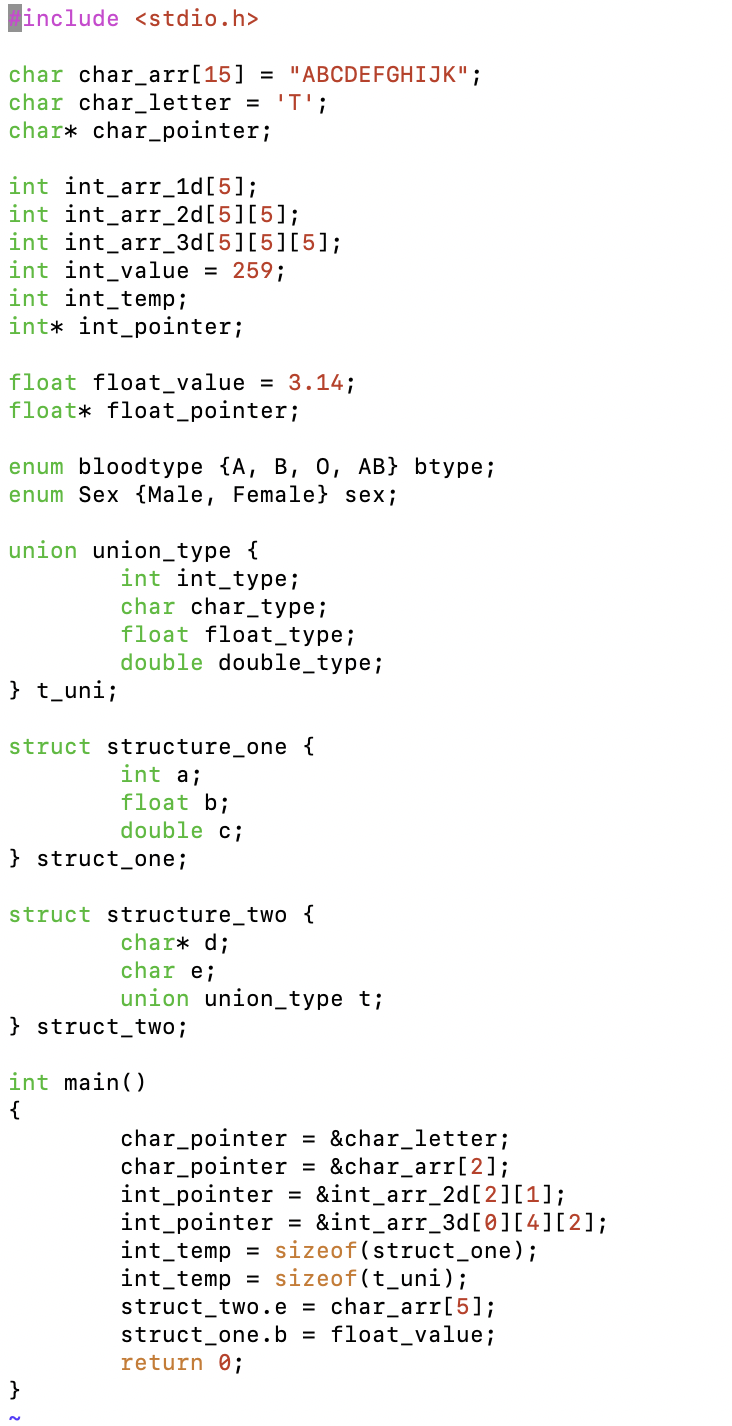
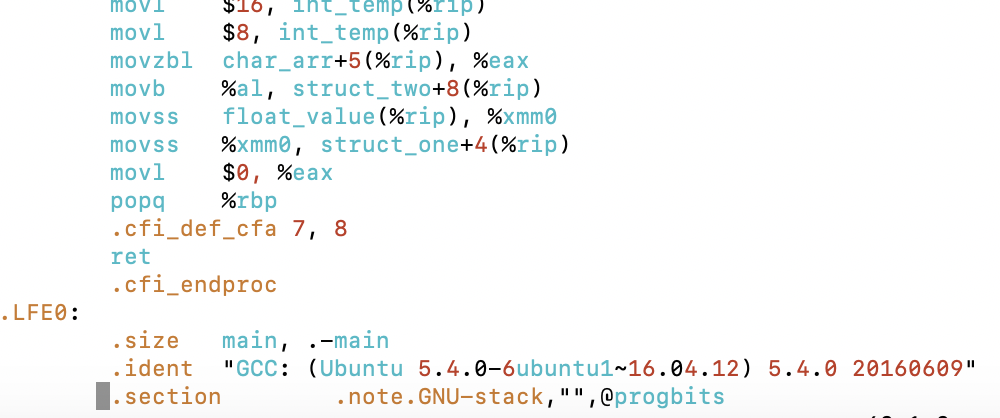
프로그래밍 언어 HW3

20171645 박찬우



**Analysis**





**.file “test.c”** – test.c 파일로부터 생성되었다.

**.global char\_arr** – char\_arr이 global 변수이다.

**.data**

**.align 8** – compiler가 8byte만큼 memory\_allignment해준다.

**.type char\_arr,** @object – char\_arr가 객체이다.

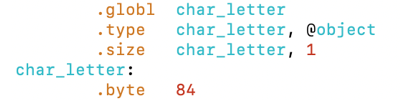
**.size char\_arr, 15** – 1byte 크기의 char 15개가 모였으므로 총 크기는 15byte이다.

**char\_arr:**

**.string “ABCDEFGHIJK”** – 문자열의 내용이 “ABCDEFGHIJK” 이다.

**.zero 3** – 문자열의 길이는 15인데, 내용은 Null문자를 포함해 12자이므로 공백은 3이다.





**.global char\_letter** – char\_letter은 global 변수이다.

**.type char\_letter, @object** – char\_letter은 객체이다.

**.size char\_letter, 1** – char\_letter의 크기는 1byte이다.

**Char\_letter:**

**.byte 84** – char\_letter의 값은 84(T)이다.

**.comm char\_pointer, 8, 8** – BSS 영역에 char\_pointer 라는 symbol을 8byte만큼 할당한다.

**.comm int\_arr\_1d, 20, 16** – BSS 영역에 int\_arr\_1d 라는 symbol을 20byte만큼 할당한다.

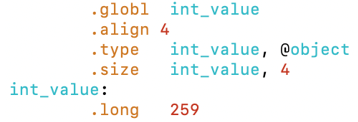
4byte의 int가 [5]만큼이므로 크기는 20byte, int\_arr\_1d 의 address는 16 multiple로 정렬된다.

.**comm int\_arr\_2d,100,32** - Int\_arr\_2d[5][5]는 size가 4\*5\*5 = 100byte이고, address는 32 multiple로 정렬된다.

**.comm int\_Arr\_3d, 500, 32** - Int\_arr\_3d[5][5][5]는 size가 4\*5\*5\*5=500byte고, address는 32 multiple로 정렬된다.





**.global int\_value** – global 변수이다.

**.align 4** – compiler가 4byte만큼 memory allignment해준다.

**.type int\_value, @object** – 객체이다.

**.size int\_value, 4** – int type이니 4byte이다.

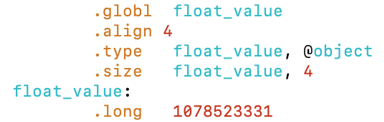
**.long 259** – int\_value의 값이 259이다.

**.comm int\_temp,4,4** – BSS 영역에 int\_temp 라는 symbol을 4byte만큼 할당한다.

**.comm int\_pointer,8,8** – BSS 영역에 int\_pointer 라는 symbol을 8byte만큼 할당한다.





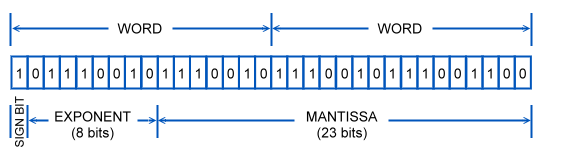
**.global float\_value** – global 변수이다.

**.align 4** – compiler가 4byte만큼 memory allignment해준다.

**.type float\_value, @object** – 객체이다.

**.size float\_value, 4** – size는 4byte이다.

**.long 1078523331** – 아래 그림과 같이 float을 binary string으로 변환하면

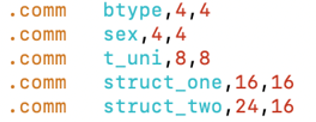
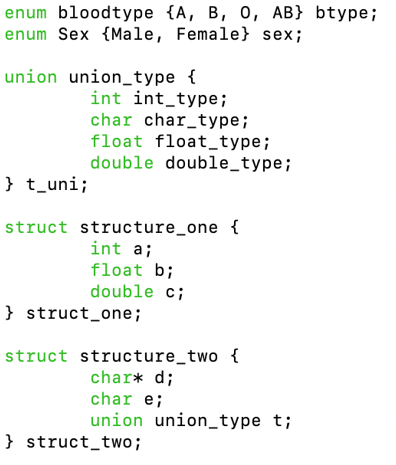


01000000010010001111010111000011 이 되고,

이를 two’s complement representation 하면 1078523331이 된다.



**.comm float\_pointer,8,8** - 앞선 포인터와 마찬가지로, 8byte만큼 BSS 영역에 할당한다.



.**comm btype,4,4**

Enum은 int type이므로 btype 변수는 4byte만큼 BSS 영역에 할당하고,

**.comm sex,4,4**

동일하게 sex 역시 Enum 내용 수와 관계없이 4byte만큼 BSS 영역에 할당한다.

**.comm t\_uni,8,8**

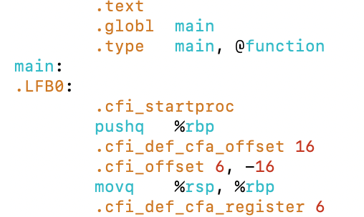
Union\_type의 경우 4개의 구성요소중 가장 크기가 큰건 double type이므로 8byte만큼 할당하고,

**.comm struct\_one,16,16**

Structure\_one은 int, float, double type의 변수를 포함하는 구조체이므로 4+4+8 = 16byte만큼 BSS 영역에 할당한다. 이때 address는 16 multiple로 정렬된다.

**.comm struct\_two,24,16**

Structure\_two는 char pointer type, char type, 앞선 8byte 크기의 union type을 포함한 구조체이므로 8+8+8 = 24byte만큼 BSS 영역에 할당한다. 이때 address는 동일하게 16 multiple로 정렬한다.

**.text** – “text” section은 실행 가능한 코드를 모아놓은 section이다.

**.global main** – main이 global type이다.

**.type main, @function** – main은 function이다.

**Main:** - main 함수의 entry point

**.LFB0:** - 함수 시작을 나타내는 local label

**.cfi\_startproc** – “call frame information” 명령어로 assembler가 dwarf format debugging information을 보내도록 지시한다.

**pushq %rbp** – rbp 레지스터에 stack을 push한다.

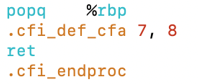
**.cfi\_def\_cfa\_offset 16** – CFA 계산을 위한 offset을 16으로 설정한다.

**.cfi\_offset 6, -16** – rbp 레지스터는 6번이고, 앞에서 push한 위치는 CFA-16이므로 (offset이 16이므로) cfi\_offset에 6(rbp), -16을 저장한다.

**movq %rsp, %rbp** – rsp의 값을 rbp에 저장해 현재 스택 위치를 저장한다.

**.cfi\_def\_cfa\_register 6** – 이제 rsp 대신 rbp를 사용해 CFA를 계산한다.

이렇게 CFA를 rpb로 바꿨으므로, main 함수의 마지막에서는 다음 사진처럼



CFA 계산 규칙을 원래 상태인 rsp(7) + 8 형태로 변경한 뒤, return 한 후 함수를 종료(.cfi\_endproc)하게 된다.





**movq $char\_letter, char\_pointer(%rip)**

movq는 quadword, 즉 64bit 값의 이동 명령이다. movq는 operation code, $char\_letter는 source, char\_pointer(%rip)는 destination이다. 따라서 이는 char\_letter의 주소값을 rip(16) register가 가리키는 char\_pointer에 넣어주는 명령이다.





**movq $char\_arr+2, char\_pointer(%rip)**

위와 동일한 형식의 문장이지만, char\_arr는 char array type이므로, 목표하는 char\_arr[2]는 char\_arr의 주소값에서 2 \* 1(size of char)이므로 source에 $char\_arr+2가 들어오는 모습이다.





**movq $int\_arr\_2d+44, int\_pointer(%rip)**

이 역시 동일하게 int\_pointer에 int\_arr\_2d의 주소값을 넣어주는 명령인데, int\_arr\_2d는 5\*5의 크기를 갖는 array이다. 따라서 int\_arr\_2d[2][1]은 11 \* 4(size of int) 만큼을 int\_arr\_2d의 주소값에 더해야한다. 따라서 source에 $int\_arr\_2d+44가 들어온 모습이다.





**movq $int\_arr\_3d+88, int\_pointer(%rip)**

마찬가지로, int\_arr\_3d는 5\*5\*5 의 array이므로, int\_arr\_3d[0][4][2]는 int\_arr\_3d의 주소값에 22 \* 4, 즉 88을 int\_arr\_3d의 주소값에 더한 것이 source가 됬다.





**movl $8, int\_temp(%rip)**

**movl $16, int\_temp(%rip)**

movl은 movq와 동일한 기능이나, quadword가 아닌 32bit 값의 이동 명령이다. 각 줄의 destination에는 동일하게 int\_temp가 나왔는데, source에 struct\_one의 크기인 8과 t\_uni의 크기인 16이 상수로 그대로 나오는 것을 보아, sizeof 함수는 특별한 과정없이 그대로 상수로 들어가는걸 확인할 수 있다.



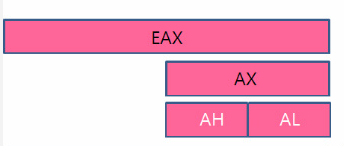


**movzbl = move zero-extended byte to double word**

즉, 무부호 데이터 확장을 해 데이터를 저장할때 쓰인다. 이 경우, char\_arr[5]의 값인 char\_arr+5(%rip)을 eax 레지스터로 옮기는데, char\_arr의 데이터값은 char type이므로 1byte이기 때문에 이를 4byte로 확장해줘야 되기 때문이다.

**movb %al, struct\_two+8(&rip)**

struct\_two.e 는 char 변수를 갖는다. char type의 size는 1byte이고, eax 레지스터의 구조는 아래와 같다.



앞서 EAX 레지스터에 필요한 데이터값을 저장했고, EAX 레지스터는 8byte지만, 전부 필요한게 아니므로 가장 작은 단위인 AL 레지스터를 연산에 사용한다. 이러한 크기에 맞춰 1byte 값의 이동명령인 movb 명령어를 사용한다. 이때 destination은 struct\_two 주소에서 앞선 char\* d의 크기인 8을 더한 struct\_two+8이 됨을 확인할 수 있다.





**movss float\_value(%rip), %xmm0**

movss 명령어는 scalar single-precision floating-point value를 이동시켜주는 명령어로, float\_value에 있는 값을 xmm0 레지스터로 옮겨준다. xmm 레지스터는 0부터 15까지 있으며, 부동소수점을 사용할 때 쓰는 레지스터이다.

**movss %xmm0, struct\_one+4(%rip)**

동일한 movss명령어로, xmm0 레지스터에 있는 값을 struct\_one의 b로 옮겨야 하는데, struct\_one의 b 앞에는 int type data를 갖는 a가 존재하므로, struct\_one의 주소값에 int size인 4를 더한 struct\_one+4가 destination으로 옴을 확인할 수 있다.





**movl $0, %eax** – eax 레지스터를 0으로 set한다. 이는 나중에 ret 명령을 만났을때, 현재 함수의 return value를 eax에 저장하기 때문이다. return 0이므로 eax를 0으로 set한다.

**popq %rbp** – 앞에서 rsp 대신 rbp를 사용하여 CFA를 계산하도록 했으므로, 이를 원래대로 돌려주기 위해 rpb의 값을 pop한다.

**.cfi\_def\_cfa 7,8** - CFA 계산 규칙을 원래 상태인 rsp(7) + 8 형태로 변경한다.

**ret** - return 한다. 이때 return value는 eax 레지스터에 있는 0이다.

**.cfi\_endproc** – 함수(main)를 종료한다.

**.LFE0:** - local label

그 아래는 compile한 환경등을 의미한다. (GCC compiler 사용, Ubuntu OS 어떤 버전에서 실행 등)