C언어 - HW4

임베디드스쿨1기
Lv1과정
2020. 08. 15
박성환

구조체

1. 구조체 특징
담고자 하는 것을 모아 카테고리로 관리에 편함
특징들을 모아 관리하기 편함
다양한 데이터 타입을 하나로 묶을 수 있음

```
typedef struct 태그 {
    자료형 멤버이름;
} 타입이름;
```

구조체 태그와 타입이름이 꼭 다를 필요가 없고 현재 태그 = 타입이름으로 사용중

```c
typedef struct Person {    // 구조체 이름은 Person
    char name[20];
    int age;
    char address[100];
} Person;                  // typedef로 정의한 타입 이름도 Person
```

구조체

2. 구조체 초기화

값 : p1.age = 30;

문자열 : strcpy(p1.name, “서울시 강서구”);

```c
#include <stdio.h>
#include <string.h>    // strcpy 함수가 선언된 헤더 파일

struct Person {   // 구조체 정의
    char name[20];        // 구조체 멤버 1
    int age;              // 구조체 멤버 2
    char address[100];    // 구조체 멤버 3
};

int main()
{
    struct Person p1;     // 구조체 변수 선언

    // 점으로 구조체 멤버에 접근하여 값 할당
    strcpy(p1.name, "홍길동");
    p1.age = 30;
    strcpy(p1.address, "서울시 용산구 한남동");

    // 점으로 구조체 멤버에 접근하여 값 출력
    printf("이름: %s\n", p1.name);       // 이름: 홍길동
    printf("나이: %d\n", p1.age);        // 나이: 30
    printf("주소: %s\n", p1.address);    // 주소: 서울시 용산구

    return 0;
}
```

Union

1. 유니온 특징
구조체와는 달리 동일한 메모리 위치에 다른 데이터 유형 저장할 수 있음
구조체와 같이 정의만 해서는 사용할 수 없음(공용체도 변수로 선언해서 사용
공용체의 전체 크기는 가장 큰 자료형의 크기

2. 유니온 기본 형태

```
typedef union _Box {    // 공용체 이름은 _Box
    short candy;
    float snack;
    char doll[8];
} Box;                  // typedef를 사용하여 공용체 별칭을 Box로 정의

typedef union {     // 익명 공용체 정의
    short candy;
    float snack;
    char doll[8];
} Box;              // typedef를 사용하여 공용체 별칭을 Box로 정의

Box b1;     // 공용체 별칭으로 공용체 변수 선언

union Box {     // 공용체 정의
    short candy;
    float snack;
    char doll[8];
} b1;           // 공용체를 정의하는 동시에 변수 b1 선언
```

일반적으로 내가 typedef를 많이 쓰므로 그 형태에 맞추어 사용하거나 하자(비트필드와 같이 쓰는 경우 말고는 아직까지 사용해본적 없음)

Union

3. 유니온 예제(1)

union.c

```c
#define _CRT_SECURE_NO_WARNINGS    // strcpy 보안 경고로 인한 컴파일 에러 방지
#include <stdio.h>
#include <string.h>    // strcpy 함수가 선언된 헤더 파일

union Box {    // 공용체 정의
    short candy;     // 2바이트
    float snack;     // 4바이트
    char doll[8];    // 8바이트
};

int main()
{
    union Box b1;   // 공용체 변수 선언

    printf("%d\n", sizeof(b1));  // 8: 공용체의 전체 크기는 가장 큰 자료형의 크기

    strcpy(b1.doll, "bear");     // doll에 문자열 bear 복사

    printf("%d\n", b1.candy);    // 25954
    printf("%f\n", b1.snack);    // 446442825660793851103692822937 6.000000
    printf("%s\n", b1.doll);     // bear

    return 0;
}
```

실행 결과

```
8
25954
446442825660793851103692822937 6.000000
bear
```

Union 키워드 {} 에서
중괄호 뒤에 ;(세미콜론)
안하는 실수 반복됨...주의

유니온(공용체)는 같은
메모리 공간을 공유하기
때문에 결과와 같이
마지막에 쓴 문자열
“bear”를 제외하고는
제대로 된 결과값을 받지
못했다.

Union

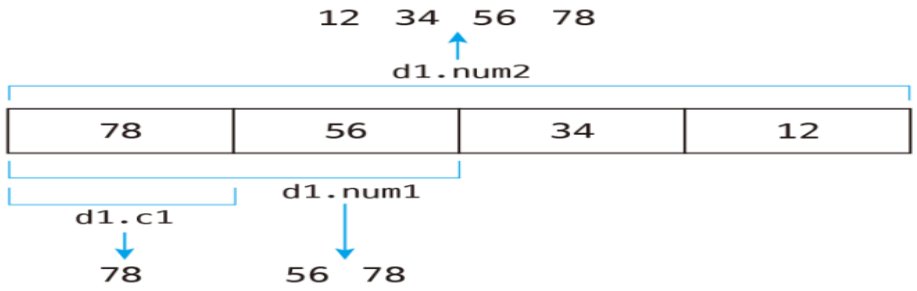
3. 유니온 예제(2)

```c
#include <stdio.h>

union Data {    // 공용체 정의
    char c1;
    short num1;
    int num2;
};

int main()
{
    union Data d1;    // 공용체 변수 선언

    d1.num2 = 0x12345678;    // 리틀 엔디언에서는 메모리에 저장될 때 78 56 34 12로 저장됨

    printf("0x%x\n", d1.num2);    // 0x12345678: 4바이트 전체 값 출력
    printf("0x%x\n", d1.num1);    // 0x5678: 앞의 2바이트 값만 출력
    printf("0x%x\n", d1.c1);      // 0x78: 앞의 1바이트 값만 출력

    printf("%d\n", sizeof(d1));   // 4: 공용체의 전체 크기는 가장 큰 자료형의 크기

    return 0;
}
```

결과
0x12345678
0x5678
0x78
4

Q. 결과값이 빅엔디안과 리틀엔디안 구조에 따라 이 어떤 차이를 보이는가?
A. 빅 엔디안 : 낮은 주소 높은 자리부터
리틀엔디안:낮은 주소 낮은 자리부터

따라서 위의 구조는 리틀엔디안 방식으로



비트 필드 구조체

1. 비트필드 구조체 특징
구조체 멤버를 비트 단위로 저장할 수 있음(구조체와 다른 점)
메모리나 데이터 저장 공간을 효율적으로 사용 가능
비트필드에 주로 unsigned 자료형을 주로 사용(실수형 사용 불가)

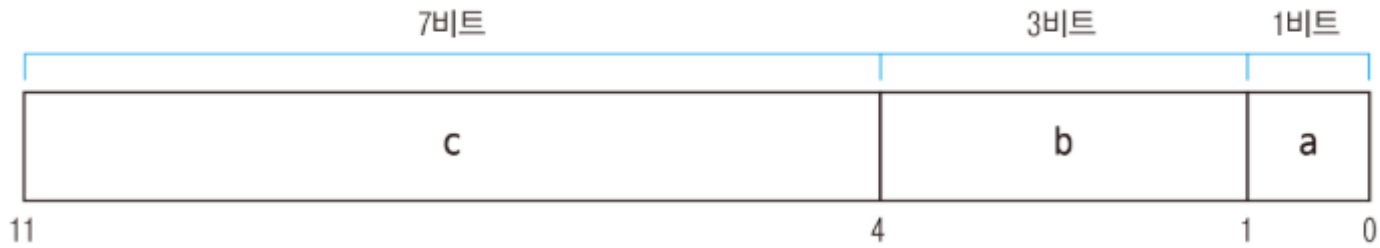
2. 비트필드 구조체 형태

비트 필드는 다음과 같이 멤버를 선언할 때 : (콜론) 뒤에 비트 수를 지정해주면 됩니다.

```
struct 구조체이름 {
    정수자료형 멤버이름 : 비트수;
};
```

이제 구조체를 7비트, 3비트, 1비트로 나눠서 비트 필드를 정의해보겠습니다.

▼ 그림 56-1 구조체 비트 필드



비트 필드 구조체

3. 비트필드 구조체 예제(1)

struct_bit_field.c

```c
#include <stdio.h>

struct Flags {
    unsigned int a : 1;     // a는 1비트 크기
    unsigned int b : 3;     // b는 3비트 크기
    unsigned int c : 7;     // c는 7비트 크기
};

int main()
{
    struct Flags f1;    // 구조체 변수 선언

    f1.a = 1;       //   1: 0000 0001, 비트 1개
    f1.b = 15;      //  15: 0000 1111, 비트 4개
    f1.c = 255;     // 255: 1111 1111, 비트 8개

    printf("%u\n", f1.a);    //   1:       1, 비트 1개만 저장됨
    printf("%u\n", f1.b);    //   7:     111, 비트 3개만 저장됨
    printf("%u\n", f1.c);    // 127: 111 1111, 비트 7개만 저장됨

    return 0;
}
```

실행 결과

```
1
7
127
```

1, 3, 7과 같이 비트를 설정해줌

1 : 1bit , 0~1 표현
3 : 3bit , 0~7 표현
7 : 7bit , 0~255 표현

구조체 크기 : 4Byte

비트 필드 구조체

3. 비트필드 구조체 예제(2)

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

struct {
    unsigned int age : 3;
} Age;

int main()
{
    Age.age = 4;
    printf("Sizeof(Age) : %d\n", sizeof(Age));
    printf("Age.age : %d\n", Age.age);

    Age.age = 7;
    printf("Age.age : %d\n", Age.age);

    Age.age = 8;
    printf("Age.age : %d\n", Age.age);

    return 0;
}
```

결과
Sizeof(Age) : 4
Age.age : 4
Age.age : 7
Age.age : 0

Q. 마지막 결과 Age.age 가 8이 아닌 0인 이유?
A. Bit 3개만 사용해서 최대 0-7까지 표현 가능함

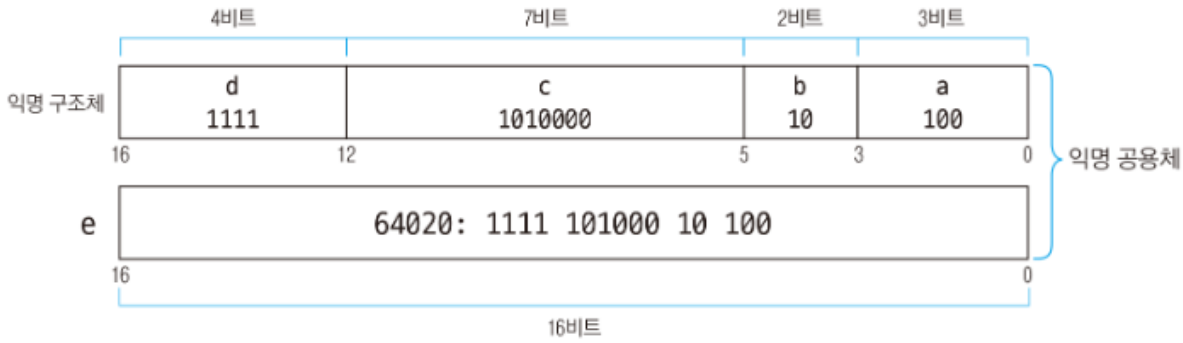
공용체 & 비트필드 함께 사용(실전)

1. 예제(1)

```c
#include <stdio.h>

struct Flags {
    union {    // 익명 공용체
        struct {    // 익명 구조체
            unsigned short a : 3;    // a는 3비트 크기
            unsigned short b : 2;    // b는 2비트 크기
            unsigned short c : 7;    // c는 7비트 크기
            unsigned short d : 4;    // d는 4비트 크기
        };                           // 합계 16비트
        unsigned short e;    // 2바이트(16비트)
    };
};

int main()
{
    struct Flags f1 = { 0, };    // 모든 멤버를 0으로 초기화

    f1.a = 4;     //  4: 0000 0100
    f1.b = 2;     //  2: 0000 0010
    f1.c = 80;    // 80: 0101 0000
    f1.d = 15;    // 15: 0000 1111

    printf("%u\n", f1.e);    // 64020: 1111 1010000 10 100

    return 0;
}
```

결과

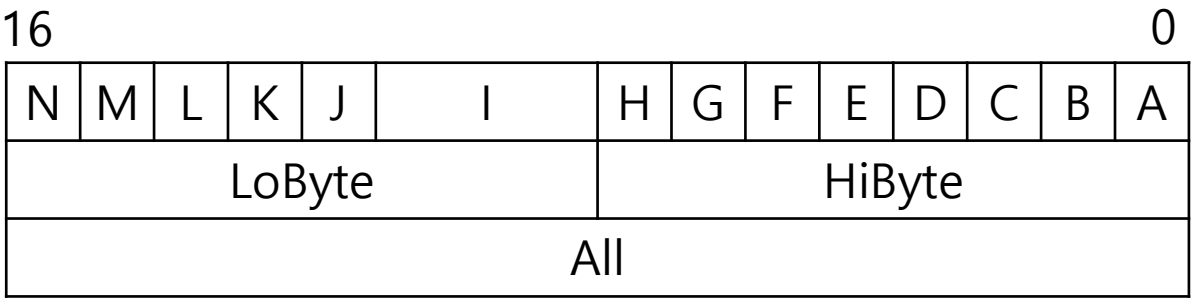
```
f1.a = 4;      //  4: 0000 0100
f1.b = 2;      //  2: 0000 0010
f1.c = 80;     // 80: 0101 0000
f1.d = 15;     // 15: 0000 1111

f1.e = 64020: // 1111 1010000 10 100
```



공용체 & 비트필드 함께 사용(실전)

1. 예제(3)

```
typedef union {
    UINT16 All;
    struct
    {
        UINT16 HiByte : 8;
        UINT16 LoByte : 8;
    }Byte;

    struct
    {
        UINT16 A : 1;
        UINT16 B : 1;
        UINT16 C : 1;
        UINT16 D : 1;
        UINT16 E : 1;
        UINT16 F : 1;
        UINT16 G : 1;
        UINT16 H : 1;

        UINT16 I : 3;
        UINT16 J : 1;
        UINT16 K : 1;
        UINT16 L : 1;
        UINT16 M : 1;
        UINT16 N : 1;
    } Bit;
}USBCREGFlag;
```



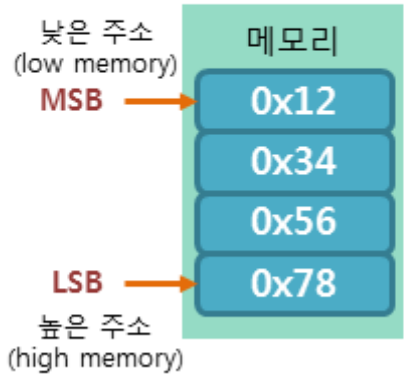
Big Endian VS Little Endian

1. Big Endian
- 낮은 주소에 높은 바이트부터 저장하는 방식
- 메모리에 저장된 순서 그대로 읽을 수 있음, 이해하기 쉬움
- RISC CPU 계열에서 주로 이 방식으로 데이터를 저장하는 편

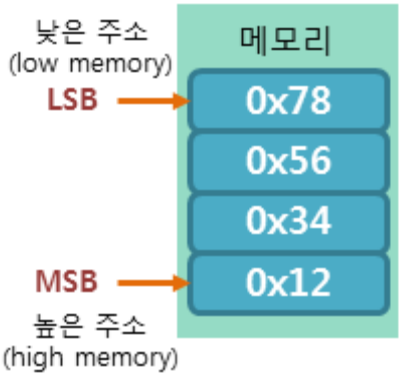
2. Little Endian
- 낮은주소에 낮은 바이트부터 저장하는 방식
- 평소 우리가 생각하는 선형방식과 반대로 거꾸로 읽어야 함
- 대부분의 인텔 CPU 계열에서는 이 방식으로 데이터를 저장

EX) 0x12345678 있을 경우

빅 엔디안(Big Endian)



리틀 엔디안(Little Endian)



Big Endian VS Little Endian

3. Big Endian VS Little Endian

- 단지 저장해야 할 큰 데이터를 어떻게 나누어 저장하는가에 따른 차이일 뿐,
  어느 방식이 더 우수하다고는 단정할 수 없음

- 물리적으로 데이터를 조작하거나 산술연산을 수행할 때에는 리틀 엔디안 방식이 더 효율적
  데이터의 각 바이트를 배열처럼 취급할 때에는 빅 엔디안 방식이 더 적합

- 인텔 기반의 윈도우는 리틀 엔디안 방식을 사용하지만
  네트워크를 통해 데이터를 전송할 때에는 빅 엔디안 방식이 사용됨
  따라서 인텔 기반의 시스템에서 소켓 통신을 할 때는 바이트 순서에 신경을 써서 데이터를 전달해야 함

```c
int i;
int test = 0x12345678;
char* ptr = (char*)&test; // 1 바이트만을 가리키는 포인터를 생성함.

for (i = 0; i < sizeof(int); i++)
{
    printf("%x", ptr[i]); // 1 바이트씩 순서대로 그 값을 출력함.
}
```

78563412 이면 리틀 엔디안 방식
12345678 이면 빅 엔디안 방식

2의 보수

- **CPU는 음수 표현을 위해 2의 보수법 사용**
  => -2 표현을 위해 +2에 2의 보수를 취한다.
  => 2 – 4 표현을 할 때 뺄샘 연산이 없으므로 +4의 2의보수로 -4를 구한 후 2와 더함(0010 + 1100 = 1110(-2)
  => 구한 1110을 정수로 읽으려면 2의 보수 취한다음 – 앞에 붙여주면 -2가 나옴

포인터

0. 포인터 : 단순히 주소값을 담는 변수

1. %x, %p, #

%x는 단순히 16진수로 표현
%p는 16진수를 대문자 빈공간에는 0을 넣어 출력(주소는 이거로 보자!)
#는 0x를 표현함

2. int **pptr = &ptr; // ptr 포인터 변수의 주소를 값으로 담고 있는 변수
// &ptr => 이중포인터
// &&ptr => 존재x

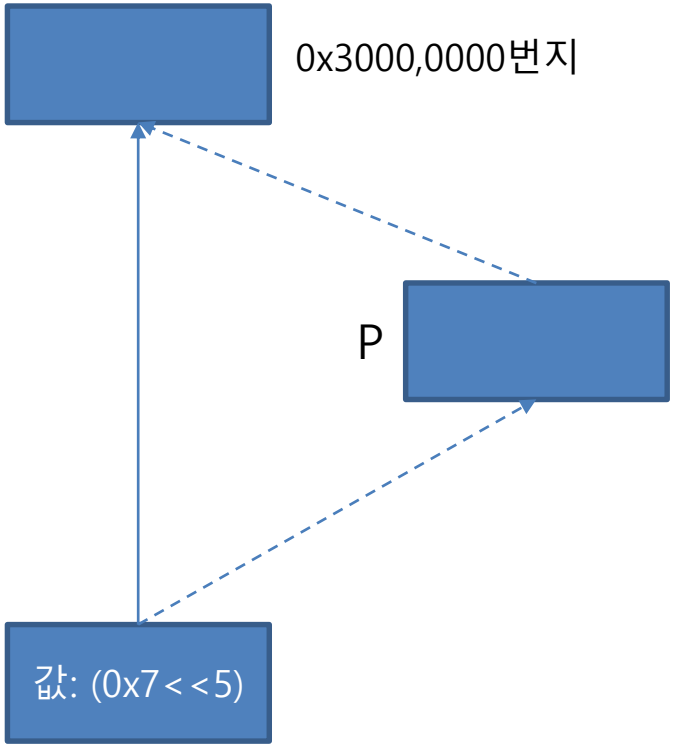
3. 메모리 접근방식(직접 접근 방식 VS 간접 접근 방식)

직접접근방식:
하드웨어 제어하는 프로그램에서 많이 사용됨
메모리 레이아웃을 정확히 알고 사용한다면 속도를 빠르게 하는 프로그래밍 가능

간접접근방식:
대부분의 상의 어플리케이션 개발자에서 많이 사용되는 방식

3. 메모리 접근방식(직접 접근 방식 VS 간접 접근 방식)

```
1)직접 방식
  #define PA (*(volatile unsigned int*)0x30000000)

  void init(void){
  PA |= (0x7 << 5); //메모리 PA[7:5]를 1로 설정
  }

2)간접 방식
  char *p =(char*) 0x30000000;

  void init(void){//메모리 PA[7:5]를 1로 설정
  *P |= (0x7 <<5);
  }
```



직접 VS 간접 비교

4. 포인터 가능연산 VS 불가능연산

정수와의 +,-		단항연산		포 - 포	대입	비교
p+1;	p-1;	p++;	p--;	p-q;	p=q;	If(p>q){}

실수와의 +,-		포인터 끼리의 +,*,/		
p+3.1;	p-3.1;	p+q;	p*q;	p/q;

포인터는 주소이므로 기본적으로 unsigned int 형

p-q는 가능하나 p+q는 불가능한 이유는
⇒ 포인터 끼리의 합은 메모리 어느 위치를 나타낼지, 무엇이 들어 있는 위치를 나타낼지 모르기 때문에, 혹은 영역을 벗어날수도 있고 해서 위험해서
⇒ 컴파일러 단에서 이러한 위험 때문에 막고 있음
⇒ p-q는 두 포인터 사이에 얼마의 단위가 존재하는지, 남아있는지 파악하는 검사를 하는데 유용함

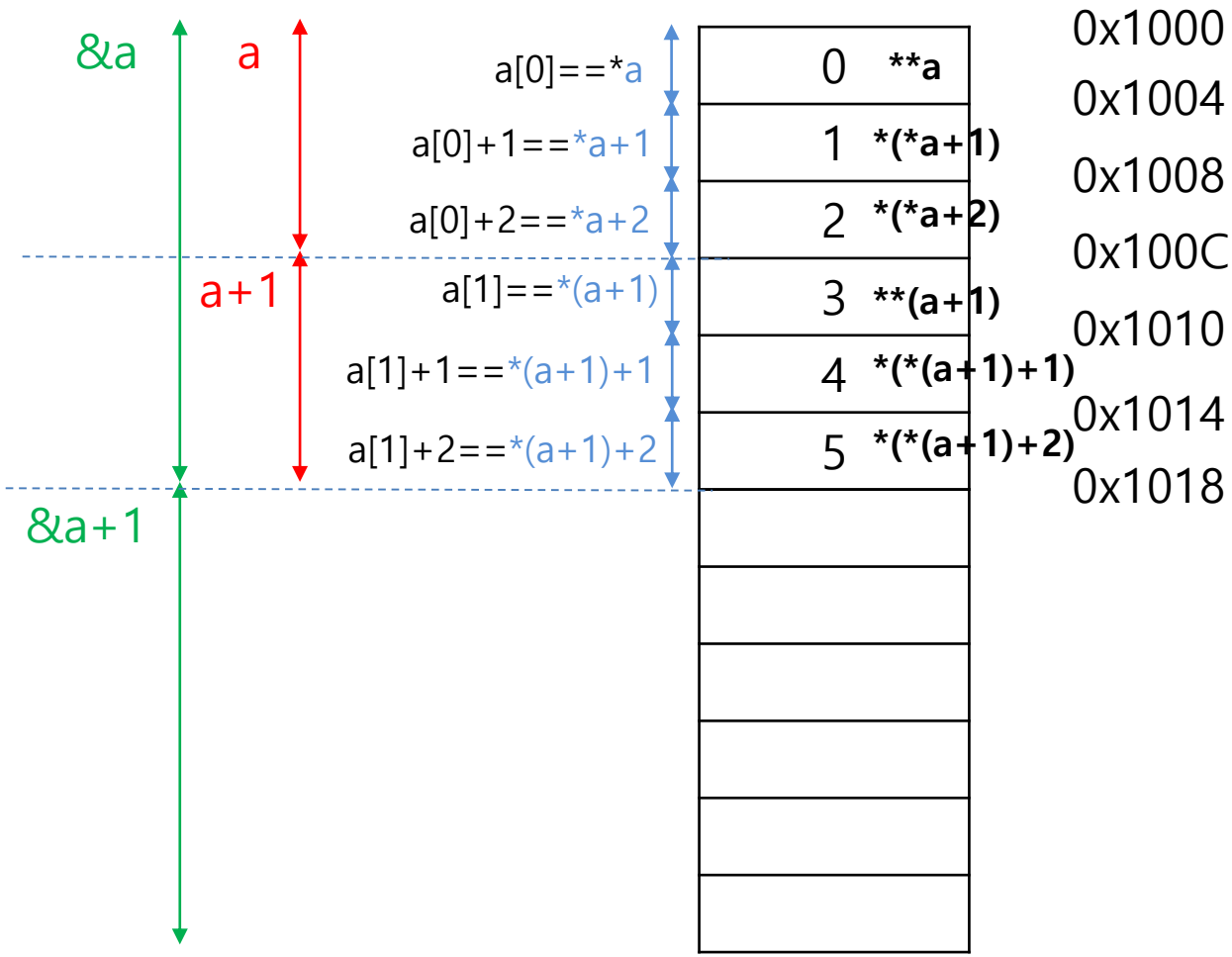
5. 2차원 배열 & 포인터 관계

a[0]은 {0,1,2}를 표현하는 하나의 **배열이름**으로 생각하기
즉, **a[0]은 값이 아닌 첫 요소의 주소 (a[0] == *a : 값이 아닌 주소)**

a[1]은 {3,4,5}를 표현하는 하나의 **배열이름**으로 생각하기
즉, **a[1]은 값이 아닌 첫 요소의 주소 (a[1] == *(a+1) : 값이 아닌 주소)**

Int a[2][3] = {{0,1,2}, {3,4,5}};

&a : a 배열 전체의 시작주소
a : a 배열 첫 요소의 주소
*a : a[0]배열의 첫 요소의 주소



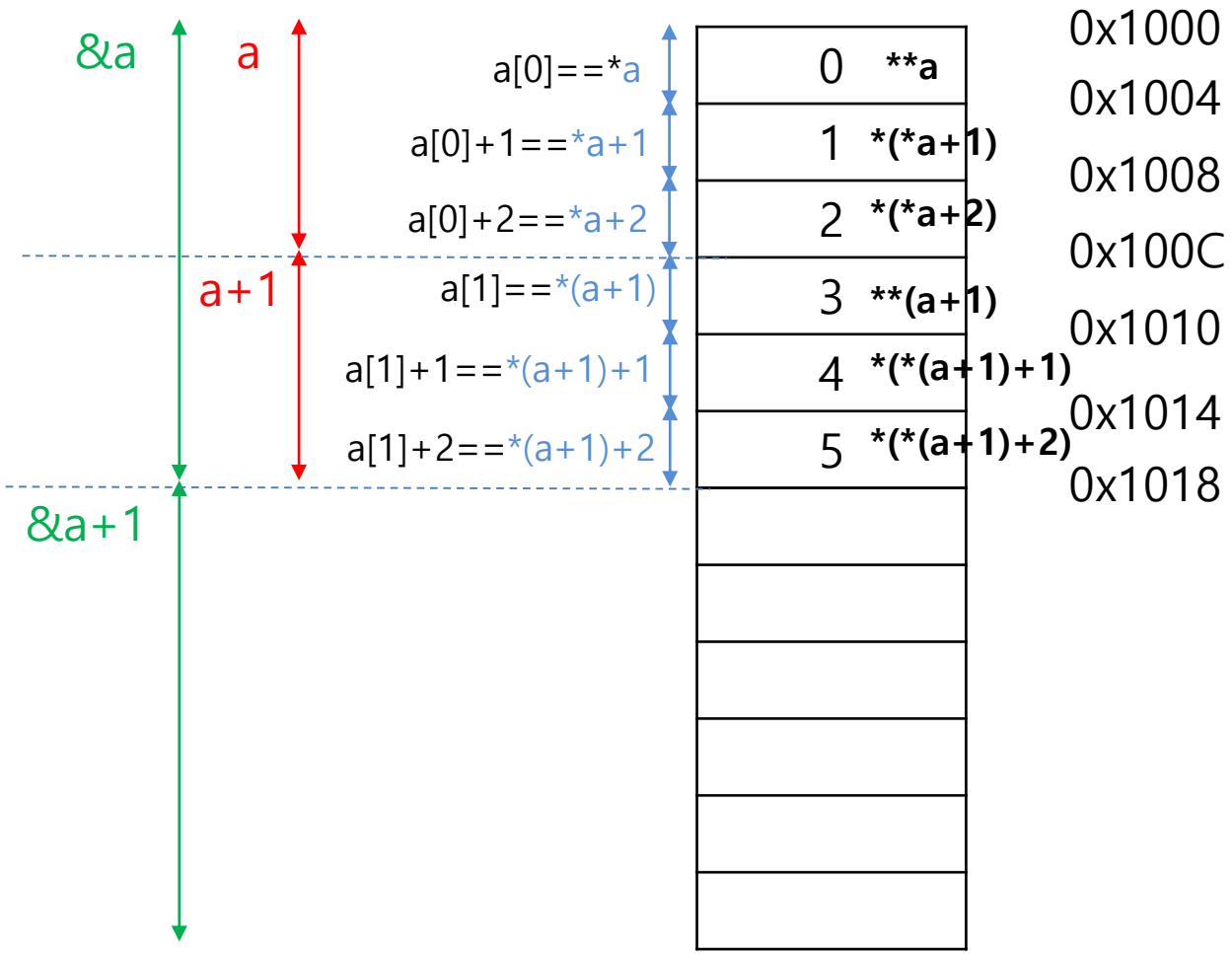
6. 2차원 배열은 배열 포인터로 사용하자!(2차원 포인터 불가능)

배열 포인터 = int (*p)[3]
: Pointer to int[3]

Int a[2][3] = {{0,1,2},{3,4,5}};
Int (*P)[3] = a;

printf("%d",p[i][j]); 로 사용 가능

Int a[2][3] = {{0,1,2}, {3,4,5}};



7. 배열로 선언된 변수를 매개 변수로 사용하는 경우

```c
#include <stdio.h>

void Test(int temp[1000])
{
    printf("Array temp size : %d\n", sizeof(temp));
}

void main()
{
    int data[1000];
    printf("Array data size : %d\n", sizeof(data));
    Test(data);
}
```

결과

Array data size : 4000
Array temp size : 4
4는 예상과 다른 값

- C언어는 함수 호출 시에 배열에 저장된 값을 전달하는 기능은 제공하지 않고, 배열의 주소를 전달하도록 구현
- Temp 배열은 컴파일러가 번역할 때 포인터로 변경해서 사용

7. 배열로 선언된 변수를 매개 변수로 사용하는 경우

```c
#include <stdio.h>

void Test(int temp[1000])
{
    temp++;     // 표기법 혼용과 상관없이 배열은 이런 표현을 사용할 수 없다.
                // 하지만 컴파일시에 오류가 발생하지 않고 정상적으로 동작함.
                // 결국 이 상황에서 C 언어는 temp의 선언 형식과 상관없이
                // 포인터로 사용한다는 뜻이다.
    *temp = 5; // temp가 가리키는 곳에 5를 대입함
                // temp는 처음에 data[0]을 가리키고 있었지만 ++ 연산을 사용했기 때문에
                //  data[1]을 가리키고 있음. 따라서 data[1]이 5로 변경됨
}

void main()
{
    int data[1000] = { 1, 2, };
    // Test 함수를 호출하기 전에 첫 번째 항목과 두 번째 항목의 값을 출력
    printf("data[0] = %d, data[1] = %d\n", data[0], data[1]);
    // 배열의 시작 주소를 Test 함수의 매개 변수로 전달
    Test(data);
    // Test 함수를 호출하고 첫 번째 항목과 두 번째 항목의 값을 출력
    printf("data[0] = %d, data[1] = %d\n", data[0], data[1]);
}
```

결과

```
data[0] = 1, data[1] = 2
data[0] = 1, data[1] = 5
```

보통 temp[1000]에서 1000은
적지 않는것이 일반적

[ ] 안의 숫자는 무시됨

결론

어차피 C언어는 함수의
매개변수로 배열을 전달할 때
강제적으로 포인터 문법을 사용

따라서 그냥 포인터로 변수를
선언해서 배열의 주소를 받아
사용하는 것이 좋음

8. 2차원 배열을 함수의 인자로 넘기는 경우

```c
#include <stdio.h>

// 2차원 배열의 시작 주소를 a_str_list로 전달 받는다.
// 그리고 a_count에는 문자열의 개수가 저장되어 있다.
void PrintString(char a_str_list[][16], int a_count)
{
    // 전달된 문자열을 화면에 출력한다.
    for (int i = 0; i < a_count; i++){
        printf("%s\n", a_str_list[i]);
    }
}

int main()
{
    // 문자열을 저장하는 2차원 배열 선언
    char str_list[3][16] = { "tipssoft", "tipsware", "twdn" };
    // 2차원 배열의 시작 주소를 PrintString 함수로 전달한다.
    PrintString(str_list, 3);
    return 0;
}
```

```c
void PrintString(char (*ap_str_list)[16], int a_count)
{
    // 두 번째 문자열인 'tipsware'의 세 번째 위치에 있는 'p' 문
    char temp = ap_str_list[1][2];
}
```

2차원 배열도 결국 포인터로
주소를 넘기는 것!

위와 같이 char(*p)[16]
형식으로 써주는 것이 좋음

[16]에 16은 무조건 값을
명시해야 함. 그래야 포인터
크기가 얼마인지 알수 있기
때문에

출력은 배열처럼하는게
알아보기 쉬움

9. 배열 포인터 VS 포인터 배열

Int(*P)[3];
: 배열포인터 (배열을 가리키므로)
Int* p [3];
: 포인터 배열(포인터 변수가 배열 형태로)

Int (*p[3])(int);
: 함수 포인터 배열(함수를 가리키므로)

나만의 생각
: 가리키는 대상이 앞으로 감(함수를 가리키면 함수가 앞으로 가서 함수포인터)

함수 포인터

1. 함수포인터
함수이름도 배열이름과 같이 시작주소이자 상수
단순히 함수를 가리키는 변수

2. 함수포인터 형태
Int*p_func(int) (x) : int형 포인터를 반환하는 그냥 함수로 됨
Int(*p_func)(int) (o) : 함수 포인터를 나타냄

3. 함수포인터 기본 예제

```c
#include <stdio.h>

void PrintValue(int a_value)
{
    printf("Value = %d\n", a_value);
}

int main()
{
    // 함수 포인터 p_func 변수를 선언하고 PrintValue 함수의 주소를 대입함!
    void (*p_func)(int) = PrintValue;

    (*p_func)(5);    // 원칙적인 표현!
    p_func(6);       // 허용되는 표현!
    return 0;
}
```

(*p_func)(5)가 포인터 문법상 맞지만 p_func(6)으로 쓰는것이 배열과 연동해서 이해하는데 덜 햇갈리고 한 것 같다.(내 생각)

- 함수의 매개변수로 함수를 받을수 없고 함수 포인터로 써야함
- 구조체의 맴버함수를 커스터마이징할 때

함수 포인터

4. 함수포인터 쓰면 편리한 단편적인 예

< 함수 포인터를 사용하지 않는 경우 >

```
switch (program_count) {
        case    0:      Zero();         break;
        case    1:      One();          break;
        case    2:      Two();          break;
        case    3:      Three();        break;
        case    4:      Four();         break;
        case    5:      Five();         break;
        case    6:      Six();          break;
        case    7:      Seven();        break;
        case    8:      Eight();        break;
        case    9:      Nine();         break;
}
```

< 함수 포인터를 사용하는 경우 >

```
void (*fp[])() = { Zero, One, Two, Three, Four, Five, Six, Seven, Eight,
        fp[program_count]();
```

임베디드 환경에서 디바이스의 종류별로 함수를 다르게 호출하는 경우가 많아서
함수 포인터 호출하는 경우가 빈번하여 확실히 이해하면 좋을 듯

5. 함수포인터 왜 쓰는가? 왜 알아야 하는가?

https://blog.naver.com/tipsware/221286052738

함수 포인터

6. 상수 함수 포인터

```
Int(*const p_func)(int)
```

7. 함수 포인터 배열

```
Int (*fun[3])(int, int);
```

Int형 자료 두 개를 입력 받아 int형 결과를 돌려주는
함수 포인터 3개를 저장할 수 있는 배열

```
Ex) fun[0] = add;    //    ==fun[0]=&add;
    fun[1] = mul;
    fun[2] = sub;
```

fun[0] 함수 포인터는 add 함수를 가리킴
fun[1] 함수 포인터는 mul 함수를 가리킴
fun[2] 함수 포인터는 sub 함수를 가리킴

함수 포인터

8. 함수포인터 실전 활용(구조체로 활용)

```
typedef struct LensCmd
{
    Lens_Cmd Cmd;
    void(*handler)(Lens_Type, Sys_DevMsg *);
    Lens_Type type;
} LensCmd;
```

```
void Control_LensCall(Sys_DevMsg *pDevMsg)
{
    LensCmd *cmdptr;

    LastLensMsg = *pDevMsg;

    for(cmdptr = lensCmds; cmdptr->Cmd; cmdptr++)
    {
        if(LastLensMsg.devCmd == cmdptr->Cmd)
        {
            LastLensMsg.devResult = SYS_DEV_CMD_RESULT_NONE;

            cmdptr->handler(cmdptr->type, &LastLensMsg);
            /*PSH*/
            if( LastLensMsg.devResult != SYS_DEV_CMD_RESULT_NONE)
            {
                Enqueue_SysResp(&LastLensMsg);
            }
            break;
        }
    }

    if(cmdptr->Cmd == false)
    {
        LastLensMsg.devResult = SYS_DEV_CMD_RESULT_FAIL;
        Enqueue_SysResp(&LastLensMsg);
    }
}
```

```
LensCmd lensCmds[] = {
    {LENS_CMD_SET_ZOOM_START_POSITION,      Lens_SetStartPosition,      LENS_ZOOM},
    {LENS_CMD_SET_FOCUS_START_POSITION,     Lens_SetStartPosition,      LENS_FOCUS},
    {LENS_CMD_SET_IRIS_START_POSITION,      Lens_SetStartPosition,      LENS_IRIS},

    {LENS_CMD_GET_ZOOM_START_POSITION,      Lens_GetStartPosition,      LENS_ZOOM},
    {LENS_CMD_GET_FOCUS_START_POSITION,     Lens_GetStartPosition,      LENS_FOCUS},
    {LENS_CMD_GET_IRIS_START_POSITION,      Lens_GetStartPosition,      LENS_IRIS},

    {LENS_CMD_SET_ZOOM_TARGET_POSITION,     Lens_SetTartgetPosition,    LENS_ZOOM},
    {LENS_CMD_SET_FOCUS_TARGET_POSITION,    Lens_SetTartgetPosition,    LENS_FOCUS},
    {LENS_CMD_SET_IRIS_TARGET_POSITION,     Lens_SetTartgetPosition,    LENS_IRIS},

    {LENS_CMD_GET_ZOOM_CURRENT_POSITION,    Lens_GetCurPosition,        LENS_ZOOM},
    {LENS_CMD_GET_FOCUS_CURRENT_POSITION,   Lens_GetCurPosition,        LENS_FOCUS},
    {LENS_CMD_GET_IRIS_CURRENT_POSITION,    Lens_GetCurPosition,        LENS_IRIS},

    {LENS_CMD_SET_ZOOM_MAX_POSITION,        Lens_SetMaxPosition,        LENS_ZOOM},
    {LENS_CMD_SET_FOCUS_MAX_POSITION,       Lens_SetMaxPosition,        LENS_FOCUS},
    {LENS_CMD_SET_IRIS_MAX_POSITION,        Lens_SetMaxPosition,        LENS_IRIS},

    {LENS_CMD_GET_ZOOM_MAX_POSITION,        Lens_GetMaxPosition,        LENS_ZOOM},
    {LENS_CMD_GET_FOCUS_MAX_POSITION,       Lens_GetMaxPosition,        LENS_FOCUS},
    {LENS_CMD_GET_IRIS_MAX_POSITION,        Lens_GetMaxPosition,        LENS_IRIS},

    {LENS_CMD_SET_ZOOM_DUTY,                Lens_SetDuty,               LENS_ZOOM},
    {LENS_CMD_SET_FOCUS_DUTY,               Lens_SetDuty,               LENS_FOCUS},
    {LENS_CMD_SET_IRIS_DUTY,                Lens_SetDuty,               LENS_IRIS},

    {LENS_CMD_GET_ZOOM_DUTY,                Lens_GetDuty,               LENS_ZOOM},
    {LENS_CMD_GET_FOCUS_DUTY,               Lens_GetDuty,               LENS_FOCUS},
    {LENS_CMD_GET_IRIS_DUTY,                Lens_GetDuty,               LENS_IRIS},

    {false,                                 NULL,                       NULL}
};
```

함수 포인터

8. 함수포인터 실전 활용(typedef 사용, 함수인자로)

```c
#include <stdio.h>

// 함수포인터 타입 정의
typedef int (*calcFuncPtr)(int, int);


// 덧셈 함수
int plus (int first, int second)
{
    return first + second;
}
// 뺄셈 함수
int minus (int first, int second)
{
    return first - second;
}
// 곱셈 함수
int multiple (int first, int second)
{
    return first * second;
}
// 나눗셈 함수
int division (int first, int second)
{
    return first / second;
}
```

```c
// 매개변수로 함수포인터를 갖는 calculator 함수
int calculator (int first, int second, calcFuncPtr func) //함수 포인터로 주소 받음
{
    return func (first, second);     // 함수포인터소 사용
}

int main(int argc, char** argv)
{
    calcFuncPtr calc = NULL;
    int a = 0, b = 0;
    char op = 0;
    int result = 0;

    scanf ("%d %c %d", &a, &op, &b);

    switch (op)    // 함수포인터 calc에 op에 맞는 함수들의 주소를 담음
    {
        case '+' :
            calc = plus;
            break;

        case '-':
            calc = minus;
            break;

        case '*':
            calc = multiple;
            break;

        case '/':
            calc = division;
            break;
    }

    result = calculator (a, b, calc); //calc 는 함수 이름이므로 포인터 주소

    printf ("result : %d", result);

    return 0;
}
```

8. 함수포인터 실전 활용(함수 포인터 배열)

```c
#include <stdio.h>

int a(int);
int b(int);
int c(int);

int (*p[3])(int) = {a, b, c};  //함수 포인터 배열

void main(void)
{
  int x, y, z, i;
  printf("\n메뉴\n1. 제곱\n");
  printf("\n메뉴\n2. 3제곱\n");
  printf("\n메뉴\n3. 4제곱\n");
  printf("\n원하는 동작을 선택하시오\n");
  scanf("%d", &i);

  //선택한 메뉴에 따라 배열 첨자를 이용해 함수 호출
  z = p[i-1](4);

  printf("%d\n", &z);
}

int a(int k){
  return k*k;
}

int b(int k){
  return k*k*k;
}

int c(int k){
  return k*k*k*k;
}
```

함수포인터 활용하여

자칫 길게 swtich case문으로 구성할수 있는 것을 간단히 한줄로 표현함

비교 구문이 생략되고, i가 배열 첨자로 활용되어 즉시 함수가 호출됨

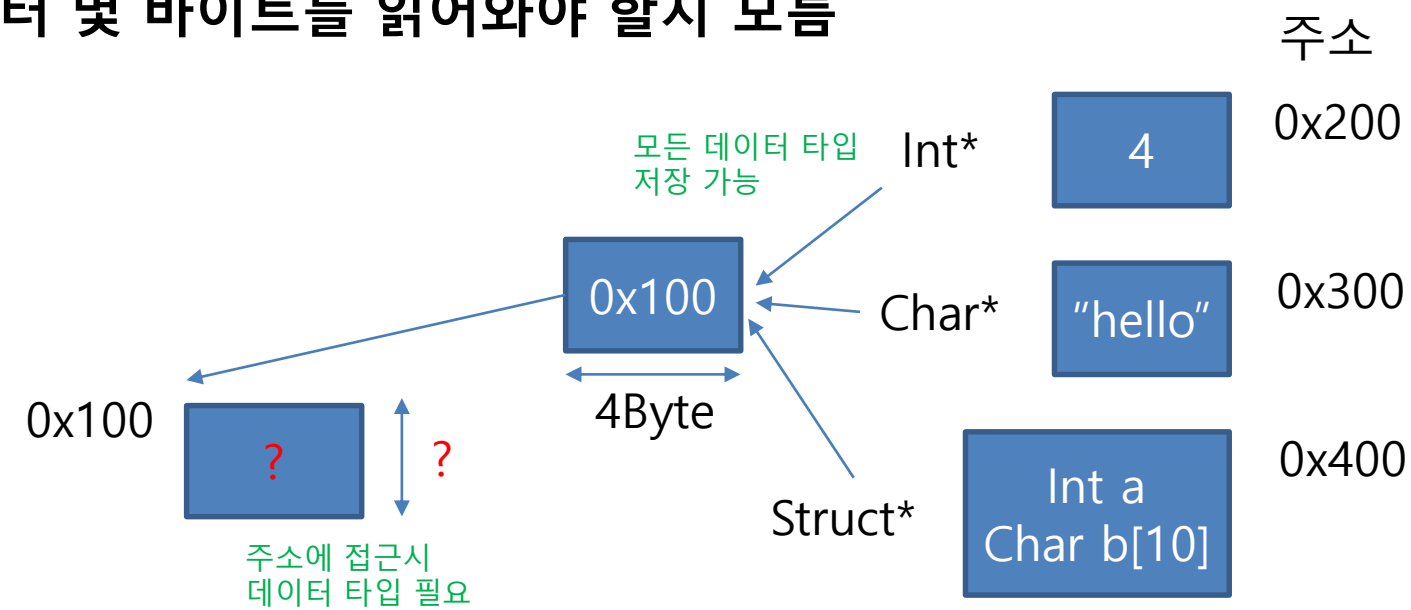
void 포인터

- **특징**
- **저장은 가능하지만 연산에 사용 못함**
- **모든 자료형의 주소 저장가능**

- **문제점**
- **Void 포인터가 증감할때:**
  => 증감에 사용할 한 단위의 크기 모름

- **포인터를 이용해 주소 안의 데이터를 읽어올 때:**
  => 시작주소로부터 몇 바이트를 읽어와야 할지 모름

- **해결방법**
- **캐스팅 기법 사용**



void 포인터 예제

```c
#include <stdio.h>

//여러 타입의 데이터를 입력 받기 위해 인자의 형을 void 포인터로 지정
void add(void *p, void *q, void *s, int op);

void main (void)
{
    int a =1, b =2, sum_i;
    float x = 1.5, y = 2.5, sum_f;

    add(&a, &b, &sum_i, 1);
    add(&x, &y, &sum_f, 2);

    printf("int의 합=%d\n", sum_i);
    printf("float의 합=%f\n", sum_f);
}

void add(void *p, void *q, void *s, int op)
{
    if (op == 1)
    {
        *(int*)s = *(int*)p + *(int*)q ; //void 포인터는 연산 시 캐스팅
    }
    else if(op == 2)
    {
        *(float*)s = *(float*)p + *(float*)q;
    }
}
```

- 함수의 인자로 어떤 타입이 들어올지 모를 경우 void포인터로 모든 형 받아들이도록 함
- 연산시에는 원래의 데이터 타입으로 캐스팅 해야 연산이 가능함

void 포인터 예제(malloc)

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

void main(void)
{
    int a, i, *p;

    printf("저장하고자 하는 수의 개수를 입력하시오\n");
    scanf("%d", &i);
    /*malloc 함수는 메모리 크기를 입력받아 힙영역에  그 크기만큼
      메모리를 할당하고 시작주소를 넘겨줌
    */
    p = (int*)malloc(sizeof(int)*i);

    for(a = 0; a < i; a++)
        p[a] = a;
    for(a = 0; a < i; a++)
        printf("p[%d] = %d\n", a, p[a]);

    free(p);
}
```

- Malloc 함수는 메모리 크기를 입력받아 힙 영역에 그 크기만큼 메모리를 할당하고 시작주소를 넘겨줌
- void *malloc(size_t size);
  size_t 는 문자열길이나 메모리 블록 크기에 사용(unsigned int)
  sizeof함수의 반환형이 unsigned int == size_t임을 알 수 있는 대목

void 포인터 예제(malloc)

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

void main(void)
{
    int a, i, *p;

    printf("저장하고자 하는 수의 개수를 입력하시오\n");
    scanf("%d", &i);
    /*malloc 함수는 메모리 크기를 입력받아 힙영역에  그 크기만큼
      메모리를 할당하고 시작주소를 넘겨줌
    */
    p = (int*)malloc(sizeof(int)*i);

    for(a = 0; a < i; a++)
        p[a] = a;
    for(a = 0; a < i; a++)
        printf("p[%d] = %d\n", a, p[a]);

    free(p);
}
```

- Malloc 함수는 메모리 크기를 입력받아 힙 영역에 그 크기만큼 메모리를 할당하고 시작주소를 넘겨줌
- void *malloc(size_t size);
  size_t 는 문자열길이나 메모리 블록 크기에 사용(unsigned int)
  sizeof함수의 반환형이 unsigned int == size_t임을 알 수 있는 대목

포인터-이중포인터

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#if 1
void InputName(char **pName)

{
    *pName=(char *)malloc(12);
    strcpy(*pName,"Cabin");
}

int main()

{
    char *Name;

    InputName(&Name);

    printf("이름은 %s입니다\n",Name);

    free(Name);

    return 0;
}
#endif
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#if 0
void InputName(char **pName)

{
    *pName=(char *)malloc(12);
    strcpy(*pName,"Cabin");
}

int main()

{
    char *Name;

    InputName(&Name);

    printf("이름은 %s입니다\n",Name);

    free(Name);

    return 0;
}
#endif

#if 1
void InputName(char *pName)
{
    printf("%p\r\n", pName);
    pName=(char *)malloc(12);
    strcpy(pName,"Cabin");
    printf("%p\r\n", pName);
}

int main()
{
    char *Name;

    InputName(Name);

    printf("이름은 %s입니다\n",Name);

    free(Name);

    return 0;
}
#endif
```

- 되는 Case : char *Name의 주소 레퍼런스를 함수인자로 받은 경우이고
- 안되는 Case : char *Name의 값을 함수 인자로 받았기 때문에 함수 안에서의 주소값 할당 및 변경이 함수 밖 main문에서 까지 이어지지 못함

포인터-문자열포인터

```c
#include <stdio.h>
void main(void)
{
    char a[] = "rose";
    char *p = "grace";

    a[0] = 'n';
    p[0] = 't';

    printf("a = %s\n", a);
    printf("p = %s\n", p);
}
```

Text Segment: 문자열 상수 저장, 변경 불가능, 시작 주소를 이용해 접근 가능

- a[] 배열로 선언:

⇒시작주소로부터 텍스트 영역의 "rose"를 불러와 Stack에 있는 배열 a에 저장한다.

- *p 문자열 포인터 선언:

=>"grace" 문자열 상수가 저장된 텍스트 세그먼트 영역에서 변경을 하려고 하기 때문에 변경 불가능

감사합니다.