

Trường Đại học Bách Khoa Hà Nội
Viện Điện tử truyền thông



Báo cáo:
Những kiến thức cơ bản về Makefile

Sinh viên thực hiện:
Họ và Tên: Phạm Văn Lâm
MSSV: 20111746

Hà Nội, 7/2014

Mục lục

1. Tổng quan về Makefile.....	4
2. Biên dịch một chương trình viết bằng ngôn ngữ C++.....	4
3. Viết một makefile đơn giản.....	5
4. Những quy luật trong makefile.....	6
4.1. Explicit rules:.....	6
4.2. Wildcards (những ký tự đại diện).....	7
4.3. Phony Targets (đích đến giả).....	8
4.4. Empty Targets (đích đến trống).....	9
4.5. Variables (biến).....	9
4.6. Automatic Variables (các biến tự động).....	9
4.7. Tìm kiếm file với VPATH và vpath.....	10
4.8. Pattern rules.....	12
4.8.1. The Patterns(Các loại mẫu):.....	12
4.8.2. Static Pattern Rules (Các quy luật tĩnh).....	13
4.8.2. Suffix Rules (Các quy tắc hậu tố).....	13
4.9. The Implicit Rules Database (Cơ sở dữ liệu của những luật ngầm định).....	14
4.9.1. Làm việc với các luật ngầm định.....	14
4.9.2. Cấu trúc của các quy luật.....	14
4.10. Special targets (Những đích đến đặc biệt).....	15
4.11. Automatic Dependency Generation (Tự động sinh ra các thành phần phụ thuộc).....	16
4.12. Quản lý thư viện.....	17
4.12.1. Xây dựng và cập nhật thư viện.....	17
4.12.2. Sử dụng thư viện như là một danh sách những tập tin phụ thuộc.....	18
5. Variables (biến) and Macros.....	18
5.1. Tác dụng của việc sử dụng biến.....	18
5.2. Các loại biến.....	18
5.2.1. Biến đơn giản.....	18
5.2.2. Biến được mở rộng một cách đệ quy.....	19
5.2.3. Các kiểu gán khác.....	19
5.3. Macros.....	20
6. Hàm.....	21
6.1. Cú pháp của hàm.....	21
6.2. Những hàm thực hiện thao tác trên văn bản.....	21
6.2.1. Hàm \$(subst from,to,text).....	21
6.2.2. Hàm \$(patsubst pattern,replacement,text).....	21
6.2.3. Hàm \$(strip string).....	22
6.2.4. Hàm \$(findstring find,in).....	22
6.2.5. Hàm \$(filter pattern...,text).....	22
6.2.6. Hàm \$(filter-out pattern...,text).....	22
6.2.7. Hàm \$(sort list).....	23
6.2.8. Hàm \$(word n,text).....	23
6.2.9. Hàm \$(wordlist s,e,text).....	23
6.2.10. Hàm \$(words text).....	23

6.2.11. Hàm \$(firstword names...)	23
6.2.12. Hàm \$(lastword names...)	23
6.3. Những hàm thao tác với tên file	24
6.3.1. \$(dir names...)	24
6.3.2. \$(notdir names...)	24
6.3.3. \$(suffix names...)	24
6.3.4. \$(basename names...)	24
6.3.5. \$(addsuffix suffix,names...)	24
6.3.6. \$(addprefix prefix,names...)	24
6.3.7. \$(join list1,list2)	25
6.3.8. \$(wildcard pattern)	25
6.3.9. \$(realpath names...)	25
6.4. Hàm foreach	25
6.5. Hàm file	25
6.6. Hàm Call	25
6.7. Hàm Value	26
6.8. Hàm Origin	26
6.9. Hàm flavor	26
7. Quá trình biên dịch	27
8. Danh sách tham khảo	28

1. Tổng quan về Makefile

- + Makefile là một tiện ích giúp tự động xác định các thành phần của một chương trình lớn và đưa ra những lệnh để biên dịch chúng.
- + Để sử dụng makefile, đầu tiên ta phải tạo ra một file có tên là makefile hay Makefile, trong đó miêu tả mối quan hệ giữa các file trong một chương trình, và cung cấp lệnh cho việc cập nhật mỗi file.
- + Từ đó, mỗi lần thay đổi nội dung file nguồn, thì chỉ cần mở terminal và chạy **make** thì mọi sự thay đổi sẽ được cập nhật.
- + Tiện ích make sử dụng dữ liệu trong makefile và lần sửa chữa cuối cùng của file nguồn để quét xem thành phần nào của file sẽ được cập nhật lại.
- + Tiện ích make không giới hạn ngôn ngữ, nhưng ở đây ta chỉ đề cập tới việc sử dụng ngôn ngữ C++

2. Biên dịch một chương trình viết bằng ngôn ngữ C++

- + Để biên dịch một chương trình viết bằng C++, ta dùng trình biên dịch g++
- + Cài g++ một cách đơn giản bằng lệnh: **sudo apt-get install g++** và **sudo apt-get update**
- + Cú pháp sử dụng là:

g++ [option] ten_file_nguon

- + Các option là:

- **-o** : sinh ra tập tin output
- **-c** : sinh ra tập tin đối tượng
- **-I** : đặc tả thư mục chứa tập tin **include**
- **-l** : đặc tả tên thư viện
- **-L** : đặc tả đường dẫn tới thư viện.

+ Ví dụ: ta có một chương trình đơn giản là: main.cpp

```
#include <iostream>

using namespace std;

int main(){

    cout<<"Hello world!"

    return 0;

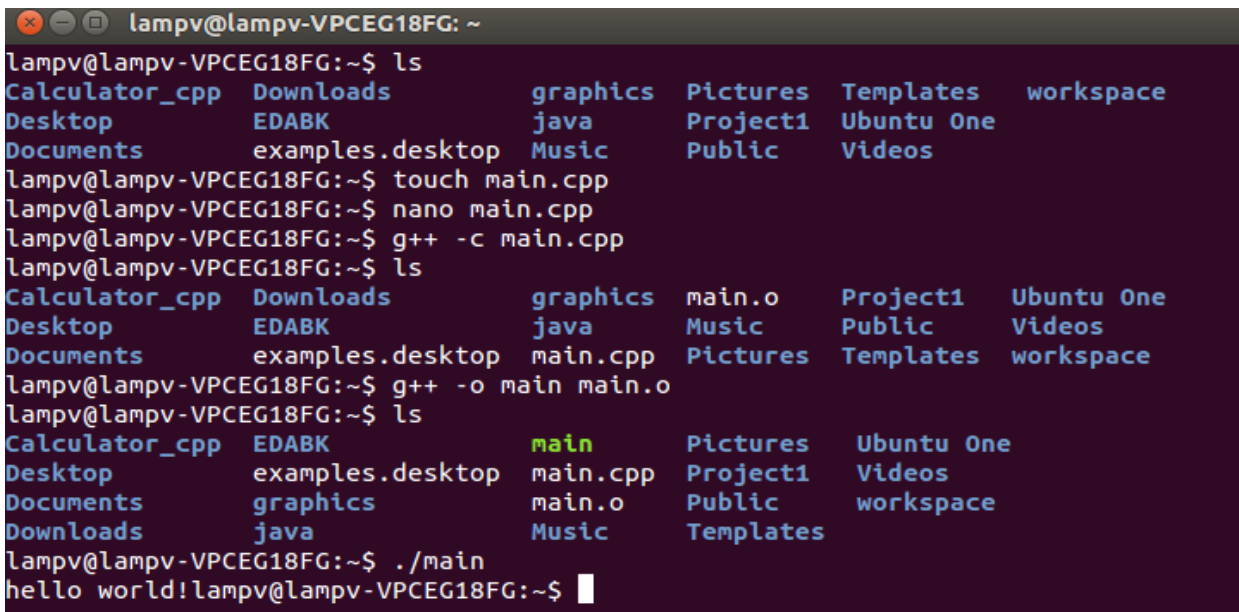
}
```

+ Để biên dịch và chạy chương trình ta làm như sau:

→ **g++ -c main.cpp**: sau lệnh này ta được file đối tượng là: **main.o**

→ **g++ -o main main.o** : sau lệnh này ta được file thực thi là : **main**

→ **./main** : chạy file run ta được kết quả là: "Hello world!"



```
lampv@lampv-VPCEG18FG: ~
lampv@lampv-VPCEG18FG:~$ ls
Calculator_cpp  Downloads      graphics  Pictures  Templates  workspace
Desktop        EDABK          java      Project1  Ubuntu One
Documents      examples.desktop  Music     Public    Videos
lampv@lampv-VPCEG18FG:~$ touch main.cpp
lampv@lampv-VPCEG18FG:~$ nano main.cpp
lampv@lampv-VPCEG18FG:~$ g++ -c main.cpp
lampv@lampv-VPCEG18FG:~$ ls
Calculator_cpp  Downloads      graphics  main.o    Project1  Ubuntu One
Desktop        EDABK          java      Music     Public    Videos
Documents      examples.desktop  main.cpp  Pictures  Templates  workspace
lampv@lampv-VPCEG18FG:~$ g++ -o main main.o
lampv@lampv-VPCEG18FG:~$ ls
Calculator_cpp  EDABK          main      Pictures  Ubuntu One
Desktop        examples.desktop  main.cpp  Project1  Videos
Documents      graphics        main.o   Public    workspace
Downloads      java            Music    Templates
lampv@lampv-VPCEG18FG:~$ ./main
hello world!lampv@lampv-VPCEG18FG:~$
```

3. Viết một makefile đơn giản

+ Đầu tiên ta tạo ra 1 file tên là makefile

+ Cú pháp trong makefile:

<target>: <danh sách các file phụ thuộc>

<TAB> lệnh

- + Chú ý: <TAB> chứ không phải dùng <Space>
- + Áp dụng với ví dụ ở phần 2: ta sẽ viết makefile như sau:

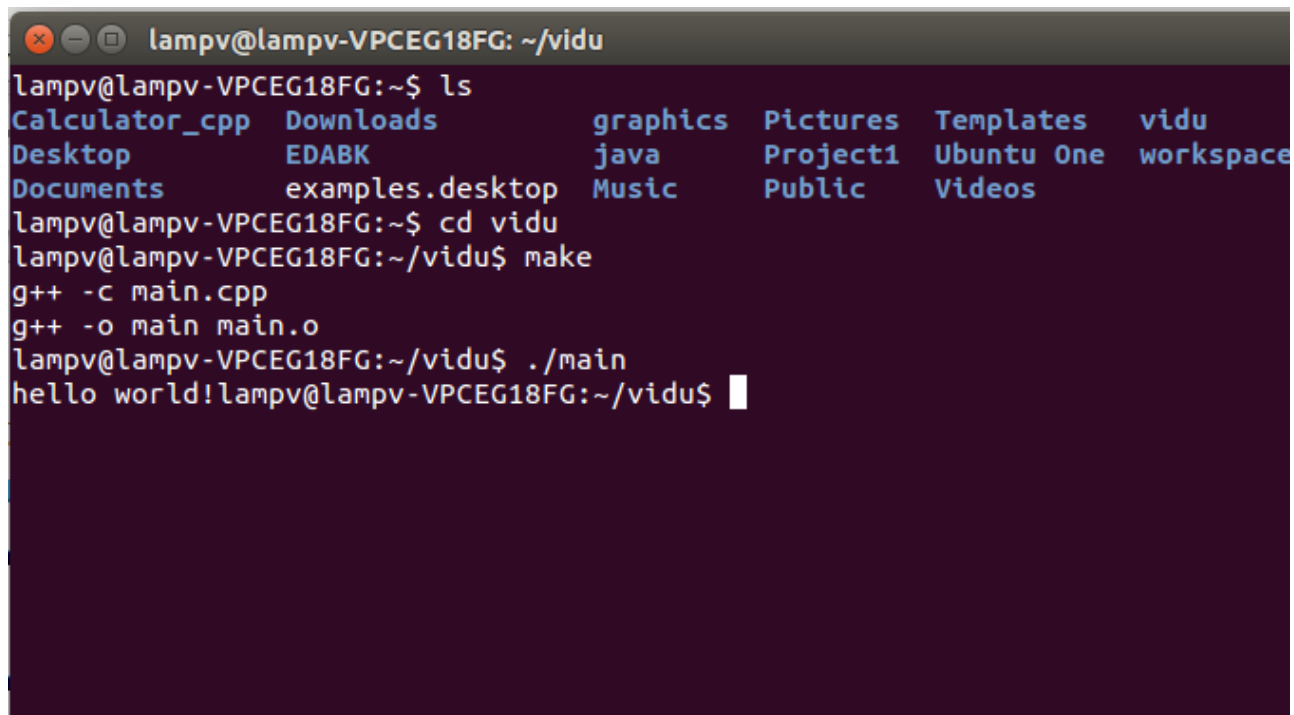
main: main.o

g++ -o main main.o

main.o: main.cpp

g++ -c main.cpp

- + Sau đó: chạy ./main ta được kết quả



```
lampv@lampv-VPCEG18FG: ~/vidu
lampv@lampv-VPCEG18FG:~$ ls
Calculator_cpp  Downloads      graphics  Pictures  Templates  vidu
Desktop        EDABK          java     Project1  Ubuntu One workspace
Documents      examples.desktop Music        Public    Videos
lampv@lampv-VPCEG18FG:~$ cd vidu
lampv@lampv-VPCEG18FG:~/vidu$ make
g++ -c main.cpp
g++ -o main main.o
lampv@lampv-VPCEG18FG:~/vidu$ ./main
hello world!lampv@lampv-VPCEG18FG:~/vidu$
```

4. Những quy luật trong makefile

4.1. Explicit rules:

- + Hầu hết những quy tắc mà ta sẽ viết đều phải qui định rõ ràng các thành phần: những cái đích đến (targets) và danh sách những thành phần phụ thuộc (prerequisites).
- + Một quy tắc có thể có nhiều hơn một đích đến, nghĩa là những đích đến đó đều có chung những thành phần phụ thuộc.
- + Nếu như những đích đến đó mà chưa được cập nhật thì sẽ có những lệnh giống nhau để

cập nhật cho mỗi đích đến đó.

+ Ví dụ:

target1.o target2.o: prerequisite1.h prerequisite2.h prerequisite3.cpp

→ tức là **target1.o** và **target2.o** đều phụ thuộc vào 3 tập tin là **prerequisite1.h prerequisite2.h và prerequisite3.cpp**

→ ta có thể viết lại là:

target1.o : prerequisite1.h prerequisite2.h prerequisite3.cpp

target2.o : prerequisite1.h prerequisite2.h prerequisite3.cpp

+ Mỗi 1 đích đến được quản lý một cách độc lập. Mỗi lần thấy một đích đến, make sẽ thêm đích đến đó và danh sách thành phần phụ thuộc vào đồ thị phụ thuộc của make.

+ Tuy nhiên, chỉ có một bộ danh sách các lệnh được liên kết với bất kỳ đích đến nào. Nếu chúng ta tạo ra 2 bộ danh sách lệnh cho cùng một thời điểm thì make sẽ cảnh báo và kết quả có được có thể sẽ sai so với những gì ta mong muốn.

* **Chú ý:** make hỗ trợ cách viết một quy tắc theo kiểu:

target : prerequisites ; command

4.2. Wildcards (những ký tự đại diện)

+ Có thể hiểu **wildcards** là các ký tự đại diện.

+ Trong makefile luôn tồn tại một danh sách rất dài các file. Vì vậy, để giúp cho make file đơn giản hơn thì make hỗ trợ việc sử dụng các ký tự đại diện.

+ Các ký tự đại diện như: ~, *, ?, [...], and [^...]

→ ~ : diễn tả đường dẫn của thư mục home

→ * : ví dụ: → *.* chỉ tất cả những file chứa dấu “.”

→ *.cpp chỉ tất cả những file có phần đuôi là “.cpp” như: node.cpp , graph.cpp , egde.cpp ...

→ [...] : diễn tả một lớp đối tượng nào đó

→ [^...]: diễn tả phủ định của lớp đối tượng trên.

+ Tuy nhiên, cần cẩn thận khi sử dụng các ký tự đại diện vì ta có thể hiểu sai ý nghĩa của chúng.

4.3. Phony Targets (đích đến giả)

+ Trong đa số các trường hợp thì đích đến (target) trong makefile là tên một tệp tồn tại thực tế.

+ Tuy nhiên, đích đến ở đây có thể chỉ đơn giản là một nhãn dùng để đại diện cho một tập lệnh.

+ Ví dụ: đích đến giả **clean**:

clean:

rm -rf *.o

+ Để thực hiện lệnh trên, chỉ cần gõ **make clean**.

+ Tuy nhiên, nếu chỉ viết như vậy ta sẽ không thể phân biệt được đây là đích đến giả hay là tên một tệp tin.

+ Đích đến giả luôn luôn không có danh sách thành phần tệp phụ thuộc, vì vậy nó luôn luôn bị xác định là chưa được thực thi. Để tránh hiện tượng này, tiện ích make cung cấp một đích đến đặc biệt là **.PHONY** để thông báo với make rằng đích đến ở đây không phải là một file thực sự.

+ Ví dụ:

.PHONY: clean

clean:

rm -rf *.o

+ Dưới đây là danh sách những đích đến giả tiêu chuẩn, hay được sử dụng trong makefile là:

→ all: thực hiện tất cả các nhiệm vụ để xây dựng ứng dụng.

→ install: cài đặt ứng dụng từ cây nguồn đã được biên dịch

→ clean: xoá các tệp tin được tạo ra từ các nguồn.

→ distclean: xoá tất cả các file mà nó không được tạo ra từ cây nguồn.

→ TAGS: tạo ra các bảng thẻ để sử dụng cho các biên tập viên.

→ info: tạo ra các tệp tin info GNU từ các nguồn Texinfo

→ check: chạy bất kỳ một bài kiểm tra nào liên quan tới ứng dụng.

4.4. Empty Targets (đích đến trống)

- + Đích đến trống gần giống với đích đến giả
- + Tuy nhiên, đích đến giả không có danh sách tập tin phụ thuộc. Còn đích đến trống thì có tồn tại danh sách tập tin phụ thuộc.
- + Đích đến trống nhằm thực hiện một số lệnh nhưng không muốn tạo ra file đầu ra.

4.5. Variables (biến)

- + Cú pháp đơn giản nhất để sử dụng một biến là:

`$(variable-name)`

4.6. Automatic Variables (các biến tự động)

- + Các biến tự động được cài đặt bởi make sau một quy luật dùng để truy cập tới các đích đến và danh sách các tập tin phụ thuộc.
- + Việc sử dụng các biến tự động giúp cho makefile gọn hơn, tránh lặp code
- + Một vài biến tự động là:
 - `$@` : tên của đích đến hiện tại.
 - `$<` : tên của tập tin phụ thuộc đầu tiên.
 - `$?` : tên của tất cả các tập tin phụ thuộc mới hơn so với đích đến, được ngăn cách bởi dấu cách.
 - `$^` : tên của tất cả các tập tin phụ thuộc được ngăn cách nhau bởi dấu cách, và trong danh sách này các tên bị trùng lặp đã bị loại bỏ.
 - `$+` : giống `$^` ở chỗ đây cũng là tên của tất cả các tập tin phụ thuộc được ngăn cách nhau bởi dấu cách, nhưng khác ở chỗ là trong danh sách này chấp nhận việc trùng lặp tên. Biến này được dùng trong các trường hợp cụ thể nơi mà các bản sao là có nghĩa.

4.7. Tìm kiếm file với VPATH và vpath

- + Từ trước tới giờ, khi ta viết makefile thì mọi file mà ta cần thì đều chứa trong cùng thư mục với makefile.
- + Tuy nhiên, thực tế thì để dễ quản lý thì ta các file thường ở các thư mục khác nhau.
- + Vì vậy, trong phần này ta sẽ đề cập tới vấn đề tìm kiếm file, tức là trong makefile ta sẽ phải chỉ rõ đường dẫn tới file để makefile có thể tìm thấy chúng.
- + Ví dụ: ta có sơ đồ file trong thư mục MyProject như sau:

~/src/MyProject

|>---- *Makefile*

|>---- *include*

|>---- *counter.h*

|>---- *lexer.h*

|>---- *src*

|>---- *count_words.c*

|>---- *counter.c*

|>---- *lexer.l*

- + Nội dung của makefile là:

count_words: count_words.o counter.o lexer.o -lfl

gcc \$^ -o \$@

count_words.o: count_words.c include/counter.h

gcc -c \$<

counter.o: counter.c include/counter.h include/lexer.h

gcc -c \$<

lexer.o: lexer.c include/lexer.h

gcc -c \$<

lexer.c: lexer.l

flex -t \$< > \$@

+ Khi chạy make: thì sẽ thông báo lỗi vì make không thể tìm thấy vị trí của file **count_words.c** vì file này không trong cùng một thư mục với Makefile.

+ Ta sửa như sau:

VPATH = src

+ Makefile thành:

count_words: count_words.o counter.o lexer.o -lfl

gcc \$^ -o \$@

count_words.o: count_words.c include/counter.h

gcc -c \$(VPATH)/\$<

counter.o: counter.c include/counter.h include/lexer.h

gcc -c \$(VPATH)/\$<

lexer.o: lexer.c include/lexer.h

gcc -c \$<

lexer.c: lexer.l

flex -t \$(VPATH)/\$< > \$@

+ Bây giờ, chạy make , ta vẫn bị thông báo lỗi, vì make không thể tìm được file thư viện.
Ta sửa bằng cách sau:

CPPFLAGS = -I include

// ta đặt tên thư viện để make biết.

+ Makefile bây giờ thành :

count_words: count_words.o counter.o lexer.o -lfl

gcc \$^ -o \$@

count_words.o: count_words.c include/counter.h

gcc \$(CPPFLAGS) -c \$(VPATH)/\$<

counter.o: counter.c include/counter.h include/lexer.h

gcc \$(CPPFLAGS) -c \$(VPATH)/\$<

lexer.o: lexer.c include/lexer.h

gcc \$(CPPFLAGS) -c \$<

lexer.c: lexer.l

flex -t \$(VPATH)/\$< > \$@

4.8. Pattern rules

+ Với những chương trình nhỏ thì ta có thể viết một cách tường minh các lệnh mà không có vấn đề gì cả.

+ Tuy nhiên, với những chương trình lớn, có tới hàng nghìn file thì việc viết cụ thể ra từng file một là không khả thi và làm cho makefile trở nên rất dài dòng và rắc rối. Khi đó, ta nên sử dụng những mẫu đã định sẵn.

+ Có 3 dạng đơn giản sau:

→ Biên dịch file .c thành file .o:

%o: %.c

\$(COMPILE.c) \$(OUTPUT_OPTION) \$<

→ Biên dịch file .l thành file .c:

%.c: %.l

@\$(RM) \$@

\$(LEX.l) \$< > \$@

→ Biên dịch file .c thành file không có phần hậu tố:

%.: %.c

\$(LINK.o) \$^ \$(LOADLIBES) \$(LDLIBS) -o \$@

4.8.1. The Patterns(Các loại mẫu):

+ Ký tự % là một quy luật mẫu, nó tương đương với ký tự * trong Shell của Unix. Nó đại diện cho bất kỳ một ký tự nào.

+ Ký tự % có thể đặt ở bất kỳ đâu trong mẫu nhưng nó chỉ được xuất hiện một lần:

Ví dụ: %,v hay s%.o .v.v.

+ Các ký tự khác trong mẫu là các ký tự còn lại trong tên tệp tin. Một mẫu có thể có tiền tố hoặc hậu tố hay cả hai.

+ Make sẽ tìm kiếm những tên tập tin có dạng giống như quy luật mẫu. Nếu tìm được thì quy luật sẽ được thực thi.

4.8.2. Static Pattern Rules (Các quy luật tĩnh)

+ Một quy luật tĩnh chỉ áp dụng cho một danh sách cụ thể những tập tin.

+ Ví dụ : ***\$(OBJECTS): %.o: %.c***

\$(CC) -c \$(CFLAGS) \$< -o \$@

+ Sự khác biệt ở đây là việc khởi tạo biến ***\$(OBJECTS)*** ,

+ Khi đó, ta đã giới hạn những file có dạng mẫu ***%.o*** chỉ là những file nằm trong ***\$(OBJECTS)***.

4.8.2. Suffix Rules (Các quy tắc hậu tố).

+ Các quy tắc hậu tố là một cách nguyên gốc để định nghĩa những quy tắc ngầm định.

+ Một quy tắc hậu tố có thể chứa một hoặc hai hậu tố liên kết với nhau và được dùng như một đích đến.

+ Ví dụ:

.c.o:

\$(COMPILE.c) \$(OUTPUT_OPTION) \$<

+ Trong ví dụ này thì hậu tố đầu tiên ***.c*** là của thành phần phụ thuộc, còn hậu tố thứ hai ***.o*** là của đích đến. Cách viết này là tương đương với cách viết sau:

%.o: %.c

\$(COMPILE.c) \$(OUTPUT_OPTION) \$<

+ Danh sách những hậu tố đã được make định nghĩa mà ta có thể sử dụng là:

.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l

+ Ngoài ra ta có thể định nghĩa thêm các hậu tố khác như sau:

.SUFFIXES: .pdf .fo .html .xml

+ Ta có thể xóa tất cả các hậu tố bằng cách :

.SUFFIXES:

4.9. The Implicit Rules Database (Cơ sở dữ liệu của những luật ngầm định)

- + Có khoảng 90 luật ngầm định được xây dựng trong make.
- + Mỗi một quy tắc ngầm định giống như một quy tắc mẫu, hay một quy tắc hậu tố.
- + Đa phần các ngôn ngữ lập trình như C,C++,Pascal đều hỗ trợ những luật ngầm định, nhưng nếu ngôn ngữ không hỗ trợ (như Java, XML) thì ta phải tự viết những quy luật ngầm định này.
- + Ví dụ một luật ngầm định như sau:

%.: %.C

commands to execute (built-in):

\$(LINK.C) \$^ \$(LOADLIBES) \$(LDLIBS) -o \$@

- + Trong luật trên, ta thấy đầu tiên là các biến, tiếp đến là chú thích cho việc sử dụng luật, dòng cuối là lệnh để thực thi.

4.9.1. Làm việc với các luật ngầm định

- + Những luật ngầm định được sử dụng ở bất kỳ nơi nào mà đích đến (target) được xem xét và không có một quy luật cụ thể nào để cập nhật chúng.
- + Vì vậy sẽ là rất dễ dàng khi ta muốn thêm một tệp đích đến vào makefile mà không cần xác định một tập lệnh nào.

4.9.2. Cấu trúc của các quy luật

- + Các quy luật được xây dựng nhằm mục đích để dễ dàng tùy biến sau này.
- + Sau đây là ví dụ về một quy luật để cập nhật một file đối tượng từ một file nguồn c:

%.o: %.c

\$(COMPILE.c) \$(OUTPUT_OPTION) \$<

- + Khả năng tùy biến của quy luật này là phụ thuộc vào 2 biến mà ta sử dụng ở trên.
- + Ở đây các biến được định nghĩa như sau:

COMPILE.c = \$(CC) \$(CFLAGS) \$(CPPFLAGS) \$(TARGET_ARCH) -c

CC = gcc

OUTPUT_OPTION = -o \$@

+ Khi ta muốn thay đổi quy luật trên thì đơn giản là ta chỉ cần thay đổi định nghĩa của các biến bên trên.

4.10. Special targets (Những đích đến đặc biệt)

+ Những đích đến đặc biệt tức nó không phải một tệp tin thực sự. Những đích đến đặc biệt mà ta đã đề cập tới ở những phần trước như: .PHONY , .SUFFIXES

+ Ngoài ra, còn một số đích đến đặc biệt khác:

*** .INTERMEDIATE:**

Danh sách những tệp tin phụ thuộc của đích đến này giống như là những tệp tin trung gian. Khi make tạo ra những file trong khi cập nhật những đích đến khác thì những file đó sẽ bị xoá khi make được thoát. Còn nếu những file đó đã tồn tại từ trước rồi thì những file đó sẽ không bị xoá. Điều này rất hữu ích khi xây dựng một loạt những quy tắc tùy chỉnh. Rất tiện ích cho việc xoá những file trung gian.

*** .SECONDARY**

Danh sách những tệp tin phụ thuộc của đích đến này cũng giống như những tệp tin trung gian nhưng nó không tự động bị xoá. Điều phổ biến nhất cho việc sử dụng .SECONDARY là để đánh dấu những tệp tin đối tượng được lưu trữ trong thư viện. Và thường nó sẽ bị xoá khi những file đối tượng này đã được lưu trữ trong một kho dự trữ nào đó. Khi đó sẽ là rất thuận tiện cho việc phát triển, cập nhật lưu trữ nhưng vẫn muốn giữ lại những file đối tượng trên.

*** .PRECIOUS**

Khi make bị lỗi trong quá trình thực thi, nó có thể xoá những tệp đích đến. Điều đó có nghĩa là make sẽ không để lại những tệp tin đối tượng bị lỗi. Tuy nhiên nếu ta muốn make không xoá những tệp tin đó đi dù cho nó có bị lỗi thì ta dùng đích đến đặc biệt này.

*** .DELETE_ON_ERROR**

Cái này là ngược lại so với .PRECIOUS, tức là mỗi khi có lỗi nào xảy ra thì make sẽ xoá ngay tệp tin đối tượng đó.

4.11. Automatic Dependency Generation (Tự động sinh ra các thành phần phụ thuộc)

- + Bình thường ta có thể thêm sự phụ thuộc giữa những file đối tượng với file nguồn vào makefile bằng tay.
- + Tuy nhiên, với những chương trình tương đối phức tạp thì việc này có thể gây ra lỗi.
- + Để khắc phục ta có thể dùng câu lệnh để máy tính tự tìm kiếm mối quan hệ giữa các tệp tin.
- + Ví dụ như:

```
$ echo "#include <stdio.h>" > stdio.c  
$ gcc -M stdio.c  
stdio.o: stdio.c /usr/include/stdc-predef.h /usr/include/stdio.h |  
/usr/include/features.h /usr/include/x86_64-linux-gnu/sys/cdefs.h |  
/usr/include/x86_64-linux-gnu/bits/wordsize.h |  
/usr/include/x86_64-linux-gnu/gnu/stubs.h |  
/usr/include/x86_64-linux-gnu/gnu/stubs-64.h |  
/usr/lib/gcc/x86_64-linux-gnu/4.8/include/stddef.h |  
/usr/include/x86_64-linux-gnu/bits/types.h |  
/usr/include/x86_64-linux-gnu/bits/typesizes.h /usr/include/libio.h |  
/usr/include/_G_config.h /usr/include/wchar.h |  
/usr/lib/gcc/x86_64-linux-gnu/4.8/include/stdarg.h |  
/usr/include/x86_64-linux-gnu/bits/stdio_lim.h |  
/usr/include/x86_64-linux-gnu/bits/sys_errlist.h
```



```
lampv@lampv-VPCEG18FG: ~
lampv@lampv-VPCEG18FG:~$ echo "#include<stdio.h>" > stdio.c
lampv@lampv-VPCEG18FG:~$ gcc -M stdio.c
stdio.o: stdio.c /usr/include/stdc-predef.h /usr/include/stdio.h \
/usr/include/features.h /usr/include/x86_64-linux-gnu/sys/cdefs.h \
/usr/include/x86_64-linux-gnu/bits/wordsize.h \
/usr/include/x86_64-linux-gnu/gnu/stubs.h \
/usr/include/x86_64-linux-gnu/gnu/stubs-64.h \
/usr/lib/gcc/x86_64-linux-gnu/4.8/include/stddef.h \
/usr/include/x86_64-linux-gnu/bits/types.h \
/usr/include/x86_64-linux-gnu/bits/typesizes.h /usr/include/libio.h \
/usr/include/_G_config.h /usr/include/wchar.h \
/usr/lib/gcc/x86_64-linux-gnu/4.8/include/stdarg.h \
/usr/include/x86_64-linux-gnu/bits/stdio_lim.h \
/usr/include/x86_64-linux-gnu/bits/sys_errlist.h
lampv@lampv-VPCEG18FG:~$ c
```

4.12. Quản lý thư viện

- + Thư viện lưu trữ, hay đơn giản gọi là một thư viện, chính là một loại file đặc biệt chứa những file khác gọi là thành viên.
- + Thư viện dùng để gộp lại những tập tin liên quan đến đối tượng sao cho ta dễ dàng quản lý hơn.
- + Thư viện được tạo ra và cập nhật bởi một chương trình tên là **ar**

4.12.1. Xây dựng và cập nhật thư viện

- + Trong makefile, file thư viện được chỉ rõ tên cũng tương tự như các file khác.
- + Ví dụ một luật đơn giản để xây dựng thư viện là:

libcounter.a: counter.o lexer.o

\$(AR) \$(ARFLAGS) \$@ \$^

- + Ở đây **AR = ar** và **ARFLAGS = rv**

- + Trong ví dụ này thì tất cả các thành viên của thư viện đều sẽ được cập nhật thậm chí là chúng không bị thay đổi. Như vậy sẽ gây lãng phí thời gian, vì vậy ta sẽ sửa lại đoạn code trên như sau:

libcounter.a: counter.o lexer.o

\$(AR) \$(ARFLAGS) \$@ \$?

+ Khi đó, make sẽ chỉ cập nhật những file đối tượng mà chúng mới được sửa đổi, như vậy sẽ giúp tiết kiệm thời gian hơn.

+ Ngoài cách viết trên, thì trong GNU make, các thành viên của thư viện có thể được viết như sau:

libgraphics.a(bitblt.o): bitblt.o

\$(AR) \$(ARFLAGS) \$@ \$<

4.12.2. Sử dụng thư viện như là một danh sách tập tin phụ thuộc

+ Khi một thư viện xuất hiện như là một danh sách tập tin phụ thuộc thì chúng được xem như là tên của những tập tin chuẩn khác.

5. Variables (biến) and Macros

5.1. Tác dụng của việc sử dụng biến

+ Việc sử dụng biến sẽ giúp cho makefile có thể thích ứng tốt với các môi trường cụ thể khác nhau.

5.2. Các loại biến

+ Có 2 loại biến là: những biến được mở rộng một cách đơn giản (hay gọi tắt là biến đơn giản) và những biến được mở rộng một cách đệ quy.

5.2.1. Biến đơn giản

+ Biến đơn giản được định nghĩa sử dụng toán tử :=

Ví dụ: **MAKE_DEPEND := \$(CC) -M**

+ Gọi là biến mở rộng đơn giản bởi vì phía bên phải của nó được mở rộng ngay lập tức sau khi đọc dòng.

+ Ở đây, ta có kết quả: **gcc -M**

+ Nếu biến CC ở trên chưa được định nghĩa thì ta có kết quả là : [SPACE] -M

+ Dù một biến không được định nghĩa thì cũng không có lỗi gì.

+ Điều này là hữu ích vì nhiều lệnh ngầm định thường chứa những biến không được định nghĩa. Khi đó sẽ rất dễ dàng để tùy chỉnh sau này.

5.2.2. Biến được mở rộng một cách đệ quy

- + Biến được mở rộng một cách đệ quy là biến được định nghĩa bằng toán tử =
- + Ví dụ: **MAKE_DEPEND = \$(CC) -M**
- + Gọi là biến mở rộng một cách đệ quy bởi vì thành phần bên phải chỉ được mở rộng khi biến đó được sử dụng.
- + Ví dụ:

MAKE_DEPEND = \$(CC) -M

...

Some time later

CC = gcc

5.2.3. Các kiểu gán khác

* Gán có điều kiện:

- + Việc gán có điều kiện sử dụng toán tử là : ?=
- + Ví dụ:

OUTPUT_DIR ?= \$(PROJECT_DIR)/out

- + Việc gán có điều kiện có đặc điểm là nó sẽ chỉ thực hiện việc gán khi mà biến đó chưa được gán giá trị từ trước.

* Gán có mở rộng:

- + Gán có mở rộng sử dụng toán tử là : +=
- + Ví dụ:

MYVAR = A

MYVAR += B

- + Kết quả là: MYVAR = A B (Có dấu cách ở giữa). Để tránh việc có dấu cách ở giữa thì ta có thể gán bằng cách dùng toán tử gán đơn giản:

MYVAR = A

MYVAR := \$(MYVAR)B

5.3. Macros

- + Việc sử dụng biến là hoàn toàn tiện ích để lưu trữ giá trị mà giá trị đó chỉ là một đoạn văn bản ngắn, thường là một dòng.
- + Nhưng khi ta cần lưu trữ nhiều hơn một dòng văn bản thì việc sử dụng biến là không còn phù hợp. Thay vào đó việc sử dụng Macros là một biện pháp thay thế hợp lý.
- + Macro giống như là một cách khác để định nghĩa biến. Một Macro có thể chứa nhiều dòng.
- + Macro dùng để diễn tả một chỉ thị nào đó.
- + Macro được xác định bởi tên macro và một dòng mới, kết thúc bởi **endif**.
- + Ví dụ định nghĩa một macro như sau:

```
define create-jar  
@echo Creating $@...  
$(RM) $(TMP_JAR_DIR)  
$(MKDIR) $(TMP_JAR_DIR)  
$(CP) -r $^ $(TMP_JAR_DIR)  
cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@.  
$(JAR) -ufm $@ $(MANIFEST)  
$(RM) $(TMP_JAR_DIR)  
endif
```

- + Và sử dụng macro như sau:

```
$(UI_JAR): $(UI_CLASSES)  
$(create-jar)
```

- + Chú ý ta sử dụng tiền tố **@** phía trước **echo** trong định nghĩa macro. Khi đó các dòng lệnh sau **echo** sẽ không được trình bày bởi make. Mà chỉ có kết quả được hiển thị.
- + Tuy nhiên, khi sử dụng tiền tố **@** trước macro như sau:

```
$(UI_JAR): $(UI_CLASSES)  
@$(create-jar)
```

+ Khi đó, toàn bộ phần thân của macro sẽ bị ẩn đi và như ví dụ trên ta chỉ thấy được:

\$ make

Creating ui.jar...

6.Hàm

+ GNU make hỗ trợ việc xây dựng hàm.

+ Hàm trong make gần giống với việc định nghĩa biến, nhưng trong hàm luôn bao gồm một hay nhiều tham số được ngăn cách bởi dấu “,”

6.1. Cú pháp của hàm

+ **$\$(function\ arguments)$** hay **$\${function\ arguments}$**

+ Ở đây:

→ function ở đây là tên của hàm

→ arguments là danh sách tên biến, biến ngăn cách với function bởi một hay nhiều dấu cách, hoặc dấu tab. Và nếu có nhiều biến thì chúng sẽ ngăn cách nhau bởi dấu phẩy

+ Chú ý trong danh sách arguments có thể có chứa một hàm khác

6.2. Những hàm thực hiện thao tác trên văn bản

6.2.1. Hàm **$\$(subst\ from,to,text)$**

+ Chức năng là thay thế **from** thành **to** trong **text**

+ ví dụ: **$\$(subst\ ee,EE,feet\ on\ the\ street)$**

→ kết quả sẽ là: “fEEt on the strEEt”

6.2.2.Hàm **$\$(patsubst\ pattern,replacement,text)$**

+ Chức năng: thay thế trong **text** thành phần có dạng mẫu **pattern** thành **replacement**

+ Ví dụ: **$\$(patsubst\ \%.c,\%.o,x.c.c\ bar.c)$**

+ Kết quả là : “x.c.o bar.o”

6.2.3. Hàm \$(strip string)

- + Chức năng: loại bỏ khoảng trắng ở đầu và cuối đoạn text và nhiều khoảng trắng giữa các ký tự bằng một khoảng trắng duy nhất
- + Ví dụ: \$(strip a b c)
- + Kết quả là : “a b c”

6.2.4. Hàm \$(findstring find,in)

- + Chức năng là: tìm thành phần **find** trong **in**
- + Ví dụ: \$(findstring a,a b c)
- + Kết quả: trả về là a
- + Còn nếu là \$(findstring a,b c) thì kết quả là khoảng trắng

6.2.5. Hàm \$(filter pattern...,text)

- + Chức năng: lọc lấy trong **text** tất cả những thành phần có mẫu pattern
- + Ví dụ:

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

```
foo: $(sources)
```

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

- + Kết quả: cc chỉ biên dịch những file có dạng .c và .s trong source, nghĩa là chỉ biên dịch file foo.c bar.c và baz.s

6.2.6. Hàm \$(filter-out pattern...,text)

- + Chức năng: loại bỏ tất cả trong **text** những file dạng **pattern**
- + Ví dụ:

```
objects=main1.o foo.o main2.o bar.o
```

```
mains=main1.o main2.o
```

```
$(filter-out $(mains),$(objects))
```

- + Kết quả: còn lại những file mà khác những file trong **main** đó là **foo.o** và **bar.o**

6.2.7. Hàm \$(sort list)

- + Chức năng: sắp xếp những từ trong danh sách theo thứ tự bảng chữ cái và bỏ đi từ trùng lặp
- + Ví dụ: **\$(sort foo bar lose bar)**
- + Kết quả là: **bar foo lose**

6.2.8. Hàm \$(word n,text)

- + Chức năng: trả về từ thứ n trong **text**
- + Ví dụ: **\$(word 2, foo bar baz)**
- + Kết quả: trả về **bar**

6.2.9. Hàm \$(wordlist s,e,text)

- + Chức năng: trả về danh sách những từ bắt đầu từ từ thứ s và kết thúc từ từ thứ e. nếu e lớn hơn s thì kết quả trả về rỗng
- + Ví dụ: **\$(wordlist 2, 3, foo bar baz)**
- + Kết quả: trả về **bar baz**

6.2.10. Hàm \$(words text)

- + Chức năng: trả về số lượng từ trong **text**
- + Ví dụ: **\$(words foo bar)**
- + Kết quả: trả về giá trị **2**

6.2.11. Hàm \$(firstword names...)

- + Chức năng: trả về từ đầu tiên trong danh sách **names**.
- + Ví dụ: **\$(firstword foo bar)**
- + Kết quả: trả về **foo**

6.2.12. Hàm \$(lastword names...)

- + Chức năng: trả về từ cuối cùng trong danh sách **names**.
- + Ví dụ: **\$(firstword foo bar)**
- + Kết quả: trả về **bar**

6.3. Những hàm thao tác với tên file

6.3.1. \$(dir names...)

- + Chức năng: lấy ra một phần đường dẫn của mỗi tên tệp trong names, nếu tên tệp không có đường dẫn thì kết quả là “./”
- + Ví dụ: **\$(dir src/foo.c hacks)**
- + Kết quả: trả về **src/ ./**

6.3.2.\$(notdir names...)

- + Chức năng: Lấy ra tất cả những thành phần đằng sau dấu gạch chéo, nếu sau dấu gạch chéo không có ký tự nào thì kết quả trả về rỗng, nếu tên không có dấu gạch chéo thì nó được giữ nguyên.
- + Ví dụ: **\$(notdir src/foo.c hacks)**
- + Kết quả trả về: **foo.c hacks**

6.3.3.\$(suffix names...)

- + Chức năng: kết quả trả về tất cả hậu tố của tên trong names
- + Ví dụ: **\$(suffix src/foo.c src-1.0/bar.c hacks)**
- + Kết quả : **.c .c**

6.3.4. \$(basename names...)

- + Chức năng: kết quả trả về thành phần của tên bỏ đi hậu tố
- + Ví dụ: **\$(suffix src/foo.c src-1.0/bar.c hacks)**
- + Kết quả là: **src/foo src-1.0/bar hacks**

6.3.5. \$(addsuffix suffix,names...)

- + Chức năng: thêm hậu tố vào tên trong **names**
- + Ví dụ: **\$(addsuffix .c,foo bar)**
- + Kết quả là: **foo.c bar.c**

6.3.6.\$(addprefix prefix,names...)

- + Chức năng: thêm tiền tố cho tên

+ Ví dụ: **\$(addprefix src/,foo bar)**

+ Kết quả là **src/foo src/bar**

6.3.7. **\$(join list1,list2)**

+ Chức năng: nối những từ trong **list1** với những từ trong **list2** theo kiểu **word by word**

+ Ví dụ: **\$(join a b,.c .o)**

+ Kết quả là: **a.c b.o**

6.3.8. **\$(wildcard pattern)**

+ Chức năng: trả về danh sách những tên file có dạng giống mô hình **pattern**

6.3.9. **\$(realpath names...)**

+ Chức năng: với mỗi tên trong **names** sẽ trả về một tên tuyệt đối, tức không chứa bất kỳ dấu “.” hay .. hay dấu / hoặc bất kỳ liên kết nào

6.4. Hàm foreach

+ Chức năng là lặp lại một vài đoạn text với những biến thể được kiểm soát.

+ Cú pháp của hàm là: **\$(foreach var,list,text)**

6.5. Hàm file

+ Chức năng là cho phép makefile viết text lên file

+ Cú pháp: **\$(file op filename,text)**

+ Toán tử **op** có thể thay thế bằng > hay >>

+ **text** là nội dung và sẽ ghi lên file có tên **filename**

6.6. Hàm Call

+ Chức năng là tạo ra các hàm với những tham số mới

+ Cú pháp: **\$(call variable,param,param,...)**

+ Tức ta có thể viết một biểu thức phức tạp biểu diễn giá trị của một biến, sau đó mở rộng ra với các giá trị biến khác nhau.

+ Ví dụ:

reverse = \$(2) \$(1)

foo = \$(call reverse,a,b)

+ Khi đó : foo = b a

6.7. Hàm Value

+ Đây là cách khác để sử dụng giá trị của một biến

+ Cú pháp: **\$(value variable)**

6.8. Hàm Origin

+ Chức năng dùng để tìm vị trí nơi mà biến được tạo giá trị

+ Cú pháp: **\$(origin variable)**

+ Các giá trị trả về là:

→ ‘undefined’ : nếu biến chưa được định nghĩa

→ ‘default’: nếu biến có giá trị mặc định

→ ‘environment’: nếu như biến được thừa hưởng từ môi trường cung cấp để thực hiện.

→ ‘environment override’: nếu biến được thừa hưởng từ môi trường cung cấp để thực hiện và như là kết quả của tùy chọn -e

→ ‘file’: nếu như biến được định nghĩa trong makefile

→ ‘command line’ : nếu như biến được định nghĩa trong command line

→ ‘override’: nếu như biến được định nghĩa với một chỉ thị được thiết lập trong makefile

→ ‘automatic’: nếu biến là biến tự động được định nghĩa để thực hiện các công thức của mỗi quy tắc.

6.9. Hàm flavor

+ Dùng để chỉ ra một số thông tin của biến, gần giống với hàm origin

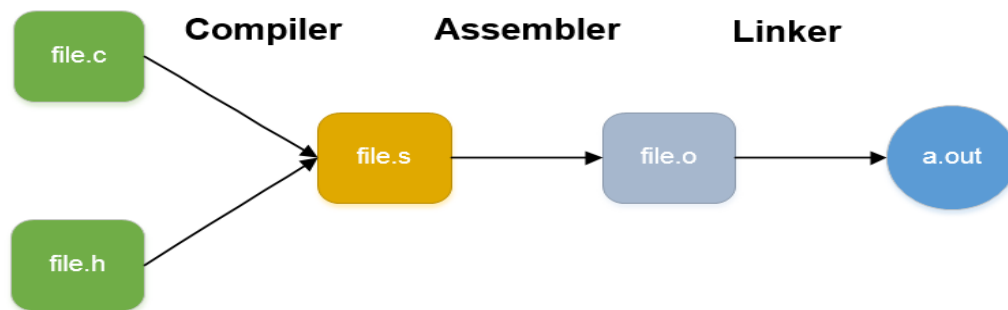
+ Cú pháp: **\$(flavor variable)**

+ Các kết quả trả về:

- undefined: nếu biến chưa được định nghĩa
- recursive : nếu biến được mở rộng một cách đệ quy
- simple : nếu là biến đơn giản.

7. Quá trình biên dịch

Quá trình biên dịch đơn giản bao gồm 3 bước:



* Bước 1: Compiler:

- + inputs: tất cả file source .c (cpp) và file header .h
- + outputs: file assembly .s

* Bước 2: Assembler:

- + inputs: file assembly .s
- + outputs: file objects .o

* Bước 3: Linker:

- + inputs: file objects .o
- + outputs: file output . Nếu không định nghĩa gì thì mặc định file output sẽ là file a.out.

8. Danh sách tham khảo

+ <http://www.gnu.org/software/make/manual/>

+ Managing Projects with GNU Make, Third Edition (3.13) - Robert Mecklenburg

<http://www.oreilly.com/catalog/make3/book/>

or <http://wanderinghorse.net/computing/make/>