Lecture - XIV
Makefiles & Libraries

CSC 4304 - Systems Programming
Fall 2008

Tevfik Koşar
Louisiana State University
October 23rd, 2008

Splitting C Programs into Multiple Files

- All our programs so far are written in a single file
- But programs can be very big!
  - E.g., Linux-2.6.0 contains 5,929,913 lines of C code
- Let’s split our programs into multiple source files
  - Easier to write and update
  - Especially with multiple programmers
    - Each programmer writes into his/her own file
  - It is easier to recompile
    - If you change a small part of the program, you can recompile just the part that has changed

Modular C Programming

- A C program usually contains:
  - Multiple .c files: contain the functions and global variables
  - Multiple .h files: contain declarations of functions, types and variables
- Unlike in Java, you can put as many functions/variables/types per file as you want
  - It is up to you to organize everything
  - But there are general rules that will help you...
  - Most important: keep related things in a single file

Definition vs Declaration

- A definition actually creates a function/variable and gives it a value
  - “From now on, variable foo of type int will be created”
  - “From now on, function bar() will have the following prototype and realize the following operations.”

```c
int foo;
double bar(double x, double y) {
    return x + y;
}
```

- A declaration simply informs the compiler that something does exist
  - “Trust me, it will be defined somewhere else”

```c
extern int foo;
double bar(double, double); /* no function code here! */
```

Calling an External Function

- If you want to call a function in a piece of code, you must first declare the prototype of the function
  - You do not need to write the full code of the function
  - A prototype (i.e., interface) is enough
  - Of course, the code of the function must be present in another file of the program!

```c
int this_func_is_defined_somewhere_else(char *);
int foo() {
    return this_func_is_defined_somewhere_else("foo");
}
```

- A function must be defined only once in a program
  - Otherwise the compiler wouldn’t know which one to use
  - But it can be declared any number of times
  - Provided all declaration are the same...

Using an External Variable

- To use a (global) variable defined in another file you must first declare it
  - Attention: you must define the variable only once

```c
/* file1.c */
int my_variable; /* the variable is declared but not defined */
int foo() {
    return my_variable++;
}
```

```c
/* file2.c */
int my_variable; /* the variable is declared and defined here */
```
Using Header Files

- Some information must be present in multiple files
  - Better to write them only once in a "header" file
  - And include the header file whenever it is needed
- Header files (*.h) should contain:
  - Function prototypes
  - Type declaration
  - Global variable declarations (but not definitions!)
- C files (*.c) should contain:
  - #include <standard_files.h>
    * Includes files from /usr/include, /usr/local/include etc.
  - #include "header_files.h"
    * Includes files from the working directory
  - Function code (definitions)
  - Global variable (definitions)
- Each C file usually has its corresponding header file...

Example

- A program that exchanges messages across a network

```c
#include <stdio.h>
#include <stdlib.h>
#include "message.h"

#define NETWORK_L

struct message { /* header */
  char *message;
  int length;
};

struct message *create_message(char *message) {
  struct message *msg = (struct message *) malloc(sizeof(struct message));
  strcpy(msg->buf, message);
  return msg;
}

int main() {
  int port_nb;
  struct message *msg;

  port_nb = 1024; /* pick a port */

  msg = create_message("Hello, World!");

  send_message(msg, port_nb);

  free(msg);

  printf("Message sent.");
}
```

message.h

- message.h contains:
  - The declaration of struct message
  - The declaration of function create_message()

```c
struct message { /* header */
  char *message;
  int length;
};

struct message *create_message(char *message);
```

message.c

- message.c contains:
  - Includes standard header files string.h and stdlib.h
  - Includes header file message.h (it contains the declaration of struct message)
  - Defines function create_message

```c
#include <stdio.h>
#include <stdlib.h>
#include "message.h"

#define NETWORK_L

struct message *create_message(char *message) {
  struct message *msg = (struct message *) malloc(sizeof(struct message));
  strcpy(msg->buf, message);
  return msg;
}
```

network.h

- network.h contains:
  - Define NETWORK_L
  - Include "message.h" /* Why is this required? */

```c
#define NETWORK_L

#include "message.h"

struct address {
  char *ip;
  int port;
};

struct address *create_address(char *ip);

int recv_message(struct message *msg, struct address *from);
```

```c
#include <stdio.h>
#include <stdlib.h>
#include "message.h"

#define NETWORK_L

#include "message.h"

struct address {
  char *ip;
  int port;
};

struct address *create_address(char *ip);

int recv_message(struct message *msg, struct address *from);
```

```c
#include <stdio.h>
#include <stdlib.h>
#include "message.h"

#define NETWORK_L

#include "message.h"

struct address {
  char *ip;
  int port;
};

struct address *create_address(char *ip);

int recv_message(struct message *msg, struct address *from);
```
Compiling it All Together

- Compile each C file separately into an object file
  ```
  $ gcc -c "main.c"
  $ gcc -c "network.c"
  $ gcc -c "main.c"
  $ gcc -c "network.c"
  $ gcc -c "main.c"
  $ gcc -c "network.c"
  $ gcc -c "main.c"
  ```
- This creates files message.o, network.o and main.o.
- Link all object files into an executable
  ```
  $ gcc message.o network.o main.o
  $ gcc network.o main.o
  ```
- This creates file a.out

Building Complex Programs

- Imagine that you write a program split into 100 C files and 100 header files
  - To compile your program, you must call gcc 101 times (perhaps with long option lines)
- What happens when you update one of these files?
  - You can recompile everything from scratch
    - But it can take a lot of time
  - You can decide to recompile only the parts which have changed
  - Much faster
- What happens if the updated file is a header file?
  - You must recompile all C files which include it
  - This is getting quite complex...
- make is a standard tool which will do the job for you
Using make

- To use make, you must write a file called `Makefile`
  - It defines dependencies between files
    - `...` and the command to generate each file from its dependencies

```bash
# This is a comment
main: message.o network.o main.o
    gcc -o main main.o message.o network.o
message.o: message.c message.h
    gcc -c -Wall message.c
network.o: network.c network.h message.h
    gcc -c -Wall network.c
main.o: main.c main.h network.h message.h
    gcc -c -Wall main.c
```

- `...` means `tab`: you cannot use spaces there!

Using make

- If you type `make main`, make will do all that is necessary to generate file main:
  - To generate main, I first need to have files message.o, network.o and main.o.
  - These files do not exist, let’s try to create them
    - To generate message.o I first need to have files message.c and message.h
    - OK, I already have them.
    - Let’s generate message.o by calling gcc -c -Wall message.c
    - To generate network.o I first need to have files network.c, network.h and message.h
    - etc...
  - Let’s generate file main by calling gcc -o main main.o message.o network.o

Using make to re-compile a program

- If you update a few files, you want to recompile just what is necessary
- `make` will check the dates of your files:

```
target: dependency1 dependency2 dependency3
    command
```

- If you updated dependency1 after target was generated, then you must re-generate target
- If the target is more recent than all its dependencies, then no re-generation is necessary

- You must not forget dependencies!
- Otherwise, make will not recompile all that is necessary

Generating Dependencies

- `make depend` will generate dependencies automatically
  - Just create one more rule:

```
depend:
    - make depend message.o network.o main.o
```

- If you type `make depend`, the program `make depend` will be called
  - It will read files message.c, network.c and main.c and generate dependencies automatically
  - Dependencies will be added at the end of your `Makefile`

```
# DO NOT DELETE
main.o: /run/include/stubs.h /run/include/features.h /sys/include/sys/defs.h
main.o: /run/include/gnu/stubs.h
main.o: /run/include/stdio.h
main.o: /run/include/stdlib.h
main.o: /run/include/bits/types.h /run/include/bits/verdana.h
main.o: message.h /run/include/string.h network.o
network.o: network.h message.h /run/include/features.h
network.o: /sys/include/sys/defs.h /run/include/gnu/stubs.h
network.o: /lib/libc/4.4-rc1-linux/h4.4/include/stdio.h
```

Implicit Rules

- Very often, the command to compile a given type of files is the same
  - `gcc -c FOO.c`
  - All *.c files depend on the corresponding *.c files and are generated using the command `gcc -c XXX.c`

```
gcc -c $< -o $@
```

- `$<` means "the name of the dependency file" (here: `FOO.c`)
- `$@` means "the name of the target" (here: `FOO.o`)

Using Variables in Makefiles

- You can create variables in your `Makefiles`
  - The list of all your *.c files, etc.

```
CC = gcc
CFLAGS = -g wall
SRC = main.c network.c message.c
OBJ = main.o network.o message.o
main: $(OBJ)
    $(CC) $(CFLAGS) $< -o $@
```

- We can write rules which do not create any file

```
clean:
    rm main *.o
```
Libraries

- Libraries are precompiled sets of functions ready to be used in programs
  - For example: you wrote a set of functions to send/receive network messages
  - Let’s put them into a library
  - You can use the library in any program which needs networking
  - You can let other programmers use your library
- Libraries always come with one or more header files
  - To declare the types/functions/variables present inside the library
- Libraries are named like this:
  - libsomething.a (static libraries)
  - libsomething.so (dynamic libraries)

Crating a Static Library

- A static library is made of a number of *.o files
  - `ar r libnetwork.a message.o network.o`
  - `ranlib libnetwork.a`
- or creates the library
  - It can also do other operations on libraries (modifying an existing library, extracting parts from a library, etc.)
  - Read the man page...
- `ranlib` creates an index of all functions in the library
  - And adds the index to the library

Crating a Dynamic Library

- This is slightly different from creating a static library:
  1. Compile each .c file using option `-DPIC`
  2. Generate the shared library using `gcc -shared`
  3. You must specify the name of the library (libsomething.so)

```
$ gcc -c message.c
$ gcc -c -DPIC network.c
$ gcc -shared -o libsomething.so message.o network.o
```

Using a Static Library

- Let’s write a program `main.c` which uses our library `libnetwork.a`
  - Write `main.c` normally (include header files message.h and network.h that come with the library)
  - Compile main.c normally (`gcc -c main.c`)
  - Link `main.o` with the library:

```
$ gcc -o main main.o -L . -lsnetwork
```

Using a Dynamic Library

- You can link your program the same way as when linking with a static library:

```
$ gcc -o main main.c
$ gcc -o main main.o -L . -lsnetwork
```

- Remember: dynamic libraries are loaded at runtime!
  - The dynamic linker must be able to find the library when you start the program
  - It will search libraries with the right name in paths present in your environment variable `LD_LIBRARY_PATH`

```
$ /main
./main: error while loading shared libraries: libmessage.so: cannot open shared object file: No such file or directory
$ export LD_LIBRARY_PATH=/home/gpiersc/pydirectory
$ /main
```
Using a Dynamic Library

- How do you know which libraries an executable requires?

```bash
$ ls /etc 
lib.so.1
```

Dynamically Loaded Libraries

- Normally, dynamic libraries are loaded automatically when a program starts
- But sometimes you want to load a dynamic library while a program is running
  - Example: you start a program which receives a dynamic library through the network and uses it for further communication
- You can explicitly load dynamic libraries:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    void *lib;
    /* ... */
    void *lib = dlopen("/path/to/library.so", RTLD_LAZY);
    /* ... */
    */
}
```

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), and J. Kim (KAIST).