

10. 프로그래밍 도구

10.1 C 컴파일러

- UNIX/Linux 운영체제 커널, 유틸리티, 라이브러리 등은 대부분 C언어로 작성됨
- UNIX C 컴파일러
 - `cc` ... 원래는 기본으로 포함되었으나, 지금은 통합개발환경이 포함된 상업용 컴파일러를 유료로 판매
- Linux C 컴파일러
 - `gcc` ... GNU C compiler
 - `g++` ... GNU C++ compiler
 - `/usr/bin/cc`는 `/usr/bin/gcc`에 symbolic link가 되어 있어서 `cc` 명령어 사용 가능
 - 사용법
 - `cc [-옵션 ...] 파일 ...`
 - `gcc [-옵션 ...] 파일 ...`

C컴파일러 옵션

옵션	동작
<code>-o outfile</code>	컴파일 결과를 파일 <code>outfile</code> 에 저장함 (<code>-o</code> 옵션이 없으면 <code>a.out</code> 에 저장)
<code>-c</code>	소스 파일을 컴파일하여 오브젝트 파일을 확장자 <code>.o</code> 인 파일에 저장
<code>-g</code>	디버거를 위한 디버깅 정보를 컴파일 출력에 포함 (<code>gdb</code> 가 사용)
<code>-pg</code>	프로파일을 위한 정보를 컴파일 출력에 포함 (<code>gprof</code> 가 사용)
<code>-Idir</code>	디렉토리 <code>dir</code> 을 헤더 파일을 검색할 디렉토리에 추가
<code>-Olevel</code>	<code>level</code> 이 지정하는 수준의 최적화 컴파일. <code>level</code> 은 0, 1, 2, 3, s를 사용. (이 옵션이 없거나 0이면 최적화를 수행하지 않고 컴파일)
<code>-O</code>	<code>-O1</code> 과 같음
<code>-Dname</code>	매크로 <code>name</code> 를 정의 (<code>#define name</code> 과 같은 역할)
<code>-Dname=def</code>	매크로 <code>name</code> 을 값 <code>def</code> 로 정의 (<code>#define name def</code> 와 같은 역할)
<code>-llib</code>	링크용 라이브러리 지정
<code>-Ldir</code>	디렉토리 <code>dir</code> 을 라이브러리 디렉토리에 추가
<code>-S</code>	기계어가 아닌 어셈블리 파일을 생성 (확장자 <code>.s</code>)
<code>-std=standard</code>	<code>standard</code> 로 지정되는 버전의 C언어를 사용 (기본적으로 <code>c89</code> 를 확장한 <code>gnu89</code> 를 사용하며, <code>c89</code> , <code>c99</code> , <code>gnu99</code> , <code>c++98</code> , <code>gnu++98</code> 등을 사용가능)

옵션 처리 함수

- 유틸리티 옵션
 - 인수에서 `-`로 시작하여 제공
 - 프로그램에서 옵션을 처리해야 할 필요가 있음
- 옵션 처리 함수
 - `<unistd.h>`

```
int getopt(int argc, char *const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```
 - `getopt()` – 옵션을 parsing하는 함수
 - `arg, argv` : main의 인수 전달
 - `optstring` : 사용하는 옵션문자들로 구성된 문자열
 - 추가 인수가 필요하면 : 와 함께 전달
 (예) `getopt (argc, argv, "f:hv")`
 - 반환값이 `-1`이 될 때까지 반복호출하며, 부가 정보는 전역변수들을 통하여 제공됨

단일 모듈 프로그램

- 단일 모듈 프로그램 - 하나의 파일로 작성
- (예) 옵션을 처리하는 C 프로그램
 - 소스코드 "opt.c" - 교과서 참조
 - 컴파일
 - \$ cc opt.c ... 실행파일: a.out
 - \$ cc opt.c -o opt ... 실행파일: opt
 - \$ cc -o opt opt.c ... 실행파일: opt
 - 실행
 - \$ a.out ... 또는 opt 현재디렉토리(.)가 경로설정된 경우
 - \$./a.out ... 또는 ./opt ...
 - 실행예
 - \$ opt -f my -h -v
 - \$ opt -fmy -h -v ... 옵션 추가정보를 붙여쓰기해도 됨
 - \$ opt -hv -f my
 - \$ opt -hvf my ... my는 f옵션의 파일이름으로 인식
 - \$ opt -fhv my ... hv 가 f옵션의 파일이름으로 인식

5

다중 모듈 프로그램

- 다중 모듈 프로그램 - 여러 개의 파일로 작성
 - 재사용 가능 함수들을 별도의 파일로 작성하면 다른 프로그램에서 쉽게 다시 사용할 수 있음
 - 재사용 가능 함수의 원형 선언은 header 파일에서 별도로 하는 것이 바람직함
- (예) 옵션을 처리하는 C 프로그램 - 동작 포함
 - f 출력파일 이름 지정 (없으면 표준출력)
 - v 인수 문자열을 역순으로 출력
 - h 사용법 출력
 - 잘못된 옵션: 사용법 출력(종료코드 1)
- 소스코드 구성 - "교과서 참조"
- reverse.c ... reverse 함수정의
- reverse.h ... reverse 함수 원형선언
- opt2.c ... main 함수

6

다중 모듈 프로그램(계속)

- 컴파일
 - \$ cc -o opt2 opt2.c reverse.c
- 실행
 - \$ opt2 123 abc
 - \$ opt2 -v 123 abc
 - \$ opt2 -f out -v 123 abc
 - \$ opt2 -vh ... h 옵션이 있으면 다른 옵션은 무시됨
- 개별 컴파일 후 링크
 - \$ cc -c opt2.c ... opt2.o 생성
 - \$ cc -c reverse.c ... reverse.o 생성
 - \$ cc opt2.o reverse.o -o opt2 ... object 파일을 링크하여 실행파일 생성
- 소스코드가 수정되면 해당 파일만 컴파일 후 링크 수행
- 소스코드 파일 수가 많을 경우 이 방법이 효율적
 - 대개 **make** 유틸리티와 함께 사용

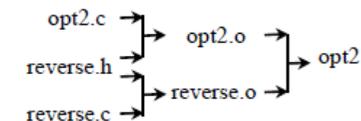
7

10.2 make 유틸리티

- make 명령어
 - 파일이 생성되는 의존 관계를 이용하여 최종 결과 파일을 생성하기 위해서 필요한 연속적인 작업들을 자동적으로 순서대로 실행
 - makefile, Makefile
 - 파일의 의존관계와 파일 생성에 필요한 명령어들을 명시한 파일
- makefile 형식 - 다음 형식의 리스트로 구성됨

```
targetList : dependency_list
<탭> 명령어 리스트
```

- 파일의 의존 관계



8

makefile과 컴파일

■ makefile

```
opt2 : opt2.o reverse.o
    cc opt2.o reverse.o -o opt2
opt2.o : opt2.c reverse.h
    cc -c opt2.c
reverse.o : reverse.c reverse.h
    cc -c reverse.c
```

■ 컴파일

```
$ make
cc -c opt2.c
cc -c reverse.c
cc opt2.o reverse.o -o opt2
```

■ opt2.c 수정 후 컴파일 - opt2.c만 재 컴파일 후 링크

```
$ make
cc -c opt2.c
cc opt2.o reverse.o -o opt2
```

9

makefile, touch

■ makefile

- makefile ... 기본
 - Makefile ... makefile이 없는 경우
 - -f 옵션을 지정하여 임의의 이름 사용 가능
- ```
$ make -f my.make ... my.make가 makefile로 사용됨
```

### ■ touch 명령어

- make 실행 후에 소스코드가 변경되지 않았을 때의 make 실행
- ```
$ make
make: `opt2'는 이미 갱신되었습니다.
```
- touch - 파일 수정시간을 현재 시간으로 변경, make 재실행시 유용
- ```
$ touch reverse.h
$ make
cc -c opt2.c
cc -c reverse.c
cc opt2.o reverse.o -o opt2
```

10

## makefile 변수

### ■ makefile 변수, 매크로

- 정의: 변수 = 문자열
- 사용: \$(변수), \${변수}

```
opt2 : opt2.o reverse.o
 cc opt2.o reverse.o -o opt2
opt2.o : opt2.c reverse.h
 cc -c opt2.c
reverse.o : reverse.c reverse.h
 cc -c reverse.c
```



```
OBJS = opt2.o reverse.o ... 변수 정의

opt2 : $(OBJS) ... 변수 값 사용
 cc $(OBJS) -o opt2 ... 변수 값 사용
opt2.o : opt2.c reverse.h
 cc -c opt2.c
reverse.o : reverse.c reverse.h
 cc -c reverse.c
```

11

## make의 미리 정의된 규칙

### ■ make의 미리 정의된 C 컴파일 규칙

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

- CC : C 컴파일러 변수 (기본: cc)
- CFLAGS : C 컴파일러 옵션 (기본: 없음)
- 확장자 .c 파일에서 확장자 .o 파일을 생성하는 경우  
명령어가 없으면 미리 정의된 컴파일 규칙이 적용됨

```
OBJS = opt2.o reverse.o ... 변수 정의
```

```
opt2 : $(OBJS) ... 변수 값 사용
 cc $(OBJS) -o opt2 ... 변수 값 사용
```

```
opt2.o : opt2.c reverse.h
reverse.o : reverse.c reverse.h
```

- 컴파일러와 옵션 지정

```
$ make
CC = gcc
CFLAGS = -O
...
gcc -O -c -o opt2.o opt2.c
gcc -O -c -o reverse.o reverse.c
cc opt2.o reverse.o -o opt2
```

12

## make의 목표 타겟 인수

- makefile에는 여러 개의 목표 타겟을 지정할 수 있음

- 인수 없이 실행하면 가장 앞의 타겟을 목표로 사용
- 인수로 목표 타겟 지정 가능

```
$ make reverse.o
cc -c -o reverse.o reverse.c
```

- 가짜 타겟(phony target)

- 파일 생성 목적이 아니라, 명령어 실행을 목적으로 사용되는 타겟

```
OBJS = opt2.o reverse.o ... 변수 정의
opt2 : $(OBJS) ... 변수 값 사용
 cc $(OBJS) -o opt2 ... 변수 값 사용
opt2.o : opt2.c reverse.h
reverse.o : reverse.c reverse.h

clean:
 -rm $(OBJS)
```

```
$ make clean
rm opt2.o reverse.o
```

13

- 예

```
all: target1 target2 target3
target1: ...
 ...
target2: ...
 ...
target3: ...
 ...
```

```
$ make ... make all - 모든 target 작업 실행
$ make target2 ... target2 작업 실행
```

- makefile 또는 정의되지 않은 target에 대한 make 실행

```
$ make my
cc my.c -o my ... C 컴파일 수행
```

14

## 10.3 디버거

- gdb

- GNU debugger - 심볼릭 디버깅 지원

- 디버거용 컴파일 옵션 : -g

- 직접 실행파일 생성
 

```
$ cc -g opt.c -o opt
$ cc -g opt2.c reverse.c -o opt2
```

- 다중 모듈 프로그램 컴파일

- 오브젝트 파일 생성할 때에
  - c와 함께 -g 옵션 사용
 

```
$ cc -g -c reverse.c
$ cc -g -c opt2.c
```
  - 오브젝트 파일을 링크할 때에는
    - g 옵션 불필요
 

```
$ cc reverse.o opt2.o -o opt2
```

### 디버깅을 위한 makefile

```
CFLAGS = -g
OBJS = opt2.o reverse.o
opt2 : $(OBJS)
 cc $(OBJS) -o opt2
opt2.o : opt2.c reverse.h
reverse.o : reverse.c reverse.h
```

15

## gdb 주요 명령어

- gdb 명령어 실행

```
$ gdb exefile
(gdb) ...
```

- gdb 주요 명령어 - 교과서 표 참조

- 실행/종료

- run 인수 ... 실행
- quit ... 종료

- 소스코드 보기

- list (l) ... 현재위치부터 10줄 출력
- list 행번호/함수
- list 파일이름:행번호/함수

- 정지점(breakpoint)

- break (b) 행번호/함수 ... 설정
- clear (c) 행번호/함수 ... 제거
- delete (d) ... 모든 정지점 제거

16

### ■ 데이터 출력

- print (p) 수식 ... 정지점에서 출력
- print /x 수식 ... 16진수 출력
  - 형식 : x, d, u, o, t (2진수), a(주소)

### ■ 연속 실행

- continue (c) ... 다음 정지점까지 실행

### ■ 자동 데이터 출력

- display (disp) 수식
- display /x 수식

### ■ 정지점 관련 추가 기능

- info break ... 현재 설정된 정지점
- condition (cond) 정지점번호 조건식 ... 정지점의 정지 조건
- disable (dis) b .. ... 정지점 비활성화
- enable (en) b .. ... 정지점 활성화

### ■ 데이터 정지점

- watch (wa) 변수/수식 .. 값이 변경되면 정지

### ■ 프로그램 실행 제어

- next (n) ... 한 줄 단위 실행 (함수 호출도 한 줄로 처리)
- step (s) ... 함수 호출은 호출되는 함수로 들어감
- finish (fin) ... 현재 함수 끝까지 실행
- until (u) ... 임시 정지점(한 번만 사용)

### ■ 함수 호출 역추적

- backtrace (bt) ... 프로그램의 스택 프레임 정보
- up ... 현재 함수를 호출한 함수
- down ... 현재 함수가 호출한 함수

### ■ 변수 값 변경 : 프로그램 실행에 영향을 줌

- print a=100
- set var a=200

## 10.4 프로그래밍 관련 유틸리티

### ■ gprof - 프로그램 수행 감시

- 함수의 호출 횟수와 소요 시간을 수집하여 출력
- 성능 개선에 도움을 줌

### ■ gprof 준비 및 실행 과정

1. -pg 옵션을 사용하여 컴파일 (object 파일 생성 및 링크 시에 모두 사용)
2. 프로그램 실행 : profile용 파일인 gmon.out 파일 생성
3. gprof 실행 :

```
$ gprof execfile [gmon.out] ... verbose output
$ gprof -b execfile [gmon.out] ... brief output
```

### ■ splint - C 프로그램 검사

- 프로그램 실행 시 발생할 수 있는 잠재적 오류 검사

## 10.5 라이브러리

### ■ 라이브러리

- 정적 라이브러리 - 링크할 때 실행파일에 포함됨
- 동적 라이브러리 - 공유 라이브러리 사용, 실행할 때에 동적으로 링크 최근에 주로 사용

### ■ ar - 정적 라이브러리 생성

```
$ ar rv libmy.a mystrcpy.o
```

### ■ 정적 라이브러리 사용

```
$ cc -o my my.c libmy.a 로컬 라이브러리
$ cc -o my my.c -lname 표준 라이브러리 디렉토리
$ cc my.c -L. -lmy -o my 라이브러리 디렉토리 추가
```

- nm – 실행파일에 포함된 심볼 출력
- strip – 실행파일에 포함된 심볼 제거
- readelf – 실행파일(elf 파일) 정보 출력
- size – 실행파일 섹션 크기

## 10.6 소스코드 관리 시스템

- 소스코드 관리
  - 개발과정에서 작성한 모든 버전을 저장하고, 접근 및 보호 수행
- 소스코드 관리 시스템 종류
  - SCCS (source code control system)
  - RCS (revision control system)
  - CVS (concurrent versions system) ... 중앙서버 사용
  - SVN (subversion) ... 중앙서버 사용
  - Git ... 분산 서버 사용

## RCS 사용하기

- RCS 디렉토리 생성
  - \$ mkdir RCS
- check in
  - \$ ci file // RCS 디렉토리에 file 대한 RCS file 생성, 현재 file 삭제함
  - \$ ci -u file // 현재의 file을 그대로 둠 (unlock) - 읽기전용
  - \$ ci -l file // 현재의 file을 그대로 둠 (lock 상태) - 계속 편집가능
  - \$ ci -r2.0 file // file의 버전을 2.0으로 하여 저장함
    - 버전은 1.1부터 생성됨, check in 할 때마다 버전이 갱신됨
- check out
  - \$ co file // RCS 디렉토리에서 file의 최신 버전을 가져와서  
// working file로 저장함 (read only)
  - \$ co -l file // locking 상태로 check-out. 다른 사람들이 사용하지 못함
  - \$ co -r1.1 file // file의 1.1버전을 가져옴
- log 보기
  - \$ rlog file // file이 revision 정보를 보여줌