This document is a list of programming advises and common knowledge.
After reading it, you'll be at least able to have some level of understanding of what your code actually does.
Its goal is to improve your programming abilities, or at least give you a better understanding of computer-science related concepts. The information given here is certainly not exhaustive.
The tone is made to be as light and clear as possible, and sometimes, the explanations may overlook some technical subtleties, although I'll do my best to prevent this from happening, or to at least tell you when they do.

Also, I'm a computer science freshman, so I'm still mostly self taught; I therefore cannot provide a guarantee on the accuracy of my knowledge.

Theme 1: Turing Machines, i.e. how it all started.

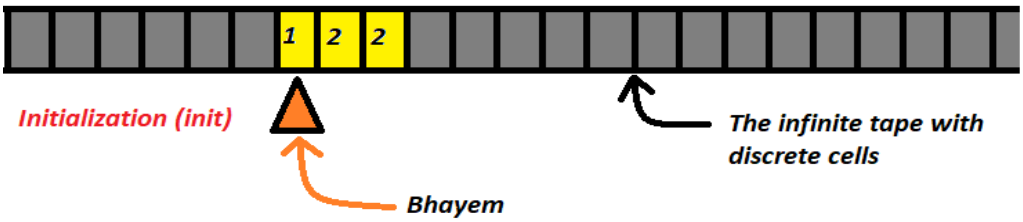
Before diving in programming on itself, I want to talk about the concept of a computer.
In my view, one of the best models for computers is what we call Turing machines, which are named after Alan Turing, whom I dare call the father of modern computer science.

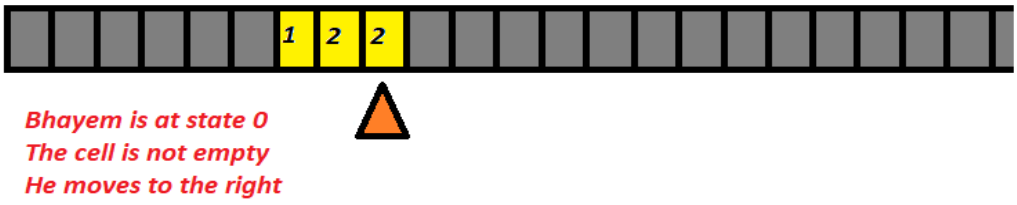
A Turing machine is not a "real" machine; rather it is a conceptual model that Turing conceived to address the "Computability Problem". Basically, the computability problem is whether or not we can find the solution for a given problem. From a programmer's perspective, it can be poorly simplified in a single question: "Will the program find the right output and stop?". That question, on itself, is something you should think about when conceiving solutions to problem. Unfortunately, this is rather a matter of algorithmic, and even though I looooove algorithmic, this document is about programming.

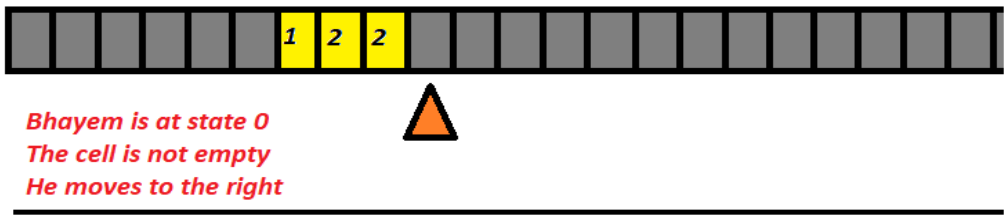
So let's actually define what a Turing machine is, since it is a crucial notion of computer science.
A Turing machine can be seen as a really dumb and obedient person who has a tape of infinite size that contains an infinite number of discrete cells. Note the term discrete here, we are dealing with something on N and not on R.
That person also has a paper on which someone wrote a list of instructions they have to follow. As they are very obedient, they will follow it.

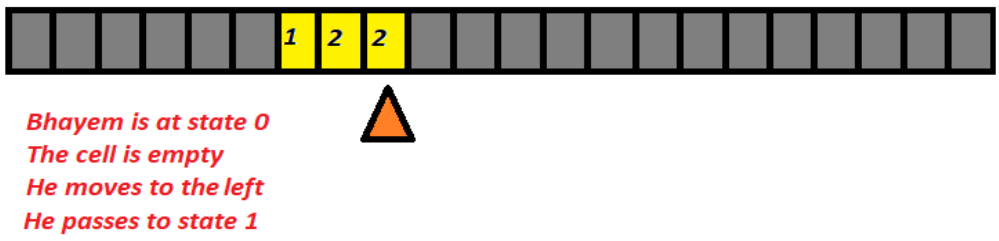
Since concepts can be vague, let's take a look at a special case.
Let's consider that our dumb and obedient person is named Bhayem, here is the Turing Machine that computes incrementation by 1, and its execution:

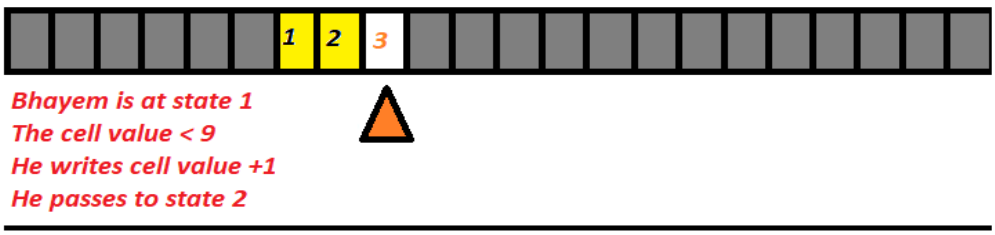
**don't mind if you don't know what it is, I plan on making a section on abstraction and levels of languages soon*

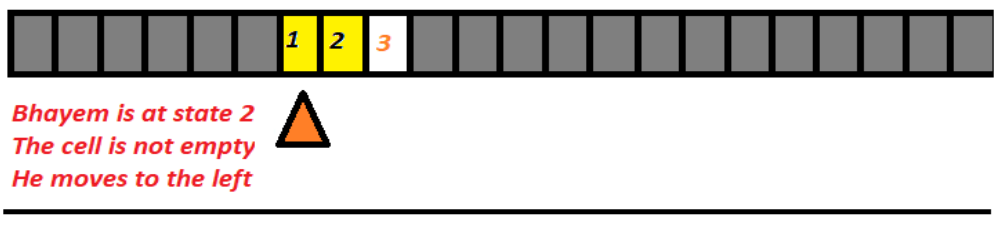


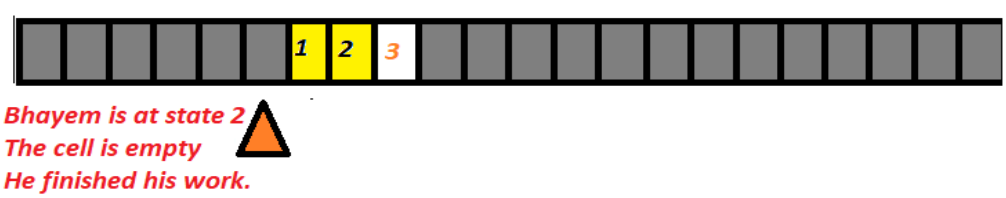













```
State 0:
    If(cell is not empty){
        move to the right
    }
    If(cell is empty){
        move to the left
        goto state 1
    }

State1:
    If(cell <9){
        write cell +1
        move to the left
        goto state 2
    }
    If(cell = 9){
        write 0
        move to the left
    }
    If(cell is empty){
        write 1
        you finished
    }

State 2:
    If(cell is not empty){
        move left
    }
    If(cell is empty){
        you finished
    }
```

As you may have understood, the Turing machine is analogous to a computer.

The tape is the memory space; the sheet of instructions is the code; and Bhayem is the processor. However, there is some main obvious differences, in real life, the tape is not infinite, though your average laptop has 500GB of hard disk, and 8GB of RAM (Random Memory Access, I'll talk about it someday). The 500GB means that your hard disk is a tape of 500 000 000 000 cells, and the 8GB of RAM means that your RAM is another tape containing 8 000 000 000 cells.

Unlike Turing Machines, though, the tapes in our computers can only be either blacked or whited, which means that it can only contain binary values. Either one or zero, A or B, true or false, cake or pizza, it doesn't really matter, we'll come back to this again when I'll write a section about primitive variable types.

But Mehdi! You said a Turing machine is a computer, when it really should be seen as a function! That Turing machine you showed us can only increment one specific number!

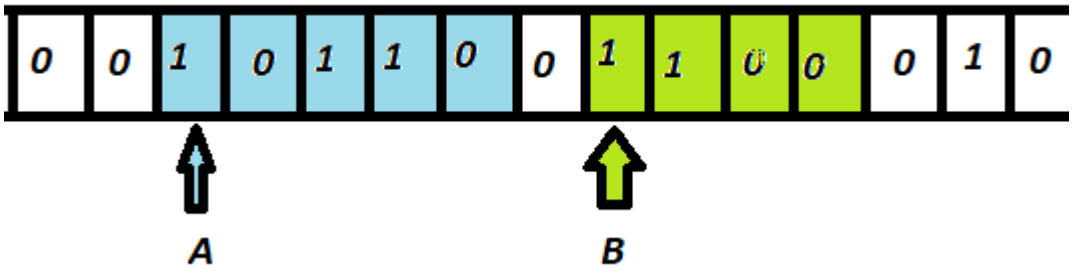
True, and it's a good point. One Turing machine can only do one task, so it is rather one function. However, there is a special Turing machine, called "Universal Turing Machine", that can simulate the behaviour of any other Turing machine. Let's say, for now, and though this is inexact, that it creates a new Turing machine with the desired purpose, then executes it, and then output its execution.

Now that the concept of Turing machines is clear; let's get more in details on memory by understanding "primitive variable types".

Theme 2: Primitive Variable Types

As we've seen it, a computer's memory can be seen as a tape of a certain size that is divided in a discrete number of cells. These cells can contain either 0s or 1s.

A single cell of memory is called a bit. We can take several cells of memory to store data, and that set of cells of memory is what we call a variable. A variable is characterized in memory by what we call a pointer that points to its start.
For instance, on the following schema, we have two variables:



Please note that only blue and green cells are variables. The white ones are just cells that are no longer in use. Also, I am strongly implying that points are not variables. This is not true. Pointers are really complex to understand at first, so I'll probably have to write a section about pointers at some point. For

**don't mind if you don't know what it is, I plan on making a section on abstraction and levels of languages soon*

now, though, you can forget about the fuss and just see the pointers as a little mark in the beginning of the variable to help our Bhayem know where his variables start.

If anything, I want you to remember that a variable is a portion of memory space that is allowed to store one piece of information. This is really important. Also, since you are most likely to be an Epita freshman, you should know that there are no variables in OCAML, and that OCAML actually changes the places of pointers instead of the value of the cells to which the pointers point.

Now that the meaning of a variable is explained, let's look at types of variable.
Classically, in what we call "low level" languages*, there is three big categories of variables:

- booleans
- numbers
- text variables

The most basic variable type is booleans. A boolean is a single cell, which is not much. As it is encoded in only one bit, it can indeed only contain two values: true or false. Booleans are extremely useful, especially with logic. They deserve an important part on their own, but since logic is not technically programming, I'll move on for now.
What you must know on booleans, however, is that they are the result of any conditional expression. Some of you may understand that last sentence, if some of you don't, I'll be extremely glad to make a section about expressions. In any case, I will eventually talk about expressions when I'll deal with programming techniques.

The second type of variables is numbers. This is where it gets a little bit complex, so I'll teach by giving an example. In C#, there are two types of 4 primitive types of number. Those are:

- Signed integers
- Unsigned integers
- Floating numbers
- Decimal

These don't have the same purpose. Signed integers are the most used numbers, they contain a number that is either positive or negative. Their use is usually rather intuitive.
Signed integers are themselves a big family in C#, as it is a family of types rather than a type on itself. That family has 4 members, which are:

- **"sbyte"**, which stands for signed byte. A byte is 8 bits, so 8 cells. Sbyte can contain data ranging from -128 to 127, which is equivalent to -2^7 to (2^7-1).

- **"short"**, which is a short number. A short is encoded in 16 bits, so 16 cells. Similarly to the sbyte, the short goes from -2^15 to (2^15 -1). It is important to know that because an sbyte can be contained in the space of a short, an sbyte can be casted (which pretty much means converted) to a short.

- **"int"**, or **"int32"** which is the most used one. It is encoded on 32 bits, and ranges from (-2^31) to (2^31-1). A short can be cast into an int, and since an sbyte is a short, it can be casted into an int too.

**don't mind if you don't know what it is, I plan on making a section on abstraction and levels of languages soon*

- **“long”,** also called **“int64”** as opposed to the classic **“int32”**. It is encoded on 64 bits, ranges from -2^63 to (2^63 -1). As for previously, the sbyte, the short and the int can all be cast to long.

Now let’s move on to the second family: unsigned integers. They are exactly equivalent to the integers except that they don’t have to use one bit to encode the sign of the value. They are:

- byte (8bits)
- ushort (16bits)
- uint (32bits)
- ulong (64bits)

A byte can be cast into a ushort, which be cast into a uint, which can be cast into a ulong.
You can cast a byte to a short, a ushort to an int, a uint to a long.

The third family is a part of the trickiest and weirdest things that can happen in computer programming. They often lead to weird and obscure behaviour in your program. If you can avoid them, do so. Of course, there exist techniques to not use them, for instance the use of array. As this is less mandatory for your understanding, and since it is rather niche, we’ll come back to it later, and unless I received demands for it, it is not high on my priority queue.
Anyway, without further ado, the third family is floating numbers, with two members:

- float, which is encoded on 32 bits. They can range from $1{,}5 \times 10^{-45}$ to $-3{,}4 \times 10^{38}$ with a precision of 7 numbers.
- Double, which is encoded on 64 bits. They can range from $5{,}0 \times 10^{-324}$ to $-1{,}7 \times 10^{308}$ with a precision of 15 numbers.

The fourth family only has one member. It can be seen as an extension of the floating number family, and I’ll also recommend not using it. In any case, I’ve participated in hackhathons, and worked for companies, and I’ve never seen it outside of books. Its only member is:

- decimal, with is encoded on 128 bits, ranges from $-7{,}9 \times 10^{-28}$ to $7{,}9 \times 10^{28}$ with a 28 numbers precision.

Now talking about casts. What I was referring to was what we call “implicit cast”. You will often try to cast a float into an int, or an int into a float. And when you do so, your C# compiler *(and yes, we will talk about compilers)* will refuse telling you that you can’t “implicitly” cast a float into an int.
You can still force the cast, usually the syntax is something like **typeA foo = (typeA) bar;**

(On a side note, “foo” and “bar” are conventional names for variables, and they both mean “we don’t care about the name”)

When you force a cast, what happens is that it’s going to be a “lossy conversion”. A classic example of “Lossy conversion” is when you cast a float to an int. What will happen is that it’s going to truncate the decimal part. So, if **float foo = 1.2f**, **int bar = (int) foo = 1**.

**don’t mind if you don’t know what it is, I plan on making a section on abstraction and levels of languages soon*

Now, the third big category of primitive variable types is text. It usually only contains 2 types, though it can vary depending on configurations (#CharSet). If you wish, I can talk about why it may vary, but for now let's look at the classic case.

The first type is the **"char",** a char is a single character. It is usually encoded on 8 bits in what we call the UTF-8 standard. *(Again, I won't talk about char sets unless you want me to, but for the record, UTF-8 stands for Universal Character Set)*. Because a char is encoded on 8 cells, it is pretty much a uint. Of course this is a bit more complicated than that, but all you need to know is that a uint can be cast into a char, and that a char can be cast into a uint. You may also want to know that at the level of our tape, Bhayem sees no real difference between a char and a uint.
Be careful, though, in C#, it is not encoded on 8 bits, but on 16, and it follows the UTF-16 standard, which I still won't talk about for now.

The second type is the **"string"**, however, unlike the other primitive types, its legitimacy as a **primitive** type is debatable. For instance, in Java, it is clear that it is not a type but an object, for Java sees a string as an array of chars.
Other languages, like C#, has a different approach. C# still considers the string to be an array of chars, but for some reason, you can treat a string in the exact same way you'd treat a primitive type and C# won't complain. Be careful, though, even in C#, a string is still not a primitive type, and it can't be cast into any primitive type, not even through lossy conversion. If you need to convert a string into another primitive type, you need to parse it using either builtin or made up functions.

**don't mind if you don't know what it is, I plan on making a section on abstraction and levels of languages soon*