Programs & Programming

High Level Programming Languages

Remember that the CPU can only understand its own Machine Language, the list of binary patterns for the simple built-in actions it was designed to do. For human convenience, we can write programs in Assembly Language, which uses codes for these actions that humans are better at remembering, but that still requires breaking down tasks into tiny, simple, hardware-specific steps.

For almost all real programming, we instead use High Level Programming languages, which allow us to solve problems with more sophisticated tools than simple arithmetic and moving data around. They provide *data structures* – ways of storing data – and *control structures* – ways to control what the program does.

Variables are a very simple form of data structure. Instead of keeping track of main memory addresses, variables allow programmers to give names to stored data.

The text that makes up a program is sometimes called the *code*, and usually is given as a list of separate lines called *statements*.

In this example, `current`, `toAdd`, and `total` are all variables.

```
public void findTotal() {
     int current = 100;
     int toAdd = 5;
     // add up to get the total
     int total = current + toAdd;
}
```

The first two variables are given values (100 and 5), and then the third gets its value by adding the values of the others together. These actions are grouped together into a *function*[1] called findTotal.

Programs written in High Level languages must be translated down to Machine Language for the program to run. Translation is very difficult, so high level languages must be fairly simple, much simpler than languages like English. The rules of a language – grammar, spelling, etc. – are called its *syntax*, and each programming language has its own syntax which must be followed by the programmer. For instance, in the example above, part of the syntax is that each statement commanding something to happen must end in a semi-colon, and the list of statements that make up a function must be enclosed in curly braces.

Modern programs are written by teams of programmers working together, while other programmers may have to maintain and update them, so it is important that they are understandable when read by humans as well as by computers. *Comments* are notes added to the body of the program for humans to read, but which are ignored by the computer, and not run as part of a program. In the example above, a comment begins with // .

[1] Similar to functions in Excel.

Creating Programs

The first part of creating a program is to get the *requirements*. Although programmers do frequently have their own individual projects to work on in their free time, in industry they are usually hired to create a program to meet someone else's needs. The process of determining what the program needs to do and how it should work takes careful communication and negotiation.

Next, some time should be spent designing the program – thinking through how data and control structures should be used to solve the problems involved. Then the program can be written in some particular programming language; writing programs is often called coding. Most programmers know several high level languages, and for a particular task choose the one that provides the most appropriate tools for the task.

When all the code of a program has been written, we still do not have a program that can be run. It must first be translated into machine language. This translation is done by another program, which can only translate if the program was written following the syntax of the language exactly. If words were misspelled, or punctuation left out, or words were in the wrong order, then the translation program will not know how to translate that code. Instead, it will report a *syntax error*. The programmer must go back and re-write the code to fix the syntax error and then try again.

When all the syntax errors have been fixed, this does not mean the programmer's job is done. A *logic error* (also known as a bug[2]) means that the program runs, but does not meet all the requirements. This may mean that it freezes up, that it gets the

[2] In honor of Rear Admiral Grace Murray Hopper, who worked on the first compilers, who once discovered that a moth in the hardware of an early computer was causing a fault.

wrong results, or that it simply does not do one of the things the requirements said it should. Fixing logic errors may be as simple as fixing a typo (e.g., the program added where it should have subtracted) or may require going back and re-designing, then writing the code again. Any changes made in this process may have added new syntax errors, which have to be eliminated before we can run the program again... and find new logic errors.

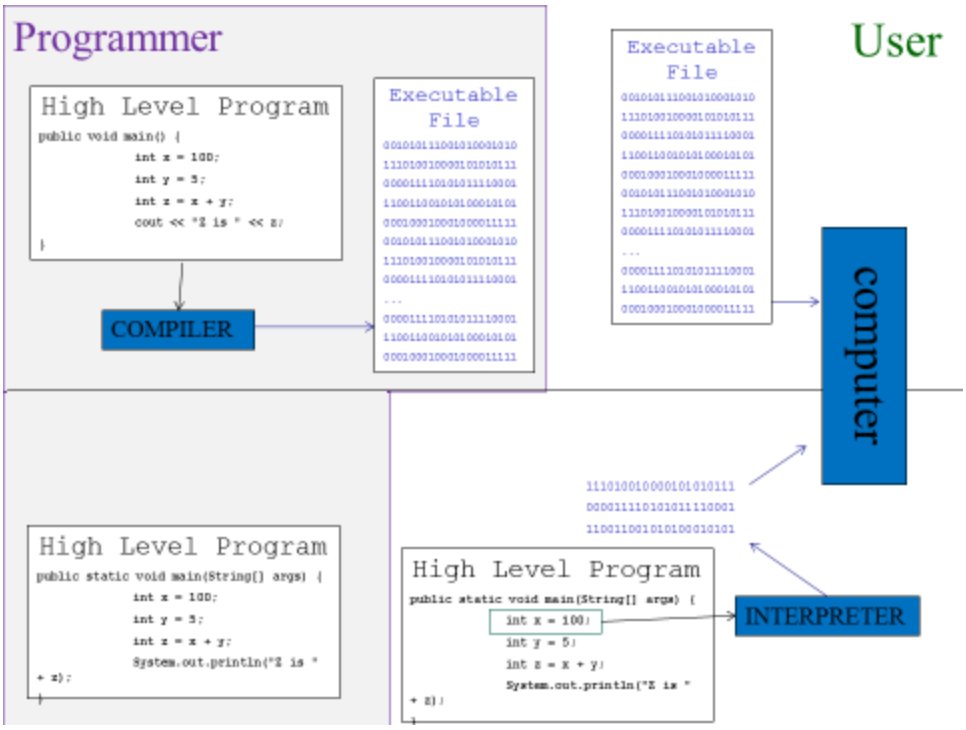
For a large and complex program, the job of testing for and eliminating logic errors takes many passes through this process.

Compiling and Interpreting

There are two standard ways to translate from a high level language program to a machine language program that can run: compiling and interpreting. In general, each programming language is either compiled or interpreted. Any language could be either compiled or interpreted. Creators of a language tend to choose one approach or the other depending on which better matches their goals.

For compiled languages, the original high level language program is given to the compiler program, which translates the whole of the program to machine language and saves this translation in a file called an *executable*. The executable is then given to the user, who can run it over and over again.

For interpreted languages, when the user wants to run the program, they give the original high level language program to the interpreter program, which translates it a little bit at a time *while* running it, every time it runs.



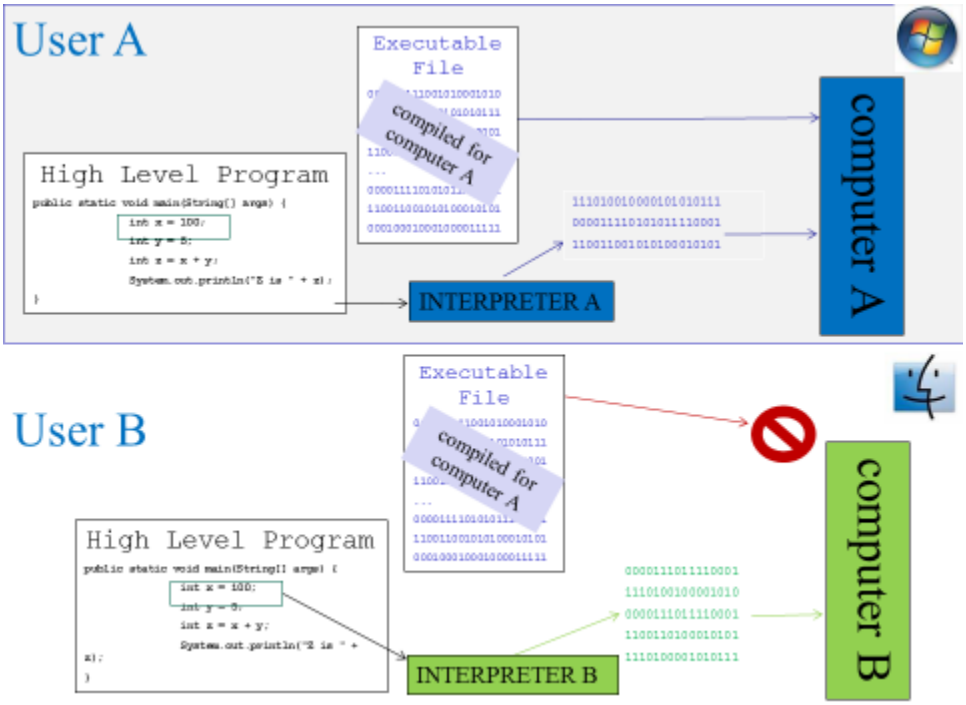
Note that the user would never need to have the compiler for a program written in a compiled language, only the programmer needs it; the user can run the program they're interested in without needing an additional translation program. For an interpreted language, the programmer would still need to have the interpreter, to test their program, but the user needs this extra program as well.

Since all the translation has been done before the program is run, programs in compiled languages run faster than programs in interpreted languages, which have to be translated while they are run every time they run.

So far, all the advantages seem to be on the side of compiled language.

To understand the advantages of interpreted languages, consider that different types of computer have different machine languages. A Mac will not understand instructions for a Windows machine, and vice versa.

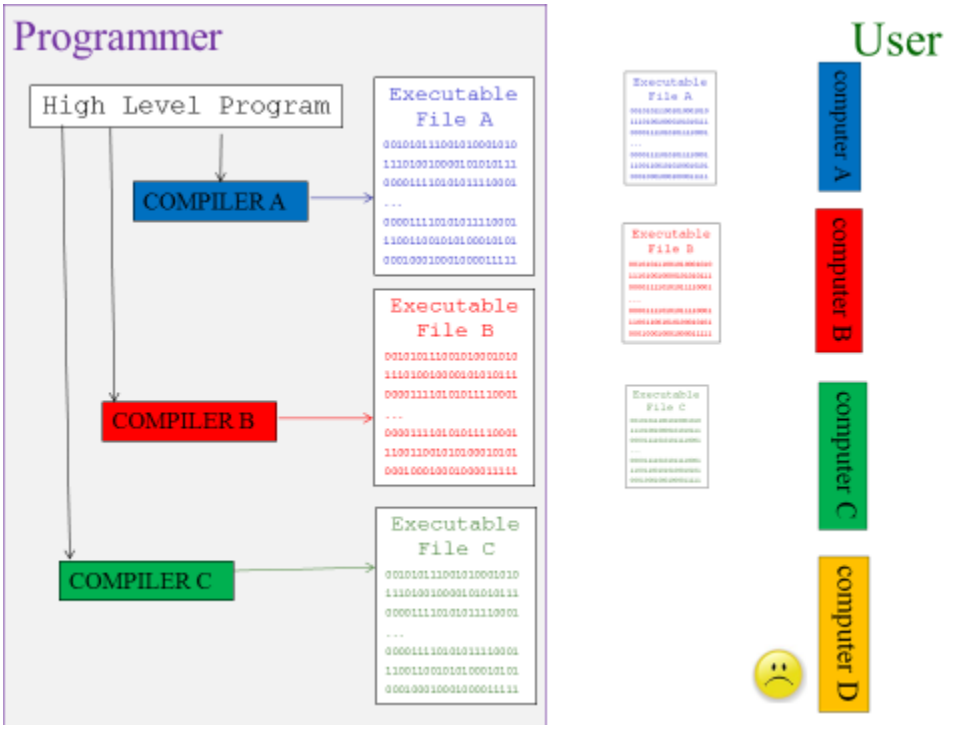
So, when we compile a program, we are compiling it for a specific type of computer. If we compile a program for a Windows machine, that executable will not work on a Mac[3]. However, if the program is in an interpreted language, then as long as the user has the interpreter on their computer, it will interpret into the correct language for the computer they have. This makes interpreted programs very *portable*, making them suitable for use in web pages which will be used by people on many different types of computer.



So, if a program is written in a compiled language, we must rely on the programmer to compile different versions for different types of computer. This could take extra work, if there are details of working with a specific operating system that must be

[3] Executables can sometimes be run on a different type of computer than they were compiled for, using a program called an emulator. The emulator is essentially acting like an interpreter, but instead of translating from a high level language to machine language as the program runs, it translates from one machine language to another.

dealt with, and will at least mean the programmer needs to have each type of system to work on. In the case of interpreted languages, we only need to rely on users getting access to the interpreter, and don't have to worry about the programmer knowing what types of computer will be involved beforehand.



It used to be that programmers sometimes preferred interpreted languages because they could avoid waiting through very long compilation processes to test their programs; however now compilation can usually be done very fast. It also used to be that compiled programs were always much, much faster than interpreted, but now interpreted programs in some languages are almost as fast as compiled[4].

[4] Now many 'interpreted' languages in fact take a hybrid approach, in which the slowest part of the translation work is done through compilation into an intermediate form, which can then be very quickly translated to the machine language of a specific kind of machine.

Installing and Uninstalling

When you get a program from a website or store, typically the first thing you do, before you can run the program itself, is to run another program, the *installer,* which sets up the program for you. An installer is needed for all but very simple programs.

A program's instructions are stored in the executable[5], but modern programs often have many other helper programs in separate files, such as spell checkers. They also often have many data files, such as help documents, templates, and clip art. All these files must be saved on your computer in the directory tree structure the program expects, or it will not work. Setting all of this up can be tedious and error prone.

The operating system also needs to store information about the program, for instance Windows would need to add it to the list of programs in the Start Menu. An operating system stores its data in files called system files, such as the Registry in Windows. It is possible to manually make changes to system files, but if you make a mistake, you might mess up the OS's data badly enough that it will no longer run.

So, to avoid the tedious and the dangerous parts of setting up a program, we have the installer do all of this for us.

When we no longer want a program, these changes have to be undone, which is again tedious and dangerous (even more dangerous, because other programs may now rely on some of the files or some of the data in a system file that this program originally changed. An uninstaller is a program that undoes all the changes made by the installer that won't mess up other

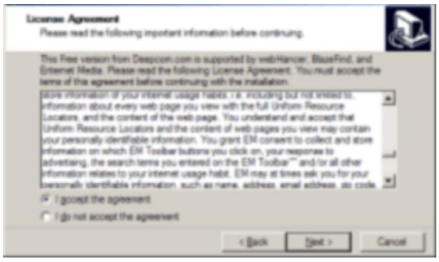
[5] Or a file of intermediate, partly translated code, for most interpreted languages.

programs. Often, an operating system provides an uninstall utility, such as Add-Remove Programs in Windows.

End User License Agreements

If you have paid money for a program at a store or on a website, what you have actually bought is a license— the right to use the software under specific conditions— but you do not own the software itself. This is like the difference between getting a membership for a gym and owning the gym.

The circumstances under which you can use the software are laid out in the EULA, which you generally do not see until you are installing the software. You must indicate that you agree to the EULA before you can finish installing. Once you do so, you are legally bound to whatever is in this document.



Note that EULAs are typically many pages long but are only presented in a tiny text box in a small window. Companies are well aware that almost no one reads these agreements.

Generally, EULAs will set out

- who may use the software — e.g. the purchaser, or their family, or their household
- how many times it can be installed — e.g. on one computer, or on up to three computers, or on all computers in a household

- where it can be used — some software is only licensed for use in certain countries
- what purposes it can be used for — some EULAs specify that you can only use the software for non-commercial purposes, and if you want to use it for making money, you need to buy a different license
- what you can do with the software — very few EULAs give you the right to sell or even give the software to someone else, whether you keep a copy or not.
- the legal responsibility the company has for damage caused by the software — generally none; if the program destroys all your data or messes up your operating system, you can't sue

EULAs might also give the company the right to.

- use any documents you make with the software, or other information they gather about you, in their advertising
- own any documents you make with the software outright
- remotely (that is, over the internet) make changes to the software, to update it
- remotely monitor your use of the software
- control what other software you use on the same computer
- install other software on your computer, possibly without further notice

Most EULAs, although they may give you less than you expect, are not actively malicious, but some are.

Types of License

Some software has non-commercial licenses for personal use, separate commercial licenses, and also site-licenses for

organizations that want to install the same program on dozens or hundreds of computers[6].

An increasingly popular[7] approach to software licenses is the subscription model, in which the user's access to the software is limited in time as well. After a certain period, the user is no longer allowed to use the software, and must pay again. Many companies now offer yearly subscriptions to their software. In many cases, they guarantee that you will always have access to their most recent product, so you don't have to pay again when they come out with a new version, but you do have to pay again when your subscription runs out.

Not all software licenses costs money.

Shareware is software that you can try for free. The EULA in this case, in addition to the usual content, will cover under what circumstances you can continue to use the software for free. In some cases, you can use it for a certain number of days, or a certain number of uses. After this, you agree to either remove the software from your system, or buy a new license for long-term use.

Shareware is essentially a marketing strategy. Software developers without a large advertising budget can get people interested in their programs by letting potential buyers try them for free, in the hope that if they like the software enough, they will pay to keep using it.

Some shareware licenses allow you to keep using the software, but it is a limited version without all the capabilities of the full program, in the hope that eventually you will be tempted to pay for the missing functionality.

---

[6] E.g. for use on a college campus.
[7] With the companies, not with the users, usually.

Nagware is shareware that repeatedly reminds you of the EULA you agreed to, encouraging you to pay for the permanent or full version.

Freeware is software that you can use for free. In this case the EULA does not require you to pay, but may still set out many of the usual limitations on your use of the software. Often there is a freeware license for personal use but a paid license for commercial use.

Open source software is usually freeware, but in addition to the executable machine language program itself, you get access to the high level language code the programmer originally wrote, so that if you are yourself a programmer you can make changes to it, learn from it, or possibly incorporate parts of it into your own programs.

It may seem odd that programmers would give away software (which takes many, many hours of work to create). Many people are suspicious and believe that any free software must contain malware[8]. However, the vast majority of free software is harmless, and some of it is very good. Since a programmer can read the original code for open source software, they could detect if it contained malware or was doing something malicious, so open source software is usually relatively safe.

In general, programmers tend to really enjoy programming, and work on their own programs as a hobby after programming all day at work, and they often like to show off their work, which is one reason for giving software away. Many programmers also have a sufficiently secure life (relatively well-paid, steady work) that they can afford to give work away simply to be nice.

---

[8] malicious software, such as viruses

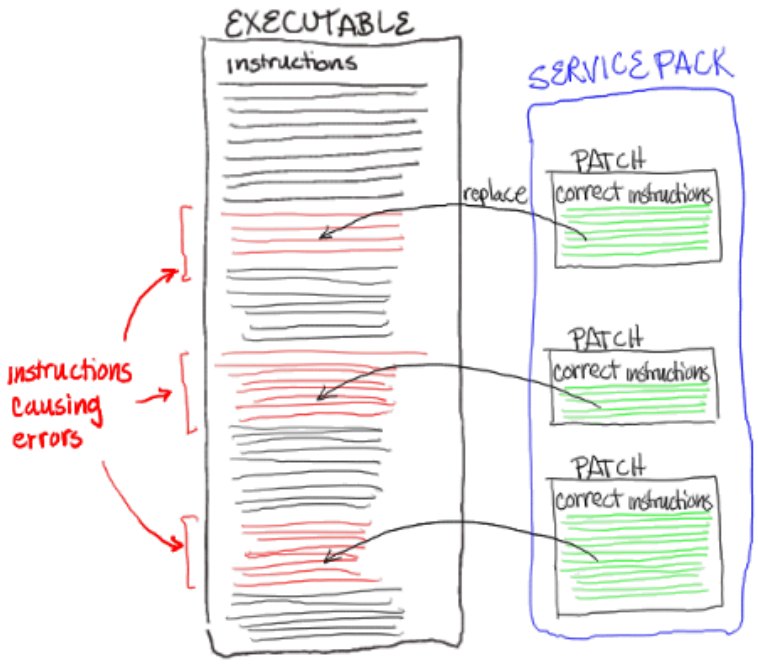
The Open Source movement in software works toward making more open source software available. Some programmers are frustrated or dissatisfied by the commercial software available, either because it is not affordable, not reliable, or because it does things without the user being aware of them (such as sending information remotely back to the company). So they write their own and release it as open source. The hope is that this will both make good software available to more people, and let programmers learn from each other so they can make even better software.

EULAs for open source software tend to give the user a lot of choice of what to do with the software (though there may still be limits, for instance it may require that if you reuse part of their program in yours, then your program will be open source as well). This is a principle called "*free as in speech*." That is, the user is given a lot of freedom in what they can do. Open source software is almost always freeware as well. Freeware is "*free as in beer*" that is, the user is not required to pay for it, but not always free as in speech.

Patches and Updates

After software is released – that is, when it is being sold or given away – it may still not work perfectly[9]. With so many people using the software, problems the programmers never found start to show up, and customers start to complain about these bugs.

[9] Actually, it certainly will not work *perfectly.*



Once they know what the problem is, the programmers find what part of the program is causing the problem, and can fix it. But how can they get the fix to the customers? Instead of having everyone download the whole executable program again, most companies create a *patch*. A user downloads the patch, and the patch replaces just the instructions that caused the problem with the corrected instructions, leaving the rest of the executable alone.

As more problems are found, more patches are released. A customer who comes along later may be faced with dozens of patches to be downloaded as soon as they install the program. To streamline the process (and to de-emphasize the number of problems that needed to be fixed) companies often package many patches up into a single download called a *service pack*.

A patch may fix an existing error, but might also introduce new errors accidentally. Some companies use patches to give their programs new features. Occasionally companies may use a patch to remove a feature (for instance, if they are being sued for using

open source code in their commercial software, breaking an EULA).

When software receives an *update*, that often means that a new version of the executable has replaced the old, instead of the old version just being patched. However some companies use "update" to mean the same thing as "patch."

Beta Testing

Companies try to limit the number of bugs in the programs they release. The programmers test the program as they create it, but many companies also use beta testers.

A *beta tester* tests out the program before it is released, and tries to find bugs the programmers missed. Programmers know what their program is supposed to do, and how people are supposed to use it. Good beta testers are good at thinking up bizarre things to try, to see if they can get the program to go wrong by doing things the programmer never expected.

Just being told there is something wrong with a program is not terribly useful. Details are important, so good beta testers are also good at telling the programmers exactly what they did — including what type of computer they were working on, what other programs were running, and step by step what actions they took—and at telling the programmers exactly what went wrong: what error messages they saw and whether the program froze up or suddenly closed or made the whole computer reboot.

Beta testing is the phase when the programmers think the program has all the features it should, but still has bugs to get rid of. Some companies also have an alpha testing phase. *Alpha testing* means that the program is not finished, there is still more

to add, but the partial program can be compiled and run, enough that it can start being tested. People who do alpha testing are often still called "beta testers."

Many beta testers are just average users who agree to beta (or alpha) test a program, generally in exchange for early access to the program and possibly some trinkets like tee shirts or desk toys. However, increasingly companies hire professional beta testers who have a background in a computing field, which gives them the understanding and vocabulary to give better reports on bugs to guide the programmer to find and fix the causes more quickly.

Some companies deliberately release their software while it is officially still "in beta" (or even alpha!) effectively making their customers into beta testers. The program is often cheaper during this phase than it will be later, and customers must accept (usually explicitly in the EULA) that the program is likely to still have many bugs.

Scripts and Macros

Scripts are short programs that typically complete some task that a human would otherwise have to do, often by using other programs. For instance, a script might search through a list of files for a certain word, copy out all the lines in the files where the word occurs, put them into a spreadsheet, and email the spreadsheet to a certain person.

Macros are short programs that complete some task within another program. For instance a company might have a macro in Word that formats every instance of the company's name in a certain font, size, and color, or someone might create a macro in

Excel to sort, filter, and add a formula for any selected set of cells. A macro may be stored inside a data file.

In many cases, macros and scripts may be created by a program "recording" a user's actions and generating the program automatically, instead of being written out by a human. Sometimes the result is in a high-level language that a human can further modify. Macros and scripts are almost always interpreted.