How to Program

This document is meant to help you structure the way you are thinking when writing a program. There are many difference approaches and not everyone will end up internalizing the programming skill in the same manner.

However, the following is the approach you will have to take on every single assignment in this offering in order to develop a way of thinking, which will enable you to start solving computational problems.

If you are already well versed in programming, feel free to apply what you've already learned with experience instead.

An overview of programming

Since we are not teaching you how to engage in software engineering yet, we are going to keep a low level focus on the activity of programming without going into all the methodology meant to handle your client's requirements and maintain the software you produce. These are important but you will have other offerings in which you will study these aspects of software development exclusively.

So this offering will assume that you are about to write a small program for someone who knows what they want but is not familiar with programming at all. The best this person might do is providing you with **requirements** for what the program is expected to be doing. Starting with this perspective in mind, here are the steps from these requirements to a functional program.

- **Defining the requirements** – This will be already done for you in these assignments. Generally, this entails a lot of back-and-forth dialog with your "client". Here, you may use the forums to ask for clarifications, it's part of the deal.
- **Designing the solution** – you take the requirements and figure out how you may use the tools at your disposal to build a program satisfying all the requirements.
- **Implementing the solution** – transforming the design you built on a piece of paper using pseudo-code or flowcharts into a computer program.
- **Testing your solution** – It is safe to assume that very few programmer design and implement a program which they know is not satisfactory with respect to he requirements. However, programmers make errors constantly so you need to validate somehow your solution before to move on.

Many textbooks will lead you to believe that these steps occur in succession, one after the other, in the sense that you need to complete the design before to start implement, and you will only test after you have implemented all the features described in the requirements document. Modern programming methodologies emphasize the need to use a more iterative process instead.

Process #1 – The main programming iterative process

Let's say that you are looking at a requirement(s) document for one of our assignments. You will go through the following steps as you work on your program;

1. You read the requirements and identify a small feature from it with which you may get started.
2. You design a program satisfying only that part of the requirements. Think about which of the tools you learned about you need to use; loops? Conditionals? Functions? And how you would use them e.g. sentinel-controlled vs. counter-controlled loops? These concepts should have been taught or learned by you in IT Programming Concepts.
3. You then take your design and implement it in the programming language you are working with. Going from the design to the implementation is trivial, assuming you do have a design in mind or on paper. If not, you are trying to play the piano at the same time you compose your music; feasible but not for novices.
4. When your program is actually written, implementing that one single feature, you are going to compile it. Ideally, you should even compile every time you have written a full statement. A quick compilation takes a few seconds and it's easier to spot the missing ';' in the 3 lines you just added from the last successful compilation than writing a whole 50 lines and then figuring out you have 300 compilation error messages
5. Now that you program is syntactically correct, try to run it on one single test. No crashes? No bad results? Good, but now we need to test it seriously.
6. Next, you are going to test your program, see the sub-section below on **Process #2** – **"How to test your program?"**
7. When your program implements the feature you picked and you made sure it is working, it is time to take a few minutes to review and improve your solution, go take a look at **process #5** – **"How to refactor your programs"**.
8. Add comments or modify the ones you wrote hastily while programming, make sure the indentation is up to standards,
9. Then you are free to go back to the beginning of this list, pick another small feature to implement from the requirements and expand your program.

Process #2 – How to test your program?

Testing your program means determining a list of tests. A single test is made of input values and expected output values. These might be parameters passed to the program, data values found in files, or even just parameters you will pass to a few functions you are writing when another programmer use them. As we progress, we will define more exactly, what we mean but for the first assignments, this will be as easy as following these steps:

- **Input values** are what the user types when prompted by your program to provide some data
- **Expected behavior** is the output or results the program is expected to display on the screen, based on the requirements.
- **Observed behavior** is what you see when running your program

It is important you figure out which input values ought to be tested, this is a skill in itself, and you will get more advice on this as we start more evolved assignments.

When you have decided on a given input value, you need to determine what the correct behavior or output for your program should be. This is your expected behavior. Do it by hand, this is the standard by which you will determine whether your program is working or not.

Any test, for which the **observed** program behavior on a given set of input values does not match the **expected** behavior, means that you have a bug of some sort and need to troubleshoot. This entails many steps; see the subsection on **process #3** below

When your program is passing all its tests and you have enough tests to be able to guarantee its correctness, then you may go back to item 7. in process #1.

Process #3 – How to troubleshoot your program?

If you are troubleshooting, this means that you thoughts you wrote your program to achieve a given behavior but you are right now staring at a screen, which shows your program doing something unexpected with the values you just entered.

In 99.999% of situations, this is not an issue with the hardware, operating system or language tools. You made an error, it happens, let's fix it. You need, again, a thought process to help you find the error;

1. Identify the part of the program responsible for the misbehavior. Troubleshooting is harder than programming. Do not test your code after writing50 lines. Test it often so that the error is most likely in the last few lines you just added.
2. Understand **why** the program features this misbehavior or erroneous output based on the input values and the code you wrote. This means tracing your program's execution by hand, see subsection on **process #4** below.
3. Determine the modification you need to apply to your program to fix the problem. Notice that we are back in a design step here.
4. Implement this modification. Sounds familiar?
5. Go back to process #2 and run the test again to see if you fixed the problem, if not, go back to the beginning of this list until you get it.

Process #4 – How to trace your program's execution?

If you are here, this means that you need to know exactly where things went wrong. You thought that writing something like

```
If (x= 42) {
        printf("x is 42\n");
}
```

Was going to display a message only when x is 42 but your tests revealed that it does when x is 0 just as well.

You may keep looking at the program and repeat yourself what you intended for it to do but it won't help you find the error. Instead, you need to take a piece of paper, go line per line through your program an "execute" each of them by hand. The paper will help you keep track on the values of each variable as your program modifies them.

If you do this very rigorously, without assuming anything, you will be able to see that we do not compare x with 42 but assign 42 to x.

You will not spot this if you are reading too fast, making assumptions instead of interpreting your program as a machine would, or do not have the knowledge of what each operator and statement in the language does.

If you lack this knowledge, though, it would be hard to write a program. However, remedying to the situation is easy since this is mostly a matter or memorization rather than learning a new way of thinking.

Tracing code is a basic skill taught in IT Programming Concepts. Trying to program without this skill is like trying to write on a piece of paper in the dark. Email your instructor immediately for help if you never learned to trace programs in your previous programming offerings.

Process #5 – How to refactor your programs?

Now that you successfully implemented a feature, let's focus on the quality of the code you are delivering, i.e. the way you wrote your solution.

Make sure you understand that we are not adding any feature here; we simply try to find a way for our program to do the same thing more elegantly. This is referred to as *refactoring* by programmers.

Examples of refactoring;

- Rename variables / functions / parameters to more meaningful names
- Remove parts of the program now useless e.g. a function you never invoke
- Simplify complicated designs such as using 3 loops where you really need only one.

Each time you make such a modification, go back to step 4 in **process #1** to make sure you didn't break anything before to do the next modification.

Remark – How important is testing?

It is as important as designing the solution. If you do not understand, from the requirements, how your program should behave when provided with some input values, how do you think you will be able to write it?

We will talk again about testing in the near future. For now, here are some things tests are able to help you with in novice programming offerings;

- Missing requirements – you forgot to implement a feature
- Misinterpreted requirements – you thought you understood what was required but you didn't
- Logical errors – your program doesn't behave as expected
- Ambiguous requirements – you implemented what the requirements dictated but your client is not happy; apparently you missed the point they failed to make clear, it's your own fault (Just kidding). Generally, this leads back to the "defining the requirements step and some more discussions with your "patron".

As a side remark, some programmers look at the requirements and write the tests before to write the program. Others do the opposite. In our situation, it doesn't really matter. We just make sure we pick a small enough feature that the whole programming / testing deal should not take us more than 30 minutes.

Remark – So is it ok for me to make errors all the time?

The Short answer is YES!

I hope you got a glimpse at how programming is really an iterative process where we make many errors but we have a process to catch them, understand them, and improve our program from what we understand.

- You will have many compilation errors, but they will be located inside a few lines and you will be able to fix them in a few minutes each.
- You will have many logic errors, but since we test every small piece of code immediately, you will not have to look far to find out what happened exactly.

Understand that programming is really not about sitting down and writing a perfect program in one swoosh. Unfortunately, that's how textbook make it look like when they show you a problem and then jump directly to the perfect solution. If programming was about memorizing problems / solutions pairs, that might work. Programming is the **process** of building the solution little steps at a time from the requirements. This is why watching the "apprenticeship videos" is important; they focus on the very skill you need to learn.

As long as you have a "plan", a thought process to program, then you are able to take errors and turn them into the next and better version of your program. If you are simply typing a loop, statement because you feel that it might be what you need because you vaguely remember that another exercise, which looked almost similar to this one, was

using a loop, then you have not learned to program and are lost. If you feel you might be in this situation, email your instructor to set up some time to go other material.