Building Complex Programs with Makefiles

Presented by Sonali Batra

University at Buffalo

October 2nd, 2012

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.
  - "The name x, variable foo of type int will be created."
- A declaration simply informs the compiler that something does exist.
  - "Tell me, it will be defined somewhere else."

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.

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.

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.
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 wherever it is needed
- Header files (*.h) should contain:
  - Function prototypes
  - Type declarations
  - 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_file.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

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

struct message {
    char text[100];
    int port;
};

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

```
// message.h

struct message {
    char text[100];
    int port;
};

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

```
// message.c

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

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

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

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

```
// network.h

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

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

```
// network.c

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

struct message *create_message(char *text) {
    struct message *message = malloc(sizeof(struct message));
    strcpy(message->text, text);
    return message;
}
```
Compiling it All Together

- Compiling each C file separately into an object file

```
gcc -c network.c
```
```
gcc -c main.c
```
```
gcc -c main.h
```

- This creates the object files

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

- Link all object files into an executable

```
gcc message.o 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 options)
- 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
    - More efficient
  - 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
make: 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.o network.o message.h
      gcc -c -Wall main.o

* "\t" means "tab", you cannot use spaces there!

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:

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

- 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 this before your targets:
make depend
```

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 file and are generated
  using the command gcc -c X.c

```
C:.c
  gcc -c $< -> $@
```

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

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
HDR = main.c network.c message.c
make: (CCC) $$(CFLAGS) \$(CC) \$(CFLAGS) -> $(CC)
```

```
1.o: L.c
  $(CCC) $(CFLAGS) \$(CC) $< \$(CFLAGS) -> $@
```

```
depend: make depend
  make depend
```

```
# We can write rules which do not create any file
```
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 (UCicago), and J.Kim (KAIST).