C/C++ Programming in UNIX (UNIX-2)

Programming In UNIX

Compiling

- Use gcc

Project Making

- Use Makefile

Debugging

- Use gdb

Use gcc

gcc: GNU 컴파일러 모음 (GNU Compiler Collection)

gcc는 C컴파일러가 아닌 C컴파일러를 실행하는 것이다.

C언어에서 실행파일을 만드는 과정

1. 컴파일 (.c -> .o) : cc1이 해준다
2. 링크 (.0 -> 실행파일 a.out) : ld라는 링커가 해준다

컴파일 순서

(1) C Preprocessing
(2) C 언어 컴파일
(3) Assemble
(4) Linking

gcc는 이 컴파일러와 링커를 불러와서 실행파일을 만드는 것이다!

Use gcc

컴파일 순서

(1) C Preprocessing
(2) C 언어 컴파일
(3) Assemble
(4) Linking

gcc version 확인: gcc -v

gcc 사용하기

1. C파일을 하나 만든다.
2. [컴파일] gcc 파일명.c
3. 잘 컴파일이 되었다면 a.out이라는 이름을 가진 실행 파일이 자동으로 생긴다.
4. 그 후 결과를 확인하고 싶다면 ./a.out ## (.)현재 디렉토리에서 실행한다는 뜻, a.out은 유닉스에서 기본으로 정해 놓은 C컴파일러의 출력 결과 바이너리 파일이다.
5. 다른 이름의 실행 파일을 만들고 싶다면? gcc -o 실행파일명 소스파일명.c 또는 gcc 소스파일명.c -o 실행파일명

   ex) gcc -o sadmonday monday.c

   -o 다음에는 반드시 출력파일이름이 와야 한다.

```
gr120190197@cspro:~/comsil$ ls
201225314_1280.jpg  monday.c  test.txt
gr120190197@cspro:~/comsil$ gcc -o sadmonday monday.c
gr120190197@cspro:~/comsil$ ls
201225314_1280.jpg  monday.c  sadmonday  test.txt
gr120190197@cspro:~/comsil$ ./sadmonday
Monday is my favorite day :-(
gr120190197@cspro:~/comsil$
```

Use gcc

gcc 옵션: man gcc

```
GCC(1)                                GNU                                GCC(1

NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-Wpedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] [@file] infile...

       Only the most useful options are listed here; see below for the
       remainder.  g++ accepts mostly the same options as gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, compilation,
       assembly and linking.  The "overall options" allow you to stop this
       process at an intermediate stage.  For example, the -c option says not
       to run the linker.  Then the output consists of object files output by
       the assembler.
 Manual page gcc(1) line 1 (press h for help or q to quit)
```

-v 옵션 (컴파일 과정을 보여준다)

현재 사용하는 gcc 버전 및 소스에서 삽입한 stdio.h가 어느 디렉토리에 있는지 등의 정보를 볼 수 있다.

gcc 컴파일러 버전은 5.4.0이고 gcc가 stdio.h등의 헤더 파일에 접근하는 경로는 /usr/lib/gcc/x86 64-linux-gnu/5/include 등이다.

```
gr120190197@cspro:~/comsil$ gcc -v monday.c
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.4.0-6ubuntu1~16
.04.10' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages
=c,ada,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 --en
able-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-ge
ttext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --e
nable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-de
fault-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --ena
ble-libmpx --enable-plugin --with-system-zlib --disable-browser-plugin --enable-
java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-5-a
md64/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-5-am
d64 --with-jvm-jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-5-amd64 --with-arch-d
irectory=amd64 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc -
-enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-mul
tilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-checking=
release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-g
nu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10)
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/5/cc1 -quiet -v -imultiarch x86_64-linux-gnu mond
ay.c -quiet -dumpbase monday.c -mtune=generic -march=x86-64 -auxbase monday -ver
sion -fstack-protector-strong -Wformat -Wformat-security -o /tmp/ccv6zJY7.s
GNU C11 (Ubuntu 5.4.0-6ubuntu1~16.04.10) version 5.4.0 20160609 (x86_64-linux-gn
u)
        compiled by GNU C version 5.4.0 20160609, GMP version 6.1.0, MPFR versio
n 3.1.4, MPC version 1.0.3
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
ignoring nonexistent directory "/usr/local/include"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_
64-linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/5/include
```

Use gcc

-D 옵션 (외부에서 define을 정의하는 옵션)

-c 옵션 (컴파일만 하고싶을 때, 여러 개의 소스파일을 컴파일 할 때 중요): gcc -c monday.c

```
gr120190197@cspro:~/comsil$ gcc -c monday.c
gr120190197@cspro:~/comsil$ ls
201225314_1280.jpg  a.out  monday.c  monday.o  sadmonday  test.txt
gr120190197@cspro:~/comsil$
```

1. gcc -o main main.c fun1.c fun2.c
2. gcc -c main.c

   gcc -c fun1.c

   gcc -c fun2.c

   gcc -o main main.o fun1.o fun2.o

   2번이 더 귀찮아 보이지만 fun1의 소스가 변경되었을 때, 목적파일을 이용하면 수정된 파일만 재컴파일하고, 기존파일과 링크만 시킴으로서 내부적으로 비효율적인 일을 하지 않을 수 있다.

-I 옵션 (#include 문장에서 지정한 헤더파일이 들어있는 곳을 정하는 옵션)

#include <stdio.h>를 사용한 경우, 시스템 표준 디렉토리인 /usr/include를 기준으로 파일을 찾아서 포함시킨다.

#include “stdio.h”를 사용한 경우, 지금 컴파일러가 실행되고 있는 현재 디렉토리를 기준으로 헤더파일을 찾는다.

이 두 디렉토리가 아닌 경우엔 -I<디렉토리>라고 명시한다.

Use Makefile

만약 관리해야 하는 소스코드가 수십, 수백개라면?

```
gcc –c main.c
gcc –c fun1.c
gcc –c fun2.c
…
gcc –c fun100.c
gcc –o program main.o fun1.o fun2.o … fun100.o
```

현실적으로 어려움

Makefile을 사용

Use Makefile

Makefile의 기본적인 구조

target … : prerequisites …

recipe

…

1) target: 목표가 되는 파일의 이름. Executable, object files.. Etc)

가장 처음에 등장하는 **target: default goal(보통 executable file), goal == make**의 최종목표

2) prerequisites: target을 위해 필요한 준비물들

3) recipe: Make가 수행하는 작업들

prerequisite이 바뀜 –> recipe에 따라 target 파일 생성
(필요한 target만 update 가능)

Use Makefile

linux상에서 반복적으로 발생하는 컴파일을 쉽게 하기 위해서 사용하는 make 프로그램의 설정 파일이다.

Makefile을 통하여 library 및 컴파일 환경을 관리 할 수 있다.

파일명: Makefile(권장)

```
CC = gcc                                                    메크로 정의

target1 : dependency1 dependency2                           룰 1
    command1
    command2                                                명령

target2 : dependency3 dependency4 dependency5               룰 2
    command3
    command4                                                명령
```

Command는 tab 으로 뛰어야 한다.

Makefile의 매크로 정의란? Makefile에서 미리 정의된 환경 변수

Makefile은 목표(target), 의존관계(Dependency), 명령(command)로 이루어진다

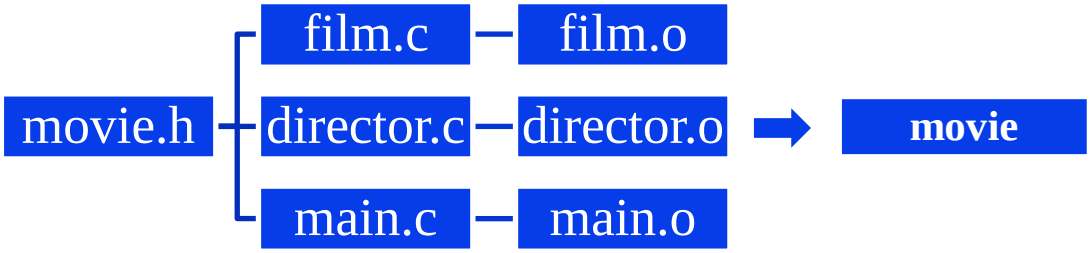
Use Makefile

Makefile과 일반적인 compile 과정의 차이점

각 파일에 대한 반복적 명령의 자동화로 인한 시간 절약

프로그램의 종속 구조를 빠르게 파악 할 수 있으며 관리가 용이

단순 반복 작업 및 재작성을 최소화




```
gr120190197@cspro:~/comsil/makeFileDir$ ls
director.c  film.c  main.c  movie.h
```

1. 일반적인 컴파일
   1.
```
gr120190197@cspro:~/comsil$ cd makeFileDir/
gr120190197@cspro:~/comsil/makeFileDir$ gcc -c -o director.o director.c
gr120190197@cspro:~/comsil/makeFileDir$ gcc -c -o film.o film.c
gr120190197@cspro:~/comsil/makeFileDir$ gcc -c -o main.o main.c
```
   2.
```
gr120190197@cspro:~/comsil/makeFileDir$ gcc -o exefile main.o director.o film.o
gr120190197@cspro:~/comsil/makeFileDir$ ls
director.c  director.o  exefile  film.c  film.o  main.c  main.o  movie.h
gr120190197@cspro:~/comsil/makeFileDir$ ./exefile
 i am a film!
i am a director!
gr120190197@cspro:~/comsil/makeFileDir$
```

Use Makefile

예제) Makefile

vi Makefile 후 밑의 내용을 입력

```
exefile2: film.o  director.o  main.o
    gcc -o  exefile2    film.o  director.o  main.o
film.o: film.c
    gcc -c  -o  film.o  film.c

director.o: director.c
    gcc -c  -o  director.o  director.c

main.o: main.c
    gcc -c  -o  main.o  main.c

clean:
    rm *.o exefile2

~
~
gr120190197@cspro:~/comsil/makeFileDir$ make
gcc -c  -o  film.o  film.c
gcc -c  -o  director.o  director.c
gcc -c  -o  main.o  main.c
gcc -o  exefile2        film.o  director.o      main.o
gr120190197@cspro:~/comsil/makeFileDir$ ls
director.c  exefile2  film.o  main.o    movie.h
director.o  film.c    main.c  Makefile
gr120190197@cspro:~/comsil/makeFileDir$ ./exefile2
 i am a film!
i am a director!
gr120190197@cspro:~/comsil/makeFileDir$ make clean
rm *.o exefile2
gr120190197@cspro:~/comsil/makeFileDir$ ls
director.c  film.c  main.c  Makefile  movie.h
gr120190197@cspro:~/comsil/makeFileDir$
```

make clean 명령으로 확장자가 .o인 파일과 exefile2가 지워진 것을 확인가능

Use Makefile

Variable을 활용하여 Makefile 간략하게 만들기

```
1  cc = gcc
2  target = movie
3  objects = main.o film.o director.o
4
5  $(target): $(objects)
6  →   $(cc) -o $(target) $(objects)
7
8  $(objects) : movie.h
9
10 .PHONY : clean
11 clean :
12 →   rm $(target) $(objects)
13
```

Use Makefile

1) cc = gcc

‘gcc를 이용하여 컴파일 하겠다’ 라는 뜻. 만약 c++ 코드를 컴파일 한다면 g++로 바꿔주면 된다.

```
1  cc = gcc
2  target = movie
3  objects = main.o film.o director.o
4
5  $(target): $(objects)
6  →   $(cc) -o $(target) $(objects)
7
8  $(objects) : movie.h
9
10 .PHONY : clean
11 clean :
12 →   rm $(target) $(objects)
13
```

2) target = movie

이 Makefile의 최종 목표가 되는 파일의 이름이다.

3) objects = …

Make에서 사용되는 object file들의 모음

prerequisite과 recipe를 보다 간편하게 작성 가능

Use Makefile

Question) 왜 main.o film.o director.o 를 위한 recipe는 없을까?

make Deduce the Recipes

각 c파일을 컴파일 하기 위해 모든 recipe를 작성할 필요는 없다.

```
1  cc = gcc
2  target = movie
3  objects = main.o film.o director.o
4
5  $(target): $(objects)
6  →   $(cc) -o $(target) $(objects)
7
8  $(objects) : movie.h
9
10 .PHONY : clean
11 clean :
12 →   rm $(target) $(objects)
13
```

.o 파일은 언급되었는데 그것을 위한 rule이 없다?

-> make가 implicit rule을 사용한다.

Implicit rule: .o 파일을 새로 생성하거나 업데이트 해야 한다면, 같은 이름의 .c 파일을 찾아 cc –c 명령어를 사용한다.

ex) film.o가 필요 -> film.o가 없다? -> film.c 발견 ->

'cc –c film.c –o film.o' 수행

cc(소문자)는 linux에서 gcc와 같은 의미

$(objects) : movie.h

Header 파일이 변경되어도 object 파일을 새로 생성해준다.

Use Makefile

.PHONY : clean

…

Phony target

실제 파일의 이름이 아니라, 단순히 recipe를 대표하는 이름이라고 생각하면 된다.

여기서의 recipe: rm $(target) $(objects)

현재 디렉토리에 있는 target 파일과 모든 object 파일들을 지운다.

- PHONY를 사용하는 이유

예상치 못한 상황을 방지하기 위하여..

Ex) 파일 중에 clean이라는 이름을 가진 것이 있다면..?

```
1  cc = gcc
2  target = movie
3  objects = main.o film.o director.o
4
5  $(target): $(objects)
6  →   $(cc) -o $(target) $(objects)
7
8  $(objects) : movie.h
9
10 .PHONY : clean
11 clean :
12 →   rm $(target) $(objects)
13
```

자주 나오는 에러

1. Makefile:17: *** missing separator. Stop.
*Makefile을 작성할 때 명령어(command)부분은 모두 TAB 문자로 시작해야 한다*고 첫 번째 장부터 강조하였다. 위의 에러는 *TAB* 문자를 쓰지 않았기 때문에 make가 명령어인지 아닌지를 구별 못하는 경우이다.
대처: 17번째 줄(근처)에서 명령어가 *TAB* 문자로 시작하게 바꾼다.
2. make: *** No rule to make target `io.h', needed by `read.o'. Stop.
위의 에러는 의존 관계에서 문제가 발생했기 때문이다. 즉 read.c가 io.h에 의존한다고 정의되어 있는데, io.h를 찾을 수 없다는 에러이다.
대처: 의존 관계에서 정의된 io.h가 실제로 존재하는지 조사해 본다. 없다면 그 이유를 한번 생각해 본다. *make dep*를 다시 실행시켜서 의존 관계를 다시 생성시켜 주는 것도 하나의 방법이다.
3. Makefile:10: *** commands commence before first target. Stop.
위의 에러는 '첫 번째 타겟이 나오기 전에 명령어가 시작되었다'는 애매한 에러 메시지이다. 필자가 경험한 이 에러의 원인은 주로 긴 문장을 여러 라인에 표시를 하기 위해서 '₩'를 사용할 때, 이를 잘못 사용했기 때문인 것 같다. 즉 '₩'부분은 라인의 가장 끝문자가 되어야 하는데 실수로 '₩'뒤에 스페이스를 몇 개 집어넣으면 여지없이 위의 에러가 발생한다.
대처: 10번째 줄(근처)에서 '₩'문자가 있거든 이 문자가 라인의 가장 끝문자가 되도록 한다. 즉 '₩'문자 다음에 나오는 글자(스페이스가 대부분) 는 모조리 없애 버린다.
4. make를 수행시키면 의도했던 실행 파일은 안생기고 이상한 행동만 한다. 가령 make clean 했을 때와 같은 행동을 보인다.
make는 천재가 아니라는 점을 생각해야 한다. *make는 Makefile의 내용을 읽다가 첫 번째 타겟으로 보이는 것을 자신이 생성시켜야 할 결과 파일이라고 생각한다.* 따라서 clean 부분을 Makefile의 첫번째 타겟으로 정해 버리면 위와 같은 결과가 나타나게 된다.
대처: 예제 7.1에서 all 이라는 필요 없는 타겟을 하나 만들어 두었다. 이것은 make가 all 을 첫 번째 타겟으로 인식시키기 위함이었다. 따라서 자신이 생성시키고 싶은 결과 파일을 첫 번째 타겟이 되게 하던지, 아니면 예제 7.1처럼 all과 같은 더미 타겟(dummy target)을 하나 만들어 둔다. 그리고 make clean, make dep 같은 부분은 Makefile의 끝부분에 만들어 두는 것이 안전하다.
5. 이미 컴파일했던 파일을 고치지 않았는데도 다시 컴파일한다.
이 행동은 make가 의존 관계를 모르기 때문이다. 즉 사용자가 의존 관계를 설정해 주지 않았다는 말이 된다. 따라서 make는 무조건 모든 파일을 컴파일해서 실행 파일을 만드는 일이 자신이 할 일이라고 생각하게 된다.
대처: 목적 파일, 소스 파일, 헤더 파일들의 의존 관계를 설정해 주어야 한다. *gccmakedep *.c 라고 하면 Makefile의 뒷부분에 자동적으로 의존 관계를 만들어 준다.* 그외의 다른 파일들에 대해서는 사용자가 적절하게 의존 관계를 설정해 주어야 한다.
main.o : main.c io.h read.o : read.c io.h write.o : write.c io.h

gcc 사용시 발생할 수 있는 에러 메세지들
https://sfixer.tistory.com/entry/GCC-Error-Message-List

Use gdb

- Compiling
  - Use gcc
- Project Making
  - Use Makefile
- Debugging
  - Use gdb

GDB의 목적

- GDB의 목적은 다른 프로그램 수행 중에 그 프로그램 `내부에서' 무슨 일이 일어나고 있는지 보여주거나 프로그램이 고장났을 때 무슨 일이 일어나고 있는지 보여주는 것이다.

- 버그를 잡는 걸 돕기 위한 GDB는 네 가지 종류의 일
  - 프로그램의 행동에 □ 향을 줄 수 있는 각종 조건을 설정한 후, 프로그램을 시작함.
  - 특정 조건을 만나면 프로그램을 정지시킴
  - 프로그램이 정지됐을 때 무슨 일이 일어났는지 검사함
  - 프로그램 내부 설정을 바꾸어서 버그를 수정함으로써 다른 버그를 계속 찾아나감

- 위의 특성을 가진 GDB는 프로그램 코드 상의 문제를 해결할 수 있는 강력한 무기이다.

GDB실행을 위한 compile 방법 및 실행

- Compile하기
  - Debug하기 전에 debug하고자 하는 program에 debugging 정보를 compile 한다. 그럼으로써 GDB가 사용했던 변수와 라인 및 함수를 실행할 수 있게 된다. gcc(또는 g++)에서 –g 옵션을 이용하여 program을 compile한다.
  - 얘 : gcc –g sample.c –o sample (실행파일 이름을 sample로 함)

- GDB 실행하기
  - GDB는 실행 program명을 매개변수로 하여 GDB명령으로 쉘 상에서 실행된다.
    - 예 : gdb sample
  - 매개변수를 주지 않고 GDB내에서 file 명령으로 debugging용 실행 program을 로드한 수 있다.
    - 예 : file sample
  - 만약 debuggging할 program의 실행에 있어서 argument를 줘야 한다면 GDB안에서 run시 주는 방법이 있다. (run은 program의 debugging의 시작을 의미함 – 단축키 : r)
    - 예 : run arg1 arg2 ...

List file

- List 명령어의 사용(단축키 : l)
  - 실행부분의 코드를 나열한다. 기본적으로 source의 시작부터 10줄의 code가 나열되며 다시 list를 사용하면 그 다음 10줄이 나열된다.
    - 예 : list
  - 자신이 보고자 하는 function의 이름과 같이 사용하면 그 function에 대한 source의 code가 10줄씩 출력된다.
    - 예 : list main
  - 자신이 보고자 하는 source의 line을 직접 입력해서 그 부분의 source code를 볼 수 있다.
    - 예 : list 10
  - 자신이 원하는 line사이의 code를 출력할 수 있다.
    - 예 : list 10 30 - (line 10에서 line30까지의 source의 내용이 출력)
  - 이 list를 사용하여 원하는 위치에 breakpoint를 잡을 수 있으며 debugging시 source의 문제를 찾는 데에도 도움을 준다.

Breakpoints

- Program내에서 무엇이 진행되는지 검토하기 위해 특정 행이나 program code 함수에 정지점을 설정하여 GDB가 여기에 이를 때 더 이상의 실행을 막는 일을 함.
- 사용법
  - 실행이 멈추길 원하는 function의 이름을 주어 break를 설정(단축키 : b)
    - 예 : break main
  - 멈추길 원하는 지점의 line numer를 적어줌으로써 break를 설정
    - 예 : break 10
- Conditional breakpoint
  - 어떤 조건이 만족할 때 정지시킬수 있다.
    - 예 : break 20 if value==10
  - 이미 설정된 breakpoint에 조건을 설정할 수 있다.
    - 예 : condition breakpoint_number value==10
  - 원하는 breakpoint의 조건을 제거할 수 있다.
    - 예 : condition breakpoint_number
- Brackpoint의 확인
  - 현재 설정되어 있는 breakpoint의 정보를 볼 수 있다.
    - 예 : info break

Breakpoints

- Breakpoint를 없애는 방법
  - 해당 번호의 breakpoint의 실행을 막는다
    - 예 : disable breakpoint_number(반대의 경우 enable하면 됨)
  - 해당 번호의 breakpoint를 제거한다.
    - 예 : delete breakpoint_number

- Breakpoint의 활용
  - Breakpoint를 한번만 실행하도록 하는 법
    - 예 : enable once breakpoint_number
  - Continue와 같이 사용함으로써 프로그램을 재개하여 다음 breakpoint나 signal까지 실행시킨다.
  - Command명령어를 사용하여 breakpoint를 만날 때마다 실행할 명령어를 setting할 수 있다.
    - 예 : command breakpoint_number
      >print value
      >....
      >end
  - Watchpoint라는 지정된 expression의 값이 변경될 때 프로그램이 정지한다.
    - 예 : watch value==10

Single step execution

- 프로그램의 실행을 한 줄씩 하면서 프로그램의 상태 및 변수의 변화를 확인하는데 사용된다.
  - Step : 한 라인을 실행한 뒤 중지하며 그 라인에 함수가 있을 경우 함수 내부로 들어가서 수행된다.
  - Next : 한 라인을 수행한 뒤에 중지한다.
  - Stepi : 정확하게 명령 하나를 수행한다. 즉 instruction수준의 명령어를 수행한다. Step과 똑같이 함수가 있을 때 함수의 내부로 들어간다.
  - Nexti : 정확하게 명령 하나를 수행한다.

- 함수에 관련된 수행
  - Call : 해당 함수를 호출하여 실행한다.
  - Finish : 현재 실행하고 있는 함수의 수행을 끝낸 뒤 return value가 존재하면 출력한다.
  - Return value : 현재 실행중인 함수의 수행을 중지한 뒤 주어진 value로 return시킨다.

Variables

- Debugging을 하기 위해서 현재 프로그램의 수행이 어떻게 되고 있는지 알기 위해 프로그램의 변수의 값을 관찰할 필요가 있다. 또한 그 변수에 값을 할당하여 어떻게 값이 바뀌는지 확인함으로써 debugging을 쉽게 수행할 수 있다.

- 변수 관찰의 명령어
  - Whatis variable : 현 variable의 type을 출력한다.
  - Print variable : 단 한번 variable의 값을 출력한다.
  - Display variable : 매번 명령이 수행될 때마다 variable의 값을 출력한다.
  - Ptype struct : struct멤버들의 정의를 출력한다.
  - Info locals : local 함수 내에 정의되어 있는 모든 변수의 값을 출력한다.

- 변수의 값 설정 명령어
  - Set variable value : variable에 value를 설정한다.
    - 예 : set variable var=10

2주차 실습 안내
(UNIX-2)

2주차 실습1

Makefile 간단히 만들기

variable을 사용하여 Makefile을 간단히 만들어본다

phony도 사용해본다

```
animal_exe : dog.o blackcow.o turtle.o main.o
	gcc -o animal.exe dog.o blackcow.o turtle.o main.o

dog.o : dog.c
	gcc -c -o dog.o dog.c

blackcow.o : blackcow.c
	gcc -c -o blackcow.o blackcow.c

turtle.o : turtle.c
	gcc -c -o turtle.o turtle.c

main.o : main.c
	gcc -c -o main.o main.c

clean :
	rm *.o animal.exe
```

2주차 실습2

gdb를 이용하여 코드의 문제를 찾고 이유를 설명해본다.

코드)

```
 1 #include <stdio.h>
 2
 3 main(void)
 4 {
 5     int i;
 6     double num;
 7
 8     for (i=0; i<5; i++){
 9         num=i/2 + i;
10         printf("num is %f \n", num);
11     }
12 }
```

breakpoint를 잡고 run해본다. breakpoint: b 포인트 잡을 줄 번호, run: r

한줄 한줄 실행해본다. 한 줄 실행: s

num값을 확인해본다. 변수에 들어있는 값을 확인하려면: display 변수명, num값 확인: p 변수명

2주차 실습3

프로그래밍 문제의 Remove_blanks_at_the_end 함수를 작성해본다.

2주차 실습 결과레포트

1. 실습 결과화면을 첨부한다.
2. fmt를 구현하기 위해 사용한 함수들과 그 함수들의 목적을 간단히 설명한다.
3. 실습시간에 작성한 Makefile의 한줄 한줄의 의미를 설명한다.
4. 규칙 R5를 어떤 알고리즘으로 구현하였는지 상세히 설명한다.
5. make의 옵션들에 대하여 정리한다.