

## 제3장 C 프로그래밍 환경

# 학습 목표

---

- 문서 편집 : vi, geidt
- C 컴파일러 사용: gcc
- 컴파일 자동화: make
- 디버깅: gdb
- 통합개발환경: Eclipse
- 라이브러리 관리: ar
- 소스 관리: ctags
- 형상 관리: CVS, SVN, git



## 3.1 컴파일러

# 컴파일러

---

- 유닉스 계열 운영체제의 유틸리티와 상용 프로그램은 대부분 C 언어로 작성
- 유닉스와 리눅스는 운영체제 자체가 C 언어로 작성
- 공개된 C 컴파일러로 gcc (GNU cc) 컴파일러가 널리 사용
  - <http://gcc.gnu.org>



# gcc 컴파일러

---

- gcc 컴파일러(GNU)                      cc 컴파일러(상업용)  
\$ gcc [-옵션] 파일                      \$ cc [-옵션] 파일
- 컴파일  
\$ gcc long.c  
\$ a.out                      // 기본 실행 파일 생성
- -c 옵션                      (목적파일까지 생성)  
\$ gcc -c long.c                      *소스 코드를 object 파일로*
- -o 옵션                      (a.out 대신의 별도의 실행파일 지정)  
\$ gcc -o long long.o  
혹은  
\$ gcc -o long long.c  
\$ ./long                      // 실행 파일



# gcc 컴파일러

- 기타 옵션
  - -O (big O) : 컴파일 최적화 수행
  - -S : 어셈블리 프로그램 확인 (xxx.s) //어셈블리 파일만 생성하고 컴파일 멈춤
  - -l : 특정 라이브러리 링크
    - -lxxx 는 보통 /usr/lib 디렉토리에 있는 libxxx.a 를 링크하라는 의미
    - gcc -o test test.c -lm (libm.a 링크)

\$ cc tabs.c

- c : object file 생성
- o : output file 생성
- g : debugging 정보 첨가

\$ cc calc.c -lm ; math library

\$ cc -O2 ledger.c acctspay.c acctsrc.c ; optimizer

- O0 : 최적화를 수행하지 않는다.
- O1, O2 : 최적화 수행 (O2가 default 옵션)
- O3 : 가장 높은 레벨의 최적화 (사용상 주의 필요)
- Os : 사이즈 최적화

\$ mv a.out tabs ; save a.out file to tabs

```
1  #include <stdio.h>
2
3  #define PI 3.14159
4
5  int main(void) {
6      double radius = 10.0;
7      double area = PI * radius * radius;
8      area = area + radius;
9      area = area * radius;
10
11     int value = 10000;
12     if (0) {
13         printf("value : %d\n", value);
14     }
15
16     int max = 100;
17     for (int i = 0; i < max + 100 200; i++) {
18         printf("%d ", i * 2);
19     }
20     printf("\n");
21
22     printf("Area of circle : %f\n", area);
23     return 0;
24 }
```

# (참고) 컴파일러 최적화

```
1: const int n = 10;
2: int z;
3: for (int k = 0; k < n + 5; ++k) {
4:   int x = sqrt(n);
5:   A[k] = A[k] + (x * k) / 4; >> 2
6:   B[k] = B[k] + (x * k) * simpleCalc(k, i, n);
7:   z = A[k] + B[k];
8: }
9: return;
10: memset(A, 0, sizeof(A));
```

Handwritten annotations on the code:

- Arrow from "register" to line 2: `int z`
- Arrow from "15" to line 3: `n + 5`
- Arrow from "inline" to line 6: `simpleCalc(k, i, n)`
- Arrow from "사용 X" to line 7: `z = A[k] + B[k];`
- Arrow from "도달 X" to line 10: `memset(A, 0, sizeof(A));`

line 1 & line 3 : 상수선언 및 상수 + 5도 상수, 즉  $n+5$ 를 15로 미리 계산

line 3 : 변수  $k \rightarrow$  register int

line 4 :  $x$ 는 loop invariant, `sqrt()` 함수와 함께 loop 밖으로 이동(loop 내부 함수 호출 제거)

line 5 & line 6 :  $(x*k)$  중복 계산 제거

line 5 : "`/4`" 연산은 shift 연산으로 대체 (연산강도 경감)

line 6 : loop 내부의 함수 호출  $\rightarrow$  inline 함수로 대체

line 7 :  $z$  계산 불필요 (사용되지 않는 코드)

line 10 : 도달 불가능 코드 (unreachable code)



# (참고) 컴파일러 최적화

- 상수 대체 (Constant Folding)
  - 같은 표현이 여러 곳에서 반복되는 경우, 중복 표현 제거

```
#include <stdio.h>
```

```
void main() {  
    int a, b, c;  
    a = 10;  
    b = a + 20;  
    c = b + 30;  
    printf("%d", c);  
}
```

```
#include <stdio.h>
```

```
void main() {  
    int a, b, c;  
    scanf("%d %d", &a, &b);  
    c = a + b;  
    printf("%d", c);  
    if((a + b) > 10)  
        c = (a + b) * 2;  
    else  
        c = (a + b) * 5;  
}
```

# (참고) 컴파일러 최적화

- 데드 코드 제거 (Dead Code Elimination)

- 코드 상 절대 실행되지 않는 코드 제거

```
#include <stdio.h>
```

```
void main() {
```

```
    int a, b = 0;
```

```
    scanf("%d", &a);
```

```
    if(a < 0 && a > 0) // 도달할 수 없는 조건, 데드 코드
```

```
        b = 10;
```

```
}
```

- 연산 강도 경감 (Strength Reduction)

- \* 연산자 vs. + 연산자

```
#include <stdio.h>
```

```
void main() {
```

```
    int i, j;
```

```
    for(i = 0; i < 10; i++) {
```

```
        j = i * 5;
```

```
        printf("%d\n", j); } } / printf( )  
                               j += 5;
```

# gcc 컴파일러

---

- 컴파일과정 중 생성될 수 있는 파일들
  - C 소스코드 (.c)
    - 전처리기(cpp)
  - 전처리된 C 소스코드 (.i)
    - C 컴파일러(cc1)
  - 어셈블리 코드 (.s)
    - 어셈블러(as)
    - -S 사용으로 .s 파일 생성
  - 오브젝트코드 (.o)
    - 바이너리 파일
    - 라이브러리(libc.a, libm.a ...)
  - 최적화된 오브젝트 코드(.o)
    - 실행 파일 최적화 (속도, 크기 등)
  - 실행파일(a.out)



# gcc 컴파일러

---

- 중간 결과의 저장 및 확인

```
vi ex6.c
```

```
#include <stdio.h>
```

```
#define VALUE 1999
```

```
int main(int argc, char **argv)
```

```
{
```

```
    printf("Hello  
World! %d\n", VALUE);
```

```
    return 0;
```

```
}
```

```
[sugar@hussein bit]$ gcc -save-temps ex6.c
```

```
[sugar@hussein bit]$ ls
```

```
a.out ex6.i ex6.s ex6.c ex6.o
```

```
[sugar@hussein bit]$ ./a.out
```

```
Hello World! 1999
```

```
[sugar@hussein bit]$ tail -7 ex6.i
```

```
int main(int argc, char **argv)
```

```
{
```

```
    printf("Hello World! %d\n", 1999 );
```

```
    return 0;
```

```
}
```



# (요약) gcc 컴파일러 옵션

옵 션	설 명
-O	<ul style="list-style-type: none"><li>. 컴파일 결과로 생성되는 파일의 이름을 명시적으로 지정한다.</li><li>. -o 옵션 뒤에 파일명을 지정한다.</li><li>. 일반적으로 소스 파일의 확장자를 제외한 부분을 실행 파일 이름으로 사용</li></ul>
-C	<ul style="list-style-type: none"><li>. 목적(Object) 파일을 생성한다.</li><li>. filename.o 파일이 생성됨</li></ul>
-I (i 대문자)	<ul style="list-style-type: none"><li>. <b>헤더 파일의 디렉토리 위치</b>를 명시적으로 지정한다.</li><li>. #include "my.h"와 같은 코드를 #include &lt;my.h&gt;로 쓰고 싶을 때, 컴파일시 옵션으로 지정 (참고) "file.h"는 현재 디렉토리에서 찾음. &lt;file.h&gt; 는 /usr/include에서 찾음</li><li>. gcc -o filename file.c -Idirname</li></ul>
-l (L 소문자)	<ul style="list-style-type: none"><li>. 컴 파일에 사용되는 <b>라이브러리</b>를 명시적으로 지정한다.</li><li>. Lib prefix 와 .a 확장자명을 제외하고 사용</li></ul>
-L	<ul style="list-style-type: none"><li>. <b>라이브러리의 디렉토리 위치</b>를 명시적으로 지정한다.</li><li>. 사용자가 별도의 라이브러리를 생성할 경우, /usr/lib에 삽입하는 것보다 별도의 디렉토리로 관리하는 것이 바람직하다.</li><li>. gcc -o filename filename.c -lmy -L.</li></ul>
-D	<ul style="list-style-type: none"><li>. 매크로를 지정한다.</li></ul>

# 다중 모듈 프로그램 (다중 파일 프로그램)

---

- 단일 모듈 프로그램
  - 코드의 재사용(reuse)이 어렵다.
  - 여러 사람이 참여하는 프로그래밍이 어렵다
  - 예를 들어 다른 프로그램에서 copy 함수를 재사용하기 힘들다
- 다중 모듈 프로그램
  - 여러 개의 .c 파일(즉, 다중 파일)들로 이루어진 프로그램
  - 일반적으로 복잡하며 대단위 프로그램의 개발에 적합



# 단일 모듈 프로그램 사례: long.c

---

```
#include <stdio.h>
#define MAXLINE 100
void copy(char from[], char to[]);
char line[MAXLINE]; // 입력 줄
char longest[MAXLINE]; // 가장 긴 줄
/*입력 줄 가운데 가장 긴 줄 프린트 */
main()
{
    int len;
    int max;
    max = 0;
    while (gets(line) != NULL) {
        len = strlen(line);
        if (len > max) {
            max = len;
            copy(line, longest);
        }
    }
}
```

```
    if (max > 0) // 입력 줄이 있었다면
        printf("%s", longest);

    return 0;
}
/* copy: from을 to에 복사; to가 충분히 크
   다고 가정*/
void copy(char from[], char to[])
{
    int i;
    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```



# 다중 모듈 프로그램: 예

---

- main 프로그램과 copy 함수를 분리하여 별도 파일로 작성
  - main.c
  - copy.c
  - copy.h // 함수의 프로토타입을 포함하는 헤더 파일
- 컴파일

```
$ gcc -c main.c
$ gcc -c copy.c
$ gcc -o main main.o copy.o
혹은
$ gcc -o main main.c copy.c
```





# main.c

---

```
#include <stdio.h>
#include "copy.h"
char line[MAXLINE]; // 입력 줄
char longest[MAXLINE]; // 가장 긴 줄
/*입력 줄 가운데 가장 긴 줄 프린트 */
main()
{
    int len;
    int max;
    max = 0;
    while (gets(line) != NULL) {
        len = strlen(line);
        if (len > max) {
            max = len;
            copy(line, longest);
        }
    }
}
```

```
    if (max > 0) // 입력 줄이 있었다면
        printf("%s", longest);

    return 0;
}
```



## copy.c

---

```
#include <stdio.h>
#include "copy.h"
/* copy: from을 to에 복사; to가 충분
   히 크다고 가정*/
void copy(char from[], char to[])
{
    int i;
    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

## copy.h

---

```
#define MAXLINE 100
void copy(char from[], char to[]);
```

이와 같이 세 개의 서로 다른 모듈의 파일로  
구성하여 프로그램 개발 가능



값이 원래 소스코드에 있지만 값을 대입하게 된다면?  
소스코드가 우선

## 심볼 정의 (-DVALUE option)

```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("Hello World! %d\n", VALUE);
    return 0;
}
```

```
[sugar@hussein bit]$ gcc ex2.c
ex2.c: In function `main':
ex2.c:4: `VALUE' undeclared (first use in this function)
```

```
[sugar@hussein bit]$ gcc -DVALUE=1999 ex2.c
```

```
[sugar@hussein bit]$ ./a.out
```

```
Hello World! 1999
```

```
[sugar@hussein bit]$
```

**-D[macro]**

macro를 외부에서 정의할 때 사용  
#define macro 를 추가한 것과 동일

**-D[macro]=[macro\_value]**

#define macro macro\_value를 추가  
한 것과 동일

→ 소스 코드에 명시적으로 수정되는 것이 아니라, 컴파일 단계에 가변적으로 다른 값을 할당 하여 디버깅의 용도로 활용될 수 있음

수정  
변경

# 심볼 정의 활용

---

- 디버깅에 사용

```
[sugar@hussein bit]$ vi ex3.c
```

```
#include <stdio.h>
int main(int argc, char **argv)
{
    #ifdef DEBUG
        printf("DEBUG!\n");
    #endif
    printf("Hello World!\n");
    return 0;
}
```

```
[sugar@hussein bit]$ gcc ex3.c
```

```
[sugar@hussein bit]$ ./a.out
```

```
Hello World!
```

```
[sugar@hussein bit]$ gcc -DDEBUG
ex3.c
```

```
[sugar@hussein bit]$ ./a.out
```

```
DEBUG!
```

```
Hello World!
```

```
[sugar@hussein bit]
```

→ 하나의 소스를 활용하여, 디버깅 단계에 실행 흐름을 조절할 수 있음.  
개발 완료 시 특별한 삭제 작업 없이 올바른 동작 수행



# 참고) 프로그램 코드내에서의 매크로

자주 쓰여

```
#define DEBUG
```

실행 안됨 X

```
int average(int x, int y) 컴파일에 포함
{
#ifdef DEBUG
    printf("x=%d, y=%d\n", x, y);
#endif
    return (x+y)/2;
}
```

```
int average(int x, int y) 컴파일에 포함되지
{
    printf("x=%d, y=%d\n", x, y);
    return (x+y)/2;
}
```

#ifdef는 프로그램을 디버깅할 때 많이 사용

디버깅 단계에서는 여러가지 디버깅 관련 정보를 출력하고,  
제품이 출시될 때에는 디버깅 정보를 정의하지 않음으로 (즉, #define 문장만 제거)  
디버깅 관련 정보 출력이 제품에 포함되지 않도록 함