Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

Programming in Modern C++

Tutorial T03: How to build a C/C++ program?: Part 3: make Utility

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Tutorial Recap

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- Understood the overall build process for a C/C++ project with specific reference to the build pipeline of GCC
- Understood the differences and relationships between source and header files
- Understood how CPP can be harnessed to manage code during build
- Understood the management of C/C++ dialects and C/C++ Standard Libraries

Tutorial Objective

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- Building a software project is a laborious, error-prone, and time consuming process. So it calls for automation by scripting
- `make`, primarily from GNU, is the most popular free and open source dependency-tracking builder tool that all software developers need to know

Tutorial Outline

1. Tutorial Recap
2. make Utility
   - Example Build
   - Why make?
3. Anatomy of a makefile
   - Simple makefile
   - Simple and Recursive Variables
   - Dependency
   - Source Organization
4. make Command
   - Options and Features
   - Capabilities and Derivatives
5. Tutorial Summary

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

make Utility

Source: Accessed 15-Sep-21
GNU make Manual
GNU Make
A Simple Makefile Tutorial

make Utility: Example Build

Tutorial T03

Partha Pratim Das

Tutorial Recap

Objectives & Outline

make Utility

Example Build

Why make?

Anatomy of a makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and Derivatives

Tutorial Summary

- Consider a tiny project comprising three files – **main.c**, **hello.c**, and **hello.h** in a folder

main.c

```
#include "hello.h"

int main() {
    // call a function in
    // another file
    myHello();
    return 0;
}
```

hello.c

```
#include <stdio.h>
#include "hello.h"

void myHello(void) {
    printf("Hello World!\n");
    return;
}
```

hello.h

```
// example include file
#ifndef __HEADER_H
#define __HEADER_H

void myHello(void);

#endif // __HEADER_H
```


- We build this by executing the command in the current folder (`-I.`):

```
gcc -o hello hello.c main.c -I. // Generates hello.o & main.o and removes at the end
```

which actually expands to:

```
gcc -c hello.c -I. // Compile and Generate hello.o
gcc -c main.c -I.  // Compile and Generate main.o
gcc -o hello hello.o main.o -I. // Link and Generate hello
rm -f hello.o main.o // Is it really necessary? . hello.o & main.o may be retained
```

Why we need `make` Utility?

Tutorial T03

Partha Pratim Das

Tutorial Recap

Objectives & Outline

`make` Utility

Example Build

Why `make`?

Anatomy of a makefile

Simple `makefile`

Variables

Dependency

Source Organization

`make` Command

Options and Features

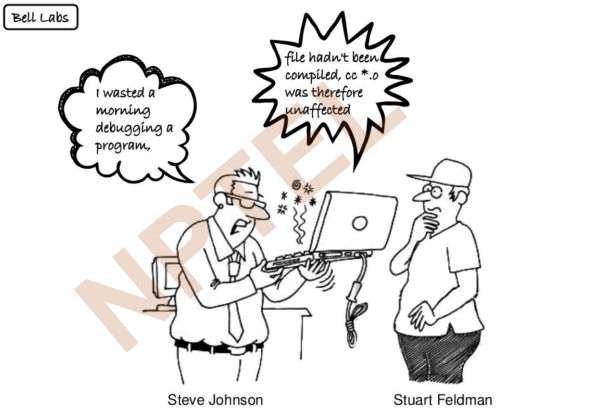
Capabilities and Derivatives

Tutorial Summary

- This manual process of build would be difficult in any practical software project due to:
  - **[Volume]** Projects have *hundreds of folders, source and header files*. They need *hundreds of commands*. It is *time-taking* to type the commands and is *error-prone*
  - **[Workload]** Build needs to be *repeated several times a day* with code changes in some file/s
  - **[Depenency]** Often with the change in one file, all translation units do not need to be re-compiled (assuming that we do not remove `.o` files). For example:
    - ▷ If we change only `hello.c`, we do not need to execute
      ```
      gcc -c main.c -I.  // main.o is already correct
      ```
    - ▷ If we change only `main.c`, we do not need to execute
      ```
      gcc -c hello.c -I.  // hello.o is already correct
      ```
    - ▷ However, if we change `hello.h`, we need to execute all

    There are *dependencies* in build that can be exploited to optimize the build effort
  - **[Diversity]** Finally, we may need to use different build tools for different files, different build flags, different folder structure etc.
- This calls for *automation by scripting*. **GNU `Make` is a tool which controls the generation of executables and other non-source files of a program from the program's source files**

Why we need make Utility?: What happened in Bell Labs



Stuart Feldman created make in April 1976 at Bell Labs

Makefile Martial Arts - Chapter 1. The morning of creation

Anatomy of a makefile

Tutorial T03

Partha Pratim Das

Tutorial Recap

Objectives & Outline

make Utility

Example Build

Why make?

Anatomy of a makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and Derivatives

Tutorial Summary


```
target [target ...]: [component ...]
Tab ↹ [command 1]
            .
            .
            .
Tab ↹ [command n]
```

Make (software), Wikipedia

Anatomy of a makefile

makefile: Anatomy

Tutorial T03

Partha Pratim Das

Tutorial Recap

Objectives & Outline

make Utility

Example Build

Why make?

Anatomy of a makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and Derivatives

Tutorial Summary

- A simple make file would be like (makefile_1.txt):

```
hello: hello.c main.c
    gcc -o hello hello.c main.c -I.
```

  Write these lines in a text file named makefile or Makefile and run make command and it will execute the command:

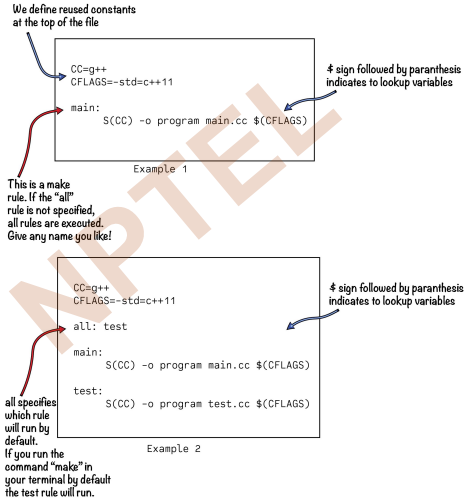
```
$ make
gcc -o hello hello.c main.c -I.
```

- Make file comprises a number of Rules. Every rule has a target to build, a colon separator (:), zero or more files on which the target depends on and the commands to build on the next line

```
hello: hello.c main.c # Rule 1
    gcc -o hello hello.c main.c -I.
```

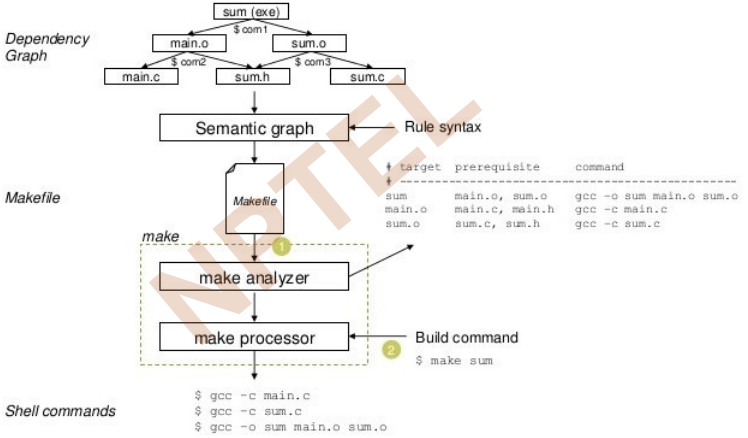
- Note:
  - **There must be a tab at the beginning of any command**. Spaces will not work!
  - If any of the file in the dependency (hello.c or main.c) change since the last time make was done (target hello was built), the rule will fire and the command (gcc) will execute. This is decided by the last update timestamp of the files
  - Hash (#) starts a comment that continues till the end of the line

makefile: Anatomy



How to Write a Makefile with Ease

makefile: Architecture



makefile: Simple and Recursive Variables

- We can make the make file smarter (makefile_2.txt) using variables:

```
# CC is a simple variable, gcc is its value
CC := gcc
# CFLAGS is a recursive variable, -I. is its value
CFLAGS = -I.

hello: hello.c main.c        # Rule 1
    $(CC) -o hello hello.c main.c $CFLAGS
# $(CC) is value gcc of CC, $CFLAGS is value -I. of CFLAGS, Variables can be expanded by $(.) or ${.}
```

- If there are several commands, to change gcc to g++, we just need to change one line CC=g++.
- There are two types of variables in make (Chapter 3. Variables and Macros, O'Reilly)
  - Simply expanded variables (defined by := operator) and *evaluated as soon as encountered*
  - Recursively expanded variables (by =) and *lazily evaluated, may be defined after use*

Simply Expanded	Recurively Expanded
MAKE_DEPEND := $(CC) -M	MAKE_DEPEND = $(CC) -M
...	...
# Some time later	# Some time later
CC = gcc	CC = gcc
$(MAKE_DEPEND)$ expands to:	
<space>-M	gcc -M

makefile: Dependency

- We are still missing the dependency on the include (header) files. If hello.h changes, the above Rule 1 will not detect the need for a re-build. So we improve further (makefile_3.txt):

```
CC=gcc
CFLAGS=-I.
#Set of header files on which .c depends
DEPS = hello.h

# Rule 1: Applies to all files ending in the .o suffix
# The .o file depends upon the .c version of the file and the .h files in the DEPS macro
# To generate the .o file, make needs to compile the .c file using the CC macro
# The -c flag says to generate the object file
# The -o $@ says to put the output of the compilation in the file named on the LHS of :
# The $< is the first item in the dependencies list
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

hello: hello.o main.o        # Rule 2: Link .o files
    $(CC) -o hello hello.o main.o -I.
```

makefile: Dependency

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- We can further simplify on the object files (makefile_4.txt):

```
CC=gcc
CFLAGS=-I.
DEPS = hello.h
#Set of object files on which executable depends
OBJ = hello.o main.o

# Rule 1: Applies to all files ending in the .o suffix
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

# Rule 2: Linking step, applies to the executable depending on the file in OBJ macro
# The -o $@ says to put the output of the linking in the file named on the LHS of :
# The $^ is the files named on the RHS of :
hello: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)
```

makefile: Code Organization

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- Finally, let us introduce a source organization that is typical of a large project where under a project `Home` folder, we have the following folders:
  - `Home`: The make file and the following folders:
    - `bin`: The executable of the project. For example `hello` / `hello.exe`
    - `inc`: The include / header (`.h`) files of the project. For example `hello.h`
    - `lib`: The local library files (`.a`) of the project
    - `obj`: The object files (`.o`) of the project. For example `hello.o` & `main.o`
    - `src`: The source files (`.c`/`.cpp`) of the project. For example `hello.c` & `main.c`

makefile: Code Tree

Tutorial T03

Partha Pratim Das

Tutorial Recap

Objectives & Outline

make Utility

Example Build

Why make?

Anatomy of a makefile

Simple makefile

Variables

Dependency

Source Organization

make Command

Options and Features

Capabilities and Derivatives

Tutorial Summary


```
Home // Project Home
  |
  ---- bin // Application binary
  |     |
  |     ---- hello.exe
  |
  ---- inc // Headers files to be included in application
  |     |
  |     ---- hello.h
  |
  ---- lib // Library files to be linked to application. Check Tutorial Static and Dynamic Library
  |
  ---- obj // Object files
  |     |
  |     ---- hello.o
  |     ---- main.o
  |
  ---- src // Source files
  |
  |     ---- hello.c
  |     ---- main.c
  |
  ---- makefile // Makefile
```

makefile: Code Organization

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- To handle this hierarchy, we modify as (makefile_5.txt):

```
CC = gcc
# Folders
BDIR = bin
IDIR = inc
LDIR = lib
ODIR = obj
SDIR = src
# Flags
CFLAGS = -I$(IDIR)
# Macros
_DEPS = hello.h # Add header files here
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_SRC = hello.c main.c # Add source files here
SRC = $(patsubst %,$(SDIR)/%,$(_SRC))
_OBJ = hello.o main.o # Add source files here
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))
# Rule 1: Object files
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS); $(CC) -c -o $@ $< $(CFLAGS) -I.
#Rule 2: Binary File  Set binary file here
$(BDIR)/hello: $(OBJ); $(CC) -o $@ $^ $(CFLAGS)
# Rule 3: Remove generated files. .PHONY rule keeps make from doing something with a file named clean
.PHONY: clean
clean: ; del $(ODIR)\*.o $(BDIR)\*.exe
#        rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

make Command

make Command: Options and Features

- The format of `make` command is:

```
make [ -f makefile ] [ options ] ... [ targets ] ...
```

`make` executes commands in the `makefile` to update one or more target `names`

  - With no `-f`, `make` looks for `makefile`, and `Makefile`. To use other files do:

```
$ make -f makefile_1.txt                // Using makefile_1.txt
gcc -o hello hello.c main.c -I.
```

  - `make` updates a target if its prerequisite files are dated. Starting empty obj & bin folders:

```
$ make -f makefile_5.txt obj/hello.o    // Build hello.o, place in obj
gcc -c -o obj/hello.o src/hello.c -Iinc -I.

$ make -f makefile_5.txt obj/main.o     // Build main.o, place in obj
gcc -c -o obj/main.o src/main.c -Iinc -I.

$ make -f makefile_5.txt bin/hello      // Build hello.exe linking .o files and place in bin
gcc -o bin/hello obj/hello.o obj/main.o -Iinc

$ make -f makefile_5.txt  clean         // Remove non-text files generated - obj/*.o & bin/*.exe
del obj\*.o bin\*.exe

$ make -f makefile_5.txt                // By default targets bin/hello and builds all
gcc -c -o obj/hello.o src/hello.c -Iinc -I.
gcc -c -o obj/main.o src/main.c -Iinc -I.
gcc -o bin/hello obj/hello.o obj/main.o -Iinc
```

make Command: Options and Features

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- More make options / features:
  - To change to directory dir before reading the makefiles, use -C dir
  - To print debugging information in addition to normal processing, use -d
  - To specify a directory dir to search for included makefiles, use -I dir
  - To print the version of the make program, use -v. We are using

```
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

This program built for i386-pc-mingw32
```

  - make can be recursive - one make file may include a command to make another
  - Multiline: The backslash ("\") character gives us the ability to use multiple lines when the commands are too long

```
some_file:
    echo This line is too long, so \
        it is broken up into multiple lines
```

  - Comments: Lines starting with # are used for comments
  - Macros: Besides simple and recursive variables, make also supports macros

make Utility: Capabilities and Derivatives

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- make was created by Stuart Feldman in April 1976 at Bell Labs and included in Unix since PWB/UNIX 1.0. He received the 2003 ACM Software System Award for make
- make is one of the most popular build utilities having the following major **Capabilities**:
  - make enables the end user to *build and install a package* without knowing the details of how that is done (it is in the makefile supplied by the user)
  - make *figures out automatically which files it needs to update*, based on which source files have changed and *automatically determines the proper order for updating files*
  - make is *not limited to any particular language* - C, C++, Java, and so on.
  - make is *not limited to building a package* - can control installation/uninstallation etc.
- make has several **Derivative** and is available on all OS platforms:
  - GNU Make (all types of Unix): Used to build many software systems, including:
    - GCC, the Linux kernel, Apache OpenOffice, LibreOffice, and Mozilla Firefox
  - Make for Windows, GnuWin32 (We are using here)
  - Microsoft nmake, a command-line tool, part of Visual Studio
  - Kati is Google's replacement of GNU Make, used in Android OS builds. It translates the makefile into Ninja (used for Chrome) for faster incremental builds

All links Accessed on 21-Sep-21

Tutorial Summary

Tutorial T03
Partha Pratim Das
Tutorial Recap
Objectives & Outline
make Utility
Example Build
Why make?
Anatomy of a makefile
Simple makefile
Variables
Dependency
Source Organization
make Command
Options and Features
Capabilities and Derivatives
Tutorial Summary

- Learnt `make`, the most popular free and open source dependency-tracking builder tool, with its anatomy, architecture and options through a series of examples