Xilinx Zynq FPGA, TI DSP, MCU기반의 프로그래밍 및 회로 설계 전문가 과정

강사 - Innov (이상훈)
gcccompil3r@gmail.com
학생 - 이유성
dbtjd1102@naver.com

swich문
전등 스위치 같은 것이라고 생각하면 된다.
a를 누르면 a가 들어오고
b를 누르면 b가 들어온다고 보면 되겠다

break문은 현재 switch문을
그냥 빠져나가겠다는 것을 의미한다

```
#include<stdio.h>
int main(void){

	int num;

	printf("상담은 1, 결제는 2, 교체는 3 : ");
	scanf("%d",&num);

	switch(num)
	{
		case 1:
		printf("상담 연결 중 입니다\n");
		break;
		case 2:
		printf("결제 중 입니다\n");
		break;
		case 3:
		printf("교체 연결 중 입니다\n");
		break;

		default:
		printf("잘못 누르셨습니다\n");
		}
		return 0;
	}
```

static문

```
#include <stdio.h>
void count_static_value(void)
{
	static int count = 1;
	printf("count = %d\n", count);
	count++;
}
int main(void)
{
```

```
	int i;
	for(i = 0; i < 7; i++)
		count_static_value();
}
```

전역 변수란 무엇인가 ?
변수의 위치에 관계 없이 data값을 참조할 수 있음
Thread Programming시 문제 발생
동기화 메커니즘이 필요한 이유 ?!
Memory의 data 영역에 load됨

```
#include<stdio.h>
	int number  = 6;

void func(){

	printf("%d\n",number+1);



	}

int main(){

	printf("%d\n",number);

	func();

	return 0;
	}
```

static 키워드의 역할은 ?
정적 변수
static 변수는 전역 변수와 마찬가지로 data 영역에 load됨
지역 변수를 static으로 선언했다면
이 변수를 선언한 함수내에서만 접근 가능함

```
#include <stdio.h>
void count_static_value(void)
{
	static int count = 1;
	printf("count = %d\n", count);
	count++;
```

```
}
int main(void)
{
	int i;
	for(i = 0; i < 7; i++)
		count_static_value();
}
```

while을 통한 무한 Loop는 어떻게 ?
조건을 무조건 참으로 만들면 된다
true는 1
false는 0

break로 조건을 줘서 멈출 수 있다.

```
#include<stdio.h>
int main(void)
{

	int number =0;
	while(1)
	{
		printf("%d\n",number);
		number++;
		if(number  ==100){
		break;
		}

	}
	return 0;
}
```

continue문은 왜 필요할까 ?
해당 case는 제끼고 반복은 계속하고 싶어서

```
#include<stdio.h>
int main(void)
{
	int number =0;
	while(1)
	{
```

```
            if(number ==5)
            continue;
            printf("%d\n",number);
            number++;
            if(number ==10)
            break;

        }
        return 0;
}
```

여기서
number가 5인 경우
아래를 수행하지 않고
while의 조건을 검사하러 돌아감

배열

```
#include <stdio.h>
int main(void)
{
 char str1[5] = "AAA";
 char str2[ ] = "BBB";
 char str3[ ] = {'A', 'B', 'C'};
 char str4[ ] = {'A', 'B', 'C',  '\0'};
 printf("str1 = %s\n", str1);
 printf("str2 = %s\n", str2);
 printf("str3 = %s\n", str3);
 printf("str4 = %s\n", str4);
 str1[0] = 'E';
 str2[1] = 'H';
 printf("str1 = %s\n", str1);
 return 0;
}
```

```
str1 = AAA
str2 = BBB
str3 = ABC
str4 = ABC
str1 = EAA
```

```
#include <stdio.h>
int main(void)
```

```
{
 int arr[3][3][3];
 int i, j, k, num = 1;
 for(i = 0; i < 3; i++)
 {
 for(j = 0; j < 3; j++)
 {
 for(k = 0; k < 3; k++)
 {
 arr[i][j][k] = num++;
 printf("arr[%d][%d][%d] = %d\n", i, j, k, arr[i][j][k]);
 }
 }
 }
 return 0;
}
```

```
arr[0][0][0] = 1
arr[0][0][1] = 2
arr[0][0][2] = 3
arr[0][1][0] = 4
arr[0][1][1] = 5
arr[0][1][2] = 6
arr[0][2][0] = 7
arr[0][2][1] = 8
arr[0][2][2] = 9
arr[1][0][0] = 10
arr[1][0][1] = 11
arr[1][0][2] = 12
arr[1][1][0] = 13
arr[1][1][1] = 14
arr[1][1][2] = 15
arr[1][2][0] = 16
arr[1][2][1] = 17
arr[1][2][2] = 18
arr[2][0][0] = 19
arr[2][0][1] = 20
arr[2][0][2] = 21
arr[2][1][0] = 22
arr[2][1][1] = 23
arr[2][1][2] = 24
arr[2][2][0] = 25
```

```
arr[2][2][1] = 26
arr[2][2][2] = 27
```

```
int main(void){
 char str1[33] = "Pointer is important!";
 char *str2 = "Pointer is important!";
 printf("str1 = %s\n", str1);
 printf("str2 = %s\n", str2);
 return 0;
}
```

```
str1 = Pointer is important!
str2 = Pointer is important!
```

Pointer 배열이란 무엇인가 ?
Pointer로 이루어진 배열을 의미함
여기서 주의해야할 것이 몇 가지 존재함

```
1. int main(void){
2.  int i, j, n1, n2, n3;
3.  int a[2][2] = {{10, 20}, {30, 40}};
4.  int *arr_ptr[3] = {&n1, &n2, &n3};
5.  int (*p)[2] = a;
6.  for(i = 0; i < 3; i++)
7.  *arr_ptr[i] = i;
8.  for(i = 0; i < 3; i++)
9.  printf("n%d = %d\n", i, *arr_ptr[i]);
10. for(i = 0; i < 2; i++)
11. printf("p[%d] = %d\n", i, *p[i]);
12. return 0;
13.}
```

```
n0 = 0
n1 = 1
n2 = 2
p[0] = 10
p[1] = 30
```

```
#include<stdio.h>
int main(void){
 int arr[7] = {10, 20, 30};
 int *arr_p = &arr[1];
 printf("현재값 = %d\n", *arr_p);
 printf("한칸 +이동 = %d\n", *(arr_p + 1));
 printf("한칸 –이동 = %d\n", *(arr_p - 1));
 printf("세칸 + 이동 = %d\n", *(arr_p + 3));
 return 0;
}
```

```
현재값 = 20
한칸 +이동 = 30
한칸 –이동 = 10
세칸 + 이동 = 0
```

```
#include<stdio.h>
int main(void){
 int arr[7] = {10, 20, 30};
 int *arr_p = &arr[1];
 printf("현재값 = %d\n", *arr_p);
 printf("한칸 +이동 = %d\n", *(arr_p + 1));
 printf("세칸 + 이동 = %d\n", *(arr_p + 3));
 return 0;
}
현재값 = 20
한칸 +이동 = 30
세칸 + 이동 = 0
```

```
#include<stdio.h>
int main(void){
 int arr[7] = {10, 20, 30,};
 int *arr_p = arr;
 printf("현재값 = %d\n", *arr_p);
 printf("한칸 +이동 = %d\n", *(arr_p + 1));
 printf("한칸 –이동 = %d\n", *(arr_p - 1));
 printf("세칸 + 이동 = %d\n", *(arr_p + 3));
 return 0;
}
```

```
현재값 = 10
한칸 +이동 = 20
```

한칸 –이동 = 0
세칸 + 이동 = 0

```
#include<stdio.h>
int main(void){
 int arr1[3][2];
 int arr2[3][3];
 int arr3[3][4];
 printf("%p\t%p\t%p\n", arr1, arr1+1, arr1+2);
 printf("%p\t%p\t%p\n", arr2, arr2+1, arr2+2);
 printf("%p\t%p\t%p\n", arr3, arr3+1, arr3+2);
 return 0;
}
```

```
0x7ffe60f5eea0        0x7ffe60f5eea8        0x7ffe60f5eeb0
0x7ffe60f5eec0        0x7ffe60f5eecc        0x7ffe60f5eed8
0x7ffe60f5eef0        0x7ffe60f5ef00        0x7ffe60f5ef10
```

typedef는 무엇인가 ? 자료형에 새로운 이름을 부여하고자 할 때 사용 주로 구조체나 함수 포인터에 사용함

```
#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;
}
```

```
num = 3
```

```
#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;
}
```

arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5

재귀함수
피보나치

```
#include <stdio.h>
int fib(int num){
 if(num == 1 || num == 2)
 return 1;
 else
 return fib(num-1) * fib(num-2);
}
int main(void){
 int result, final_val;
 printf("피보나치 수열의 항의 개수를 입력하시오 : ");
 scanf("%d", &final_val);
 result = fib(final_val);
 printf("%d번째 항의 수는 = %d\n", final_val, result);
 return 0;
}
```

팩토리얼

```
#include <stdio.h>
int fact(int num){
 if(num == 0)
 return 1;
 else
 return num*fact(num-1);
}
int main(void){
 int result, fact_val;
 printf("계산할 팩토리얼을 입력하시오 : ");
 scanf("%d", &fact_val);
 result = fact(fact_val);
 printf("%d번째 항의 수는 = %d\n", fact_val, result);
 return 0;
}
```

malloc()은 무엇을 하는가 ? Memory 구조상 heap에 data를 할당함 data가 계속해서 들어올 경우 얼만큼의 data가 들어오는지 알 수 없음 들어올 때마다 동적으로 할당할 필요성이 있음

free()은 무엇을 하는가 ? Memory 구조상 heap에 data를 할당 해제함 malloc()의 반대 역할을 수행함

```
#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;
}
Input String : string = asdasd
```

volatile은 언제 사용하는가 ?
주로 Device Driver를 만지게되면 사용함
HW의 I/O를 읽을때 주소값이 바뀔 수 있음
Compiler가 최적화를 수행해버리면
HW의 변경된 주소값에서 가져오는 것이 아닌
최적화가 수행된 일정한 주소에서 값을 가져옴
Linux Kernel의 Memory Barrier등에서도 볼 수 있음

memmove()는 언제 사용하는가 ?
Memory Move의 합성어임
메모리의 값을 복사할 때 사용함
memove(목적지, 원본, 길이);로 사용함
memcpy보다 느리지만 안정적임

memcpy()는 언제 사용하는가 ?
memory 공간에 겹치는 부분이 없을때 사용
겹치는 부분이 없다면 성능에 좋음

```
#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;
```

}

before memmove = This is amazing
after memmove = This amazing

Compiler에 따라 정상적으로 처리될 수도 있지만
가급적이면 사용하고자하는 Memory 공간이 겹칠때는
memmove를 사용하도록 하자

put() 계열 함수는 언제 사용하나 ?
printf 대용으로 사용할 수 있다
실제로 Compiler에서는 이러한 최적화 과정도 거침
puts()는 string을 인자로 받음
putchar()는 'A'같은 녀석을 인자로 받음

gets(), getchar() Function get() 계열 함수는 언제 사용하나 ?
put() 함수의 상반되는 개념임
scanf()와 유사함

strlen() 함수는 언제 사용하나 ?
주로 strcpy()등의 함수와 함께 사용하는편 문자열의 길이를 구하는데
사용함 strlen("문자열");처럼 사용함

구조체는 왜 사용하는가 ?
자료를 처리하다보니 하나로 묶어야 편함 문자열과, 숫자를 한 번에 묶어서 관리하고
싶을때등

```
#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) = %d\n", sizeof(position));
 return 0;
}
```

```
sizeof(position) = 16
```

```
#include <stdio.h>
typedef struct __id_card{
 char name[30];
 char id[15];
 unsigned int age;
} id_card;

typedef struct __city{
 id_card card;
 char city[30];
} city;

int main(void){
 int i;
 city info = {
 {"Marth Kim", "800903-1012589", 34}, "Seoul"
 };
 printf("city = %s, name = %s, id = %s, age = %d\n",
 info.city, info.card.name, info.card.id, info.card.age);

 return 0;
}
```

```
city = Seoul, name = Marth Kim, id = 800903-1012589, age = 34
```

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


typedef struct __id_card{
 char *name;
 char *id;
 unsigned int age;
} id_card;


typedef struct __city{
 id_card *card;
 char city[30];
} city;


int main(void){
 int i;
 city info = {NULL, "Seoul"};
 info.card = (id_card *)malloc(sizeof(id_card));
 info.card->name = "Marth Kim";
 info.card->id = "800903-1012589";
 info.card->age = 33;
 printf("city = %s, name = %s, id = %s, age = %d\n",
 info.city, info.card->name, info.card->id, info.card->age);

 free(info.card);
 return 0;
}
```

```
city = Seoul, name = Marth Kim, id = 800903-1012589, age = 33
```

공용체는 왜 사용하는가 ?
경우에 따라 각각 사용하고자 할 때 사용 메모리를 절약하고자 할 때 사용
객체 관리를 조금 더 깔끔하게 하고자 할 때 사용

```
#include <stdio.h>
typedef union __data{
 int val;
```

```
 char ch;
 double real;
} data;
int main(void){
 data data;
 printf("sizeof(data) = %d\n", sizeof(data));
 data.real = 1;
 printf("data.val = %d\n", data.val);
 printf("data.ch = %d\n", data.ch);
 printf("data.real = %d\n", data.real);
 return 0;
}
```

```
sizeof(data) = 8
data.val = 0
data.ch = 0
data.real = 0
```

열거형은 왜 사용하는가 ?
편하게 define을 할 수 있음

```
#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;
}
```

```
enum num = 0
enum num = 1
enum num = 2
```

enum num = 3
enum num = 4
enum num = 5
enum num = 44
enum num = 45
enum num = 46

함수 포인터를 왜 사용하는가 ?
상황에 따라 다양한 함수를 호출 할 수 있으므로
Event 방식의 동작을 구현하고 이해하는데 필수적 Interrupt의 동작 방식

선언문은 선언 대상이 되는 변수 명에서 시작해서 오른쪽으로 가면서 해석한다.
선언문의 끝이나 오른쪽 괄호를 만나면 방향을 바꾸어 왼쪽으로 가면서 해석한다.
왼쪽으로 가면서 해석을 하다 왼쪽 괄호를 만나면 다시 오른쪽으로 가면서 해석한다.

int **a;
a는 int형을 가리키는 포인터의 포인터다.

int *a[3];
a는 int형을 가리키는 포인터를 3개 저장하는 배열이다.

int (*a)[3];
a는 int형을 3개 저장하고있는 배열을 가리키는 포인터다

int (*p)(char);
p는 char를 인자로 갖고, int형을 반환하는 함수에 대한 포인터다.

char** (*p)(float, float);
p는 float, float을 인자로 갖고, char을 가리키는 포인터의 포인터를 반환하는 함수에 대한 포인터다

void* (*a[5])(char* const, char* const);
a는 char* const, char *const를 인자로 갖고, void를 가리키는 포인터를 리턴하는 함수에 대한 포인터를 5개 저장하는 배열이다

int* (*(fp1)(int))[10];
fp1은 int를 인자로 갖고, int를 가리키는 포인터를 10개 저장하고 있는 배열의 포인터를 리턴하는 함수다.

int* (*(arr[5])())();
arr는 int를 가리키는 포인터를 리턴하는 함수에 대한 포인터를 리턴하는 함수 5개를 저장하는 배열이다

float (*(*b())[])();

b는 float를 리턴하는 함수에 대한 배열을 저장하고있는 포인터를 리턴하는 함수에 대한 포인터다.

void* (c)(char, int ()());
c는 char, int()()를 인자로 갖고, void를 가리키는 포인터를 리턴하는 함수다.

void** (*d)(int &, char**(*) (char*, char**));
d는 int &, char**(*)(char *, char *)를 인자로 갖고, void를 가리키는 포인터의 포인터를 리턴하는 함수에 대한 포인터다

float (*(*e[10])(int &))[5];
e는 int &를 인자로 갖고, float을 5개 저장하고있는 배열의 포인터를 리턴하는 함수에 대한 포인터를 10개 저장하고 있는 배열이다.

int *(*(*fp1)(int))[10];
fp1은 int를 인자로 갖고, int를 가리키는 포인터 10개를 저장하고있는 배열의 포인터를 리턴하는 함수에 대한 포인터다

char *(*(**foo[][8])())[];
foo는 char를 가리키는 포인터를 저장하고 있는 배열의 포인터를 리턴하는 함수에 대한 포인터의 포인터 8개를 저장할 수 있는 배열에 대한 배열이다.

void bbb(void(*p)(void));
bbb는 인자로 (void(*p)(void)를 갖고, void를 리턴하는 함수다.

void(*bbb(void))(void);
bbb는 인자로 void를 갖고, void를 인자로 갖고 void를 리턴하는 함수에 대한 포인터를 리턴하는 함수다.

void(*bbb(void(*p)(void)))(void);
bbb는 인자로 void(*p)(void)를 갖고, void를 인자로 갖고 void를 리턴하는 함수에 대한 포인터를 리턴하는 함수다

int (*(*bbb(void))(void))[2];
bbb는 인자로 void를 갖고, void를 인자로 갖고 int를 2개 저장하고있는 배열의 포인터를 리턴하는 함수에 대한 포인터를 리턴하는 함수다.

#include <stdio.h>

```
void aaa(void){
 printf("aaa called\n");
}
void bbb(void(*p)(void)){
 p();
 printf("bbb called\n");
}
int main(void){
 bbb(aaa);
 return 0;
}
```

```
aaa called
bbb called
```