# How to do Well on the Projects

EECS 381: Object-Oriented and Advanced Programming
David Kieras, University of Michigan

### Overview

***Why you should be diligent and industrious on the projects.*** Fluency in programming can not be attained by simply reading and studying; you must practice the skills and consolidate your understanding by designing, writing, and debugging computer programs on your own. Thus, there are several substantial programming projects in this course. The main reason, and for most people the *only* reason, why you will have trouble with this, or any other, programming course, is simply not allowing yourself enough time to work on the projects. *Start the projects as soon as they are assigned*; then you can either finish them promptly, or if you have trouble, you have enough time to learn the concepts, get help, reread the book or specs, or simply get some sleep - which is often the most effective way of solving a difficult design problem or finding a bug.

### Working with the Project Specifications

The projects in this course have detailed specifications. Here's how to deal with them.

1. The specification information is found in:

- ***The project document (doc.pdf)***. You will have to read this carefully and refer back to it frequently. There is a lot of information there!
- ***The "starter" files***, including the supplied complete files and skeleton files; both the supplied code and the comments are part of the specifications. The comments especially are very useful. The "strings.txt" file includes not only what message strings the program should use, but also information on which module should be outputting that string (and the associated data).
- ***The demo files*** provided in some projects provide concrete examples of how a specified module is supposed to be used.
- ***The behavior samples*** show samples of how the program is supposed to behave (the "big picture"), and also specify some of the details of the output. Note that your program must reproduce the samples *exactly*.
- ***The Corrections & Clarifications*** page posted and updated during each project - check this before starting each work session and before sending for help.

2. While these materials are prepared by a human who can make errors, they have been pretty carefully worked on. So chances are they are accurate and complete. Therefore, we expect you to have studied the specifications very thoroughly before asking about them. That is, if you ask about something that we believe is in the specs adequately clearly, we will just briefly reply "check the specs" so that we have time free for more difficult questions.

3. If you have done your best to understand the specifications, by all means ask questions if they seem incomplete or don't make sense. In this case, you must tell us what part of the specification seems incorrect, confusing, or incomplete, so that a Correction or Clarification can be prepared and posted if necessary.

### Getting Help on the Projects

***Review the Corrections & Clarifications and the FAQ.*** The project specifications, while very detailed, are not supposed to be confusing to a careful reader. You are expected to ask questions if the specifications are not clear, but check the Corrections and Clarification page for the project first - maybe your question has already been answered. Likewise, maybe your question has already been answered in the project FAQ. Check there first. There are some useful general FAQ pages on the course home page which may also be useful.

***How to get help by email.*** Clarifications and simple questions can be handled effectively and efficiently by email. If you ask a question by email about why your program isn't working, please include the appropriate information so that time isn't wasted by asking you to supply it. Here is what you should do with your email to make this process work well:

- Include the compiler version and platform, and where: e.g., **gcc 4.1.1, Linux, CAEN lab**
- If an error message is puzzling you, copy-paste the *exact and complete* error message into your email. If it is a compile error, you should also copy-paste the statement where the error occurred plus the few lines before it. Linker errors are different, but the exact wording is also important. If you have a zillion errors of the same kind, the first one or two are the most useful to supply.

- Do not expect us to be helpful when you supply only a vague and general description - e.g. “the compiler barfs at my file I/O code.”
- If you aren’t sure whether something is legal or will work, give it a try before you ask about it. “Playing with code” this way is very educational, and the actual results of bad code make it easier to understand and explain what the problem is.
- Since we often read mail with efficient plain-text mail programs, please turn off all the html and styles nonsense in your email program - don’t make us read through gobbledegook when we try to help you.
- Do not send big hunks of code, especially as attachments, unless we ask you to; the skill of isolating the problem and asking a well-framed question is worth your developing. We will want the big chunk or whole file only if we are puzzled and need to compile it ourselves to figure out the problem. Unsolicited files will be deleted without inspection.

Rather than simply telling you what to do, often our answers will also explain what is going on, and provide useful tips: E.g. why a particular way of reading the input doesn’t work like you thought it would, or how to figure out a yard-long error message. Please take the time to read and digest such answers when we have taken the time to write it - it will add to your skill and save you a lot of time in the future when a similar issue arises.

***How to get help - in person.*** Even very experienced programmers encounter serious bugs that seem to yield only when another person looks at the code. Less experienced programmers often benefit from some coaching on debugging. So for really nasty bugs, come in person to the office hours, and we will work through it with you. Again, our primary goal is to help you learn how to do this better, so we will be expecting you to have done as much as you can, and to be prepared to explain what seems to be going on and what you have already checked out. If you have a conflict with the staff office hours, then email us to make arrangements for a special meeting outside of those hours.

***Beware of legal but lousy advice***: I have often seen students get “advice” from other people that turned out to be ignorant, mistaken, dead wrong, obsolete, superstitious, dangerous, or contrary to the approach and guidelines presented in this course. Other people have found similar bad “advice” by searching the web. While it is possible to get such “advice” legally with regard to the Academic Integrity policies for the course (see the Syllabus), it is likely to mess you up. The material presented in this course is my digest of the best advice available from the best people, the “gods” of programming. Making it available to you is a key purpose of this course, so review the course material first, then ask us - there is no need to go fishing elsewhere. For example, the code examples on the course web pages are likely to show you exactly what you need - why rummage through the whole web?

***Beware of illegal advice***: Reread the section in the Syllabus about academic integrity. If you get help with the actual design, writing, or testing of your code from anybody except the TA or Prof, it is a violation of the course rules. *If detected, you will be reported to the Honor Council.*

## Project Grading and Evaluation

***Code quality.*** The general quality of the code will be emphasized throughout this course. The principle is that in the real world, code is often used, maintained, and modified long after it has been written. This means that code is read and modified far more often than it is written. Therefore it is critical that the code be easy for a human reader to understand. It must be straightforward and simple, make good use of the language facilities, have good organization and structure, and be *idiomatic*. As in natural languages, an “idiom” is the common or accepted way to say something, which has evolved over time to work well in that particular language. Part of becoming a good programmer is learning what the idioms for a particular language are, understand why they work well, and then use them where applicable.

Some examples of poor code quality are:

- Unnecessarily duplicated code (“copy-paste coding”) - the most common and worst bad habit of beginning programmers.
- Lack of well-designed sub-functions that eliminate duplication and improve code clarity (the everything-in-main newbie error).
- Convoluted, unnecessarily complex, hard-to-read code, such as excessive nesting, clumsy functions, or bad layout.
- Egregiously inefficient code - a bit of inefficiency can be a good trade for simplicity, but inefficiency with nothing gained in return, or serious inefficiency that could be remedied easily, is just bad design or sloppy work.
- Bizarre (non-idiomatic) code - such as oddly specified for-loops for no good reason.
- Code that fails to follow course guidelines and recommendations for increasing reliability, clarity, quality, portability, and reusability. There is a reason for these principles, developed over decades of experience by professionals. Learning and using them is a major point of the course. One example is the guideline not to use leading underscores for names - a source of difficult and completely unnecessary errors.

Ideas about code quality will be presented throughout the course: in lectures, in the handouts on specific topics, in the textbooks, and the grading feedback on the projects. Some very specific and critical rules are presented in the Coding Standards documents for the

course that are posted on the course web page.You will be expected to take advantage of these ideas and improve the quality of your code substantially during the course.

Even if they get all the autograder points, students typically make a horrible mess of the code quality on the first project, and have to struggle for the remainder of the semester to recover from it. In almost all cases the cause is simply ignoring the guidelines for writing quality code. The Coding Standards documents contain specific industry-based rules chosen in light of the mistakes made by students in the past. Hint: Pay attention to this resource! Plan your work so that you have time to compare your code to the Coding Standards and other guidance and revise it before the deadline. Turning in garbage early is no way to earn a good score in this course!

***Living with the autograder.*** Most projects will be graded for correct behavior by an "autograder" computer system which will compile and execute your program, feed it with a set of test inputs and determine if the outputs match those produced by the instructor's solution. Additional tests will check for certain aspects of whether your code meets specifications for content and structure. For example, on most projects, we will combine components of your program with components of our own and see if the resulting combination compiles and performs correctly. This is a test for whether your components conform to the specifications for their interfaces and behavior. The autograder system is actually very similar in concept to the automated testing systems that sophisticated software development groups use, especially those that follow the tenants of "agile" or "extreme" programming - "write the tests *first*," "no code exists except to pass a test." The difference is that you do not know what our tests are, but this is because we want you to devise your own tests based on the specifications - learning how to do this is a critical software skill. See the web site essay "General Suggestions for Testing Your Project Solutions" for some ideas.

Our goal with the autograder is to provide accurate and timely feedback about the correctness of your work faster and more consistently than possible with human graders. But the grading system is complex, so please bear with us if we have difficulty with it. Any problems that are our fault will not be held against you, and allowance will be made for delays or bugs that are our fault. But do not mistake our corrections and adjustments of the grading system for leniency or slackness in the grading. Announcements and web pages will keep you posted on the policies and status of the grading system. See the course web page "Autograder Information and Policies" for more details.

***Suspect your code first.*** Students have sometimes complained that the system was grading their projects incorrectly, claiming that they *know* that their code is correct. However, in almost all cases, the student had not tested his or her program at all thoroughly, and it turned out to simply not work as specified. Student programmers have to develop the same humble attitude about their code that professionals have: *There are always bugs in your code; you just haven't found them yet!* So suspect your own work before doubting the grading system. That said, no matter how hard we try, the grading system is still a program with potential bugs, and flaws in specifications and testing are a fact of real as well as academic life. Note that significant problems that are our fault will not be held against you; with luck, they will be uncommon.

***Code with undefined effects is incorrect code.*** It is possible to have a program that appears to run correctly in your own programming environment, and then fail miserably in the grading system. In almost all cases, this is due to the fact that your code has errors that are tolerated by your programming environment but not by the grading system. Typically, such errors result from your using nonstandard syntax or facilities, but most commonly are a result of relying on ***undefined behavior*** in the C/C++ language. For example, in Standard C and C++, uninitialized variables are not required to have any particular value, but depending on the platform and implementation, uninitialized variables might have either a "good" value in them (such as zero) or simply random garbage. If your code expects a "good" value in an uninitialized variable, the result can be inconsistent platform-dependent behavior.

Another example is that some implementations of strcmp will return -1, 0, or +1, while the Standard just calls for values that are negative, zero, or positive. If your code assumes that exactly -1 will be returned, it might work correctly in some C/C++ implementations, but not others.

*A program that relies on undefined behavior is not a correct program; no credit will be given for an incorrect program that happens to run correctly on your platform*. If you submit your projects to the grading system early enough, any such problems will appear and you can get help with them if necessary. Compiling and testing your program on another platform prior to submitting it will often reveal such problems.

## A Final Word

***The Perils of Procrastination.*** Since the very beginning, computers have always been equipped with hidden dedicated deadline-detection and programmer-stress level sensors that are hardwired to random bug and catastrophic crash generators. This is a fact of life; your only defense is to allow extra time. Even professional software developers can't reliably predict project completion times, so it is foolish for students to push their luck! Crowded labs, flakey computers, dead hard drives, or slow networks are no excuse, and you will get no sympathy if you start a project late and then run out of time. By far, the most common reason that students have

problems in this course is simply waiting too late to start the projects. The course schedule and projects are designed so that you can start working on the projects right away, and all the concepts and ideas needed are presented at least a week before the due date. So do your all-nighter a week before, not the day before! You'll either get the project done, or have time to get help!