

15-01. 이중 포인터와 배열 포인터

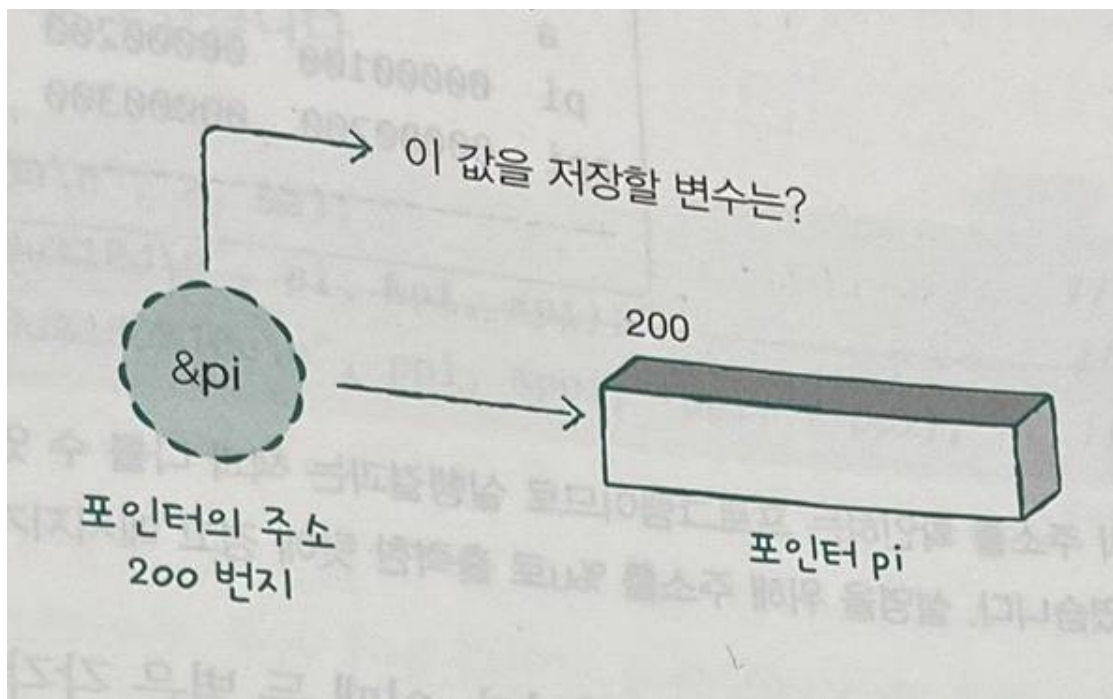
포인터를 사용하는 목적은 가리키는 데이터를 사용하는 것인데, 이번 장부터는 주소 값 자체를 처리할 데이터로 생각해보자.

즉, 주소를 저장한 포인터도 하나의 변수이고 따라서 그 주소를 구할 수 있으면 다른 포인터에 저장하고 가리키는 것도 가능하다.

- 이중 포인터 개념

주소 연산으로 포인터의 주소도 구할 수 있다.

예로, 어떤 변수를 가리키는 포인터 `pi`가 있고, 포인터 `pi`가 할당된 메모리의 시작 위치가 200 일 때 그 주소를 구하면 다음과 같다.



=> '포인터의 주소 `&pi = 200`'를 저장할 변수는?

이 주소를 저장하는 포인터를 **이중 포인터**라고 한다. 즉, 포인터의 주소는 이중 포인터에 저장하며 포인터를 가리킨다.

포인터의 주소가 저장된 이중 포인터에 간접 참조 연산을 수행하면 가리키는 대상인 포인터를 쓸 수 있다.

- 예시

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

```
int main(void)
```

```
{
```

```
    int a = 10;
```

```
    int* pi;
```

```
    int** ppi;
```

```
    pi = &a;
```

```
    ppi = &pi;
```

```
    printf("-----\n");
```

```
    printf("변수 변수값 &연산 *연산 **연산\n");
```

```
    printf("-----\n");
```

```
    printf("a%10d%10u\n", a, &a);
```

```
    printf("pi%10u%10u%10d\n", pi, &pi, *pi);
```

```
    printf("ppi%10u%10u%10u%10d\n", ppi, &ppi, *ppi, **ppi);
```

```
    printf("-----\n");
```

```
    return 0;
```

```
}
```

Microsoft Visual Studio 디버거 콘솔

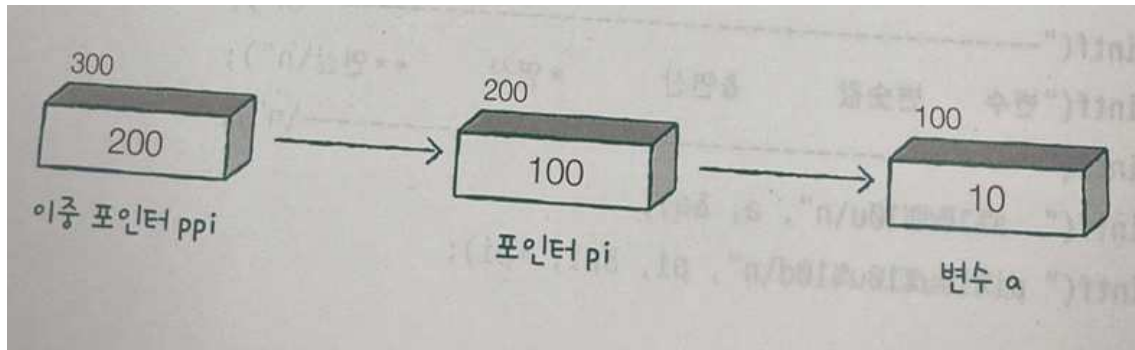
변수 변수값 &연산 *연산 **연산

| | | | | |
|-----|----------|----------|----------|----|
| a | 10 | 11532336 | | |
| pi | 11532336 | 11532324 | 10 | |
| ppi | 11532324 | 11532312 | 11532336 | 10 |

C:\Users\hnu612\source\repos\Project2\Debug\Proj
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...

int **ppi:에서 첫 번째 별은 ppi가 가리키는 자료형이 포인터임을 뜻하며 두 번째 별은 ppi 자신이 포인터임을 뜻한다. 즉, 이중 포인터이다.

이중 포인터를 선언하여 메모리에 저장 공간이 할당되면 그 이후에 이중 포인터를 사용할 때는 변수명을 쓴다.



변수 a를 저장한 포인터 ip의 주소를 저장한 이중 포인터 ppi이다.

이제 다음 그림을 보며 원칙을 적용하면 포인터 연산을 이해할 수 있다.

규칙1. 포인터를 변수명으로 쓰면 그 안의 값이 된다.

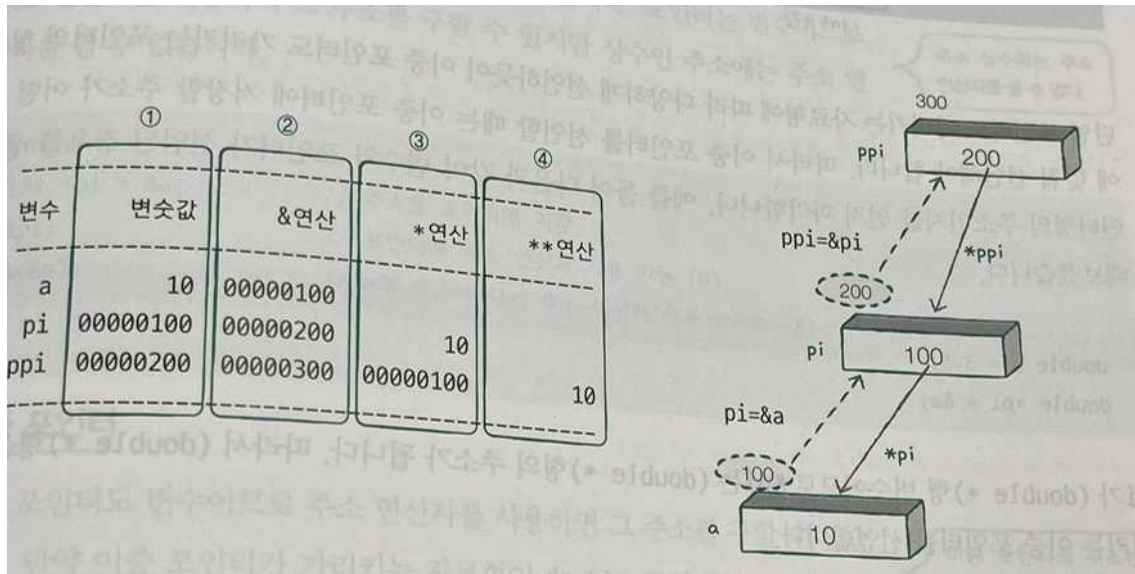
규칙2. 포인터에 &연산을 하면 포인터 변수의 주소가 된다.

규칙3. 포인터의 *연산은 화살표를 따라간다.

```
printf("a%10d%10u\n", a, &a);
```

```
printf(" pi%10u%10u%10d\n", pi, &pi, *pi);
```

```
printf(" ppi%10u%10u%10u%10d\n", ppi, &ppi, *ppi, **ppi);
```



1. pi와 ppi가 변수명으로 사용되어 그 안의 값이 된다.
2. pi와 ppi에 & 연산을 한 결과는 자신의 주소값을 의미한다.
3. ppi에 * 연산을 하면 ppi가 가리키는 대상 pi를 뜻한다.
4. ppi에 ** 연산을 하면 ppi가 가리키는 pi가 가리키는 대상이 된다.

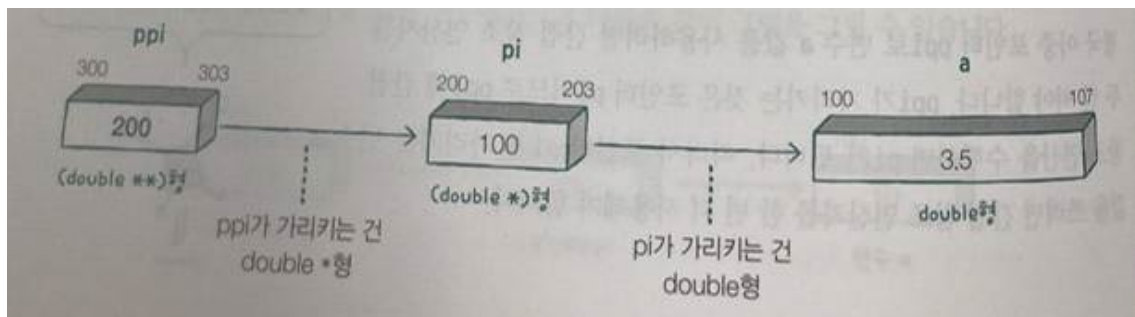
ppi로 변수 a를 사용하려면 간접 참조 연산자를 두 번 써야 한다. ppi가 가리키는 것이 포인터 pi이기 때문에 pi가 가리키는 변수 a를 쓰려면 간접 참조 연산자를 한 번 더 사용해야 한다.

- 이중 포인터의 형태

포인터에서 형태를 얘기할 때는 두 가지를 구분 해야 한다.

포인터가 가리키는 것의 형태와 포인터 자신의 형태이다.

예로, int형 변수의 주소를 저장하는 포인터는 가리키는 자료형이 int형이고 자신의 형태는 (int *)형이 되기 때문이다.



위 사진처럼 이중 포인터를 지정하면 ppi가 가리키는 자료형은 (double *)형이고 pi가 가리키는 자료형은 double형이다.

요 사진에서 ppi의 byte가 4byte인 이유는 선언할 때 포인터 앞에 double을 적어주는 건 포인터가 가리키는 자료형에 대한 정보일 뿐 포인터 자체를 의미하는게 아니기 때문이다. 따라서 값이 100,200이니 4byte로 크기가 결정나는 것이다.

- 주소와 포인터의 차이

포인터 : 변수이므로 주소 연산자를 사용하여 그 주소를 구할 수 있다.

주소 : 상수이므로 주소 연산자를 쓸 수 없다.

예) int a;

int *pi = &a;

π 가능!!!!

&(&a) 안대!!!!

- 다중 포인터

이중 포인터도 변수이므로 주소 연산자를 사용하면 그 주소를 구할 수 있다. 만약 이중 포인터가 가리키는 자료형이 double 포인터일 때 이중 포인터를 가리키는 3중 포인터는 다음과 같이 선언한다. => double ***ppp

같은 방식으로 진행하는데 가독성을 떨어뜨리기 때문에 이중 이상의 포인터를 다중 포인터라고 부른다.

- 이중 포인터 활용1 : 포인터 값을 바꾸는 함수의 매개변수

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

```
void swap_ptr(char** ppa, char** ppb);
```

```
int main(void)
```

```
{
```

```
char* pa = "success";
```

```
char* pb = "failure";
```

```
printf("pa->%s, pb->%s\n", pa, pb);
```

```
swap_ptr(&pa, &pb);
```

```
printf("pa->%s, pb->%s\n", pa, pb);
```

```
return 0;
```

```
}
```

```
void swap_ptr(char** ppa, char** ppb)
```

```
{
```

```
char* pt;
```

```
pt = *ppa;
```

```
*ppa = *ppb;
```

```
*ppb = pt;
```

```
}
```

Microsoft Visual Studio 디버깅

pa->success, pb->failure

pa->failure, pb->success

C:\Users\whu612\source\repos

디버깅이 중지될 때 콘솔을 가

하도록 설정합니다.

이 창을 닫으려면 아무 키나

문자열을 바꿔 출력하지만 문자열 자체를 바꾸지 않는다.

문자열을 연결하는 포인터의 값을 바꾸면 상태가 바뀌므로 이후에 포인터를 사용하면 마치 문자열을 바꾼 것처럼 사용할 수 있다.

- 이중 포인터 활용 2 : 포인터 배열을 매개변수로 받는 함수

```
#include <stdio.h>

void print_str(char** pps, int cnt);

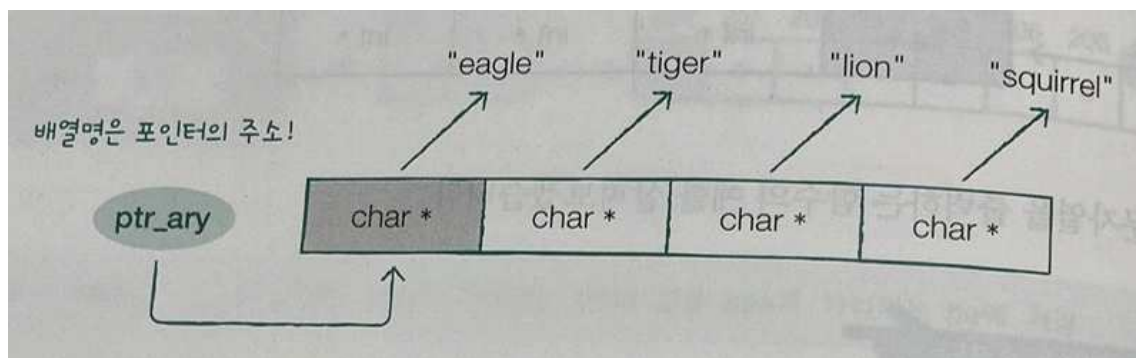
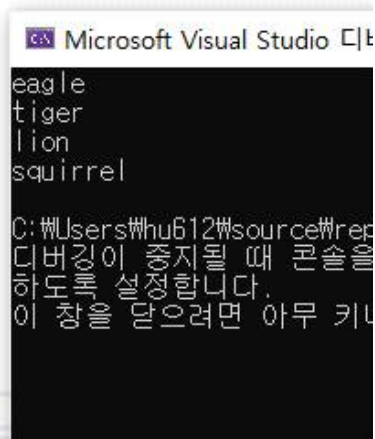
int main(void)
{
    char* ptr_ary[] = { "eagle", "tiger", "lion", "squirrel" };
    int count;

    count = sizeof(ptr_ary) / sizeof(ptr_ary[0]);
    print_str(ptr_ary, count);

    return 0;
}

void print_str(char** pps, int cnt)
{
    int i;

    for (i = 0; i < cnt; i++)
    {
        printf("%s\n", pps[i]);
    }
}
```



- 배열 요소의 주소와 배열의 주요

배열명을 첫 번째 요소의 주소로 사용해왔다. 이제는 배열 전체를 하나의 변수로 생각하고 그 주소를 구해보자.

배열에 주소 연산자를 사용하면 배열을 가리키는 주소이다.

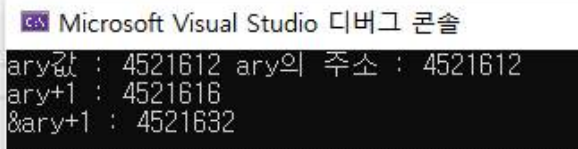
- 예시


```
#include <stdio.h>

int main(void)
{
    int ary[5];

    printf("ary값 : %u\n", ary);
    printf("ary의 주소 : %u\n", &ary);
    printf("ary+1 : %u\n", ary + 1);
    printf("&ary+1 : %u\n", &ary + 1);

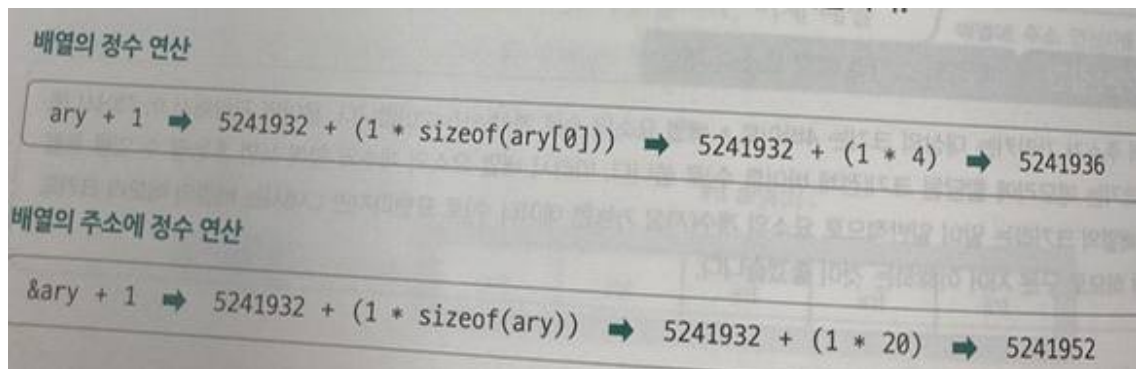
    return 0;
}
```



ary 자체가 주소로 쓰일 때는 첫 번째 요소를 가리키므로 가리키는 대상의 크기는 4이다. 하지만 배열의 주소 &ary는 배열 전체를 가리키므로 가리키는 대상의 크기가 20이 된다. 위를 자세히 이해하려면 규칙을 먼저 이해 해야 한다.

규칙1. 배열은 전체가 하나의 논리적인 변수이다.

규칙2. 배열의 주소에 상수를 더하면 배열 전체의 크기를 곱해서 더한다.



1차원 배열에서는 굳이 하지 않는다.

- 2차원 배열과 배열 포인터

2차원 배열은 1차원 배열로 이루어져 있으므로 논리적인 배열 요소가 1차원 배열이다.

또한 배열명은 첫 번째 요소의 주소이므로 2차원 배열의 이름은 1차원 배열의 주소이며 배열을 가리키는 포인터에 저장된다.


- 예시

```
#include <stdio.h>

int main(void)
{
    int ary[3][4] = { {1,2,3,4},{5,6,7,8},{9,10,11,12} };
    int(*pa)[4];
    int i, j;
    pa = ary;

    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 4; j++)
        {
            printf("%5d", pa[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```



Microsoft Visual Studio 디버그 콘솔

```
1 2 3 4
5 6 7 8
9 10 11 12
```

C:\Users\Wuhu612\source\repos\Project2\WD...
디버깅이 중지될 때 콘솔을 자동으로 닫으...

int (*pa)[4]; : 가리키는 것은 int 4개의 1차원 배열이고 pa는 포인터이다.
pa = ary; : 포인터에 배열명을 저장하면 배열처럼 쓸 수 있다.

- 예시

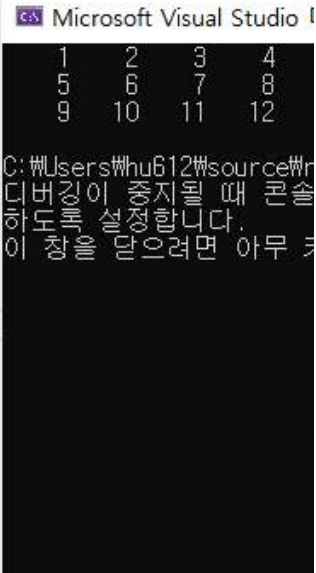
```
#include <stdio.h>

void print_ary(int(*)[4]);

int main(void)
{
    int ary[3][4] = { {1,2,3,4},{5,6,7,8},{9,10,11,12} };
    print_ary(ary);
    return 0;
}

void print_ary(int(*pa)[4])
{
    int i, j;

    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 4; j++)
        {
            printf("%5d", pa[i][j]);
        }
        printf("\n");
    }
}
```



Microsoft Visual Studio 디버그 콘솔

```
1 2 3 4
5 6 7 8
9 10 11 12
```

C:\Users\Wuhu612\source\repos\Project2\WD...
디버깅이 중지될 때 콘솔을 자동으로 닫으...
하도록 설정합니다.
이 창을 닫으려면 아무 키를 누르세요.

- 2차원 배열 요소의 두 가지 의미

2차원 배열은 논리적으로 1차원의 부분 배열을 뜻하고 물리적으로는 실제 데이터를 저장하는 부분 배열의 요소를 뜻한다.

```
int ary[3][4];
```

2차원 배열 ary의 논리적 배열 요소의 개수는? 3개

2차원 배열 ary의 물리적 배열 요소의 개수는? 12개

- 2차원 배열의 요소를 참조하는 원리

12개의 배열 요소에서 7번째 요소를 꺼내는 원리

$\text{ary} + 1 \rightarrow 100 + (1 * \text{sizeof}(\text{ary}[0])) \rightarrow 100 + (1 * 16) \rightarrow 116$

$*(\text{ary}+1) = \text{ary}[1]$ 이므로 여기에 2를 더하면 된다.

$*(\text{ary}+1)+2 \rightarrow *(\text{ary}+1)+(2 * \text{sizeof}(\text{ary}[1][0])) \rightarrow 116 + (2 * 4) \rightarrow 124$

$*(*(\text{ary}+1)+2) \rightarrow \text{ary}[1][2]$ //두 번째 부분배열의 세 번째 배열 요소

2차원 배열 `int ary[3][4];`에서 다음 주소는 모두 같은 값을 가진다.

`&ary` // 2차원 배열 전체의 주소

`ary` // 첫 번째 부분 배열의 주소

`&ary[0]` // 첫 번째 부분 배열의 주소

`ary[0]` // 첫 번째 부분 배열의 첫 번째 배열 요소의 주소

`&ary[0][0]` // 첫 번째 부분 배열의 첫 번째 배열 요소의 주소

- 마무리

✓ 포인터도 하나의 변수이므로 그 주소가 있다.

✓ 이중 포인터에 간접 참조 연산자 `*`를 사용하면 단일 포인터가 된다.

✓ 2차원 배열의 배열명은 첫 번째 부분 배열의 주소가 된다.

✓ 배열 포인터에 간접 참조 연산자를 사용하면 가리키는 배열이 된다.

| 구분 | 기능 | 설명 |
|--------|-------|------------------------------|
| 이중 포인터 | 선언 방법 | <code>int **p;</code> |
| | 사용 예1 | 포인터를 교환하는 함수의 매개변수로 사용 |
| | 사용 예2 | 포인터 배열을 처리하는 함수의 매개변수로 사용 |
| 배열 포인터 | 선언 방법 | <code>int (*pa)[4];</code> |
| | 사용 예1 | int형 변수 4개짜리 1차원 배열을 가리킨다. |
| | 사용 예2 | 2차원 배열의 배열명을 받는 함수의 매개변수에 사용 |

- 확인 문제

15-2. 함수 포인터와 void 포인터

프로그램을 만들 때는 호출 함수를 알 수 없고 프로그램이 실행될 때 결정된다면 호출할 함수의 주소를 받기 위해 함수 포인터가 필요하다.

- 함수 포인터의 개념

함수명의 의미를 파악하는 것이다. 함수명은 함수 정의가 있는 메모리의 시작 위치이다. 함수명이 주소이므로 포인터에 저장하면 함수를 다양한 방식으로 호출할 수 있다.

```
#include <stdio.h>

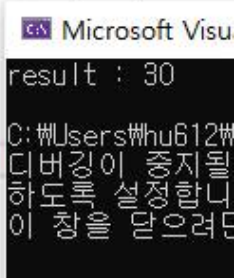
int sum(int, int);

int main(void)
{
    int (*fp)(int, int);
    int res;

    fp = sum;
    res = fp(10, 20);
    printf("result : %d\n", res);

    return 0;
}

int sum(int a, int b)
{
    return (a + b);
}
```



`int (*fp)(int, int);` : 함수의 주소 `sum`을 저장할 함수 포인터를 선언한다. 함수 포인터는 변수명 앞에 별(*)을 붙여 포인터임을 표시한다. 그리고 가리키는 함수의 형태를 반환값과 매개변수로 나누어 적는다.

이때 포인터는 변수명을 번호와 함께 괄호로 묶어야 한다. 괄호가 없으면 주소를 반환하는 함수의 선언이 되기 때문에 주의해야 한다.

- 함수 포인터의 활용

- 예시

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

void func(int (*fp)(int, int));
int sum(int a, int b);
int mul(int a, int b);
int max(int a, int b);

int main(void)
{
    int sel;

    printf("01 두 정수의 합\n");
    printf("02 두 정수의 곱\n");
    printf("03 두 정수 중에서 큰 값 계산\n");
    printf("원하는 연산을 선택하세요 : ");
    scanf("%d", &sel);

    switch (sel)
    {
        case 1: func(sum); break;
        case 2: func(mul); break;
        case 3: func(max); break;
    }
}

void func(int (*fp)(int, int))
{
    int a, b;
    int res;

    printf("두 정수의 값을 입력하세요 : ");
    scanf("%d%d", &a, &b);
    res = fp(a, b);
    printf("결과값은 : %d\n", res);
}

int sum(int a, int b)
{
    return (a + b);
}

int mul(int a, int b)
{
    return (a * b);
}

int max(int a, int b)
{
    if (a > b) return a;
    else return b;
}

```

```

01 두 정수의 합
02 두 정수의 곱
03 두 정수 중에서 큰 값 계산
원하는 연산을 선택하세요 : 2
두 정수의 값을 입력하세요 : 3 7
결과값은 : 21

```

- void 포인터

주소를 가리키는 자료형이 일치하는 포인터에만 대입이 가능하다. 따라서 가리키는 자료형이 다른 주소를 저장하는 경우라면 void 포인터를 사용한다.

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

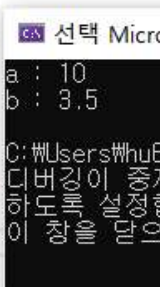
int main(void)
{
    int a = 10;
    double b = 3.5;
    void* vp;

    vp = &a;
    printf("a : %d\n", *(int*)vp);

    vp = &b;
    printf("b : %.1lf\n", *(double*)vp);

    return 0;
}

```



void 포인터를 선언하였다. void 포인터는 가리키는 자료형이 정해지지 않은 포인터이다. 따라서 어떤 주소든 저장할 수 있다. 또한 같은 이유로 간접 참조 연산이나 정수를 더하는 포인터 연산이 불가능하다. 따라서 사용하려면 원하는 형태로 변환하여 사용하면 된다.

- 마무리

- ✓ 함수명의 의미부터 보자면 함수명은 함수 정의가 있는 메모리의 시작 주소이다.
- ✓ 함수 포인터에 함수명을 대입하면 함수처럼 호출할 수 있다.
- ✓ void 포인터에는 임의의 주소를 저장할 수 있다.
- ✓ void 포인터는 간접 참조 연산과 주소에 대한 정수 연산이 불가능하다.

| 구분 | 기능 | 설명 |
|----------|--------|----------------------------|
| 함수 포인터 | 선언 방법 | int (*fp)(int, int); |
| | 함수의 호출 | fp(10,20); |
| | 용도 | 함수명을 대입하여 호출 함수를 결정한다. |
| void 포인터 | 선언 방법 | void *vp; |
| | 의미 | 가리키는 자료형에 대한 정보가 없다. |
| | 용도 | 임의의 주소를 받는 함수의 매개변수에 사용한다. |