7. Make Utility

Make utility란?

- make utility 이란?
  - 프로그램을 작성하고 컴파일 할 때 컴파일의 일련의 과정을 하나의 파일로 만들 수 있다.
  - 특히 수많은 **object file**과 실행 프로그램을 가지고 있는 시스템이 있다고 하면 **header, source code file, object,** 실행파일의 모든 관계를 기억하고 명령어 프롬프트 상에서 입력하는 것은 쉬운 일이 아닐 것이다. 이러한 문제의 해결에 도움을 주는 것이 **make utility**이다.
  - 하나의 파일만 변경하게 되더라도 다시 컴파일 해야 하는 등의 수정하고 컴파일하고 테트스하는데 많은 시간을 소모하게 될 것이다. 이것 같은 문제를 해결방법은 Make Utility 이용이다.
  - make 유틸리티는 많은 기능을 제공하지만, 자동으로 입력 파일을 인식하고, 컴파일하여 출력 파일을 생성하지는 못하기 때문에 이 것에 대한 정보를 make가 알 수 있도록 파일을 제공해야 하는데 이것이 Makefile이라는 파일이다.

Makefile에 대하여

Makefile에 대하여

- 메이크 파일은 기본적으로 대상(Target), 의존 관계(dependency), 명령(command)등에 대한 규칙들(rules)로 이루어져 있다. 아래의 표는 기본적인 메이크 파일의 플랫폼이다.

```
Target . . . : Dependency . . .
<-tab ->    Command  . . .
                 . . .
```

- Target : Command에 의해서 수행 되어서 나온 결과 파일을 지정하므로, 일반적으로 목적 파일(Object file)이나 실행 파일이다.

- Dependency : 생성되는 파일인 대상(Target)과 이것이 의존하는 파일들의 관계를 나타낸다.

- Command : Command 부분에 정의된 명령들은 의존 관계(Dependency)부분에 정의된 파일의 내용이 바뀌었거나, 대상(Target) 부분에 해당하는 파일이 없을 때 이곳에 정의된 것들이 차례대로 실행된다. 일반적으로 쉘에서 쓸 수 있는 모든 명령들을 사용할 수 있으며 Bash에 기반한 쉘 스크립트도 지원한다.

실행파일 만들기

- Makefile 만들기
  - 프로그램은 main.c, read.c, write.c로 구성되어 있고 모두 io.h을 사용하고 read.c는 defs.h을 더 포함하고, wirte.c는 buffer.h라는 라는 헤더 파일을 사용한다고 가정한다.

```
$gcc -c main.c

$gcc -c read.c

$gcc -c write.c

$gcc -o test main.o read.o write.o
```

쉘 상에서 실행 파일 만들기

Makefile 만들기

❖ Makefile 만들기

```
test: main.o read.o write.o
	gcc -o test main.o read.o write.o

main.o : io.h main.c
	gcc -c main.c

read.o : io.h defs.h read.c
	gcc -c read.c

write.o : io.h buffer.h write.c
	gcc -c write.c
```

```
$make
gcc -c main.c
gcc -c main.c
gcc -o test main.o read.o write.o
```

Makefile을 작성하여 실행 파일 만들기와 실행 화면

Makefile 만들기

- 설명

  - 쉘 상에서 make명령을 내리면 먼저 make는 먼저 현재 디렉토리에서 makefile이라는 파일을 찾는다. 이 파일이 존재하지 않으면 다시 Makefile이라는 파일을 찾는다.전통적인 유닉스 프로그래머는 makefile을 사용한다.

  - make는 Makefile의 내용을 보고, 내부적으로 어떻게 파일들이 의존하고 있는지 조사한다.

  - 위의 예제에서는 sample라는 실행 파일을 만들기 위해서는 main.o는 io.h, read.o는 io.h와 defs.h, 그리고 write.o는 io.h와 buffer.h의 헤더 파일과 모두 자신의 소스에 의존 관계임을 알 수 있다. 만약에 main.c를 고친다면 main.o가 컴파일되어 다시 생기고, sample도 다시 링크되어 갱신된다. 만약 io.h가 바뀌었다고 가정하면 모든 파일들이 컴파일되어서 목적 파일이 생기고, 그것들이 링크가 되어 sample가 생긴다

Makefile 만들기

❖ 변수를 사용하여 더 간단하게 만들기

```
# Variables make Makefiles simpler
Objects = main.o read.o write.o

test: $(Objects)
	gcc -o test $(Objects)

main.o : io.h main.c
	gcc -c main.c

read.o : io.h defs.h read.c
	gcc -c read.c

write.o : io.h buffer.h write.c
	gcc -c write.c
```

변수이용 Makefile만들기

Makefile 만들기

- 설명
  - Makefile에서의 주석문은 #으로 시작하고, 한 줄의 끝까지 계속된다.
  - Makefile에서는 변수를 사용하기 위해서는 값을 대입하기 위해서는 “=”을 사용하고 $(변수), ${변수}의 형식을 사용하여 값을 쓸 수 있게 된다.
  - Makefile에서 사용되는 모든 규칙은 탭으로 시작하는 줄에 있어야 한다. 빈 칸은 허용되지 않는다. 다수의 빈 칸과 탭은 같은 것으로 보이고, 유닉스 프로그래밍의 다른 모든 경우에 빈 칸과 탭 사이에는 거의 차이가 없으므로 이것은 문제를 일으킬 수 있다. 또한, Makefile에서 줄의 마지막에 있는 빈 칸은 make명령이 실패하게 할 수 있다.

Makefile 만들기

- 명령들을 추론하여 만들기
  - make명령은 C 소스인 경우 .o을 이용하여 .c를 추론하여 명령을 내부 규칙에 따라 gcc -c을 수행하게 된다.

```
 #Letting make deduce the commands
OBJECTS = main.o read.o write.o

test: $( OBJECTS)
	gcc -o test $( OBJECTS)

  main.o : io.h

  read.o : io.h defs.h

write.o : io.h buffer.h

.PHONY : clean
clean:
	-rm –f $( OBJECTS)
	@rm –f test
```

명령을 추론하여 만든 Makefile

Makefile 만들기

- 설명
  - 위의 clean처럼 대상(Target) 부분에 해당하는 부분이 그냥 레이블과 같이 사용될 수 있다. 이것을 Phony Target이라고 하면 위의 형식처럼 사용하는데 .PHONY는 생략하여도 된다.
  - 위의 예제에서 rm명령 앞에 있는"-"는 make가 에러를 무시하게 한다. 예를 들어, 디렉토리를 만들기 원하고 디렉토리가 이미 존재할 경우에 에러를 무시하기 원한다면, mkdir 앞에 "-"를 추가하면 된다
  - 위의 예제에서 rm명령 앞에 있는 @는 make가 명령을 실행하는 것을 표준 출력 장치로 출력하지 않도록 지시한다.

Makefile 만들기

- 또 다른 형태의 Makefile 만들기
  - 내부 규칙에 따라 Target대신에 의존성(dependency)만을 사용하여 만든다.

```
# Alternative sytle of makefile

OBJECTS = main.o read.o write.o

test: $( OBJECTS)
      gcc -o test $( OBJECTS)

  $(Objects) : io.h

  read.o : defs.h

  write.o : buffer.h
```

의존성을 이용하여 Makefile만들기

Makefile 만들기(매크로이용)

- **Makefile의 기능을 향상시키 주는 매크로 와 확장자 규칙**
  - 매크로는 특정한 코드를 간단하게 표현하기 위해 사용되며,프로그램을 작성할 때 변수를 지정하는 것처럼 하면된다. 매크로를 Makefile에서 사용하기 위해서는 $(..)나 ${..}, $를 사용하는데, 일반적으로 $(..)을 사용하기를 권고한다

```
# Macro makes Makefile happy.

OBJECTS = main.o read.o write.o

test: $( OBJECTS)
	gcc -o test $( OBJECTS)

 $(Objects) : io.h

 read.o : defs.h

 write.o : buffer.h
```

간단한 매크로 예제

Makefile 만들기(매크로이용)

- 미리 정해져 있는 매크로(Macro)
  - 프로그래머가 매크로를 만들어 사용할 수 있지만, 미리 정의된 매크로들이 있다. 이 매크로들을 보기 위해서는 'make –p'라고 입력해 보면 make에서 미리 정의 되어 있는 모든 값들(매크로, 환경 변수등등)을 볼 수 있다

```
ASFLAGS = (as 명령어의 옵션 세팅)
AS = gas
CFLAGS = (gcc 의 옵션 세팅)
CC = gcc
CPPFLAGS = (g++ 의 옵션)
CPP = g++
LDFAGS = (ld 의 옵션 세팅)
LD = ld
```

- 매크로는 관습적으로 대문자를 작성된다.
- 쉘 상에서 정의한 환경 변수 값들은 그대로 이용된다.

Makefile 만들기(매크로이용)

❖ 미리 정의된 매크로를 사용한 예제 와 실행 화면

```
OBJECTS = main.o read.o write.o
SRCS = main.c read.c write.c

CC = gcc
CFLAGS = -g -c

TARGET = test

$(TARGET) : $(OBJECTS)
$(CC) -o $(TARGET) $(OBJECTS)

clean :
rm -f $(OBJECTS) $(TARGET)

main.o : io.h main.c
read.o : io.h read.c
write.o: io.h write.c

$make
gcc -g -c main.c -o main.o
gcc -g -c read.c -o read.o
gcc -g -c write.c -o write.o
gcc -o test main.o read.o write.o

$make clean
rm -rf main.o read.o write.o test
```

Makefile 만들기(Suffix)

- 확장자 규칙(Suffix rule)란
  - 파일의 확장자를 보고, 그에 따라 적절한 연산을 수행시키는 규칙이다. 가령 가령 .c 파일은 일반적으로 C 소스 코드를 가리키며, .o 파일은 목적 파일(Object file)을 말하고 있다. 그리고 당연히 .c 파일은 컴파일 되어서 .o 파일이 되어야 하는 것이다. .SUFFIXS라는 매크로를 사용하여 확장자를 등록하여 줄 수 있다

```
.SUFFIXES : .c .o

OBJECTS = main.o read.o write.o
SRCS = main.c read.c write.c

CC = gcc
CFLAGS = -g –c

TARGET = test

$(TARGET) : $(OBJECTS)
	$(CC) –o $(TARGET) $(OBJECTS)

clean:
	rm –f $(OBJECTS) $(TARGET)

main.o : io.h main.c
read.o : io.h defs.h read.c
write.o : io.h buffer.h write.c


$make
gcc -g -c main.c -o main.o
gcc -g -c read.c -o read.o
gcc -g -c write.c -o write.o
gcc -o test main.o read.o write.o
```

Makefile 만들기(Suffix)

- 확장자 규칙에 의해서 make는 파일들 간의 확장자를 자동으로 인식해서 필요한 작업을 자동으로 수행한다. 위의 예제에서는 자동적으로 아래와 같은 루틴이 자동적으로 동작하게 된다.

```
.c .o :
$(CC) $(CFLAGS) -c $< -o $@
```

- make 내부에서 기본적으로 서비스를 제공해 주는 확장자는 다음과 같다.

.out .a .ln .o .c .cc .C .p .f .F .r .y .l .s .S .mod .sym .def .h .info .dvi .tex .texinfo .texi .txinfo

- Makefile내부에서 .SUFFIXES 매크로의 값을 세팅해 주면 내부적으로 정의된 확장자의 연산이 동작을 하게 된다. 따라서 확장자 규칙은 make가 어느 확장자를 가진 파일들을 처리할 것인가를 정해 주는 것이다.

Makefile 만들기(내부매크로)

- 내부 매크로(Internal macro)
  - make에서는 내부 매크로라는 것이 있다. 이것은 우리가 맘대로 정할 수 있는 매크로는 절대 아니다. 대신 매크로를 연산, 처리하는데 쓰이는 매크로라고 하는 것이 더 적당할 것이다.

내부 매크로	설 명
$*	확장자가 없는 현재의 대상(Target) 파일
$@	현재의 대상(Target) 파일
$<	현재의 대상(Target) 파일보다 더 최근에 갱신된 파일 이름 (dependency중에서 처음에 위치한 파일 이름)
$?	현재의 대상(Target) 파일보다 더 최근에 갱신된 파일 이름 (모든 dependency 파일의 이름)

Makefile 만들기(내부매크로)

- 내부 매크로(Internal macro)를 사용한 예

```
main.o : main.c io.h
    gcc -c $*.c
    ($* 는 확장자가 없는 현재의 대상 파일이므로 $* 는 결국 main에 해당된다.)

test : $(OBJECTS)
    gcc –o $@ $*.c ($@ 는 현재의 대상 파일이므로 $* 는 결국 test에 해당된다.)


.c .o :
    gcc -c $< (또는 gcc -c $*.c)
($< 는 현재의 대상 파일보다 더 최근에 갱신된 파일 이름이므로, .o 파일보다 더 최근
에 갱신된 .c 파일은 자동적으로 컴파일이 된다. 가령 main.o를 만들고 난 다음에
main.c를 갱신하게 되면 main.c는 $<의 작용에 의해 새롭게 컴파일 된다.)
```

Makefile 활용(팁)

- Makefile를 작성할 때 알면 좋은 것들
  - 긴 명령어를 여러 라인으로 표시하기

    Makefile을 작성할 때 명령어가 한 줄을 넘어간다면, '\'문자를 이용해서 여러 라인으로 나타낼 수 있다.

```
OBJECTS = shape.o \
          rectangle.o \
          circle.o \
          line.o \
          main.o \
          read.o \
          write.o \
```

Makefile 활용(팁)

Makefile를 작성할 때 알면 좋은 것들

매크로 치환(Macro substitution)

매크로를 지정하고, 그것을 이용하는 것을 이미 알고 있다. 그런데 필용에 의해 매크로의 내용을 조그만 바꾸어야 할 때가 있다. 매크로 내용의 일부만 바꾸기 위해서는 $(MACRO_NAME:OLD=NEW)과 같은 형식을 이용하면 된다.

```
MY_NAME = Hello World
YOUR_NAME = $(MY_NAME:Hello=Hi)

(Jack이란 부분이 Jook으로 바뀌게 된다. 즉 YOUR_NAME이란 매크로의 값은 Hi World이 된다.)


OBJS = main.o read.o write.o
SRCS = $(OBJS:.o=.c)

(SRCS에서는 OBJS에서 .o가 .c로 바뀌게 된다. 다음과 같이 변경되는 것이다.
SRCS = main.c read.c write.c)
```

Makefile 활용(팁)

- Makefile를 작성할 때 알면 좋은 것들
  - 자동 의존 관계 생성(Automatic dependency)

make 구조는 대상(target), 의존 관계(dependency), 명령(command)이 연속적으로 정의 되어 있는데 실행되는데, 명령이 없이 대상과 의존 관계만 표시가 되면 이는 대상이 어느 파일에 의존하고 있는지 표시해 주는 정보 역할을 하게 됩니다. Makefile을 작성할 때 없어서는 안되는 부분입니다. 그런데 일일이 이런 정보를 만든다는 것은 쉬운 일이 아닙니다. 이런 단조롭고 귀찮은 일을 자동으로 해주기 위해서 gcc -M XX.c의 형식을 사용하여 의존 관계를 만들게 된다. 프로그램을 설치할 때 make dep라는 것을 친 기억이 있을 것이다. 파일들의 의존 관계를 작성해 준다는 의미이다.

Makefile 활용(팁)

❖ 자동 의존 관계 생성 예제 및 실행화면

```
.SUFFIXES : .c .o
CFLAGS = -O2 -g

OBJS = main.o read.o write.o
SRCS = $(OBJS:.o=.c)

test : $(OBJS)
        $(CC) -o test $(OBJS)

dep :
$(CC) –M $(SRCS)
```

```
$make dep
$vi Makefile
```

```
main.o: main.c /usr/include/stdio.h /usr/include/features.h  \
/usr/include/sys/cdefs.h /usr/include/libio.h  \
/usr/include/_G_config.h io.h
read.o: read.c io.h defs.h
write.o: write.c io.h buffer.h
```

(Makefile의 뒷부분에 위에 내용이 첨가 된다.)

Makefile 활용(팁)

- Makefile를 작성할 때 알면 좋은 것들
  - 다중 타켓(Multiple Target)

    하나의 Makefile에서 꼭 하나의 결과만 만들어 내라는 법은 없다.
    결과 파일이 3개가 필요한 경우도 있다.

```
.SUFFIXES : .c .o
CC = gcc
CFLAGS = -O2 -g

OBJS1 = main.o test1.o
OBJS2 = main.o test2.o
OBJS3 = main.o test3.o
SRCS = $(OBJS1:.o=.c) $(OBJS2:.o=.c) $(OBJS3:.o=.c)

all : test1 test2 test3

test1 : $(OBJS1)
        $(CC) -o test1 $(OBJS1)
```

Makefile 활용(팁)

```
test1 : $(OBJS2)
        $(CC) -o test2 $(OBJS2)

test1 : $(OBJS3)
        $(CC) -o test3 $(OBJS3)

dep :
        $(CC) -M $(SRCS)
```

$make all (또는 make)
gcc -O2 -g -c main.c -o main.o
gcc -O2 -g -c test1.c -o test1.o
gcc -o test1 main.o test1.o (test1 의 생성
gcc -O2 -g -c test2.c -o test2.o
gcc -o test2 main.o test2.o (test2 의 생성)
gcc -O2 -g -c test3.c -o test3.o
gcc -o test3 main.o test3.o (test3 의 생성)

Makefile 활용(팁)

- Makefile를 작성할 때 알면 좋은 것들
  - 순환 make(Recursive make)

규모가 큰 프로그램들은 파일들이 하나의 디렉토리에 있지 않는 경우가 많다. 여러 개의 서브시스템이 전체 시스템을 구성한다고 가정하면 각 서브시스템에 Makefile이 존재한다. (서브시스템 = 서브디렉토리) 따라서 여러 개의 Makefile을 동작시킬 필요가 있도록 Makefile을 고쳐 보자. 서브디렉토리에 있는 Makefile을 동작시키는 방법은 의외로 간단하다.

```
subsystem:
        cd subdir; $(MAKE) .....(1)

subsystem:
        $(MAKE) -C subdir .....(2)
```

(위의 예제에서 (1)과 (2)는 동일한 명령을 수행한다 우리가 만들 시스템의 타겟이 subsystem이다. 우선 subdir이라는 곳으로 가서, 거기에 있는 Makefile을 동작시키게 된다. MAKE라는 것은 그냥 make라는 명령어를 표시하는 매크로일 뿐이다.)

```
SUFFIXES : .c .o
CC = gcc
CFLAGS = -O2 -g
```

Makefile 활용(팁)

```
all : DataBase Test <- 요기에 집중.

DataBase:
    cd db ; $(MAKE)  (db 로 이동해서 make 실행 )

Test:
    cd test ; $(Make)  ( db 로 이동해서 make 실행 )
```

```
$make
cd db ; make
make[1]: Entering directory`/home/raxis/TEST/src'
gcc -O2 -g -c DBopen.c -o DBopen.o
gcc -O2 -g -c DBread.c -o DBread.o
gcc -O2 -g -c DBwrite.c -o DBwrite.o
make[1]: Leaving directory `/home/windows/TEST/src'
cd test ; make
make[1]: Entering directory `/home/raxis/TEST/test'
gcc -O2 -g -c test.c -o test.o
make[1]: Leaving directory `/home/windows/TEST/test'
(make뒤의 대괄호의 1이라고 나타난 것은 현재의 레벨을 의미한다. 원래 디렉토리의
레벨이 0이고, 여기서는 레벨이 하나 더 내려갔으므로 1이라고 표시된 것이다.)
```

Makefile 활용(팁)

- Makefile를 작성할 때 알면 좋은 것들
  - 불필요한 재컴파일 막기

의존 관계 규칙에 의해 하나가 바뀌면 그에 영향받는 모든 파일이 바뀐다고 앞에서 말했다. 그러나 다른 파일들에게 아무 영향을 주지 않도록 수정하였는데도 재컴파일을 시도한다면 시간 낭비가 될 수도 있다. 가령 모든 .c 파일에서 include 하는 헤더 파일에서 새로운 #define PI 3.14 라고 정의를 했다고 가정하자. 그리고 PI라는 것은 아무 곳에서도 사용을 하지 않는다. 이때는 'make -t' 라고 해보자. -t 는 touch를 의미하는 옵션으로써 컴파일을 하지 않는 대신 파일의 생성 날짜만 가장 최근으로 바꾸어 놓는다. 새로 컴파일 된 것처럼 처리를 하는 것이다. touch유틸리티 명령어에 익숙한 사람이라면 이해할 것이다. touch는 파일의 생성 날짜를 현재로 바꾸어 주는 간단한 유틸리티이다.

Makefile 실행

❖ Makefile에 사용되는 옵션

-C dir
위에서도 밝혔듯이 Makefile을 계속 읽지 말고 우선은 dir로 이동하라는 것이다. 순환 make에 사용된다.

-d
Makefile을 수행하면서 각종 정보를 모조리 출력해 준다. (-debug) 결과를 파일로 저장해서 읽어보면 make 의 동작을 대충 이해할 수 있다.

-h 옵션에 관한 도움말을 출력한다. (-help)

-f file file 에 해당하는 파일을 Makefile로써 취급한다. (-file)

-r
내장하고 있는 각종 규칙(Suffix rule 등)을 없는 것으로 (-no-builtin-rules)간주한다. 따라서 사용자가 규칙을 새롭게 정의해 주어야 한다.

-t 파일의 생성 날짜를 현재 시간으로 갱신한다. (-touch)

-v make의 버전을 출력한다. (전 GNU make 3.73 을 씁니다.) (-version)

-p make에서 내부적으로 세팅되어 있는 값들을 출력한다. (-print-data-base)

-k make는 에러가 발생하면 도중에 실행을 포기하게 되는데 (-keep-going) -k 는 에러가 나더라도 멈추지 말고 계속 진행하라는 뜻.

Makefile 예제

Makefile의 실제 예제

```
HOSTARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ \
                -e s/arm.*/arm/ -e s/sa110/arm/ -e s/macppc/ppc/)

HOSTOS := $(shell uname -s | tr A-Z a-z)

ifeq ($(HOSTARCH),ppc)
CROSS_COMPILE =
else
CROSS_COMPILE =
endif

export CROSS_COMPILE HOSTARCH

TOPDIR      := $(shell if [ "$$PWD" != "" ]; then echo $$PWD; else pwd; fi)

export TOPDIR

include $(TOPDIR)/config.mk

SUBDIRS = common driver

#OBJS = ascu/libascu.a
OBJS += driver/libdriver.a
#OBJS += net/libnet.a
OBJS +=     common/libcommon.a
```

Makefile 예제

```
all:        ascu
            @for dir in $(SUBDIRS); \
            do \
                        $(MAKE) -C $$dir || exit 1 ; \
            done
ascu:       depend subdirs $(OBJS) $(LDSCRIPT)
                        $(CC) -o ascu_prog $(OBJS)  -D_REENTRANT -lpthread
subdirs:
                        @for dir in $(SUBDIRS) ; \
                        do \
                                    $(MAKE) -C $$dir || exit 1 ; \
                        done
depend dep:
                        @for dir in $(SUBDIRS) ; \
                        do \
                                    $(MAKE) -C $$dir .depend ; \
                        done
clean:
            rm -f `find . -type f \
                        \( -name '*.o' -o -name '*.a' \) -print`
            rm          -f ascu_prog ascu.elf ascu.map
clobber:    clean
            rm -f `find . -type f \
                        \( -name .depend -name '*.o' -o -name '*.a' \) \
                        -print`
            rm -f ascu_prog ascu.elf ascu.map
```

Makefile 예제

```
include $(TOPDIR)/config.mk

LIB = ascu

LIBDIR    = lib$(LIB).a

OBJS      = $(patsubst %.c,%.o,$(wildcard *.c))

$(LIBDIR):.depend $(OBJS)
          $(AR) crv $@ $^

#########################################################################

.depend:  Makefile $(SOBJS:.o=.S) $(OBJS:.o=.c)
                  $(CC) -M $(CFLAGS) $(SOBJS:.o=.S) $(OBJS:.o=.c) > $@

sinclude .depend

#########################################################################
```

Makefile 예제

Makefile에 의해 생성된 dependency 파일을 보여 주고 있다.

```
main.o: main.c /usr/include/stdio.h /usr/include/features.h \
 /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
 /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include/stddef.h \
 /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include/stdarg.h \
 /usr/include/bits/types.h /usr/include/libio.h \
 /usr/include/_G_config.h /usr/include/bits/stdio_lim.h \
 /usr/include/bits/stdio.h
```