Navigate:

Style Guidelines for Programming in C/C++

- General
  - 80% of programming is maintenance programming and 80% of maintenance programming is figuring out how the maintained code is supposed to work and why it doesn't. Therefore, *emphasize clarity over all else!!*
  - Don't be clever. Avoid quirks of the language in favor of clarity. Avoid implicit or obscure features of the language. And always, say what you mean.
  - Maintain a *consistent* style throughout a program.
  - Compile with all warnings on, and remove them.
  - If it's constant, declare it const.
- Comments
  - Keep code and comments visually separate, e.g., space between code and trailing comments.
  - It's helpful to line up trailing comments.
  - Use header comments for each file and function. Include the function's purpose, what parms it takes, what it returns, etc.
  - Give pre- and post-conditions for major blocks of code, preferably in the form of assert statements, which are active comments.
  - Explain any tricky code.
  - Comments should be an aid to understanding. Do not comment the obvious (assume the reader knows C++). For instance,

```
length++;   // Update length to reflect appended char.
```

as opposed to

```
length++;   // Increment length.
```

- Names
  - Use meaningful identifiers for the names of functions, objects and types.
  - Avoid single character names except for loop control.
  - Capitalize or use underscore to separate words in a name (doThings or do_things)
  - Capitalize the first letter of class names (e.g., List) and only class names.
  - Use enumerations to give meaningful names.
- Scope
  - Minimize the scope of each name, e.g., define variables close to where they are used.
  - Minimize the use of global variables, and clearly define and describe them.
- Functions
  - Refinement: break down complexity into simpler coherent chunks.
  - Abstraction: consolidate activities or statement sequences that are performed in several places into a function that can be called at each of them.
  - Always specify the type of the return value, even for main().
  - Keep all functions simple, including main().
  - Keep the number of parameters to a function as small as possible.
  - Keep function length to at most one page, preferably shorter.

TOP ▲

Navigate:

Example of C++ Style and Commenting

```
// Name
// Login
// Course
// Lab Section Number
// Program Name (as1, as2, lab1, ...)
// Program Description

includes

prototypes

constants

void main()
{
    variable declaration           // describe variable
    variable declaration           // describe variable

    // comment set of related statements
    // if comment runs long, use two lines
    statement 1
    statement 2

    // comment set of related statements
    // if comment runs long, use two lines
    statement 1
    statement 2

    // comment what the "if" does
    if ( ... )
    {
```

```
        // comment what the "while loop" does
        while( ... )
        {

            // comment set of related statements
            statement 1
            statement 2
        }

        // comment set of related statements
        statement 1
        statement 2
    }

    // comment what the "for loop" does
    for( ... )
    {
        // comment set of related statements
        statement 1
        statement 2

        // comment set of related statements
        statement 1
        statement 2
    }
}

// comment function
function definition - following commenting and style shown above
```

TOP

Navigate:

Debugging Guidelines

Overview

Unfortunately, professional programmers spend the majority of their time testing and debugging. Students tend to spend an even higher percentage than the professionals, so it is important that you learn to test and debug efficiently.

A *bug* is any discrepancy between intended program behavior and the way it would actually work, under any feasible circumstance. This definition is both subjective, because it involves human intention, and hypothetical, in the sense that a bug can be there even if one of those circumstances hasn't yet occurred.

Bugs can enter at any stage of the software-development process. There can be discrepancies between the customer's intentions and the written specifications, between the programmer's understanding of the specifications and the customer's, or between the programmer's intentions and the code he/she writes because of a misunderstanding either features of the language or of modules produced by other programmers. These bugs are present before the first line of the program has been entered into the editor. Therefore, as a programmer, you should make sure that the specifications make sense and that you understand them. Also, you should attempt to understand your tools, both your programming language and the software written by others that is involved in your project.

Strategy

The later a bug is eradicated, the more harm it does. Obviously, failure of operational safety systems can result in loss of human life, but more often the result is project delays. Because of the compounding effects of bugs, finding and eradicating a bug when other bugs are present is on the average much more difficult than finding and eradicating a single bug. It is important to *eradicate bugs as early as possible*.

To keep the number of bugs in a program at any point in time to a minimum, it is

best to make small, incremental changes to the code and test after each modification. Also, it is important to have the help of as many tools as possible. Always compile with all warnings turned on, and always step through each function with the debugger as soon as you have written it.

Finding bugs is a matter of repeatedly:

- testing, which involves running the program and noting any deviations from expected behavior -- which implies that there must be definitive a priori expectations about the outcome of the test,
- formulating a hypothesis about what caused any discrepancies from expected outcome,
- designing a test for that hypothesis, which typically requires modification to the code,
- testing again.

In a local experiment, we gave graduate students a buggy program and asked them to think out loud as they debugged it. Those who finished quickest were those who had the clearest notion of the hypothesis behind each modification and each test.

From information theory, we know that the test that gives the most information is the one that cuts the possibilities in half. Unfortunately, most students work from definitive hypotheses, i.e., ones that say, ``The cause of the bug must be such and such.'' They perform tests which, if the hypothesis is correct, fix the bug, but, if the hypothesis is false, give no information. This is an inefficient search strategy. (Try to imagine how you would find a particular number between 1 and 1000 by asking questions of the form, ``Is it ...?'' A much more efficient approach would be the standard binary search.)

More than half the bugs introduced by professional programmers are introduced during the debugging process. Similarly, students introduce many bugs and destroy the structure of their code in the process of making changes associated with such testing their hypotheses. It is important to use RCS carefully and frequently so that you can back out of any changes done to test a particular hypothesis.

Asking For Help

There is an old maxim to the effect that it is better to teach a hungry person how to fish than to give them a fish. In a similar sense, the teaching staff of your school is there to teach you how to find and correct bugs rather than to find them for you. They do you (and the institution) a disservice if they allow you to use them as a substitute for an intelligent, orderly effort to do your own work. On the other hand, it is a waste of everyone's resources, most especially your time, if you have to flounder for hours over a technicality that someone could help you with instantly.

To make efficient use of the time of someone you ask for debugging help, have a current, commented, properly formatted listing. The comments at the top of each file should include:

- Name of the file.
- Programmer's name and e-mail address.
- Date.
- What the program is supposed to do and how it is supposed to do it.

In addition, whoever helps you needs to know the exact symptoms of the bug:

- What kind of error it is: compilation, linker, run-time.
- The exact wording of any diagnostic messages.
- In case of run-time errors,
  - A list of what tests that have been run and their results (including the results of debugger tracing).
  - How to repeat the bug, if it is repeatable.
  - An indication of what the correct output would be, and why.

Also, you should know what has been changed since the program last ran as expected, and why it was changed -- frequent RCS checkins can make this easier.

Compile-Time Bugs

Many compile time bugs can be resolved by a very careful reading of the manual or on-line man pages for the exact compiler you are dealing with. I learned this the hard way when my first C, the hello-world program, typed character for character from Kernighan and Ritchie's book, failed to compile despite my best efforts for three days. The infuriating diagnostic message was: ``Bad magic number,'' a concept mentioned nowhere in the book. The error turned out to be that the cc compiler on my new Unix system expected the name of any C program file to end in ``.c,'' a rule mentioned nowhere in the book, but only in the man page for cc.

It is important to know that the command that invokes the compiler is actually a script that invokes either the compiler, the linker or both, depending on the suffixes of the files specified and the options invoked. It is also important to know that all the options specified before the last .cc-file are sent to the compiler and the later options are passed on to the linker.

Compilation Errors

In the case of a compilation bug, the exact symptom is exactly the first error message produced by the compiler.

- The compiler's diagnostic message tells the point at which it (finally) figured out there was an error and a crude guess as to what is wrong. What you

know is that there is an error at *or before* this point -- it could be a long way before.

- Error messages after the first one are suspect, because they are often another result of the error reported earlier.
- If the reported point of error is in a well-tested header file from a standard library look at your source code prior to the point at which that header was included.
- If the compiler fails to give a line number, you should use conditional compilation to cut out code from the bottom, homing in on the offending line via a binary search.

Linkage Errors

The linker requires that every object name with external linkage have one and only one definition throughout the set of object files it is linking, hence there are two possible errors: ``undefined symbol'' and ``multiply defined symbol.'' (Note that *undeclared ...* implies a compilation error, while *undefined...* implies a linkage error -- there is a big difference.)

There is a problem with linkage errors in C++ programs: C++ decorates object names (both data and functions) with type information, producing what are called ``mangled names.'' Linkers that were written for other languages such as C report errors in terms of mangled names, making the messages very obscure. On many systems, there is a command called `dem` or `demangle` that will demangle these names, making them more meaningful to the programmer. Otherwise, the programmer needs to look for a non-type identifier inside the decorations.

In the case of an undefined-symbol error, the programmer should look at the portion of code where he/she intended to define this symbol and determine why the declaration is not a full definition. Often the problem can be as simple as a misspelling. Sometimes it is much more subtle, e.g., the math portion of the standard C library has to be explicitly linked in, via the `-lm` specification, while other portions are automatically linked -- failure to make this specification results in an undefined symbol error, even though you include `<math.h>`.

Another source of linkage difficulties can come when there are multiple versions of a particular library, and the wrong version is linked in, especially when there is a version mismatch between included header file and the compiled library.

The linker has the ability to produce a linkage trace, giving a great deal of information. Consult the manual (via the `man` command) for details on how to enable such tracing and interpret its output.

In the case of a multiply defined symbol, the programmer needs to look at each declaration of the symbol to see why more than one of them is a definition. Often

using M-x grep within emacs can facilitate this process. The next-error command can be used to scan through all occurrences of the symbol in the specified set of files.

Finding Run-Time Bugs

By now all warnings should have been removed from the compilation of the program or be totally explained to anyone assisting. We will discuss techniques for finding hard (reproducible) bugs. Intermittent bugs are considerably more difficult to track down.

The debugger is to the programmer what an oscilloscope is to an electronics technician: the first place to look when you want to know what's going on.

Whenever you write a new function or modify an old one, step through it and watch what each instruction does -- simply set a breakpoint at that function and single-step under your favorite debugger. Use the debugger to set values in such a way that every branch of every construct (ifs, switches, &&, ||, ?:) is exercised. Check the initial values of objects and the values that get assigned.

This should be a matter of habit for all new code, before other testing indicates the presence of a bug. In some cases, it pays to do this at the assembly language level.

As a matter of course, you should also run the profiler gprof on your program to see where it spends its time. To do so you must compile with the -pg option set. If your program suddenly starts running slower than expected, use the profiler to find out where it is now spending its time. The profiler can do a lot to educate you about where the time goes in the running of your program. (Also, don't neglect the time command.)

Aborts

If execution of a program aborts leaving a core file in the directory run gdb *program-name* core and then type run in gdb. It will tell what aborted the program and where, and you backtrack to see where you are in each unreturned function call. Check first in the most recently called function of the modules being debugged -- they are more suspect than functions that they call from a more stable library, and often one does not have access to code from such libraries.

Generally, checking the values of all the variables on the highest call whose code you have access to will resolve the problem. In the {case of library calls, you may not have code for the most recently called function, but it is likely that the bug was in the most recently called function that you are working on and have source code for.

This is preferable to re-running the program to the point of the abort, since that could take a long time and might depend on reproducing certain input or

asynchronous program behavior. Remember that intermittent bugs are by far the hardest to track down, because they are so difficult to reproduce.

Typically there are three kinds of aborts: assertion failures, library aborts, and system traps (bus error, segmentation fault, and arithmetic overflow). Nine out of ten segmentation faults have to do with dereferencing an invalid pointer, usually one with a null value.

Infinite Loops (and other weird modes)

If your program enters a weird mode such as an infinite loop, attach `gdb` to it via the command `gdb` *program-name process-id-number*, the latter of which can be determined by examining the output of the `ps` command. In the case of a loop, one can then trace it. Trace and check the values of the variables at each conditional branch. (If there are no conditional branches, your problem is obvious.)

This method works even when the code you are debugging is not your own and even if the infinite loop is an intermittent problem.

Incorrect Output

You should first of all know what the right output is. Then you should trace back from the statement that produced the incorrect output. There are some rather subtle difficulties one can run into however:

1. Core files generated by programs compiled under SunOS and run under Solaris are not usable on either.
2. One-off errors.
3. A program must be recompiled before it is run on a different architecture, and to completely recompile it, one must first `make clean` to remove all old object files. Also, one must rerun `make depend` because the names of some of the header files may be different. (Perhaps our makefile could be made sensitive to this.)
4. You shouldn't recompile a program while a process is running it. Linux may keep the old version cached, for instance.
5. Don't encode different types of return values via special values, like getch does.
6. Know about one-time flags.

```
static int firstTry = 1;
if ( firstTry ) {
   firstTry = 0;
    ...
}
```

7. Know about fence posts.

```
...
cout << "Enter a positive number:  ";
cin  >> n;
while ( n <= 0 ) {
   cout << "That's not positive.  Try again:  ";
   cin >> n;
}
...
```

or even more succinctly

```
...
while (1) {
  cout << "Enter a positive number:  ";
  cin  >> n;
  if ( n > 0 ) break;
  cout << "That's not positive!  ";
}
...
```

8. Run your program with checkerg++ to check for memory leaks etc.

The Debugging Macros

To facilitate program testing and bug location, we use the following three macros:

- The assert macro is part of the Standard C Library and is defined by including assert.h. If at the point of inclusion the preprocessor symbol NDEBUG is defined, the assert macro has no effect. If it is undefined, however, any subsequent occurrence of assert(*expression*) expands to code that tests the expression and if it is false, generates an error line giving file, line number, and the expression itself, and then an abort, whose core file should be checked by applying gdb.
- The cdbg macro is a localism:

  ```
  #define cdbg cerr<<endl<<__FILE__<<":"<<__LINE__<<": "
  ```

  It prints out a newline, followed by the current file name and line number, e.g.,

  ```
  cdbg << "Bad data encountered!";
  ```

  would print out something like

  ```
  widget.cc:379: Bad data encountered!
  ```

  which can be used by the next-error faciltiy of emacs.
- The debug macro is another localism:

  ```
  #ifdef DEBUG
     #define debug(expression) { expression }
  #else
  ```

```
   #define debug(expression)
#endif
```

which causes stuff to be expanded if and only if the preprocessor variable DEBUG is defined. For instance,

```
debug(
   for( i = 0; i < tableSize; ++i ) {
      if( table[i] < 0) {
         cdbg << i << "-th entry is out of range!";
      }
   }
);
```

Make liberal use of these in your code, and consider them to be a permanent part that can be activated for subsequent testing and debugging whenever the code is revised.

TOP

Navigate:

Guidelines For Software Design

The activities performed by software engineers during a development project can be roughly divided into three phases: requirements analysis, design, and implementation. In *requirements analysis*, the software engineer asks the questions "What program should I write? What are the true needs of the user(s) of the software?" *Design* involves taking the specifications produced by requirements analysis, and devising a strategy for realizing those requirements in a program. *Implementation* is actually writing the code. Thus, requirements analysis, design, and implementation are concerned with, respectively, what to do, how to do it, and actually doing it.

There is a fourth phase that is also important: testing. You may think that software testing is only performed in the latter stages of a project, after or in parallel with implementation. Not true! The testing (or *verification* as it is more formally known) of software is an activity that exists *throughout* the development process – it is intricately interwoven with analysis and design as well as implementation.

It may interest you to know that the value a software engineer possesses in industry increases with his/her ability to work at more abstract levels. Hence, people who design software generally have higher salaries than those who merely implement it; and people who are good at requirements analysis are worth even more.

In a computer science curriculum, requirements analysis is not emphasized heavily (except perhaps in an upper-division software engineering course). Usually the instructor gives you the project requirements, and your task is implementation. Often, especially in beginning CS courses, the presented problem is small, and the solution straightforward, sometimes even obvious. Other times, however, the solution is not so apparent, and students find themselves asking "How should I proceed?" This is the domain of software design. As you progress through your computer science curriculum, you will find

decreasing emphasis on implementation and, correspondingly, increasing emphasis on design. The ability to take a set of requirements and develop a clear and extendible strategy for realizing those requirements is a skill that all successful software engineers possess. This document presents a number of useful *heuristics* (or rules of thumb) that you may find helpful when faced with a software design problem. These guidelines, most of which are based on sound, established software engineering principles, were devised primarily to address the issues that medium- and large-sized programs present. But we believe you will also find them useful for organizing your thinking regarding small (e.g. CS140-style) projects.

Thinking and doing.

Above all, THINK FIRST! Many is the programmer who, always eager to start coding, plunges ahead blindly and ultimately finds him/herself with code that is disorganized, ill thought-out, and worst of all, does not address the true requirements – in short, a mess. At this point the programmer sees two options: start over and think the problem through more thoroughly, or "go for broke" in an all out attempt to get the code to actually work. The temptation to choose the second option is great, because the programmer has already invested considerable effort and may view starting over as an acknowledgement that he/she has wasted time – never a pleasant admission. Moreover, frustration has likely gained a foothold at this point, and the programmer may feel an intense urge to finish the program quickly, simply to get it over with. So the programmer continues, and if he/she is "talented", will, after some long nights at the keyboard, probably get the program to work. The software that results in this situation, however, is inevitably hard to read, difficult to maintain, lacking logical structure, and probably not very *robust* – meaning it is likely to behave incorrectly when presented with input unanticipated by the programmer. All of these problems – which admittedly may or may not get you a lower grade in a class assignment, but all of which are anathema to good software engineering – could likely have been avoided by properly thinking the problem through before starting to code.

This sounds very nice in the abstract, but how do we accomplish it? After all, truly thinking about something can be *hard*. Try brainstorming. Take a sheet of paper and write down the things you know, the things you understand about the problem. Organize the things you know into related clusters of information. Ask yourself how the items in the clusters are related. What are the interrelationships among the clusters? Work through any algorithms in English or in pseudocode. Some programmers find that they are more relaxed and can concentrate better with pencil and paper than they can with a keyboard. Some may even write the entire program by hand before they log on to the computer! This latter approach may be extreme, but its spirit is admirable: at least it forces the programmer to think the problem through before starting to code. Whatever your tastes and temperament, remember that software design is an intellectual and creative process, not a mechanical one. Find the way you work best, then work that way!

Understanding the specifications.

Before beginning design, take a hard, critical look at the program's specifications. Do they make sense? Do they contain unrealizable or contradictory requirements? Do they address how the software should behave in important special cases? What implicit assumptions did the writer of the specifications make? Will the software as specified satisfy the true needs of the customer? Do not be afraid to ask these questions. Conduct a dialogue with the person who wrote the specs (in CS classes this is probably your instructor). As a software engineer, it is your right – in fact, your *responsibility* – to clarify the specifications, elicit hidden assumptions, and determine if the specifications meet the user's needs. It is very rare indeed for a project's specifications to be complete and unambiguous on the first try (or the second, or the third...). Expect refinements, clarifications, and even outright changes to occur. This is not meant to demoralize you. If it does, you may be pursuing the wrong career.

Choosing the programming paradigm.

This is perhaps the highest level, most fundamental design decision. A *paradigm* can be thought of as a model or pattern of thinking. A programming paradigm is the framework you use for designing and describing the structure of the software system. You may choose a paradigm independent of programming language, but the implementation may be more straightforward if the language you use provides abstractions and constructs that directly support the paradigm. The two most popular programming paradigms are the procedural paradigm and the object-oriented paradigm.

The *procedural* approach organizes the program around a set of functions or procedures. The focus is on the algorithms, which are loosely coupled with the data on which they operate. In contrast, an *object-oriented* design conceptualizes the software as a set of interacting objects which encapsulate both data and behavior. In an object-oriented program, individual entities (objects) are responsible for carrying out their own operations.

Neither paradigm is superior to the other in all cases; the one you choose should depend on the application. As with most of software design, choosing the right design paradigm for a particular problem is a black art; there are no hard-and-fast rules. Ask yourself which paradigm more closely corresponds to your mental model of the problem domain. Which is more central to the problem at hand: the *structure* of the data, or *what happens to* the data? If the structure of the data is more important, then it may be easier to organize the software as a collection of object classes, so consider an object-oriented approach. If, on the other hand, the transformations that the data undergoes are more important, then functions or procedures may provide a more useful central abstraction, which suggests the procedural paradigm.

Often a hybrid approach, i.e. one that combines procedural and object oriented ideas, is best. C++ is a good programming language to use in this case, because

it supports both styles of programming. Note, however, that care should be taken when mixing fundamental paradigms. Make sure you use the two techniques in a complimentary, not contradictory, fashion.

Top-down versus bottom-up design. A software designer must deal with much complexity. The chief means for dealing with complexity, in software engineering as well as other technical disciplines, is the notion of *abstraction*. In our context, abstraction involves separating the details of the design into hierarchical levels, whereby one level hides the details of the levels below from the level above. The universal consensus is that abstraction is a vital tool in software design. What is less agreed upon is whether it is better to develop the abstractions starting at the highest level and working down, or starting at the lowest level and working up. This so called "top-down vs. bottom up" debate has raged for quite a while; you can find successful software engineers and computer scientists wholly devoted to one or the other.

The virtues of bottom-up design include the following. It is easier to think in terms of low-level (bottom) entities; they tend to be more concrete, hence easier to conceptualize. So, say the advocates of bottom-up design, it is better to start from the bottom, where the complexity is manageable, and work upward, with each successive level relying on the abstractions provided by the level below. The design of each level, in essence, therefore merely involves piecing together the components and services provided by lower levels, enabling the management of complexity at successively higher levels of the hierarchy. When you find that the services provided by a given level are sufficient to solve the entire problem, you create one final "top" level and you are done.

The bottom-up design philosophy has its drawbacks, however. Perhaps the major difficulty it presents is in determining what the lowest level components should be. How can you know exactly what services are required by higher levels if you haven’t designed them yet? You may design the low-level entities of the software, only to find later that they are inadequate (or unnecessary) for solving the problems encountered at higher levels.

A third approach is the so called "meet-in-the-middle" design methodology, whereby you work top-down and bottom-up simultaneously, with the two ends of the design hopefully coming together somewhere in between. We emphasize *hopefully*. While seemingly combining the benefits of both the top-down and bottom-up methods, experience has shown that successfully joining the high-abstraction levels with the lower levels can be quite difficult. In real life, the top and bottom often do not come together as nicely as planned.

The top-down design approach is often combined with bottom-up implementation. This methodology avoids the pitfalls of bottom-up design, but still allows components to be implemented completely, since all of the components upon which they rely will already have been built. The main difficulty this idea presents is the testing of the components. To test each component, you

must create a test environment. This test environment may not be a useful software artifact in general, however, and so is likely to be discarded after testing of the component is complete. Worse, one is often uncertain whether an apparent "bug" in the component being tested is not actually caused by an error in the test environment!

Finally, consider a methodology that involves top-down design with concurrent top-down implementation. When components (either functions or objects) at a lower level of abstraction are deemed necessary, design their interface and implement them as "stubs". These stubs need not provide meaningful functionality; they must merely act as placeholders around which you can implement the rest of the abstraction level you are currently working on. Best of all, when it comes time to design and implement the stub, i.e. make it functional, a working test bed already exists – the program itself.

Output, then input, then computation.

You no doubt find it satisfying when a program you are writing produces correct output, or at least the output you expect. You gain confidence that you are on the right track. Consider building your software such that the first thing to work is the output, then the input, and finally the computation that transforms the input to the output. Of course, the output you see initially will not be correct, since the computation hasn't been implemented. But when you get the input working, you can start to see how the input influences the output and hence gain confidence that the basic logic and control flow of your program is correct. Finally, fill in the computational details so that the output is the correct function of the input. Throughout, try to always maintain a "working" program – one that behaves correctly, except perhaps for some of the details – and gradually fill in whatever is missing.

Flattening the tree.

When designing a program in top-down fashion, a short, fat design hierarchy is easier to manage than a tall, narrow hierarchy. Try to minimize the design tree's height, instead distributing the design complexity "sideways". Why? Think about working with a team of developers. If the complexity is horizontal rather than vertical, then many different programmers can work in parallel, each developer concentrating on one particular node of the tree. But, you say, I'm just a CS student working on homework projects by myself. Well, flattening the tree is still to your benefit. A node with a deep hierarchy below it is more difficult to design and implement than a node with a wide but shallow descendent hierarchy. The node rooting the wide, shallow hierarchy has a greater number of child "service providers", each of which can help with a different aspect of the current node. Contrast this with the node rooting the deep but narrow hierarchy. Fewer services are available from the level below. Sure, those services may be more powerful, but they are necessarily more complex as well. Distributing the complexity sideways, rather than vertically, allows for the easier distribution and management of the system's complexity.

Modularization and the separation of concerns.

Especially with larger programs, it is vital to organize the design into several subsystems, or *modules*. Each module concentrates on one particular aspect of the system, and often provides services to other modules. This organizational mechanism is more in concert with object orientation than the hierarchical top-down scheme presented above. (Indeed, if your program consists of a set of interacting objects, as opposed to a set of functions, there may not even exist a clear notion of top or bottom.) When dividing a system into subsystems, try to achieve as much modularity as possible. A modular design is one in which the details of one subsystem are invisible (and irrelevant) to the other subsystems. If subsystem *A* does not know about the internals of subsystem *B*, then *A*'s interaction with *B* will be simpler and easier to manage. Also, subsystem *A* will be unable to corrupt the state of subsystem *B*, thus making the system more maintainable and less prone to bugs. To facilitate the effective use of modules, try to organize them such that their interfaces are as clean and simple as possible. If an interface seems awkward or overly complicated, it may be because you have not identified the best breakdown of the system into subsystems.

Identifying application entities.

If you think an object-oriented design is best, then a natural question is "What should the object classes be?" There exists a very popular analysis and design methodology known as the Object Modeling Technique [1]. It consists of a series of general object-oriented guidelines to follow in the creation of software, focusing primarily on requirements analysis and design. The authors of this technique provide a simple yet powerful recipe for determining the object classes that a program should contain. In short, the method is to take a written description of the program's requirements, and find all the nouns. Common nouns correspond to likely candidates for object classes, while proper nouns correspond to instances, or specific objects, of that class. For example, if the requirements specification for some sort of banking application contains the sentence "Human cashiers enter account and transaction data", then this may indicate that the application should have object classes for *cashier*, *account*, and *transaction*. Likewise, verbs in the specification are frequently candidates for operations. So the "account" object class may need to have an "enter transaction" operation.

Code reuse.

There is a common saying among software engineers: "Do not reinvent the wheel." Obviously, making effective use of existing code and existing design hierarchies can greatly increase your productivity. Indeed, software development is increasingly becoming a matter of component assembly and integration. Make sure you leverage existing resources. If you are programming in C++, make sure you learn the basics of object-oriented programming (i.e. inheritance and polymorphism) and generic programming (templates), and then look for opportunities to use existing libraries, especially the Standard C++ Library. In

particular, try to harness and exploit the powerful Standard Template Library (STL), which is part of the standard library. The STL has a plethora of generic container classes, such as linked lists, maps, etc, and generic algorithms, such as sorting, searching, etc, that eliminate the need to hand-code these common, basic entities.

Designing with diagnostics in mind.

The state of an object-oriented program is the collection of states of the active objects in the program. It is helpful to always have the ability, at whatever stage of the implementation, to be able to inspect the state of a given object. Your design should keep this in mind. Consider including a "dump" member function for each object class. This operation should write the contents of the object, in human-readable form, to some output stream. If an object contains other objects as members (i.e. is a *composite* object), then the dump operation for the containing class may call the dump member functions of the contained objects, and so on.

Another useful diagnostic technique is to include a member function of the "assert-valid" variety. An assert-valid member function checks the object for internal consistency, i.e. verifies that the object's invariants are satisfied. An object's *invariants* are those conditions that must be true in order for the object to make sense. This member function should be called during or after all operations on the object, to verify that the operation did not corrupt the object's state.

For procedures, functions, or operations on objects, consider having the procedure or function validate its input by means of preconditions. A *precondition* of a procedure is a boolean function of the procedure's arguments that evaluates true if and only if the arguments satisfy the assumptions made by the procedure. Think of each procedure/operation as having a "contract" with its user (caller): the procedure will behave correctly if the arguments the caller satisfy the preconditions. In C++, the checking of preconditions often takes the form of an "assert" statement (usually the first statement of the procedure).

In short, design your program, and the components thereof, so that they are easily tested, and so that bugs become visible at the earliest possible time, when they are more easily dealt with.

Designing for testability.

What does software testing have to do with design? Everything! It used to be common practice to separate testing from design and implementation. The software was designed, implemented, and then "thrown over the wall" to the test engineers, who subsequently determined whether it was fit for distribution or use. (This approach can be called the *quality assurance* – QA – approach.) It became apparent, however, that this process is seriously flawed: detecting software inadequacies at this late a stage is simply *too late*. A software flaw can result from a *bug* (implementation mistake by the programmer), a design error,

or a misunderstanding regarding the requirements. It is the latter two cases that are devastating. Discovering these types of errors after implementation usually requires that much work be redone, and this is unacceptable.

So what has become more common is the practice of *quality ensurance*. Under this process, the test engineers are involved from the very beginning. They work with the requirements analysts to devise tests that will determine whether the software meets the requirements. They work with the designers to devise tests that ensure the implementation correctly implements the chosen design. Quality ensurance is not an activity or stage of development; rather, it is a state of mind and a commitment.

Therefore, when you design software, you should also design test cases. Even if in a CS class your instructor does not explicitly *require* you to do so, it is still your responsibility –it is *inherently* part of software design.

Prototyping.

One popular design technique we have yet to mention is prototyping. In software engineering, a *prototype* is an executable model of an application or subsystem. Its purpose is to explore those aspects of the requirements, the user interface, or of the program's internal design, that are poorly understood. The chief benefit derived from prototyping is *risk reduction*. Customers (and bosses) are notoriously bad at telling you, the software developer, what they want. They are very good, however, at telling you what they *don't* want. So, creating a quick "mock-up" of the system that the customer can actually see and (albeit to a limited degree) interact with can be a highly effective means of highlighting misconceptions and revealing hidden assumptions about the user interface and how the software should perform. And this can be achieved relatively early in the development process by means of a prototype. Prototyping allows you to reduce the risk of creating software that does not address the customer's true needs.

The other use of prototyping is in investigating the questionable, or poorly understood, subsystems of a program. If there is some aspect of your program that you are unsure about, create a prototype to model and experiment with that portion of the program. The benefits of doing this are substantial. You may uncover assumptions or dependencies that influence the rest of the design. Or you may discover that the subsystem itself was poorly conceived, or its functions totally misunderstood. Obviously, it is far better to discover these things early in a development project. This use of prototyping is the software engineer's version of the old maxim "Deal with bad news first." By doing so, you reduce the risk of having to redo your design at a later stage in the development.

For further reading.

We have only scratched the surface of the fascinating field of software design. Here we list some of the classic texts which treat the subject more thoroughly. A classic work on procedural top-down design is Yourdon and Constantine [5]. Arguably the two most influential books on object-oriented design methods are

Rumbaugh *et al* [1] and Booch [4]. Two other important works on the subject are Coad and Yourdon [2] and Coad [3]. For a comprehensive discussion of programming paradigms, see Sethi [6] or any comparable text dealing with principles of programming languages. The Standard C++ Library (and particularly the STL) has been evolving rapidly, and no authoritative reference has yet emerged, although one reasonably complete introduction to the STL is Glass and Schuchert [7]. For a broad survey of the field of software engineering, see Sommerville [8].

References

1. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

2. Peter Coad, Edward Yourdon. *Object-Oriented Analysis*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

3. Peter Coad. *Object-Oriented Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

4. Grady Booch, *Object-Oriented Analysis and Design with Applications*. Reading, Massachusetts: Addison-Wesley, 1994.

5. Edward Yourdon, Larry Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1989.

6. Ravi Sethi. *Programming Languages: Concepts and Constructs*. Reading, Massachusetts: Addison-Wesley, 1989.

7. Graham Glass, Brett Schuchert. *The STL Primer*. Englewood Cliffs, New Jersey: Prentice Hall, 1995.

8. Ian Sommerville. *Software Engineering*. Wokingham, England: Addison-Wesley, 1996.

TOP

Navigate:

- Style Guidelines for C/C++
- Example of C++ Style and Commenting
- Debugging Guidelines
- Guidelines for Software Design
- Program Documentation

Programming Documentation

1. Specifications

Problem Specifications

The first step is to analyze the problem and precisely specify the information to be used in solving the problem. Clarify any questions.

Give a description of the programming assignment. Most of this is already done for you in the handout that I have prepared. Read the problem thoroughly. Now is the time to figure what questions you have, not once you have started programming. Your program description should also address the questions and clarifications to make sure the program is not vague.

Object Specifications

Describe the objects you will be using. Essentially you will be describing each class thoroughly, which means that you will have had to plan out your classes and how they will work together. When you describe each object, try not to use terms like *class, members, inheritance,…* You should just describe them using English. Remember, code is often read by other people. Someone may know how to program, but not know the language you are using. Essentially, try and make the description language independent.

For each function, describe the problem it will solve. Describe the input to the function, for each parameter you should describe its type, restrictions on its value, what it will be used for, … Describe the output of the function. Then give a brief algorithm showing how the function will solve its problem. Make the algorithm be language independent. Only show the pertinent information (for example, you do not need to show all the variable declarations). Always be sure to state any restrictions and specific requirements.

2. Testing

For each function in the program, testing will need to be performed. One cannot verify correctness, if one does not know what the expected results are for some given output. For each function, pick test cases and specify the expected results. Be sure to pick enough test cases to cover all situations, including errors. For example, if the function was to remove an item from a linked list, you might want to try the following tests:

- remove the item at the head of the list
- remove a few items in the middle of the list
- remove the item at the end of the list
- try and remove an item that is not in the list
- try and remove an item from an empty list

The picking of test cases and specifying their expected results should be done prior to doing any coding. This will make sure that you address all possible errors when coding since you have thought the problem out thoroughly. Then after coding you can verify your tests.

3. User Manual

Describe to the user how the program will work and what they should be entering. For example, when a user menu is given, what is the user supposed to enter (integer, character)? Are there any constraints to what they can enter? What will happen if they enter incorrect data? How do they quit? If they are supposed to enter a name, do they enter first name only, last name only, full name, can that include middle name?

4. Bug Listing

Describe any known bugs in the program. For example, trying to remove an item at the end of a list will produce a segmentation fault.

TOP