TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
6일차(2018. 02. 28), 7일차(2018. 03. 02)

과제1. 배운 내용 복습

- Segmentation Fault 가 나는 이유 ?
- Typedef keyword
- malloc() Function, free() Function
- calloc() Function
- Stack 메모리와 Heap 메모리
- 구조체(커스텀 데이터 타입)
- enum(열거형)
- 함수 포인터
- memmove() Function , memcpy() Function
- puts(), putchar() Function / gets(), getchar() Function / strlen() Function
- Stack 예제 그림 그리기

Segmentation Fault 가 나는 이유 ?

•사용자가 건들이지 말아야 할 곳을 건드렸기 때문에 발생하는 에러이다.

기계어에서 살펴봤던 주소값들은 사실 전부 가짜 주소이다. 이 주소값은 엄밀하게 가상 메모리 주소에 해당하고 운영체제의 Paging 메커니즘을 통해서 실제 물리 메모리의 주소로 변환된다.

윈도우도 가상 메모리 개념을 이용해서 사용한다. 그렇다면 당연히 맥(유닉스)도 사용함을 알 수 있다. 가상 메모리는 리눅스의 경우 32 비트 버전과 64 비트 버전이 나뉜다.

32 비트 시스템은 2^32 = 4GB 의 가상 메모리 공간을 가진다. 여기서 1:3 으로 1 을 커널이 3 을 유저가 가져간다. 1 은 시스템(HW, CPU, SW 각종 주요 자원들)에 관련된 중요한 함수 루틴과 정보들을 관리하게 된다. 3 은 사용자들이 사용하는 정보들로 문제가 생겨도 그다지 치명적이지 않은 정보들로 구성된다.

64 비트 시스템은 1:1 로 2^63 승에 해당하는 가상메모리를 각각 가진다. 문제는 변수를 초기화하지 않았을 경우, 가지게 되는 쓰레기값이 0xCCCCC...CCC 로 구성됨이다.

32 비트의 경우에도 1:3 경계인 0xC0000000 을 넘어가게 된다. 64 비트의 경우엔 시작이 C 이므로 이미 1:1 경계를 한참 넘어간다.

그러므로 접근하면 안 되는 메모리 영역에 접근하였기에 엄밀하게는 Page Fault(물리 메모리 할당되지 않음) 가 발생하게 되고 원래는 Interrupt 가 발생해서 Kernel 이 Page Handler(페이지 제어기)가 동작해서 가상 메모리에 대한 Paging 처리를 해주고 실제 물리 메모리를 할당해주는데 문제는 이것이 User 쪽에서 들어온 요청이므로 Kernel 쪽에서 강제로 기각해버리면서 Segmentation Fault 가 발생하는 것이다.

> 실제 Kernel 쪽에서 들어온 요청일 경우에는 위의 메커니즘에 따라서 물리 메모리를 할당해주게 된다.

Typedef keyword

```
#include <stdio.h>

typedef int        INT;
typedef int *      PINT;

 int main(void)
{      INT num = 3;
      PINT ptr = &num;

     printf("num = %d₩n", *ptr);

      return 0; }
```

```
#include <stdio.h>

typedef int        INT[5];

int main(void)
{      int i;
      INT arr = {1, 2, 3, 4, 5};

     for(i = 0; i < 5; i++)
          printf("arr[%d] = %d₩n", i, arr[i]);

     return 0; }
```

• # define와 같은 역할을 한다. **자료형**에 새로운 이름을 부여하고자 할 때(자료형에만) 사용된다.

Ex) real_time_schedule > rts

Typedef real_time_schedule rts

예와 같이 매번 쓰기 불편하므로 간편하게 바꿔쓰기 위함이다. #define과 다른 점은 선행처리기가 처리하지 않고, 컴파일러에 의해 처리되는 내장 명령어라는 것이다.

• 주로 구조체나 공용체, 함수 포인터에 사용한다.

즉, typedef를 사용하는 이유는

1. 자주 사용되는 자료형에 대해 인식하기 쉬운 명칭을 제공하기 위해서이다. 또한 복잡한 자료형을 **간단하게 표현**하기 위함이다.
2. **프로그램 이식성**을 높이기 위해서 이다.

Typedef keyword

```
Typedef   struct{

int x;
int y;
} PosArray 10;
```

- typedef가 없다면 타입을 정의하는 것이 아닌 그냥 이름없는 구조체{x, y}타입의 크기가 10인 배열을 선언하는 것이다.

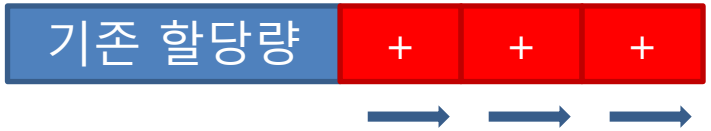
  하지만 typedef가 붙어서 타입을 정의하는 것이 된다.

malloc() Function, free() Function

- **Memory 구조상 heap에 data를 할당**한다. data가 계속해서 들어올 경우 얼만큼의 data가 들어오는지 알 수 없을 때가 있다. 이때, 들어올 때마다 **동적으로 할당할 필요성**이 있기 때문이다. 즉, **주어진 크기만큼 메모리를 동적으로 할당**한다.
- 게임, 검색에 사용되며, 도청기에도 사용된다. 메모리 스택에만 할당 해도 문제 없을 텐데 왜 malloc으로 heap에 동적 할당을 주어야 할까?

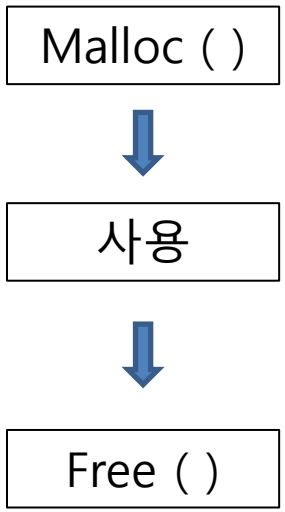
  게임을 예로 들어보자. 처음 게임을 만들 때, 필요한 공간을 할당했을 것이다. 그런데 게임이 점점 흥해서 가입자와 동시 접속자 수가 늘어난다고 가정해보자.

  동시간대에 몇 명이, 몇 대가 사용하게 될 지 알 수 없다. 이런 경우 배열 수를 정할 수가 없다. 정한 배열보다 더 많이 들어오면 커버가 되지 않아 그 배열 밖의 나머지 사람들은 접속 할 수 없기 때문이다. 이런 상황을 처리하는데 이 함수가 사용된다.



  즉, **어떤 상황에서 어떤 수가 얼마나 들어올지 모르기 때문에 malloc 함수가 사용되는 것**이다.
- 단점은 malloc 함수는 속도가 느리다. 동적 할당되는 속도가 느린 것이다. 이를 이용해 서버 등 공격하기도 한다. malloc이 속도를 따라가지 못하기 때문이다. 그래서 게임이나 이런 서버들은 아예 1GB 정도로 할당량을 크게 잡는다. 그럼 처음 접속속도는 느리더라도 나중에 쾌적하게 사용할 수 있기 때문이다.
- 네트워크 사용에 필수적으로 필요하다.

malloc() Function, free() Function



메모리는 위와 같이 Malloc > 사용 > free 패턴으로 사용된다.

- void*malloc(size_t size); > 정상적으로 메모리가 할당되면 **메모리 주소를 반환**한다. 만약 메모리 할당에 실패하면 NULL 포인터를 반환한다.
- void free(void *p); > 반환 값이 없다. 메모리 할당을 해제하는 역할을 하기 때문이다.

- Memory 구조상 heap에 data를 할당 해제한다. malloc()의 반대 역할을 수행하는 것이다. 즉, free 함수는 malloc함수로 할당된 동적 메모리 주소를 운영체제에게 해제할 것을 요청하는 것이다.
  free 함수와 malloc 함수는 거의 한 쌍으로 쓰인다. 하지만 동적 메모리 할당을 받아쓰는 함수인 만큼 많으면 좋지 않다.

- 동적 메모리 할당 순서
  1. 동적으로 메모리를 할당 받을 자료형의 유형에 맞는 포인터 변수 선언
  2. 동적으로 메모리 할당 > malloc 함수
  3. 메모리 사용이 끝나면 메모리를 해제 > free 함수

malloc() Function, free() Function

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

int main(void)
{
int *str_ptr = (char *)malloc(sizeof(char) * 20);

    printf("Input String : ");
     scanf("%s", str_ptr);

    if(str_ptr != NULL)
        printf("string = %s₩n", str_ptr);

    free(str_ptr);

    return 0; }
```

- free함수가 실행되면 운영체제는 다른 응용 프로그램이 heap을 사용할 수 있도록 malloc 함수를 이용해 할당된 주소 값을 해제한다. 만약 free 함수를 이용해 더 이상 사용하지 않는 메모리를 해제해 주지 않는다면, **메모리 누수**가 발생해 다른 프로그램이 사용할 수 있는 heap을 쓸데없이 낭비하게 된다.

따라서, **malloc()을 한 메모리는 꼭 free()로 해제**해 주어야 한다.

calloc() Function

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

int main(void)
{
  int *num_ptr = (int *)calloc(2, sizeof(int));

   printf("Input Integer : ");
   scanf("%d%d", &num_ptr[0], &num_ptr[1]);

     if(num_ptr != NULL)
printf("Integer = %d, %d₩n", num_ptr[0], num_ptr[1]);

    free(num_ptr);

     return 0; }
```

- void * calloc(int n, int size);
- malloc()과 완전히 동일하다. 차이점은 사용 방법에 있다. 1번째 인자는 **할당할 개수**, 2번째 인자는 **할당할 크기**를 나타낸다. size의 크기만큼 n개의 메모리를 할당 받는다는 것이다. 즉, calloc(2, sizeof(int))는 8byte 공간을 할당한다는 뜻이다.

  또한, **할당된 메모리를 0으로 초기화**해야 하는 차이점이 있다.
- malloc()과 마찬가지로 메모리 할당에 실패하면 NULL 값을 반환한다.

Stack 메모리와 Heap 메모리

• 메모리는 **기본적으로 데이터를 저장하는 공간**이다. CPU가 데이터를 연산하고 처리하는 장치라면 메모리는 **입력 및 출력 데이터를 저장하는 역할**을 한다. 메모리는 물리적 또는 기능적으로 구분할 수 있다.

• 물리적 메모리 : HDD, RAM, Register, Cache

하드디스크(HDD, Hard Disk Drive)

상대적으로 **용량당 단가가 낮아** 많은 용량을 구입할 수 있다. 전원이 꺼져도 데이터는 남아 있어, 반영구적으로 데이터를 소장할 수 있다. 다만 처리 속도가 느리다.

램(RAM, Random Access Memory)

단가가 높아 많은 용량을 구입하기 힘들지만 **처리 속도는 매우 빠르다**. 또한 하드디스크와 다르게 전원이 꺼지면 데이터도 날아가는 특성이 있다. 이 때문에 보통 프로그램은 하드디스크에 저장되어 있다가 프로그램이 실행되는 순간 램으로 탑재되어 CPU 연산을 보조하는 메모리로서의 역할을 한다.

레지스터(Register)와 캐쉬(Cache)

CPU 칩 안의 메모리이다. 이들은 **램보다 처리속도가 더 빠르기** 때문에 CPU를 좀 더 가까이서 보조한다.

• 가상(기능) 메모리 : Code, Data, Stack, Heap

메모리는 그 활용과 기능에 따라 Code, Data, Stack, Heap로 구분하기도 한다. 프로그래밍 실행 중에 수시로 메모리 할당과 반환이 이루어지는 경우 **동적인 영역**이라고 하며, 변함없이 프로그램이 끝날 때까지 계속 살아 있는 경우 **정적인 영역**이라고 한다.

Code(정적)

함수와 상수가 저장되는 공간이다. 프로그램이 실행되는 중에 변하지 않으므로 정적이다.

Data(정적)

Global(전역) / Static(정적) 변수가 저장되는 공간이다. 프로그램이 실행되는 동안 계속 여기저기서 필요한 변수들이기 때문에 프로그램이 끝날 때까지 메모리에 남아있는다. 초기화시키지 않은 전역/정적 변수들 BSS(Block Stated Symbol) 영역에 따로 저장된다.

Stack 메모리와 Heap 메모리

•가상(기능) 메모리 : Code, Data, Stack, Heap

heap 영역 (동적)

필요에 의해 동적으로 메모리를 할당 하고자 할 때 위치하는 메모리 영역이다. **동적 데이터 영역**이라고 부르며, **메모리 주소 값에 의해서만 참조되고 사용되는 영역**이다. 즉, Reference Type들이 저장되는 곳이다. C처럼 메모리 관리를 개발자가 직접해야하는 프로그래밍 언어의 경우 참조형 데이터를 이 Heap 영역 메모리에 동적으로 할당하고 반환하는 작업을 일일이 해주어야 한다. 이 영역에 데이터를 저장 하기 위해서 C는 malloc(), C++은 new() 함수를 사용한다. Heap에 저장된 데이터는 모두 참조형이기 때문에 원본 데이터가 갑자기 Heap에서 사라지게 되면 이 원본을 참조했던 변수들에 에러가 발생된다.

stack 영역(동적)

지역(local) / 매개 (parameter) 변수가 저장되는 공간으로 함수의 파라미터나 함수 안에서 선언된 변수들 및 리턴값들이 저장된다. 프로그램이 자동으로 사용하는 **임시 메모리 영역**으로 함수가 실행될 때 메모리에 생성되어 함수가 실행 완료 되면 할당 받았던 메모리를 반환한다.(동적)

함수의 호출 순서에 따라 물건이 쌓이듯(Stack) 메모리에 쌓이고 나가는 구조라서 Stack이라 한다. 가장 나중에 생성된 변수가 가장 위에 스택 되었다가 해당 함수가 종료되면 가장 먼저 소멸되는 LIFO(Last In First Out) 구조이다. 즉, A함수가 B함수를 호출하고, B함수가 C함수를 호출했을 때, C함수 내의 지역변수가 Stack 메모리의 가장 위에 쌓였다가 C함수가 종료되면서 가장 먼저 소멸하게 된다. 스택 사이즈는 각 프로세스마다 할당 되지만 프로세스가 메모리에 로드 될 때 스택 사이즈가 고정되어 있어, 런타임 시에 스택 사이즈를 바꿀 수는 없다.

•Heap에 주소만 있으면 함수가 없어져도 기억하니까 불러올 수 있다.

•code, data, heap 영역은 하위 메모리부터 할당되고, stack 영역은 상위 메모리부터 할당 된다.

구조체(커스텀 데이터 타입)

```
#include <stdio.h>

struct pos{
    double x_pos;

    double y_pos; };

int main(void){
    double num;
    struct pos position;

    num = 1.2;
    position.x_pos = 3.3;
    position.y_pos = 7.7;

    printf("sizeof(position) = %lu₩n", sizeof(position));
    printf("%lf₩n", position.x_pos);
    printf("%lf₩n", position.y_pos);

    return 0; }
```

• 자료를 처리하다 보니 하나로 묶어야 편하기 때문이다. 문자열과, 숫자를 한 번에 묶어서 관리하고 싶을 때 등 어떠한 정보와 그 연결 된 정보를 관리하고 싶을 때 쓴다. 즉, **관련 정보를 하나의 의미로 묶을 때** 쓴다.

전화 번호부를 예시로 볼 수 있다. 연관단축키 문자열와 숫자를 한 번에 묶어서 관리하고 싶을 때 쓸 수 있다. 또한, 자율운전 차량에도 사용된다. 어떤 경로로 가야 하는가? 데이터와 어디에 연결 될 것인가? 와 같은 식으로 자료 구조 유용하게 쓰인다.

• 구조체는 일반적으로 함수 외부에 선언한 후에 사용한다. 함수 외부에 구조체를 선언하면, 이후에 나오는 모든 함수에서 구조체를 호출해서 사용할 수 있다. 만약 함수 안에 구조체를 정의하면, 그 함수 내부에서만 사용할 수 있다.

•구조체는 여러 타입의 변수를 모아둔 커스텀 변수로 커스텀 데이터타입이다.

•구조체 안에 있는 있는 x 포지션을 쓰겠다 하면 앞에 '. '을 찍으면 된다. 포인터는 '->'를 쓴다. 포인터가 지시하는 구조체 변수의 멤버를 지정한다는 뜻이다.

enum(열거형)

```
#include <stdio.h>

typedef enum __packet{
      ATTACK,
      DEFENCE,
      HOLD,
      STOP,
      SKILL,
      REBIRTH,
      DEATH = 44,
      KILL,
      ASSIST
 } packet;

int main(void){
      packet packet;
      for(packet = ATTACK; packet <= REBIRTH; packet++)
          printf("enum num = %d₩n", packet);
      for(packet = DEATH; packet <= ASSIST; packet++)
          printf("enum num = %d₩n", packet);
      return 0; }
```

- 열거형 상수를 지정할 때 사용한다. 숫자, 색깔 등의 일정한 패턴을 지닌 열거형 상수를 선언할 때 사용한다.
- 편하게 #define 할 수 있다.

  define을 이용한 기호 상수 선언은 한번에 하나만 사용이 가능한 반면 enum를 이용한 경우, 한번에 여러 개의 상수를 정의할 수 있기 때문이다.

- '__' 변경하지 말라는 뜻이다. 바꾸게 되면 시스템에 치명적일 수도 있으며 뜻이 안 통할 수도 있기 때문이다.

함수 포인터

```
#include <stdio.h>

void (* aaa(void))[2]{
      static int a[2][2] = {10, };
      printf("aaa called₩n");
      return a; }

int (*(* bbb(void))(void))[2]{
      printf("bbb called₩n");
      return aaa; }

int main(void){
      int (*ret)[2];
      int (*(*(*p[][2])(void))(void))[2] = {{bbb, bbb}, {bbb, bbb}};
      int (*(*(*(*p1)[2])(void))(void))[2] = p;
      ret = ((*(*(*(*p1)[2]))())());
      printf("%d₩n", *ret[0]);

      return 0; }
```

• **특정 함수에 대한 메모리 주소를 담을 수 있는 포인터**이다. 함수도 실행이 시작되는 주소를 가지고 있기 때문에 이 주소를 포인터에 넣을 수 있다. 포인터에 저장된 함수 주소를 이용하여 함수를 호출할 수 있다. 일반적인 포인터는 변수가 저장되어 있는 주소를 가리키지만 함수 포인터는 함수가 시작되는 주소를 가리킨다.

• 함수포인터를 쓰는 이유는 프로그램 코드가 간결해지기 때문이다. 함수포인터를 배열에 담아서도 사용할 수 있으므로 중복되는 코드를 줄일 수 있다. 상황에 따라 해당되는 함수를 호출할 수 있으므로 굉장히 유용하다. 그 외에도 함수 포인터를 이용하여 콜백함수를 구현할 수 있게 되는 등 편리하고 유용한 코드를 작성할 수 있게 된다.

void(*signal(int signum, void (* handler)(int)))(int) ; // signal은 함수다

• 함수 프로토타입이란?

리턴, 함수명, 인자에 대한 기술서이다. 그렇다면 위 함수에 대한 프로토 타입은 뭘까?

푸는 방식은 이전에 배웠던 int (*p)[2]; → int(*)[2]p 생각하면 된다.

리턴 : void(*)(int)

함수명 : signal

인자 : int signum 과 void(*handler)(int)

void(*p)(void) :

void를 리턴하고 void를 인자로 취하는 함수의 주소값을 저장할 수 있는 변수 p를 뜻한다.

함수 포인터

```
#include <stdio.h>

void (* aaa(void))[2]{
      static int a[2][2] = {10, };
      printf("aaa called₩n");
      return a; }

int (*(* bbb(void))(void))[2]{
      printf("bbb called₩n");
      return aaa; }

int main(void){
      int (*ret)[2];
      int (*(*(*p[][2])(void))(void))[2] = {{bbb, bbb}, {bbb, bbb}};
      int (*(*(*(*p1)[2])(void))(void))[2] = p;
      ret = ((*(*(*(*p1)[2]))())());
      printf("%d₩n", *ret[0]);

      return 0; }
```

•int (*(* bbb(void))(void))[2]; // 실제 문법

•int (*)[2](*)(void) bbb(void) // 사람이 보기 편하게 만든 것

푸는 방법은 아래와 같다.

case1 : criteria : function name
int (*(*)(void))[2] bbb(void)
int (*)[2](*)(void) bbb(void)00

case2 : first'(*' last')~~'
int (*)[2](* bbb(void))(void)
int (*)[2](*)(void) bbb(void)

배열 2개짜리 묶인 주소를 반환하고 인자로 void를 취하는
함수 포인터를 반환하며 인자로 void를 취하는 함수 bbb
return : int(*)[2](*)(void)
name : bbb
parameter : void

int (*)[2] aaa(void)
int (*)[2] (*)(void) bbb(void)
> 두개의 형태가 비슷한 것으로 보아 bbb가 aaa를 리턴할 수 있다는 뜻이다.

함수 포인터를 왜 쓰는가?

1. **비동기 처리**
2. HW개발 관점에서 인터럽트
3. 시스템 콜(유일한 SW인터럽트임)

여기서 인터럽트들 (sw, hw)은 사실상 모두 비동기 동작에 해당한다. 결국 비동기 처리가 핵심이라는 의미다. 그렇다면 비동기 처리라는 것은 무엇일까?

기본적으로 동기 처리라는 것은 송신하는 쪽과 수신하는 쪽이 쌍방 합의에만 달성된다. (휴대폰 전화 통화 등등) 반면, 비동기 처리는 이메일, 카톡 등의 메신저에 해당한다. 그래서 그냥 일단 던져 놓으면 상대방이 바쁠 때는 못 보겠지만 그다지 바쁘지 않은 상황이라면 메시지를 보고 답변을 줄 것이다. 이와 같이 언제 어떤 이벤트가 발생할지 알 수 없는 것들을 다루는 것이 바로 함수 포인터다. 다른 예시를 들어보자. 불이 났다고 가정을 해보자. 이때 소방훈련에서 배웠던 소화기를 찾을 것이다. 이 소화기를 찾는 것이 컴퓨터에서는 함수포인터로 등록해놓은 상황인 것이다.

사람이 이런 상황에서 임기응변을 잘 해야 하듯이 컴퓨터 관점에서 임기응변을 잘 하도록 만들어주는 것이 바로 함수 포인터다.

즉, **비동기적 상황은 함수포인터로 지정해서 해결**한다.

memmove() Function, memcpy() Function

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

int main(void){
      int i = 0;
      int src[5] = {1, 2, 3, 4, 5};
      int dst[5];

    memmove(dest, src, sizeof(src));

    for(i = 0; i < 5; i++)
          print(dst[%d] = %d₩n", i, dst[i]);

    return 0; }
```

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

int main(void){
      char src[30] = "This is amazing";
      char *dst = src + 3;

    printf("before memmove = %s₩n", src);
     memcpy(dst, src, 3);
     printf("after memmove = %s₩n", dst);

    return 0; }
```

- #include <string.h> 선언을 해주어야 쓸 수 있다.
- **memmove()**는 Memory Move의 합성어임 메모리의 값을 복사할 때 사용한다. **memove(목적지, 원본, 길이);**로 사용한다. 어떤 정보를 복사하고 싶을 때, 가운데에 입력하고 복사하고 싶은 자리는 앞에, 크기는 뒤에 지정해 주는 것이다. memcpy보다 느리지만 안정적이다. **메모리를 보호해준다.**
- **memcpy()**는 메모리 보호 안 해주는 반면, **속도가 memmove()보다 빠르다.** 즉, 메모리 공간에 겹치는 부분이 없을 때, 사용 겹치는 부분이 없다면 성능에 좋다.
- Compiler에 따라 정상적으로 처리될 수도 있지만 **가급적이면 사용하고자하는 Memory 공간이 겹칠때는 memmove를 사용**하도록 하자.

puts(), putchar() Function / gets(), getchar() Function / strlen() Function

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

int main(int argc, char **argv){
      char *str = "This is the string";
     int len = strlen(str);
      printf("len = %d₩n", len);
      return 0; }
```

- put() 계열 함수는 언제 사용하는가?
  printf 대용으로 사용할 수 있다. 실제로 Compiler에서는 이러한 최적화 과정도 거친다. puts()는 string을 인자로 받는다. putchar()는 'A'같은 것을 인자로 받는다.

- get() 계열 함수는 언제 사용하는가?
  put() 함수의 상반되는 개념으로 scanf()와 유사하다.

- strlen() Function
  문자열의 길이를 구하는데 사용한다. 문자열 할당할 때 쓰인다.

strcpy(), strncpy() Function / strcmp(), strncmp() Function

- #include <string.h> 선언을 해주어야 쓸 수 있다.
- strcpy() 함수는 언제 사용하는가?
  문자열을 복사하고 싶을 경우 사용한다. strcpy(dst, src), strncpy(dst, src, length)로 사용한다.

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

int main(int argc, char **argv){
      char src[20];
      char dst[20];
      strcpy(src, "pretty");
      strcpy(dst, src);
      printf("dst = %s₩n", dst);
      return 0; }
```

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

int main(int argc, char **argv){
      char src[20] = "abcdef";
      char dst[20];
      strncpy(dst, src, 3);
      printf("dst = %s₩n", dst);

      return 0; }
```

- strcmp() 함수는 언제 사용하는가?
  문자열이 서로 같은지 비교하고 싶을 때 사용한다. 서로 같은 경우 0을 반환하게된다. strcmp(str1, str2), strncmp(str1, str2, len)처럼 사용한다.
- strncpy 안 쓰면 해킹 당할 수 있다. 그러므로 무조건 strncpy를 쓰는 것이 좋다. 보완 코드 권장사항이다. strcmp(), strncmp() 도 같은 이유로 strncmp를 써야 한다.

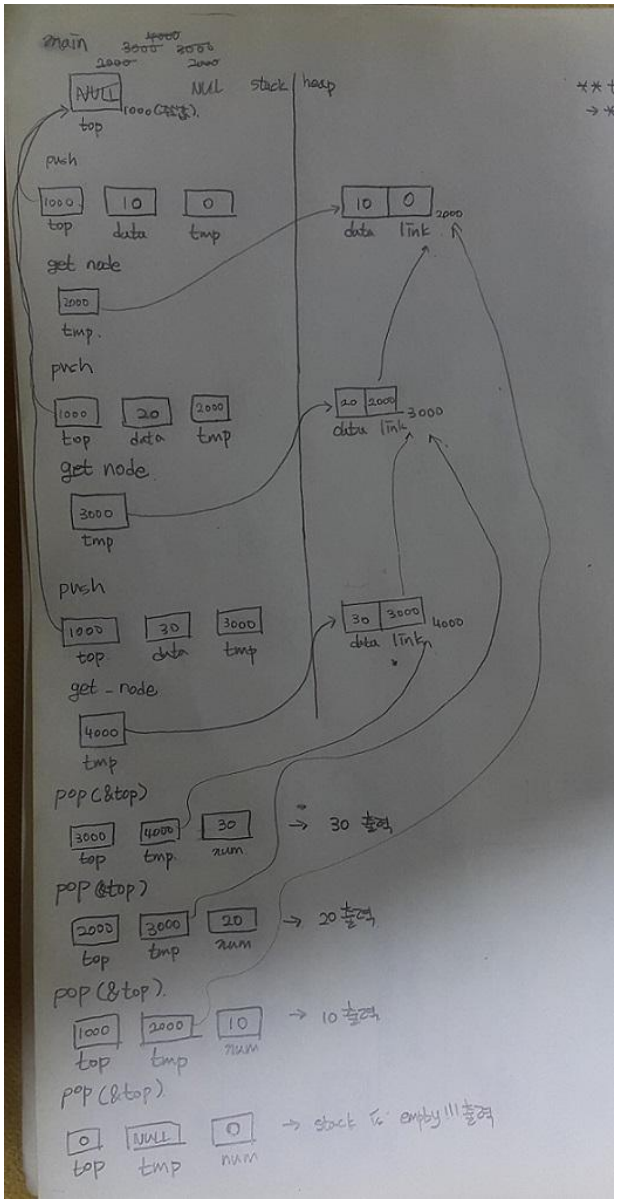
```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv){
      char src[20] = "Korea";
      char dst[20] = "korea";
      if(!strcmp(src, dst))
          printf("src, dst는 서로 같음₩n");
      else
          printf("src, dst는 서로 다름₩n");
      return 0; }
```

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

int main(int argc, char **argv){
      char src[20] = "made in korea";
      char dst[20] = "made in china";
      if(!strncmp(src, dst, 8))
          printf("src, dst는 서로 같음₩n");
      else
          printf("src, dst는 서로 다름₩n");
      return 0; }
```

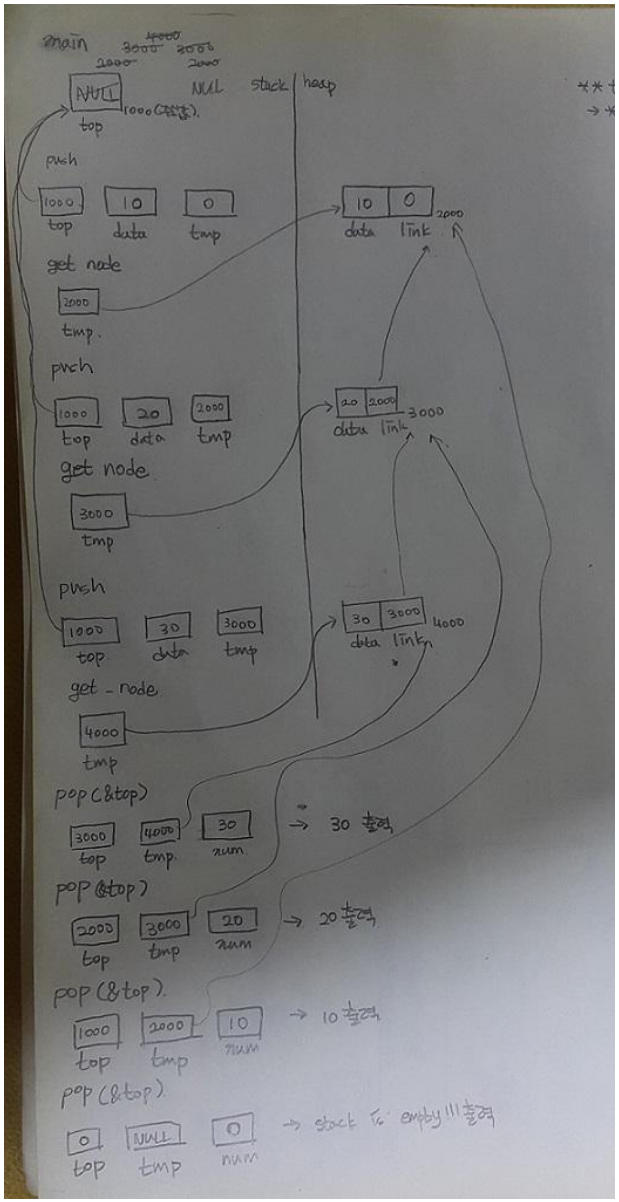
Stack 예제

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

#define EMPTY 0

struct node{
        int data;
        struct node *link;
 }; typedef struct node Stack;

Stack *get_node() {
        Stack *tmp;
        tmp=(Stack *)malloc(sizeof(Stack));
        tmp->link=EMPTY;
        return tmp; }

void push(Stack **top, int data) {
        Stack *tmp;
        tmp = *top;
        *top = get_node();
        (*top)->data = data;
        (*top)->link = tmp; }
```

```
int pop(Stack **top) {
        Stack *tmp;
        int num;
        tmp = *top;
        if(*top == EMPTY)
        {
                printf("Stack is empty!!!₩n");
                return 0;          }
        num = tmp->data;
        *top = (*top)->link;
        free(tmp);
        return num; }


int main(void) {
        Stack *top = EMPTY;
         push(&top, 10);
         push(&top, 20);
         push(&top, 30);
         printf("%d₩n", pop(&top));
         printf("%d₩n", pop(&top));
         printf("%d₩n", pop(&top));
         printf("%d₩n", pop(&top));
         return 0; }
```



Stack



- int pop(Stack **top) {
  Stack *tmp;
  int num;
  tmp = *top;
  if(*top == EMPTY) {
  printf("Stack is empty!!!₩n");
  return 0; }
  num = tmp->data;
  *top = (*top)->link;
  free(tmp);
  return num; }
- POP 하면서 top , tmp, num을 생성한다.
- tmp는 *top 즉, top의 주소가 가르키는 값을 가지므로 4000이 된다.
- 이때 *top는 0이 아니므로 if문은 지나가게 된다.
- num은 tmp의 주소가 접근하는 data 값을 대입하게 되므로 값이 30이 된다.
- *top는 *top의 가지던 link로 접근하여 그 값을 가지게 되므로 3000이 된다.
- free() 함수는 tmp를 반환하고 나오므로, 마지막 push()가 없어지고 main에 있는 top는 3000으로 바뀐다.
- 그리고 num을 반환하면서 값은 30이 출력되고 남은 3번의 pop은 같은 동작을 반복한다.

과제 2. C언어 코드 복습

int data = 5 , result = 0 ;

으로 변수가 선언했다고 가정합니다.

1.
문제 result = data ==5;
답 (result : , data :)

2.
result = data != 5 && (data = 0);
(result : , data :)

3.
result = --result && (data = 0);
(result : , data :)

4.
result = result-- l l (data = 0);
(result : , data :)

5.
result = result-- && (data = 0);
(result : , data :)

```
#include <stdio.h>

int main(void)
{
int data = 5;
int result = 0;

result = data ==5;
printf("문제 1번은 %d, %d ₩n", result, data);

result = data !=5 && (data=0);
printf("문제 2번은 %d, %d ₩n", result, data);

result = --result &&( data=0);
printf("문제 3번은 %d, %d ₩n", result, data);

result = result--|| (data=0);
printf("문제 4번은 %d, %d ₩n", result, data);

result = result--&& (data=0);
printf("문제 5번은 %d, %d ₩n", result, data);

return 0;
}
```