컴파일과 링킹 – PART I
(COMPILE & LINKING)

안양대학교 컴퓨터공학과

gcc, 컴파일 과정에서 cpp, cc1, as, ld 의 역할

내용

- C 소스 전체 컴파일 과정
- 전처리(preprocessing)
- 컴파일링(compiling)
- 어셈블링(assembling)
- 링킹(linking)

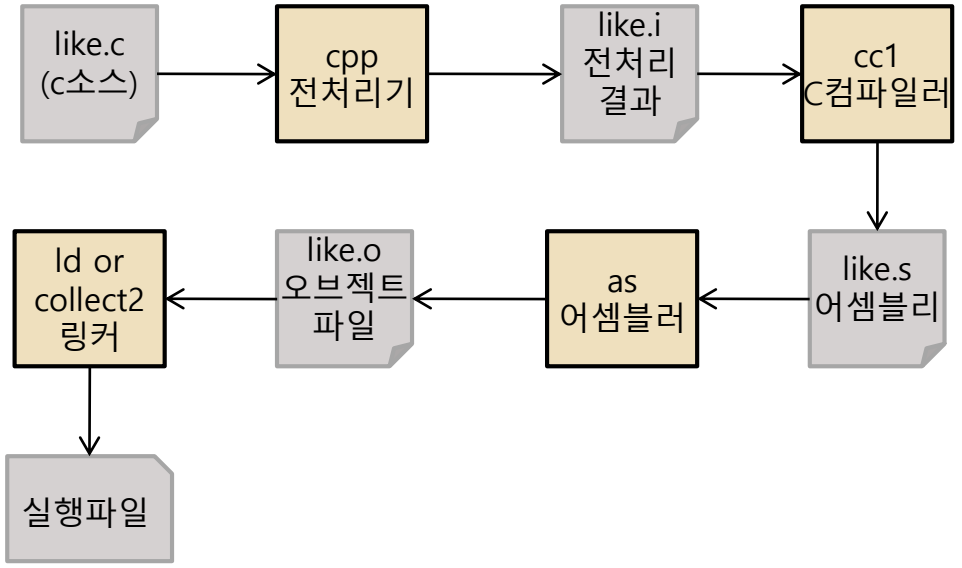
gcc의 특징

- gcc란
  - GNU 프로젝트의 일환으로 만들어진 C컴파일러
  - GNU Compiler Collection
    - 다양한 언어 컴파일: C, C++, Fortran, Pascal, Ada, Objective-C, ..
    - 다양한 아키텍처 지원: ARM, DEC, AVR, i386, PPC, SPARC, M68XX
  - gcc는 front-end 소프트웨어, 일종의 구동기(driver)
- gcc의 구성 요소
  - 전처리기(cpp), 컴파일러(cc1, cc1obj, cc1plus, f771, jc1), 어셈블러(as), 링커(ld)

전체 컴파일 과정 (1)

구성 요소들의 위치

- /usr/libexec/gcc/i686-redhat-linux/4.4.3: cc1, cc1plus, collect2
- /usr/bin: cpp, ld, as



전체 컴파일 과정 (2)

hello.c

```
#include "hello.h"
int main() {
    hello_linux();
    hello_win32();
    return 0;
}
```

hello.h

```
void hello_linux();
void hello_win32();
```

linux.c

```
#include <stdio.h>
void hello_linux() {
    printf("Hello Linux!!\n");
}
```

win32.c

```
#include <stdio.h>
void hello_win32() {
    printf("Hello Windows!!\n");
}
```

가장 간단한 컴파일

```
$ gcc hello.c linux.c win32.c
$ ls -F
a.out* hello.c hello.h linux.c win32.c
$ a.out
Hello Linux!!
Hello Windows!!
$
```

실행 파일 이름 지정

```
$ gcc -o hello hello.c linux.c win32.c
$ ls -F
hello* hello.c hello.h linux.c win32.c
$ hello
Hello Linux!!
Hello Windows!!
$
```

중간 결과물 확인하기

중간 결과물

- 중간 결과물들을 지우지 말고 저장하기
- $ gcc -v --save-temps -o like like.c
- v: verbose, 컴파일되는 과정을 표준출력
- --save-temps: 컴파일과정에서 생성되는 중간 결과물을 지우지 않고 현재 디렉터리에 저장

저장된 파일들

- 실행파일, 소스, 전처리된 파일, 목적파일, 어셈블리

```
$ ls
like.c
$ gcc -v --save-temps -o like like.c
$ ls -F
like* like.c like.i like.o like.s
```

cpp에 의한 전처리 (1)

전처리

like.h

```
struct person {
    int age;
    char *name;
};
```

like.c

```
#include "like.h"
#define YAGE  20
int main()
{
  struct person p = {25, "steven"};
  if( p.age > YAGE ) /* compare */
    printf("I like %s.\n", p.name);
  return 0;
}
```

```
$ cpp -o like.i like.c
(혹은 gcc -E -o like.i like.c)
$ more like.i
# 1 "like.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "like.c"
# 1 "like.h" 1
struct person {
 int age;
 char *name;
};
# 2 "like.c" 2

int main()
{
 struct person p = {25, "steven"};
 if( p.age > 20 )
  printf("I like %s.\n", p.name);
 return 0;
}
```

cpp에 의한 전처리 (2)

헤더파일 삽입

- 헤더 파일의 종류: 표준 헤더 파일, 사용자 정의 헤더 파일
- 전처리기가 표준 헤더파일을 찾는 기본 경로
  - / usr/local/inlcude, /usr/lib/gcc/i686-redhat-linux/4.4.3/include, /usr/include
- 전처리기가 사용자 정의 헤더 파일을 찾는 기본 경로
  - 소스코드가 존재하는 현재 경로, 표준 헤더파일 경로

hello1.c

```
#include <stdio.h>
#include <hello.h>
int main()
{ printf("hello"); }
```

hello2.c

```
#include "stdio.h"
#include "hello.h"
int main()
{ printf("hello"); }
```

```
$ gcc -o hello1 hello1.c
hello1.c:2:19: error: hello.h: 그런 파일이나 디렉터리가 없습니다.
```

cpp에 의한 전처리 (3)

- 기본 경로가 아닌 경로에 존재하는 헤더 파일

```
$ tree
.
|-- inc
|   `-- like.h
`-- like.c
$ cpp -o like.i like.c
like.c:1:18: error: like.h: 그런 파일이나 디렉터리가 없습니다.
$ cpp -I./inc -o like.i like.c
$
```

like.h

```
struct person {
    int age;
    char *name;
};
```

- 또다른 방법은 명시적 경로로 include

like.c

```
#include <stdio.h>
#include "inc/like.h"
#define YAGE  20
int main()
{
   :
```

like.c

```
#include <stdio.h>
#include "like.h"
#define YAGE  20
int main()
{
  struct person p = {25, "steven"};
  if( p.age > YAGE ) /* compare */
    printf("I like %s.\n", p.name);
  return 0;
}
```

cpp에 의한 전처리 (4)

명령 옵션으로 매크로 지정

- 명령 옵션 '-D'를 사용하여 명령 프롬프트에서 명령시 매크로를 정의

like.c

```
#include <stdio.h>
#include "like.h"
int main()
{
  struct person p = {25, "steven"};
  if( p.age > YAGE ) /* compare */
    printf("I like %s.\n", p.name);
  return 0;
}
```

```
$ ls
like.c like.h
$ cpp -o like.i like.c
$ cat like.i
   :
   :
if( p.age > YAGE )
  printf("I like %s.\n", p.name);
 return 0;
}
$ cpp -DYAGE=20 -o like.i like.c
   :
   :
if( p.age > 20 )
  printf("I like %s.\n", p.name);
 return 0;
}
```

cpp에 의한 전처리 (5)

- 조건부 컴파일: 이식성(portability)이 높은 코드 작성을 위한 선택적 컴파일(전처리)

clear.c

```
#include <stdio.h>
#include <stdlib.h>
#ifdef __WINDOWS__
#include <conio.h>
#endif

int main()
{
#ifdef __LINUX__
    system("clear");
#elif __WINDOWS__
    system("cls");
#endif
    return 0;
}
```

```
$ cpp -D__LINUX__ -o clear.i clear.c
$ cat clear.i
   :
   :
int main()
{
    system("clear");
    return 0;
}
$
```

cpp에 의한 전처리 (6)

- 디버깅 코드의 삽입

debug.c

```
#include <stdio.h>
int func(int a, int b)
{
    int c;
    c = (a++)+(++b);
#ifdef DEBUG
    printf("a=%d b=%d c=%d\n",a,b,c);
#endif
}
int main()
{
    int i, j;
#ifdef DEBUG
    printf("i=%d, j=%d\n", i, j);
#endif
    i = 5, j = 8;
    printf("result=%d\n",func(i,j));
    return 0;
}
```

```
$ cpp -DDEBUG -o debug.i debug.c
$ cat debug.i
    :
    :
int func(int a, int b)
{
    int c;
    c = (a++)+(++b);
    printf("a=%d b=%d c=%d\n",a,b,c);
}
int main()
{
    int i, j;
    printf("i=%d, j=%d\n", i, j);
    i = 5, j = 8;
    printf("result=%d\n",func(i,j));
    return 0;
}
$
```

중간 결과물을 이용한 디버깅 (1)

파일 열기

- 전처리된 파일은 언제 볼 필요가 있을까?

my.h

```
struct my_struct {
    int a;
} my;

#define my    MY
```

my.c

```
#include "my.h"
int main() {
    my.a = 0;
    printf("You! my.a = %d\n", my.a);
    return 0;
}
```

```
$ gcc --save-temp -o my my.c
my.c: In function 'main':
my.c:4: error: 'MY' undeclared (first use in this function)
my.c:4: error: (Each undeclared identifier is reported only once
my.c:4: error: for each function it appear in.)
my.c:5: warning: incompatible implicit declaration of built-in function 'printf'
$
```

중간 결과물을 이용한 디버깅 (2)

전처리된 파일

```
$ cpp -o my.i my.c      or gcc -E
my.c
$ more my.i
# 1 "like.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "my.c"
# 1 "my.h" 1
struct my_struct {
  int a;
} my;
# 2 "my.c" 2
int main()
{
  MY.a = 0;
  printf("You! my.a = %d\n", MY.a);
  return 0;
}
$
```

컴파일

컴파일이란

- 전처리된 코드 파일을 입력으로 하여 어휘분석, 구문분석, 의미분석, 중간언어 생성, 최적화 과정을 거쳐 최종적으로 어셈블리 코드를 생성
- 이 과정에서 문법 오류가 검사됨

컴파일 명령

- 컴파일(까지)만 수행하기 위한 명령

```
$ ls
like.c
$ /usr/libexec/gcc/i686-redhat-linux/4.4.3/cc1 like.c
혹은 $ gcc -S like.c
$ ls
like.c like.s
```

```
$ ls
like.i like.c
$ /usr/libexec/gcc/i686-redhat-linux/4.4.3/cc1 like.i
혹은 $ gcc -S like.i
$ ls
like.i like.c like.s
```

컴파일과 링킹

컴파일 – 경고 옵션

경고 옵션

- 문법 오류 검사시 옵션을 조정하여 경고 수위를 조절
- 일반적으로 사용되는 옵션: -Wall –Wextra (-W와 동일)
  - -Wall: 대부분의 일반적인 문제에 대해 경고
  - -Wextra: -Wall이 보고하지 않는 덜 심각한 문제에 대해 경고, 기술적으로는 문제가 되지 않지만 잠재적으로 문제로 발전할 가능성이 있는 것에 대해 경고

foo.c

```
#include <stdio.h>
int main()
{
    unsigned int x = 1;
    int y = -2;
    if( y < x ) x += 2;
    else x += 1;
    printf("x=%d\n",x);
}
```

```
$ gcc -S foo.c        // no errors
$ gcc -Wall -S foo.c
foo.c: In function `foo':
foo.c:6: warning: control reaches end of non-void function
$ gcc -Wextra -S foo.c
foo.c: In function `foo':
foo.c:4: warning: comparison between signed and unsigned integer expression
$ gcc -Wall -Wextra -S foo.
```

컴파일 – 최적화 옵션

- 최적화의 두 가지 관점
  - 실행 파일의 크기를 줄여 메모리와 하드디스크 공간 절약
  - 실행 속도를 향상시키는 것
- 최적화 수준에 따른 옵션
  - -O0: 최적화를 수행하지 않음, 기본
  - -O1: 생성되는 기계어코드(또는 실행파일)를 가능한 작게 하면서 컴파일 시간이 오래 걸리지 않는 범위 내에서 최적화
  - -O2: 실행코드가 가능한 빠르게 수행되지만 코드의 크기가 커지지 않는 범위 내에서 최적화
  - -O3: '-O2'에서 사용되는 모든 최적화 기능을 다 사용하지만, 코드 크기가 증가되는 최적화 기능 제외, 따라서 역으로 명령코드 개수가 증가하여 경우에 따라 느려질 수도 있음
  - -Os: 크기의 최적화, 임베디드 시스템에서 주로 사용

어셈블링

어셈블링이란

- 생성된 어셈블리 코드를 목적코드(relocatable object code; 재배치가능 목적 코드)로 변환하는 과정

어셈블링 명령

- 어셈블링만 수행하기 위한 명령

```
$ ls
hello.c hello.s
$ as -o hello.o hello.s
$ ls
hello.c hello.o like.s
```

-o 옵션이 없으면 a.out이라는 이름으로 목적 파일 생성

```
$ ls
hello.c hello.s
$ gcc -c hello.s
$ ls
hello.c hello.o hello.s
```

- 어셈블링까지만 수행하기 위한 명령

```
$ ls
hello.c
$ gcc -c hello.c
$ ls
hello.c hello.o
```

목적 코드

- 어셈블된 목적코드는 완전한 기계어 코드일까?

test1.c

```
int func3();
extern int mydata;
int func2()
{
    ...
}
int func1()
{
     ...
     func2();
     func3();
     ...
     i = mydata+3;
     ...
     printf(...);
     ...
}
```

test2.c

```
int mydata = 3;
int func3()
{
    ...
}
```

as → test2.o

as → test1.o

test1.o에서 symbol mydata와 func3, printf는 주소가 결정되었을까?

- 소스코드 파일별로 서로 개별적으로 목적코드가 생성. 따라서 외부 참조 심볼의 주소를 결정할 수 없음
- 심볼의 주소 결정은? 링킹(linking)

*.o 파일 보기 명령

```
$ xxd *.o (헥사 덤프)
$ readelf -a (-s) *.o
$ objdump -S *.o
```

링킹(Linking) (1)

링킹이란

- 하나 이상의 목적 파일과 라이브러리 파일을 외부 참조 심볼 주소 결정(symbol reference resolving) 과정을 거쳐 기계어로 된 하나의 파일을 생성하는 과정
- 링킹을 수행하는 프로그램: ld 혹은 collect2

링킹 명령

```
$ ls
hello.c
$ gcc -o hello hello.c
$ ls -F
hello* hello.c
```

```
$ ls
hello.c hello.o
$ gcc -o hello hello.o
$ ls -F
hello* hello.c hello.o
```

링킹(Linking) (2)

라이브러리 링킹

- 링킹 명령시 표준 함수가 정의된 라이브러리 파일(libc.a, libc.so)은 명시하지 않아도 링커(linker)가 알아서 링킹 수행
  - 링킹하려는 라이브러리가 libc가 아닌 경우, -l라이브러리명

찾는 디렉터리

- 라이브러리는 기본적으로 /usr/lib/gcc/i686-redhat-linux/4.4.3, /usr/lib 디렉터리에서 찾음
- 기본 검사 디렉터리 확인: gcc --print-search-dirs
  - 링킹하려는 디렉터리가 다른 위치에 있다면, -L디렉터리

```
$ gcc -o like like.c -lm
$ gcc -o like like.o -L/opt/lib -llike
```

요약

- 컴파일러 구성 요소
  - cpp, cc1, as, collect2, ld
- 전처리 옵션
  - -E, -I, -D
- 컴파일 옵션
  - -S, -Wall, -Wextra, -O0, -O1, -O2, -O3, -Os
- 어셈블 옵션
  - -c
- 링킹 옵션
  - -l, -L
- 기타 옵션
  - --save-temps, -v, -o
  - --print-search-dirs