
while(k != 2*i-1)
if (count <= rows-1)
printf("%d ", i+k);
++count;

Makefiles

Software Development Tools

1. Configuring và Building một Program
   - GCC
   - Makefiles
   - Autotools

2. Writing và Managing code

3. Packaging và Distributing the application

4. Debugging và Profiling

Why makefiles

- Program thường cấu thành từ rất nhiều source file:
  - Được viết bởi nhiều người.
  - Thường được tổ chức bằng nhiều thư mục con.
  - Và có dependency phức tạp.
    (to compile X you first have to compile Y, that requires to compile the library Z..)

- Khó khăn:

- Compile lại toàn bộ project cho mỗi sự thay đổi nhỏ là một việc rất kém hiệu quả và có thể yêu cầu thời gian compile lâu.
- Maintain một project phức tạp sẽ trở thành một cơn ác mộng.

Make và Makefiles

- Makefiles có từ những năm 1977 nhưng hiện tại vẫn là công cụ dùng để quản lý build process được sử dụng rộng rãi nhất.
  - Có nhiều versions khác nhau: e.g. GNU `make`, bsd `make`, Microsoft `nmake`

- Makefile là một file script mô tả:
  - Project structure: các file và các dependency giữa chúng.
  - Cách để tạo ra target program từ mỗi dependency.

- Instructions để compile mỗi file.

- Các instruction bên trong mỗi makefile được interpreted và executed bởi chương trình make.
  - Bằng việc kiểm tra các dependencies, make chỉ compile lại những cái cần thiết để compile lại.

Cấu trúc của một Make File

- Makefile là một text file chứa một hoặc nhiều rule.

```
target: dependency1 dependency2 …
    action1
    action2
    ...
```

TABs (bắt buộc) Nếu thoả các dependency, target sẽ được build bằng việc thực thi các action

- Makefile cũng chứa các variable (macro) assignments:
  - Definition: CC = gcc
  - Cách sử dụng: echo ${CC} hay echo $(CC)
  - Theo convention, tất cả các variable trong một makefile đều phải IN HOA.

A simple Makefile

```
all: func.o main.o
    gcc func.o main.o -o main

func.o: func.c func.h
    gcc -g -c func.c

main.o: main.c func.h
    gcc -g -c main.c
```

```
clean:
    rm -f *.o
    rm main
```

Target "all" phụ thuộc vào hai target func.o và main.o

Target "func.o" phụ thuộc vào func.c

Thực tế, bất kỳ .o file nào cũng mặc định phụ thuộc vào các .c file tương ứng của nó. (func.c có thể bỏ qua)

Target "clean" không phụ thuộc vào bất kỳ dependency nào

How It works

- Đầu tiên, `make` command sẽ tìm kiếm một file tên là `makefile` hoặc `Makefile` trong current directory.
- Sau đó, nó sẽ parse nội dung của file và tạo ra dependency tree.
- Tiếp theo, kiểm tra nếu có bất kỳ prerequisite target nào không tồn tại hoặc mới hơn target (bằng việc kiểm tra timestamps)
  - Nếu có, đầu tiên sẽ thực thi các prerequisite target (theo trình tự đệ quy, tức sẽ thực thi

các prerequisite target trước đó nữa nếu cần thiết).
  - Nếu target không phải là một filename, make không thể xác định được timestamp của nó, vậy nên make sẽ luôn thực thi các target đó.

- Với mỗi action trong một target, `make` sẽ in ra action đó và thực thi nó. (trong một subshell riêng lẻ cho mỗi action command)

A simple Makefile

```
# This is a comment

all: func.o main.o
    gcc func.o main.o -o main

func.o: func.c func.h
    gcc -g -c func.c

main.o: main.c func.h
    gcc -g -c main.c
```

```
clean:
    rm -f *.o
    rm main
```

make target : thực thi các action tương ứng với target (và các required dependency)

make : thực thi target đầu tiên trong makefile

Variable Assignment

- Có bốn cách để gán một variable.

  := (simple assignment)

  = (recursive assignment)

  - Vế bên phải của phép gán sẽ không được kiểm tra lập tức cho đến khi variable được sử dụng.
  - Example:

    COMPILE = ${CC} ${CFLAGS}

  ?= (conditional assignment)

- Assignment chỉ được thực hiện khi variable chưa được gán giá trị.

+= (append)

- Append text vào variable

Patterns

- Định nghĩa pattern:
  - Dấu phần trăm, %, có thể được sử dụng để thực hiện wildcard matching để viết các target có thể tổng quan.
  - Khi dấu % xuất hiện trong dependency list, nó sẽ được thay thế cho text tương ứng ở target.
  - Example:

```
%.o : %.c
    gcc -c ????
```

Patterns

- Định nghĩa pattern:
  - Dấu phần trăm, %, có thể được sử dụng để thực hiện wildcard matching để viết các target có thể tổng quan.
  - Khi dấu % xuất hiện trong dependency list, nó sẽ được thay thế cho text tương ứng ở target.
  - Example:

```
%.o : %.c
    gcc -c ????
```

PROBLEM: How can I reference the filename in the action?

Special Variables

- Special variables
  - `$@` - tên đầy đủ của target hiện tại
  - `$?` - kiểm tra danh sách các đối tượng tiên quyết mới hơn so với target hiện tại
  - `$*` - kiểm tra tên các target hiện tại với tiền tố đã được loại bỏ (thực tế, giá trị của % wildcard trong target)
  - `$<` - Tên của first dependency
  - `$^` - Tên của tất cả dependency ngăn cách bởi các khoảng trắng.

Using Shell Wildcards in Makefiles

- Giả sử bạn muốn chỉ ra rằng target *foo* được tạo ra (phụ thuộc) vào tất cả các object files trong thư mục hiện tại.

```
objects = *.o
foo : $(objects)
    cc -o foo $(CFLAGS) $(objects)
```

- Mỗi file '.o' trở thành một dependency của `foo` và sẽ được compile lại nếu cần thiết.

Using Shell Wildcards in Makefiles

- Giả sử bạn muốn chỉ ra rằng target *foo* được tạo ra (phụ thuộc) vào tất cả các object files trong thư mục hiện tại.

```
objects = *.o
foo : $(objects)
    CC -o foo $(CFLAGS) $(objects)
```

- Mỗi file '.o' trở thành một dependency của `foo` và sẽ được compile lại nếu cần thiết.
- Nhưng điều gì xảy ra nếu không có .o files?

○ Bởi vì wildcard không thể match với bất cứ file nào, nó sẽ để giống như cũ. ○
Vậy nên foo sẽ phụ thuộc vào một file name tên là '*.o'
○ Và bởi vì '*.o' không tồn tại, nên nó sẽ báo lỗi bởi vì nó không biết cách để build chương trình.

Functions

- Functions cho phép xử lý text bên trong một makefile
- Thường được sử dụng để:
  - tạo ra một danh sách các file để thực thi trên đó.
  - tạo ra các commands để sử dụng.
- Functions được khởi gọi bằng syntax:
  - `$(function_name arg1, arg2, … argN)`
- `function_name` phải là function đã được định nghĩa ○
  Kiểm tra Makefile documentation cho danh sách các functions. ○

Không thể định nghĩa một function mới.

Useful Functions

- **$(subst from,to,text)**
  - Đổi <from> bằng <to> bên trong <text>
- **$(patsubst pattern, replacement, text)**
  - Tìm các từ (được ngăn cách bằng khoảng trắng) trong <text> nếu match <pattern> và đổi chúng thành <replacement>
  - `$(patsubst %.c,%.o,foo.c bar.c)` → `foo.o bar.o`
- **$(filter pattern..., text)**
  - Xóa tất cả các từ (được ngăn cách bằng khoảng trắng) trong <text> mà không match với bất cứ pattern nào, và chỉ trả về những từ match.

- sources := foo.c bar.c baz.s ugh.h

```
foo: $(sources)

    cc $(filter %.c %.s,$(sources)) -o foo
```


Useful Functions

- **$(shell command)**
  - Thực thi shell command và trả kết quả về cho command gốc. ■

```
contents := $(shell cat foo)
```

- **$(wildcard pattern)**
  - Trả về danh sách các tên file match pattern.
- **$(dir names...)**

  $(notdir names...)
  - Extract và trả về các thư mục (hoặc filenames) từ danh sách các file names. ○

```
$ (dir dir1/foo.c dir2/bar.h) → dir1 dir2
```

```
$ (notdir dir1/foo.c dir2/bar.h) → foo.c bar.h
```


Useful Functions

- **$(addprefix prefix,names...)**
  - Thêm <prefix> vào đầu mỗi tên file
    - `$(addprefix src/,foo bar) → src/foo src/bar`
- **$(basename names...)**
  - Trả về tên file mà bỏ đi phần extension.
    - `$(basename src/foo.c bar.x new) → src/foo bar new`
- **$(addsuffix suffix,names...)**
  - Thêm phần suffix vào cuối mỗi file name.
    - `$(addsuffix .c,foo bar) → foo.c bar.c`

Action Modifiers

- Mỗi action có thể được thay đổi bằng thêm một trong những prefixes sau:
  - - (dash) bất kì error nào xuất hiện khi thực thi sẽ được bỏ qua
    - Mặc định, thực thi sẽ dừng khi một trong những action trả về kết quả (status) khác 0
    - Với - : giúp in ra một message với status code được trả về và báo rằng lỗi đã được bỏ qua.
  - Example: `-rm file` # delete the file but does not stop if it does not exist
  - @ (at) command sẽ không in ra standard output trước khi nó được thực thi.
  - Example: `@echo "hello"` # print hello, without printing the echo command

```
CC = gcc # compiler to use  CC_OPT = -O2 -g -Wall #
compilation parameters  INCLUDES = -I. # path to the .h
files  CFLAGS = ${CC_OPT}
          OBJS = func.o main.o # list of object file to build

all: ${OBJS}
    ${CC} ${CFLAGS} ${OBJS} -o main ${INCLUDES}

%.o: %.c
    ${CC} -c ${CFLAGS} $*.c ${INCLUDES}

clean:
    @echo "Cleaning..."
    rm -f *.o
    rm main
```

Phony Targets

- Một target không nhất thiết là một filename.
    - Nhưng nếu có một file bên trong thư mục hiện tại giống tên với target, make sẽ bị nhầm lẫn

- Một cách để giúp chỉ ra target đó không phải là một file name là khai báo đó là một phony target.

```
.PHONY:clean

clean:
    rm -f *.o

    rm main
```

Nested Makefiles

```
subsystem:
    cd subdir && $(MAKE)
```

- Hữu ích cho một hệ thống lớn chứa đựng nhiều directory, và mỗi directory sẽ có makefile riêng và bạn có thể thêm makefile để sử dụng trong subdirectory.
- Khi sử dụng $(MAKE) variable thay vì gọi make program.
  - Để đảm bảo make luôn được thực thi (cả khi user sử dụng makefile với -t và -n)

Useful stuff

- Nếu bạn muốn biết các commands có thể được thực thi để tạo ra file, sử dụng option -n
- Nếu bạn muốn đổi Makefile variable từ command line, chỉ định nó phía sau target, i.e.

```
make test.o CC=gcc CFLAGS=”-g -O0”
```