프로그래밍 기초

14장. 함수와 포인터 활용 Part 1

학습목표

- **함수의 인자전달 방식을 이해하고 설명할 수 있다.**
  - 값에 의한 호출과 참조에 의한 호출 방식
  - 함수에서 인자와 반환값으로 배열을 주고 받는 활용
  - 가변인자의 필요성과 사용 방법

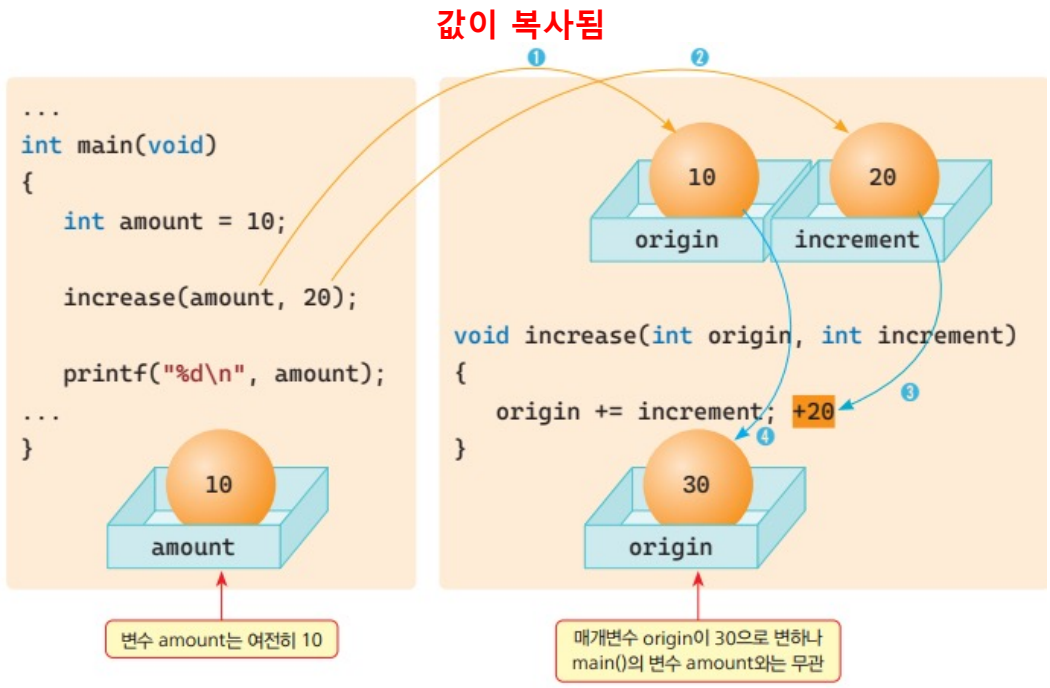
- **함수에서 인자로 포인터의 전달과 반환으로 포인터형의 사용을 이해하고 설명할 수 있다.**
  - 매개변수 전달과 반환으로 포인터 사용
  - 포인터 인자전달 시 키워드 const의 이용
  - 함수에서 구조체 전달과 반환

- **함수 포인터를 이해하고 설명할 수 있다.**
  - 함수 포인터의 필요성과 사용
  - 함수 포인터 배열의 사용
  - void 포인터의 필요성과 사용

함수에서 값의 전달: Call by value

- **값에 의한 호출(call by value) 방식**
  - 함수 호출 시 **인자(argument)의 값**이 **매개변수(형식인자, parameter)에 복사**
  - 함수 외부의 변수(amount)를 함수 내부에서 수정할 수 없음

- **함수 increase(int origin, int increment) 함수 호출 시**
  - origin += increment; 를 수행하는 함수
    - ➢ 변수 amount의 값 10이 매개변수 origin에 복사
    - ➢ 상수 20이 매개변수 increment에 복사
  - 함수 increase() 내부 실행
    - ➢ 매개변수인 origin 값이 30으로 증가
  - 변수 amount와 매개변수 origin은 아무 관련성이 없음
    - ➢ origin은 증가해도 amount의 값은 변하지 않음
    - ➢ 변수 origin은 함수 increase() 내부에서만 사용



함수에서 주소 전달: Call by address

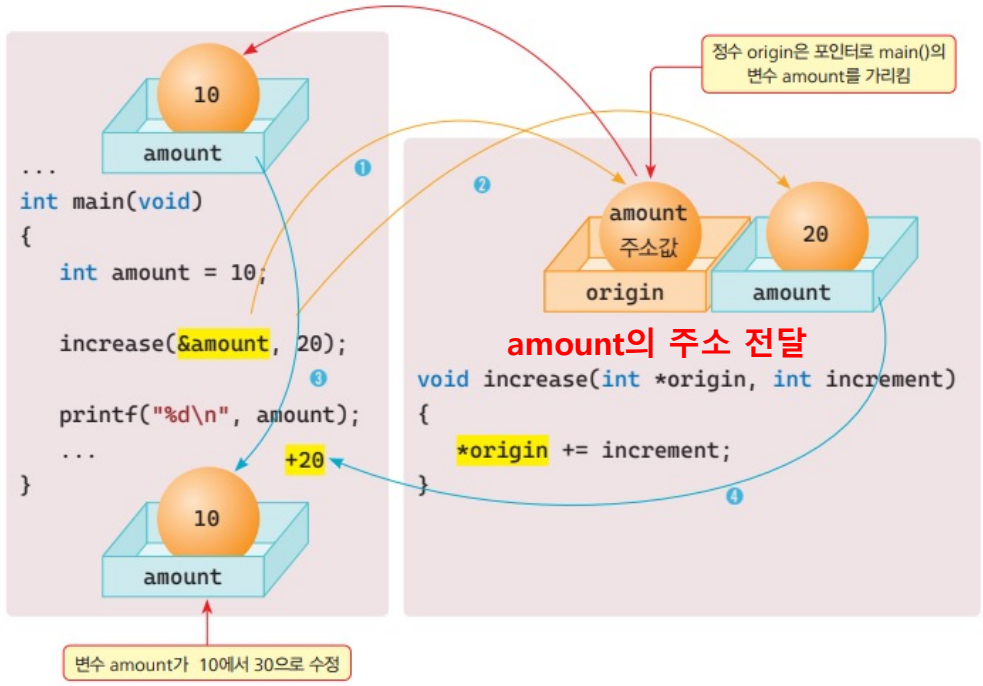
- **주소에 의한 호출(call by address)**
  - 포인터를 매개변수로 사용
  - 함수로 전달된 실인자의 주소를 이용하여 그 변수를 참조 가능

- **increase(int *origin, int increment)**
  - 함수 호출 시 첫 번째 인자가 &amount
    - ➢ 변수 amount의 주소를 매개변수인 origin에 복사

```
int amount = 10;
increase(&amount, 20);
```

  - 함수 increase() 내부 실행
    - ➢ *origin은 변수 amount 자체를 의미
    - ➢ *origin을 증가시키면 amount의 값도 증가

  - main() 내부에서 amount의 값이 30으로 증가



Call by Value 값 전달 과정

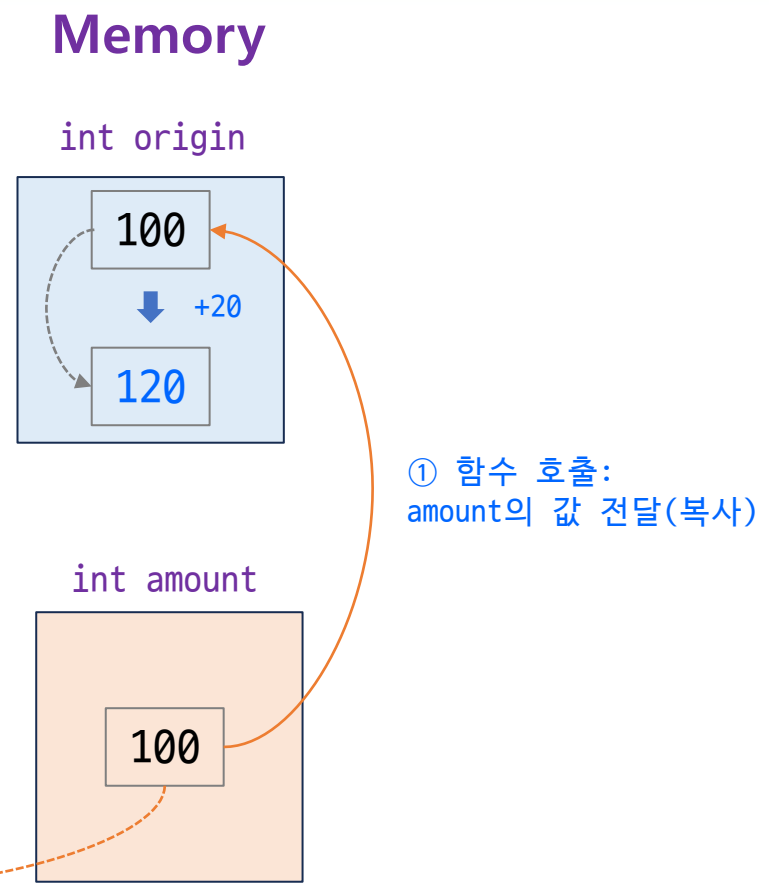
```
void increase(int origin, int increment)
{
    origin += increment;
}
```

② origin의 값 변경

```
int main(void)
{
    int amount = 100;
    increase(amount, 20);
    printf("Call by value: %d\n", amount);
    return 0;
}
```

① 함수 호출: amount의 값 전달(복사)

③ amount의 값을 읽고 출력



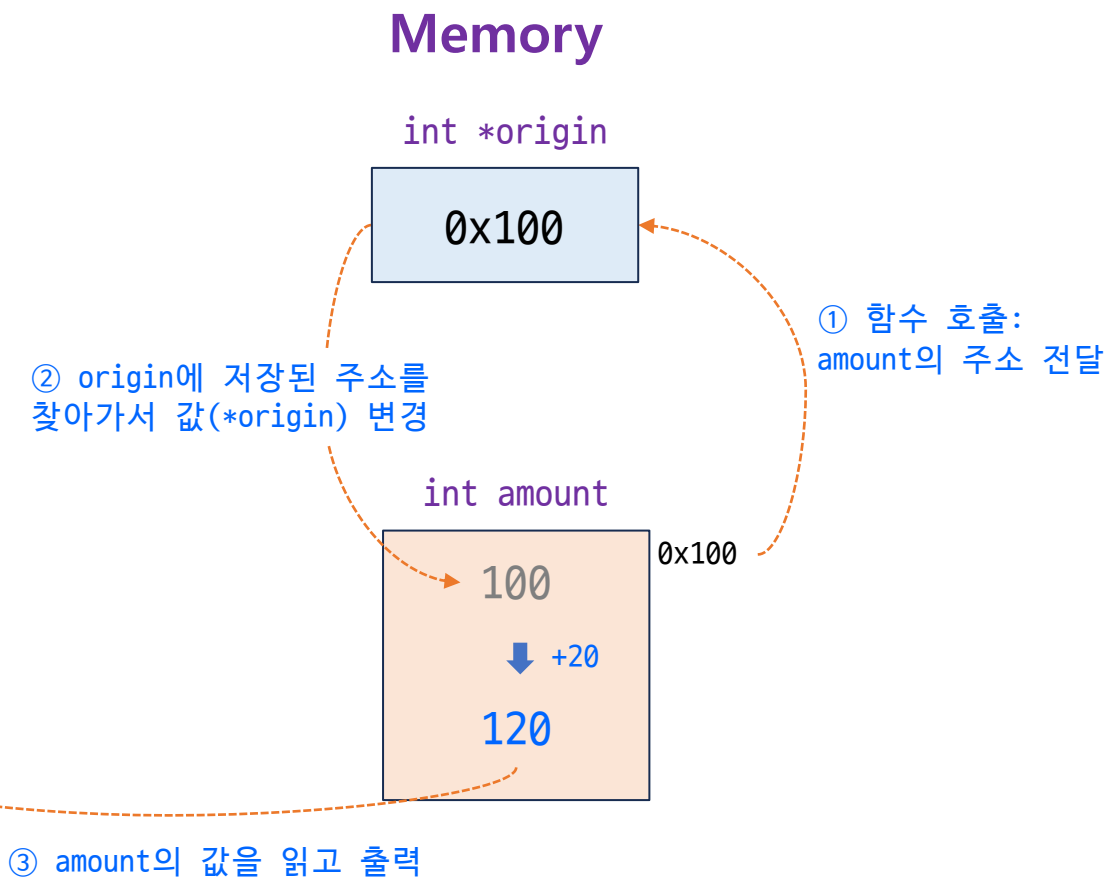
Call by Address 의 주소 전달 및 값 변경 과정

```
void incbyaddress(int *origin, int increment)
{
    *origin += increment;
}
```

주소의 값 → ②

```
int main(void)
{
    int amount = 100;

    incbyaddress(&amount, 20);

    printf("Call by address: %d\n", amount);

    return 0;
}
```

① ③



함수의 값과 주소의 전달 방식 (실습)

<01callby.c>

```c
#include <stdio.h>

void increase(int origin, int increment);
void incbyaddress(int* origin, int increment);

int main(void)
{
    int amount = 10;
    increase(amount, 20); //amount의 값을 전달
    printf("Call by value: %d\n", amount);

    amount = 100;
    incbyaddress(&amount, 20); //amount의 주소를 전달
    printf("Call by address: %d\n", amount);

    return 0;
}
```

```c
void increase(int origin, int increment)
{
    origin += increment;
}

void incbyaddress(int* origin, int increment)
{
    //*origin은 origin이 가리키는 주소의 값
    *origin += increment;
}
```

실행 결과

```
Call by value: 10
Call by address: 120
```

예제: 값에 의한 호출과 주소에 의한 호출 비교 (실습)

<swap.c>

```c
#include <stdio.h>
void swap1(int x, int y);
void swap2(int *x, int *y);

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

    // 값에 의한 호출
    printf("Before swap1() a= %d, b= %d\n", a, b);
    swap1(a, b);
    printf("After swap1() a= %d, b= %d\n", a, b);
    printf("\n");

    // 참조에 의한 호출
    printf("Before swap2() a= %d, b= %d\n", a, b);
    swap2(&a, &b);
    printf("After swap2() a= %d, b= %d\n", a, b);
    return 0;
}
```

```c
// Call by value
void swap1(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
    printf("swap1 function x= %d, y= %d\n", x, y);
}

// Call by address
void swap2(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    printf("swap2 function x= %d, y= %d\n", *x, *y);
}
```

실행 결과

```
Before swap1() a= 100, b= 200
swap1 function x= 200, y= 100
After swap1()  a= 100, b= 200

Before swap2() a= 100, b= 200
swap2 function x= 200, y= 100
After swap2()  a= 200, b= 100
```

함수로 배열 전달: 배열 이름으로 전달

- **함수의 매개변수로 배열을 전달하는 것**
  - 배열의 시작 주소 전달: 배열의 첫 원소의 주소(&data[0])를 매개변수로 전달하는 것과 동일

```
double data[] = {2.3, 3.4, 4.5, 5.6, 6.7 };
sum(data, 5);
```

```
double data[] = {2.3, 3.4, 4.5, 5.6, 6.7 };
sum(&data[0], 5);
```

- **배열을 매개변수로 하는 함수 double sum(double ary[], int n)을 구현**
  - 함수 내부에서는 매개변수로 전달된 배열의 배열 크기를 알 수 없음
    - ➢ double ary[5]보다는 double ary[]라고 기술하는 것을 권장
    - ➢ 매개변수를 double ary[]처럼 기술해도 double *ary 처럼 포인터 변수로 인식
    - ➢ 배열 크기를 두 번째 인자로 사용

```
double data[] = {2.3, 3.4, 4.5, 5.6, 6.7 };

sum(data, 5);
```

배열 이름만 전달

double *ary와 동일

```
double sum(double ary[], int n)
{
    double total = 0.0;
    for (int i = 0; i < n; i++)
        total += ary[i];
    return total;
}
```

함수로 배열 전달: 배열 이름으로 전달

- **배열을 매개변수로 하는 함수 sum(double ary[], int n)을 구현**
  - 함수 내부에서는 매개변수로 전달된 배열의 크기를 알 수 없음
    - 배열의 주소만 전달(double *ary로 인식)
  - 두 번째 파라미터에 배열의 크기를 전달

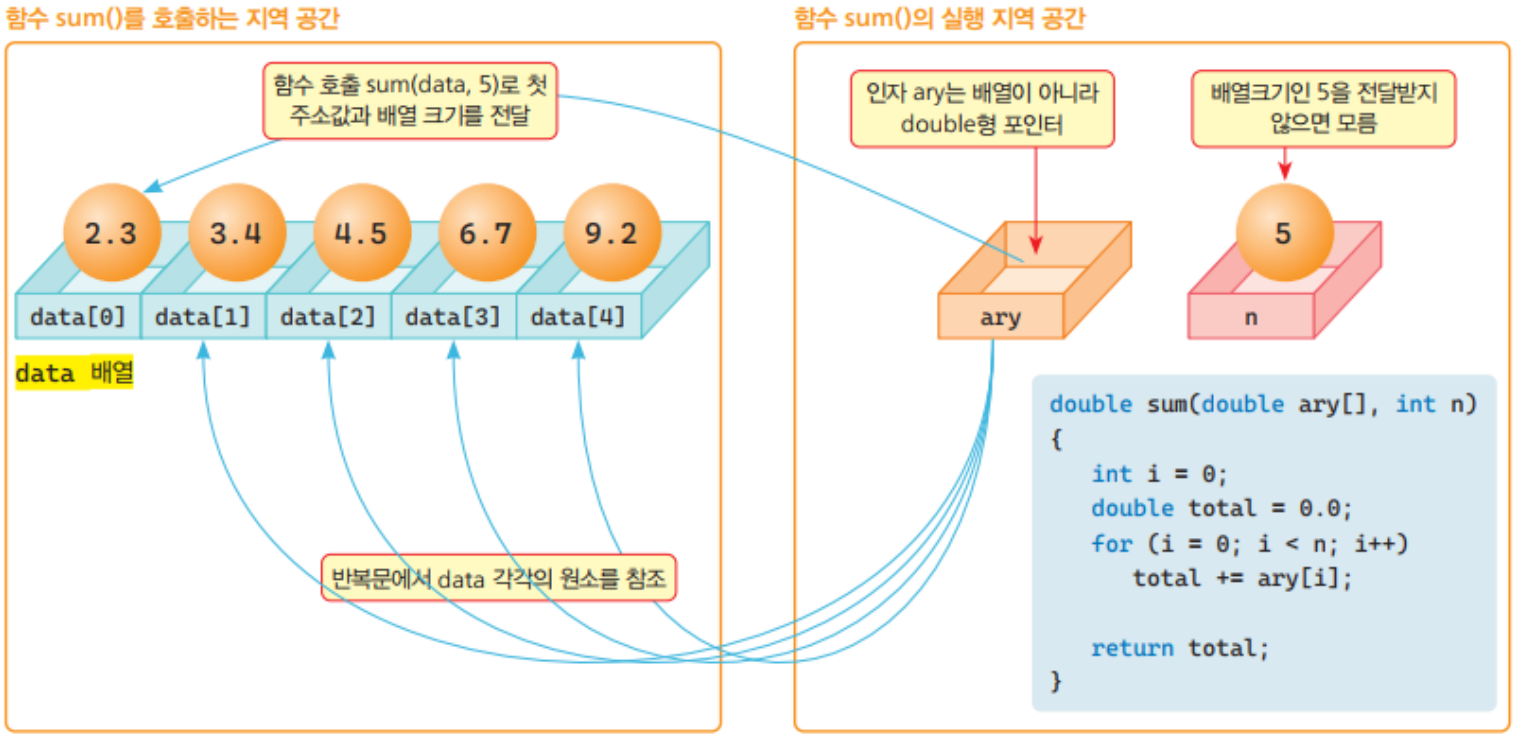
배열의 크기

```c
#include <stdio.h>

double sum(double ary[], int n)
{
    double total = 0.0;

    for (int i = 0; i < n; i++)
        total += ary[i];

    return total;
}
```

```c
int main()
{
    double result = 0.0;
    double data[] = {2.3, 3.4, 4.5, 5.6, 6.7};

    result = sum(data, 5);
    //result = sum(&data[0], 5); // 배열[0]의 주소

    printf("sum: %.2f\n", result);
    return 0;
}
```

배열 크기를 함수의 매개변수로 사용

- **만일 배열 크기를 함수의 매개변수(인자)로 사용하지 않는다면**
  - 정해진 배열 크기 상수를 함수 정의 내부에서 사용해야 함: 하드 코딩 방식
    - ➢ 이런 방법은 배열 크기가 변하면 소스를 수정해야 하므로 비효율적
  - 배열 크기에 관계없이 배열 원소의 합을 구하는 함수를 만들려면
    - ➢ 배열 크기도 하나의 매개변수로 사용



다양한 배열 원소 참조 방법

- **1차원 배열 point에서**
  - 간접연산자를 사용한 배열 원소의 접근 방법: `*(point + i)`
    - 배열의 합을 구하려면 `sum += *(point + i);` 문장을 반복
  - 문장 `int *address = point;`
    - 배열 point를 가리키는 포인터 변수 address를 선언하여 point를 저장
    - 문장 `sum += *(address++)`으로도 배열의 합 가능
  - 배열 이름 point는 주소 상수임: `sum += *(point++)`는 사용 불가능
    - **증가 연산식 point++의 피연산자로 상수인 point를 사용할 수 없기 때문**

```
int i, sum = 0;
int point[] = {95, 88, 76, 54, 85, 33, 65, 78, 99, 82};
int *address = point;
int aryLength = sizeof (point) / sizeof (int);
```

가능

```
for (i=0; i<aryLength; i++)
  sum += *(point+i);
```

가능

```
for (i=0; i<aryLength; i++)
   sum += *(address++);
```

오류

```
for (i=0; i<aryLength; i++)
  sum += *(point++);
```

형식 매개변수 int ary[]와 int *ary

- **함수 선언에 배열을 인자로 기술하는 다양한 방법**
  - 매개변수 `int ary[]`로 기술하는 것은 `int *ary`로도 대체 가능

같은 의미로 사용됨

```
int summary(int ary[], int size)
{
        . . .
}
```

```
int summary(int *ary[], int size)
{
        . . .
}
```

```
for (int i=0; i<size; i++)
{
    sum += ary[i];
}
```

권장 방법

```
for (int i=0; i<size; i++)
{
    sum += *(ary + i);
}
```

권장 방법

```
for (int i=0; i<size; i++)
{
    sum += *ary++;
}
```

```
for (int i=0; i<size; i++)
{
    sum += *(ary++);
}
```

함수에서 배열 인자의 사용 (실습)

<02aryparam.c>

```c
#include <stdio.h>

int summary(int *, int);
// int summary(int ary[], int size)

int main(void)
{
    int point[] = {95, 88, 76, 54, 85, 33, 65, 78, 99, 82};
    int aryLength = sizeof(point) / sizeof(int); // 배열크기 구하기

    int *address = point;
    int sum = 0;
    for (int i = 0; i < aryLength; i++)
        sum += *(point + i); //*(address++), *(address + i)도 가능
        // sum += *(point++); // 오류발생
    printf("main()에서 구한 합은 %d\n", sum);

    // 함수 호출로 합 구하기, 첫 인자 &point[0] 또는 address로도 가능
    printf("함수 summary() 호출로 구한 합은 %d\n",
                            summary(point, aryLength));

    return 0;
}
```

배열의 이름과
배열 크기 전달

```c
// int summary(int ary[], int size) 가능
int summary(int *ary, int size)
{
    int sum = 0;
    for (int i = 0; i < size; i++)
    {
        sum += *(ary + i); // 권장
        // sum += ary[i]; // 권장
        // sum += *ary++; //*(ary++)와 동일
    }

    return sum;
}
```

실행 결과

```
main()에서 구한 합은 755
함수 summary() 호출로 구한 합은 755
```

다차원 배열을 함수 매개변수로 전달하는 방법

- **이차원 배열을 함수 매개변수로 이용하는 방법**
  - 다차원 배열을 인자로 이용하는 경우,
    - ➢ 첫 번째 대괄호 내부의 크기는 생략 가능
    - ➢ 다른 모든 크기는 반드시 기술

- **다차원 배열 전달 함수 원형 및 정의**

```
//2차원 배열값을 모두 더하는 함수 원형
double sum(double data[][3], int, int);

// 2차원 배열값을 모두 출력하는 함수 원형
void printarray(double data[][3], int, int);
```

배열로 선언

```
void printarray(double data[][3], int rowsize, int colsize) {
  ...
}
```

포인터로 선언

```
void printarray(double (*data)[3], int rowsize, int colsize) {
  ...
}
```

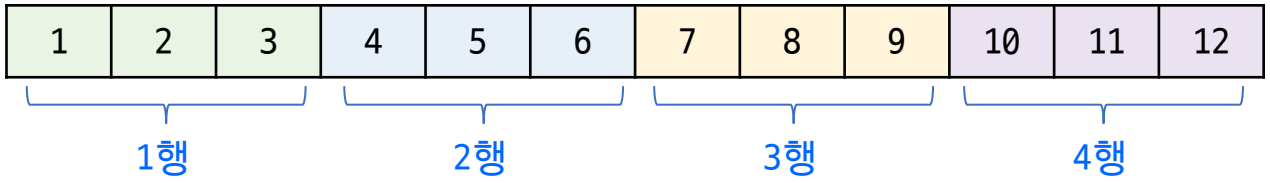
다차원 배열을 함수 매개변수로 전달하는 방법

- **2차원 배열을 함수 매개변수로 사용할 경우**
  - 반드시 컬럼(열)의 크기를 명시해야 되는 이유

column의 크기는 반드시 입력

```
double data[][3] = {{1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9},
                    {10, 11, 12}};
```

2차원 배열 Memory 저장 현황



  - 2차원 배열은 실제 메모리상에는 연속된 블록(1차원 배열)으로 저장됨
    - ➢ data[i][j]의 주소 계산:

```
data의 시작주소 + sizeof(자료형) * (i * col + j)
```

    - ➢ 컬럼의 크기가 없으면, 포인터(주소) 연산에 필요한 행의 크기(건너뛰기 간격)를 컴파일러가 알 수 없음

예제: 2차원 배열 전달

<02twodimparam.c>

```c
#include <stdio.h>

double sum(double data[][3], int, int);
void printarray(double data[][3], int, int);

int main()
{
    double x[][3] = {{1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9},
                     {10, 11, 12}};

    int rowsize = sizeof(x) / sizeof(x[0]); // 행의 수
    int colsize = sizeof(x[0]) / sizeof(x[0][0]); // 열의 수

    printf("2차원 배열의 자료값은 다음과 같습니다.\n");
    printarray(x, rowsize, colsize);

    printf("2차원 배열 원소합은 %.3lf 입니다.\n",
                sum(x, rowsize, colsize));

    return 0;
}
```

```c
void printarray(double data[][3], int rowsize, int colsize)
{
    for (int i = 0; i < rowsize; i++)
    {
        printf("%d 행 원소: ", i + 1);
        for (int j = 0; j < colsize; j++)
            printf("x[%d][%d]= %5.2lf ", i, j, data[i][j]);
        printf("\n");
    }
    printf("\n");
}

double sum(double data[][3], int rowsize, int colsize)
{
    double total = 0;

    for (int i = 0; i < rowsize; i++)
        for (int j = 0; j < colsize; j++)
            total += data[i][j];

    return total;
}
```

실행 결과

```
2차원 배열의 자료값은 다음과 같습니다.
1 행 원소: x[0][0]=  1.00  x[0][1]=  2.00  x[0][2]=  3.00
2 행 원소: x[1][0]=  4.00  x[1][1]=  5.00  x[1][2]=  6.00
3 행 원소: x[2][0]=  7.00  x[2][1]=  8.00  x[2][2]=  9.00
4 행 원소: x[3][0]= 10.00  x[3][1]= 11.00  x[3][2]= 12.00
```

가변 인자가 있는 함수

- **함수 printf() 함수원형**
  - 첫 인자 char *format 이후에 ... 표시

```
int printf(const char *format, ...);
```

- **함수 printf()를 호출하는 경우를 살펴보면**
  - 출력할 인자의 수와 자료형이 결정되지 않은 상태에서 함수를 호출
  - 출력할 인자의 수와 자료형은 인자 format에 %d %s 등으로 표현

```
printf("%d %d %f", 3, 4, 3.67);  // 4개 인자 사용
```

```
printf("%d %d %f %f", 3, 4, 3.67, 8.78);  // 5개 인자 사용
```

가변 인자(variable argument)

- **가변인자 함수**
  - 인자의 개수와 자료형이 고정되어 있지 않고, 함수 호출 시 다양하게 바뀔 수 있는 함수
  - 최소 1개 이상의 고정인자를 반드시 선언해야 됨
  - 매개변수에서 중간 이후부터 마지막 위치에 가변 인자 가능
  - 함수 정의 시 가변인자의 매개변수는 ... 으로 기술

- **함수 vatest의 함수 헤드: void vatest(int n, …)**
  - 가변 인자인 …의 앞 부분에는 반드시 매개변수가 int n처럼 고정인자가 있어야 됨
  - 고정 매개변수 int n 의 역할: 가변인자를 처리하는데 필요한 정보를 지정하는데 사용

```
int vatest(int n, ...);                 // 정상
double vasum(char *type, int n, ...); // 정상
```

```
int vafun1(char *type, ..., int n);   // 오류: 마지막 인자로 고정인자는 안됨
double vasum(...);                    // 오류: 첫 인자를 가변인자로 사용 안됨
```

가변인자가 있는 함수 구현 #1

- **가변인자가 있는 함수 구현 과정**
  - 헤더파일 `#include <stdarg.h>` 추가
  - 1. **가변인자 선언**: `va_list`
    - 변수 선언처럼 가변인자로 처리할 변수를 선언
  - 2. **가변인자 처리 시작**: `va_start()`
    - 선언된 변수에서 마지막 고정 인자를 지정해 가변인자의 시작 위치를 알리는 방법
  - 3. **가변인자 얻기**: `va_arg()`
    - 가변인자 각각의 자료형을 지정하여 가변인자를 반환 받는 절차
  - 4. **가변인자 처리 종료**: `va_end()`
    - 가변 인자에 대한 처리를 끝내는 단계

처리 절차	구문	설명
❶ 가변인자 선언	va_list argp;	va_list로 변수 argp을 선언
❷ 가변인자 처리 시작	va_start(va_list argp, prevarg)	va_start()는 첫 번째 인자로 va_list로 선언된 변수이름 argp과 두 번째 인자는 가변인자 앞의 고정인자 prevarg를 지정하여 가변인자 처리 시작
❸ 가변인자 얻기	type va_arg(va_list argp, type)	va_arg()는 첫 번째 인자로 va_start()로 초기화한 va_list 변수 argp를 받으며, 두 번째 인자로는 가변인자로 전달된 값의 type을 기술
❹ 가변인자 처리 종료	va_end(va_list argp)	va_list로 선언된 변수이름 argp의 가변인자 처리 종료

가변인자가 있는 함수 구현 #2

- **함수 int sum(int numargs, …)**
  - int numargs: 가변인자의 수를 전달 받음
  - 리턴값
    - ➢ int형인 가변인자를 처리하고 그 결과를 반환

가변인자 선언
- va_list 가변인자 변수;

가변인자 처리 시작
- va_start(가변인자 변수, 첫 고정인자);

가변인자 얻기
- va_arg(가변인자 변수, 반환 자료형);

가변인자 처리 종료
- va_end(가변인자 변수);

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

int sum(int numargs, ...)
{
    int total = 0;
    va_list argp;

    // 가변인자 처리 시작
    va_start(argp, numargs);

    . . .

    total += va_arg(argp, int);
    . . .

    va_end(argp);


    return total;
}
```

가변인자 처리 함수의 정의와 호출

<03vararg.c>

```c
#include <stdio.h>
#include <stdarg.h> //가변인자를 위한 헤더 파일

double avg(int, ...); //int 이후는 가변인자 ...

int main(void)
{
    printf("평균 %.2f\n", avg(5, 1.2, 2.1, 3.6, 4.3, 5.8));
    printf("평균 %.2f\n", avg(6, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0));

    return 0;
}

// 매개변수 numargs는 가변인자의 수를 지정
double avg(int numagrs, ...)
{
    double total = 0; //합이 저장될 변수

    va_list argp;     //1. 가변인자 변수 선언
    va_start(argp, numagrs); //2. 가변인자 처리 시작

    for (int i = 0; i < numagrs; i++) //3. 가변인자 얻기
        total += va_arg(argp, double); // double 형으로 가변인자 반환
    va_end(argp); //4. 가변인자 처리 종료

    return total / numagrs;
}
```

가변 인자의 개수를 지정

실행 결과

```
평균 3.40
평균 3.50
```

Lab 여러 정수 인자에서 최소값 찾기

- **가변인자 활용 함수 min(int n, ...)**
  - 여러 정수 중에서 최소 값을 찾아내는 함수
  - int min(int n, ...): 첫 인자 수만큼 뒤 이은 정수 중에서 최소 값을 반환하는 함수
    - min(count, 20, 30, 33, 99, 6)
    - 5개 정수 20, 30, 33, 99, 6 중에서 최소 값을 찾도록 min() 함수를 호출

<lab1minvararg.c>

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

int min(int n, ...);

int main()
{
    int count = 5;
    printf("최소 값: %d\n", min(count, 20, 30, 33, 99, 6));

    return 0;
}
```

```c
int min(int n, ...)
{
    int min, dum;

    va_list ap;
    va_start(ap, n);

    min = va_arg(ap, int); //최소 값 찾기, 첫 인자를 최소 값으로
    for (int i = 1; i < n; i++) //n-1번 반복
        if ((dum = va_arg(ap, int)) < min) // 현재 최소와 비교
            min = dum; //최소 값 수정
    va_end(ap);

    return min;
}
```

실행 결과

```
최소 값: 6
```

매개변수로 포인터를 사용하는 함수

- **함수 매개변수로 포인터 사용: 주소에 의한 호출**
  - 함수원형 void add(int *psum, int a, int b);
    - 첫 매개변수가 포인터(int *)
    - 두 번째(int a)와 세 번째(int b) 인자를 합해 첫 번째 인자가 가리키는 변수에 저장
  - main() 함수
    - int sum을 선언하여 add(&sum, m, n)을 인자로 호출
      - int형 변수 sum의 주소 전달

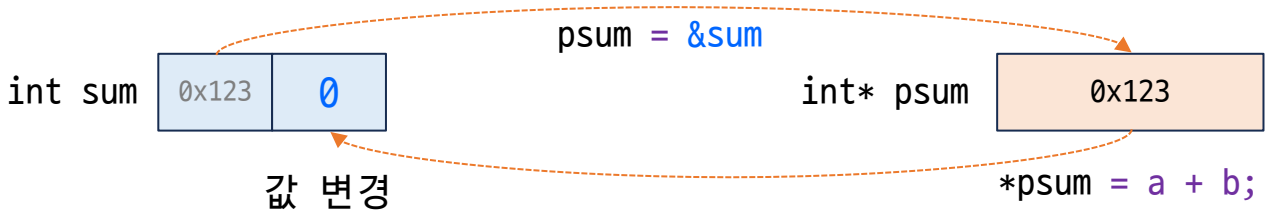
int sum 변수의 주소 전달

```
int m = 0, n = 0, sum = 0;
scanf("%d %d", &m, &n);

add(&sum, m, n);

printf("두 정수 합: %d\n", sum);
```

포인터 변수로 선언 - 호출 시 주소 전달

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

int sum 변수의 주소를 찾아가서 값 변경



함수로 변수의 주소 전달

<04ptrparam.c>

```c
#include <stdio.h>

void add(int*, int, int);

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

    printf("두 정수 입력: ");
    scanf("%d %d", &m, &n);
    add(&sum, m, n);
    printf("두 정수 합: %d\n", sum);

    return 0;
}

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

실행 결과

```
두 정수 입력: 10 20
두 정수 합: 30
```

주소(포인터)를 반환하는 함수

- **함수의 결과를 포인터로 반환하는 예**
  - 함수 int* add(int* psum, int a, int b)
    - 반환값: 정수형 포인터 (int *)
    - 두 수의 합을 첫 번째 인자가 가리키는 변수에 저장한 후
      - **포인터인 첫 번째 인자를 그대로 반환**
  - add()를 *add(&sum, m, n) 호출
    - 변수 sum에 합 a+b가 저장
    - 반환값인 포인터가 가리키는 변수인 sum을 바로 참조

```
int m = 0, n = 0, sum = 0, diff = 0;

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

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

add()함수가 리턴하는 주소의 값

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

    return psum;
}
```

sum변수의 주소 리턴

주소값 반환 함수의 정의와 이용

- 지역변수는 함수가 종료되는 시점에 메모리에서 제거되는 변수
  - 지역변수 주소값의 반환은 문제를 발생시킬 수 있음
    - ➢ 제거될 지역변수의 주소값은 반환하지 않는 것이 바람직

<05ptrretuen.c>

```c
#include <stdio.h>

int* add(int*, int, int);
int* subtract(int*, int, int);
int* multiply(int, int);

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

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

    printf("두 정수 합: %d\n", *add(&sum, m, n));
    printf("두 정수 차: %d\n", *subtract(&diff, m, n));
    printf("두 정수 곱: %d\n", *multiply(m, n));

    return 0;
}
```

합이 저장된 주소(psum)를 리턴하기 때문에 바로 출력 가능

```c
int* add(int* psum, int a, int b)
{
    *psum = a + b;
    return psum;
}

int* subtract(int* pdiff, int a, int b)
{
    *pdiff = a - b;
    return pdiff;
}

int* multiply(int a, int b)
{
    int mult = a * b;
    return &mult;
}
```

warning: address of stack memory associated with local variable 'mult' returned

Questions?