How *Not* to Go About a Programming Assignment

Agustín Cernuda del Río
Department of Informatica
University of Oviedo
33007 Oviedo – Asturias, Spain
guti@lsi.uniovi.es

Abstract

Computer programming students invariably fall into more than one bad habit. It can be extremely difficult to eradicate them (and many lecturers and professional programmers keep succumbing to them time and again). I wrote this when, in the days leading up to an assignment deadline, I saw these things happening so often that I couldn't help but recall my classmates and I a decade earlier… doing exactly the same things as my students. This article is an attempt to show these irrational attitudes in an ironical way, intending to make our students aware of bad habits without admonishing them.

Keywords: Programming assignments, best practices, humor

1. Programming, in a Strict Sense of the Word

1.1 Ignore Messages

Compilers, operating systems, etc. generate error messages designed only to be read by their creators (maybe to justify their salaries). Precious time is wasted reading these messages; time that could be better spent … writing code, of course! Error messages make us less productive. Don't fall into the trap. Ignore them.

As for warning messages, ignoring them makes you feel like a professional programmer who's not scared of computers. What better way of showing one's experience as a programmer than delivering a program that generates dozens, no, hundreds of warning messages when it compiles without its author feeling the slightest bit concerned? Everyone can see that you're an experienced, laid-back programmer who is too busy to waste time on drivel.

1.2 Don't Stop To Think

Let's not kid ourselves here. What are we building? A program. What is the only thing that really matters in a program? Code. What really works? Code. Why use outdated resources like pencils, pens or paper? You are a paid-up member of the SMS generation; you don't make a fool of yourself writing time-consuming syllables, right? Then, stop messing around thinking about nothing when there's so much code to write.

You should never stop coding. We all know that error messages are an unacceptable interruption, a pointless obstacle as we go about our work. So what do you do if you get a compiler error message? As you should know by now, reading and understanding it is just not an option.

You can try making some random change to the source code. You never know, you might pull the wool over the compiler's eyes. But if this doesn't work, don't waste any more time. NO, don't be tempted by trying to read the message or understanding it. Just keep churning out code - that's the only way of finishing off this horrendous assignment. You'll get to sort the error out later on. And as we all know, errors tend to disappear by themselves if they're ignored. At the end of the day you'll compile, you'll run, and even if you had tested (not that you needed to) you'd have seen that everything was OK.

If the code compiles but does something wrong, it doesn't really matter; sort it out later, when it's finished. Anyway, you might get lucky and find out that the lecturers have changed the assignment outline and that it fits in with your program after all. So don't take the risk of fixing programs that seem to be off track – you might be wasting your time.

1.3 I Don't Want Any Trouble

If your program contains a bug that crops up every now and again, it will be difficult to find and it won't probably show up during the exam demo. Maybe it will disappear by itself. Don't worry. But if the bug comes up again and again, change things at random until it disappears. We've already said that pausing for thought is not an option. If you decide to get rid of the bug - simply because the urge takes you - just write the same code in different ways. Maybe the problem will disappear; something you'll have achieved without 1) understanding what caused it, and 2) having to stop writing code. Clearly, this is the most professional approach.

Don't compile on a regular basis, don't tiptoe your way forward. You're a professional and professionals take giant steps. Write thousands of lines of code first and leave the compiling for later; it will be far more entertaining and worthwhile to look for compiling errors.

The same rule applies for runtime errors. If you try to keep your program correct as it grows, it will be too easy to pinpoint a new bug. Only cowards do that. A real programmer writes the entire program and then digests it whole like a boa constrictor. Looking for a bug hidden in the last 10,000 lines is exciting but if there are only 10 or 20 lines, well, what fun is there in that?

And… why use debuggers? It's up to the lecturer to look for your bugs. Programming errors are the exception, not the norm, and when you become a pro you won't have to face them. Why waste time then or expend your energy learning to deal with them as part of your education?

2. If Only I Could Find the Words

2.1 Reading

Outlines and specifications are a real drag. These tedious and long-winded tracts refer to irrelevant problems and are nothing more than an opportunity for lecturers to display their narcissistic traits. You only need to take a quick look at them and get the gist of what they are after. Reading them for a second time only gets in the way of our real mission, which is nothing other than… writing code. So once you've got a rough idea of what's expected from you just stick the assignment outline at the bottom of the biggest heap of paper on your table.

On the other hand, coding and presentation rules show how arrogant our lecturers are. They like controlling us, forcing us to do pointless exercises - that's why they write rules in the first place. Don't play their game. Reading or applying rules won't make our work any better or worse. And as for making our exercises easier to handle, well, they get paid to correct them, don't they? Don't even bother to put your name or your class on them. Lecturers will have little trouble remembering your face and your unmistakable programming style so they'll know it's yours anyway.

2.2 Writing

Don't write comments. We've said it before and we'll say it again: what's the point of all this? To create a program, i.e. code. Non-executable stuff is unnecessary and explanations are an insult to a programmer's intelligence - after all, he or she can read the source, right?

If there are mandatory comments to write (function descriptions and stuff like that) then write them, even if you have nothing interesting to say. Lecturers like this drivel and you'll get higher marks.

As for the docs, write them at the end. How can you write a document describing a program that doesn't exist yet? What's the point in writing documents *for yourself* about what *you've* just done? The only reason for writing documentation for a program is that the lecturers ask for it. It's something you can sort out the day before the deadline. What's more, there's no chance of you forgetting anything as it will all be fresh in your mind.

Also, use abbvns 'n strange konsonants when u write. Lecturers are old fogeys. You are a member of the SMS generation. Try to write messages that are difficult to read. Although he might not notice it, the lecturer will have to make an extra effort, after a long day stuck in front of a computer screen. All of which should help raise the old concentration levels and put him or her in a really good mood.

What about spelling? Spelling is a drag. Even Juan Ramón Jiménez[1] put his letter j's wherever he wanted and Gabriel García Márquez once called for spelling to done away with for good. Obviously, you're just as good as them and so you've just as much right to write however you want.

Let's face it, who doesn't make spelling mistakes? It's all too easy. And there's a brutal poetry in abrupt contractions and semantic hijackings that fling treacherous letters at the reader. Ever wanted to give your lecturer a slap in the face but never had the guts to do it? Drop him a line such as:

I'm trying to do you're exercise. I think its two difficult.

It'll have the same effect, don't worry.

3. Your Relationship With Your Lecturer

3.1 Don't Ask For Help

If there's something you can't do, if you have a query or if you're lost, don't look for help, don't ask questions during the lecture and don't go to your tutorials. There are thousands of reasons why you shouldn't but here's just a few of them:

- Going to a tutorial and asking questions is tantamount to admitting you're stupid.
- Better to be ignorant than to run the risk of revealing that you don't know something you should.
- Ask a question during the lecture and your fellow students will think you're stupid. You don't think that of them when they ask a question, but they will about you. This argument holds true for each and every student in a lecture room at any given moment; that's why none of them ask any questions. Conclusion: never ask for help or go to a tutorial.

There is, however, an exception to this rule; you are allowed to turn to the lecturer in the last few days before a

[1] A Spanish poet who won the Nobel Prize for Literature in 1956. He liked to flaunt spelling rules by writing almost phonetically (in Spanish the change involves only a handful of letters, g/j among them).

deadline. There may well be a long queue, he will dedicate his time to helping students while neglecting other duties but don't worry, he won't be able to resist helping you in those dark, gloomy hours of need.

3.2 Challenge Your Lecturer

If, despite everything we've said, you decide to ask for help, always remember a golden rule that'll also help you in your professional career - after all a whole host of pros and computer users follow it too. NEVER give a detailed description of a problem.

Here's an example. If something untoward happens while you're building a program, go and see the lecturer and tell him: "Something strange happened with my program yesterday." He'll look at you expecting more details but don't give in, don't say anything else. Don't even think of going into details such as:

1 Whether the strange event happened while compiling the program or while running it.
2 Whether the strange event caused the program to end suddenly or to keep running indefinitely, or simply, the program didn't do what you expected.

Here's another one. If the strange event happened while you were compiling, don't tell the lecturer what the error message said or the line of code where it appeared. Just say something like: "It gave some error message, or something."

Here's yet another example. If the strange event happened during runtime and caused the program to terminate suddenly, never write down the error message or tell the lecturer what it said. Just say: "It gave some error message, or something."

Of course, if the strange event involved the program not doing what you expected it to do, don't even think of telling the lecturer the exact circumstances of how it happened. Avoid descriptions like: "This error comes up whenever I load a second file and the first one was empty." Just say the magic words: "It gave some error message, or something." Have you got that?

Let's suppose that you're a stubborn ingrate who goes see the lecturer to ask about a *specific* problem. That's two mistakes rolled into one but you can at least get something right - take the wrong source code with you. If you have a bug and the things you try out only make the situation worse, take the most recent code to your tutorial but ask about the original problem. That way the lecturer will embark on a fruitless search for an error when, in actual fact, another one will show up. When it does, just say something like: "Oh yeah, I tried something out. Delete that line there ..." Perfect this art and you'll be able to do a whole coding session in the tutorial. I know - I've seen it done.

If you insist on being irresponsible and asking for help in tutorials, don't even think of pinpointing the problem before you go. If there is an error in a 1-MB input file, don't try smaller files until you identify the cause of the error. Don't try to create a mini-program with that selfsame error. If you do, the lecturer will probably find the problem straightaway. What kind of challenge is that for him? Better to make him read thousands of lines of code and make traces with hundreds of steps. That'll give him a chance to practice his clairvoyance skills and you'll be able to check out his powers of deduction.

3.3 Be Clever When Using Electronic Mail

Some questions are almost impossible to answer by e-mail, if you put them in the right way. Nurture this skill and make your questions completely vague. Here's an example: "It gave some error message, or something. I've attached the source code". You can go the other way as well, if you want, by asking a more specific question but forgetting to send the code. "The constructor in my TDevice class gave some error message, or something." It goes without saying that you should write your message straightaway and send it. Never reread messages.

There's another reason why email is so much fun. You can sound off without the guy knowing which group you're from or your name. Everything will be OK if you take the informal approach – it makes it all so much cozier, making your name an irrelevant detail.

4. And, Of Course...

4.1 Leave It All for the Last Minute

Right from day one your lecturers will tell you to hand your work in the following week. They'll tell you to work at a steady, constant pace from the off. Don't listen to them.

Although it might be a relatively new discipline, computer programming has already built up a number of sacred traditions, one of which is the last-minute rush to get your work in on time. Subjecting yourself to this stress is an essential part of preparing yourself for the world of work. Relax. Let your work pile up gradually and blithely ignore all the warnings and telltale signs that you're behind schedule. Don't let studying get in the way of your life. Don't duck out of that skiing trip in a vain attempt to make up for lost time. And just when you're on the edge of the precipice, just when you've only got two weeks to hand in a program that you've had four months to do, then the code will start to flow like there's no tomorrow.

What attraction would computer programming have if we didn't put together programs in a breathless, last-minute dash? What would become of the image of the long-haired, bearded, smelly (there's no time to shave, trim your beard or have a shower, you see), Megadeth-T-shirt-wearing programmer (remember that stains show up less on dark Heavy Metal T-shirts with their elaborate designs) tapping away at a keyboard for 48 hours non-stop? Would you have the stamina to go to the local LAN Party, park your bum down on a plastic chair and spend three days cooped up in a marquee in 35-degree heat gunning down monsters on a screen? What right would we have to call ourselves heroes if we had a kip every day just because we felt a bit tired? Just think about it. What would happen to Coca Cola and all its factories? What would happen to Juan Valdés? (Valdés is the name of the coffee grower in Café de Colombia's TV adverts.) And what would happen to all the coffee factories that dedicate half of their production to computer programmers? When Sandra Bullock and Robert Redford became hackers, did they put their notes down by the side of the computer, sit and think for a while and then methodically tap away on the keyboard for an hour or two before heading off to the gym or the bar on the corner, day after day for four months? And what about that bloke in Operation Swordfish? Would he have cracked the Pentagon password if one of Travolta's hitmen hadn't been pointing a pistol at his head while another Travolta hit woman was trying to distract him? The answer, my friend, is no. You want an easy life? Go and take another course.

Being up to date with your work and understanding what's going on in the lecture room is for swots and wimps. You know what to do - leave it all for the last minute.

4.2 Cheat With You're Assignment

Copy the programs. Lecturers will probably have to mark dozens of them, making it difficult to spot similarities between them. And even if they do, it sure as hell ain't easy to prove. Appeal against your mark and take it to the High Court if necessary. That will take much more money and effort than writing the programs, but the goal is to prove that you're smarter than the lecturer and never, ever give way.