# Tool Chain

*CIS*2750*

*Advanced Programming Concepts*

# What is a “tool chain”?

- Computers consume *binary* instructions and data—but humans don’t want to write those
  - Boring, error prone, inefficient, impractical, not debuggable, not maintainable
- We invented **tools** (complex SW programs) to take understandable **high level** symbolic inputs and crunch them into binary form
  - Called **“chain”** because used in certain order

# Purpose of Tool Chain lecture

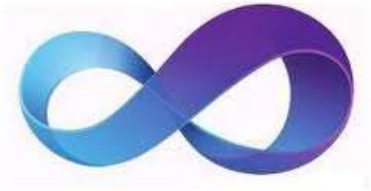
- Show the **Big Picture** of the tools C programmers use constantly
  - Expose them all to view
    - Some are "hiding" inside others!
  - How they "chain" together in sequence
  - How to control them


- Bring them up from the level of ritualized mumbo-jumbo to one of knowledge

# The Usual Suspects


C Preprocessor  C Compiler  Linker  Loader  Make

- **Make** is to manage the others
- *Those look like Linux tools…*

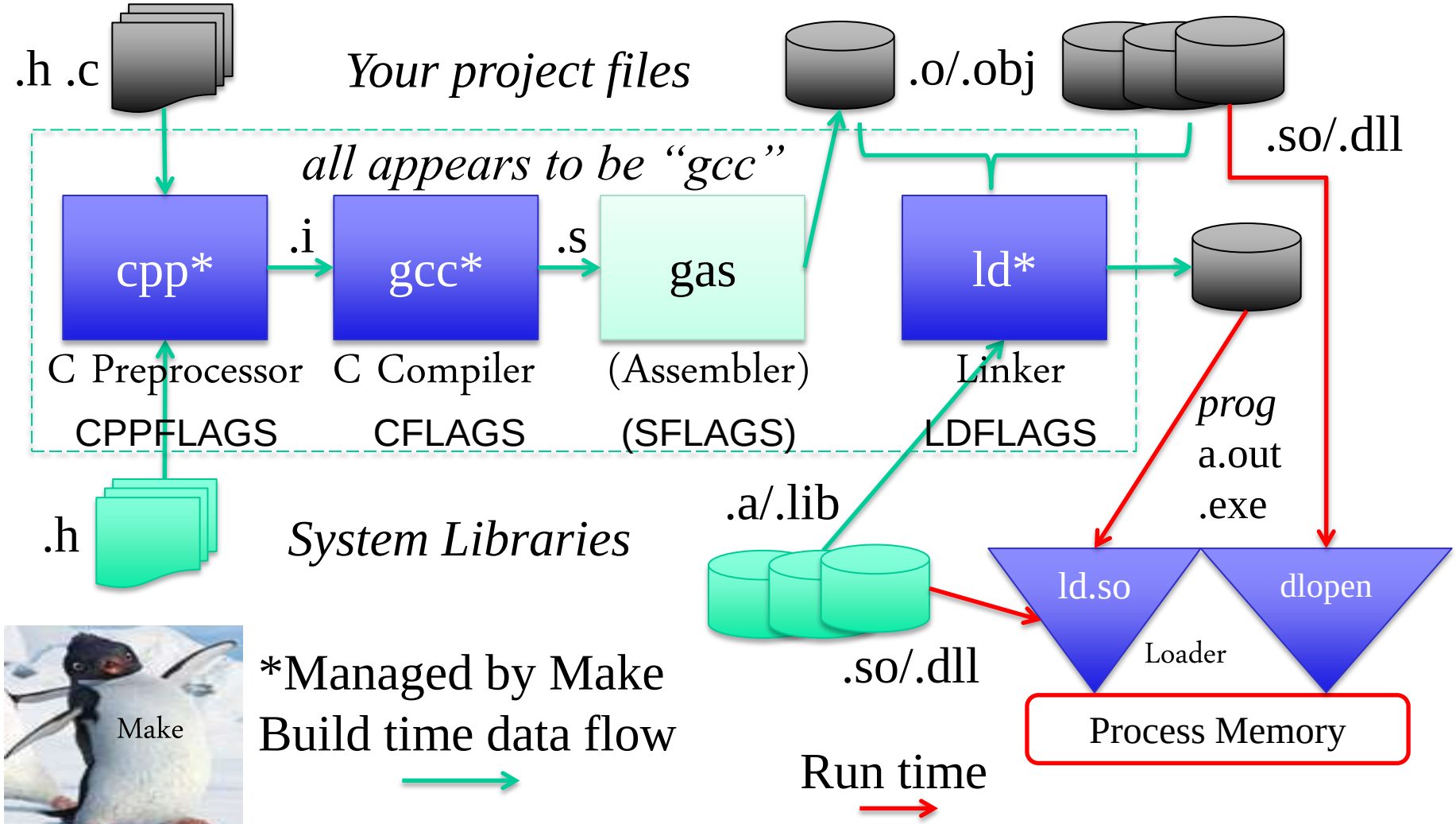


# Windows World

- The same tools are in use “under the hood,” but normally managed for you by the IDE (Visual Studio)
  - “IDE” = Interactive Development Environment
- Can get same effect in *NIX world via IDE like Eclipse

- Other languages have different tool chains

# The Big Picture




# (1) C Preprocessor = CPP

- *Purpose:* Interpret all the # directives in the .h and .c files *before* compiler sees the source code (intermediate .i file)
  - #include: merge in header files
  - #define: macros (with and without arguments)
    - macros are "expanded" by CPP
  - #ifdef, if, else, endif: conditional compilation
  - **Exception: #pragma** is passed on as compiler directive (ignored if compiler doesn't recognize)


# Prevent multiple/circular includes

- "Best practice" .h pattern (see 2500 Labs #8)

```
#ifndef CALUTIL_H
#define CALUTIL_H
```

...body of .h file...

```
#endif
```

  - On 1st #include "calutil.h", symbol gets defined
  - If it's #included again (even indirectly), the file's contents is skipped

# (2) C Compiler = GCC

- *Purpose:* Compile C language source code (.i file) into assembly language (.s)
  - Diagnoses abuses of language, and issues **warnings** and/or **errors**
  - Intermediate .i and .s files normally deleted after assembly → not seen by user unless requested (-save-temps option)

# (2+) Assembler = [G]AS

- *Purpose:* Assemble assembly code (.s) from compiler into object code (.o)
  - This tool step is normally transparent
  - There are options for controlling it, but rare to use them
  - You can insert assembly statements into your C program, e.g., to utilize vector instructions if CPU has vector processing unit

# (3) Linker (link editor) = ld

- *Purpose:* To stitch together objects (.o) into a single executable program file…
  - All the .o object files that make up the user program, plus
  - All the referenced system libraries
- Library linkage can be (Linux/Windows):
  - Static (.a/.lib): inserted in the executable file
  - Dynamic (.so/.dll): linked in at run time

# What is the linker doing?

- In the object files, there are tables of **external references** (e.g., call “printf”)
  - It reads all the libraries till it finds a matching **external definition** (printf() function in stdio)
  - *Static linking:* It pulls the definitions (function code and global variables) into the program file, and fixes up all the refs to point to their locations
  - *Dynamic linking:* It “makes a note” of what library file to get the definition from at run time

# (4) Loader = ld.so

- Command shell (or program) tells OS to execute a program file…
  - OS opens a fresh **process** (ref. CIS*3110) and calls loader to "fill 'er up" by copying the **segments** of the program file into different regions of the process's (virtual) memory
    - Program instructions ("text" segment)
    - Static data ("data" segment)
  - OS creates a new **stack** and empty **heap**, then transfers control to first instruction of program


# (4+) Loader = dlopen

Program using dynamically linked libraries:

- ld.so will load these when program starts
- but refs (calls) will be fixed up on demand
  - Strategy is because much/most of lib. won't be ref'd

- Some programs use **dlopen** to load shared object "plugins" on demand at run time
  - OS pauses process execution while loader grabs the object and links it into already-loaded code, then continues where left off


# (5) Make (minimalist approach)

- There are *many* (crazy) ways to achieve the same effects with makefiles
  - Programmers tend to develop their preferential ways of coding them
  - Good to get exposure to several people's
    - Learn/understand features you didn't realize exist
    - Avoid treating makefile as "magical incantation"
- 2 key concepts: **flags** and **targets**

# Flags control tools' options

Here are commonly-used flags; there are tons of others. (IDEs let you set flags indirectly by clicking options on property pages.)

**CC=gcc** *select gcc C compiler as front end*

1. Preprocessor: **CPPFLAGS=**

| | |
|---|---|
| **-I***include_file_dir* | -I~/myproj/include |
| **-D***symbol*[=*value*] | -DNDEBUG *disable assertions* |

- Equivalent to `#define` *symbol value*

# Compiler flags

2. Compiler: **CFLAGS=**

| | |
|---|---|
| **-g** | *save symbols for debugger* |
| **-O***n* | *optimization level (0,1,2,3)* |
| **-Wall -std=c11** | *all warnings, C11 standard* |
| **-fpic** | *position-independent code (for shared object library)* |

*This flag goes in the command, not in CFLAGS:*

| | |
|---|---|
| **-c** | *compile to .o file (don't link)* |

# Linker flags

3. Linker: **LDFLAGS=**

   **-L***library_dir* -L~/myproj/lib

   **-l***library* -lfoo → link in libfoo.a/.so

   *These flags go in the command, not in LDFLAGS:*

   **-o** *executable* -o caltest

   **-shared** *create shared object lib.*

# Why is gcc involved in tools 1-3?

- Gcc acts as a *front end* for {cpp, gcc, ld}
  - You keep executing "gcc" (or whatever **CC** designates) to get all these tools invoked
  - Nonetheless, gcc is *not* cpp or ld; it's just calling them for you!
  - Reason → convenience: gcc knows where all *its* libraries are installed, so it secretly beefs up *FLAGS for you, otherwise your commands would be much longer/messier and less portable

# How to make “make” do the heavy lifting for you

- Create a minimalist Makefile
  - Work backwards from the end result, the executable program, and what goes into it
  - Write a series of targets:

```
targetname: dependencies…
    commands
```

  - It’s like a recipe for preparing a dish!
  - *Special rule:* indent commands with Tab, not spaces!

# Makefile target: link step

- Start with the target=program you want to build and list its .o files for the link step

```
caltest: caltest.o calutil.o
	$(CC) $^ $(LDFLAGS) -o $@
```

$(*x*) the value of variable *x*

*Save typing with special make variables:*

$^ the target's dependencies (.o files)

$@ the target name itself

# Makefile targets: compile step

- Make a target for each of the .o object files, and list its .c (first) and .h dependencies

```
caltest.o: caltest.c calutil.h
   $(CC) $(CPPFLAGS) $(CFLAGS) -c $<

calutil.o: calutil.c calutil.h
   $(CC) $(CPPFLAGS) $(CFLAGS) -c $<
```

$<  target's *first* dependency (.c file)

# Full example makefile

```
CC=gcc
CPPFLAGS=
CFLAGS= -Wall -std=c11 -g
LDFLAGS=
#indent using tab (this is a comment)
caltest: caltest.o calutil.o
	$(CC) $^ $(LDFLAGS) -o $@
calutil.o: calutil.c calutil.h
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $<
caltest.o: caltest.c calutil.h
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $<
clean:		#always provide a "clean" target!
	rm -f *.o caltest
```

# “Commando style” makefile

```
CC=gcc
CPPFLAGS=
CFLAGS= -Wall -std=c11 -g
LDFLAGS=
#indent using tab (this is a com
caltest: caltest.o calutil.o

calutil.o: calutil.c calutil.h

caltest.o: caltest.c calutil.h

clean:      #always provide a “c
      rm -f *.o caltest
```

Why does this work?!

Make has **“implicit rules”** that know what should be done with .c .h and .o files.

Just set the flags properly, and you’re ready for action.

Try it both ways and observe the commands it executes.


# To use: just “make”

- **make** (first target by default) or **make caltest**
  - Executes commands for any target where age(dependency) is *newer* than target file
  - So, if you edit a .c file, its .o is recompiled and the executable is relinked automagically
  - If you edit a .h file, *all* .o’s that use it are recompiled and relinked!
- **make clean; make**
  - Deletes all .o’s and executable, rebuilds

# Without the Tool Chain gang…


C Preprocessor C Compiler Linker Loader Make

**You're on thin ice!**