C basic language

임베디드스쿨 2기

Lv1과정

2021. 04. 30

김효창

1. TEST

Keyword 또는 Reserved word

return : 함수를 호출한 곳으로 데이터를 전달하고 함수를 끝내는 명령어

void : 함수 정의 시 앞에 붙어서 return 값이 없음을, 파라미터에서는 파라미터가 없음을 의미
void *형태로 사용하여 정해지지 않는 데이터 type 을 의미

sizeof : 변수나 상수 등이 차지하고 있는 크기를 byte로 얻는 명령어

if : 값이 1이면 실행

else : if 값이 0 이면 분기되어 실행

while : 값이 1이면 계속 반복

break : while , do ~ while , for , switch 에서 탈출

continue : while , do ~ while , for 에서 continue 아래 부분 수행하지 않고 다음 회차 실행

case : switch 구성 요소 , 분기로 사용

default : switch 에서 정해진 case 범위가 아닌 분기로 사용

goto : 정해진 lable 로 jump , 반복문 한 번에 탈출하기 위한 용도

struct : 여러 변수들을 하나의 그룹으로 묶어서 사용자 정의 data type 을 만든다.

typedef : 이미 존재하는 data type 를 새로운 이름으로 추가

1. TEST

Function

f(x) = 5x + 1 // f(3) 이면 결과 16을 반환

```
int f(int x)
{
	return 5 * x + 1;
}

int main(void)
{
	int result;
	result = f(3);
}
```

return : 함수를 호출한 곳으로 데이터를 전달하고 함수를 끝내는 명령어

```
return_type function_name(parameter_datatype1 name1, ...)
{
	return value;
}
```

value 의 data type 과 함수의 return_type 의 data type 이 일치해야 한다.
return 은 함수를 실행하다가 특정한 조건에 함수를 끝내고 호출한 함수에게 결과를 전달한다.

1. TEST

```
void function(int x)
{
        int y;

        if(x <= 0){
                return;
        }

        printf("₩n");
}
```

return 없는 함수는 void 로 표시, 중간에 함수를 종료하기 위해 return 을 사용할 수 있다
return type 이 없기 때문에 return; 으로 기술
함수의 끝에는 전달할 데이터가 없기 때문에 return; 붙일 필요 없다

```
int function(int x)
{
        int x[10];
        scanf("%s",&x);

        return 0;
}
```

parameter 가 없는 함수는 void , input 데이터가 필요 없거나 내부에서 얻을 수 있는 경우
return 은 수학적 계산의 결과를 얻거나 정상 처리 되었는지 판단
[정상 : 양수, 0]
[오류 : 음수]
함수 앞 return type 이 int 이므로 정수형 데이터를 return

int function(void) : 함수의 parameter 는 확실히 없다
int function() : 함수의 parameter 가 무엇인지 모른다.

1. TEST

배열

short a[20];
short (자료형) , a (배열 이름) , [10] (배열 크기)
배열 크기의 인덱스는 0 부터 시작.
short 가 20 개 있는 a를 생성. (short 2 Byte , 총 40 Byte)
&a[20] : 배열의 주소 값
배열 이름 = 포인터 상수

a + i = &a[i]
*(a + i) = a[i] = p [i]
int a[10]; a 는 int 100개의 배열
int *a[10]; a 는 포인터 100개의 배열

포인터

참조 연산자 * : 지시하는 주소에 저장된 값을 반환
특정한 타입을 저장할 수 있는 메모리 공간의 주소 값을 저장
데이터 타입 * 포인터 이름 = &변수 이름;
데이터 타입 * 포인터 이름 = 주소 값;
int *p , int 형 주소 값을 가질 수 있는 포인터 변수 p 를 선언
선언만 하고 초기화 되지 않았으므로 의미 없는 값
포인터 변수의 명칭은 p
p 에서 참조할 수 있는 데이터 타입은 int

```
int num[5] = { 0, 1, 2, 3, 4 };
int *p;
p = num;
```
p 는 num[0] 을 항상 지시한다.

배열명 앞에 * 를 붙이면 '포인터 배열'

```
a = *p++; // *(p++) 로 해석
```
p의 주소 값이 사용된 후에 증가된다.
```
a = *(p+1); // p[1]
a = (*p)++; // p 가 지시하는 값 증가
a = (*p)+1 // 값 + 1
a = *++p; // *(++p)
```
p의 값이 증가된 후에 사용된다.
```
a = ++*p; // ++(*p) , 값 + 1
a = *p+=1; // a = *p; , *p = (*p) + 1;
char a[3];
char *p = a;
```
포인터 p 의 주소 값 , p 는 a[0] 을 지시한다.
```
a[0] // == *a, p[0]
a[1] // == *(a+1), p[1]
a[2] // == *(a+2), p[2]
```
포인터와 배열은 같은 의미.

1. TEST

```c
int main(void)
{
    char *a;
    int *b;
    double *c;

    a = (char*)1000;
    b = (int*)1000;
    c = (double*)1000;

    a++;
    b++;
    c++;

    // a: 1001 , b: 1004, c: 1008
    // a+2: 1003 , b+2: 1012, c+2: 1024
    printf("증 가  후 \n a : %d\n b : %d\n c : %d\n", a, b, c);
    printf("+2 증 가 \n a + 2 : %d\n b + 2 : %d\n c + 2 : %d\n", a+2, b+2, c+2);


    return 0;
```

1. TEST

동적 할당

#include 〈stdlib.h〉 선언해야 사용 가능
실행 중에 동적으로 메모리를 할당 , Heap 영역에 할당
함수 원형은 void* malloc(sizeof)
해당 함수 사용 시 #include〈stdlib.h〉 선언
동작은 매개변수에 해당하는 sizeof 크기만큼 메모리를 할당
성공 : 할당한 메모리의 첫 번째 주소 리턴
실패 : NULL 리턴

int *p;
p = (int *)malloc(sizeof(int) * 10); // int형 10개 동적 메모리 할당
(int *) : malloc 의 반환형이 void 이므로 (int *) 형태로 형변환 요청
free(p); // 할당된 동적 메모리 반납
p[i] = array[i]; 동적 메모리 할당하면 포인터는 배열처럼 사용할 수 있다

sizeof(int) : 괄호 안에 자료형 타입을 바이트로 연산
sizeof(int) * 10 : 배열 [ ] 괄호 안의 값과 동일한 크기의 메모리 할당 위해 int*10
malloc 함수는 인수로 할당 받고자 하는 메모리의 크기를 바이트 단위로 전달 받는다.

dangling pointer
메모리가 해제된 곳을 지시하는 포인터 , 계속 남아있다 , 코드 사이즈가 증가하면 문제 발생

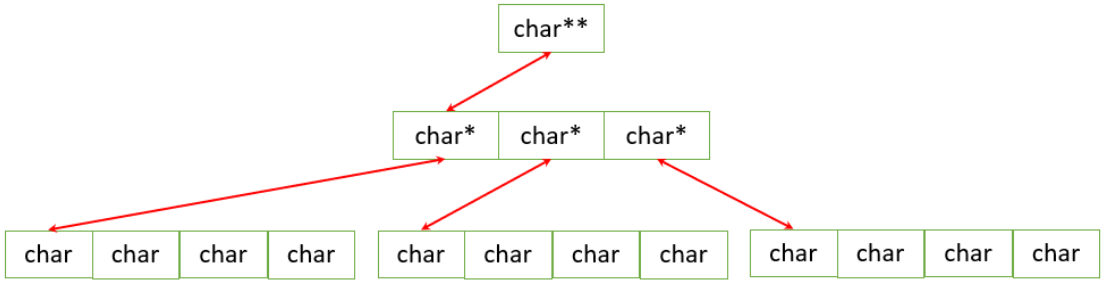
1. TEST

배열 동적 할당

배열을 추가로 동적으로 할당하기 위해서는 각각의 시작 주소 값을 알고 있는 포인터 배열이 필요하고
포인터 배열의 시작 주소 값을 알고 있는 포인터 변수가 필요하다.

동적 할당은 시작 주소 값을 지시할 수 있는 포인터 변수를 만들고 이후에 배열을 추가로 만드는 것

메모리 반납은 역순으로 설정




```
65      66      67      68
69      70      71      72
73      74      75      76
p[0] 대표 주소 안에 들어 있는 값  : 65
p[1] 대표 주소 안에 들어 있는 값  : 69
p[2] 대표 주소 안에 들어 있는 값  : 73
```

```c
int main(void)
{
    char **p;
    int i, j;

    // 동적 할당 선언
    p=(char**)malloc(sizeof(char*)*3);
    *p=(char*)malloc(sizeof(char)*12);

    for(i=1;i<3;i++)
    {
      p[i] = p[0] + 4*i;
    }

    for(i=0; i<12; i++)
    {
        // p[0][i] ==  *(p[0]+i)
        p[0][i]='A'+i;
    }
    for(i=0; i<3; i++)
    {
        for(j=0;j<4;j++)
        {
          printf("%d\t", p[i][j]);
        }
      printf("\n");
    }
    for(i=0; i<3; i++)
    {
        printf("p[%d] 대표 주소 안에 들어 있는 값  : %d\n", i, *p[i]);
    }
    free(*p);
    free(p);
}
```

1. TEST

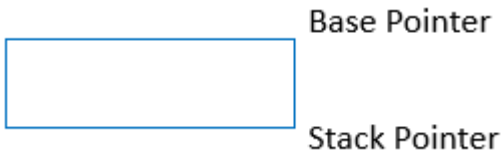
1. C언어는 함수를 호출할 때마다 무엇을 생성하는가 ?
어째서 재귀호출이 일반적인 Loop 보다 성능이 떨어지는 것인가 ?

C언어는 함수를 호출 할 때마다 Stack을 생성한다.

함수 호출의 경우 어셈블리어를 보면 Stack Frame을 만들고 해제하는 코드가 들어간다

Stack Pointer : 스택의 최상위 주소
Base Pointer : 현재 함수에서 stack의 시작 위치 저장 (지역변수 영역 시작 위치)

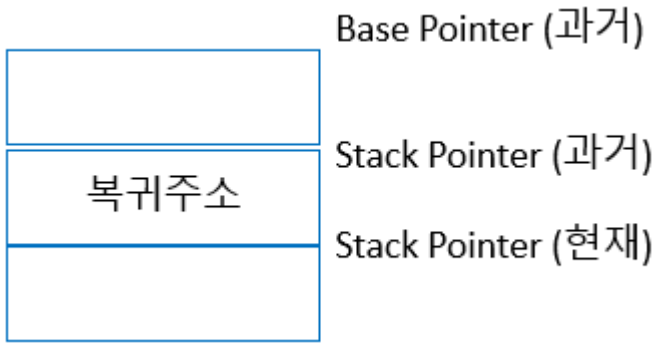
새로운 함수가 호출되면 현재 Stack Pointer 가 처음 Stack 위치이다.
Stack 은 비어 있는 상태



push rbp 수행 시 Stack 의 최상위에 값을 저장 , rsp 는 포인터의 크기만큼 8 Byte 증가

1. TEST

call = push + jmp
push : 복귀 주소를 Stack 에 저장
jmp : call 다음에 오는 주소 값으로 이동



함수 실행되면 Stack 공간 추가 할당
함수에 필요한 작업이 모두 완료되면 pop 으로 이전 상태로 복원

stack frame
호출된 함수가 실행되는 동안 지역변수와 호출한 함수의 실행에 대한 정보가 손실되지 않도록
스택에 저장할 때 취하는 구조

1. TEST

함수의 호출을 구현하기 위해서 필요한 것

실행할 함수의 코드 위치를 갖고 있는 포인터
값을 가져올 수 있는 함수 이름
스레드가 현재 처리하고 있는 함수의 정보를 저장할 공간
함수 실행이 끝난 뒤 리턴 값을 돌려 받을 공간
함수의 실행에 필요한 인자를 넘겨줄 공간

아래 그림은 나중에 어셈블리어로 st 돌려서 확인

```c
void x(int n)
{
    if( n ==  0 )
        return 0;

    else
        printf("%3d",n);
        // output : 3 , 2 , 1

    return x(n-1);
}

int main(void)
{
    x(3);
    return 0;
}
```

```c
void x(int n)
{
    if( n ==  0 )
        return 0;

    x(n-1);
    printf("%3d",n);
    // output : 1 , 2 , 3
}

int main(void)
{
    x(3);
    return 0;
}
```

```c
int x(int n)
{
    if( n ==  1 )
        return 0;

    else
        return 1 + x(n/2);
}

int main(void)
{
    printf("%d", x(8));
    // output : 3
    return 0;
}
```

1. TEST

```c
void x(int n)
{
    if ( n == 0 )
        return 0;

    x(n/2);
    printf("%d", n%2);
    // output : 101
}

int main(void)
{
    x(5);
    return 0;
}
```

```c
void x(int n)
{
    if ( n <= 0 )
        return 0;

    printf("%d\n", n);
    x(2*n);
    printf("%d\n", n);
}

int main(void)
{
    x(5);
    return 0;
}
```

```c
//재귀함수 사용X
int x(int n)
{
    int res = 1;
    while(n != 0)
    {
        res *= n;
        n--;
    }
}

int main()
{
    printf("%d", x(5));
    return 0;
}
```

```c
//재귀함수 사용
int x(int n)
{
    if(n==1)
        return 1;
    else
        return n * x(n-1);
}

int main()
{
    printf("%d", x(5));
    return 0;
}
```

1. TEST

파이프

파이프는 단방향 통신을 위한 용도로 사용
하나의 파이프는 그 이전 파이프에서 전달된 결과를 파라미터로 삼아 또 다른 결과를 도출한다

같은 입력 값에선 같은 반환 값을 보장한다.
함수 외부 스코프의 그 어떠한 변수의 값도 바꾸지 않는다

한 파이프가 반환하는 값은 그 다음 파이프의 입력 값으로 전달된다.
파이프가 상황에 따라서 다른 값을 반환하면 그 다음 파이프의 입장에서는 상황에 따라 바뀔 수 있는
불안정한 값이 입력 값으로 들어오면 결국 반환 값 역시 불안정할 수 밖에 없다

더 작은 시간에 더 많은 명령어를 처리하는 방법

한 개의 사이클이 진행된 이후에 명령어는 새로운 명령어가 들어올 수 있는 공간을 마련해주고 진전되게 된다

파이프라인의 효율은 브랜치나 서브루틴 콜이 많아질수록 떨어진다
브랜치나 서브루틴 콜이 이루어지면 , 파이프라인에서 처리되던 명령어들이 모두 취소되고 새로 브랜치나 서부 루틴의 명령어를 처리해야 하기 때문이다.
최신 아키텍처는 분기 예측 등의 기법을 통해 문제를 회피한다.

1. TEST

2. 회원 이름, 나이, 전화번호, 거주지를 입력

3. 여러 회원을 입력 후 거주지가 같은 사람들만 출력

4. 10, 20, 30대 종류로 출력

```
int a;
char b[20];

scanf("%d", &a); // & 필수 작성
scanf("%s" , b); // b 자체가 주소

printf("%d", b); // b 의 대표 주소 값 출력
printf("%s", b); // b 의 입력 받은 값 똑같이 출력

문자열을 입력 받는 경우 & 를 사용하지 않는다
문자열 ( 포인터 또는 배열 ) = 주소 값

name[0] ← 1 Byte
name[1] ← 1 Byte
name[2] ← 1 Byte
name[?] ← 1 Byte 고정 = char
```

```c
#include <stdio.h>

int main()
{
 int i;
 // 한글은 2byte 차지, 3글자 * 2 Byte + 1(널문자)
 char name[7]="김 효 창 ";

 printf("전체 배열 출력 : %s \n", name);

 //  한글은 2byte 이므로 %c를 2개 작성
 printf("김 : %c%c\n", name[0],name[1]);
 printf("효 : %c%c\n", name[2],name[3]);
 printf("창 : %c%c%c\n",name[4],name[5],name[6]);
 // %c를 띄어 쓰면 깨지게 된다!
 printf("%c %c%c\n",name[4],name[5],name[6]);

 for(i=0;i<6;i++)
 // putchar는 한글자씩 출력. i < 6으로 하면 정상출력
 {
  putchar(name[i]);
 }

 printf("\n");

  for(i=0;i<5;i++)
  // i < 5로 하면 글자가 무너진다.
  //( Null 문자까지 포함해야 정상출력 )
 {
  putchar(name[i]);
 }

return 0;

}
```

1. TEST

Ampersand

& : 주소 값을 지시하는 역할, 어디에 저장되어 있는지를 나타낸다

값을 입력하려면 주소 값이 필요하여 & 를 사용한다.

scanf

stdin : 키보드의 입력을 처리하는 버퍼

키보드로 입력되는 모든 정보는 일시적으로 stdin 에 저장하다가 입력이 종료되면 한 번에 처리하는 것.

stdin 에 아무 것도 없다면 사용자의 입력을 기다리고 있다.
무언가 있다면 그 것을 곧바로 가져온다.

%d 는 숫자를 읽어온다 123abc , 123 까지 읽는다.
%s 도 버퍼가 깨끗한 것은 아니다.

공백 문자 ' ' , '₩n' , '₩t'
scanf 는 stdin 으로부터 의미 있는 문자가 나올 때까지 공백 문자들을 무시.

1. 값을 입력 후 레지스터에 임시 저장
2. 주소 값을 지시하고 있는 메모리에 입력한 값을 저장
3. 레지스터 초기화

1. TEST

if , else

if (조건)
조건이 true 이면 처리 , 0 이 아니면 무조건 처리

else
if 의 조건이 false 이면 처리

bool

#include 〈 stdbool.h 〉 선언해야 사용 가능
true / false 여부로 사용

strcpy (대상 문자열, 원본 문자열)

#include 〈 string.h 〉 선언해야 사용 가능
원본을 대상으로 복사
문자열 끝(₩0)까지 복사

1. TEST

5. 구조체를 사용하는 이유는 ?

구조체를 사용하는 이유는 앞선 문제와 같이

여러 개의 데이터를 한 번에 묶어서 처리하고자 할 때 유용하다.

계속 데이터를 모두 파라미터로 넘기는 것은 너무 불편하고 관리하기에도 힘들다.

반면 구조체로 관리하면 어떤 목적으로 사용하는지가 보다 명시적이 된다.

배열에 값을 할당할 때는 strcpy() 를 사용하여 문자열을 할당한다.
a.name = “김효창” X
strcpy(a.name, 김효창) O

typedef 를 사용하여 새로운 이름을 생성

```
3  struct data{
4      int x, y;
5  }a, *p;
6
7  int main(void)
8  {
9      p = &a;
10     (*p).x = 10;
11     printf("(*p).x = %d\n",(*p).x);
12
13     p -> y = 20;
14     printf("p -> y = %d\n", p -> y);
15 }
```

(*p).x == p -> x : 같은 의미 ,

typedef struct data data;
typedef struct data {
int x, y;
}data;

struct data 를 지금부터 data 로 쓰겠다

```
(*p).x = 10
p -> y = 20
```

1. TEST

구조체 사용 방법

```
struct student
{
int num;
char name[10];
int classs;
};

struct student box1;    //기본적인 사용법

struct student
{
int num;
char name[10];
int classs;
}box1;              //구조체를 선언과 동시에 지정

typedef struct student
{
int num;
char name[10];
int classs;
}ABC; //구조체에 새로운 이름을 생성

ABC box1;
```

문자열 비교 시 문제점

strcmp
내용 같으면 0 반환
왼쪽이 크면 양수 반환
오른쪽이 크면 음수 반환

```
#include <stdio.h>

int main(void)
{
    char p[10];

    scanf("%s",p);

    // if(p == "hello")
    // p는  배열의 대표 주소 , "hello" 는 문자열
    // 비교 성립할 수 없다.
    if(strcmp(p,"hello") == 0 )
        printf("안녕");

    return 0;
}
```

1. TEST

구조체 크기

구조체 type 중 가장 큰 자료형을 기준으로 배수만큼 커진다

패딩 : 메모리 할당은 되었는데 사용되지 않는 공간

#pragma pack(1) : 1, 2, 4, 8, 16 단위로 지정할 수 있다.
1로 설정하면 1바이트 단위로 정렬, 자료형 크기 그대로 메모리에 저장

#pragma pack 사용 전

```c
struct xyz{
    char a;
    int b;
    float c;
};

int main(void)
{
    struct xyz result;

    // 결과 값 : 12 Byte
    printf("%d", sizeof(result));
}
```

#pragma pack 사용 후

```c
#pragma pack(1)

struct xyz{
    char a;
    int b;
    float c;
};

int main(void)
{
    struct xyz result;

    // 결과 값 : 9 Byte
    printf("%d", sizeof(result));
}
```

1. TEST

Fibonacci program

6. 숫자 나열에서 20번째 숫자를 구하세요

연산 1
시작 0 을 기준으로 지정한 범위까지 증가하면 홀수는 +1 , 짝수는 −1 을 곱셈

연산 2
주어진 숫자 나열에서 ‘ 연산 1 ‘ 과 같이 곱셈

연산 3
arr[0] 선택 + arr[2] 를 덧셈
arr[1] 선택 + arr[3] 를 덧셈
arr[2] 선택 + arr[4} 를 덧셈
⋮
arr[16] 선택 + arr[18] 를 덧셈

연산 4
arr[3] 부터 arr[19] 까지의 결과 값

홀수 : arr[i] = arr [i - 1] + arr [i - 3]
짝수 : arr[i] = arr [i - 1] - arr [i - 3]

		연산1	연산2		연산3		연산4
arr[0]	1		1	+	2	=	3
arr[1]	6	-	6	+	3	=	-3
arr[2]	2		2	+	-3	=	-1
arr[3]	3	-	3	+	-1	=	-4
arr[4]	-3		-3	+	-4	=	-7
arr[5]	-1	-	-1	+	-7	=	-6
arr[6]	-4		-4	+	-6	=	-10
arr[7]	-7	-	-7	+	-10	=	-3
arr[8]	-6		-6	+	-3	=	-9
arr[9]	-10	-	-10	+	-9	=	1
arr[10]	-3		-3	+	1	=	-2
		-	-9	+	-2	=	-7
			1	+	7	=	8
		-	-2	+	9	=	10
			7	+	10	=	17
		-	8	+	17	=	9
			10	+	9	=	19

1. TEST

7. 1 ~ 10 사이의 숫자 30개를 생성

8. 1 ~ 10 사이의 숫자를 중복되지 않게 10개 생성

9. 주사위를 굴려서 나온 숫자를 출력

rand 함수 사용 전 → #include 〈stdlib.h〉 입력
time 함수 사용 전 → #include 〈time.h〉 입력

rand 는 예측할 수 없는 하나의 일정한 난수 생성
srand 사용하면 시간 값을 매개로 초기화하면 일정하지 않고 불규칙적인 난수 생성

time 시간에 대한 정보

srand(time(NULL));
time(NULL) 을 호출하면 1970/1/1 0시부터 현재까지 흐른 시간 반환
프로그램 실행될 때마다 srand 의 seed 값이 변경됨

rand() % n : 0 ~ (n - 1)
rand() % n + m : (0 + m) ~ (n - 1) + m
rand() % n * m : 0 ~ (n - 1) 의 나오는 숫자에 m 을 곱셈 , m 의 배수
rand() % n * m + o : m = 2, o = 0 이면 짝수
m = 2, o = 1 이면 홀수 생성
rand() % (n+1-m) + m : (0 + m) ~ (n-1+m+1-m), a ~ b까지

1. TEST

C program

10. 물류 센터에 물건이 이동한다.
    물류 센터 면적은 3000평 정도.
    물건 하나하나 박스 안에 넣어서 배치한다.
    박스 크기는 28평 , 박스와 다른 박스 사이의 간격은 4평.
    물류를 배치하는 효율적인 방법을 프로그래밍하여 구현

1. TEST

C program

11. if문 어셈블리의 특성을 상세히 기술하시오.

if 문의 경우엔 mov로 비교할 대상을 가져와서 cmp를 통해 비교를 수행한다.
이후 EFLAGS 레지스터나 비교 연산의 결과를 가지고 분기를 결정하는 형식으로 구동된다.
결국 mov, cmp, jmp 형식으로 구성된다.

1. TEST

C program

12. 배열 작성 시 나타나는 어셈블리의 특성을 상세히 기술하시오.

```
0x0000555555555184 <+27>:    movl   $0x1,-0x14(%rbp)
0x000055555555518b <+34>:    movl   $0x2,-0x10(%rbp)
0x0000555555555192 <+41>:    mov    -0x10(%rbp),%edx
0x0000555555555195 <+44>:    mov    -0x14(%rbp),%eax
0x0000555555555198 <+47>:    mov    %edx,%r8d
0x000055555555519b <+50>:    mov    $0x1,%ecx
0x00005555555551a0 <+55>:    mov    %eax,%edx
0x00005555555551a2 <+57>:    mov    $0x0,%esi
0x00005555555551a7 <+62>:    lea    0xe56(%rip),%rdi        # 0x555555556004
0x00005555555551ae <+69>:    mov    $0x0,%eax
0x00005555555551b3 <+74>:    callq  0x555555555070 <printf@plt>
```

배열의 시작 주소를 edx에 배치한다.

r8d의 경우엔 r8 레지스터의 32비트 표현이다.

int형 자료이므로 순차적으로 4바이트씩 공간에 배치하는 것을 볼 수 있다.

데이터 자체에 접근할 때도 edx를 기준으로 활용함을 볼 수 있다.

즉, 배열 사용시에는 dx 레지스터가 활용됨을 볼 수 있다.

1. TEST

C program

13. 반복문 작성 시 나타나는 어셈블리의 특성을 상세히 기술하시오.

반복문 작성시 if 문과 마찬가지로 mov, cmp, jmp로 구성된다.
다만 반복을 하기 위해 다시 초기 위치로 되돌리는 jmp와
반복을 탈출하기 위한 jmp로 구성된다.
if와의 차이점이라면 이와 같이 jmp가 두 개 존재한다는 것이다.

1. TEST

C program

14. 함수 호출할 때 발생하는 어셈블리의 특성을 상세히 기술하시오.

함수를 호출할 때 나타나는 특성은 call이다.
이 call의 경우엔 push + jmp를 수행하므로
push에서 Stack에 복귀 주소를 저장하며 jmp에서는 특정 주소로 이동을 하게 된다.
뿐만 아니라 함수 호출시에는 반드시 스택 프레임을 형성하기 위해
push와 mov가 함께 세트로 움직인다.
또한 마지막에 스택을 해제하기 위해 pop과 ret가 동작하게 되어 있다.

1. TEST

C program

15. 프로그래머가 반드시 알아야 하는 가상메모리 4 종류에 대해 상세히 기술하시오

스택(Stack): 지역 변수가 배치된다.

힙(Heap): malloc, calloc등의 동적 할당된 데이터가 배치된다.

데이터(Data): 전역 변수 및 static 변수가 배치된다.

텍스트(Text): 기계어 및 함수가 배치된다.

1. TEST

C program

16. 배열에 대문자로 작성된 문장을 소문자로 변경하여 출력해보세요.

1. TEST

C program

17. 리눅스를 사용하며 알고 있는 명령어를 5가지 이상 작성하고 기능에 대해 기술하세요.

ls: 현재 디렉토리 위치에서 파일 리스트를 보는 것

pwd: 현재 디렉토리 위치 보기

mkdir: 디렉토리 만들기

cp: 복사

mv: 이름 바꾸기 및 위치 옮기기

rm: 삭제

gcc: 컴파일러

vi: 편집기

1. TEST

C program

18. 사원 5명 , 사원의 초봉은 3000 ~ 3500 , 연마다 1 ~ 10 %의 임금 상승폭을 적용.
    10년 후 가장 연봉이 높은 사원의 임금과 이름을 출력

19. 5명의 연도별 평균과 표준 편차를 출력하세요.

End of Document