2023/05/15

APPDONG
C언어 멘토링

5주차
멘토 : 김민수

Contents

포인터 기초

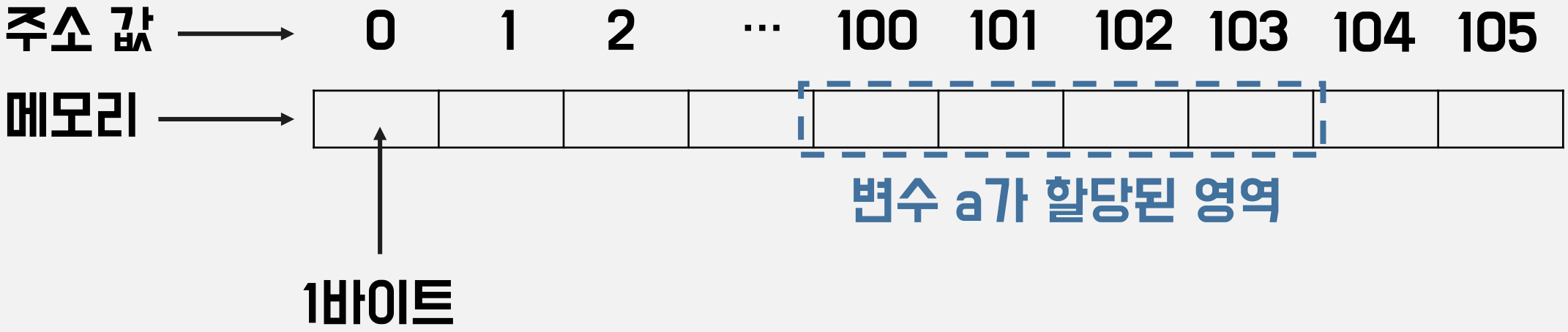
메모리의 주소

데이터를 저장 해놓는 메모리의 위치를 주소 값으로 식별한다

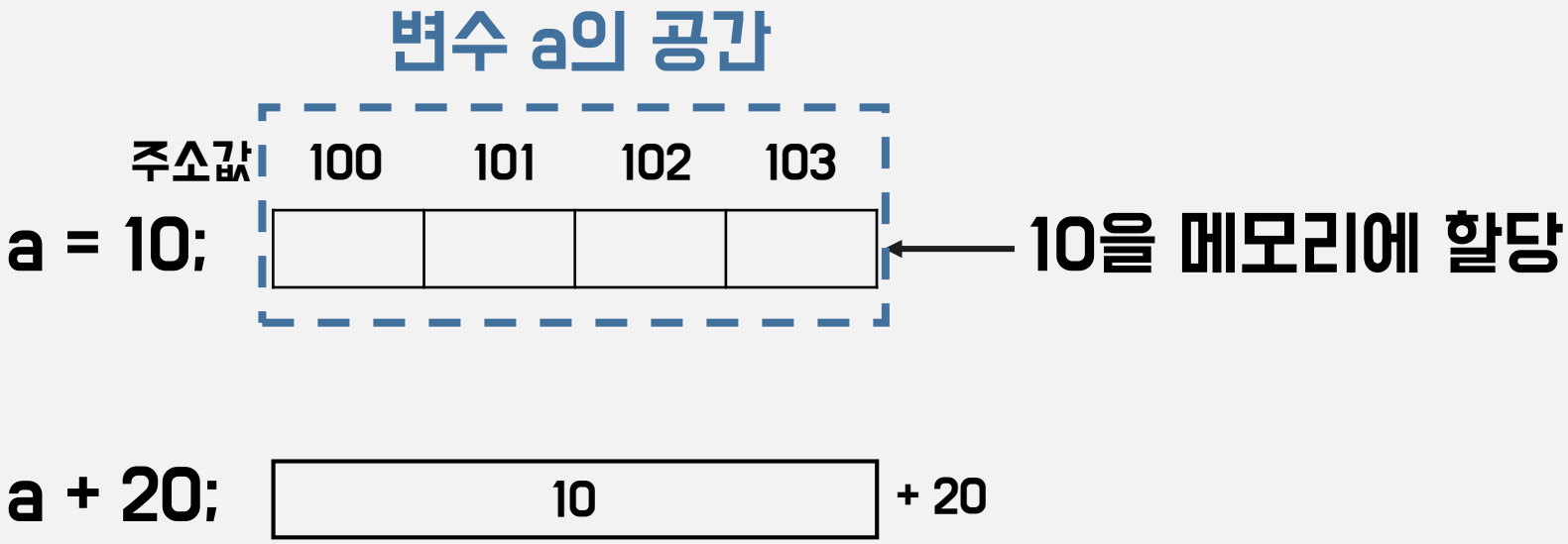
주소 값은 0부터 시작하고 바이트 단위로 1씩 증가하며
2바이트 이상의 크기를 갖는 변수는 여러 개의 주소 값에 걸쳐 할당된다

포인터 기초

int a;



포인터 기초



a = 10; -> 메모리의 100번지에서 103번지까지 4바이트 공간에 10을 저장한다
a + 20; -> 메모리 100번지부터 103번지까지 4바이트에 저장된 값과 20을 더하는 연산을 수행한다

포인터 기초

주소 연산자 &

여기서 말하는 "주소"는 변수가 할당된 메모리 공간의 시작 주소를 의미한다
주소 연산자를 사용해서 주소 값을 알 수 있다

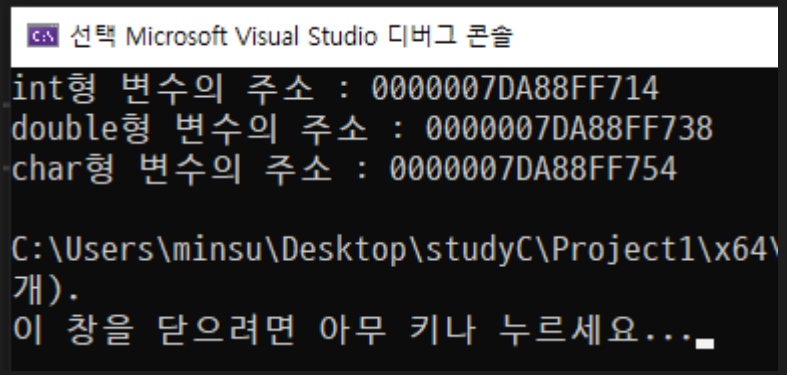
&(피연산자)로 사용해서 피연산자(변수)의 메모리 주소를 반환(return)할 수 있다

주소는 보통 16진수로 표현하기 때문에 주소를 출력할 때는 %p를 사용해서
출력하면 된다

scanf(); 를 사용할 때 변수의 메모리의 주소 값이 인자로 사용된다

포인터 기초

```c
#include <stdio.h>

int main() {
    int a;
    double b;
    char c;

    printf("int형 변수의 주소 : %p₩n", &a);
    printf("double형 변수의 주소 : %p₩n", &b);
    printf("char형 변수의 주소 : %p₩n", &c);

    return 0;
}
```



주소 값을 출력할 때는 %p를 사용한다.

포인터 기초

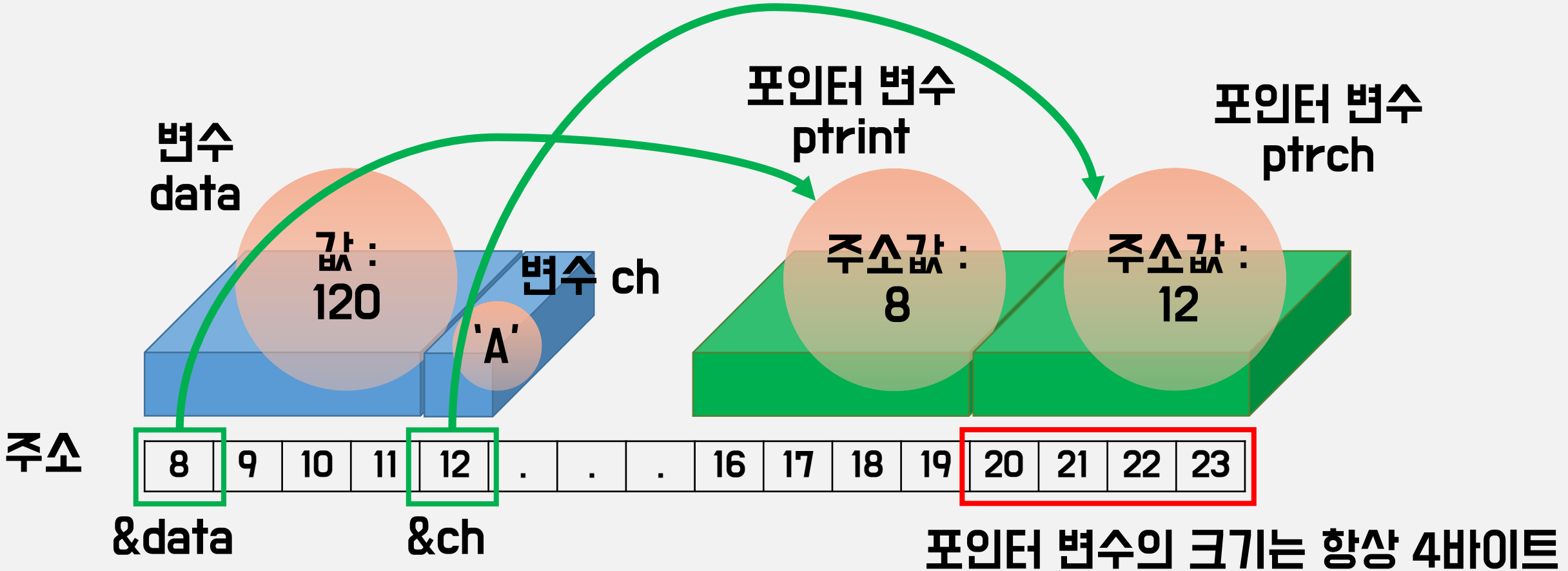
간접 참조 연산자 *

메모리의 주소를 필요할 때마다 계속 주소 연산자로 구하는 것 보다는
한 번 구한 주소를 저장해서 사용하면 편하다

이 때, 포인터를 선언해서 변수의 메모리 주소를 할당하면 된다

포인터 자체는 변수의 메모리 주소 값을 가리키고 있는데
간접 참조 연산자(*)로 변수의 데이터 값에 접근할 수 있다

```c
int data = 120;
char ch = 'A';
int* ptrint = &data;
char* ptrch = &ch;

printf("간접참조 출력 : %d %c", *ptrint, *ptrch);
```

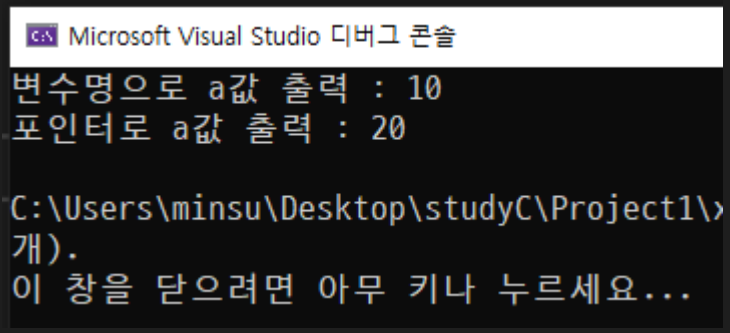


포인터 기초

```c
#include <stdio.h>

int main() {
    int a = 10;
    int* pa;

    pa = &a;

    printf("변수명으로 a값 출력 : %d₩n", a);
    *pa = 20;

    printf("포인터로 a값 출력 : %d₩n", *pa);

    return 0;
}
```

포인터 변수에 a의 주소 할당

간접참조로 a값 변경



포인터 기초

int a;
int *pa;

pa = &a;

값을 입력할 때
scanf("%d", &a); 대신에 scanf("%d", pa); 사용 가능

sizeof 연산자로 포인터 크기를 확인하면 가리키는 자료형과 관계없이
크기가 같다 -> 환경에 따라 다른데 보통 4바이트 또는 8바이트

포인터 변수의 자료형과 가리킬려는 변수의 자료형은 일치해야함
int a; -> int *pa; double b; -> double *pb;

포인터 기초

포인터 상수 표현 (const) 1

```
int a, b;
const int *pa = &a;

*pa = 20;     // 에러

pa = &b;      // 가능
```

pa가 가리키는 변수 a는 pa를 간접참조하여 변경할 수 없다는 것을 의미

단, 다른 변수를 다시 가리키는 것은 가능

포인터 기초

포인터 상수 표현 (const) 2

```
int a, b;
int *const pa = &a;

pa = &b;      // 에러

*pa = 20;     // 가능
```

pa값을 변경할 수 없다는 것을 의미
(다른 변수를 가리킬 수 없음)

단, 가리키는 변수의 값을 간접 참조하여
변경할 수는 있음

포인터 기초

포인터의 덧셈, 뺄셈

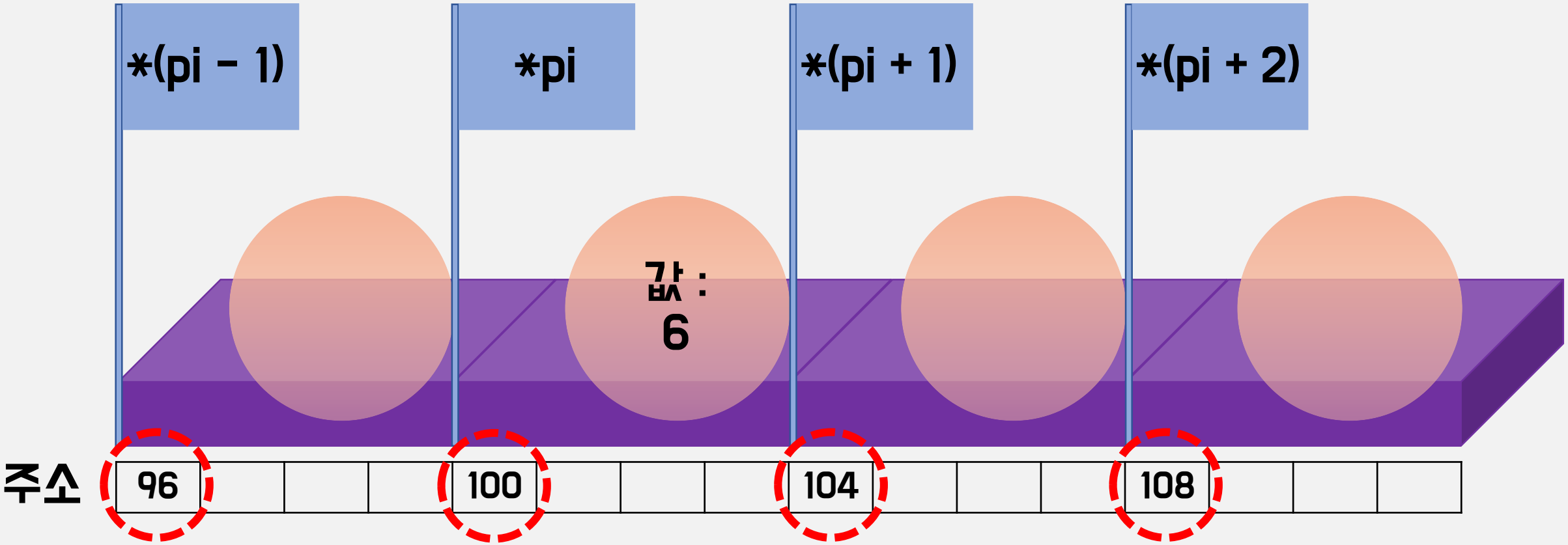
-> 포인터가 가리키는 변수 크기에 비례한 연산
즉, 포인터에 저장된 **주소값**의 연산이다.

```
int data = 6;
int* pi = &data;
```

만약 data의 주소값(&data)이 100일 때,
pi + 1 의 결과는 101이 아닌 104가 된다. (int형이 4바이트이기 때문)

포인터 기초

```
int data = 6;
int* pi = &data;
```

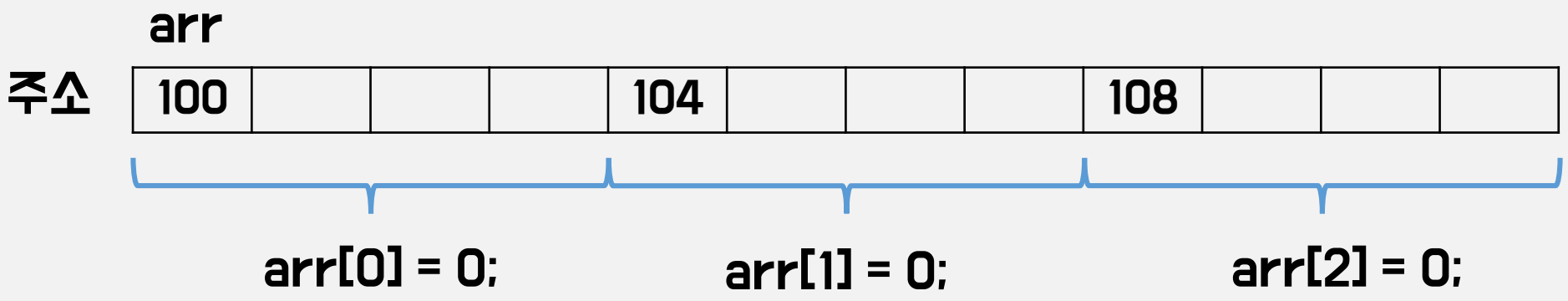


int 형 변수 -> 4바이트

배열과 포인터의 관계

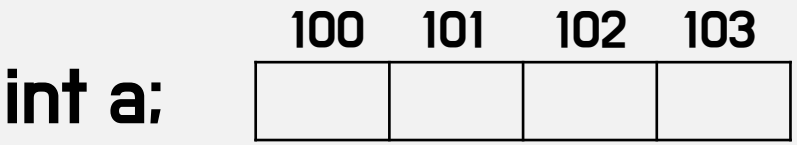
배열은 자료형이 같은 변수를 메모리에 연속으로 할당한다
따라서 각 배열 요소는 일정한 간격으로 주소를 갖게 된다

int arr[3] = {0, };
-> 컴파일러는 배열명(arr)을 첫 번째 배열 요소의 주소로 변경한다



배열과 포인터의 관계

주소 + 정수 -> 주소 + (정수 * 주소를 구한 변수의 크기)

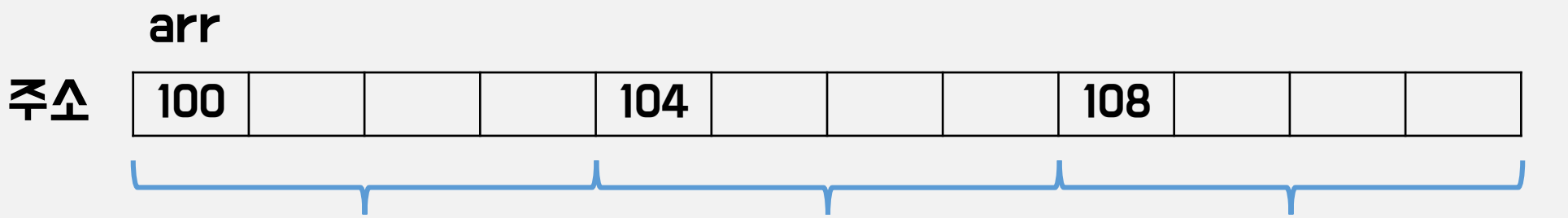


&a : 변수 a의 주소 값 (100)

&a + 1 -> 100 + (1 * sizeof(int)) -> 104

배열은 위 규칙을 잘 이용하여 각 요소에
쉽게 접근할 수 있다

배열과 포인터의 관계



arr[0] = 10;
-> *arr = 10;

arr[1] = 20;
-> *(arr + 1) = 20;

arr[2] = 30;
-> *(arr + 2) = 30;

간접참조연산자 *로 각 배열 원소에 접근해서 값을 변경할 수 있다.

*arr = 10; -> arr[0]에 접근
*(arr + 1) = 20; -> arr[1]에 접근
*(arr + 2) = 30; -> arr[2]에 접근

배열과 포인터의 관계

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

int main() {
    int arr[3];

    arr[0] = 10;
    arr[1] = arr[0] + 10;

    printf("arr[2] 값 입력 >> ");
    scanf("%d", &arr[2]);

    for (int i = 0; i < 3; i++) {
        printf("%3d", arr[i]);
    }

    return 0;
}
```

위 코드를 포인터를 사용한 코드로 다시 구현해보자.

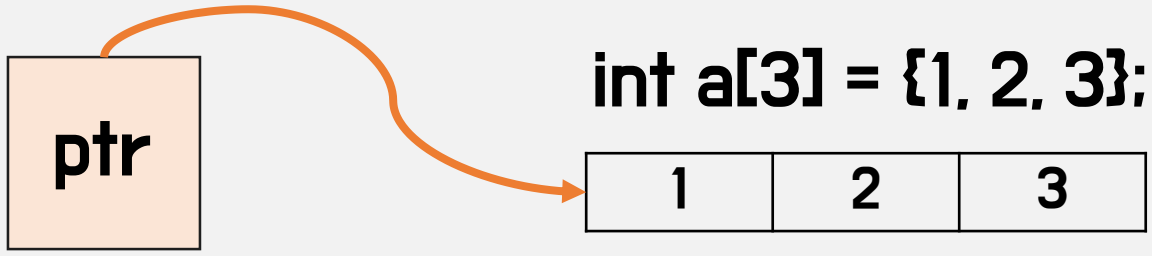
배열과 포인터의 관계

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

int main() {
    int arr[3];

    *arr = 10;
    *(arr + 1) = *arr + 10;

    printf("arr[2] 값 입력 >> ");
    scanf("%d", arr + 2);

    for (int i = 0; i < 3; i++) {
        printf("%3d", *(arr + i));
    }

    return 0;
}
```

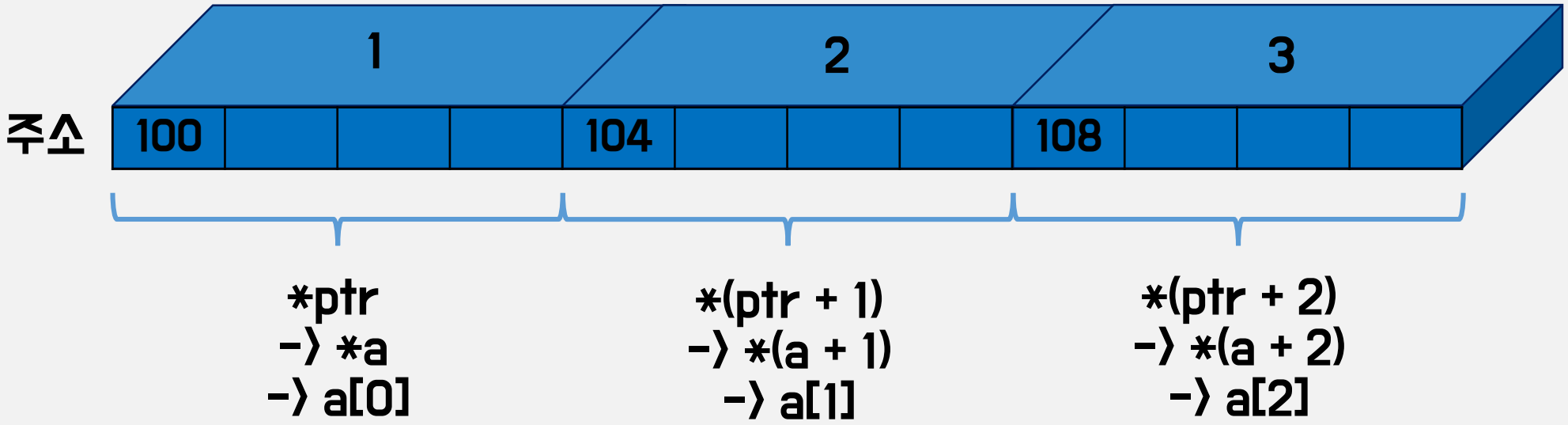
배열과 포인터의 관계

배열명 대신에 포인터 변수를 선언해서 배열 주소를 참조한 후,
배열 원소에 접근할 수 있다 (이 때의 포인터 변수를 "배열 포인터"라고 함.)

int *ptr = a;
-> 배열명을 가리킨다 (첫 번째 주소를 가리킨다)



배열과 포인터의 관계



Q. 배열 이름을 사용해도 되는데 왜 포인터 변수를 또 선언해서 사용을 하는가?

A. 배열 이름은 "포인터 상수"이다. int a[3]; 에서 a 값을 변경할 수 없기 때문에 a++, ++a 같은 a 값을 직접 바꾸는 증감 연산이 불가능하다.
이런 증감 연산을 사용하려면 포인터 변수를 선언해야한다.

배열과 포인터의 관계

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

int main() {
    int arr[3];
    int* ptr = arr;

    *ptr = 10;
    *(ptr + 1) = 20;

    printf("arr[2] 값 입력 >> ");
    scanf("%d", ptr + 2);

    for (int i = 0; i < 3; i++) {
        printf("%3d", *ptr++);
    }

    return 0;
}
```

함수

함수를 작성할 때 고려해야할 사항

1. 정의 : 어떤 기능을 구현할 것인지 (여러 번 반복되는 작업인 경우, 복잡한 코드 구조인 경우, 특수한 기능인 경우 등등..)

2. 호출 : 어떤 인자를 통해 함수를 호출할 것인지 (int 변수, char 변수, 포인터 변수 등등..)

3. 선언 : 헤더 파일과 main() 함수 사이에 함수 이름, 인자를 명시적으로 표현

함수

함수 구조

반환 타입 함수이름 (함수인자)
{
	기능 작성;
	return 반환 값; (반환 타입에 따라 return 값 설정)
}

함수

```
1   #define _CRT_SECURE_NO_WARNINGS
2
3   #include <stdio.h>
4
5   int sum_array(int ary[], int size);
6
7   int main() {
8       int arr[5] = { 0, };
9
10      for (int i = 0; i < 5; i++) {
11          scanf("%d", &arr[i]);
12      }
```

함수 선언

함수

```
13
14      printf("평균 점수 : %.4lf₩n", (double)sum_array(arr, 5) / 5);
15
16      return 0;
17  }
18
19  int sum_array(int ary[], int size) {
20      int sum = 0;
21
22      for (int i = 0; i < size; i++) {
23          sum += ary[i];
24      }
25
26      return sum;
27  }
```

함수 호출

함수 정의

함수

* 여러가지 함수 종류 *

1. 매개변수(함수 인자)가 없는 함수
2. 반환 값(리턴 값)이 없는 함수 (=void 타입 함수)
3. 매개변수와 반환 값이 모두 없는 함수
4. 재귀 호출 함수

함수

1. 매개변수가 없는 함수

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

int get_num();

int main() {
    int num;

    do {
        printf("출력 : %d₩n----------₩n", num = get_num());
    } while (num);

    return 0;
}
```

```c

int get_num() {
    int n;

    printf("입력 : ");
    scanf("%d", &n);

    return n;
}
```

함수

2. 반환 값이 없는 함수

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

void print_char(char ch, int count);

int main() {
    char c;
    int n;

    printf("입력한 문자 여러 번 출력하기₩n");
    printf("문자 입력 >> ");
    scanf("%c", &c);
    printf("숫자 입력 >> ");
    scanf("%d", &n);

    print_char(c, n);

    return 0;
}
```

```c

void print_char(char ch, int count) {
    for (int i = 0; i < count; i++) {
        printf("%c", ch);
    }
}
```

함수

3. 매개변수와 반환 값이 모두 없는 함수

```c
#include <stdio.h>

void print_line();

int main() {
    print_line();
    printf("학번     이름     전공          학점₩n");
    print_line();
    printf("2000     김민수   컴퓨터학부    3.0₩n");
    printf("2011     홍길동   전자공학부    4.3₩n");
    print_line();

    return 0;
}
```

함수

3. 매개변수와 반환 값이 모두 없는 함수

```
15
16  void print_line() {
17      for (int i = 0; i < 37; i++) {
18          printf("-");
19      }
20
21      printf("₩n");
22  }
```

함수

4. 재귀호출 함수

재귀 호출(recursive call)이란?

함수 내부에서 함수가 자기 자신을 또 다시 호출하는 행위.
자기가 자신을 계속 호출하기 때문에 함수 내에서 재귀 호출을 중단하도록 조건 명령문을 반드시 작성해야한다.

탈출 조건이 없는 경우 프로그램이 사용할 수 있는 메모리를 모두 사용할 때까지 무한 반복된다.

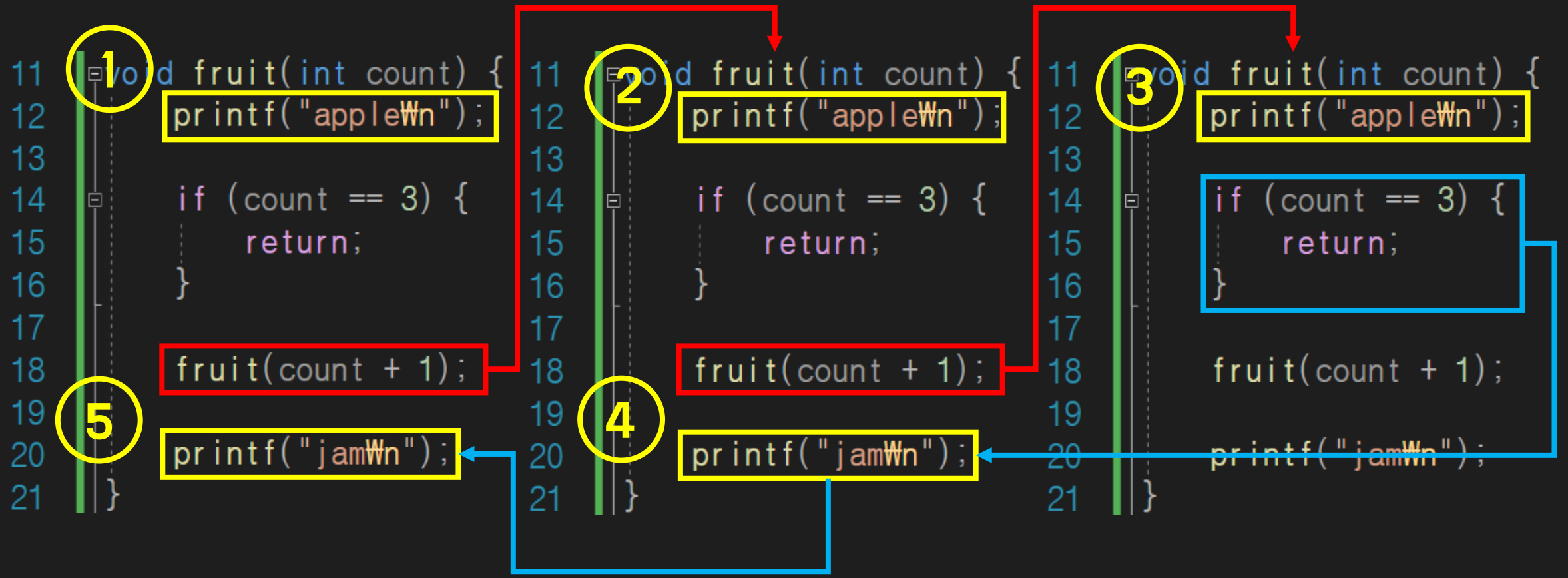
함수

4. 재귀호출 함수

```c
#include <stdio.h>

void fruit(int count);

int main() {
    fruit(1);

    return 0;
}
```

```c
void fruit(int count) {
    printf("apple₩n");

    if (count == 3) {
        return;
    }

    fruit(count + 1);

    printf("jam₩n");
}
```

실행 결과는??


1
2
3
4
5
11 void fruit(int count) {
12 printf("apple₩n");
13
14 if (count == 3) {
15 return;
16 }
17
18 fruit(count + 1);
19
20 printf("jam₩n");
21 }

함수와 포인터의 활용

함수의 인자 전달 방식

값에 의한 호출 (Call by Value)
-> 함수 호출 시 실인자 값이 형식인자에 복사되어 저장된다

참조에 의한 호출 (Call by Reference)
-> 함수로 전달된 실인자의 주소를 이용하여 변수를 참조한다

함수와 포인터의 활용

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

void increase_CallByValue(int origin, int increment);
void increase_CallByReference(int* origin, int increment);

int main() {
    int amount = 10;

    increase_CallByValue(amount, 20);
    printf("Call by Value : %d\n", amount);

    increase_CallByReference(&amount, 20);
    printf("Call by Reference : %d", amount);

    return 0;
}
```

함수와 포인터의 활용

```
19
20  void increase_CallByValue(int origin, int increment) {
21      origin += increment;
22  }
23
24  void increase_CallByReference(int* origin, int increment) {
25      *origin += increment;
26  }
```

Call by Value의 경우 함수 내부의 변수 값은 함수 외부에 영향을 끼치지 못한다

반면에 Call by Reference의 경우 해당 변수의 주소를 참조하기 때문에 값이 변경되면 해당 변수의 값도 변경된다

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

void swap1(int x, int y);
void swap2(int* x, int* y);

int main() {
    int a = 100, b = 200;

    printf("a= %d, b= %d\n\n", a, b);

    // Call by Value
    swap1(a, b);
    printf("After swap1() a= %d, b= %d\n\n", a, b);

    // Call by Reference
    swap2(&a, &b);
    printf("After swap2() a= %d, b= %d\n\n", a, b);

    return 0;
}
```

```
23
24  void swap1(int x, int y) {
25      int temp;
26
27      temp = x;
28      x = y;
29      y = temp;
30
31      printf("swap1() a= %d, b= %d\n", x, y);
32  }
33
34  void swap2(int* x, int* y) {
35      int temp;
36
37      temp = *x;
38      *x = *y;
39      *y = temp;
40
41      printf("swap2() a= %d, b= %d\n", *x, *y);
42  }
```

함수 내에서만 x값, y값이 바뀌지
실제 a값과 b값에 영향을 줄 수 없다

a와 b의 주소를 참조하는 값이 변경되면
a값과 b값도 영향을 받는다

함수와 포인터의 활용

배열을 함수의 인자로 전달하는 방법

```
double sum(double arr[], int n);
...
double data[] = {2.1, 3.3, 5.5};
sum(data, 3);
```

-> 배열이름으로 배열인자를 명시한다

배열을 함수 인자로 전달할 때 배열 크기도 함께 전달하는 것이 효율적이다
-> 전달하지 않으면 정해진 상수를 함수 내부에서 사용해야하는 데
배열 크기가 변경될 경우 코드를 수정해야하므로 비효율적이다

함수와 포인터의 활용

sum(data, size); -> 배열 이름으로 함수 인자를 전달한다
+ 배열 크기도 함께 전달한다

배열을 인자로 가지는 함수 선언 방법

double sum(double data[], int n); 또는
double sum(double *data, int n); 으로 함수를 선언하면 된다

함수와 포인터의 활용

배열 크기 구하는 방법

sizeof(배열명) / sizeof(배열의 자료형 또는 배열원소)

ex)

```
int arr[] = {1.2, 3.3, 5.5, 7.7, … };
int arr_size1 = sizeof(arr) / sizeof(int);
int arr_size2 = sizeof(arr) / sizeof(arr[0]);
```

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

double sum(double ary[], int n);

int main() {
    double data[] = { 2.3, 3.4, 4.5, 5.6, 7.8 };
    int size = 0;

    size = sizeof(data) / sizeof(data[0]);

    printf("Total= %lf\n", sum(data, size));

    return 0;
}
```

배열 크기를 계산해서

배열 이름과 배열 크기를 함수 인자로 전달

```
17
18  double sum(double ary[], int n) {
19      int i = 0;
20      double total = 0.0;
21
22      for (i = 0; i < n; i++) {
23          total += ary[i];
24      }
25
26      return total;
27  }
```

total += *(ary + i); 로 작성 가능

함수와 포인터의 활용

배열을 전달할 때, 함수에서는 인자를 배열 대신 포인터로 받아도 된다

ex)

```
int sum(int arr[], int size);    <->    int sum(int *arr, int size);
...                                     ...
sum += arr[i];                          sum += *(arr + i);
or                                      or
sum += *arr++;                          sum += *(arr++);
```

함수와 포인터의 활용

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

void input_arr(double* pa, int size);
double find_max(double* pa, int size);

int main() {
    double arr[5];
    double max;

    int size = sizeof(arr) / sizeof(arr[0]);

    input_arr(arr, size);
    max = find_max(arr, size);

    printf("배열의 최댓값 : %lf", max);

    return 0;
}
```

함수와 포인터의 활용

```
21
22  void input_arr(double* pa, int size) {
23      int i;
24
25      printf("%d개의 실수값 입력 : ", size);
26      for (i = 0; i < size; i++) {
27          scanf("%lf", pa + i);
28      }
29  }
30
```

```
31  double find_max(double arr[], int size) {
32      double max;
33      int i;
34
35      max = arr[0];
36      for (i = 1; i < size; i++) {
37          if (arr[i] > max) max = arr[i];
38      }
39
40      return max;
41  }
```

함수와 포인터의 활용

포인터를 반환하는 함수

int* add(int *, int, int);
-> 반환값 : 포인터(int*)
-> 호출 예시 : *add(&sum, m, n);

반환되는 값(주소값)을 다시 간접 참조하여 저장된 데이터 값 접근 가능

단, 반환값은 함수 내부의 지역변수의 주소값이면 안됨.
함수가 종료되는 시점에 메모리에서 제거 되는 변수이기 때문에 문제가 발생

```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>

int* add(int*, int, int);

int main() {
    int m = 0, n = 0, sum = 0;

    scanf("%d %d", &m, &n);

    printf("두 정수의 합 : %d\n", *add(&sum, m, n));

    return 0;
}

int* add(int* psum, int a, int b) {
    *psum = a + b;

    return psum;
}
```

함수 반환 값을 참조해서
정수의 합 출력

포인터를 반환(주소 값을 반환)

문제 해결

백준 단계별로 풀어보기 6단계 (https://www.acmicpc.net/step)

"심화" 2번(3003), 3번(2444), 7번(4344)

단계	제목	설명
1	입출력과 사칙연산	입력, 출력과 사칙연산을 연습해 봅시다. Hello World!
2	조건문	if 등의 조건문을 사용해 봅시다.
3	반복문	for, while 등의 반복문을 사용해 봅시다.
4	1차원 배열	배열을 사용해 봅시다.
5	문자열	문자열을 다루는 문제들을 해결해 봅시다.
6	심화 1	지금까지의 프로그래밍 문법으로 더 어려운 문제들을 풀어봅시다.
7	2차원 배열	배열 안에 배열이 있다면 어떨까요? 2차원 배열을 만들어 봅시다.

해결 못한 문제는 다음 멘토링 시간 전까지 해오기

끝