Session: Makefiles
Topic: Program Translation Makefiles

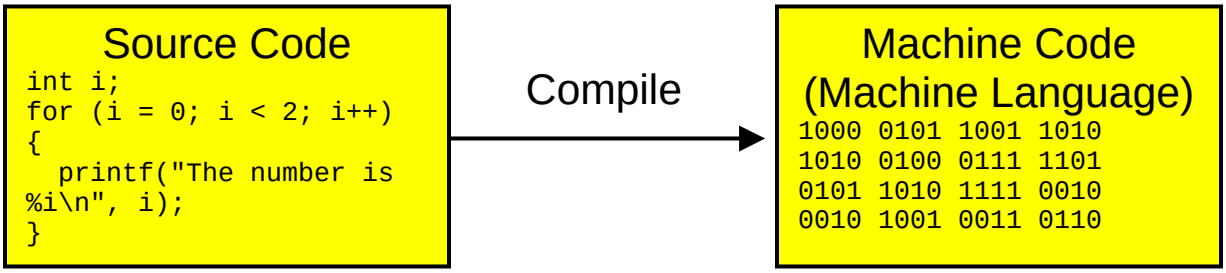
Daniel Chang

Program Translation

High-level programming languages (e.g. C, C++, C#) start as "Source Code"

```
int i;
for (i = 0; i < 2; i++)
{
  printf("The number is %i\n", i);
}
```

- "Source Code" is language for programmers, but machines only read "Machine Language" (details later)
- So Source Code must be "compiled" (translated) into Machine Language before being readable by the computer
- Shell scripts are actually "interpreted" on the fly, rather than precompiled



The Source Code is "compiled" in order to "make" the Machine Language file

- The Machine Language file is "made"
- If the Machine Language file is a complete program that can be run, it is called a "executable" file
- If the Machine Language file is not complete and is to be combined with other files it is called an "object" file

Makefiles

- The "make" command is a utility program that reads instructions from a text file called a "Makefile"
- The Makefile specifies instructions on how to build a high-level programming project with multiple source code files
- Often multiple source code files reference each other, so must be compiled in a certain order
- Makefiles must be named "makefile" or "Makefile" and must exist in the directory where the project source code files exist (generally).

SourceFile2:

(Uses "Print" function)

- SourceFile2 "depends" on objects defined by SourceFile1

SourceFile1:

(Defines "Print" function)

- So SourceFile1 must be compiled first

Makefile Components

Makefiles contain

- Comments
- Rules, consisting of
    - Dependency Lines, consisting of
        - Target
        - Dependency List
    - Commands (Shell Lines)
- Macros
- Inference Rules
- (Other stuff)

The "make" utility is ***obnoxiously picky*** (i.e. buggy) about the format of makefile contents

Comments

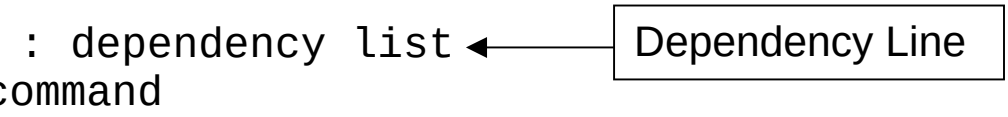
- Comments are indicated by a "#"
- All text from the "#" to the end of the line is ignored by the "make" utility
- Comments can start anywhere

Example

```
#
# this is a comment
projecte.exe : main.obj io.obj  # more comment
```

Rules

- Rules tell the "make" utility when and how to compile a source code file
- A rule has a Dependency Line and a Command Line



- Rules must be separated from other rules by a blank line

Dependency Line

- Dependency Lines indicate when a particular target file must be made

Target

- The "target" is usually the name of a machine language file that must be made as part of the project
- The target must be separated from the dependency list by a colon (:)
- The target name must start in the first column of the line (no leading white space)

Dependency List

- The "dependency list" is a list of files that must exist in order to make the target
- The "make" utility checks the dates of the files in the dependency list and if they are newer than the target will re-make the target
- The files in the dependency list must be separated by spaces
- The files in the dependency list must all be on the same line (no enters/returns)

Example

```
main.obj : part1.c part2.c main.h
      [some command goes here]

Project.exe : main.obj io.obj
      [some command goes here]
```

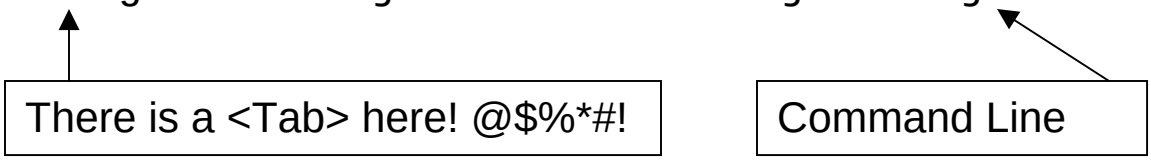
- If more than one target depends on the same dependency list, they can be placed before the (:) separated by spaces

Command Line

- Command lines indicate how to make the target
- Typically the command to the "compiler" needed to create the target from the dependency list files
- May also specify some file maintenance like deleting old files
- Command lines must be ***indented with a Tab***
- A rule may have more than one Command line, each on a separate line

Example

```
Project.exe : main.obj io.obj
    gcc -o Project.exe main.obj io.obj
```




```
cleanbuild : prog.c lib.o
    gcc -o cleanbuild prog.c lib.o
    rm lib.o
```

Macros

- A shorthand alias used in a makefile
- Essentially a string (traditionally all capitals) is associated with another (usually larger) string
- Macros are created using the format

```
[MACRONAME] = [string to substitute]
```

- Inside the makefile the macro string is expanded using the format

```
$(string)
```

Example

```
PROJ = myprog.exe
BIN = /usr/bin
CPP = $(BIN)/g++

$(PROJ): proj.c proj.h
    $(CPP) -o $(PROJ) proj.c proj.h -lm
```

Inference Rules

- A wildcard (matching) notation for generalizing the make process
- "%" is used to indicate a wildcard, and will match other "%"

Example

```
%.obj: %.c
    $(CPP) -o $(PROJ) $(.SOURCE)
```

- This indicates that all ".obj" files have dependencies with all ".c" files with the same name (where the "%" are the same)
- ".SOURCE" is a macro provided by the "make" utility which refers to the dependency inferred by the current inference rule (%)
- ".TARGET" would refer to the current rule target