Build Tools

Build Tools

Building a program for a large project is usually managed by a build tool that controls the various steps involved.

These steps may include:

1. Compiling source code to binaries
2. Linking to binaries or libraries
3. Running software tests
4. Creating different build targets
5. Generating documentation

Build Tools

There are a number of different build automation tools available.

Make

Automake

CMake

Apache Ant

Make

The `make` utility is a commonly used build tool that can manage the process of building and rebuilding a project.

Make was originally written in 1976 by Stuart Feldman, who received the 2003 ACM Software System award for his work.

> – For MAKE – there is probably no large software system in the world today that has not been processed by a version or offspring of MAKE.

Make

Make is similar to a declarative programming language in that it describes the conditions for certain command but not the order in which they should be executed.

This can sometimes cause confusion for programmers who are used to imperative programming.

Make

A makefile consists of a set of rules. Each of these rules has a textual dependency line which defines the target of the rule as well as a optional set of dependencies:

```
target: depenency1
        command1
        command2
```

Make

Example – manual compilation:

```
main.cpp
class1.cpp
class2.cpp

$ g++ main.cpp class1.cpp class2.cpp –o main
$ ./main
```

Manually compile all files using gcc from the command line.

Make

Example – simple makefile:

```
all:
	g++ main.cpp class1.cpp class2.cpp –o main
```

```
$ g++ main.cpp class1.cpp class2.cpp –o main
$ ./main
```

Makefile with a single (default) target that compiles all the files.

Make

Example – Makefile with dependencies

```
all: main
main: main.o class1.o class2.o
	g++ main.o class1.o class2.o –o main
main.o: main.cpp
	g++ -c main.cpp
class1.o:
	g++ -c class1.cpp
class2.o:
	g++ -c class2.cpp
```

Makefile that considers each file and dependencies and recompiles only if necessary.

Make

Example – Makefile with clean

```
clean:
	rm –rf *o main

$ make clean
$ make
$ ./main
```

Makefiles often contain a clean target that can be used to completely clean the project of all compiled files and completely rebuild from scratch.

Make

Example – Makefile using Macros

```
# Comment – Selecting g++ as the compiler
CC=g++
all: main
main: main.o class1.o class2.o
	$(CC) main.o class1.o class2.o –o main
main.o: main.cpp
	$(CC) -c main.cpp

...
```

Macros can be used in Makefiles to easily switch between different options and avoid rewriting large sections.

Make

The problem with writing makefiles like this is that they can require a lot of work to set up and maintain them.

Each object file has its own rule and set of dependencies that must be updated.

Another option is to use wildcards and Generic Rules.

Make

Example – Makefile - Macros and Generic rules.

```
# Comment – Selecting g++ as the compiler
CC=g++

SOURCES=$(wildcard *.cpp)
OBJECTS=$(SOURCES:.cpp=.o)

all: main

main: $(OBJECTS)
	$(CC) $(OBJECTS) –o main
.cpp.o:
	$(CC) –c $< -o $@
```

Make

This makefile is an improvement. The generic rules for generating object files from source files allows us to add/remove source files without changing the makefile.

However, it doesn't take into account any dependencies based on header files.

Make

Example – Makefile

```
# Comment – Selecting g++ as the compiler
CC=g++

SOURCES=$(wildcard *.cpp)
HEADERS=$(wildcard *.h)
OBJECTS=$(SOURCES:.cpp=.o)

all: main

main: $(OBJECTS)
	$(CC) $(OBJECTS) –o main
%.o: %.cpp $(HEADERS)
	$(CC) –c $< -o $@
```

Make

This makefile now ensures that header files are included in the dependencies but also included every header file as a dependency for every object file.

Any change to any header file will cause the entire project to be recompiled.

Make

Rather than this all-inclusive approach for dependencies, most C compilers can generate a set of dependencies for you using the '-M' flag.

This can be included into a makefile using

```
-include
```

Make

Example – Makefile

```
# Comment – Selecting g++ as the compiler
CC=g++
SOURCES=$(wildcard *.cpp)
HEADERS=$(wildcard *.h)
OBJECTS=$(SOURCES:.cpp=.o)
DEPS   =$(SOURCES:.cpp=.d)
all: main
main: $(OBJECTS)
	$(CC) $(OBJECTS) –o main
%.o: %.cpp
	$(CC) –c $*.cpp -o $*.o
	$(CC) –M $*.cpp –o $*.d
-include $(DEPS)
```

Make

This makefile generates a .d file for each .cpp file that contains the dependency rule for that .cpp file.

This .d file is then included into the makefile with all of the dependency information for that file. If the .cpp file or any of the dependencies change, the .o file will be recompiled and will generate a new dependency file.

Make

This allows us to write a makefile that will only recompile files that actually need to be recompiled (to ensure fast compile times) but also uses generic rules so we don't need to keep maintaining the makefile.

Make

For more information see:

http://www.gnu.org/software/make/manual/make.html

Make

Make doesn't support cross-platform compilation directly but it can be supported to some degree by platform-specific conditionals in the Makefile.

```
ifeq($(OS), Windows)
      LIBS=-lopengl.dll
else ifeq($(OS), MAC)
      LIBS=-framework OpenGL
else ifeq($(OS), UBUNTU)
      LIBS=-lopengl
endif
```

Automake

Automake is a higher-level language that allows the programmer to avoid manually writing makefiles.

For most simple cases it it enough to give the name of the program, a list of source files and a list of compile/link options. From this Automake can generate a makefile for your project.

Automake

Automake is a part of a set of tools called *The Autotools*. These tools can automate some of the process of writing a Unix build system.

This provides the user with a set of instructions:

```
./configure
make
make install
```

Automake

To generate makefiles with autotools, we write two files:

```
Makefile.am:
bin_PROGRAMS = main
main_SOURCES = main.cpp class1.cpp class2.cpp

configure.ac:
AC_INIT([ammain], [1.0])
AM_INIT_AUTOMAKE
AC_PROG_CXX
AC_PROG_RANLIB
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Automake

To generate makefiles with autotools, we write two files:

```
$ aclocal
$ autoheader
$ autoconf
$ automake --add-missing
```

This process generates the configure script and makefile for the project.

Automake

The project can then be built using:

```
$ ./configure
$ make
```

To build the project on a different system, simply re-run configure and then make the project.

Automake

To generate and link to libraries:

```
Makefile.am:
noinst_LIBRARIES = libclass1.a libclass2.a

libclass1_SOURCES = class1.cpp
libclass2_SOURCES = class2.cpp

bin_PROGRAMS = main
main_SOURCES = main.cpp
main_LDADD = libclass1.a libclass2.a
```

Automake

For more information on Automake:

http://www.gnu.org/software/automake/

CMake

CMake is the cross-platform, open-source build system designed to control the software compilation using platform and compiler independent configuration files.

This tool automatically generates build scripts for different operating systems - Visual Studio projects for Windows and makefiles for Unix/ Linux.

CMake

CMake generates build scripts from files named:

`CMakeLists.txt`

These files must be put in each subdirectory of the project as required.

CMake

Example – Simple CMake

```
cmake_minimum_required(VERSION 2.6)

project(Main)

add_executable(Main main.cpp class1.cpp class2.cpp)

$ cmake .
-- The C compiler identification is Clang 5.1.0
-- The CXX compiler identification is Clang 5.1.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
...
```

CMake

Example – Simple CMake

```
$ make
Scanning dependencies of target Main
[ 33%] Building CXX object CMakeFiles/Main.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/Main.dir/class1.cpp.o
[100%] Building CXX object CMakeFiles/Main.dir/class2.cpp.o
Linking CXX executable Main
[100%] Built target Main
```

CMake

Rather than giving CMake explicit commands for how to compile your program.

Instead you give instructions of what you want it to build.

CMake

Example – CMake with separate libraries

```
cmake_minimum_required(VERSION 2.6)

project(Main)

add_library(class1 class1.cpp)
add_library(class2 class2.cpp)

add_executable(Main main.cpp)

target_link_libraries(Main class1 class2)
```

CMake

Example – CMake with separate directories

```
./main.cpp
./CMakeLists.txt

class1/class1.h
class1/class1.cpp
class1/CMakeLists.txt

class2/class2.h
class2/class2.cpp
class2/CMakeLists.txt
```

CMake

Example – CMake with separate directories

```
class1/CMakeLists.txt
add_library(class1 class1.cpp)

class2/CMakeLists.txt
add_library(class2 class2.cpp)

./CMakeLists.txt
project(Main)
include_directories("${Main_SOURCE_DIR}/class1")
include_directories("${Main_SOURCE_DIR}/class2")
add_subdirectory(class1)
add_subdirectory(class2)
add_executable(Main main.cpp)
target_link_libraries(Main class1 class2)
```

CMake

CMake will generate makefiles for each of the different directories along with the necessary code to:

1. Build each subdirectory
2. Include files from the subdirectories
3. Link binaries together

CMake

CMake has support for finding packages and including/linking to them. This cross-platform support covers packages that have different names and methods of including/linking on different operating systems.

CMake

Example – CMake with packages

```
project(Main)

add_executable(Main main.cpp)

find_package(OpenGL)
include_directories(${OpenGL_INCLUDE_DIRS})
target_link_libraries(Main ${OpenGL_LIBRARIES)
```

CMake

For more information on CMake:

http://www.cmake.org/cmake/help/cmake_tutorial.html

Apache Ant

Apache Ant is a Java library for building large projects. It is used mainly for building Java Applications but does also have support for other languages such as C or C++.

The build of a project is defined by an XML file called build.xml

Apache Ant

Structure of build.xml:

```
<project>
  <property name="src.dir"   value="src"/>
  <property name="build.dir" value="$build"/>
  <property name="jar.dir"   value="${build.dir}/jar"/>

  <target name="compile">
    <mkdir dir="${classes.dir}"/>
    <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
  </target>
  <target name="jar" depends="compile">
    <mkdir dir="${jar.dir}"/>
    <jar destfile ...   />
  </target>
</project>
```

Advanced Build Automation

As projects become larger and more complex, more advanced build tools have been required to manage working with such large projects.

Automation of the build process extends beyond simply managing the compiling and linking of a program.

Advanced Build Automation

Distributed build automation is still largely a compilation/linking feature that farms out the compilation of different parts of the program to multiple locations or cores.

Advanced Build Automation

An advanced build tool will not only build the project but can perform a series of tasks and tests to catch problems early on.

Scheduled builds – Many projects have scheduled, (eg nightly) builds where the current version of the project is compiled each night and run through code tests to pick up any errors. Prevents errors in the code from propagating.

Advanced Build Automation

Triggered builds – Some projects may be triggered to be rebuilt each time any developer commits new code.

Can immediately catch problems before they become part of the project.

Version Control

This type of build automation usually assumes the use of a version control system. Version control allows multiple developers to work on the same project at the same time.

CVS - Concurrent Versions System

SVN – Subversion

Git

Version Control

Trunk - A set of source code, resources etc making up a project. Basically the project.

Branch - A branch splits from the trunk at some point in time, can be edited and updated separately. Used to try out ideas.

Merging – A successful branch can be merged back into the trunk.

Version Control

Check out – Creates a local copy of the project (specific to a trunk or branch) from the repository.

Commit – Send your changes to the repository, creates a 'new version' of the project.

Version Control

Version control software such as SVN is built using the client-server model where each client connects to a server that 'owns' the software repository.

The server is responsible for accepting commits, it may 'lock' certain parts of the project etc.

Version Control

Git is an example of a distributed version control system. Each user maintains a local copy which counts as a repository itself. Every user 'owns' the project as much as any other user or server.

Changes can be pushed/pulled from any other copy of the project repository.