readable and thy days pleasant and productive.*

Numberless are the unwashed heathen who scorn their libraries on various silly and spurious grounds, such as blind worship of the Little Tin God (also known as "Efficiency"). While it is true that some features of the C libraries were ill-advised, by and large it is better and cheaper to use the works of others than to persist in re-inventing the square wheel. But thou should take the greatest of care to understand what thy libraries promise, and what they do not, lest thou rely on facilities that may vanish from under thy feet in future.

## A.8 Braces

*Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.*

These words, alas, have caused some uncertainty among the novices and the converts, who knoweth not the ancient wisdoms. The One True Brace Style referred to is that demonstrated in the writings of the First Prophets, Kernighan and Ritchie. Often and again it is criticized by the ignorant as hard to use, when in truth it is merely somewhat difficult to learn, and thereafter is wonderfully clear and obvious, if perhaps a bit sensitive to mistakes.

While thou might think that thine own ideas of brace style lead to clearer programs, thy successors will not thank thee for it, but rather shall revile thy works and curse thy name, and word of this might get to thy next employer. Many customs in this life persist because they ease friction and promote productivity as a result of universal agreement, and whether they are precisely the optimal choices is much less important. So it is with brace style.

As a lamentable side issue, there has been some unrest from the fanatics of the Pronoun Gestapo over the use of the word "man" in this Commandment, for they believe that great efforts and loud shouting devoted to the ritual purification of the language will somehow redound to the benefit of the downtrodden (whose real and grievous woes tendeth to get lost amidst all that thunder and fury). When preaching the gospel to the narrow of mind and short of temper, the word "creature" may be substituted as a suitable pseudo-Biblical term free of the taint of Political Incorrectness.

## A.9 Identifiers

*Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.*

Though some hasty zealots cry "not so; the Millennium is come, and this saying is obsolete and no longer need be supported", verily there be many, many ancient systems in the world, and it is the decree of the dreaded god Murphy that thy next employment just might be on one. While thou sleepest, he plotteth against thee. Awake and take care.

It is, note carefully, not necessary that thy identifiers be limited to a length of six characters. The only requirement that the holy words place upon thee is uniqueness within the first six. This often is not so hard as the belittlers claimeth.

## A.10 Portability

*Thou shalt forswear, renounce, and abjure the vile heresy which claimeth that "All the world's a VAX", and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.*

This particular heresy bids fair to be replaced by “All the world's a Sun” or “All the world's a 386” (this latter being a particularly revolting invention of Satan), but the words apply to all such without limitation. Beware, in particular, of the subtle and terrible “All the world's a 32-bit machine”, which is almost true today but shall cease to be so before thy resume grows too much longer.