Programming

Learn C With Babbo

What is programming?

In order to define programming, when must also understand several other terms. Nonetheless, I will first give a definition of programming and then continue to define terms until that definition may be understood.

Programming is the process of writing code, and someone who writes code is called a programmer. **Code** is a series of characters (such as letters and digits) and symbols (like an exclamation point) that conforms to the syntax of some programming language. A **programming language** is a scheme for translating code into a program. A **program** is a series of instructions to be **executed** by a computer. An **instruction** is one of several tasks that a computer is natively able to perform. They are essentially the "atoms" of programs. Some instructions may include doing arithmetic on two numbers, reading data from hardware (e.g., "Is the enter key pressed?"), or copying data.

The Programming Process

The definition alone isn't substantially helpful for understanding programming, so now I will explain how what you type becomes what the computer executes. First, the programmer opens a bare-bones text processing program such as Notepad, vim, Sublime, Text Wrangler, etc. Note that programs such as Microsoft Word or Libre Office are poor choices as programming mediums, because they store formatting data, such as font type and size, in addition to the raw text. This is problematic because the vast majority of programming languages expect code written as raw text, and the extra formatting data is likely to result in errors. Once the programmer has opened the text processor, they begin typing characters and symbols (again, "code") according to the programming language they are using. As an example, look at the following Java code and see if you can figure out what it does:

```
int i = 1;
while (i < 5) {
    if (i < 3) {
        System.out.println(i * i);
    }
    else {
        System.out.println(i + 5);
    }
    i = i + 1;
}
```

Simple programs are usually run inside a **terminal**, which is more or less synonymous with **shell** or **prompt**. A terminal is a textual user interface where a user may navigate their file system or execute programs. See lesson 0c for more information on using terminals. When a program is run, its **output** is printed in the terminal. The output of the previous program is

```
1
4
8
9
```

Even if you do not understand the program, you can still see that it is not entirely cryptic. For example, it uses English words such as "while", "if", and "else", it appears to be performing arithmetic operations such as addition and multiplication, and appears to be roughly organized according to the curly braces "" and "". Indeed, Java, like most programming languages, uses words and syntax to make the program more human-readable and the intentions of the programmer more clear. Such programming languages are called **high-level** programming languages. Predictably, their counterparts are low-level languages, which use short abbreviations that correspond directly to your computer's native instructions. High-level languages, on the other hand, must be translated into **machine code**, a programming language which is understood by your operating system. To do so, code must either be compiled or interpreted.

Compiled Languages

Compiled languages have their code processed by a program called a **compiler**, which translates the source code into machine code. The output of a compiler is a runnable program, unlike the uncompiled source code. Generally, compilers may perform optimizations on your code provided they do not change the result. For example, if in your code you added 3 to a number twice, a compiler might optimize the code so that 6 is added once instead, which would be more efficient. Because of these optimizations, compiled languages are generally very efficient. Examples of compiled programming languages include C, C++, and Java.

Interpreted Languages

Interpreted languages, on the other hand, interpret high-level code in a program called an **interpreter** on the fly. Thus, while the source code for compiled languages is not runnable, the source code for interpretted languages *is* runnable. Interpreted languages usually run in **shells**, where the programmer types code into a terminal and instantly sees the result of running the code, provided the code itself is not complicated or inefficient enough to incur significant processing time. This convenience isn't free, however: because all code that runs on a computer must be translated to machine code, the interpreter program must do such translations while it is running, in contrast to compiled languages which are translated to machine code only during the compilation process. Consequently, interpreted languages are usually *much* less efficient than compiled languages.

Running a Program

When a program is executed, the resources it is expected to use including the program itself is loaded into your computer's **RAM**, which stands for **random access memory**, and your program's instructions are handled by your computer's **processor**. The memory used in RAM is further subdivided into what are called the stack and the heap. The **stack**, roughly speaking, holds all data and memory for portions of your code which require a predictable amount of memory. Since the program itself has a constant size, it lives on the stack. If your program prompts the user to enter two numbers and then outputs their sum, the two numbers the user enters and the sum will be stored on the stack. The **heap**, on the other hand, contains memory for portions of your program that require generally unpredictable resources. For example, let's say you have a program which indefinitely takes input from the user until that user enters "Quit". All the program will do, is output "I've seen that before!" whenever a user enters something that has already been entered, and otherwise outputs "That's new!" when it receives new input. Because there are infinitely many inputs a user could give, the program must keep track of an unpredictable number of inputs because it does not know when the user will terminate the program by typing "Quit". Thus, all the user's previous inputs are stored in the heap instead of the stack.

If you do not have a working knowledge of the stack, the heap, compiled languages, or interpreted languages at this point, that's okay, and in fact I'd be surprised if you did. We will revisit these concepts at a later time when we have more understanding of the programming process.