You Had Me At Hello [*]

Saad Mneimneh
Computer Science
Hunter College of the
City University of New York (CUNY)
New York, USA
saad@hunter.cuny.edu

ABSTRACT

The hello world program (or some variant thereof) is typically considered one of the simplest programs possible in most programming languages. It is often used to illustrate to beginners the basic use of the language. But does it serve its purpose? I argue that there is nothing simple or basic about the hello world program. On the one hand, it is not an effective ice breaker for syntax; this is especially true with imperative object oriented languages. On the other hand, it does not highlight any of the basic elements of programming, as it involves no thought experiment whatsoever. In this paper, I suggest what I believe is a better alternative to the hello world program. To say the least, I believe my exposition should instigate interesting discussions among computer science educators.

1. HELLO WORLD, AKA INTRODUCTION

The hello world program is often used as an introductory tool to expose the students to their **first** program. This was historically influenced by an example that appeared in the seminal book "The C Programming Language" [2]. That example program prints "hello, world".

One would imagine that this first program should be somewhat simple and universal, or to use a mathematical term, a base case, or at least something that can be extended, and more importantly, a good source for the basic elements of programming. But it is none of these things. In fact, putting everything aside, it is not even that interesting. Here are few points about the hello world program that are backed up by the illustrations in Figure 1.

- depending of the language, it is not necessarily simple
- it looks very different across languages
- it does not illustrate any of the basic elements of programming because it is not associated with any thought experiment
- it is not enough to capture the syntax of the language
- it is not the smallest program you can make and, at any rate, not much smaller than other programs that are much more interesting and insightful

```
print "hello world"
```

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("hello world");
  }
}
```

```
#include <iostream>
using std::cout;
int main() {
  cout<<"hello world\n";
}
```

Figure 1: The hello world program in three languages: Python, Java, and C++.

As observed in Figure 1, the hello world program can be very simple but also very complex. It also varies tremendously in form depending on the language used, making it not an ideal choice for illustrating what, in principal, is a basic programming task: output hello world on the screen. In my opinion, the screen as a device should be outside the realm of programming for beginners. In fact, one often finds it unavoidable to declare that it is to be taken for granted that this is how we output things to the screen (especially in C++). Therefore, the mere act of producing output is not part of the algorithmic repertoire (therefore also logical) [1] at this early stage of a programmer's life. As a result, it should not constitute the starting point.

[*]I make no extensive use of references for this exposition, as there are only few that I found to be necessary. I must reference the title, however, because it is borrowed from a scene in the Jerry Maguire movie of 1996 [1].

[1]Except possibly for using output to debug.

In C++, one must understand operator overloading for an ultimate grasp of the hello world program. In Java, one must even know object oriented programming and classes. Therefore, given that a beginner has to take certain parts, such as output in C++ and class wrappers in Java, for granted, it is better to treat such parts as a way to visualize or obtain the results, and not a as a typical program in the language.

On the other hand, the simplicity of the hello world program in some languages is deceptive, both in terms of syntax and semantics. To a beginner, for instance, the Python statement in Figure 1 suggests that computers can magically understand our intention, as there is a missing important notion of a function (which is not the case with functional programming). For example, why not replace

```
print "hello world"
```

with

```
solve "x-3=5"
```

which apparently does not violate the syntax? One cannot expect a student to know that the second statement "does not make sense" without an appropriate introduction to some basic elements of programming (which obviously hello world does not provide).

Finally, it is clear that the hello world program is not necessarily the smallest possible program, and even if it were, it is not doing anything interesting that the human being cannot do with just a pen and a paper. Also it cannot be extended except by making it print something other than the hello world message (in Java, an extension of the class itself is possible, but it is irrelevant for this discussion). I believe an introduction to programming should ignite a better interest. This is especially true when students don't lack the ability to think algorithmically. A slightly larger (and universal) program can perform a much more interesting task that can be extended in many ways.

2. AN ALTERNATIVE TO HELLO WORLD

In my introductory programming classes, I often deliver my standard speech to advocate that programming is not just about writing code, and that algorithms are at the heart of computer science. I emphasize that programs are simply a representation of these algorithms. To make the students experience this, I then usually engage them with a thought experiment and the task of finding an algorithm to compute the sum of all the natural numbers up to 100. I also ask them not to use what I call the Gaussian trick $1 + 2 + \ldots + n = n(n+1)/2$ [2] in order to focus on the non-trivial algorithmic aspect of the problem.

We all eventually agree that we keep a running sum and we repeatedly add the next number to it (and increment that number by 1). This means that we actually perform the following operation $(\ldots(((1+2)+3)+4)+\ldots+100)$, which is identical to a left-to-right evaluation of the expression $1+2+\ldots+100$ in all programming languages. The problem with this expression is the lack of abstraction, which was not lacking in the mental process that created it. In addition, there is nothing special about the number 100, since we could have done the same for any value, say n. In fact, for a given value n, it becomes clear to the students that our brain keeps two placeholders, say s and i, that evolve as shown in Figure 2 (in pseudocode).

```
sum($n$)
  $s \leftarrow 0$
  $i \leftarrow 1$
  **while** $i \le n$
      $s \leftarrow s + i$
      $i \leftarrow i + 1$
  **return** $s$
```

Figure 2: Mental process for computing $\sum_{i=1}^{n} i$.

In Figure 2, the "length" of the algorithm is not proportional to n, the length of the expression $1+2+\ldots+n$, so it becomes practical, for instance, to write the code required to compute the sum up to a million. Moreover, the resulting program will be flexible enough to compute the sum up to any number by simply changing the value of n (there is an added value here if one needs to eventually talk about functions). These features were possible due to the following four basic elements of programming (all are illustrated by the algorithm):

- naming: we give names to things (variables), e.g. n, i, and s, and assign them values
- initialization: we initialize our brain with some state, e.g. s starts at 0 and i starts at 1
- repetition: we repeat the same process over and over, e.g. adding the next number
- testing: we test whether some condition occurs, e.g. $i \le n$

These four elements, in an informal way, are sufficient to create abstractions and algorithms, and hence programs. Therefore, every programming language has them (I am putting aside the lack of assignment in functional programming), and every programming language uses them in almost identical ways. Figure 3 shows the translation of the algorithm into programs in three languages, all of which are strikingly similar.

At this point, students don't need to understand every single aspect of the program, e.g. the fact that we have just created a function, as long as they have a good grasp of the four elements listed above, and recognize how they occur in the program. This is not a drawback; for instance, in C++ one has to create the main function anyway. I believe it is quite an achievement already if students realize that programming is the act of putting our four elements together in a special

[2] As the story goes, the mathematician Gauss had discovered this equality at the age of ten when asked to perform the exact same task! [3]

way to achieve a desired goal. One could spend some time fiddling around with that idea. Everything else that makes the program run and produce an output is part of what is to be *taken for granted*, as I discussed earlier. At this stage, taking something for granted is acceptable, as long as it does not interfere with the logic of the program, and does not itself constitute the program (which was not the case with hello world). In effect, what Figure 3 shows is a typical, simple, and basic program in almost every language. It is very reasonable to believe that a random person could potentially figure out what the programs in Figure 3 do, but not those in Figure 2 (excluding Python). It would be interesting to actually conduct such an experiment.

Unlike the hello world program, this alternative program is simple and basic, it looks the same across languages, it exposes the syntax of the language in use, and it is small enough while performing an interesting task. In addition, it can be extended in many ways, and this is where the students can find an opportunity to experiment. For instance, one could ask to sum up only the odd number, only the even numbers, or only those that are multiples of a given number. In addition, one could require that the starting point be a given number, or that the last number of the sum must be at most a given fraction of n, or that the sum must stop if a particular value is ever reached. There are plenty of other possibilities. These variations are not arbitrary, as they reinforce the understanding of the four basic elements listed previously: Students have to experiment with different ways of initializing the variables, adding new ones, modifying the repetitive process, and working with different conditions. Upon success, students can even discover new identities similar to $\sum_{i=1}^{n} i = n(n+1)/2$ using the programs they have written as experimental tools.

The thought experiment that lead to the alternative program can also be explored to introduce concepts such as memory, scope, types, and even pointers. The crucial aspect is that we had to **collectively** think about the algorithm before creating the program. Consequently, we identified what we called programming elements before even writing a single line of code. Therefore, students had already been able to associate the concept of naming with memory (their own!), and while everyone in the class may be using the same names, the variables are physically different. This immediately evolves into the concept of memory and scope. Questions will soon arise: how much memory does a variable consume (type), how long does it stay in memory (scope), and where precisely in memory does it reside (pointer)? All these question become highly relevant from the programming perspective. We just converted an introductory program into an open door to explore a *new world*!

3. CONCLUSION

I feel that, as teachers, we are unlikely to succeed to unleash the genuine computer scientist by exposing students to standard introductory examples such as hello world. Many students fail to learn anything beyond “cliché” programs, and lack any further development of their computational and algorithmic skills. Therefore, I believe we should abandon any kind of hello world programs, because we should strive to always ignite the interest of talented computer scientists, especially with creative introductory examples inspired by

```
def sum(n):
  s=0
  i=1
  while i<=n:
    s=s+i
    i=i+1
  return s
```

```
class Sum {
  public static int sum(int n) {
    int s=0;
    int i=1;
    while (i<=n) {
      s=s+i;
      i=i+1;
    }
    return s;
  }
}
```

```
int sum(int n) {
  int s=0;
  int i=1;
  while (i<=n) {
    s=s+i;
    i=i+1
  }
  return s;
}
```

Figure 3: The alternative program in three languages, Python, Java, and C++, not including the parts that will actually produce the result, when n is a 100 for example. These parts, i.e. output statements and main functions and class wrappers, are generally assumed to be *taken for granted* and must be present in almost every program. Without them, observe the striking similarity of the three programs. A retroactive note on the choice of $\leq$ in the algorithm of Figure 2: The use of $<=$ makes it possible for students to predict how to say less than, but not the other way around. It also illustrates that syntax is not necessarily what we think it is (there is no $\leq$ in the language). Finally, it is closer to the intuitive process since n itself must be part of the sum. At any rate, $<= n$ can be replaced by $< n+1$, or by $< n$ if the order of the two update statements are flipped (and the initial conditions adjusted accordingly). This usually makes a nice exercise to reinforce the logic of the program.

algorithmic thought. Moreover, it is unfair for a student to expect that computer programming is a cut and paste experiment, where cliché programs such as the hello world become the main fabric of their learning experience. A student should either learns a good deal, or he/she should be identified as lacking some capabilities from the beginning, which may then be rectified. My alternative to hello world does just that: it grasps the talented students and identifies the deficiencies of others, almost from the first lecture. With that in mind, I wish every computer science educator can help make all kinds of hello world programs become just history and never visit the classrooms again.

4. REFERENCES

[1] James Brooks at al. (Producer) and Cameron Crowe (Screenplay and Director) (1996), *Jerry Maguire*, Motion Picture, TriStar Pictures.

[2] Kernighan Brian and Ritchie Dennis (1988), *The C programming Language*, Second Edition. Prentice Hall. See also https://en.wikipedia.org/wiki/The_C_Programming_Language.

[3] Howard Eves (2002), *In Mathematical Circles: A Selection of Mathematical Stories and Anecdotes*. The Mathematical Association of America.