5장. 프로그래밍 개발 환경

GCC

C 언어의 개요 및 특징

- **Unix**를 다시 쓰기 위한 언어로 **Dennis Ritchie**에 의해 설계
- 타 기종간의 이식성**(Portability)**와 유연성**(Flexible)**을 지닌 강력한 언어
- 간결**(Compact)**한 문법구조와 실행속도가 매우 빠른 언어
- 구조화 프로그래밍**(Structured Programming)** 구현
- 시스템 프로그래밍**(System Programming)** 가능
- 다양한 데이터 유형**(Data Type)**
- 가독성**(Readability)**이 뛰어나 유지보수**(Maintenance)**
- 모듈**(Module)**와 함수**(Function)**에 의한 프로그램 구조
- **Unix**에서 여러 **s/w** 개발 가능
- 프로그래머에 대한 융통성 제공

GCC

GCC (GNU Compiler Collection)

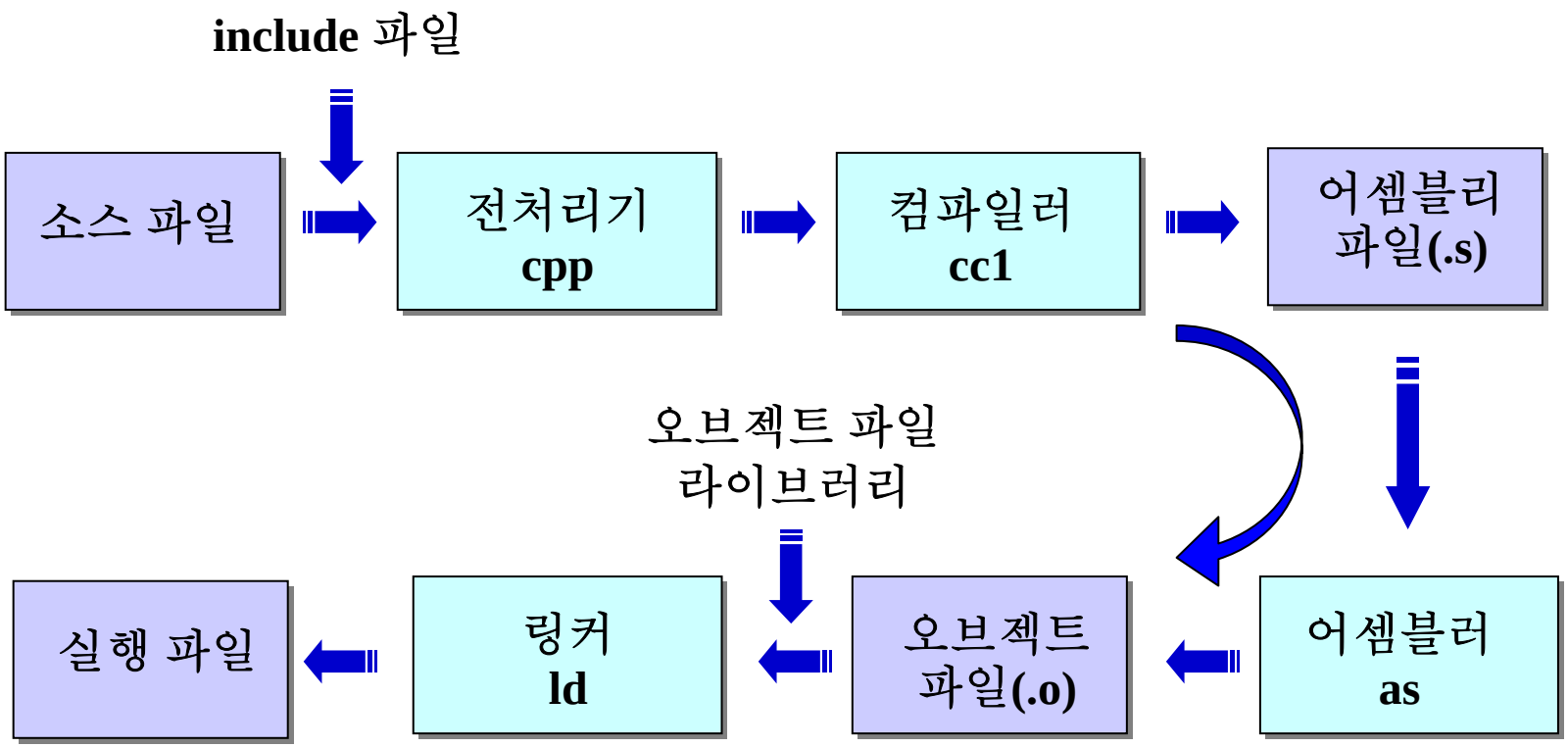
- 수 많은 컴퓨터 프로그래밍 언어들을 위한 라이브러리와 컴파일러들의 모음
- **C, C++, Objective-C, Pascal, Fortran, Java, Ada** 등의 언어 지원
- **GCC** 패키지의 주요 프로그램

파일	설 명
gcc	**GNU C** 와 **C++** 컴파일러, **cc1** 과 **g++** 의 기능을 모두 포함
cpp	**C** 언어 전처리기
cc1	실제 **C** 컴파일러
as	어셈블러
ld	링커

- 공식 홈페이지 : http://www.gnu.org/software/gcc/

GCC

- 소스 프로그램부터 최종 실행 파일이 만들어지는 과정



실행 예제

GCC 정보

```
[cprog2@seps5 cprog2]# gcc -v
Reading specs from /usr/lib/gcc-lib/i386-hancom-linux/3.2.3/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --enable-shared --enable-threads=posix --disable-checking --with-system-zlib --enable-__cxa_atexit --host=i386-hancom-linux
Thread model: posix
gcc version 3.2.3 20030422 (Hancom Linux 3.2.3)
[cprog2@seps5 cprog2]#
```

GCC

▣ 실행 방법

◈ “gcc” 명령 실행

- ➢ 컴파일과 링크를 동시에 수행

◈ 간단한 실행 예제

- ➢ C 언어 소스 파일(.c) 작성 후 실행하면, “a.out” 실행파일로 만듦

```
#include <stdio.h>
main()
{
    printf("Hello, World\n") ;
}
```

```
[cprog2@seps5 cprog2]# gcc hello.c
[cprog2@seps5 cprog2]# ./a.out
Hello, World
[cprog2@seps5 cprog2]#
```

GCC

gcc 명령어 옵션

옵션	기능
-v	실행 명령어들과 버전을 출력한다
-E	전처리만 실행한다; 컴파일하거나 어셈블하지 않는다
-S	컴파일만 실행한다; 어셈블하거나 링크하지 않는다
-c	컴파일 또는 어셈블한다; 링크하지 않는다
-g	운영체제 고유 형식으로 디버깅 정보를 만든다
-o <file>	출력을 **file** 에 둔다
-I<dir>	헤더 파일을 검색할 디렉토리를 추가한다
-L<dir>	**-l** 을 위해 검색할 디렉토리를 추가한다
-D<macro>	매크로를 미리 지정한다
-O<level>	최적화 수전을 지정한다. **level** 이 없으면 **–O1** 과 같다
-l<lib>	링크할 라이브러리 파일을 지정한다

GCC 실행 예제

- 출력 파일 지정 예

```
[cprog2@seps5 cprog2]#  gcc -o hello hello.c
[cprog2@seps5 cprog2]# ./hello
Hello, World
cprog2@seps5 cprog2]#
```

GCC 실행 예제

▣ 전처리 예

◈ 전처리를 수행하면, C 구문에서 “#”으로 시작하는 모든 코드가 해석되어 소스에 포함된다.

➢ 헤더 파일들, 매크로 등

```
[cprog2@seps5 cprog2]#  gcc -E -o hello.i hello.c
[cprog2@seps5 cprog2]#  cat hello.i
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3
....
[cprog2@seps5 cprog2]#
```

GCC 실행 예제

- 어셈블러
  - 어셈블리 코드 생성
  - CPU 종류마다 어셈블리 명령어가 다름

```
[cprog2@seps5 cprog2]# gcc -S hello.c
[cprog2@seps5 cprog2]# cat hello.s
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello, World\n"
        .text
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
movl    $0, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
          .ident   "GCC: (GNU) 3.2.3 20030422 (Hancom Linux
3.2.3)"
[cprog2@seps5 cprog2]#
```

GCC

프로그램 링킹(Linking)

- 하나 또는 여러 개의 목적(Object) 파일을 통합하여 하나의 실행 가능한 실행 프로그램으로 만드는 작업
- 링커(Linker) – 링킹에 사용하는 도구 프로그램
  - gcc, ld
- 로더(Loader) – 실행 프로그램을 주기억장치에 설치한 후 실행시킴

링킹 예

```
[cprog2@seps5 cprog2]# gcc -c hello.c
[cprog2@seps5 cprog2]# ls
hello.c  hello.o
[cprog2@seps5 cprog2]#
```

```
[cprog2@seps5 cprog2]# gcc -o hello hello.o
[cprog2@seps5 cprog2]# ls
hello   hello.c  hello.o
[cprog2@seps5 cprog2]#
```

GCC 실행 예제

- 소스 파일이 2개 이상인 경우의 링킹 예

```
/* main.c */
#include <stdio.h>
extern void sub();
main()
{
        printf("This is main file.\n");
        sub();
}
```

```
/* sub.c */
sub()
{
        printf("This is sub file.\n");
}
```

```
[cprog2@seps5 cprog2]# gcc -c main.c
[cprog2@seps5 cprog2]# gcc -c sub.c
[cprog2@seps5 cprog2]# gcc -o main main.o sub.o
[cprog2@seps5 cprog2]# ./main
This is main file.
This is sub file.
[cprog2@seps5 cprog2]#
```

GCC 실행 예제

▣ 헤더 파일 포함(include)

◈ 헤더파일이 “header” 라는 서브디렉토리에 있을 때

```
/* main.c */
#include <stdio.h>
#include "main.h"
main()
{
        printf("This is %s file.\n",
MAIN);
}
```

```
/* main.h */
#define MAIN "main.c"
```

```
[cprog2@seps5 cprog2]# gcc -o main -I./header main.c
[cprog2@seps5 cprog2]# ./main
This is main.c file.
[cprog2@seps5 cprog2]#
```

라이브러리

▣ 링킹의 종류

◈ 정적 링킹(static linking)

- ➢ 최종 실행 파일에 필요한 오브젝트 파일들을 미리 링크하여 실행 파일에 함께 포함
- ➢ 장점 - 실행 파일만 있으면 별도의 파일 없이 실행 가능
- ➢ 단점 - 실행파일 크기가 커지고, 라이브러리 갱신 시 관련된 실행 파일들을 모두 컴파일하여 갱신

◈ 동적 링킹(dynamic linking)

- ➢ 필요한 파일들을 미리 링크하지 않고, 실행하려고 할 때 필요한 프로그램 모듈들을 결합하여 실행 계속
- ➢ 장점 - 실행파일 크기가 작아져 메모리를 조금 차지
- ➢ 단점 - 실행 파일 이외에 별도의 필요한 라이브러리를 제공
- ➢ **UNIX** 또는 **LINUX** 에서 대부분의 실행 파일에 해당

라이브러리

- 유용한 기능을 가진 프로그램 모듈들을 모아 놓은 파일
  - **C library(libc) – printf** 등의 **C** 표준 함수들 제공
  - **Math library(libm) – sin** 등의 수학 함수들 제공
  - **UNIX/LINUX** 에서 **/lib, /usr/lib, /usr/local/lib** 등에 둠

- 라이브러리의 종류
  - 정적 라이브러리**(static library) –** 정적 링킹 시 사용
    - **.a** 확장자를 가짐
  - 공유 라이브러리**(shared library) –** 동적 링킹 시 사용
    - **.so** 확장자를 가짐
  - 동적 라이브러리**(dynamic library) –** 동적 로딩 시 사용
    - 실행 도중에 동적으로 로딩됨
    - 필요한 라이브러리가 수시로 등록, 실행, 제거 가능하므로 유연함
    - 예) 웹 응용에서 플러그인 모듈

라이브러리

라이브러리 관련 gcc 옵션

옵션	기능
-L경로	라이브러리가 있는 경로를 삽입
-l라이브러리	라이브러리가 있는 표준 디렉토리 “/usr/lib", "/lib"를 탐색
-n	동적 모듈에 반대되는 독립적 수행 모델 생성

링킹 예제

```
1   /* ex3_5.c */
2   #include <stdio.h>
3   #include <math.h>
4   #define PI      3.14159265
5
6   main()
7   {
8           printf("sin(PI/2) = %g.\n", sin(PI/2));
9   }
```

라이브러리

링킹 및 실행

```
[cprog2@seps5 cprog2]# gcc -o ex3_5 ex3_5.c
/tmp/ccovioHZ.o(.text+0x21): In function `main':
: undefined reference to `sin'
collect2: ld returned 1 exit status
[cprog2@seps5 cprog2]# gcc -o ex3_5 ex3_5.c -lm
[cprog2@seps5 cprog2]# ./ex3_5
sin(PI/2) = 1.
[cprog2@seps5 cprog2]#
This is main.c file.
[cprog2@seps5 cprog2]#
```

링킹 확인 – “ldd” 명령 사용

```
[cprog2@seps5 cprog2]#  ldd ex3_5
        libm.so.6 => /lib/libm.so.6 (0x40026000)
        libc.so.6 => /lib/libc.so.6 (0x40048000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[cprog2@seps5 cprog2]#
```

라이브러리

▣ 정적 링킹

◈ 링킹 시에 gcc 명령에서 “-static” 옵션 사용

```
[cprog2@seps5 cprog2]# gcc -static -o ex3_5 ex3_5.c -lm
[cprog2@seps5 cprog2]# ./ex3_5
sin(PI/2) = 1.
[cprog2@seps5 cprog2]# ldd ex3_5
        not a dynamic executable
[cprog2@seps5 cprog2]#
```

라이브러리

▣ 정적 라이브러리

◈ 정적 라이브러리 제작

- ➢ **"ar"** 명령을 사용하여 오브젝트 파일들을 묶음
- ➢ 아카이브**(.a)** 파일 생성

ar	
일반형식	ar [options] archive files…
주요옵션	d : 아카이브로부터 오브젝트 모듈들을 제거 r : 아카이브에 오브젝트 모듈들을 삽입. 이전에 존재하는 같은 모듈이 있으면 새로운 모듈로 대체. t : 아카이브 내용을 출력 x : 아카이브로부터 오브젝트 모듈을 추출 c : 아카이브 파일을 생성 s : 아카이브에 오브젝트 파일 인덱스를 기록

라이브러리

▣ 정적 라이브러리 사용

◈ 정적 라이브러리 제작 예

```
/* max.c */
int max(int a, int b)
{
        if (a > b) return a;
        else return b;
}
```

```
/* min.c */
int min(int a, int b)
{
        if (a < b) return a;
        else return b;
}
```

```
/* testlib.h */
int max(int a, int b);
int min(int a, int b);
```

```
[cprog2@seps5 lib]$ gcc -c max.c
[cprog2@seps5 lib]$ gcc -c min.c
[cprog2@seps5 lib]$  ls
max.c  max.o  min.c  min.o  testlib.h
[cprog2@seps5 lib]$ ar rcs libtest.a max.o min.o
[cprog2@seps5 lib]$
```

라이브러리

정적 라이브러리 사용

```
/* ex3_ .c */
#include <stdio.h>
#include "testlib.h"

main()
{
        printf("max(1,2) = %d\n", max(1,2));
        printf("min(1,2) = %d\n", min(1,2));
}
```

```
[cprog2@seps5 cprog2]$ gcc -I./lib -L./lib main.c -ltest
[cprog2@seps5 cprog2]$ ls
a.out  lib  main.c
[cprog2@seps5 cprog2]$ ./a.out
max(1,2) = 2
min(1,2) = 1
[cprog2@seps5 cprog2]$
```

라이브러리

▣ 공유 라이브러리

◈ 공유 라이브러리 제작

➢ "gcc" 명령의 여러 옵션 사용

- "-shared" 는 공유 라이브러리를 사용한다는 명령
- "-Wl,…." 은 이후에 나오는 메개변수들을 링커에 전달

```
[cprog2@seps5 lib]$ gcc -shared -Wl,-soname,libtest.so.1 -o libtest.so.1.0.1 max.o min.o
[cprog2@seps5 lib]$ ln -s libtest.so.1.0.1 libtest.so.1
[cprog2@seps5 lib]$ ln -s libtest.so.1.0.1 libtest.so
[cprog2@seps5 lib]$
```

```
[cprog2@seps5 cprog2]$ gcc -I./lib -L./lib main.c -ltest
[cprog2@seps5 cprog2]$ ./a.out
./a.out: error while loading shared libraries: libtest.so.1: cannot open shared object file: No such file or directory
[cprog2@seps5 cprog2]$ ldd a.out
        libtest.so.1 => not found
        libc.so.6 => /lib/libc.so.6 (0x40027000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[cprog2@seps5 cprog2]$
```

라이브러리

▣ 공유 라이브러리 사용

◈ 공유 라이브러리와 링크하기 위해서는 공유 라이브러리 지정 경로가 설정되어 있어야 함.

- ➢ 시스템 지정은 **"/etc/ld.so.conf"** 에 공유 라이브러리 디렉토리를 포함한 다음 **"ldconfig"** 명령으로 **"/etc/ld.so.cache"** 갱신
- ➢ 사용자 지정은 **"LD_LIBRARY_PATH"** 환경변수에 경로 설정

```
[cprog2@seps5 cprog2]$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/lib
[cprog2@seps5 cprog2]$ ldd ./a.out
        libtest.so.1 => /home/cprog2/lib/libtest.so.1 (0x40011000)
        libc.so.6 => /lib/libc.so.6 (0x40029000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[cprog2@seps5 cprog2]$ ./a.out
max(1,2) = 2
min(1,2) = 1
[cprog2@seps5 cprog2]$
```

라이브러리

▣ 동적 라이브러리 사용

	동적 라이브러리 인터페이스
형식	**#include <dlfcn.h>** **void *dlopen (const char *filename, int flag);** **const char *dlerror(void);** **void *dlsym(void *handle, char *symbol);** **int dlclose (void *handle);**
기능	dlopen 함수는 filename 의 라이브러리를 적재한다. dlerror 함수는 dlopen, dlsym, dlclose 함수에 의해 발생한 가장 최근의 오류를 출력한다. dlsym 함수는 dlopen 에 의해 적재된 라이브러리를 사용할 수 있도록 심볼들을 찾는다. dlclose 함수는 적재된 라이브러리의 handle 에 대한 참조계수를 감소시킨다.
반환값	dlopen 함수는 성공적으로 호출되면 적재 라이브러리에 대한 handle을 반환. 실패한 경우 NULL을 반환. dlerror 함수는 초기에 호출되면 오류 메시지 주소 반환. 연속적으로 두 번 이상 호출되면 NULL 반환. dlsym 함수는 성공적으로 호출되면 심볼의 적재된 주소를 반환. 심볼을 찾지 못하면 NULL 반환. dlclose 함수는 성공적으로 호출되면 NULL 반환. 그렇지 않으면 다른 값 반환.

라이브러리

▣ 동적 라이브러리 사용

◈ 동적 라이브러리 인터페이스 함수들을 사용하여 소스 제작

◈ 링크 시 **libdl.so** 과 링크하고**(-ldl), "-rdynamic"** 옵션 사용

```
/* ex3_6.c */
#include <stdio.h>
#include <dlfcn.h>

main()
{
        void *handle;
        int (*max)(int, int), (*min)(int, int);
        char *error;

                handle = dlopen ("./lib/libtest.so",
RTLD_LAZY);
        if (!handle) {
                fputs (dlerror(), stderr);
                exit(1);
        }
        max = dlsym(handle, "max");
        if ((error = dlerror()) != NULL)  {
                fprintf (stderr, "%s", error);
                exit(1);
        }
        min = dlsym(handle, "min");
        if ((error = dlerror()) != NULL)  {
                fprintf (stderr, "%s", error);
                exit(1);
        }
        printf ("max(1,2)=%d\n", (*max)(1,2));
        printf ("min(1,2)=%d\n", (*min)(1,2));
        dlclose(handle);
}
```

◈ 실행 결과

```
[cprog2@seps5 cprog2]$ gcc -rdynamic
ex3_6.c -ldl
[cprog2@seps5 cprog2]$ ./a.out
max(1,2)=2
min(1,2)=1
[cprog2@seps5 cprog2]$
```

Make

- 모듈화된 프로그램
  - 여러 개의 부분 프로그램 또는 모듈을 포함
  - 장점
    - 재사용 및 디스크 공간을 효율적으로 사용
  - 단점
    - 수정 시에 수작업으로 소스 프로그램 컴파일
    - 하지만, 유지 보수에 어려움이 존재함

- **Make**
  - 많은 프로그램 모듈로 구성된 대규모 프로그램 소스를 효율적으로 유지하고 일관성 있게 관리하도록 도와주는 도구
    - 여러 파일들 간의 의존성을 저장하고 수정된 파일에 연관된 소스 파일들만 재컴파일 가능

Make

Makefile

- **Make** 도구에서 가장 중요한 파일
- 최종 응용 프로그램 구성에 있어서 소스 파일들 사이의 다양한 의존성에 대한 규칙을 기술
- 형식

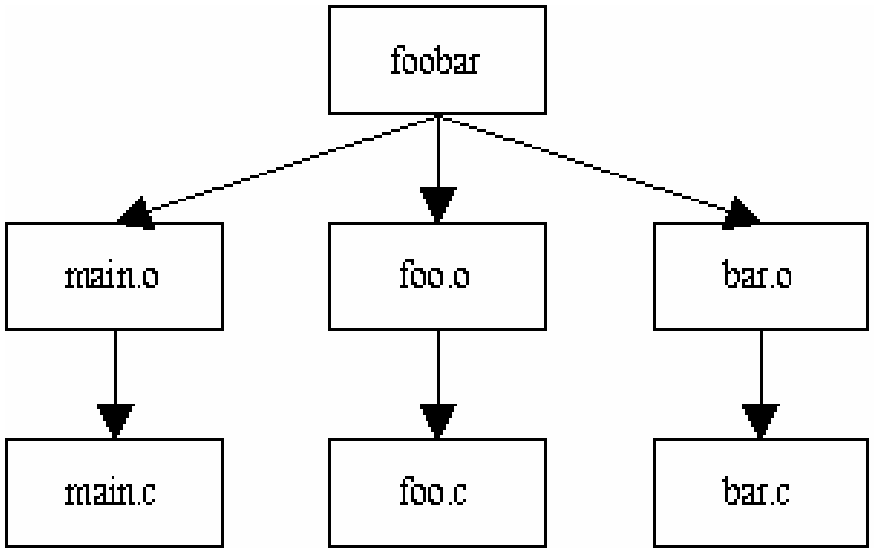
```
대상(TARGET) ... : 의존하는 파일들(PREREQUISITES) ...
    명령(COMMAND)
    ...
    ...
```

  - 대상 : 프로그램이 생성하는 목적 파일 이름 (오브젝트/실행 파일 등)
  - 의존 파일 : 대상을 생성하기 위해 필요한 파일들
  - 명령 : **'make'** 가 실행하는 명령어

Make

▣ 실행 예제

◈ 최종 실행 파일 **“foobar”** 는 **main.o, foo.o, bar.o** 에 의존하며, 마찬가지로 **main.o, foo.o, bar.o** 는 각각 **main.c, foo.c, bar.c** 에 의존



Make

▣ 실행 예제

◈ 의존관계

```
foobar : main.o foo.o bar.o
main.o : main.c
foo.o : foo.c
bar.o : bar.c
```

◈ 실행 명령

```
gcc -o foobar main.o foo.o bar.o
```

```
gcc -c main.c
gcc -c foo.c
gcc -c bar.c
```

◈ 최종 Makefile

```
foobar : main.o foo.o bar.o
        gcc -o foobar main.o foo.o bar.o
main.o : main.c
        gcc -c main.c
foo.o : foo.c
        gcc -c foo.c
bar.o : bar.c
        gcc -c bar.c
clean :
        rm –f foobar main.o foo.o bar.o
```

Make

▣ 실행 예제

◈ 실행 결과

```
[cprog2@seps5 make]$ make
gcc -c main.c
gcc -c foo.c
gcc -c bar.c
gcc -o foobar main.o foo.o bar.o
[cprog2@seps5 make]$
```

```
[cprog2@seps5 make]$ ./foobar
Hello, world!
Goodbye, my love.
[cprog2@seps5 make]$
```

Make

▣ 실행 예제

◈ 실행 결과

➢ 모든 파일이 갱신되면, 추가 변경 사항 없음

```
[cprog2@seps5 make]$ make
make: `foobar'          ↑       .
[cprog2@seps5 make]$
```

➢ 수정한 파일에 관련된 부분만 재실행됨

```
[cprog2@seps5 make]$ touch main.c
[cprog2@seps5 make]$ make
gcc -c main.c
gcc -o foobar main.o foo.o bar.o
[cprog2@seps5 make]$
```

Make

clean 기능

- Make 실행 전 상태로 되돌리기 위해, 디렉토리 파일들 정리
- 컴파일 과정에서 만들어진 오브젝트/실행 파일들을 제거
- 실행 예

```
[cprog2@seps5 make]$ make clean
rm –f foobar main.o foo.o bar.o
[cprog2@seps5 make]$ ls
Makefile bar.c foo.c main.c
[cprog2@seps5 make]$
```

Make

주석과 매크로

주석 – '#' 기호로부터 시작하여 줄의 끝까지 모든 문자

```
# COMMENT ......
```

매크로

- 긴 문자열을 하나의 매크로로 정의하여 여러 곳에 사용 가능
- 형식

```
NAME = value
```

- 정의 예

```
INCLUDES = /usr/local/include1 /usr/local/include2
DEBUGS = -g -O2
```

Make

매크로 사용 예

```
OBJECTS = main.o foo.o bar.o
foobar : $(OBJECTS)
        gcc -o foobar $(OBJECTS)
main.o : main.c
        gcc -c main.c
foo.o : foo.c
        gcc -c foo.c
bar.o : bar.c
        gcc -c bar.c
clean :
        rm -f foobar $(OBJECTS)
```

Make

내부 매크로

매크로 이름	설 명
$<	의존 파일 중 첫째 파일의 이름
$*	확장자를 제외한 현재 대상 파일의 이름
$@	현재 대상 파일의 이름
$?	현재 대상보다 최근에 변경된 의존 파일들
$^	현재 모든 의존 파일들

```
OBJECTS = main.o foo.o bar.o
foobar : $(OBJECTS)
        gcc -o $@ $^
main.o : main.c
        gcc -c $<
foo.o : foo.c
        gcc -c $<
bar.o : bar.c
        gcc -c $*.c
clean :
        rm -f foobar $(OBJECTS)
```

Make

- 규칙
  - 명시적(**explicit**) 규칙 – 의존 관계를 명시하여 언제 어떻게 대상을 만들어야 하는지를 지정
  - 암시적(**implicit**) 규칙 – 특정 종류의 대상 파일을 언제 어떻게 만들어야 하는지를 암시적으로 지정
    - 예) 접미사 규칙 – **".c"** 확장자의 **C** 소스 파일로부터 같은 이름의 **".o"** 오브젝트 파일 생성
    - 사용 예

```
OBJECTS = main.o foo.o bar.o
CC = gcc
CFLAGS = -g -O2
TARGET = foobar

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

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

Make

▣ 규칙 확인

◈ "make –p" 명령 수행

```
[cprog2@seps5 ch3]$ make -p
# GNU Make version 3.79.1, by Richard Stallman and Roland McGrath.
# Built for i386-hancom-linux-gnu
# Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 2000
#       Free Software Foundation, Inc.
.....
#         :
.c.o:
#  Implicit rule search has not been done.
#        ┌          ↑        .
#  File has not been updated.
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
...
[cprog2@seps5 ch3]$
```

Make

▣ 접미사 규칙(suffix rule)

◈ 예) C 소스 파일(.c)로부터 같은 이름의 오브젝트 파일(.o) 생성

```
.c.o:
        gcc $(CFLAGS) -c $<
```

◈ 사용 예

```
OBJECTS = main.o foo.o bar.o
CC = gcc
CFLAGS = -g -O2
TARGET = foobar

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

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

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

Make

▣ 확장 패턴 규칙(extended pattern rule)

- ◈ **GNU make** 는 대상 파일 이름 매칭(%)이 가능한 규칙 지원
- ◈ 예) C 소스 파일(%.c)로부터 같은 이름에 **"_obj"** 가 붙은 오브젝트 파일(%_obj.o) 생성

```
%_obj.o: %.c
        gcc $(CFLAGS) -c -o $@ $<
```

```
OBJECTS = main_obj.o foo_obj.o bar_obj.o
CC = gcc
CFLAGS = -g -O2
TARGET = foobar

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

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

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

프로그램 개발 보조 도구

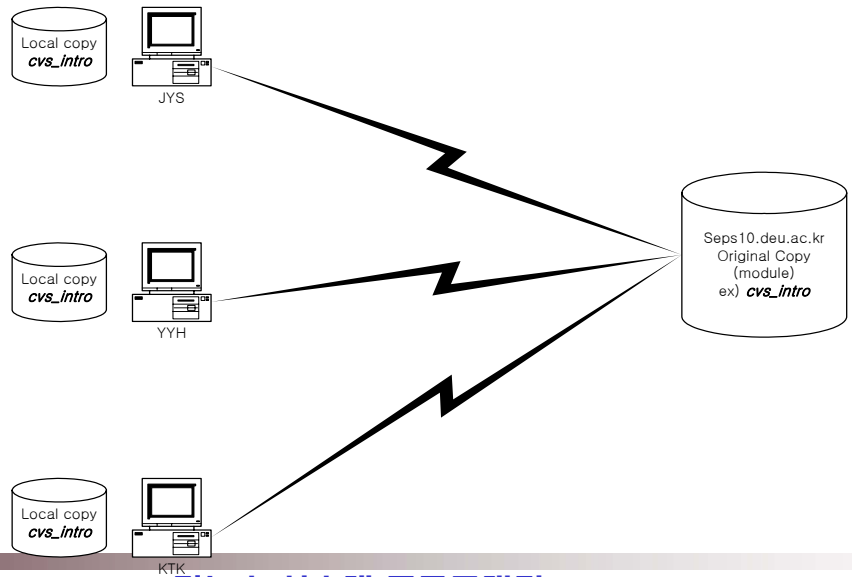
- **gprof**
  - 프로그램 내에서 함수가 몇 번이나 호출되는지, 특정 함수의 수행 시간이 어느 정도 소요되는지를 파악할 수 있다.

- 소프트웨어 관리 소스 생성 도구
  - **GNU autotools (autoconf, automake, libtool)**
  - **Kdevelop** 등의 통합 개발 환경에서 이용

- 소스코드 관리 시스템
  - **SCCS – UNIX** 의 소스 코드 관리 프로그램
  - **RCS** – 텍스트 기반 프로그램 수정 및 버전 관리 시스템
  - **CVS** – 편리한 소스 코드 관리 시스템**(WinCVS,** 원도우버전**)**
  - **Microsoft Visual SourceSafe (VSS)**

CVS

- **CVS (Concurrent Versions System)**
  - 각종 파일의 버전을 쉽게 관리할 수 있도록 하는 도구
- 장점
  - 여러 단계로 파일을 저장 가능
  - 여러 프로그래머가 동시에 같은 파일 작업 가능
  - 네트워크를 통해 원격으로 작업 가능
  - 공동 개발 프로젝트에 유용하게 사용 가능
    - 대부분의 공개 프로젝트에서 사용 : **Apache server, Mozilla**



CVS 동작 방식

- **CVS 프로젝트 수행 절차**
  - 저장소초기화 **(CVS 관리자)**
    - 저장소 **(Repository) :** 파일을 보관할 장소
  - 프로젝트 초기화 **(프로젝트 관리자)**
  - 반복 개발 작업
    - 작업 공간 마련 **(개발자)**
    - 실제 작업 **(개발자)**
    - 자신의 작업 저장 및 타 개발자 작업 가져옴 **(개발자)**
- **CVS 사용법**
  - http://wiki.kldp.org/wiki.php/DocbookSgml/CVS_Tutorial-KLDP

저장소 설정

초기화

저장소 위치가 **/home/cvs** 인 경우

```
$ cvs –d /home/cvs init
```

계정 및 디렉토리 권한 설정

사용자 계정을 **cvs** 그룹으로 설정

- **groupadd** 명령 또는 **/etc/group** 파일 수정

```
cvs:*:60:libero,user1,user2…
```

디렉토리 권한 허용

```
$ chgrp -R cvs /home/cvs
$ chmod ug+rwx /home/cvs /home/cvs/CVSROOT
```

저장소 이용

▣ 네트워크 설정

◈ Xinetd 사용시 cvspserver 설정

- ➢ /etc/xinetd.d/cvspserver 생성
- ➢ "service xinetd restart" 수행

```
service cvspserver
{
        disable         = no
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
        server          = /usr/bin/cvs
        server_args     = -f --allow-root=/home/cvs pserver
        log_on_failure  += USERID
}
```

▣ 모든 CVS 명령은 저장소의 위치를 통해 실행

◈ 저장소 위치 지정 : CVSROOT 환경변수 설정

```
$ export CVSROOT=/home/cvs
```

◈ rsh 또는 ssh 를 사용하는 경우

```
$ cvs –d :ext:userid@hostname:/home/cvs 명령
```

◈ 네트워크 암호 인증 방식

```
$ export CVSROOT=:pserver:userid@hostname:/home/cvs
$ cvs login
```

프로젝트 초기화

▣ 프로젝트 추가 (import)

◈ 새로운 프로젝트를 시작한 뒤 처음으로 **CVS Repository**에 프로젝트를 추가할 때 사용

◈ 생성한 프로젝트 디렉토리 안에서 다음 명령을 수행

```
$ cvs import –m “메시지이름” 프로젝트이름 vendor_tag release_tag
```

◈ 간단한 예

```
$ ls
Makefile hello.c
$ cvs import –m “프로젝트 시작” test hello start
N test/Makefile
N test/hello.c

No conflicts created by this import
$
```

Makefile

```
hello: hello.c
	$(CC) -o hello hello.c
```

hello.c

```
#include <stdio.h>

int main()
{
	printf(“Hello, World₩n”);
}
```

프로젝트 진행

▣ 작업 공간 생성 (checkout)

◈ **CVS Repository**에 저장되어 있는 프로젝트 파일들을 작업 디렉토리로 가져옴

◈ 다음 명령을 수행

```
$ cvs checkout 프로젝트이름
```

◈ 간단한 예

```
$ cvs checkout test
cvs checkout: Updating test
U test/Makefile
U test/hello.c
$ ls test
CVS Makefile hello.c
```

프로젝트 진행

▣ 작업 내용의 저장 (commit)

◈ 변경된 프로젝트 파일을 CVS Repository에 저장

- ➢ 전체 또는 원하는 파일만 저장 가능

◈ 다음 명령을 수행

```
$ cvs commit –m “메시지” [파일이름]
```

◈ 간단한 예 : hello.c 파일 수정

```
#include <stdio.h>

int main()
{
        printf(“Hello, World₩n”);
        printf(“How are you?₩n”);
}
```

```
$ cvs commit –m “인사말 추가” hello.c
Checking in hello.c;
/home/cvs/test/hello.c,v  <--  hello.c
new revision: 1.2; previous revision: 1.1
Done
$
```

프로젝트 진행

▣ 프로젝트 갱신 (Update)

◈ **CVS Repository**에서 갱신된 내용을 가져옴

◈ 다음 명령을 수행

```
$ cvs update
```

◈ 간단한 예 :

```
$ cvs update
cvs update: Updating .
$
```

➢ 다른 개발자가 **hello.c** 파일 수정

```
#include <stdio.h>

Int main()
{
        /* 주석 추가 */
        printf("Hello, World₩n");
        printf("How are you?₩n");
}
```

```
$ cvs update
cvs update: Updating .
RCS file: /home/cvs/test/hello.c,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into hello.c
M hello.c
$
```

프로젝트 진행

▣ 충돌(conflict)의 해결

◈ 둘 이상의 개발자가 같은 부분을 수정할 때 충돌 발생

◈ CVS 는 충돌 사실을 개발자에게 알려줌

◈ 간단한 예 :

➢ 다른 개발자가 **hello.c** 파일 수정

```
$ cvs update
cvs update: Updating .
RCS file: /home/cvs/testj/hello.c,v
retrieving revision 1.3
retrieving revision 1.4
Merging differences between 1.3 and 1.4
into hello.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in hello.c
C hello.c
$ cat hello.c
```

```
#include <stdio.h>

Int main()
{
        /* 주석 추가 */
        printf("Hello, World₩n");
        printf("Welcome₩n");
}
```

```
#include <stdio.h>

Int main()
{
        /* 주석 추가 */
        printf("Hello, World₩n");
<<<<<<< hello.c
        printf("How are you?₩n");
=======
        printf("Welcome₩n");
>>>>>>> 1.4
}
```

➢ 수정한 후 다시 **commit**

프로젝트 진행

▣ 파일 추가 (Add)

- ◈ 프로젝트에 추가된 파일을 **CVS Repository**에 추가할 때 사용
- ◈ 명령 수행 후 **commit** 해야 함

```
$ cvs add 파일또는폴더이름
```

▣ 파일 삭제 (Remove)

- ◈ 프로젝트에서 먼저 파일을 삭제한 뒤 **cvs**의 **remove** 명령을 수행
- ◈ 디렉토리 삭제의 경우 모든 파일을 삭제한 후 **"cvs update –P"**
- ◈ 명령 수행 후 **commit** 해야 함

```
$ cvs remove 파일이름
```

▣ 작업 기록 열람 (log)

- ◈ 그 동안 작성한 작업 기록 열람

```
$ cvs log 파일또는폴더이름
```

프로젝트 진행

태그 달기 (Tag)

◈ 프로젝트에 태그를 단다

```
$ cvs tag 태그이름
```

가지 생성 (Branch)

◈ 가지를 생성한다.(Branch)

```
$ cvs tag –b 가지이름
```

배포 (Export)

◈ 지정한 태그나 날짜 상태의 프로젝트 파일들을 배포하기 위해 내부에 CVS폴더가 없는 상태로 가져온다.

◈ 태그를 지정할 경우

```
$ cvs export -r Release-1_0 -d project-1.0 project
```

◈ 날짜를 지정할 경우

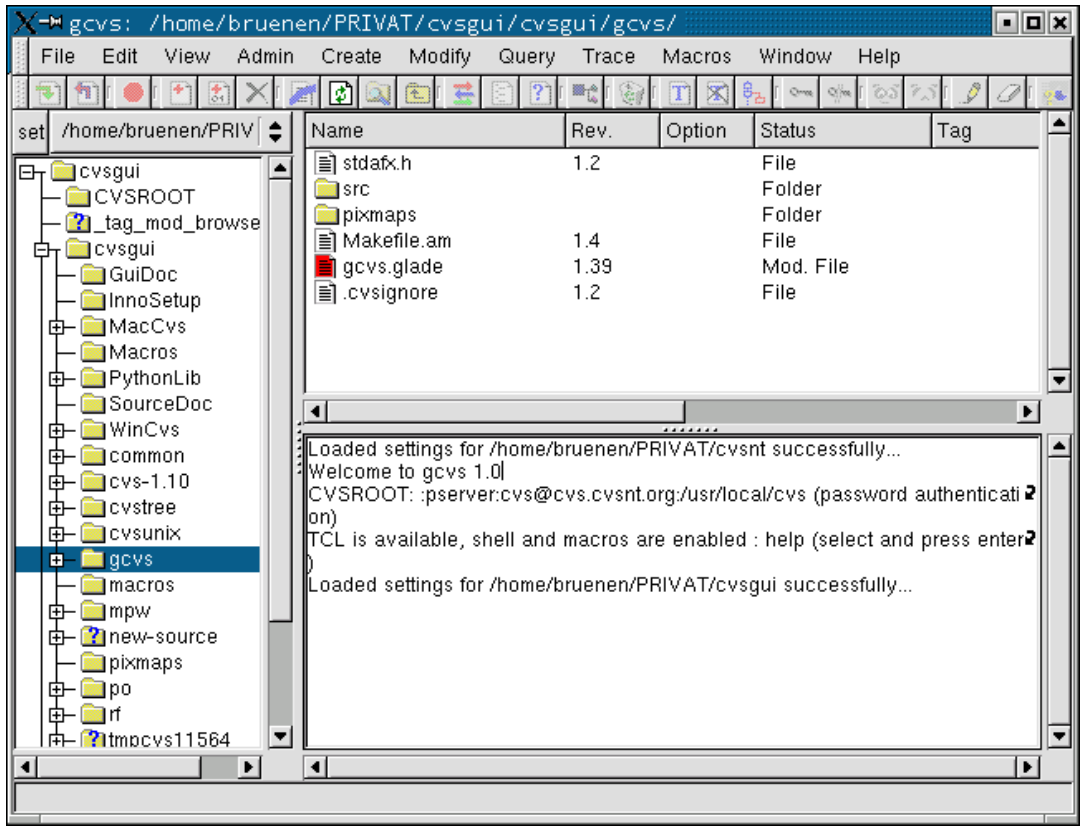
```
$ cvs export -D "2003-12-01 18:00" -d project-20031201 project
```

◈ 현재 날짜를 지정할 경우

```
$ cvs export -D "now" -d project-current project
```

gCVS

- gCVS
  - GNOME 라이브러리 기반 GUI 인터페이스 제공
  - http://wincvs.org



윈도우 버전 소프트웨어

- **Wincvs**
  - **WinCvs13b17-2.zip: recommended install version**
- **Winmerge**
  - **An Open Source visual text file differencing and merging tool for Win32 platforms**
  - **Highly useful for determing what has changed between project versions, and then merging changes between versions**
  - **WinMerge202-exe.zip**
- **Python**
  - **Python-2.3.4.exe**