Lecture - XIV

Makefiles & Libraries

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 baz() will have the following prototype and realize the following operations."

```c
int foo;

double baz(double x, double y) {
    return x*x + y*y;
}
```

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

```c
extern int foo;

double baz(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 somewherelse(char *);
int foo() {
    return this_func_is_defined somewherelse("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 */
extern 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 informations must be present in multiple files
  - Better to write them only once in a “header” file
  - And include the header file wherever 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

![Diagram of a program that exchanges messages across a network]

```c
#include <standard_files.h>
#include "header_files.h"

int main()
{
    int port_nb;  // (global variable)

    create_message();
    send_message();
    recv_message();

    struct message
    {
        // Message fields
    } message;

    struct address
    {
        // Address fields
    } address;

    printf("Hello, World!");
    return 0;
}
```
Example

- A program that exchanges messages across a network

```
#include <message.h>

main()

int port_nb;
  (global variable)

create_message()

send_message()

recv_message()

strncpy()

create_address()

printf()

struct message

struct address

message.c, message.h

network.c, network.h
```

message.h

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

```c
#ifndef _MESSAGE_H_
#define _MESSAGE_H_

struct message {
  char buf[1024];
  int length;
};

struct message *create_message(char *message);

#endif /* _MESSAGE_H_ */
```
message.c

message.c:

- Includes standard header files string.h and stdlib.h (they contain the prototypes of strcpy and malloc)
- Includes header file message.h (it contains the declaration of struct message)
- Defines function create_message

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

struct message *create_message(char *message) {
    struct message *m = (struct message *) malloc(sizeof(struct message));
    strcpy(m->buf, 1023);
    return m;
}
```

network.h

```c
#define _NETWORK_H_

#include "message.h" /* Why is this required? */

struct address {
    char ip[16];
    int port;
};

struct address *create_address(char *ip);
int send_message(struct message *m, struct address *dest);
int recv_message(struct message *m, struct address *from);

#undef _NETWORK_H_ */
```
#include "network.h"
#include "main.h"

struct address *create_address(char *ip) {
    struct address *a = (struct address*) malloc(sizeof(struct address));
    strcpy(a->ip, ip);
    a->port = port_nb;
    return a;
}

int send_message(struct message *m, struct address *dest) {
    /* ... */
}

int recv_message(struct message *m, struct address *from) {
    /* ... */
}

- Can you guess what main.h contains?
- Why don’t we include message.h?
- What would happen if we included it?
main.c

#include "main.h"
#include "network.h"

int port_nb; /* instantiate the global variable */

int main() {
    struct message *m = create_message("Hello, world!");
    struct address *a = create_address("130.37.193.66");
    send_message(m, a);
    recv_message(m, a);
    printf("Received: %s\n", m.buf);
}

Compiling it All Together

- Compile each C file separately into an object file

```bash
$ gcc -c -Wall message.c
$ gcc -c -Wall network.c
$ gcc -c -Wall main.c
$
```

- This creates files message.o, network.o and main.o.

- Link all object files into an executable

```bash
$ gcc message.o network.o main.o
$
```

- This creates file a.out
Compiling it All Together

- One object file must define a main() function:

```bash
$ gcc message.o network.o main.o
/usr/lib/gcc/x86_64-redhat-linux/3.4.2/../../../lib64/crt1.o(.text+0x21): In function `start':
    undefined reference to `main'
collect2: ld returned 1 exit status
```

- All functions and variables must be defined:

```bash
$ gcc message.o network.o main.o
main.o(.text+0xa): In function `main':
    undefined reference to `create_message'
collect2: ld returned 1 exit status
```

- They must be defined only once:

```bash
$ gcc message.o network.o main.o
network.o(.text+0x0): In function `create_message':
    multiple definition of `create_message'
message.o(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

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 takes 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

```make
# This is a comment

main: message.o network.o main.o
  -> gcc -c 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 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

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

```
depend:
        makedepend message.c network.c main.c
```

- If you type "make depend", the program `makedepend` 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: /usr/include/stdio.h /usr/include/features.h /usr/include/sys/cdefs.h
main.o: /usr/include/gnu/stubs.h
main.o: /usr/lib/gcc/x86_64-redhat-linux/3.4.2/include/stdbool.h
main.o: /usr/include/bits/types.h /usr/include/bits/wordsize.h
main.o: message.h /usr/include/string.h network.h
network.o: network.h message.h /usr/include/string.h /usr/include/features.h
network.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
network.o: /usr/lib/gcc/x86_64-redhat-linux/3.4.2/include/stddef.h
# etc...
```
Implicit Rules

- Very often, the command to compile a given type of files is the same
  - gcc -c F00.c
  - All *.o files depend on the corresponding *.c file and are generated using the command gcc -c XXI.c

  %.o: %.c
  gcc -c $< -o $@

  - `$<` means “the name of the dependency file” (here: F00.c)
  - `$@` means “the name of the target” (here: F00.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) -o $@ $(OBJ)

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

depend:
  makodepend $(SRC)

clean:
  # We can write rules which do not create any file
  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)

Static vs Dynamic Libraries

There are two kinds of libraries:

- **Static libraries:**
  - When compiling a program with the library, the library code will be added to the program
  - If the library is big, this makes a big executable file!
    - For example, the standard C library takes around 2.5 MB.
  - All programs must be linked with this library!
    - This is a waste of storage space

- **Dynamic libraries:**
  - The linked executable does not contain the library
  - It just contains a reference: "this executable requires library /usr/lib/libc.so"
  - When you execute the program, the library will be loaded automatically before the program starts
Crating a Static Library

- A static library is made of a number of *.c files

```
$ ar r libnetwork.a message.o network.o
$ ranlib libnetwork.a
$
```

- `ar` 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

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-network
$
```

- `gcc` will search for a file called `libnetwork.a` or `libnetwork.so` in the standard directories (`/usr/lib`, `/usr/local/lib`)
- If you don’t have write access in `/usr/lib`, then you must specify where `libnetwork.a` can be found:

```
$ gcc -o main main.o -L -l-network
$
```
**Crating a Dynamic Library**

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

```bash
$ gcc -c -fPIC message.c
$ gcc -c -fPIC network.c
$ gcc -shared -o libnetwork.so message.o network.o
$ ...
```

**Using a Dynamic Library**

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

```bash
$ gcc -c main.c
$ gcc -o main main.o -L -lnetwork
$ ...
```

- 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`

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

- How do you know which libraries an executable requires?

```bash
$ ldd main
    libmessage.so => not found
    libc.so.6 => /lib64/ld-linux-x86-64.so.2 (0x0000000000000000)
    /lib64/ld-linux-x86-64.so.2 (0x0000000000000000)
$ LD_LIBRARY_PATH=$PWD
$ ldd main
    libmessage.so => /home/gpierre/foo/libmessage.so (0x0000000000000000)
    libc.so.6 => /lib64/ld-linux-x86-64.so.2 (0x0000000000000000)
    /lib64/ld-linux-x86-64.so.2 (0x0000000000000000)
$```

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 <dlfcn.h>
void *dlopen(const char *filename, int flag);

#include <dlfcn.h>

int main() {
    void *mylibrary;
    /* ... */
    mylibrary = dlopen("/home/gpierre/mydirectory/libnetwork.so", RTLD_NOW);
    /* ... */
}
```
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 (UCheicago), and J.Kim (KAIST).