Tools of the Trade

make

Building Software

gcc is our *compiler*

- Turns C code into machine code

ar is our *librarian*

- Gathers machine code files into groups called libraries

But calling these over and over is tedious!

```
$ gcc -std=c99 -c list.c
$ ar rcu liblist.a list.o
$ gcc -std=c99 -o trends trends.c -L. -llist
```

Building Software

Luckily, the process of building (and rebuilding) software can be automated!

make

- Automates the software build process

make

The basics:

- When you type **`make`** on the command line, **`make`** looks for a file named **`Makefile`**
- **`Makefile`** describes how your sources are converted into programs
- **`make`** can also take arguments, to change how the program is built:

```
make CFLAGS="-g"
```

make is a very common way to build software, but there are others

- autoconf, cmake, scons, Xcode, Visual Studio, complicated mess of scripts, …

Makefile

A **`Makefile`** is a list of *targets*, *dependencies*, and *commands*

A simple **`Makefile`**:

```
hello: hello.c world.c
	gcc -std=c99 -o hello hello.c world.c
```

Makefile

A **`Makefile`** is a list of *targets*, *dependencies*, and *commands*

A simple **`Makefile`**:

Target Dependencies

```
hello: hello.c world.c
        gcc -std=c99 -o hello hello.c world.c
```

Command

Makefile

A simple **`Makefile`**:

```
hello: hello.c world.c
	gcc -std=c99 -o hello hello.c world.c
```

Makefile

A simple **`Makefile`**:

Capital M

Yes, case does matter!

```
hello: hello.c world.c
	gcc -std=c99 -o hello hello.c world.c
```

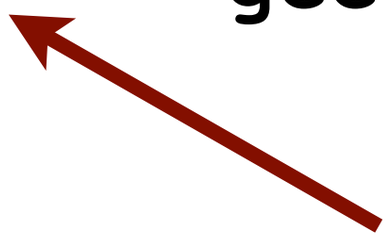
Makefile

A simple `Makefile`:

Capital M

Yes, case does matter!

```
hello: hello.c world.c
	gcc -std=c99 -o hello hello.c world.c
```



Tab (NOT spaces!)

Some editors will put spaces even if you ask for a tab, so be careful!

Building with make

Building with make

When you `make`, if the *target* does not exist, or *dependencies* have changed, it builds

Building with make

When you `make`, if the *target* does not exist, or *dependencies* have changed, it builds

Building with make

When you **`make`**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
$ ls
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
$ ls
Makefile    hello    hello.c    world.c
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
$ ls
Makefile    hello    hello.c    world.c
$ make
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
$ ls
Makefile    hello    hello.c    world.c
$ make
make: `hello' is up to date
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
$ ls
Makefile    hello    hello.c    world.c
$ make
make: `hello' is up to date
$ vi hello.c
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
$ ls
Makefile    hello    hello.c    world.c
$ make
make: `hello' is up to date
$ vi hello.c
$ make
```

Building with make

When you **make**, if the *target* does not exist, or *dependencies* have changed, it builds

```
$ ls
Makefile    hello.c    world.c
$ make
gcc -std=c99 -o hello hello.c world.c
$ ls
Makefile    hello    hello.c    world.c
$ make
make: `hello' is up to date
$ vi hello.c
$ make
gcc -std=c99 -o hello hello.c world.c
```

Multiple targets

If your Makefile lists multiple targets, only the first is default

- Makefile:

```
hello: hello.c world.c
        gcc -std=c99 -o hello hello.c world.c

goodbye: goodbye.c world.c
        gcc -std=c99 -o goodbye goodbye.c world.c
```

- Command line:

```
$ make                  # builds hello
$ make goodbye          # builds goodbye
```

Dependencies and targets

Dependencies can also be targets!

- Makefile:

```
hello: hello.c libworld.a
        gcc -std=c99 -o hello hello.c -L. -lworld

libworld.a: world.c
        gcc -std=c99 -c world.c
        ar rcu libworld.a world.o
```

- Command line:

```
$ make
gcc -std=c99 -c world.c
ar rcu libworld.a world.o
gcc -std=c99 -o hello hello.c -L. -lworld
```

Pseudo-targets

Some targets may not actually be programs or files

```
all: hello goodbye

hello: hello.c world.c
	gcc -std=c99 -o hello hello.c world.c

goodbye: goodbye.c world.c
	gcc -std=c99 -o goodbye goodbye.c world.c

clean:
	rm hello goodbye
```

Pseudo-targets

Some targets may not actually be programs or files

There is no program called "all"

But building "all" builds both hello and goodbye

```
all: hello goodbye

hello: hello.c world.c
	gcc -std=c99 -o hello hello.c world.c

goodbye: goodbye.c world.c
	gcc -std=c99 -o goodbye goodbye.c world.c

clean:
	rm hello goodbye
```

Pseudo-targets

Some targets may not actually be programs or files

There is no program called "all"

But building "all" builds both hello and goodbye

```
all: hello goodbye

hello: hello.c world.c
	gcc -std=c99 -o hello hello.c world.c

goodbye: goodbye.c world.c
	gcc -std=c99 -o goodbye goodbye.c world.c

clean:
	rm hello goodbye
```

"make clean" is a common pseudo-target for cleaning up

Compiling vs linking

When a GCC command includes multiple C files, each are compiled, then all are linked into a single program:

```
gcc -std=c99 -o hello hello.c world.c
```

- Builds `hello.c` into `hello.o`
- Builds `world.c` into `world.o`
- Links `hello.o` and `world.o` into `hello`

If some C files don't change, you're wasting time recompiling them. `make` to the rescue!

Compiling vs linking


```
hello: hello.o world.o
        gcc -std=c99 -o hello hello.o world.o

hello.o: hello.c
        gcc -std=c99 -c hello.c

world.o: world.c
        gcc -std=c99 -c world.c
```

Variables

Avoid repetition and be flexible by making variables

- Makefile:

```
CC=gcc
CFLAGS=-std=c99 -O2 -g

hello: hello.c
        $(CC) $(CFLAGS) -o hello hello.c
```

- Command line:

```
$ make
gcc -std=c99 -O2 -g -o hello hello.c
$ rm hello ; make CFLAGS="-g"
gcc -g -o hello hello.c
```

Patterns

Very common patterns (such as compiling .c files into .o files) can be grouped

e.g. a target for all .o files:

```
%.o: %.c
        $(CC) $(CFLAGS) -c $<
```

gdb

Debugg□□Segmentation fault

Misuse of memory can cause crashe□□□□

□□

Bus error

Debugging

Misuse of memory can cause crashes and odd behavior

gdb is our *debugger*

- Helps understand why misbehaving code misbehaves

gdb is just one tool in your arsenal

- don't forget how useful printfs can be!

gdb has its flaws

- "But my code works under gdb!"

gdb basics

Compile your program with -g

- `$ make CFLAGS="-g"`

Run your program under gdb:

- `$ gdb ./hello`

```
...
(gdb) run
...
Program received SIGSEGV, Segmentation fault.
... in main () at hello.c:3
3                       *((int *) NULL) = 0;
(gdb)
```

gdb demo

(Demo)

Recap

gdb commands:

- bt: Tells you the “backtrace” (all functions in the call stack)
- print: Shows the value of variables, expressions, etc

  print <expression>
- list: Shows surrounding code
- step: Single-step execution
- next: Bigger single-step execution
- break: Sets breakpoints

  break foo.c:42

mudflap

Bounds Checking

C does not perform bounds checking on arrays

```
int main()
{
    int arr[10];
    for (int i = 1; i <= 10; i++) arr[i] = i;
    printf("%d\n", arr[1]);
    return 0;
}
```

Without bounds checking, buggy code has undefined behavior; it may work on some systems, but fail on others

mudflap

mudflap is a library that comes with gcc

To use mudflap, compile with -g -fmudflap -lmudflap

- gcc outofbounds.c -g -fmudflap -lmudflap

mudflap will tell you where things go wrong

mudflap output


```
$ ./a.out
*******

mudflap violation 1 (check/write): time=1329410522.640484 ptr=0x7fff2b4ffda0
size=44

pc=0x7f03809ef311 location=`outofbounds.c:6:42 (main)'

      /usr/lib/x86_64-linux-gnu/libmudflap.so.0(__mf_check+0x41) [0x7f03809ef311]
      ./a.out(main+0xa4) [0x400a88]
      /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xfd) [0x7f038067cead]
Nearby object 1: checked region begins 0B into and ends 4B after

mudflap object 0x23a55a0: name=`outofbounds.c:5:9 (main) arr'

bounds=[0x7fff2b4ffda0,0x7fff2b4ffdc7] size=40 area=stack check=0r/4w
liveness=4

alloc time=1329410522.640468 pc=0x7f03809eea51

number of nearby objects: 1

1
```

mudflap output


```
$ ./a.out
*******
mudflap violation 1 (check/write): time=1329410522.640484 ptr=0x7fff2b4ffda0
size=44
pc=0x7f03809ef311 location=`outofbounds.c:6:42 (main)'
    /usr/lib/x86_64-linux-gnu/libmudflap.so.0(__mf_check+0x41) [0x7f03809ef311]
    ./a.out(main+0xa4) [0x400a88]
    /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xfd) [0x7f038067cead]
Nearby object 1: checked region begins 0B into and ends 4B after
mudflap object 0x23a55a0: name=`outofbounds.c:5:9 (main) arr'
bounds=[0x7fff2b4ffda0,0x7fff2b4ffdc7] size=40 area=stack check=0r/4w
liveness=4
alloc time=1329410522.640468 pc=0x7f03809eea51
number of nearby objects: 1
1
```

valgrind

Memory Leaks

C does not have garbage collection

"If it's not one thing (segfaults), it's another (leaks)"

By failing to free memory, programs "leak"

If the leak is in a loop or often-used function, can cause huge problems!

Memory Leaks

valgrind is a tool for detecting memory leaks

(and about 1,000 other things)

(demo)

Summary

Summary

Decades of use means C has a rich suite of tools available

I've only shown you a few:

- make
- gdb
- mudflap
- valgrind

With judicious use of tools, programs can be error fr□