Code Review Guidelines

Input validation

Command line arguments

Check that the number of command line arguments matches what the program expects. If possible, check the data types of arguments. For example: you want to input a positive number as a command line argument you might use atoi() on the argument to convert it from the string representation of a positive number to an int. However, if the user inputs an erroneous value like "abc" the return from atoi() will be 0. The output of atoi(), and any other conversion functions that you use, such as atof(), strtol(), etc, should always be checked against expectations.

Environment variables

If you read a value from the environment using getenv() always check that the result is not NULL. Also, apply data type checking as described above. Check the length of the string returned as well; the actual upper bound on the string length is OS-dependent. On Windows, the same applies to values read from the registry.

Configuration files

If your program uses a configuration file, you should have a separate parse() function that handles parsing the configuration file, and that returns a status code indicating whether or not the parse was successful. The same applies to system configuration files that you might read/parse in your program. If the name of the configuration file is a command line argument, be aware that the user may well specify an erroneous filename. Your configuration parser must be prepared to handle every possibly contingency gracefully, including being given a binary file, a file of enormous length, a file that has zero length, a socket, and so forth. The same applies to all input files.

Fixed length buffers

Don't put quantities of indeterminate length in fixed length buffers. For example, don't use the following pattern: char buf[256]; sprintf(buf, "%s.txt", argv[1]); because sooner or later the user will supply a value for argv[1] that is larger than 251 characters, and a buffer overflow will occur. This can lead to memory pool corruption, stack corruption and a number of other hard-to-find errors.

Execution environment

Temporary files

If you are using temporary files try to use a system function such as mkstemp() to create them, and check the return code from this function. If mkstemp() is unavailable and you

must use mktemp(), insure that the resulting file is open()ed with the O_EXCL flag to enforce single access. Don't assume that particular directories such as /tmp will always exist or be writable. On Windows, don't assume that the directory named by %TEMP% will exist or be writable. For added security consider putting your temporary files into a directory that you yourself create using mkdir() (as a subdirectory of an existing temporary directory, for example), and to which you give the most restrictive permissions possible.

Output directories

If your function creates files, insure that the output directory exists and is writable. This amounts to checking the return code from functions such as open() and creat(). Also be aware that these functions behave differently in native Windows applications, in Cygwin, and on Unix systems, particularly with regard to the default permissions on the resulting files. This can lead to ugly situations in which you have created a file but cannot write it. Therefore, use umask() before opening/creating the file, providing the most restrictive mask possible; use open() with a third argument (again, the most restrictive possible), or use chmod() after creating the file, or use fstat() or stat() to determine the file's permission bits. Beware of a possible race condition with stat() in that the value you get is only valid at the particular instant that you executing the stat() call, and can potentially change later (particularly if the file has inappropriately generous permissions). Finally, always check the return code from any write() to insure that the bytes have really been written, and if you get a permission denied error, try to fix up the situation using one of the steps outlined above.

Disk full

One rarely expects that the disk will be full, but sometimes it is. Again, always check the return code from write() or WriteFile() and take appropriate action or post a meaningful error message.

Exceptions

Catch an appropriate set of signals and perform appropriate cleanup. Always ask yourself: if the user types Control-C at the worst possible moment, what will happen? Code accordingly.

In C++ and Java make sure that the appropriate fine-grained exception handlers are installed; don't default to trivial handlers that do nothing, like: catch (Exception e) {}. Insure that state is cleaned up by using appropriate "finally" clauses or by setting an error indication in the handler that is acted upon when the code block is exited.

Always check for out of memory conditions. Always check the return code from anything that allocates memory, including the obvious ones (malloc, calloc, realloc) and the less obvious ones (strdup). In C++ arrange for a mechanism to insure that constructors have really constructed the object in question, not just when using "new" but

also when constructed on the stack. For example, a constructor can init a private Boolean variable "ok" when it is completely done and then provide a Boolean status() method that returns the value of ok as a check that the object has really been fully constructed, or it can arrange to throw an exception if the it failed.

General quality

Testing arguments

Any function or method that is public should always validate its arguments. This can also be applied to private functions that are called by multiple callers, but at a minimum it should always be enforced for public functions. A particularly nasty situation can occur when a private function becomes public. For example, suppose a function a() calls another function with the signature b(char *). The function a() has already validated the pointer that it passes to b(), and b() is static, so why bother checking the pointer again in b()? This is called "implicit knowledge" and is a reasonable coding strategy, except when b() becomes a public function, or when b() is called not just from a() but from several functions. Argument validation should always be performed so long as the performance impact is minimal.

Error handling

If a function can fail it should return a status code indicating this, or it should throw an exception (or possibly raise a signal). This also implies that callers should test the return codes for any functions that have them, and do so uniformly. That is, if a function returns a status code it should always be tested. If it is never tested, the function should be made "void". Another acceptable strategy is that "all errors are fatal." In this case, the program just exits after performing appropriate cleanup. It is not acceptable to ignore errors.

Callers should also test the return codes from system functions, where meaningful (e.g. you don't have to test the return code from printf()). Some of the critical functions have already been listed: open(), creat(), write(). In general, if you are asking the OS to perform an operation on your behalf you should check that it has really done so.

OS dependencies

Operating system dependencies and platform dependencies should be isolated to a set of functions, preferably in a single file. If OS dependencies are scattering in the code bodies of several functions debugging is significantly more difficult. Things may work perfectly well on your test platform, but fail horribly on a very slightly different system. This applies very strongly to programs that must run on two or more of the following platforms: native Windows, Cygwin, Unix/Linux.

State

All state must be cleaned up on any exit from the program, other than exits due to fatal signals such as segmentation fault or GPF. Flush all output files, close all files, pipes and sockets, delete all temporary files, free all allocated memory, reset and detach from system-wide resources such as global semaphores or shared memory, explicitly unload any explicitly loaded shared libraries, terminate and reap all threads and subprocesses, and so forth. Despite what the manual tells you, do not assume that the operating system will clean up after you. It might do so (and take its own sweet time about it), or it might not. Also beware of cleanup paradigms that are OS/platform specific. For example, in Unix/Linux it is common to create a temporary file and then immediately unlink() it, knowing that when the file is ultimately closed the unlink() will complete and it will be deleted. This works fine in some versions of Cygwin, but not all. It also requires a slightly different set of function calls in native Windows programs. In summary: if you did X during the execution of your program, be sure to undo X when you exit.

Advanced topics

Logging

Some type of logging, debug output or instrumentation is always desirable. Consider having a –v (verbose) flag for your program that outputs various levels of debug information, or that enables state/progress logging, or both. If you are going to log to stdout, make sure you call setbuf(stdout, NULL) at the very beginning of your program to insure that stdout is not buffered.

The same effect can also be accomplished with conditional compilation, e.g. DEBUG and RELEASE builds on Windows.

printf() is the oldest debugging technique there is. Don't be shy about using it.

Threads

Practice good function and global variable hygiene when using threads. Use mutexes and/or critical sections as appropriate. Be very careful with synchronization. Plan for unexpected thread death; code accordingly. Create a notification mechanism so that the parent can ask any threads (or subprocesses) to cleanup and exit; forcibly terminate threads (and subprocesses) only as a last resort.

Character sets

Someday your program might have to handle multi-byte character sets, e.g. Unicode. Code defensively, and at a minimum try to be aware of the places that you are explicitly assuming ASCII. If you are using the Windows registry you are already implicitly handling multi-byte characters, so write your code accordingly.