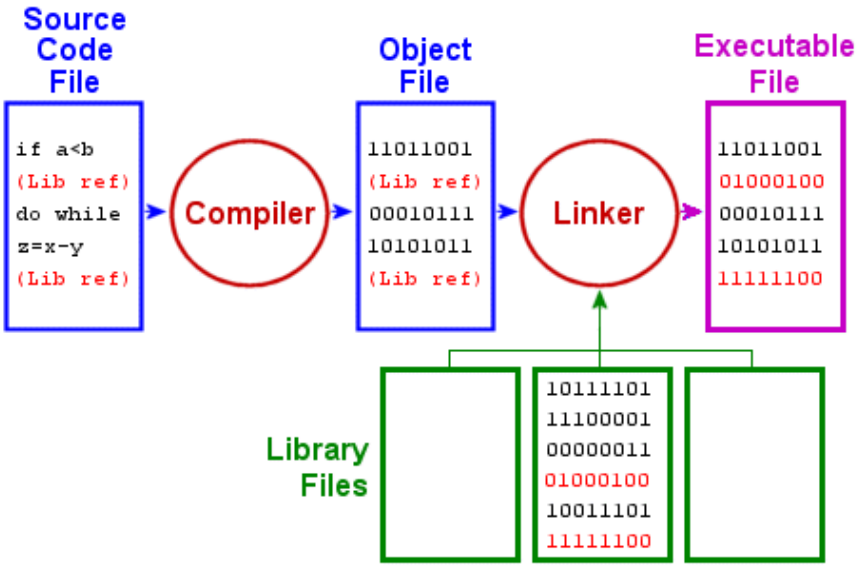
Build Tools

Build tools, sounds like something you might use to help fix someone's house, not software. You may not see it yet (and no I'm not crazy), but stay with me for a second. By looking at a program from a broader angle you will see it has much in common with your home. You will find a foundation (main function), lines of communication (input and output) as well as rooms (classes) that have a specific purpose. Software even has curb appeal in the form of a GUI (graphical user interface).

So let's rein in this analogy a little and bring it back to focus. Houses tend to expand when a new purpose arises, such as when more space is needed, replacing worn out parts, etc. Software also tends to follow this example as new features are added or when code needs to be updated to a new standard are a few examples. In the wide world of software a build tool usually is a programming utility used when building software (expanding/updating your house).

Before we dive into the deep end of build tools maybe we should take a closer look at what a build is exactly. You have built software by now, and have seen the term build somewhere (Visual Studio, books, etc.…). But you may not really even know what that magic button really is doing. Well quite a bit really goes into a build to make a piece of software usable by you or anyone else. A build includes compiling, linking as well as packaging the code to be executable. As that may be a little hard to understand, see figure 1 for a look into the build process.

Figure 1



Notes on the build process

Your source code is the code that you write. Which in turn is passed to the compiler (a program that compiles or translates source code to another format), we will go more into that later. Which leads use to an object file, which contains the compiled code as well as symbols and other things used by the linker (a linker is a program that takes object files, symbols, library files and combines them to a single executable, linking them together). The linker then uses the object file to create the executable file.

We now know a little about the build process having pulled back the curtain on the magic behind the scenes of building software. But we still don't really know the place of build tools in this or any other universe. Well before I explained what a build tool is by definition, but that does not really give you much to go no does it. Alright then stop tap dancing around it and tell me what it is then already. A build tool is a program that automates the creation of executable applications from source code. In essence it takes the everyday mundane tasks, and automates the process. This frees you from having to compile source code into binary, package binary code, run tests, and other things by yourself.

Now we will take a walk on the build side and examine a few of the build tools available to you. First off just as there are quite a few programming languages, there are many build tools as well. Notable ones for Java include Apache Ant, Maven. C++ includes CMake, Boost.Build, MSBuild, C# has MSBuild, NAnt etc. We will look a little more in depth into one for each language just for fun (a little knowledge never hurt anyone).

Build Tools In-depth

MSBuild----------

What if I told you that you already used a build tool many, many times before now. You might give me a strange look and shake your head no. But I can guarantee it because our next contestant on build tools is MSBuild. The Microsoft Build Engine (or MSBuild if you will) has been making life easier for you since 2005 (.Net 2.0). Originally part of .Net, but now shipped as a part of Visual Studio 2013 and newer. This tool is one of the reasons why using Visual Studio to build software is so seamless and user friendly.

MSBuild can really be thought of as its own language that helps to control aspects of the build process. This is accomplished through XML files contained inside of VS projects. Figure 2 contains a small snippet from an MSBuild XML file. You may notice that the XML is quite similar to HTML, not surprising considering both are markup languages (Hypertext markup language HTML, Extensible markup language XML).

Figure 2

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>HelloWorldCS</appname>
    </PropertyGroup>

    <!-- Specify the inputs by type and file name -->
    <ItemGroup>
        <CSFile Include = "consolehwcs1.cs"/>
    </ItemGroup>
```

As you can see from figure 2, this file makes sure that the compiler knows what file to use, embedded resources are included where needed etc… Doing this yourself is quite a time consuming and can be prone to errors. Now you can see what MSBuild does behind the scenes (automating the process) to help take source code to its executable form.

Apache Ant ----------

Now for a little coffee break if you will pardon the pun (get it Java, coffee). But seriously let's talk about another build tool in the form of Apache Ant. This is one of the more popular Java build tools, and the power behind the scenes of NetBeans (although NetBeans also supports notable Java build tools like Maven and Gradle). Ant or Another Neat Tool is one more build automation tool used in developing software, this time for Java. Similar to MSBuild, Ant uses XML to describe the build process and dependencies. Ant was written in Java and allows its users to develop their own "antlibs", which are small XML files containing tasks and types. Ant is similar to MSBuild in another way as it allows you to build C/C++ applications as well. If you look at figure 3, you will see a small sample of an Ant XML file.

Figure 3

```
<project name="MyProject" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>
```

You will see quite a few similarities between figure 3 and figure 2 which is not surprising as both are written in XML. Ant is a flexible and powerful build tool that helps to make your day if you program Java and run NetBeans. Ant is also platform independent thanks to being written in Java (that means you can use it on Linux, Windows). All in all out Ant performs the same basic tasks as MSBuild and other sets of build tools.

CMake----------

Way back in the ancient times of programming existed a very dinosaur like animal called a "make". Ok so it wasn't an animal per se, just a really old dinosaur of a build tool used in UNIX. So in a way you could call it the grandfather of more modern build tools like CMake. Make, and those based on the same system just like CMake, create an alternative to the XML files of other build tools. These files are called makefiles and try to accomplish the same thing in the end. This file tells other programs how to compile and link a program (sound familiar). CMake is quite powerful, allowing you to generate makefiles for many IDE's, as well as UNIX, Windows, Mac, etc. Take a look at figure 4, an example makefile.

Figure 4

```
# The CMake executable.

CMAKE_COMMAND = "E:\Curriculum\Tools course\Student Files\cmake-3.3.0-win32-x86\cmake-3.3.0-win32-x86\bin\cmake.exe"

# The command to remove a file.

RM = "E:\Curriculum\Tools course\Student Files\cmake-3.3.0-win32-x86\cmake-3.3.0-win32-x86\bin\cmake.exe" -E remove -f

# Escaping for special characters.

EQUALS = =

# The top-level source directory on which CMake was run.

CMAKE_SOURCE_DIR = E:\Test

# The top-level build directory on which CMake was run.

CMAKE_BINARY_DIR = E:\Test\Build
```

You will notice quite a bit of difference between the makefile and the previous XML build files. CMake automatically generates these to fit our needs, so there is usually no need to modify it.

Summary

If you build software, then build tools are an important part of the process, even if you never see them. They help to automate and simplify the process of a build, freeing you from tedious and complicated processes. Sometimes they are integrated inside of other tools, sometimes they stand apart. But a build tool always helps to speed the process of software development. If you master them (even if they are a headache), they will reward you in many ways (saving you from other headaches, how ironic).

Compilers

One word that really describes computer programming more than any other, compiling. But what does a compiler actually do when all is said and done. Well source code does not run on its own, it needs to be translated into the target language that you program in. So a compiler translates source code into target code. Most often this process is used to create an executable, so you can actually use your program. But to really understand compilers we need to have a little history lesson (groan…. really, sorry it can't be helped).

The first compiler can be traced back to the A-0 programming language way back in 1952. But the first compiled programming language is considered to be FORTRAN. Followed by other early compiled languages such as COBOL, and LISP. These early compilers were written in assembly language, unlike modern compilers that are written in the code it targets. This process is called self-hosting, one example is when a compiler can compile its own source code.

Enough history, its history after all, let's continue onto the structure of a compiler (yes they have structure). Compilers have a front, middle and back end, each serving its own particular purpose. Of course we will dive in and examine each part (double groan……).

Parts of a compiler

Front end	*Performs type checking; Generates errors and warnings*
Middle end	*Optimizes code; Removes unreachable and useless code*
back end	*Generates assembly code; deals with register allocation in process, etc….*

Of course each part of a compiler has one or more phases which it performs, which can get quite complicated. Don't worry I won't go into them right now, just keep in mind that each end is separated into different phases.

Compilers are targeted for languages and architectures (like Windows, Linux, etc.). Not surprising there are quite a few so compiling a list (no pun intended) is quite a chore, but here are a few examples.

Short compiler list

Language	***Compiler***
C++	*Borland C++, GCC, Clang, Turbo C++, etc…*
C#	*Roslyn, Mono, SharpDevelop, Visual C# etc…*
Java	*Javac, GCJ, Jikes, Power, etc….*

Compiled Vs Interpreted

You may well have heard the term interpreted in programming before but don't know what the difference is compared to a compiled language. Well at the core of both lies the same general purpose, translate a high level language into usable instructions. But interpreters differ quite a bit in execution of this process. Basically where as a compiler converts source code into machine language, an interpreter converts high level instruction (source code) into an intermediate language. It then executes this instructions. At their heart both accomplish the same task by means of translation. Figure 5 compares a few basic compiler and interpreter differences.

Figure 5

Compiler/interpreter differences
-A complier converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
-Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
-List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
-An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.

Summary

Compilers are a fundamental part of using a compiled programming language, even if you don't see the process. Compilers are a part of everyday programming, from command line to IDE. Each compiler simply translates your wants (source code) into a usable form. You tend not to notice a compiler until an issue comes up, like an error. But compilers take a programmers dream and make them reality, helping a programmer to help the rest of the world.