Recitation - IV

Building Complex Programs with Makefiles
Prof. Tevfik Kosar

Presented by Fatih Bulut

University at Buffalo
October 2013
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 somewherewhere_else(char *);

int foo() {
    return this_func_is_defined somewherewhere_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 */
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
Example

A program that exchanges messages across a network
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

Includes standard header files string.h and stdlib.h (they contain the prototypes of strncpy 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));
    strncpy(m->buf, 1023);
    return m;
}
```
#ifndef _NETWORK_H_
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_ /* _NETWORK_H_ */
```c
#include "network.h"
#include "main.h"

struct address *create_address(char *ip) {
    struct address *a = (struct address*) malloc(sizeof(struct address));
    strncpy(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.h

#include "main.h"
#define _MAIN_H_

extern int port_nb;    /* Declare the global variable */

/* Do we need to declare the prototype of function main() here? */
#endif /* _MAIN_H_ */
#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

```
$ 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

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

This creates file `a.out`
Compiling it All Together

- One object file must define a `main()` function:
  
  ```
  $ gcc message.o network.o main.o
  /usr/lib/gcc/x86_64-redhat-linux/3.4.2/.../.../.../.../lib64/crti.o(.text+0x21): In
  function ‘_start’: undefined reference to ‘main’
collect2: ld returned 1 exit status
  $
  ```

- All functions and variables must be defined:
  
  ```
  $ 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:
  
  ```
  $ 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

```
# 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
```

'\t' → '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:

<table>
<thead>
<tr>
<th>target: dependency1 dependency2 dependency3</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
</tr>
</tbody>
</table>

  - 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/undef.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/undef.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 XXX.c

<table>
<thead>
<tr>
<th>%.o: %.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc -c $&lt; -o $@</td>
</tr>
</tbody>
</table>

- '$<$' 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.

```makefile
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:
    makedepend $(SRC)

clean:
    # We can write rules which do not create any file
    rm main *.o
```
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).