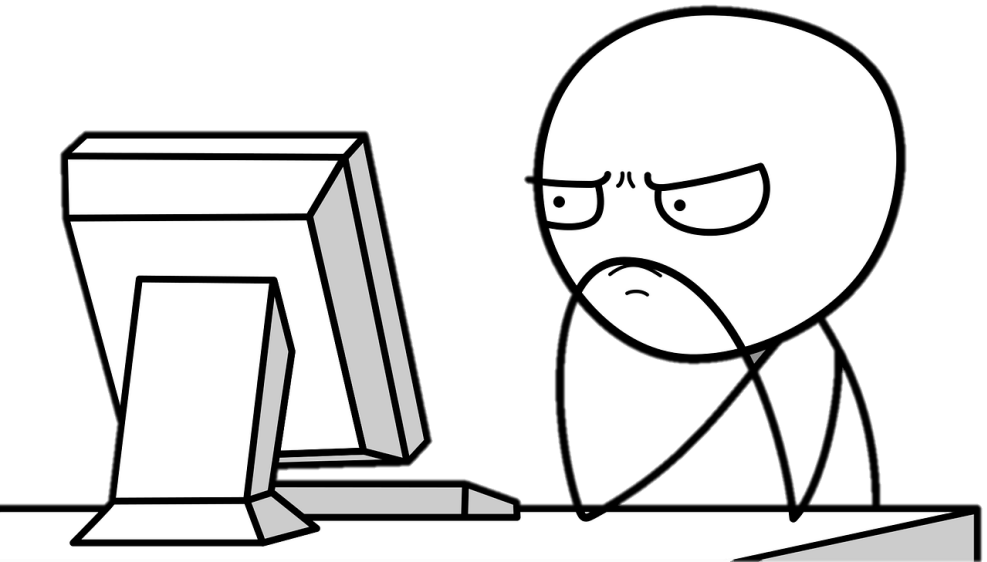

컴퓨터 프로그래밍
Computer Programming

14
함수와 포인터 활용

목차

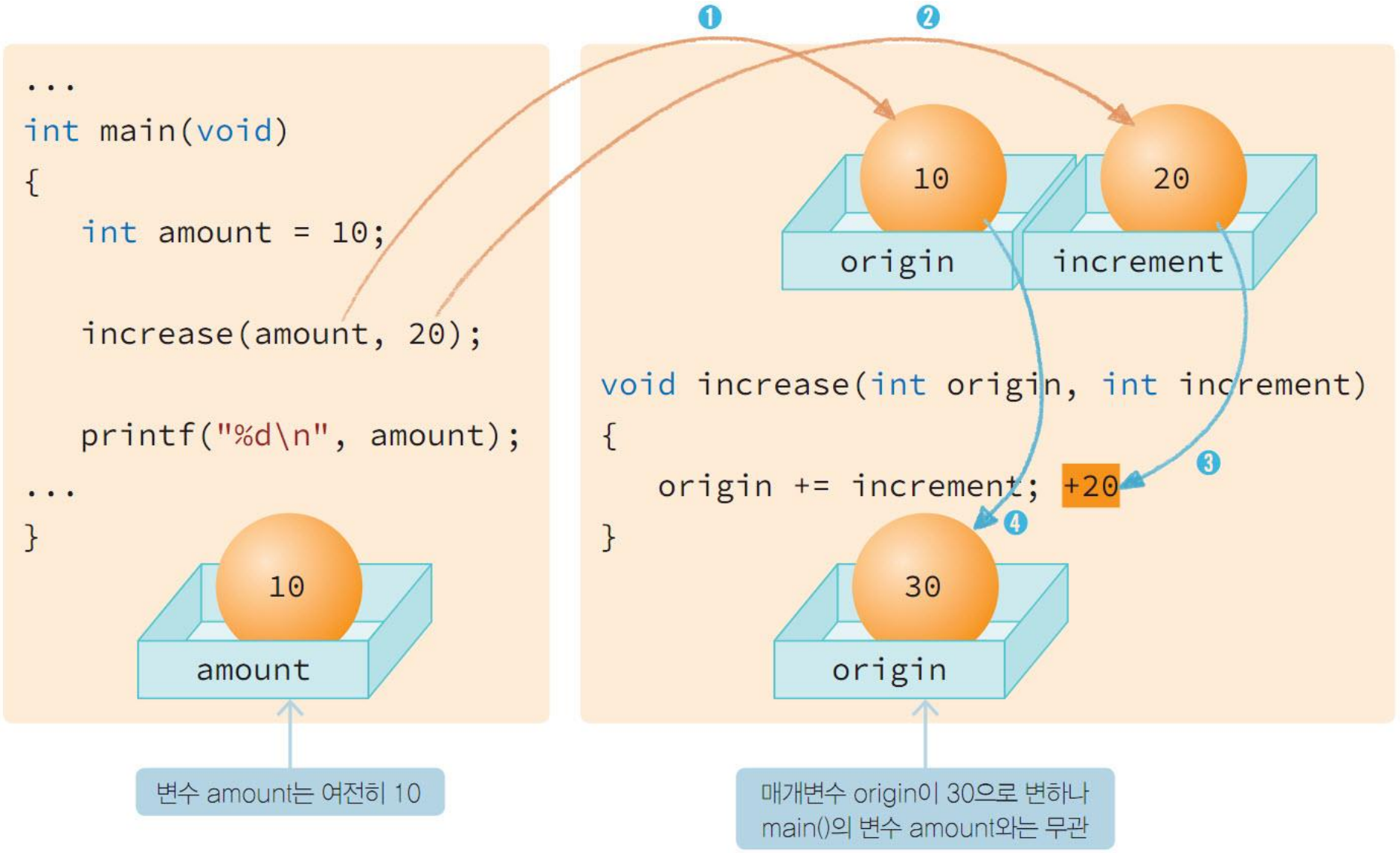
1. 함수의 인자전달 방식



함수에서 값의 전달

- C 언어는 함수의 인자 전달 방식
  - 기본적으로 값에 의한 호출call by value 방식
  - 함수 호출 시 실인자의 값이 형식인자에 복사되어 저장된다는 의미
- 함수 increase(int origin, int increment)
  - origin+= increment; 를 수행하는 간단한 함수
  - 함수 호출 시 변수 amount의 값 10이 매개변수인 origin에 복사되고,
  - 20이 매개변수인 increment에 복사
  - 함수 increase() 내부실행
  - 매개변수인 origin 값이 30으로 증가
  - 변수 amount와 매개변수 origin은 아무 관련성이 없음
  - origin은 증가해도 amount의 값은 변하지 않음
  - 함수 외부의 변수를 함수 내부에서 수정할 수 없는특징

함수에서 값의 전달


...
int main(void)
{
int amount = 10;
increase(amount, 20);
printf("%d\n", amount);
...
}
10
amount
변수 amount는 여전히 10
1
2
10
20
origin
increment
void increase(int origin, int increment)
{
origin += increment; +20
}
3
4
30
origin
매개변수 origin이 30으로 변하나
main()의 변수 amount와는 무관

Source Code #01: callbyvalue.c

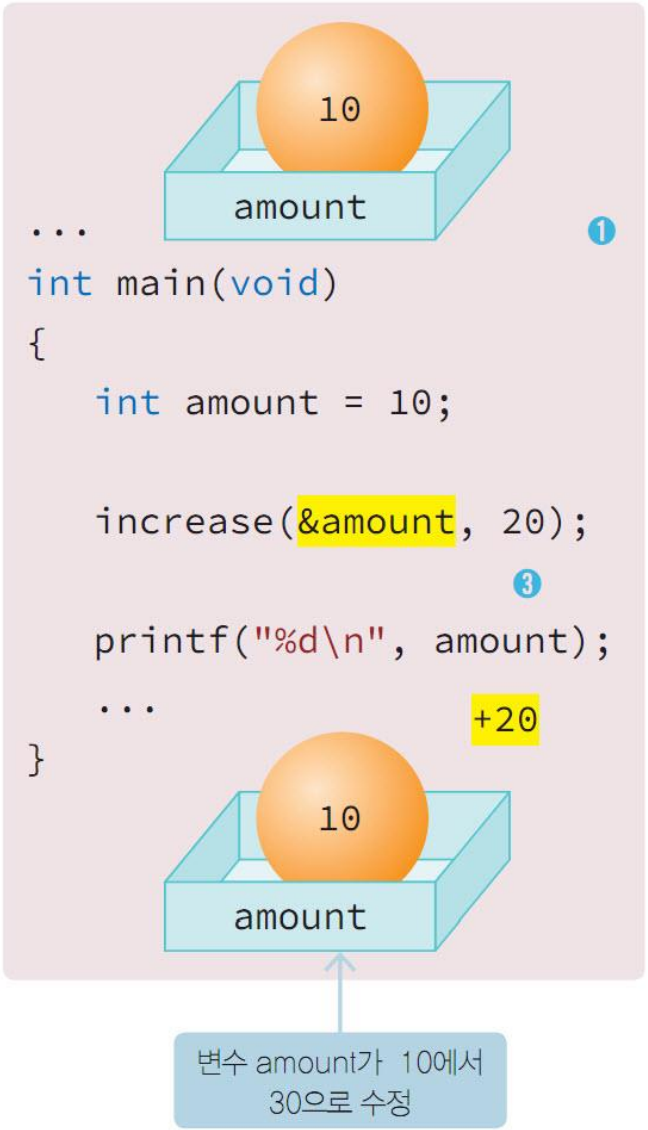
```c
// file: callbyvalue.c
#include <stdio.h>

void increase(int origin, int increment);

int main(void)
{
    int amount = 10;
    //amount가 20 증가하지 않음
    increase(amount, 20);
    printf("%d\n", amount);

    return 0;
}

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

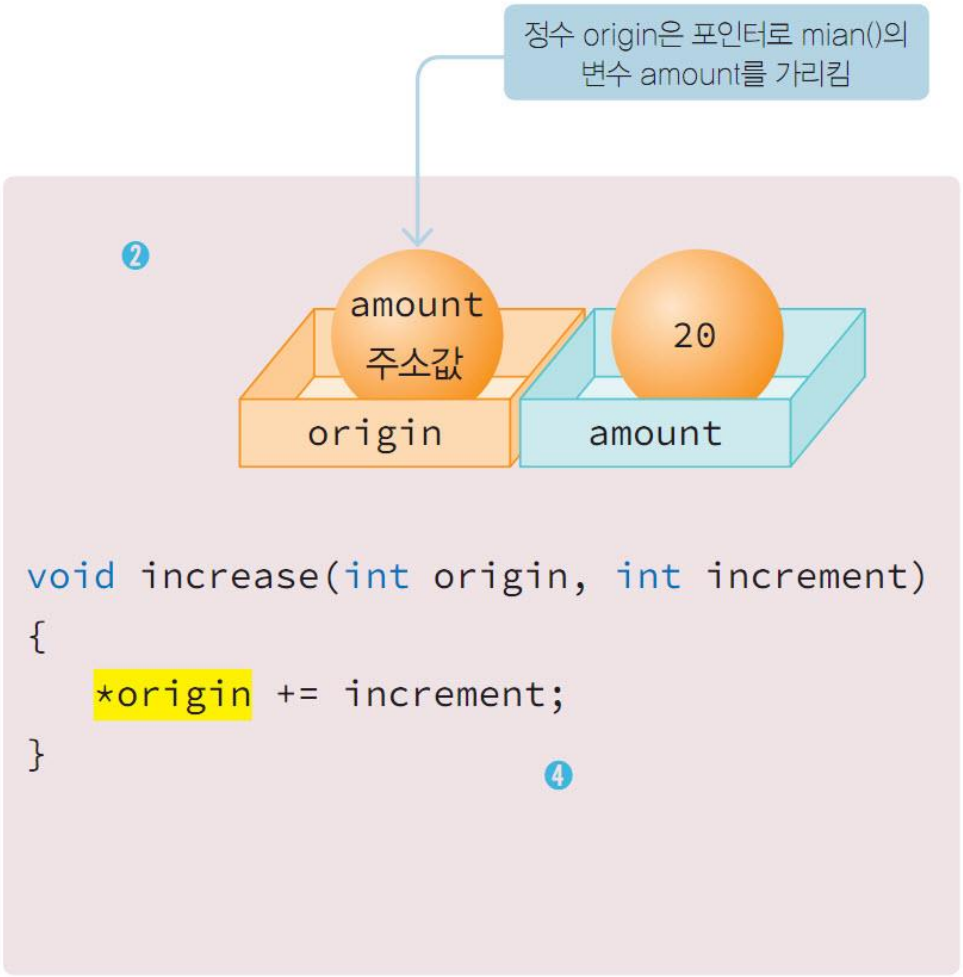
```
10
```

- 일반 매개변수로 실인자 값을 수정 불가능
- 함수 increase()
  - 첫 번째 매개변수를 int *로 수정
  - 함수 구현도 *origin += increment;로 수정하여 구현
  - 함수 호출 시 첫 번째 인자가 &amount이므로 변수 amount의 주소값이 매개변수인 origin에 복사
  - 20이 매개변수인 increment에 복사
  - 함수 increase() 내부실행
  - *origin은 변수 amount 자체를 의미
  - *origin을 증가시키면 amount의 값도 증가
  - main() 내부에서 amount의 값이 30으로 증가

참조에 의한 호출call by reference

- 포인터를 매개변수로 사용하면 함수로 전달된 실인자의 주소를 이용하여 그 변수를 참조 가능
- 함수에서 주소의 호출





Source Code #02: callbyreference.c

```c
// file: callbyreference.c
#include <stdio.h>

void increase(int *origin, int increment);

int main(void)
{
    int amount = 10;
    //&amount: amount의 주소로 호출
    increase(&amount, 20);
    printf("%d\n", amount);

    return 0;
}

void increase(int *origin, int increment)
{
    //*orogin은 origin이 가리키는 변수 자체
    *origin += increment; //그러므로 origin이 가리키는 변수 값이 20 증가
}
```

- 매개변수가 포인터이면 실인자의 값 수정 가능

```
30
```

배열이름으로 전달

- 함수의 매개변수로 배열을 전달하는 것
  - 배열의 첫 원소를 참조 매개변수로 전달하는 것과 동일
- 배열을 매개변수로 하는 함수 sum()을 구현
  - 실수형 배열의 모든 원소의 합을 구하여 반환하는 함수
  - 함수 sum()의 형식매개변수는 실수형 배열과 배열크기
  - 첫 번째 형식매개변수에서 배열자체에 배열크기를 기술하는 것은 아무 의미가 없음
  - double ary[5]보다는 double ary[]라고 기술하는 것을 권장
  - 실제로 함수 내부에서 실인자로 전달된 배열의 배열크기를 알 수 없음
  - 배열크기를 두 번째 인자로 사용
  - 매개변수를 double ary[]처럼 기술해도 단순히 double *ary처럼 포인터 변수로 인식

배열이름으로 전달

함수원형과 함수호출

```
double sum(double ary[], int n);
//double sum(double [], n); 가능

    ...

double data[] = {2.3, 3.4, 4.5, 6.7, 9.2};

    ... sum(data, 5);
```

함수호출 시 배열이름으로 배열인자를 명시한다.

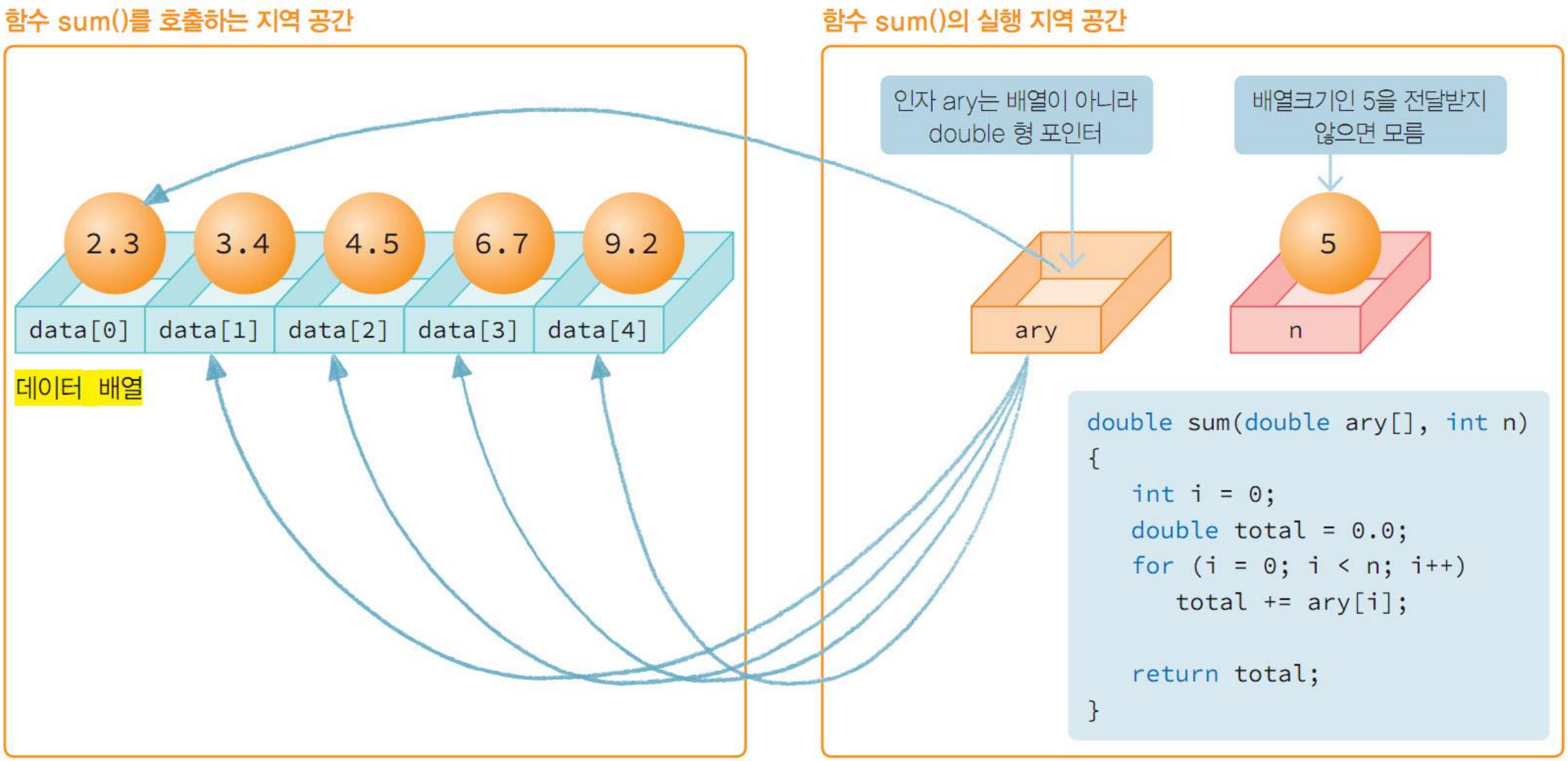
함수정의

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

    return total;
}
```

배열크기로 인자로 사용

- 만일 배열크기를 인자로 사용하지 않는다면 정해진 상수를 함수 정의 내부에서 사용해야 함
- 이런 방법은 배열크기가 변하면 소스를 수정해야 하므로 비효율적
- 배열크기에 관계없이 배열 원소의 합을 구하는 함수를 만들려면 배열크기도 하나의 인자로 사용

배열크기로 인자로 사용


함수 sum()를 호출하는 지역 공간
함수 sum()의 실행 지역 공간
2.3
3.4
4.5
6.7
9.2
data[0]
data[1]
data[2]
data[3]
data[4]
데이터 배열
인자 ary는 배열이 아니라 double 형 포인터
배열크기인 5을 전달받지 않으면 모름
ary
5
n
double sum(double ary[], int n)
{
int i = 0;
double total = 0.0;
for (i = 0; i < n; i++)
total += ary[i];
return total;
}

Source Code #03: arrayparameter.c

```c
// file: arrayparameter.c
#include <stdio.h>

#define ARYSIZE 5
double sum(double g[], int n);  //배열 원소 값을 모두 더하는 함수원형

int main(void)
{
    //배열 초기화
    double data[] = { 2.3, 3.4, 4.5, 6.7, 9.2 };

    //배열원소 출력
    for (int i = 0; i < ARYSIZE; i++)
        printf("%5.1f", data[i]);
    puts("");

    //배열 원소 값을 모두 더하는 함수호출
    printf("합: %5.1f\n", sum(data, ARYSIZE));

    return 0;
}

//배열 원소 값을 모두 더하는 함수정의
double sum(double ary[], int n)
{
    double total = 0.0;
    for (int i = 0; i < n; i++)
        total += ary[i];

    return total;
}
```

- 함수 sum()을 구현하고 이용
- 함수정의가 구현되면 함수원형을 선언한 뒤 함수호출이 가능
- 함수원형에서 매개변수는 배열이름 생략 가능
- double[]와 같이 기술 가능
- 함수호출에서 배열 인자에는 반드시 배열 이름으로 sum(data, 5)로 기술

```
  2.3  3.4  4.5  6.7  9.2
합:  26.1
```

다양한 배열원소 참조 방법

- 배열 point에서 간접연산자를 사용한 배열원소의 접근 방법은 *(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
- 배열이름 point는 주소 상수
  - sum += *(point++)는 사용 불가능
  - 증가 연산식 point++의 피연산자로 상수인 point를 사용할 수 없기 때문

다양한 배열원소 참조 방법

```
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

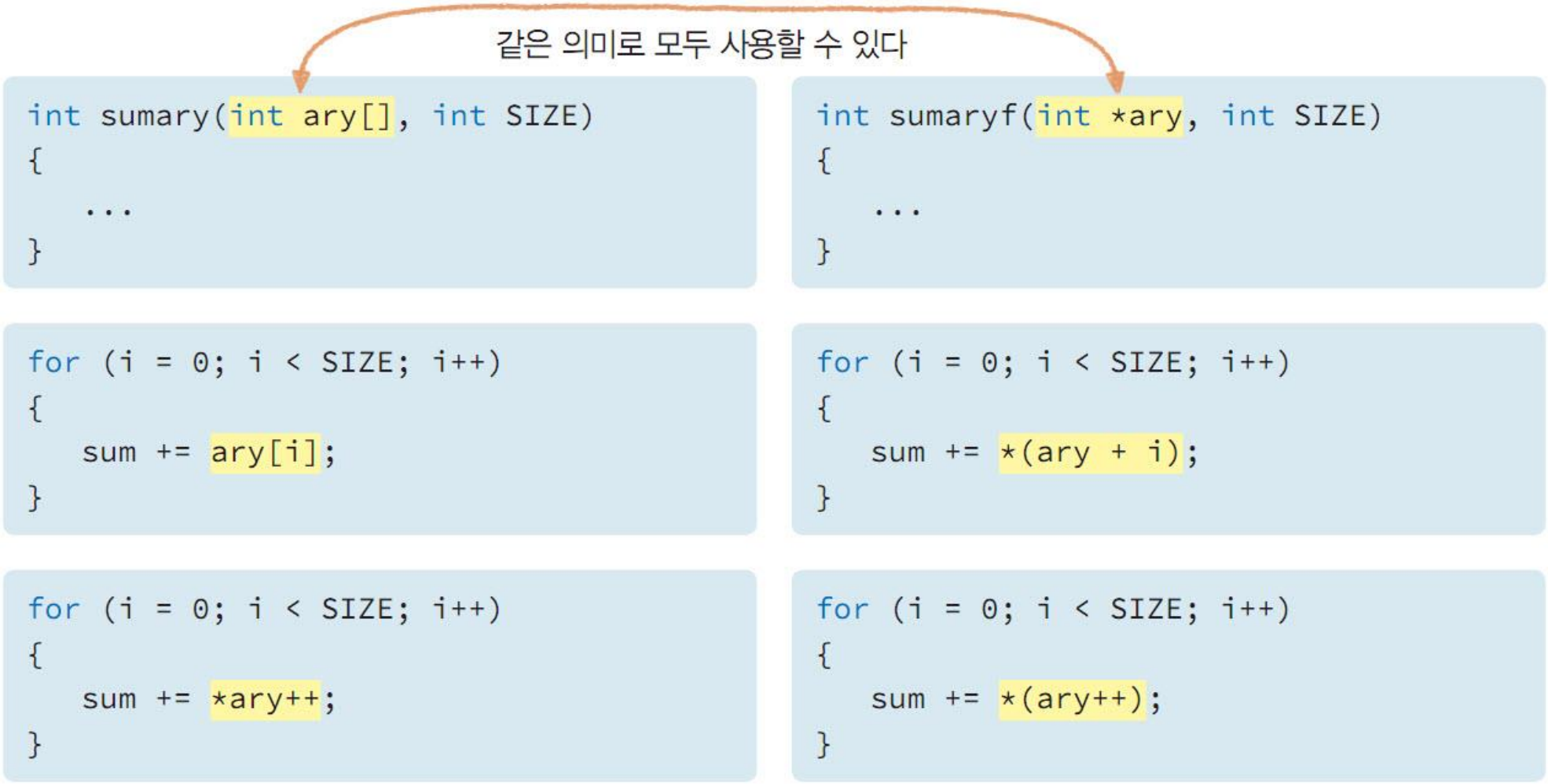
```
for (i=0; i<aryLength; i++)
   sum += *(address++);
```

오류

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

함수헤더에 배열을 인자로 기술하는 다양한 방법

- 함수헤더에 int ary[]로 기술하는 것은 int *ary로도 대체 가능



Source Code #04: arrayparam.c

```c
// file: arrayparam.c
#include <stdio.h>

int sumary(int *ary, int SIZE); //int sumary(int ary[], int SIZE)도 가능

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

    //address는 포인터 변수이며 point는 배열 상수
    int *address = point;
    //메인에서 직접 배열 합 구하기
    int sum = 0;
    for (int i = 0; i < aryLength; i++)
        sum += *(point + i);
    //sum += *(point++);    //오류발생
    //sum += *(address++);  //가능
    printf("메인에서 구한 합은 %d\n", sum);

    //함수호출하여 합 구하기
    printf("함수sumary() 호출로 구한 합은 %d\n", sumary(point, aryLength));
    printf("함수sumary() 호출로 구한 합은 %d\n", sumary(&point[0], aryLength));
    printf("함수sumary() 호출로 구한 합은 %d\n", sumary(address, aryLength));

    return 0;
}
```

```
메인에서 구한 합은 755
함수sumary() 호출로 구한 합은 755
함수sumary() 호출로 구한 합은 755
함수sumary() 호출로 구한 합은 755
```

```c

int sumary(int *ary, int SIZE) //int sumary(int ary[], int SIZE)도 가능
{
    int sum = 0;
    for (int i = 0; i < SIZE; i++)
    {
        //sum += ary[i];        //가능
        //sum += *(ary + i);    //가능
        sum += *ary++;
        //sum += *(ary++);      //가능
    }

    return sum;
}
```

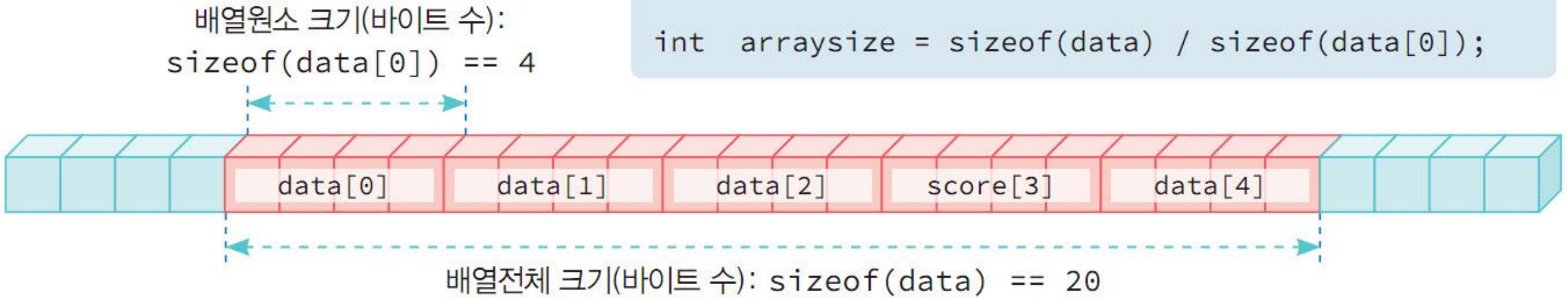
- 변수 ary는 포인터 변수로서 주소값을 저장하는 변수
  - 증가연산자의 이용이 가능
- 연산식 *ary++s는 *(ary++)와 같은 의미
- 후위 증가연산자 (ary++)의 우선순위가 가장 높기 때문에

배열크기 계산방법

- 배열이 함수인자인 경우, 대부분 배열크기도 함수인자로 하는 경우가 일반적
- 배열크기: (sizeof(배열이름) / sizeof(배열원소))

```
int data[] = {12, 23, 17, 32, 55};
```

```
배열크기(배열원소의 수) = sizeof(배열이름) / sizeof(배열원소)

int  arraysize = sizeof(data) / sizeof(data[0]);
```



Source Code #05: arrayfunction.c

```c
// file: arrayfunction
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

void readarray(double[], int);  //배열 원소값을 모두 표준입력 받는 함수원형
void printarray(double[], int); //배열 원소값을 모두 출력하는 함수원형
double sum(double[], int);      //배열 원소값을 모두 더하는 함수원형

int main(void)
{
    double data[5];
    int arraysize = sizeof(data) / sizeof(data[0]);

    printf("실수 5개의 값을 입력하세요. \n");
    readarray(data, arraysize);
    printf("\n입력한 자료값은 다음과 같습니다.\n");
    printarray(data, arraysize);
    printf("함수에서 구한 합은 %.3f 입니다.\n", sum(data, arraysize));

    return 0;
}
```

```
실수 5개의 값을 입력하세요.
data[0] = 3.22
data[1] = 4.22
data[2] = 5.33
data[3] = 9.63
data[4] = 5.98

입력한 자료값은 다음과 같습니다.
data[0] = 3.22 data[1] = 4.22 data[2] = 5.33 data[3] = 9.63 data[4] = 5.98
함수에서 구한 합은 28.380 입니다.
```

```c
//배열 원소값을 모두 표준입력 받는 함수
void readarray(double data[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("data[%d] = ", i);
        scanf("%lf", &data[i]); //(data + i)로도 가능
    }
    return;
}

//배열 원소값을 모두 출력하는 함수
void printarray(double data[], int n)
{
    for (int i = 0; i < n; i++)
        printf("data[%d] = %.2lf ", i, *(data + i));
    printf("\n");
    return;
}

//배열 원소값을 모두 더하는 함수
double sum(double data[], int n)
{
    double total = 0;
    for (int i = 0; i < n; i++)
        total += data[i]; //*(data + i)
    return total;
}
```

다차원 배열 전달

- 이차원 배열을 함수 인자로 이용하는 방법
  - 이차원 배열에서 모든 원소의 합을 구하는 함수를 구현
  - 다차원 배열을 인자로 이용하는 경우
  - 첫 번째 대괄호 내부의 크기를 제외한 다른 모든 크기는 반드시 기술
  - 이차원 배열의 행의 수를 인자로 이용하면 보다 일반화된 함수를 구현 가능
- 함수 sum()
  - 이차원 배열값을 모두 더하는 함수
  - 함수 printarray()는 인자인 이차원 배열값을 모두 출력하는 함수

다차원 배열 전달

함수원형과 함수호출

```
...
//이차원 배열값을 모두 더하는 함수원형
double sum(double data[][3], int, int);
//이차원 배열값을 모두 출력하는 함수원형
void printarray(double data[][3], int, int);

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

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

printarray(x, rowsize, colsize);

... sum(x, rowsize, colsize) ...

...
```

함수정의

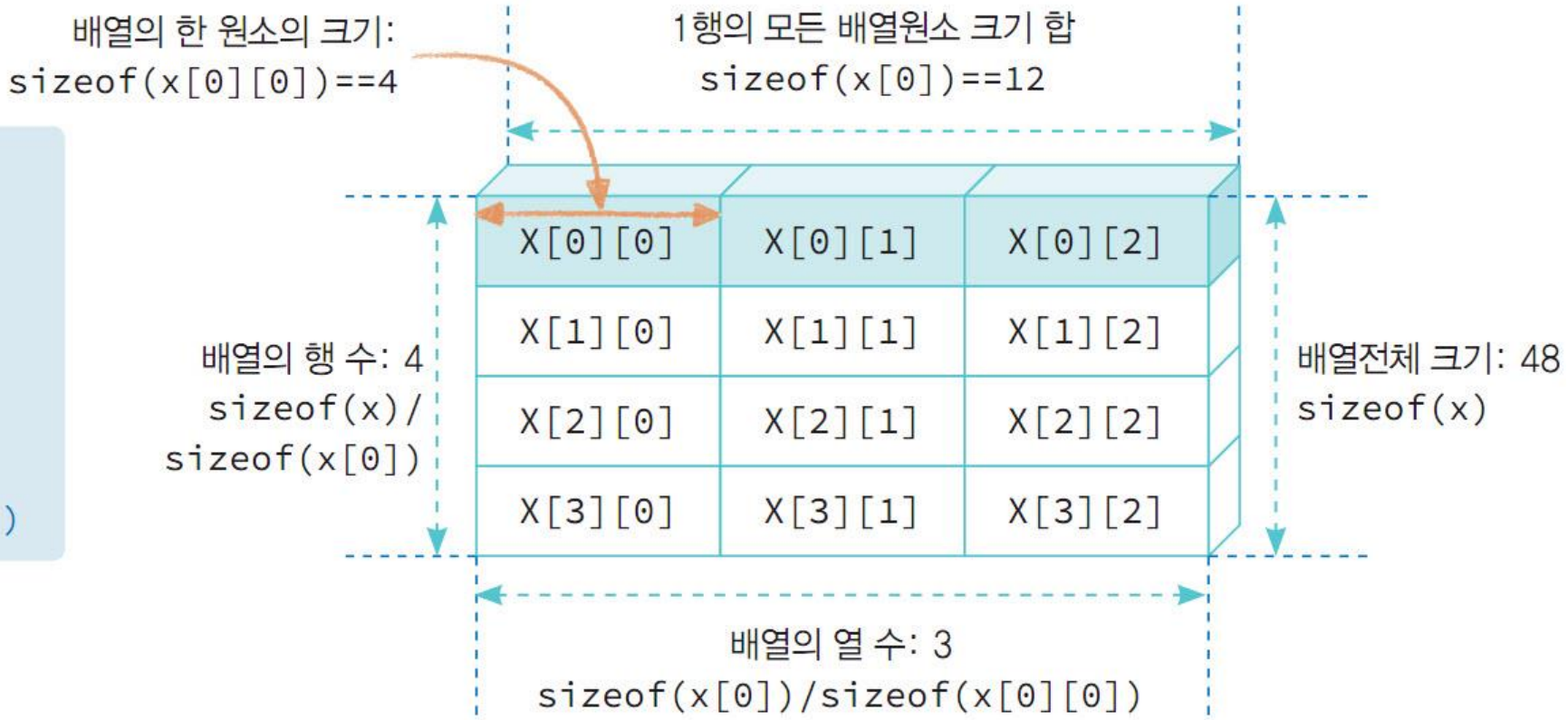
```
//이차원 배열값을 모두 출력하는 함수
void printarray(double data[][3], int rowsize, int colsize)
{

...

}

//이차원 배열값을 모두 더하는 함수
double sum(double data[][3], int rowsize, int colsize)
{
...
    for (i = 0; i < rowsize; i++)
        for (j = 0; j < colsize; j++)
            total += data[i][j];
    return total;
}
```

이차원 배열 행과 열

- 함수 sum()을 호출하려면 배열이름과 함께 행과 열의 수가 필요
- 이차원 배열의 행의 수: (sizeof(x) / sizeof(x[0]))
- 이차원 배열의 열의 수: (sizeof(x[0]) / sizeof(x[0][0]))
- sizeof(x)는 배열 전체의 바이트 수, sizeof(x[0])는 1행의 바이트 수
- sizeof(x[0][0])은 첫 번째 원소의 바이트 수

이차원 배열 행과 열

```
배열의 전체 원소 수: 12
sizeof(x) / sizeof(x[0][0])

배열의 행 수: 4
sizeof(x) / sizeof(x[0])

배열의 열 수: 3
sizeof(x[0]) / sizeof(x[0][0])
```



Source Code #06: twoarrayfunction.c

```c
// file: twodarrayfunction.c
#include <stdio.h>

//2차원 배열값을 모두 더하는 함수원형
double sum(double data[][3], int, int);
//2차원 배열값을 모두 출력하는 함수원형
void printarray(double data[][3], int, int);

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

    int rowsize = sizeof(x) / sizeof(x[0]);
    int colsize = sizeof(x[0]) / sizeof(x[0][0]);
    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
//배열값을 모두 출력하는 함수
void printarray(double data[][3], int rowsize, int colsize)
{
    for (int i = 0; i < rowsize; i++)
    {
        printf("% d행원소: ", i + 1);
        for (int j = 0; j < colsize; j++)
            printf("x[%d][%d] = %5.2lf   ", i, j, data[i][j]);
        printf("\n");
    }
    printf("\n");
}

//배열값을 모두 더하는 함수
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;
}
```

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

2차원 배열 원소합은 78.000 입니다.
```

가변 인자가 있는 함수머리

- 함수 printf() 함수원형
  - 첫 인자는 char *_Format을 제외하고는 이후에 … 표시
- 함수 printf()를 호출하는 경우를 살펴보면
  - 출력할 인자의 수와 자료형이 결정되지 않은 체 함수를 호출
  - 출력할 인자의 수와 자료형은 인자 _Format에 %d

```
//함수 printf()의 함수원형
int printf(const char *_Format, ...); //...이 무엇일까?

//함수 사용 예
printf("%d%d%f", 3, 4, 3.678); //인자가 총 4개
printf("%d%d%f%f%f", 7, 9, 2.45, 3.678, 8.98); //인자가 총 5개
```

가변 인자variable argument

- 함수에서 인자의 수와 자료형이 결정되지 않은 함수 인자 방식
- 처음 또는 앞 부분의 매개변수는 정해져 있으나
- 이후 매개변수 수와 각각의 자료형이 고정적이지 않고 변하는 인자
- 매개변수에서 중간 이후부터 마지막에 위치한 가변 인자만 가능
- 함수 정의 시 가변인자의 매개변수는 …으로 기술

가변 인자variable argument

- 함수 vatest의 함수 헤드
  - void vatest(int n, …)
  - 가변 인자인 …의 앞 부분에는 반드시 매개변수가 int n처럼 고정적이어야 함
  - 가변인자 ... 시작 전 이전 고정 매개변수
  - 가변인자를 처리하는데 필요한 정보를 지정하는데 사용

```
int vatest(int n, ...);
double vasum(char *type, int n, ...);
double vafun1(char *type, ..., int n); //오류, 마지막이 고정적일 수 없음
double vafun2(...); //오류, 처음부터 고정적일 수 없음
```

가변 인자가 있는 함수 구현

- 함수에서 가변 인자를 구현 과정
- 필요 매크로함수와 자료형을 위해 헤더파일 stdarg.h가 필요
- 가변인자 선언: 마치 변수선언처럼 가변인자로 처리할 변수를 하나 만드는 일
- 가변인자 처리 시작: 선언된 변수에서 마지막 고정 인자를 지정해 가변 인자의 시작 위치를 알리는 방법
- 가변인자 얻기: 가변인자 각각의 자료형을 지정하여 가변인자를 반환 받는 절차
  매크로 함수 va_arg()의 호출로 반환된 인자로 원하는 연산을 처리
- 가변인자 처리 종료: 가변 인자에 대한 처리를 끝내는 단계

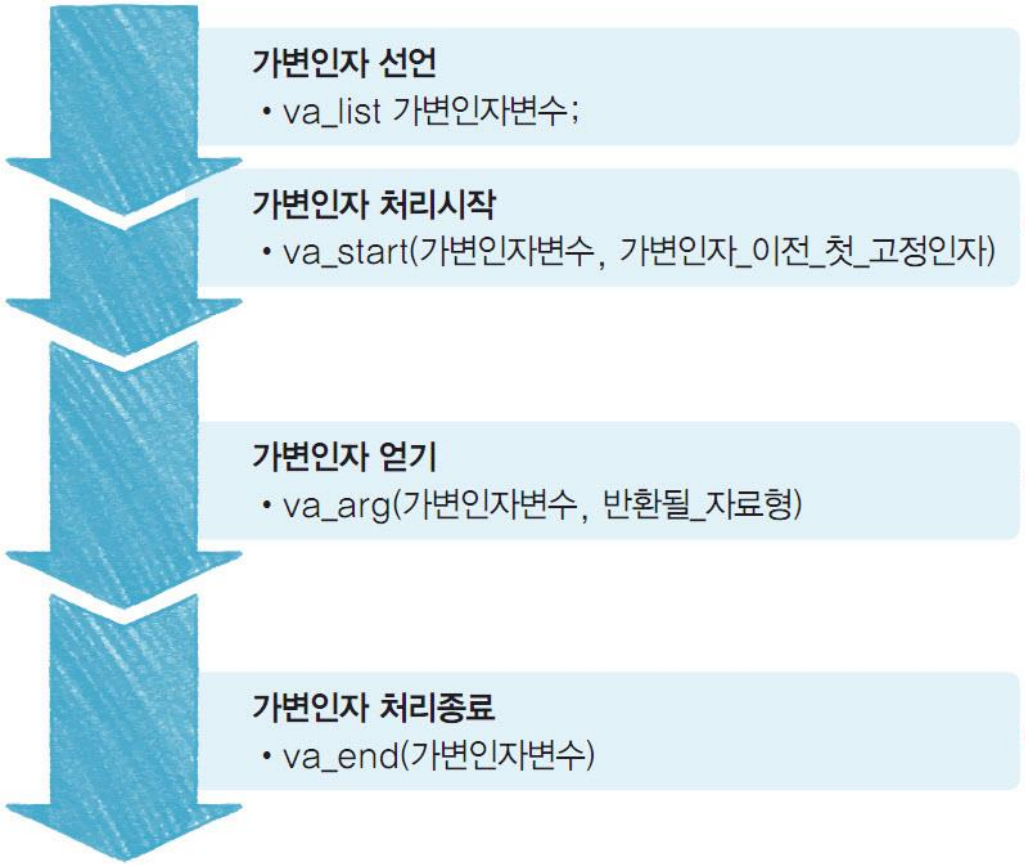
가변 인자가 있는 함수 구현

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

가변 인자가 있는 함수 구현

- 가변인자 처리 절차와 가변인자가 있는 함수 sum(int numargs, …)
- 가변인자 앞의 첫 고정인자인 numargs는 가변인자의 수
- int 형인 가변인자를 처리하여 그 결과를 반환하는 함수
- 가변인자 ... 시작 전 첫 고정 매개변수
- 이후의 가변인자를 처리하는데 필요한 정보를 지정하는데 사용

가변 인자가 있는 함수 구현



가변인자 선언
- va_list 가변인자변수;

가변인자 처리시작
- va_start(가변인자변수, 가변인자_이전_첫_고정인자)

가변인자 얻기
- va_arg(가변인자변수, 반환될_자료형)

가변인자 처리종료
- va_end(가변인자변수)

```
int sum(int numagrs, ...)
{
    va_list argp;

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

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

    ...

    //가변인자 처리 종료
    va_end(argp);

    ...
}
```

Source Code #07: vararg.c

```c
// file: vararg.c
#include <stdio.h>
#include <stdarg.h>

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

int main(void)
{
    printf("평균 %.2f\n", avg(5, 1.2, 2.1, 3.6, 4.3, 5.8));

    return 0;
}

//가변인자 ... 시작 전 첫 고정 매개변수는
//이후의 가변인자를 처리하는데 필요한 정보를 지정
//여기서에서는 가변인자의 수를 지정
double avg(int numagrs, ...)
{
    //가변인자 변수 선언
    va_list argp;

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

    double total = 0; //합이 저장될 변수
    for (int i = 0; i < numagrs; i++)
        //지정하는 double 형으로 가변인자 하나 하나를 반환
        total += va_arg(argp, double);

    //가변인자 처리 종료
    va_end(argp);

    return total / numagrs;
}
```

- 가변인자를 처리하는 함수 avg(int count, …)를 구현
- 함수 avg()의 마지막 고정인자인 count는 가변인자의 수
- double 형인 가변인자를 모두 더한 후 평균을 반환하는 함수

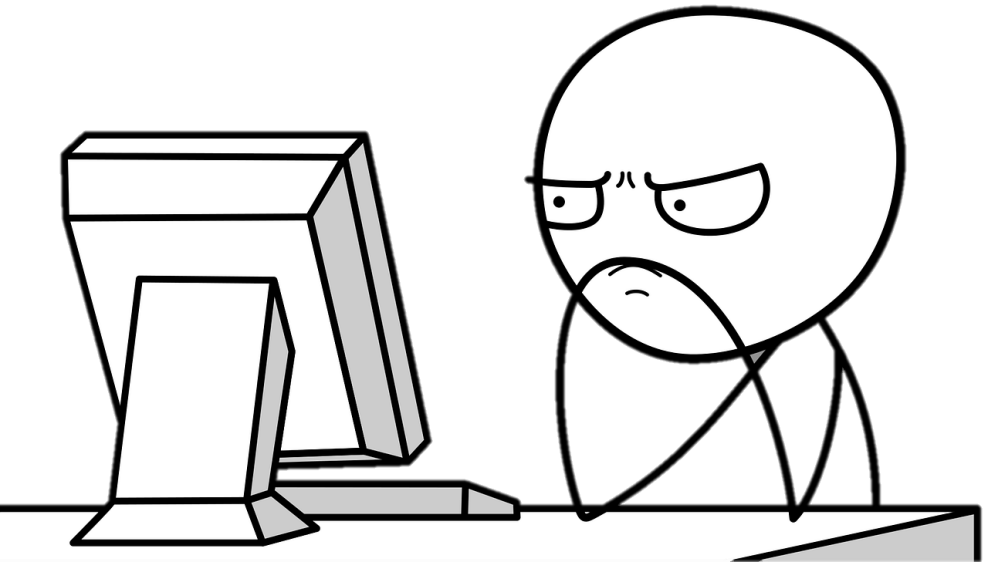
```
평균 3.40
```

Lab #01: 함수에서 배열 활용

- 함수에서 인자로 배열을 사용하면 배열은 무조건 참조에 의한 호출
  - 함수에서 배열의 내용을 수정하더라도 그 내용이 원래의 배열에 그대로 반영
  - 함수 aryprocess()는 인자로 사용된 배열의 내부 원소를 모두 1 증가시키는 함수
- 일차원 배열에서 배열의 크기
  - (배열전체 바이트 수)/(배열원소 바이트 수)
- 결과
  - 2 4 6 8 10

```c
// file: aryprocess.c
#include <stdio.h>

void aryprocess(int *ary, int SIZE);

int main(void)
{
    int data[] = { 1, 3, 5, 7, 9 };

    int aryLength = ____________________;
    aryprocess(____________);
    for (int i = 0; i < aryLength; i++)
        printf("%d ", __________);
    printf("\n");

    return 0;
}

void aryprocess(int *ary, int SIZE)
{
    for (int i = 0; i < SIZE; i++)
        ______________________
}
```

2. 포인터 전달과 반환



매개변수와 반환으로 포인터 사용

- 주소연산자 &
  - 함수에서 매개변수를 포인터로 이용하면 결국 참조에 의한 호출
  - 함수원형 void add(int *, int, int); 에서 첫 매개변수가 포인터인 int *
  - 함수 add()는 두 번째와 세 번째 인자를 합해 첫 번째 인자가 가리키는 변수에 저장 함수
  - 변수인 sum을 선언하여 주소값인 &sum을 인자로 호출

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

add(&sum, m, n);
```

함수호출 후 변수 sum에는 m과 n의 합이 저장

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

함수 add() 정의에서 합을 저장하는 문장

Source Code #08: pointerparam.c

```c
// file: pointerparam.c
#define _CRT_SECURE_NO_WARNINGS
#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;
}
```

- 함수의 결과를 포인터로 반환하는 예
- 함수원형을 int * add(int *, int, int) 로 하는 함수 add()
  - 반환값이 포인터인 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

```
두 정수 입력: 10 7
두 정수 합: 17
```

주소값 반환

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

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

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

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

Source Code #09: ptrreturn.c

```c
// file: ptrreturn.c
#define _CRT_SECURE_NO_WARNINGS
#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;
}
```

```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;
}
```

- 함수 multiply()
  - 인자인 두 수의 곱을 지역변수인 mult에 저장한 후 &mult로 포인터를 반환
- 지역변수는 함수가 종료되는 시점에 메모리에서 제거되는 변수
  - 지역변수 주소값의 반환은 문제를 발생시킬 수 있음
  - 제거될 지역변수의 주소값은 반환하지 않는 것이 바람직

```
두 정수 입력: 6 9
두 정수 합: 15
두 정수 차: -3
두 정수 곱: 54
```

상수를 위한 const 사용

- 포인터를 매개변수로 이용하면 수정된 결과를 받을 수 있어 편리
  - 이러한 포인터 인자의 잘못된 수정을 미리 예방하는 방법
  - 즉 수정을 원하지 않는 함수의 인자 앞에 키워드 const를 삽입하여 참조되는 변수가 수정될 수 없게 함
  - 키워드 const는 인자인 포인터 변수가 가리키는 내용을 수정 불가능
- 인자를 const double *a와 const double *b로 기술
  - *a와 *b를 대입연산자의 l-value로 사용 불가능
  - 즉 *a와 *b를 이용하여 그 내용을 수정 불가능
  - 상수 키워드 const의 위치는 자료형 앞이나 포인터변수 *a 앞에도 가능
  - const double *a와 double const *a 는 동일한 표현

상수를 위한 const 사용

```
// 매개변수 포인터 a, b가 가리키는 변수의 내용은 수정하지 못함
void multiply(double *result, const double *a, const double *b)
{
    *result = *a * *b;
    //다음 2문장 오류 발생
    *a = *a + 1;
    *b = *b + 1;
}
```

Source Code #10: constreference.c

```c
//file: constreference.c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

void multiply(double *, const double *, const double *);
void devideandincrement(double *, double *, double *);

int main(void)
{
    double m = 0, n = 0, mult = 0, dev = 0;

    printf("두 실수 입력: ");
    scanf("%lf %lf", &m, &n);
    multiply(&mult, &m, &n);
    devideandincrement(&dev, &m, &n);
    printf("두 실수 곱: %.2f, 나눔: %.2f\n", mult, dev);
    printf("연산 후 두 실수: %.2f, %.2f\n", m, n);

    return 0;
}

//매개변수 포인터 a, b가 가리키는 변수의 내용을 곱해 result가 가리키는 변수에 저장
//매개변수 포인터 a, b가 가리키는 변수의 내용은 수정하지 못함
void multiply(double *result, const double *a, const double *b)
{
    *result = *a * *b;
    //오류발생
    //*a = *a + 1;
    //*b = *b + 1;
}

//매개변수 포인터 a, b가 가리키는 변수의 내용을 나누어 result가 가리키는 변수에 저장한 후
//a, b가 가리키는 변수의 내용을 모두 1 증가시킴
void devideandincrement(double *result, double *a, double *b)
{
    *result = *a / *b;
    ++*a; //++(*a)이므로 a가 가리키는 변수의 값을 1 증가
    (*b)++; //b가 가리키는 변수의 값을 1 증가, *b++와는 다름
}
```

- 함수 devideandincrement(double *result, double *a, double *b)
  - 포인터 인자가 모두 const가 아니므로 그 인자가 가리키는 변수의 내용을 모두 수정 가능
  - *a와 *b를 나누어 그 결과를 *result에 저장한 후 *a와 *b를 각각 1씩 증가시키는 함수

```
두 실수 입력: 4.4 6.6
두 실수 곱: 29.04, 나눔: 0.67
연산 후 두 실수: 5.40, 7.60
```

복소수를 위한 구조체

- 구조체 complex
  - 실수부와 허수부를 나타내는 real과 img를 멤버로 구성

```
struct complex
{
    double real; //실수
    double img;  //허수
};
typedef struct complex complex;
```

자료형 struct complex

자료형 complex

자료형 struct complex와 complex 모두 복소수 자료형으로 사용 가능

복소수를 위한 구조체

- 복소수complex number
  - 실수의 개념을 확장한 수로 a + bi로 표현
  - 여기서 a와 b는 실수이며, i는 허수단위로 i2 = -1을 만족
  - a는 실수부, b는 허수부
  - 복소수에서의 사칙 연산

- 복소수의 합: (a + bi) + (c + di) = (a + b) + (c + d)i
- 복소수의 곱: (a + bi) * (c + di) = (ac − db) + (ad + bc)i
- (a + bi)의 켤레 복소수: (a − bi)
- (a − bi)의 켤레 복소수: (a + bi)

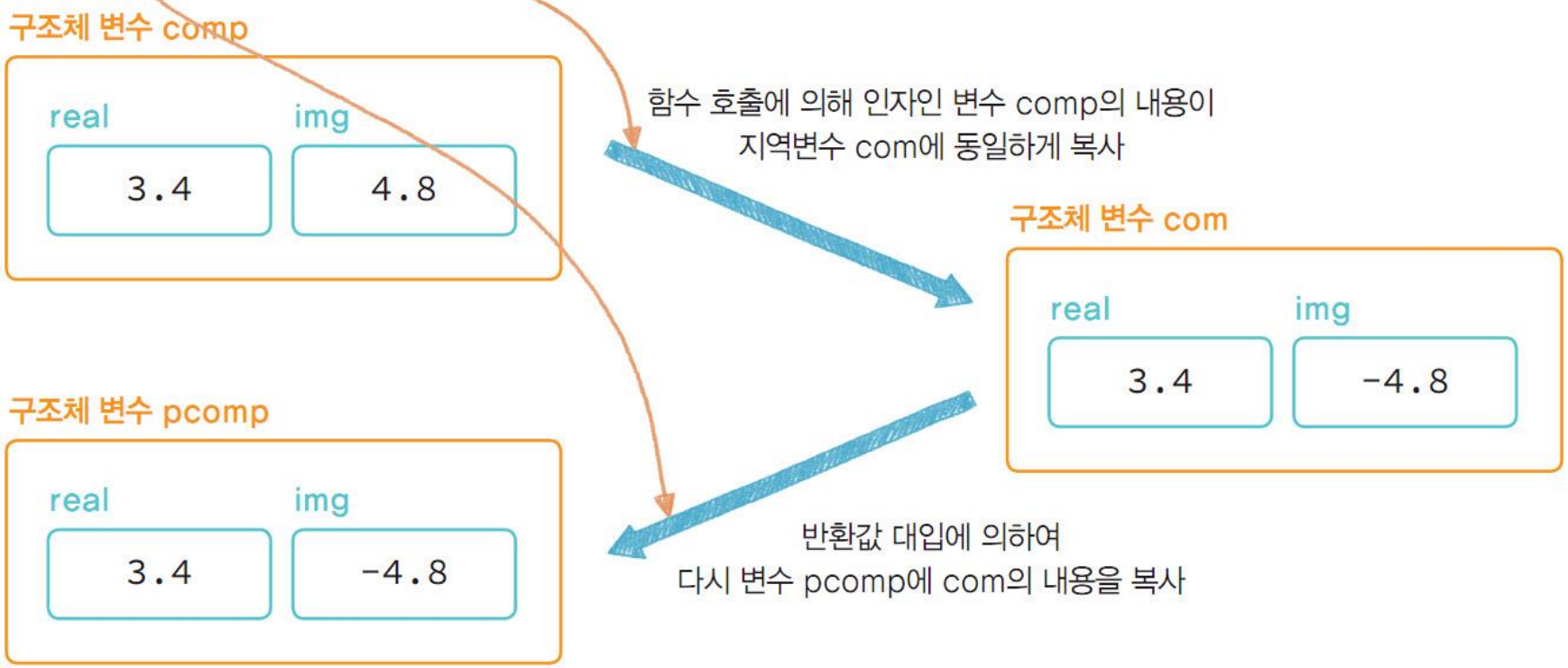
인자와 반환형으로 구조체 사용

- 함수 paircomplex1()
  - 인자인 복소수의 켤레 복소수pair complex number를 구하여 반환하는 함수
  - 복소수 (a + bi)의 켤레 복소수는 (a - bi)
  - 구조체는 함수의 인자와 반환값으로 이용이 가능
  - 다음 함수는 구조체 인자를 값에 의한 호출call by value 방식으로 이용
  - 함수에서 구조체 지역변수 com을 하나 만들어 실인자의 구조체 값을 모두 복사하는 방식으로 구조체 값을 전달 받음

인자와 반환형으로 구조체 사용

```
complex comp = { 3.4, 4.8 };
complex pcomp;

pcomp = paircomplex1(comp);
```

```
complex paircomplex1(complex com)
{
    com.img = -com.img;
    return com;
}
```



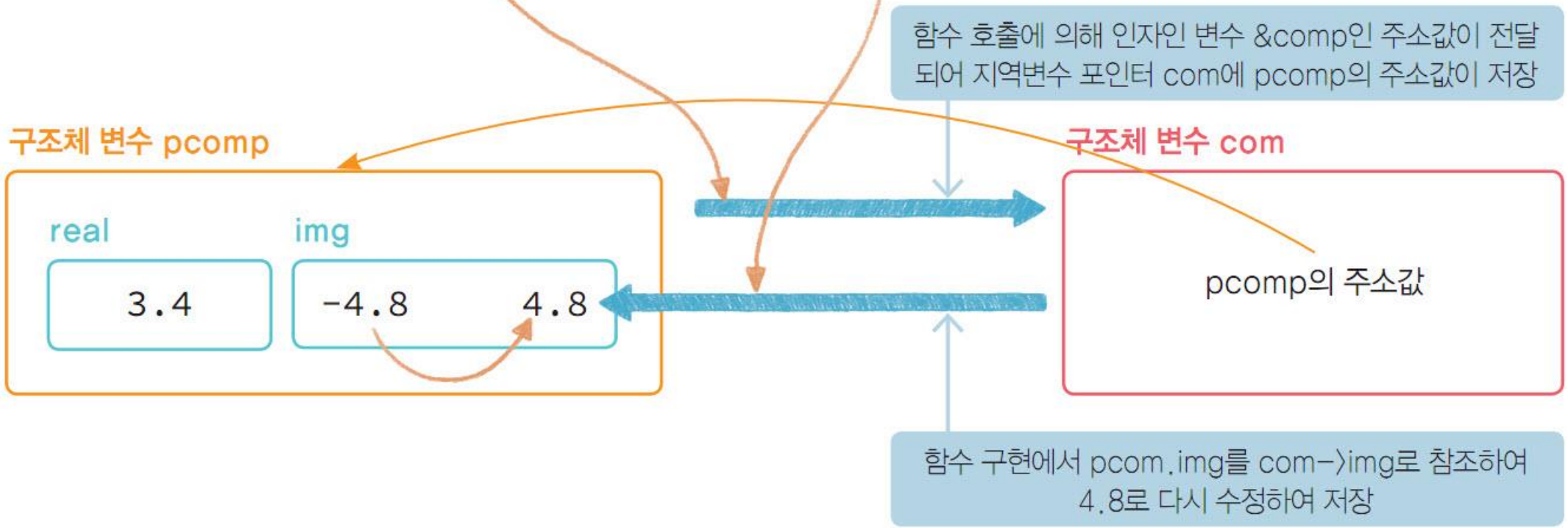
인자와 반환형으로 구조체 사용

- 이 함수를 참조에 의한 호출call by reference 방식으로 수정
- paircomplex2()는 인자를 주소값으로 저장
- 실인자의 변수 comp의 값을 직접 수정하는 방식
- 이 함수를 호출하기 위해서는 &pcomp처럼 주소값을 이용해 호출

인자와 반환형으로 구조체 사용

```
paircomplex2(&pcomp);
```

```
void paircomplex2(complex *com)
{
    com->img = -com->img;
}
```



Source Code #11: complexnumber.c

```c
//file: complexnumber.c
#include <stdio.h>

//복소수를 위한 구조체
struct complex
{
    double real; //실수
    double img;  //허수
};
//complex를 자료형으로 정의
typedef struct complex complex;

void printcomplex(complex com);
complex paircomplex1(complex com);
void paircomplex2(complex *com);

int main(void)
{
    complex comp = { 3.4, 4.8 };
    complex pcomp;

    printcomplex(comp);
    pcomp = paircomplex1(comp);
    printcomplex(pcomp);
    paircomplex2(&pcomp);
    printcomplex(pcomp);

    return 0;
}
```

```c
//구조체 자체를 인자로 사용
void printcomplex(complex com)
{
    printf("복소수(a + bi) = %5.1f + %5.1fi \n", com.real, com.img);
}

//구조체 자체를 인자로 사용하여 처리된 구조체를 다시 반환
complex paircomplex1(complex com)
{
    com.img = -com.img;
    return com;
}

//구조체 포인터를 인자로 사용
void paircomplex2(complex *com)
{
    com->img = -com->img;
}
```

```
복소수(a + bi) =   3.4 +   4.8i
복소수(a + bi) =   3.4 +  -4.8i
복소수(a + bi) =   3.4 +   4.8i
```

- 구조체를 함수의 인자로 사용하는 방식은 다른 변수와 같이 값에 의한 호출과 참조에 의한 호출 방식을 사용 가능
- 구조체가 크기가 매우 큰 구조체를 값에 의한 호출의 인자로 사용한다면 매개변수의 메모리 할당과 값의 복사에 많은 시간이 소요
- 이에 반해 주소값을 사용하는 참조에 의한 호출 방식은 메모리 할당과 값의 복사에 드는 시간이 없는 장점

Lab #02: 책 정보를 표현하는 구조체 전달

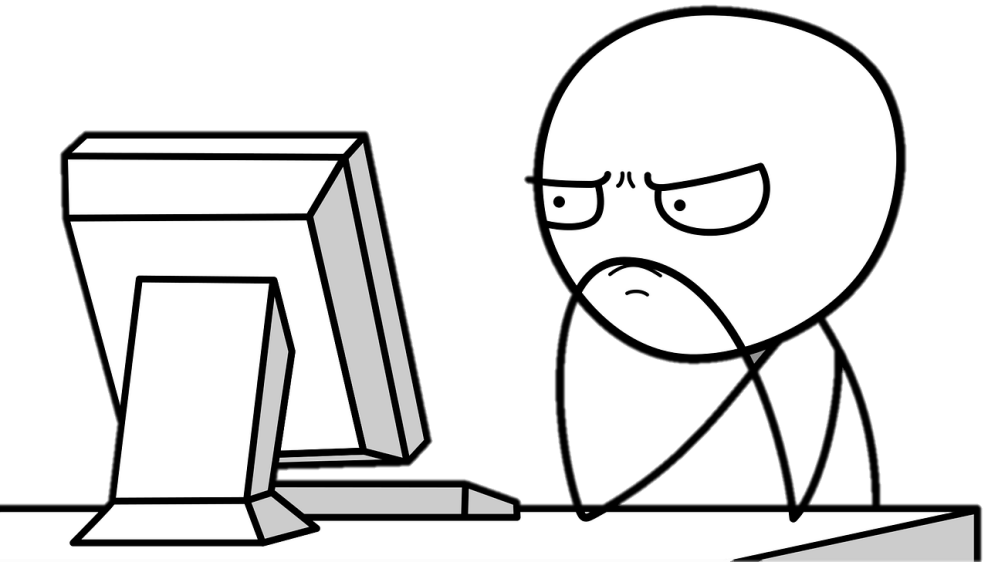
- 구조체 struct book
  - 책이름과 저자, 책번호(ISB: 국제표준도서번호 International Standard Book Number) 표현
  - 참조에 의한 호출로 출력하는 함수를 구현
  - 구조체 struct book을 자료형 book으로 정의

```
typedef struct book
{
    char  title[50];
    char  author[50];
    int   ISBN;
} book;
```

  - 함수 print()는 인자가 (book *b)으로 구조체를 포인터로 받아 책 정보를 출력
- 결과
  - 제목: 절대자바, 저자: 강환수, ISBN: 123987
  - 제목: 파이썬웹프로그래밍, 저자: 김석훈, ISBN: 2398765

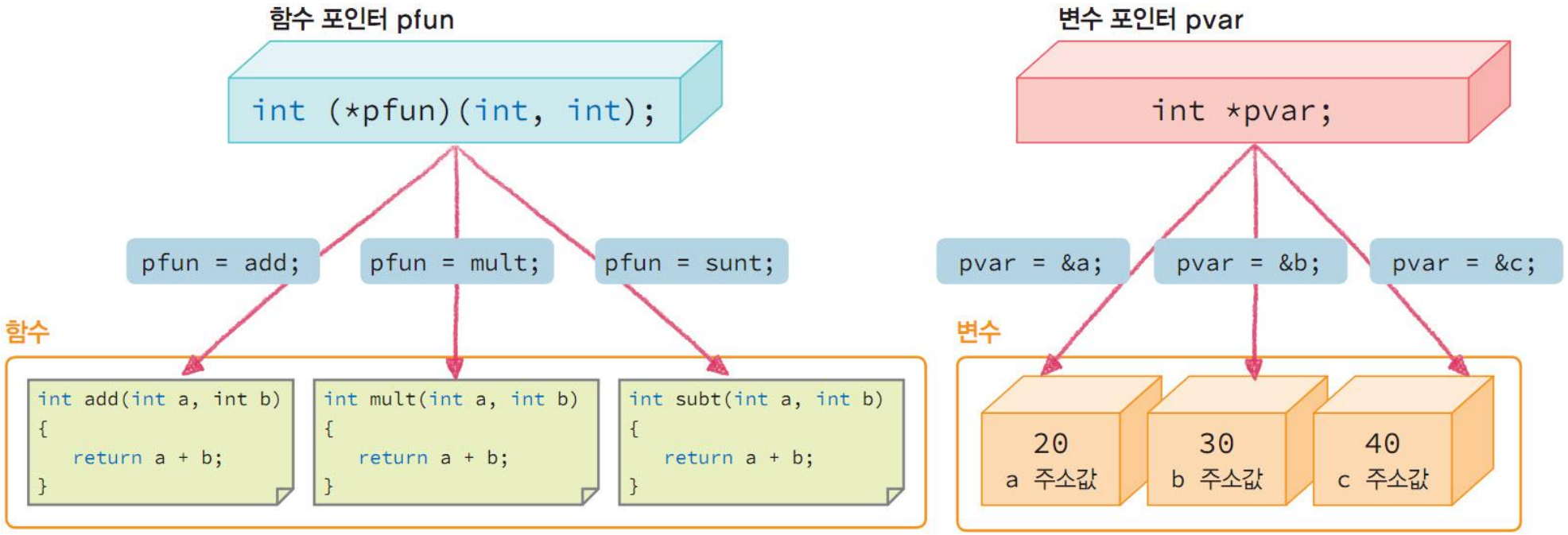
```
1  //file: bookreference.c
2  #define _CRT_SECURE_NO_WARNINGS
3  #include <stdio.h>
4  #include <string.h>
5
6  typedef struct book
7  {
8      char  title[50];
9      char  author[50];
10     int   ISBN;
11 } book;
12
13 void print(book *b);
14
15 int main()
16 {
17     book python = [redacted];
18     book java;
19     strcpy(java.title, "절대자바");
20     strcpy(java.author, "강환수");
21     java.ISBN = 123987;
22     print([redacted]);
23     print(&python);
24
25     return 0;
26 }
27
28 void print([redacted])
29 {
30     printf("제목: %s, ", b->title);
31     printf("저자: %s, ", b->author);
32     printf("ISBN: %d\n", b->ISBN);
33 }
```

3. 함수 포인터와 void 포인터



함수 포인터

- 함수 주소 저장 변수
  - 포인터의 장점은 다른 변수를 참조하여 읽거나 쓰는 것도 가능
- 함수 포인터
  - 하나의 함수 이름으로 필요에 따라 여러 함수를 사용하면 편리
  - 함수 포인터 pfun은 함수 add()와 mult() 그리고 subt()로도 사용 가능



함수 포인터

- 함수 포인터pointer to function
- 함수의 주소값을 저장하는 포인터 변수
- 즉 함수 포인터는 함수를 가리키는 포인터
- 반환형, 인자목록의 수와 각각의 자료형이 일치하는 함수의 주소를 저장할 수 있는 변수
- 함수 포인터 선언
  - 함수원형에서 함수이름을 제외한 반환형과 인자목록의 정보가 필요

함수 포인터

함수 포인터 변수 선언

```
반환자료형   (*함수  포인터변수이름)( 자료형1  매개변수이름1,  자료형2  매개변수이름2,
...);

또는

반환자료형   (*함수  포인터변수이름)( 자료형1, 자료형2, ...);
```

```
void add(double*, double, double);
void subtract(double*, double, double);
   ...
   void (*pf1)(double *z, double x, double y) = add;
   void (*pf2)(double *z, double x, double y) = subtract;
   pf2 = add;
```

함수 포인터

- 변수이름이 pf인 함수 포인터를 하나 선언
- 함수 포인터 pf는 함수 add()의 주소를 저장 가능
  - 함수원형이 void add(double*, double, double);인 함수의 주소를 저장
  - 함수원형에서 반환형인 void와 인자목록인 (double *, double, double) 정보 필요
- 여기서 주의할 점
  - (*pf)와 같이 변수이름인 pf 앞에는 *이 있어야 하며 반드시 괄호를 사용
  - 만일 괄호가 없으면 함수원형
  - pf는 함수 포인터 변수가 아니라 void *를 반환하는 함수이름

```
//잘못된 함수 포인터 선언
void *pf(double*, double, double);    //함수원형이 되어 pf는 원래 함수이름


   void (*pf)(double*, double, double); //함수 포인터
   pf = add; //변수 pf에 함수 add의 주소값을 대입 가능
```

함수 포인터

- 물론 위 함수 포인터 변수 pf
  - 함수 add()만을 가리킬 수 있는 것이 아니라 add()와 반환형과 인자목록이 같은 함수는 모두 가리킬 수 있음
  - subtract()의 반환형과 인자목록이 add()와 동일하다면 pf는 함수 subtract()도 가리킬 수 있음
  - 문장 pf = subtract; 와 같이 함수 포인터에는 괄호가 없이 함수이름만으로 대입
  - 함수 이름 add나 subtract는 주소 연산자를 함께 사용하여 &add나 &subtract로도 사용 가능
  - subtract()와 add()와 같이 함수호출로 대입해서는 오류가 발생

```
void (*pf2)(double *z, double x, double y) = add();      //오류발생
pf2 = subtract();                                         //오류발생
pf2 = add;                                                //가능
pf2 = &add;                                               //가능
pf2 = subtract;                                           //가능
pf2 = &subtract;                                          //가능
```

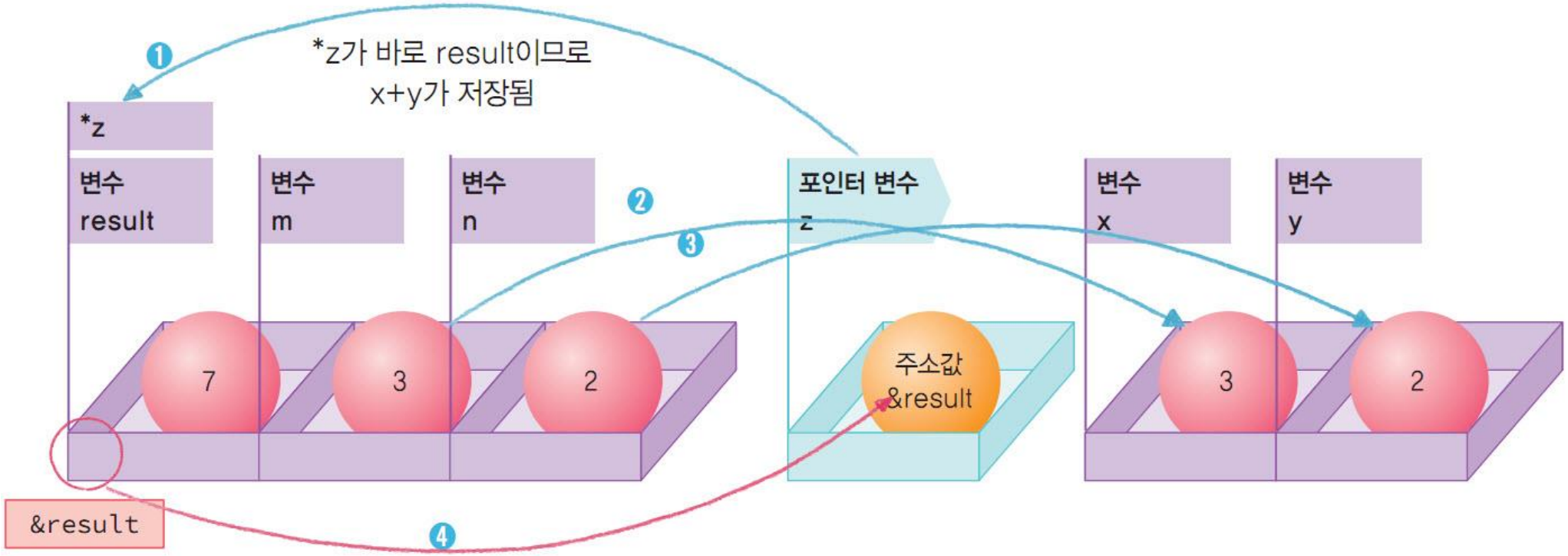
함수 포인터를 이용한 함수 호출

- 다음 예제에서 함수 add()의 구현
  - 함수 add()에서 x + y의 결과를 반환하지 않고 포인터 변수 z에 저장
  - 인자를 포인터 변수로 사용하면 함수 내부에서 수정한 값이 그대로 실인자로 반영
- 문장 pf = add;
  - 함수 포인터 변수인 pf에 함수 add()의 주소값이 저장
  - 변수 pf를 이용하여 add() 함수를 호출 가능
- 포인터 변수 pf를 이용한 함수 add()의 호출방법은 add() 호출과 동일
  - 즉 pf(&result, m, n);로 add(&result, m, n) 호출을 대체
  - 이 문장이 실행되면 변수 result에는 m + n의 결과가 저장
  - 함수 add()에서 m + n이 반영된 변수 result를 사용
  - pf(&result, m, n)은 (*pf)(&result, m, n)로도 가능

함수 포인터를 이용한 함수 호출

```
double m, n, result = 0;
void (*pf)(double*, double, double);
....
pf = add;
pf(&result, m, n);  //add(&result, m, n);
//(*pf)(&result, m, n); //이것도 사용 가능
```

```
void add(double *z, double x, double y)
{
    *z = x + y;
}
```



Source Code #12: funcptr.c

```c
// file: funcptr.c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

//함수원형
void add(double*, double, double);
void subtract(double*, double, double);

int main(void)
{
    //함수 포인터 pf를 선언
    void(*pf)(double*, double, double) = NULL;

    double m, n, result = 0;
    printf("+, -를 수행할 실수 2개를 입력하세요. >> ");
    scanf("%lf %lf", &m, &n);

    //사칙연산을 수행
    pf = add; //add() 함수를 함수 포인터 pf에 저장, &add도 가능
    pf(&result, m, n);  //add(&result, m, n);
    printf("pf = %p, 함수 add() 주소= %p\n", pf, add);
    printf("더하기 수행: %lf + %lf == %lf\n\n", m, n, result);

    pf = subtract; //subtract() 함수를 함수 포인터 pf에 저장, &subtract도 가능
    pf(&result, m, n);  //subtract(&result, m, n);
    printf("pf = %p, 함수 subtract() 주소= %p\n", pf, subtract);
    printf("  빼기 수행: %lf - %lf == %lf\n\n", m, n, result);

    return 0;
}
```

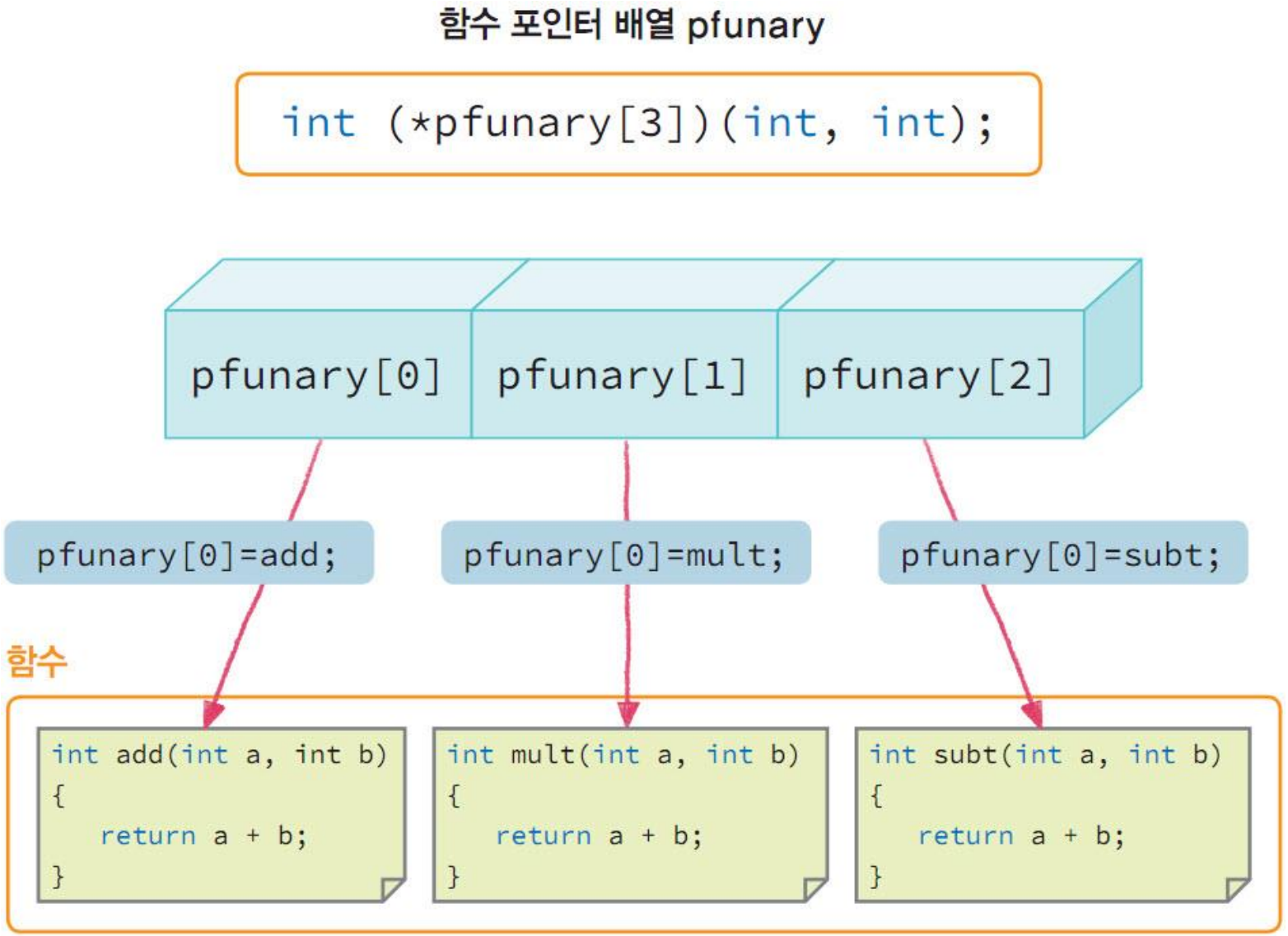
```c
// x + y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void add(double *z, double x, double y)
{
    *z = x + y;
}
// x - y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void subtract(double *z, double x, double y)
{
    *z = x - y;
}
```

```
+, -를 수행할 실수 2개를 입력하세요. >> 3.45 4.78
pf = 001011E0, 함수 add() 주소= 001011E0
더하기 수행: 3.450000 + 4.780000 == 8.230000

pf = 00101280, 함수 subtract() 주소= 00101280
  빼기 수행: 3.450000 - 4.780000 == -1.330000
```

함수 포인터 배열array of function pointer

- 원소로 여러 개의 함수 포인터를 선언하는 함수 포인터 배열
- 크기가 3인 함수 포인터 배열 pfunary는 문장 int (*pfunary[3])(int, int); 으로 선언
- 배열 pfunary의 각 원소가 가리키는 함수
- 반환값이 int이고 인자목록이 (int, int)

함수 포인터 배열array of function pointer



함수 포인터 배열 선언

- 함수 포인터 배열선언 구문
  - 배열 fpary의 각 원소가 가리키는 함수
  - 반환값이 void이고 인자목록이 (double*, double, double)

함수 포인터 배열 선언

```
반환자료형   (*배열이름[배열크기])( 자료형1 매개변수이름1, 자료형2 매개변수이름2, ...);

또는

반환자료형   (*배열이름[배열크기])( 자료형1, 자료형2, ...);
```

```
void add(double*, double, double);
void subtract(double*, double, double);
void multiply(double*, double, double);
void devide(double*, double, double);
   ...
   void (*fpary[4])(double*, double, double);
```

함수 포인터 배열 선언

- 함수 포인터 배열선언구문
  - 배열 fpary을 선언한 이후에 함수 4개를 각각의 배열원소에 저장
  - 배열 fpary을 선언하면서 함수 4개의 주소값을 초기화하는 문장

```
void (*fpary[4])(double*, double, double);
fpary[0] = add;
fpary[1] = subtract;
fpary[2] = multiply;
fpary[3] = devide;
```

배열의 선언과 초기화 문장으로 간단히 처리

```
void (*fpary[4])(double*, double, double) = {add, subtract, multiply, devide};
```

Source Code #13: fptrary.c

```c
// file: fptrary.c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

//함수원형
void add(double*, double, double);
void subtract(double*, double, double);
void multiply(double*, double, double);
void devide(double*, double, double);

int main(void)
{
    char op[4] = { '+', '-', '*', '/' };
    //함수 포인터 선언하면서 초기화 과정
    void(*fpary[4])(double*, double, double) = { add, subtract, multiply, devide };

    double m, n, result;
    printf("사칙연산을 수행할 실수 2개를 입력하세요. >> ");
    scanf("%lf %lf", &m, &n);
    //사칙연산을 배열의 첨자를 이용하여 수행
    for (int i = 0; i < 4; i++)
    {
        fpary[i](&result, m, n);
        printf("%.2lf %c %.2lf == %.2lf\n", m, op[i], n, result);
    }

    return 0;
}
```

```c
// x + y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void add(double *z, double x, double y)
{
    *z = x + y;
}
// x - y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void subtract(double *z, double x, double y)
{
    *z = x - y;
}
// x * y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void multiply(double *z, double x, double y)
{
    *z = x * y;
}
// x / y 연산 결과를 z가 가리키는 변수에 저장하는 함수
void devide(double *z, double x, double y)
{
    *z = x / y;
}
```

```
사칙연산을 수행할 실수 2개를 입력하세요. >> 3.87 6.93
3.87 + 6.93 == 10.80
3.87 - 6.93 == -3.06
3.87 * 6.93 == 26.82
3.87 / 6.93 == 0.56
```

void 포인터

- void 포인터 개념
  - 포인터는 주소값을 저장하는 변수
  - int *, double * 처럼 가리키는 대상의 구체적인 자료형 포인터로 사용이 일반적
  - 주소값이란 참조를 시작하는 주소에 불과
  - 자료형을 알아야 참조할 범위와 내용을 해석할 방법을 알 수 있음
- void 포인터(void *)는 무엇일까?
  - void 포인터는 자료형을 무시하고 주소값만을 다루는 포인터
  - 대상에 상관없이 모든 자료형의 주소를 저장할 수 있는 만능 포인터로 사용 가능
  - void 포인터에는 일반 포인터는 물론 배열과 구조체 심지어 함수 주소도 저장 가능

void 포인터

```
char ch = 'A';
int data = 5;
double value = 34.76;

void *vp;                    //void 포인터 변수 vp 선언

vp = &ch;                    //ch의 주소 만을 저장
vp = &data;                  //data의 주소 만을 저장
vp = &value;                 //value의 주소 만을 저장
```

Source Code #14: voidptrbasic.c

```c
// file: voidptrbasic.c
#include <stdio.h>

void myprint(void)
{
    printf("void 포인터, 신기하네요!\n");
}

int main(void)
{
    int m = 10;
    double d = 3.98;

    void *p = &m; //m의 주소 만을 저장
    printf("주소 %d\n", p); //주소 값 출력
    //printf("%d\n", *p); //오류발생

    p = &d; //d의 주소 만을 저장
    printf("주소 %d\n", p);

    p = myprint; //함수 myprint()의 주소 만을 저장
    printf("주소 %d\n", p);

    return 0;
}
```

- 다양한 자료형의 포인터를 저장하는 void 포인터

```
주소 13630856
주소 13630840
주소 1708917
```

void 포인터 활용

- void 포인터는 모든 주소를 저장 가능
- 가리키는 변수를 참조하거나 수정이 불가능
- 주소값으로 변수를 참조하려면 결국 자료형으로 참조범위를 알아야 하는데
- void 포인터는 이러한 정보가 전혀 없이 주소 만을 담는 변수에 불과하기 때문
- void 포인터는 자료형 정보는 없이 임시로 주소 만을 저장하는 포인터
- 그러므로 실제 void 포인터로 변수를 참조하기 위해서는 자료형 변환이 필요

void 포인터 활용

```
int m = 10;      double x = 3.98;


void *p = &m;
int n = *(int *)p; //int * 로 변환
n = *p; //오류
```

오류: "void" 형식의 값을 "int" 형식의 엔터티에 할당할 수 없습니다.

```
p = &x;
int y= *(double *)p; //double * 로 변환
y = *p; //오류
```

오류: "void" 형식의 값을 "int" 형식의 엔터티에 할당할 수 없습니다.

Source Code #15: voidptr.c

```c
// file: voidptr.c
#include <stdio.h>

void myprint(void)
{
    printf("void 포인터, 신기하네요!\n");
}

int main(void)
{
    int m = 10;
    double d = 3.98;
    char str[][20] = { { "C 언어," }, { "재미있네요!" } };

    void *p = &m;
    printf("p 참조 정수: %d\n", *(int *)p); //int * 로 변환

    p = &d;
    printf("p 참조 실수: %.2f\n", *(double *)p); //double * 로 변환

    p = myprint;
    printf("p 참조 함수 실행 : ");
    ((void(*)(void)) p)(); //함수 포인터인 void(*)(void) 로 변환하여 호출 ()

    p = str;
    //열이 20인 이차원 배열로 변환하여 1행과 1행의 문자열 출력
    printf("p 참조 2차원 배열: %s %s\n", (char(*)[20])p, (char(*)[20])p + 1);
    printf("str 참조 2차원 배열: %s %s\n", str, str + 1);

    return 0;
}
```

- void 포인터로 다양한 자료의 참조

```
p 참조 정수: 10
p 참조 실수: 3.98
p 참조 함수 실행 : void 포인터, 신기하네요!
p 참조 2차원 배열: C 언어, 재미있네요!
str 참조 2차원 배열: C 언어, 재미있네요!
```

Lab #03: 함수 포인터 배열의 활용

- 배열크기가 3인 함수 포인터 배열을 선언
  - 각각 더하기와 빼기, 그리고 곱하기를 수행하는 함수를 각각 저장
  - 연산을 수행하는 방법과 순서를 나타내는 문자열 "*+-"를 저장하여 문자열 순서대로 곱하기, 더하기, 빼기를 수행하는 프로그램을 작성
- 연산의 피연산자는 3과 5로 고정하여 다음과 같은 결과로 수행
  - * 결과: 15
  - + 결과: 8
  - - 결과: -2

```c
// file: funcpointer.c
#include <stdio.h>

int add(int a, int b);
int mult(int a, int b);
int subt(int a, int b);

int main(void)
{
    ____________;
    pfunary[0] = ____;
    pfunary[1] = mult;
    pfunary[2] = subt;

    char *ops = "*+-";
    char op;
    while (__________)
        switch (op) {
        case '+': printf("%c 결과: %d\n", op, pfunary[0](3, 5));
            break;
        case '-': printf("%c 결과: %d\n", op, __________);
            break;
        case '*': printf("%c 결과: %d\n", op, pfunary[1](3, 5));
            break;
        }

    return 0;
}

int add(int a, int b)
{
    return a + b;
}
int mult(int a, int b)
{
    return a * b;
}
int subt(int a, int b)
{
    return a - b;
}
```


I
PROGRAMMING