UGA Software Standards

Matthias M. Wloka, Nate Huang, and D. Brookshire Conner

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-01
January 1992

UGA Software Standards*

Matthias M. Wloka† Nate Huang† D. Brookshire Conner†

January 13, 1992

 This work was supported in part by NSF, DARPA, IBM, Sun Microsystems, NCR, Hewlett Packard and Digital Equipment Corporation.

† Department of Computer Science, Box 1910, Brown University, Providence, RI 02912

Contents

Chapter 1

Introduction

This document describes the Brown Graphics Group's Software Standards. These standards are not derived from a whim of the authors, but rather represent the experience of the Graphics Group over the past decade.

1.1 Software Standards in General

As soon as more than one programmer is working on a project, software standards are the only way to ensure that the software will work together without conflict. Software in excess of 2000 lines of code becomes difficult to maintain if no software standards are employed. The problem is amplified when the person maintaining the code is not the original author and in the worst case results in a complete rewrite; a potential waste of several man-months. Furthermore, enforcing software standards makes it easier for external groups to understand imported source code.

Often, software must be ported across architectures, a time-consuming task. Steps may be taken, however, to make software architecture-independent, saving considerable amounts of time.

A convincing reason for a programmer to employ software standards is the simplification of debugging. Errors resulting from messy code are not only unnecessary, but also hard to find. Errors resulting from bad function-interface design are extremely hard to trace. Errors resulting from non-standard usage of resources make programs dependent on cosmic rays. Software standards do not prevent errors; they do, however, make it easier to correct errors. This is particularly true when programming in C, which is considered a low-level language that is far less stringent in all aspects of good programming style than, say, MODULA-2 [WIRT82].

1.2 Software Standards in the UGA Environment

In the Brown Graphics Group's programming environment, the above mentioned conditions are met: the group is large (more than 20 programmers), with everyone working within a single framework. Most projects undertaken within that framework are team efforts of two to four people working closely together.

Individual programmers (i.e., students) only stay with the Brown Graphics Group for one to five years. Software maintenance is therefore seldom done by the original author, yet, the software needs to be continuously maintained over long periods of time.

The Brown Graphics Group always maintains a number of different architectures due to our multiple industrial sponsors. In addition, sponsors typically change their architecture over time. Therefore, software produced by the Brown Graphics Group needs to be portable.

Currently, the UGA framework consists of more than 100,000 lines of C source code, making it a large software project. Even though safeguards are in place [STRA91], it is clear that the UGA framework depends on the discipline of every single programmer, as every programmer is able to affect the work of everyone else.

Frequently, UGA source code is shipped to other institutions. It is embarrassing for the Brown Graphics Group if the code is poorly documented, poorly written, or hard to read and understand. Without software standards, it is difficult to reuse code outside the Brown Graphics Group environment.

1.3 Overview

This document concerns itself more with mundane things like indenting patterns and naming conventions than with the mechanics of the software environment. The reader is assumed to be familiar with the programming environment of the Brown Graphics Group [STRA91] and is expected to be able to program in C [KERN88].

This document is structured to resemble the software development process. We start by introducing existing software: the file `opsys.h` and the STD (STanDard) package. The general style of code written within the UGA system is discussed next, before guidelines on naming conventions and commenting style are presented. The chapter on formatting style sets forth rules on how source code is to be formatted. Also included in this document are standards for the external documentation of source code, i.e., man pages and more detailed design documents. We conclude by describing the mechanisms used to ensure that all C code placed in the UGA hierarchy adheres to these standards.

Chapter 2

Base-level Software

The "base-level" of UGA refers to:

- Architecture-dependent definitions to handle low-level operating systems issues.
- Common functionality that is used throughout the UGA hierarchy.

The purpose of this chapter is to present the base-level of UGA to the reader. It is essential for the reader to become accustomed to the concepts, since they will be repeatedly used throughout UGA system development.

2.1 The `opsys.h` Include File

The file `opsys.h` contains architecture-dependent definitions that all packages and commands use. It makes low-level system definitions transparent to the programmer. These definitions include basic data type sizes and alignment, memory and binary data utility routines and I/O routine declarations. Should a particular architecture lack a standard systems feature, then `opsys.h` is used to define it.

A good example of the use of `opsys.h` concerns the DEC operating system Ultrix 4.2. The Unix `strdup()` routine is commonly used to duplicate strings. However, Ultrix 4.2 does not define `strdup`. Thus, `opsys.h` contains the lines in Figure 2.1.

```
#ifdef ultrix
#   define strdup(a) strcpy(malloc(strlen(a)+1), a)
#endif
```

Figure 2.1: Sample architecture-dependent code in `opsys.h`.

Every package and command includes the file `opsys.h` before including any other files.[1] The file `opsys.h` itself includes `<stdio.h>`, `<unistd.h>`, `<stdlib.h>`, `<sys/types.h>`, `<fcntl.h>`, `<netinet/in.h>`, `<signal.h>`, and `<time.h>`. It is therefore not necessary to include the above header files again in your files.

[1] The UGA programmer need not explicitly include it, as it is included in `std.h`, explained in the next section.

The file `opsys.h` also defines common data types and macros to be used by all C programmers. These definitions are architecture-independent and are guaranteed to pass `lint`. All programmers are required to use these definitions to maintain software portability and maintenance. A list of these definitions with explanations on how and when to use them follows.

2.1.1 Data Types

`UGAgeneric` defines a generic data type for a given architecture.

`UGAptr` defines a generic pointer for an architecture, a pointer to the generic data type. A generic pointer is used when a function expects an arbitrary data type as, for example, the function `memcpy()` or arbitrary callback functions do.

2.1.2 Macros

`UGA_CAST(TYPE, PTR)` is used when casting an `UGAptr` to another pointer type to avoid "possible pointer alignment" compiler warnings. (For example, an integer must begin at an word-aligned address, while an `UGAgeneric` need not.) Figure 2.2 shows an example of its use.

```
int     *foo;
UGAptr  bar;

foo = (int *) bar;          /* This generates a lint error */
foo = UGA_CAST(bar, int);  /* This passes lint            */
```

Figure 2.2: Example of the use of `UGA_CAST()`.

`CEXTERN(RET, FUNC, ARG)` is used instead of the `extern` declaration in public include files. The value `RET` is the return type of the function of name `FUNC`. The parameter `ARG` is a comma-separated list of the data types of the arguments of that function. It is necessary to include the data types of the function arguments for ANSI C and C++ function prototyping.[2]

`brcase` is defined as `break; case`. This abbreviation is to be used in `switch` statements to save space. Similarly, `brdefault` is defined to be `break; default`. Thus, the two segments of code in Figure 2.3 are equivalent. These macros are convenient, but not mandatory. Note that using `brcase` on the first case generates a "statement never reached" lint error.

2.2 The STD Package

The STD package is the base-level package into which the most common and useful functionality of the UGA hierarchy is placed. It contains macros for memory management, debugging, accessing environment variables, error messages, hash tables, string pools, temporary files, and other common tasks. These are each discussed in detail below.

[2]Each package will have a list of these `CEXTERN` macros for its public and private routines. These are generated automatically by the "externs" target of every package's `Makefile`.

```
switch (i) {
   case  0: do_this();
            break;
   case  1: do_that();
            break;
   default: do_nothing();
}

switch (i) {
     case  0: do_this();
   brcase  1: do_that();
   brdefault: do_nothing(); /* Saved two lines of code */
}
```

Figure 2.3: Example of the use of `brcase` and `brdefault`.

Every package in UGA must use the STD package. The initial step is to call `STDinit()` to make sure STD is initialized. Next, `STDnew_glob()` is called to register a package with STD. This function returns a pointer to a `STDglob` structure. This `STDglob` structure allows memory allocation and error information to be identified on a per-package basis, and is referenced subsequently in calls to STD. Therefore, every package should have its own `STDglob` pointer. Furthermore, this `STDglob` must be aliased to `STD__GLOB` in the package's `defs.h` file, since many STD macros will reference `STD__GLOB`. Since each package will have its own `STDglob` registered, STD can track memory allocations or turn tracing on and off on a package by package basis. A package's `done()` routine should call `STDdone()`.

2.2.1 Memory allocation and freeing

Allocation and freeing of memory are perhaps the most common operations in UGA. The BAGS (Brown Animation Generation System) hierarchy — BAGS is the predecessor of UGA — allowed memory allocation by FALLOC, MEM, FO, and macros defined (identically) in multiple packages. These packages do not exist in UGA. Instead, the STD package unifies these disparate methods for memory management.

In the public include file `std_util.h`, one will find a set of macros for memory management. This file is automatically included in `std.h`. *These macros are the only means by which memory should be allocated or freed in UGA.* The macros are:

- `STD_ALLOC()`, `STD_ALLOCM()`, `STD_ALLOCN`, `STD_ALLOCNM()` allocate memory.
- `STD_CALLOCN()` and `STD_CALLOCNM()` allocate and clear memory.
- `STD_FREE()` and `STD_FREEN()` free memory.
- `STD_REALLOCN()` and `STD_REALLOCNM()` reallocate memory.
- `STRDUP()`, and `STRDUPM()` duplicate strings. `STD_FREE_STR()` frees a string.

The `ALLOC` and `FREE` macros are used as a pair. That is, if memory is allocated by one of the `ALLOC` macros, then it must be freed with one of the `FREE` macros. If memory is obtained through

REALLOC, then only REALLOC should be used to allocate and free it. A pointer should never be passed to both REALLOC *and* ALLOC or to both REALLOC *and* FREE. This rule is essential for memory statistics to be tracked accurately. Figure 2.4 shows the proper use of REALLOC.

```
int *ptr;

STD_REALLOCN(ptr, int, 1, 0); /* Allocates 1 integer        */
    ...
STD_REALLOCN(ptr, int, 5, 1); /* Reallocates to 5 integers */
    ...
STD_REALLOCN(ptr, int, 0, 5); /* Frees the memory          */
```

Figure 2.4: Correct usage of REALLOC to allocate and free memory.

The above ALLOC and FREE macros either call `malloc()` and `free()` directly, or use the functionality in `STDmem.c` for faster allocation and freeing. Typically, the faster allocation and freeing should be used, since they are implemented very efficiently with preallocated free lists of variable-sized data records, thus replacing calls to `malloc()` with simple list management.[3]

To access the faster macros, the line

```
#define STD__FAST_MEM
```

should be included in a package's `defs.h` file *before* including `std.h`. If that `#define` is absent, then the normal calls to `malloc()` and `free()` will be used.

2.2.2 Debugging Aids

Independently of which type of allocation and freeing routines are used, STD automatically keeps statistics regarding the quantity and sizes of any allocations and frees in debugging versions of code, and provides routines to print tables and histograms of memory allocation. This management can even be done on a package-by-package basis.

Very extensive tracking of allocations and frees can be turned on by setting the environment variable UGA_TRACK before running an application (or setting the variable `UGA_track` to be non-zero inside a debugger). Enabling extensive tracking will associate pointer addresses with memory sizes in a hash table, and associate data type names with numbers of allocations in a string pool. The hash table prevents the same address from being allocated or freed more than once, while the string pool tracks how many structures of a particular type are currently allocated. The hash table and the string pool can also be printed out.

Tracing can be enabled and disabled on a per package basis by calling the functions `STDtrace()`, `STDtron()`, and `STDtroff()`. In addition, the environment variables UGA_TRON and UGA_TROFF may be set to a list of package names to control tracing, e.g., `setenv UGA_TRON 'trip cot'` will enable tracing for the TRIP and COT packages. If tracing is turned on for a package, a message will be printed to `stdout` whenever a routine in that package is entered.

[3] REALLOC currently does not take advantage of this fast allocation.

2.2.3 Error Handling

STD provides standardized error handling and error message macros. There are various macros for the different levels of errors that may occur in a program. They are:

- `STD_INFO()` is used for general information messages.
- `STD_WARNING()` is used for minor errors that can still be ignored.
- `STD_SEVERE()` is used for unexpected errors that can affect program execution.
- `STD_FATAL()` is used for fatal errors. The program will exit.

Each error macro is passed a piece of text. The text is itself enclosed in parenthesis and consists of a comma separated pair: a *format* and an *arg-list*. The possible formats are listed in `std.h`. The contents of the string must correspond to the format. For example, if the format is `STD_ERR_SS`, then the arg-list is expected to be two strings separated by a comma. If the format is `STD_ERR_SI`, then the arg-list is expected to be a string and an integer separated by a comma.

There are also macros for errors that will only print out a single string. These are `STD_INFOS()`, `STD_WARNINGS()`, `STD_SEVERES()`, and `STD_FATALS()`. An example of the usage of the `STD_FATALS()` macro is `STD_FATALS("Time to die")`.

2.2.4 Hash Tables and String Pools

Packages commonly need to associate specific data with a particular number or string. STD provides support for both cases. The functionality to associate arbitrary data with a number is located in `STDhash.c`. The functionality to associate arbitrary data with a character string is in `STDstrpool.c`. See the STD man page for more information.

2.2.5 Environment Variables

UGA contains a number of environment variables that are used by various packages. STD provides routines for accessing these variables. The enumerated type `STDenv_var` enumerates the UGA environment variables that may be accessed. The file `STDenv.c` contains routines to access these variables, including `STDenv_get()`,`STDenv_is_set()`,`STDenv_set()`,and others.

2.2.6 Temporary files

Certain packages in UGA require the creation of temporary files in the `/tmp` directory on the workstation that the application is executing on. STD provides a common mechanism for packages to create temporary files through the function `STDenv_create_temp_file()`. Given a file name, this function creates that file in the `/tmp` directory and registers it with STD. Once created, the client does not need to worry about removing the temporary file — it will be removed automatically when the application is finished (and `STDdone()` is called).

2.2.7 Math Macros

STD supplies macros for all of the standard math library functions. These STD macros cast all parameters to `(double)` as expected by the math library functions. The return values are cast to `STDreal`. These macros make the UGA code cleaner. The STD man page lists these macros.

2.2.8 Common Macros and Constants

STD contains the following useful macros and constants.

True/False constants STD_TRUE, STD_FALSE, STD_YES, STD_NO, STD_ON, STD_OFF, STD_SUCCESS, and STD_FAILURE.

Null pointer constants STD_CNULL, STD_INULL, STD_DNULL, and STD_PNULL.

Special characters STD_SPACE, STD_NEWLINE, STD_TAB, and STD_ZERO.

String equality macro STD_STREQL() and STD_STRNEQL().

Common macros STD_ABS(), STD_MIN(), and STD_MAX().

File I/O macros STD_FOPEN(), STD_FOPENM(), STD_CLOSE(), and STD_CLOSEM().

2.2.9 Localization

STD will define "local" names for the STD macros and constants so they can be referred to with ALLOC, WARNING, and TRUE instead of STD_ALLOC, STD_WARNING, and STD_TRUE. The file std_local.h contains localized names for all the STD macros and constants. This file is automatically included when std.h is included.

2.3 Makefiles

This section briefly describes the techniques used to actually compile code and generate man pages. For a complete list of make targets as defined by UGA, see "The UGA Programming Environment"[STRA91].

UGA uses the gnumake program from the GNU Foundation. This allows the same mechanisms to be used on every architecture the Brown Graphics Group supports. See the gnumake manual [GNUMAKE] for information about how gnumake works.

Each package's source directory contains a file Makefile. This file specifies variables for gnumake such as the following:

- C_FILES: Lists the .c source files for the package.
- PUB_INC_FILES: Lists the include files that will be made available as public header files.
- INC_FILES: Lists the include files that are only used in this package.

The following gnumake targets are used to compile the source files in a package and create an archived library:

- gnumake externs
  This creates two files, public.h and private.h which contain all the external function declarations (in the form of CEXTERN macros) for a package. The file public.h contains extern declarations for all public routines. The file private.h contains extern declarations for all private routines. If in the course of compiling (see below) the compiler produces an error in one of these files, then a function was declared incorrectly. For instance, if a static function is declared in the public section, then the generated public.h will be incorrect and the compiler will report which line in public.h generates an error. This in turn tells the programmer which function header is wrong.

- `gnumake dep`
  This runs the command `bmakedep` on all the source files and creates a file `Makefile.dep` containing the dependencies for source files in the package. By maintaining these dependencies, a source file will recompile only if a change affecting it has occurred since the last compilation.

- `gnumake lint`
  This runs `lint` [DARW90] on all the source files.[4] All source code must pass `lint` quietly.[5]

- `gnumake Library`
  This compiles the C files and makes an archived library out of the resulting object files. As with `lint`, the source code must compile cleanly. "Possible pointer alignment" warnings should not be ignored.

- `gnumake man`
  This generates a man page for the package. The `README` file in the package's source directory gives a general description of the package, a description suitable for inclusion at the top of the man page. The `README` file also lists known bugs, diagnostics, and authors. The remainder of the man page is generated from the source file comments, as described in Chapter 4.

[4] Currently, this can only be done on the DEC architecture: `lint` currently does not work on the Sun Workstations, since no ANSI C compatible `lint` is supplied with them.

[5] Except for "Function returns value that is always/sometimes ignored." messages on the DECstations. These are unavoidable.

Chapter 3

General Coding Style

While developing software, the programmer has to face many decisions. This chapter helps the programmer make these decisions by supplying a set of guidelines. These guidelines include a general description of how software should be developed as well as how issues of efficiency and portability should be resolved.

3.1 The Software Development Process

Software is developed by designing it on paper first. The design phase decides the high level algorithms to be used and the logical structure imposed on the solution. While structuring a project, the software developer should be guided by an object-oriented methodology, even if the implementation will be in C. The chosen logical structure should indicate how functionality will be grouped into separate source files. It should also provide enough detail to allow the problem to be broken down into separate, logical parts which will directly correspond to functions in the implementation.

Each major function is then implemented by first writing pseudo-code for it. This process will suggest possible subroutines to the programmer. After all the subroutines have been written and the pseudo-code is translated into a compilable form, `lint` [DARW90] should be run on the software. The programmer should compile and attempt to test the software only after it passes `lint` quietly.

The above process of pseudo-coding, writing, `lint`ing, and testing of code should be performed on a function-by-function basis, since it ensures that only a small number of mistakes is introduced for each iteration. More importantly, since mistakes will be localized in a small portion of the code, finding and correcting them will be easier for the programmer.

3.2 Efficiency

Efficiency can mean being maintenance-efficient (reliable, extensible, and easy to debug code), execution-time efficient, or memory efficient. All three kinds of efficiency are desirable in a program. Normally, however, making code more efficient will involve trade-offs. For example, one can choose a very complicated, but fast algorithm over a simple, but slower one, thereby trading execution speed for maintainability. Similarly, it is sometimes possible to reduce execution time by using more memory. However, the programmer should be aware that excessive memory usage will result in page swapping, which in turn will slow down a computation.

We advise the programmer to optimize for maintainability and speed. When speed is not absolutely critical, a general and straightforward algorithm should be chosen over a convoluted special case.

If an implementation is not fast enough or uses too much memory, then the code should be profiled (see the various profiling programs, e.g., `gprof(1)`). Changes to the code should only be made based on the profile data, never because the implementer thinks or feels that a certain change would optimize the code. The programmer should also refrain from making peephole optimizations or declaring register variables: modern compilers are much better than humans in this respect. In particular, the readability of code should not be compromised by using constructs like `*(p+i)`, instead of `p[i]`.

The most efficiency is often gained by choosing a different high-level algorithm. Changing algorithms will probably involve rewriting large parts of already working code. To avoid such a waste of man-hours, it is essential to allow efficiency considerations to influence the design phase of the project. The usage patterns and scope of the project need therefore to be studied carefully, even before the design begins. Please refer to the book by Bentley [BENT82] for much more on efficient programming.

3.3 Portability

The philosophy adapted for UGA is to be able to port the system to any architecture without modifying source code at all. This philosophy is not only sensible for a 100,000 line source code system like UGA, but also realizable as long as all programmers follow the guidelines of this report. In particular, no file names or directory structures should be assumed and environment variables should be used instead (as already discussed in Section 2.2). Since different architectures use different byte orders, writing binary data should be avoided or at least be done by using the BYTEORDER package (see `byteorder(3N)`).

Other rules that aid portability while improving readability are:

- The return type of a function should always be declared explicitly.
- The success or failure of a function should be returned as `SUCCESS` and `FAILURE` (see Section 2.2), respectively.
- Global variables should be avoided. For example, always passing a "handle" structure makes global variables obsolete. If global variables are unavoidable, they should be declared as static to confine them to a single file.
- Pointer structures should not be dereferenced directly. Instead accessor macros should be used. For example, use `NEXT(list_elem)` instead of `list_elem->next`.
- Type-conversions, including function parameter conversions, should be made explicit by the cast operator.
- All variables should be initialized explicitly.
- All constants should be named, i.e., code should not contain magic numbers.

A helpful tool to make programs portable is `lint` [DARW90]. A program that does not pass `lint` quietly, will most likely not be portable across architectures. For this reason, we require that all code passes `lint`.

The `lint` tool will flag inconsistencies in a program even when those are unavoidable. For example, non-void functions whose return values are not used or unused function parameters will generate warnings. These should be fixed by explicitly casting non-void functions to `void` and using the `/*ARGSUSED*/` `lint` directive, respectively. The programmer should know about all the `lint` directives (see the `lint` man page) and use them where appropriate.

Chapter 4

Naming Conventions

4.1 File Names

There are many files in the UGA software environment. Having a predictable structure for files is therefore important. This structure should be evident both outside the file (when listing the directory) and inside the file (when looking at its contents).

All code files that are part of a package should begin with the name of the package in all capitals. The remainder of the file name should be all lower-case, with underscores for spaces. If the file contains public functions or methods, there should be no underscore between the package name and the remainder of the file name. If the file does not contain *any* public functions or methods, it should have an underscore between the package name and the remainder of the file name. Lisp source files should use dashes instead of underscores.

Files must have suffixes indicating their type. Table 4.1 summarizes file suffixes based on file type.

Suffix	Type	Suffix	Type
`.c`	C source	`.s`	Assembler
`.h`	C header	`.el`	Emacs-Lisp
`y.y`	Yacc	`.lisp`	Common Lisp
`l.l`	Lex	`.tex`	TeX or LaTeX
`.C`	C++ source	`.texinfo`	TeXinfo or LaTeXinfo
`.H`	C++ header	`.#`	man pages (# is section)
`Y.Y`	Yacc file containing C++	`.csh`	`csh` scripts
`L.L`	Lex file containing C++	`.uil`	`uil` scripts

Table 4.1: Summary of file suffixes and their corresponding file types.

4.1.1 Standard Files

All packages need to initialize themselves by creating a `STDglob` (see the STD man page), as well as any global variables. The routine to perform this action should be found in a file consisting of the package's name, followed by the suffix indicating the language the file is written in, such as `TRIP.c` for the TRIP package. A smaller number of packages will have globals that can be used by other packages without needing to use the rest of the package. Such packages should contain a file named with the package name, followed by `defs`, followed by the appropriate suffix.

All C and C++ packages should provide a public header, named with the package name (in lower-case), followed by the appropriate suffix (either `.h` or `.H`). C++ packages that provide a C binding should provide a second header file, with `.h` for a suffix.

C and C++ packages should also have a file named `defs`, followed by the appropriate suffix. This file is for macros, types, and externs private to the package. It should include the public header file, so that source files need only include the `defs` file. Macros, types, and externs should always be defined in these header files, never in a source file.

4.1.2 Other Files

In general, all file names should be chosen to indicate something about the functionality of the code they contain. If you have a file in the BAKE package containing only a function to perform sorting, do not call the file `BAKEand_shake.c`. Instead, call it `BAKEsort.c`.

Many packages have conceptually similar functionality. For example, all functions that allocate and initialize C data structures should be contained in a file named with the package name, followed by `new`, and the appropriate suffix. Routines to free these data structures should be contained in the same file. Functions to print C data structures should go in a file named with the package name, followed by `print`, and the appropriate suffix. Functions to prepare data structures for network transmission should be defined in a file named with the package name, followed by `stream` and the appropriate suffix (see `stream(3U)` for more information). Subparts of the UGA environment will have additional file naming conventions. See the appropriate programmer's manuals for these. For example, 3D object methods have particular standards for their names.

C++ packages that define classes should define conceptually separate classes in separate files. For instance, a package might define several classes as support classes for one public class (for an example, see the OD package). These support classes can be defined in the same header as the public class, since they are only relevant to that class.

All methods for a C++ class should be defined in a single file, named with the class's name (which will include the package name, discussed below), followed by `.C`, unless the class is exceptionally large (see Section 6.1.1 for information on length of files), or provides functionality implemented in another language (such as a parser class that uses `yacc` and `lex`, or a class that calls Lisp or Fortran code).

Much as C++ files are named for their class, Lisp files should be named for their primary function (which will include the package name, see Section 4.2.2 for more). If a Lisp file does not provide any single primary function, but instead provides several functions as related utilities (e.g., the file `uga-environment.el`[1]), a descriptive name, starting with the package name, should be chosen. Of course, the name should contain the appropriate suffix.

Files should have a consistent format, including a descriptive header comment, RCS information, and comments breaking the file into sections for global variables, static, private, protected, and public functions and methods. This formatting is provided for you by the `nf` program. All source files should be created using `nf`. The man page generator (see Section 5.1) depends on comments provided by the `nf` command. See `nf(1U)` for more information.

[1] The elisp files in `/pro/uga/lbin/lisp` do not yet adhere to all the standards described here.

4.2 Code Names

Names of code elements[2] are even more crucial than file names. A good name indicates the usage, and thus the type, of the code element being named. Good names are as important to readable code as good comments.

4.2.1 Spelling

Most coding environments allow names that include non-alphanumeric characters. One of these characters should be used in place of a space in names consisting of several words. The Brown Graphics Group has standardized on this scheme (rather than the other predominant scheme involving capitalizing the first letter of new words), and UGA includes tens of thousands of lines of code that conform to this standard.

In C or C++, the character to use for a space is an underscore, '_'. `yacc` and `lex` source, as well as UIL source should also use underscores. Lisp code should use a dash, '-'.[3] TeX or LaTeX should use colons and dashes in label names (see Chapter 7).

Environments, such as UIL, that optionally allow case-sensitivity should enable case sensitivity.

4.2.2 The Scope Name

A name should indicate where a code element came from and who can use it (i.e., its scope). The name of any publicly available code element in a package should begin with the package's name in all capitals. If a name has an origin that is clearly indicated by the programming environment, it is not necessary to prepend the package name, thus preventing needless typing. As an example of this, consider a C++ method. The class containing this method should begin with the package name if it is in the public header file. The method name, however, does not need to begin with the package name, because the origin of the method is known by the C++ programming environments. Debuggers and class browsers can indicate the class containing the method, and thus (since the class contains the package name) can indicate the origin of the method.

It should be noted that names that are not publicly available need not include the package name. However, you must be especially careful as to what is and is not publicly available. For example, a C function used by several source files in one package, but not externed in the public header file is *still* available, because C does not require a function name to be defined before it is called. C++ is somewhat stricter about what is available, but there can still be problems. Even features that are truly not available outside of the package might want to include the package name. For example, static C functions do not need to include the name of the package, but might want to anyway, so that a stack trace in a debugger will immediately indicate what package the program is in.

In general, every C code element, public or not, should include the package name at the beginning in all capitals. Languages with better support for software engineering (such as C++) relax this restriction. Languages with worse software engineering support, such as Lisp, make this naming convention all the more essential.

Since some Lisp dialects are not case sensitive (notably, Common Lisp), there should be two colons between the package name and the name of the Lisp function or variable, for example, `RCS::steal-lock`.

[2]The phrase "code element" is used to indicate any variable, function, type, macro, class, method, or other first-class entity in a programming environment.

[3]Although most punctuation characters are permitted in Lisp identifiers, the dash is by far the most prevalent character to use in place of spaces.

Elements that are meant to be private, but include the package name anyway, should indicate their private status. This should be done by putting a "space" character between the package name and the rest of the name of the code element. In C or C++, this means you should put an underscore after the package name. For example, `PACKAGEpublic_ftn` is a public function name, whereas `PACKAGE_private_ftn` is a private function name. This standard holds for other languages, such as Lisp (e.g., `RCS::-remove-silly-buffers`).

4.2.3 Macros

Macros and constants should have names that are all caps to distinguish them from functions and variables. An additional "space" character is required between the package name and the remainder of the name to separate the two in macros and constants. Private macros and constants will thus contain *two* spaces between the package name and the rest of the name. In C or C++, public macros look like this: `NUPKG_MACRO_FUN`, and private macros look like this: `NUPKG__PRIVATE_DANCER`. Lisp public macros would have names like this: `KEYMAP::-MACROS-HAVE-NEW-SYNTAX`, whereas private macros would look like this: `KEYMAP::--QUIK-KEY-FIX`.

Macros that are primarily for use with a particular package data type, such as accessor macros, should include the type name immediately after the package name and the required spaces. The name of the value the macro is accessing should follow the type name. The type name is not necessary if the package defines only one type: `TRIP_NUM_FACES`, since the TRIP package only defines the `TRIPobject` type, but `HIGRAV_VERTEX_ID`, since the `HIGRAV` command defines several types, `HIGRAVvertex`, `HIGRAVedge`, `HIGRAVgraph`, and `HIGRAVlayer`.

Macros that perform loops should include the phrase "FOREACH" immediately after the package name and any necessary spaces and type indicators. This should be followed by a phrase or word indicating what the loop macro is looping over, e.g., `HIGRAV_NODE_FOREACH_OUTGOING_EDGE` or `TRIP_FOREACH_FACE`.

4.2.4 Types

Type names should include the package name, with no "space" character after it, unless the type is private: `HIGRAVvertex` is a public type, but `EM_event_queue` is not a public type. This naming convention holds for C++ classes as well, since they are types.

4.2.5 Variables

Global variables should begin with the package name, followed by spaces according to whether or not they are private. The number of spaces is the same as for types: zero spaces indicate a public global, whereas one space indicates a private global. Local variables have no restrictions on their names, save that they are descriptive. Three-letter names are, in general, not descriptive. Very few four- and five-letter names are descriptive. Modern programming language environments generally allow names to be of any length. "It is shorter to type," is not a valid reason for giving a variable a short name.

When a large number of functions with the same prototype exist, such as a series of X Toolkit callbacks, the parameters to these functions may be given a shorter name, provided this name is used consistently, and the use of this name as a standard is well-documented at the beginning of the file.

Parameters to callbacks that are not used should be called `unused`, so that the output of program checkers (such as `lint`) will indicate that the programmer realizes that a particular value is unused. If more than one parameter is unused, append a number to the name.

If a variable is passed to a function by reference (i.e., in C, by taking its address), the variable is serving as a return value to that function. As such, their names should include a "space" character, and `rtn` at the end, so that the output of program checkers will indicate that the programmer is using these variables correctly. Normally, program checkers will claim that these variables are being used before set.

Local variables which will be returned by a function should be named `return_val`. This makes their use within the function, i.e., as a place to accumulate the return value, clear.

4.2.6 Functions

As with other code elements, functions should begin with the package name and the appropriate "space" characters. Methods, as previously noted, may omit the package name and spaces.

There are some additional conventions for naming functions which operate on data types. A function which creates a particular data structure should be named with the data structure it is allocating, a "space" character, and the word `new`, unless the data structure is the only new public structure defined by the package, in which case `new` is sufficient.

In general, this pattern holds for all functions operating on a specific type. The type name should be included, unless it is implicit (as in C++), or the package supports only one type. The word to use at the end of the function depends on the exact operation. See Table 4.2 for a list of the words to use for various common operations.

Word	Operation
`new`	Allocate and initialize a data structure
`free`	Deallocate a data structure
`copy`	Copy a data structure
`own`	Increment a reference counter on a data structure
`print`	Print a data structure to `stdout`
`write`	Print a data structure to an arbitrary file
`scan`	Read a data structure from `stdin`
`read`	Read a data structure from an arbitrary file

Table 4.2: Summary of words indicating the operation carried out by a function.

Chapter 5

Comments

Commenting source code is a two-edged sword. While some argue that the lack of comments makes it more difficult to understand code, others find excessively commented code hard to read. Unfortunately, these opinions are dependent on a programmer's personal preferences. Nevertheless, the following guidelines should be followed.

5.1 Source Files

All source files must be created by the `nf` command (see `nf(1U)`). The `nf` command provides each file with a skeleton based on the file type and name. Each source file has, at the top, a place to provide a brief, 1–2 sentence description of the source file. The description should always be present in all source files, since it is scanned by the automatic man-page generator (`manpage(1U)`). In addition, each source file will have "line across the screen" comments, shown in Figure 5.1. These comments partition a file into the different types of routines that can exist. These comments

```
/* -------------------- Static Routines  --------------------- */

/* -------------------- Private Routines --------------------- */

/* -------------------- Public Routines  --------------------- */
```

Figure 5.1: Automatically generated partitioning comments.

should *never* be removed, even if a file contains no private or static routines. They are used by the automatic man-page generator and the automatic prototype generator (`proto(1U)`). Similarly, `.h` files are partitioned into enumerations, constants, types, macros, and entries. These comments should never be removed either.

5.2 Function Headers

Every function should contain the following comments:

1. A short description of the routine, including how to use it and necessary preconditions.
   This description should be preceded by the keyword `DESCR :`.

2. An optional, detailed description of the mechanics of the routine.
   This description should be preceded by the keyword `DETAILS :`.

3. A description of the return values.
   This description should be prefaced by the keyword `RETURNS :`. If the function type is `void`, this comment should be omitted.

4. A description of each argument.
   Each argument comment should begin on the same line as the argument it describes. In some cases, a comment is unnecessary, as the argument name itself describes the purpose of the argument.

Note the space before the colon in each keyword. The keywords `DESCR :` and `RETURNS :`, and the text that follows them are used by the automatic man-page generator. Similarly, the argument comments must be placed on the same line as the argument for the man-page generator. These comments and keywords should be present in all functions, regardless of whether they are static, private, or public.

Function header comments should be block comments, i.e., the first line of the comment is a line across the page, a '`*`' in column two before each line of commented text, and a closing line across the page on the last comment line. Note that `grep '^.\*'` will catch all block comments in the file. Table 5.1 shows a number of Emacs macros that have been defined to insert function header comments.

Keystroke	**Fields present in comment inserted into code**
`C-x g c`	`DESCR :,RETURNS :,DETAILS :.`
`C-x g C`	`DESCR :,RETURNS :.`
`C-x g v`	`DESCR :,DETAILS :.`
`C-x g V`	`DESCR :.`

Table 5.1: Emacs macros to insert function header comments.

Comments should be as concise as possible. It is not necessary to begin a function description comment with "This function..." (as if to differentiate it from some other function?). It is best to start a comment with a verb, since the primary purpose of the comment is to describe *what* the function does. Function names can also describe a lot — it is unnecessary to begin the description of `ERRregister_package()` with "Registers a package with ERR..."

To illustrate the use of keywords, Figure 5.2 shows a well-commented header for the non-existent routine `ERRregister_package()`. The generated man page states the information necessary to properly use this function.

Note the special character '`@`' in the header. Text delimited by '`@`' characters will be italicized in the man-page. All variables and types should be surrounded by '`@`' characters.

```
/* -----------------------------------------------------------------
 * DESCR   : Creates an @ERRid@, stores the package name in it,
 *           and returns it to the caller.
 *
 * RETURNS : A pointer to the @ERRid@, which is to be used in
 *           subsequent @ERR@ calls
 * ----------------------------------------------------------------- */
ERRid *
ERRregister_package(
   char   *name            /* The package name being registered */
)
```

Figure 5.2: A well-written function header.

5.3 In-Line Comments

In-line comments are the most difficult types of comments to determine guidelines for. Depending on the programmer's personality, speed of reading, level of intelligence, as well as his cholesterol level, the readability of the same code will be judged differently. Rumor has it that the number of in-line comments in a package are inversely proportional to the programmer's academic-year salary.

Readability is significantly improved if left-sides (and, to a lesser extent, right-sides) of comments are aligned. The two code fragments in Figure 5.3 illustrate this.

```
if (dim == 3) {
   do_this3(x, y, z); /* Compute for three dimensions */
   return x / h; /* Return homogenized x coordinate*/
} else {
   do_that2(x, y); /* Compute for two dimensions, saving time*/
   return y / h; /* Return homogenized y coordinate*/
}


if (dim == 3) {
   do_this3(x, y, z);    /* Compute for three dimensions             */
   return x / h;         /* Return homogenized x coordinate          */
} else {
   do_that2(x, y);       /* Compute for two dimensions, saving time */
   return y / h;         /* Return homogenized y coordinate          */
}
```

Figure 5.3: Unreadable and readable comments.

If the comment is on a line by itself, it should be indented as far as the indentation of the next line of code. Short comments, as in Figure 5.3, should be placed on the same line as the code described.

Keep the following in mind when writing in-line comments:

1. It is unwise to assume that the code will "explain itself," thus making comments unnecessary.

Rarely do two people interpret, understand, or execute (in their head, at least) code in the same manner.

2. Extensive commenting (multiple occurrences of chunks of 10-line comments) probably warrants writing a separate document.

3. Short comments should describe *what*, like "compute mean value," instead of *how*, like "sum values and divide by n," unless the *how* is not obvious, like "find the root using Newton's Method."

5.4 Public `.h` Files

The enumerated types, macros, and constants in a package's public `.h` file should be well commented. Again, the automatic man-page generator will look for keywords in the `.h` file when building a man page.

5.4.1 Macros and Constants

Comments need not be written for all macros and constants. Often, one comment suffices for a group of macros. Comments with the keyword `DESCR :` will be placed in the man page. The keyword `RETURNS :` should be used to specify the data type returned by a macro. Alternatively, a macro may have a return type defined in an in-line comment after the `#define`. For simplicity, a macro's return type listed in the man page will be obtained from the closest, preceding return-type comment. For example, the code in Figure 5.4 generated the man page in Figure 5.5.

```
/* DESCR    : Accessor macros for @STDglob@ data structure
              @G@ is a pointer to the @STDglob@.
 * RETURNS : int
 */
#define STD_GLOB_HISTOGRAM(G)        ((G)->histogram)
#define STD_GLOB_FREES(G)            ((G)->frees)
#define STD_GLOB_BYTES_FREED(G)      ((G)->bytes_freed)
#define STD_GLOB_MALLOCS(G)          ((G)->mallocks)
#define STD_GLOB_BYTES_MALLOCED(G)   ((G)->bytes_mallocked)
#define STD_GLOB_TOTAL_ERRORS(G, S)  (STD__GLOB_ERR(G).tots[(int)S])

#define /*  STDbool   */ STD_GLOB_TRACING(G)      ((G)->tracing)
#define /*  char  *   */ STD_GLOB_PACKAGE_NAME(G) ((G)->pname)
```

Figure 5.4: Portion of a commented `.h` file.

5.4.2 Enumerated Types

The comment for an enumerated type comes directly in front of the enumerated type declaration. The keyword `DESCR :` precedes the description of the enumerated type in the comment.

```
Accessor macros for STDglob data structure
G is a pointer to the STDglob.
  int         STD_GLOB_HISTOGRAM(G)
  int         STD_GLOB_FREES(G)
  int         STD_GLOB_BYTES_FREED(G)
  int         STD_GLOB_MALLOCS(G)
  int         STD_GLOB_BYTES_MALLOCED(G)
  int         STD_GLOB_TOTAL_ERRORS(G, S)
  STDbool     STD_GLOB_TRACING(G)
  char  *     STD_GLOB_PACKAGE_NAME(G)
```

Figure 5.5: Man page generated from Figure 5.4.

Chapter 6

Formatting Style

This chapter describes how the UGA programmer should format code. The given style rules are at a low-level; for example, we prescribe the number of spaces to be used for indenting and how function headers are to be specified. These rules introduce a consistent formatting style to make code more readable.

The style itself is not important as long as it is consistent. The Brown Graphics Group has chosen the particular style described here. Even though the choice was based on the personal preferences of the authors, it is essential that all programmers adhere to these rules. The code can only be consistent if *all* programmers follow the rules. If you dislike a particular rule, talk to one of the authors or UGA administrators. Please do not simply ignore rules.

6.1 Managing White Space

This section illustrates how to make code more readable by applying a set of simple formatting rules. The rules are concerned with the effective use of white space. In particular, the last subsection (Section 6.1.3) will demonstrate how inserting additional blanks can improve readability considerably.

6.1.1 Vertical White Space

No file should be longer than 1000 lines. This rule is rather conservative. Typically, a file is less than 500 lines long. It is not only a sign of bad software engineering, if a file is longer than 1000 lines, but it is also very inefficient. In general, all applications used with source code files will run slower on large files. For example, it will take longer to start up an editor with a large file than with a small one. Compiling and linking two small files is faster than compiling and linking one large file. However, the most compelling reason not to have large files is that the human brain is not capable of grasping files that have 1000 or more lines.

No function should be longer than 80 lines. The 80 lines should include the whole body of the function, but not necessarily the header comment. If a function seems to need more than 80 lines, then the logical structure (i.e., use of subroutines, etc.) of the code is probably wrong. It is necessary to rethink the approach, not only because this rule would be violated, but the overall software engineering is probably faulty.

Normally a function should fit into 60 lines, so that with a reasonable font size it is possible to view the whole function at once. Seeing the whole function body at once is helpful in understanding what the function does and how it achieves its result.

Use blank lines wisely. There should be two blank lines between the end of one function and the beginning of the header comment for the next function. The blank lines will visually set the two functions apart.

A number of related rules follow (see also Figure 6.1):

- There should be exactly one blank line between `lint` directives and the beginning of the function header comment.
- There should be no blank lines between the static, private, and public "line across the screen" comments and `lint` directives.
- There should be no blank lines between the header comment of a function and its function header.
- There should be exactly one blank line between the local variable declarations and the first line of code inside the function body.
- Blank lines visually separating code segments inside the function body are allowed. The blank lines, if used sparingly, make the code easier to read. Sometimes the need for blank lines inside the function body indicates that the function should be split into several subroutines.

6.1.2 Horizontal White Space

Never use tabs. Never use tabs. Let us say this again: Never use tabs! The UGA system is supported on several different architectures; each of those architectures defines the tab-character to be a different width. Files that look beautiful on one machine are unreadable on the next machine. Tabs also do not print well in general.

Please change your editor to insert the appropriate number of spaces instead of a tabulator character. (The UGA Emacs macro library includes this functionality.) Reprogramming your editor to display tabs uniformly for all supported architectures is not a solution, since it cannot be expected that all new students and external contacts reprogram their editors as well.

No line should be longer than 80 columns. Most terminals and printers support 80-column text as a standard. If you have the need for longer lines, i.e., because of deep control structure nesting, you should re-examine your code. Deep nesting is often a sign of poorly organized code.

Long boolean expressions in if-statements, complicated algebraic expressions, or function calls with a large number of arguments can also generate long lines. These can be broken up by introducing local macros (which should be defined in the `defs.h` file) or temporary variables. Each temporary variable should be assigned a part of the expression, so that the expression can be rewritten using only the newly introduced variables. If this technique can not be applied, then an expression needs to be broken into several lines. It is important that the break appears just after (and not before) the operator. The second subexpression is indented exactly one more space in comparison to the opening parenthesis and the subexpressions should be matched up. For example:

```
...
   /* This is the end of some function. Note the following */
   /* two blank lines after the end of this function.      */
}


/* ------------------ Public Routines  --------------------- */
/*LINTLIBRARY*/

/* ----------------------------------------------------------
 * DESCR : There are no blank lines between this comment and
 *         the function header. There is exactly one blank
 *         line between this comment and the lint directive
 *         above and no blank lines between the Public
 *         Routines separator and the lint directive.
 * ---------------------------------------------------------- */
void
PKGfct(
   PKGargument arg      /* Some argument */
   )
{
   double local_double; /* Notice there is exactly one       */
   int    local_int;    /* blank line separating the local   */
   int    another_int;  /* variables from the code.          */

   if (!VALID_VAR(arg))
      FATAL((ERR_SI, "bad variable number:", (int)arg));

   if (!VAR(arg)) {
      FATAL((ERR_SS, "Variable unset: ", VAR_NAME(arg)));
      return NULL;
   }

   return VAR(arg);
}


/* ----------------------------------------------------------
 * DESCR : This is the next function. There are exactly
 *         two blank lines between beginning of this
 *         comment and the end of the previous function.
...
```

Figure 6.1: Correct use of blank lines.

```
if ((a >= 0.0 && a <= 1.0) ||
    (b >= 0.0 && b <= 1.0)) {
   a = b + c;
   d = e * f;
}
```

Three spaces should be used to indent code. To quote from the movie "Monty Python and the Holy Grail" [PYTH76]:

> *Second Brother:* And the Lord spake, saying, '[...] Then, shalt thou count to three, no more, no less. Three shalt be the number thou shalt count, and the number of the counting shalt be three. Four shalt thou not count, nor either count thou two, excepting that thou then proceed to three. Five is right out. Once the number three, being the third number, be reached, then [...]'
>
> *Brother Maynard:* Amen.
>
> *All:* Amen.

Blanks should be used around binary operators and after commas. All binary operators, i.e., '+', '*', '=', etc. should be surrounded by blanks. There are exceptions to this rule; it is sometimes of advantage to omit the blanks in complex formulas to emphasize operator precedence.

A comma should always be followed by at least one blank to enhance readability. Since commas are most often used to separate function arguments, a related rule concerns how to separate the surrounding parentheses of a function call: There should be no space after the left parenthesis and no space before the right parenthesis. A function taking three arguments would therefore be written thus:

```
PKGfct(arg1, arg2, arg3);
```

The lack of a blank after the left parenthesis is particularly important. Allowing a blank in that position can lead to errors when the rule is applied to macro definitions. Also, single-argument functions, formatted consistently with such a rule, would be hard to read.

There should be at most one statement per line. It is easy to overlook statements that are on the same line as other statements. In addition, it is hard to examine the state of a program, if a line contains multiple statements: most debuggers only allow to insert breakpoints at the beginning of a line.

Figure 6.2 summarizes the above rules in an example function.

6.1.3 Matching Up Statements

Match up in-line comments. Beginning and end of in-line comments should be matched up. An example is:

```
if (e < 0.0) return 0;              /* no rational roots */
else {
   if FZERO(e) return 1;            /* one root          */
   else        return 2;
}
```

```
/* ---------------------------------------------------------
 * DESCR :  The line length is less than 80 columns.
 *          No tabs were used while writing this function!
 *          Each line contains at most one statement.
 * --------------------------------------------------------- */
void
PKGexample(
   int hungry;                       /* can be TRUE or FALSE */
   )
{
   int   i;

   if (hungry) {
      make_trip(DUNKIN_DONUTS);
      wealth = money - price(donut, DOZEN);  /* The minus is */
                                     /* surrounded by spaces */
      for(i = 0; i < DOZEN; i++)     /* Note one space after */
         eat(donut, i);              /* the comma; no blanks */
   }                                 /* around the parens    */
}
```

Figure 6.2: How to use horizontal white space.

Match up assignments. Assignments occupying consecutive lines in the source file should be matched up so that the assignment operator is in the same column. The following code illustrates this rule.

```
q        = (a1*a1 - 3.0 * a2) / 9.0;
a1_third = a1 / 3.0;
q_cube   = q * q * q;
```

Match up the arguments of consecutive function calls. Function calls and other expressions occupying consecutive lines in the source file should be matched up so that the arguments start in the same column. The following is an example for an assignment expression.

```
roots_rtn[0] = tmp * cos( phi             / 3.0) - a1_third;
roots_rtn[1] = tmp * cos((phi + 2.0*M_PI) / 3.0) - a1_third;
```

An example involving function calls is shown here.

```
MAT3perp_vec(arm,         axis, TRUE);
MAT3mult_vec(rotated_arm, arm,  mat);
```

Match up the arguments of long function calls. If a function takes a large number of arguments, it is sometimes not possible to fit a call to it into a single 80-column line. Instead of violating the 80-column rule, the call should be broken into several lines, with the arguments formatted as shown.

```
    RATI_store_hit(RATI_TREE_HIT(tree),
                   RATI_TREE_TRIP_OBJECT(tree),
                   RATI_NODE_FACE(node), pierce, t);
```

The above rules should be extrapolated to other programming constructs as well, since matching up is a powerful method to enhance readability. Doing so should not violate any rules stated in the earlier subsections.

6.2 Indenting Patterns

The indenting patterns to be used with common programming constructs are introduced here. In general, the indenting style used in the White Book [KERN88] was adopted.

6.2.1 Functions

The general indenting pattern for functions is shown in Figure 6.3. The same style was illustrated earlier, refer to Figures 6.1 and 6.2. In particular, the following points should be observed.

```
/* ----------------------------------------------------------
 * DESCR   : Header Comment
 * ---------------------------------------------------------- */
int
PKGfct(
   MAT3vec           unused,
   double           *coeffs,
   double           *roots_rtn
   )
{
   TRIPobject       *more_complex;
   int               ret_val;

   if (FZERO(coeffs[0]))
      ret_val      = 0;
   else {
      roots_rtn[0] = -coeffs[1] / coeffs[0];
      ret_val      =  1;
   }
   return ret_val;                   /* not return(ret_val); */
}
```

Figure 6.3: Indenting pattern for functions.

- The return type and the function name are both flush with the left margin.
- All arguments to the function have to be listed in ANSI C style and are indented by three spaces. Each argument occupies exactly one line.

- All argument names have to start in the same column. Note that the asterisk is not part of the argument name, but of the argument type.
- If possible, the arguments should be ordered from most complex data type to least complex.
- All local variables need to be declared at the beginning of the function, i.e., no block variables are allowed. The local variable declaration is indented by three spaces.
- All local variable names have to start in the same column. The asterisk is not part of the local variable name.
- The declaration of local variables need to be ordered such that the more complex data types are declared before the less complex ones.
- All statements inside the function body need to be indented by three spaces.
- A function of type `void` should not use the return statement. All other functions have to use the return statement without enclosing parenthesis.

6.2.2 If-Statements

An if-statement should be formatted as described in the White Book [KERN88], i.e.,

```
if (a > b) {
   c = a + b;
   d = a - b;
} else {
   c = b;
   d = b + 2.0 * a;
}
```

or in the case of else-if statements

```
if (a < 0.0) {
   neg_fct(a);
} else if (a < 1.0) {
   one_fct(a);
} else if (a < 2.0) {
   two_fct(a);
} else {
   big_fct(a);
}
```

If-statements that have short enough then- and else-parts should be formatted like this:

```
if (a == b) some_fct(a);
else        uneq_fct(a);
```

Notice that the starting column of the else-statement is chosen to match the beginning of the if-part.

Complex conditions might not fit onto a single line. As discussed in the rule for 80-column lines, this is avoided by introducing local macros or temporary boolean variables.

```
cond1 = really_long_boolean_expression1;
cond2 = really_long_boolean_expression2;
cond3 = really_long_boolean_expression3;
if (cond1 && cond3 && cond3)
   a = b;
```

This technique does not affect the performance of the code, since today's compilers are able to optimize away these artifacts.

We encourage testing pointers for `NULL` by using the pointer itself as the condition, e.g., `if (ptr)` instead of `if (ptr == NULL)`. Similarly, boolean expressions need not be explicitly tested for equality to `TRUE`, instead the expression should be used directly. However, when testing for equality of character, integer, or floating point expressions the test should be stated explicitly.

6.2.3 Switch-Statements

All switch-statements should be formatted in the following manner:

```
switch (var) {
    case CONST1:
         function1();
         function2();
  brcase CONST2:
         function3();
         function4();
  brdefault:
         function5();
         function6();
}
```

Note that the `brcase` and `brdefault` macros are indented two spaces, whereas all the statements are indented nine spaces.

6.2.4 For-Statements

All for-Statements should be formatted like thus:

```
for(i = 1; i < const; i++)
   a += b[i];
```

Elaborate for-loops should be split onto different lines with the following indentation:

```
/* Traverse a circular linked-list once */
for (first = NULL, elem = LIST_HEAD(list);
     elem && (elem != first);
     elem = LIST_ELEM_NEXT(elem), first = LIST_HEAD(list)) {
   LIST_ELEM_VISITED(elem) = TRUE;
   num++;
}
```

Empty loops should be formatted in a way that makes it obvious that the loop has no body. Use, for example,

```
for(i = 0; i < const; sum += i, i++)
    ;
```

6.2.5 The `?:` Expression

There are two indenting styles that can be used for the conditional expression `?:`.

```
a = (c == d) ? value1
             : value2;

a = (c == d)
      ? value1
      : value2;
```

6.2.6 All Other Statements

The indenting patterns for all programming constructs that were not mentioned in this section should be inferred from *The C Programming Language* [KERN88]. However, the use of some of these constructs is not common or desirable, since they are hard to read. In particular, block variables and do-while loops should be avoided.

Chapter 7

Documentation

Good documentation allows beginners to learn about and use available functionality. Equally important, experts can maintain software over long periods of time only, if it is sufficiently documented. For these reasons, all packages and commands in the UGA hierarchy need to be documented by comments in the source code, man pages, and optional long documents.

The format of comments and source code as a form of documentation has already been discussed in previous chapters. It was briefly mentioned that man pages are generated automatically with the `manpage` command (`manpage(1U)`). The `manpage` command is invoked every time the Makefile-target `man` is given. It composes a project's manual page from the `README` file (located in the project's source directory) as well as the project's source code comments (as discussed in Sections 5.2 and 5.4).

For complicated or large packages, source code comments and man pages might not suffice. In these cases, additional long documents describing the project on a high level need to be provided. Long documents have the form of a report, should be written in English [STRU79], and need to be spell checked (`spell(1)`). The formatting language of these documents has to be LaTeX [LAMP86]. Familiarity with LaTeX is therefore expected, in particular, rules such as using `\em` instead of `\bf` or using `\verb` and `\verbatim` instead of `\tt` and `\it` should be obeyed.

A long document is created via the `request_project` command (`request_project(1U)`), which will create a `doc` subdirectory in the project's main directory. The `doc` subdirectory will contain a standard documentation Makefile that defines `preview` and `print` targets.[1] The `print` target should be used cautiously to save paper.

The UGA environment also provides a style file, `/pro/graphics/latex/graphics.sty`, which is accessed by including it as a documentstyle option, i.e.,

```
\documentstyle[graphics]{report}
```

The graphics style file defines formatting commands which should be used by authors of long documents. Table 7.2 lists the commands; more elaborate examples of how to use these commands can be found in the file `/pro/graphics/latex/example.tex`.

[1] The user needs to set the following environment variables before these targets can be used:

```
setenv TEXINPUTS .:/pro/uga/pkg/typo/data/tex:/cs/lib/tex/inputs
setenv TEXPKS    /pro/uga/pkg/typo/data/bitstream:/cs/lib/tex/fonts/pk
setenv TEXFONTS  /pro/uga/pkg/typo/data/bitstream:/cs/lib/tex/fonts/tfm
setenv XDVIFONTS /pro/uga/pkg/typo/data/bitstream:/cs/lib/tex/fonts/pk:/cs/lib/tex/fonts/pspk
```

Since all LaTeX files should be under RCS control, each file's first line should be

```
% $Id: documentation.tex,v 2.5 92/01/13 17:47:19 mmw Exp $
```

The main file (the LaTeX file containing the `\documentstyle` line) should also contain entries for `\RCSdate$Date$` and `\RCSrevision$Revision$`, located just after the `\documentstyle` line to control the date and version printed on the document.

Writers of long documents should adopt a consistent internal naming scheme for labels, that allows for quick recognition of what the label refers to. For example, we suggest prefixing labels with an abbreviated decription of the referred to structure followed by a colon, i.e., when referring to a chapter prefix the label with `ch:` (see table 7.1 for a complete listing). Label names should resemble the referred to structure name with dashes substituted for spaces, for example, a chapter named "General Coding Style" would be referred to as `ch:general-coding-style`. For TeXnical reasons, hyphens must be used in labels and LaTeX file names instead of underscores.

Structure	**Prefix**
Chapter	`ch:`
Section	`sct:`
Figure	`fig:`
Table	`tbl:`

Table 7.1: Summary of suggested label prefixes.

Command	**Example Usage**	**Output**	**Explanation**
`\AnsiC`	`\AnsiC`	ANSI C	
`\ASCII`	`\ASCII`	ASCII	
`\BCGG`	`\BCGG`	BCGG	Brown Computer Graphics Group
`\Bezier`	`\Bezier`	Bézier	
`\C++`	`\C++`	C++	
`\GNU`	`\GNU`	GNU	
`\PS`	`\PS`	POSTSCRIPT	
`\RCS`	`\RCS`	RCS	File revision control system
`\SIGGRAPH`	`\SIGGRAPH`	SIGGRAPH	
`\UNIX`	`\UNIX`	UNIX	
`\<>`	`\<letter>`	⟨letter⟩	For syntactic elements
`\Draft`	`\Draft`		To mark a document as a draft
`\edcmnt`	`\edcmnt{Really?}`	◇*Really?*	For editorial comments
`\file`	`\file{defs.h}`	`defs.h`	For filenames
`\intro`	`\intro{newterm}`	*newterm*	For introducing new terms
`\key`	`\key{C-x C-a}`	`C-x C-a`	For keystroke sequences
`\man`	`\man spell(1)`	`spell(1)`	For man page references
`\prog`	`\prog{spell}`	`spell`	For non-UGA programs
`\BAGS`	`\BAGS`	BAGS	
`\UGA`	`\UGA`	UGA	
`\SCEFO`	`\SCEFO`	SCEFO	
`\FLESH`	`\FLESH`	FLESH	
`\cmd`	`\cmd{grim}`	`grim`	For UGA command names
`\pkg`	`\pkg{trip}`	TRIP	For UGA package names

Table 7.2: Summary of formatting commands provided by `graphics.sty`.

Chapter 8

Enforcement

Before a project is "finished", the newly written source code must be approved by a committee of three people to verify the correct application of the standards described in this document. The committee will consist of the UGA administrators, senior members of the group, and others who have proven to consistently follow the standards. If the code is found inadequate, the author will be asked to make modifications.

This should not be looked upon as fascism, nor should it be taken personally. The UGA framework is a team effort, and everyone in the Brown Graphics Group is a part of that team. The enforcement of coding standards ensures that everyone's code will be usable by everyone else on the team. No one on the team should be forced to waste time trying to decipher someone else's poorly written code.

Bibliography

[BENT82] Jon Louis Bentley, *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, 1982

[DARW90] Ian F. Darwin, *Checking C Programs with lint*, O'Reilly & Associates, Inc., Sebastopol, California, 1990.

[GNUMAKE] Richard M. Stallman and Roland McGrath, *GNU Make*, Free Software Foundation, Inc., Cambridge, MA, 1990

[INDHILLS] L. W. Cannon, R. A. Elliott, L. W. Kirchoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, and Mark Brader, *Recommended C Style and Coding Standards*. Bell Labs and elsewhere. Also known as the "Indian Hills Guide".

[KERN88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, Second Edition, 1988.

[LAMP86] Leslie Lamport, LaTeX*: A Document Preparation System*, Addison-Wesley, Reading, MA, 1986

[PYTH76] Monty Python, *Monty Python and the Holy Grail*, MGM Studios, 1975.

[STRA91] Paul S. Strauss, Michael J. Natkin, and Nate Huang, "The Programming Environment for the Brown Computer Graphics Group," Computer Graphics Group Documentation, Brown University, Providence, RI 02912, July 1991

[STRU79] William Strunk Jr. and E. B. White, *The Elements of Style*, Macmillan, New York, Third Edition 1979.

[WIRT82] Niklaus Wirth, *Programming in MODULA-2*, Springer Verlag, Berlin Heidelberg, 1982