프로그래밍 기초

14장. 함수와 포인터 활용 Part 2

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

함수 매개변수에 const *사용

- **포인터를 매개변수로 이용하면 수정된 결과를 받을 수 있어 편리하지만**
  - 의도하지 않게 포인터 변수의 값을 수정할 수 있음
- **포인터 인자의 잘못된 수정을 미리 예방하는 방법**
  - 수정을 원하지 않는 함수의 인자 앞에 키워드 const를 삽입
- **const double *a, const double *b**
  - 즉 *a와 *b를 이용하여 그 내용을 수정할 수 없음
  - 상수 키워드 const의 위치는 자료형 앞이나 포인터변수 *a 앞에도 가능
    - `const` double *a와 double `const` *a 는 동일한 표현

```
void multiply(double *result, const double *a, const double *b)
{
    *result = *a * *b;

    *a = *a + 1; //오류발생
    *b = *b + 1; //오류발생
}
```

매개변수 a, b는 내용을 수정할 수 없음

함수 매개변수에 const의 사용 방법과 의미 (실습)

<06constparam.c>

```c
#include <stdio.h>

void multiply(double*, const double*, const double*);
void divideandincrement(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);
    divideandincrement(&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;
}
```

```c
void multiply(double* result, const double* a, const double* b)
{
    *result = *a * *b;
    //*a = *a + 1; //오류발생
    //*b = *b + 1; //오류발생
}

void divideandincrement(double* result, double* a, double* b)
{
    *result = *a / *b;
    ++*a;   //++(*a): *a = *a + 1, a가 가리키는 변수의 값을 1 증가
    (*b)++;  //b가 가리키는 변수의 값을 1 증가, *b = *b + 1
}
```

실행 결과

```
두 실수 입력: 12.5 4.5
두 실수 곱: 56.25, 나눔: 2.78
연산 후 두 실수: 13.50, 5.50
```

함수 수행 후 입력된 값이 각각 1 증가

함수의 구조체 전달과 반환

- **복소수를 위한 구조체 complex**
  - 실수부와 허수부를 나타내는 real과 img를 멤버로 구성

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

- **복소수(complex number)**
  - 실수의 개념을 확장한 수로 $a + bi$로 표현
  - 여기서 a와 b는 실수이며, i는 허수단위로 i^2 = -1을 만족
    - a는 실수부, b는 허수부
  - 복소수에서의 사칙 연산

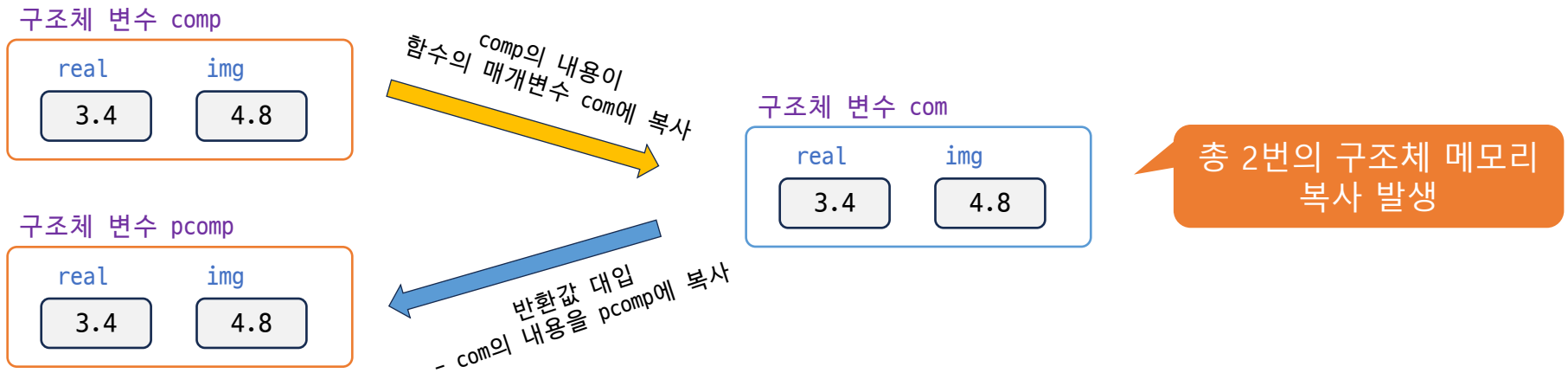
복소수의 합: *(a+bi) + (c+di) = (a+b) + (c+d)i*
복소수의 곱: *(a+bi) * (c+di) = (ac-db) + (ad+bc)i*
*(a+bi)*의 켤레 복소수(complex conjugate): *(a-bi)*
*(a-bi)*의 켤레 복소수: *(a+bi)*

인자와 반환형으로 구조체 사용: Call by value 방식

- **함수 complex pcomplexvalue(complex com)**
  - 인자의 켤레 복소수(complex conjugate)를 계산해서 반환하는 함수
    - 복소수 (a + bi)의 켤레 복소수는 (a - bi)
  - 함수의 인자와 반환값을 구조체로 사용: 사용 자제
    - 구조체 인자를 값에 의한 호출(call by value) 방식으로 이용
    - 매개변수 complex com은 실인자의 구조체(comp)의 값을 모두 복사해서 구조체 값을 전달 받음

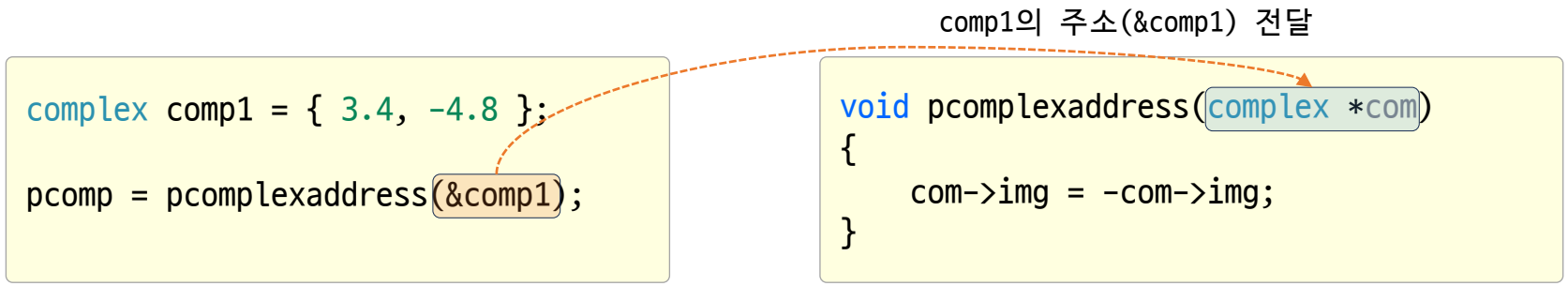
```
complex comp = { 5.8, 7.2 };
complex pcomp;

pcomp = pcomplexvalue(comp);
```

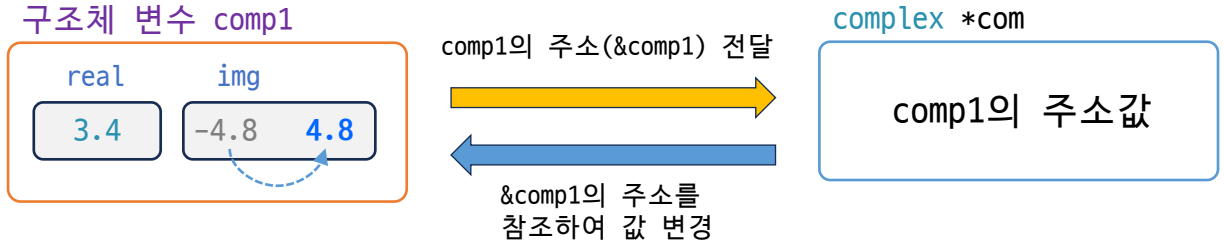
```
complex pcomplexvalue(complex com)
{
    com.img = -com.img;
    return com; //구조체를 반환
}
```



인자와 반환형으로 구조체 사용: Call by address 방식

- **함수 void pcomplexaddress(complex *com)**
  - 주소에 의한 호출(call by address) 방식으로 수정
    - 인자를 주소값(complex *com)으로 저장
    - 실인자의 변수 comp의 값을 직접 수정하는 방식
    - 이 함수를 호출하기 위해서는 &comp1 처럼 주소를 함수로 전달해서 호출





구조체를 사용한 값에 의한 호출과 주소에 의한 호출

- **함수의 인자로 구조체를 사용**
  - 값에 의한 호출과 참조에 의한 호출 방식을 사용 가능

- **구조체를 값에 의한 호출(Call by value)**
  - 크기가 매우 큰 구조체라면 매개변수의 메모리 할당과 값의 복사에 많은 시간이 소요
  - 사용 자제

- **구조체를 주소에 의한 호출(Call by address)**
  - 메모리 절약 및 속도 향상
    - ➢ 메모리 할당과 값의 복사에 소요되는 시간이 필요 없음
  - 주소에 의한 호출 방식 사용을 권장

함수의 인자에 구조체의 값 및 주소 사용 예제 (실습)

<07complexnum.c>

```c
#include <stdio.h>

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

void printcomplex(complex com);
complex pcomplexvalue(complex com);
void pcomplexaddress(complex* com);
int main(void)
{
    complex comp = { 5.8, 7.2 };
    complex pcomp;
    complex comp1 = {3.4, -4.8};

    pcomp = pcomplexvalue(comp);
    printcomplex(pcomp);

    pcomplexaddress(&comp1);
    printcomplex(pcomp);

    return 0;
}
```

Call by value
- 2번의 메모리 복사

Call by address
- 단순히 메모리 참조

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

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

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

권장하지 않음

실행 결과

```
복소수 =   5.8 +   7.2i
복소수 =   3.4 +   4.8i
```

LAB 책 정보를 표현하는 구조체 전달

- **구조체 struct book**
  - 책이름과 저자
  - 책번호(ISBN: 국제표준도서번호 International Standard Book Number) 표현
  - 주소에 의한 호출로 출력
  - 자료형 book으로 정의

- **함수 void print(book *b)**
  - 구조체를 포인터로 받아 책 정보를 출력

lab2bookaddress.c 난이도: ★

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

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

[                    ]

int main()
{
   book python = { "파이썬으로 배우는 누구나 코딩", "강환수", 979117 };
   book comintro;
   strcpy(comintro.title, "소프트웨어 중심사회의 컴퓨터개론");
   strcpy(comintro.author, "강환수");
   comintro.ISBN = 437894;
   [                    ]
   print(&python);

   return 0;
}

void print(book* b)
{
   printf("제목: %s, ", b->title);
   printf("저자: %s, ", b->author);
   printf("ISBN: %d\n", b->ISBN);
}
```

```c
12  void print(book* b)
21    print(&comintro);
```

함수 포인터 정의

- **함수 포인터(pointer to function)**
  - 함수의 주소를 저장하는 포인터 변수
    - 해당 주소를 통해 함수를 호출할 수 있는 포인터
    - 반환형, 매개변수의 수와 각각의 자료형이 일치하는 함수의 주소를 저장할 수 있는 변수
  - 함수 포인터 선언
    - 함수원형에서 함수 이름을 제외한 반환형과 인자 목록의 정보가 필요

```
반환자료형 (*함수포인터변수이름) (자료형1 변수이름1, 자료형2 변수이름2, ...);
```

```
반환자료형 (*함수포인터변수이름) (자료형1, 자료형2, ...);
```

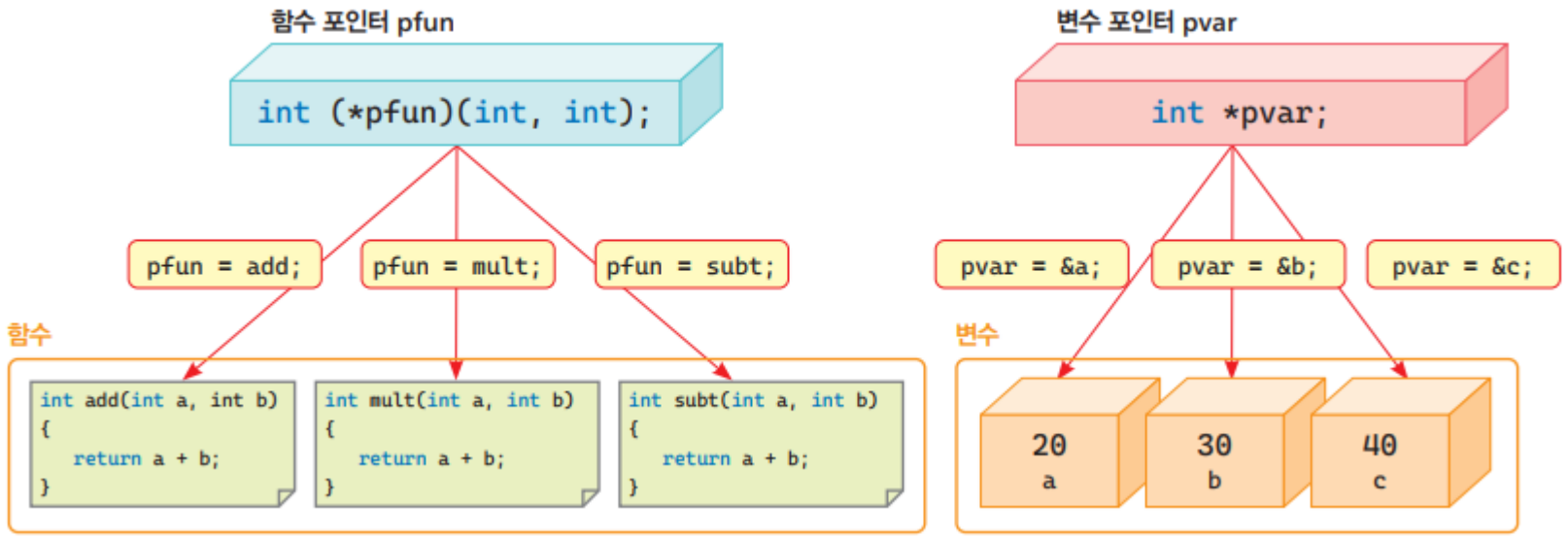
    - int형 매개변수 2개, int를 반환하는 함수 포인터 선언 예시

```
int (*fp) (int, int);
```

함수 포인터 사용 장점

- **포인터 장점**
  - 포인터의 장점은 여러 변수를 참조하여 읽거나 쓰는 것이 가능

- **함수 포인터 장점**
  - 하나의 함수 이름으로 필요에 따라 여러 함수를 사용
  - 함수 포인터 pfun: 함수 add()와 mult() 그리고 subt()로도 사용 가능



함수 포인터 선언

- **함수 void add(double *, double, double)의 주소 저장하는 함수 포인터 선언**
  - 함수 포인터 pf 선언
    - 연결할 함수(add)의 반환형인 void와 인자목록인 (double *, double, double) 정보 필요
    - (*pf)와 같이 함수 포인터 이름 pf 앞에는 *가 있어야 하며 반드시 괄호를 사용

```
void (*pf)(double*, double, double);

pf = add;  // pf에 함수 add의 주소 대입
```

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

    - 함수 포인터에는 함수 이름만 대입해야 됨

  - 함수 포인터 선언 시 주의할 점

```
void *pf(double*, double, double); // 잘못된 선언
```

⇒

```
void* pf(double*, double, double);
```

    - *pf를 둘러싸는 괄호가 없음: void *를 반환하는 함수 pf가 됨

함수 포인터 변수에 대입

- **함수 포인터 변수 pf2**
  - add()와 반환형과 인자 목록이 같은 함수는 모두 가리킬 수 있음
  - subtract()의 반환형과 인자 목록이 add()와 동일하다면
    - pf는 함수 subtract()도 가리킬 수 있음

  - 문장 pf2 = subtract;
    - 함수 포인터에는 **괄호가 없이 함수이름만 대입**
    - 함수 add나 subtract는 주소 연산자를 함께 사용하여 &add나 &subtract로도 사용 가능
    - subtract()와 add()와 같이 함수 호출로 대입해서는 오류가 발생

✖
```
void *pf(double*, double, double) = add();      // 오류 발생
pf = subtract();                                // 오류 발생
```

○
```
pf = add;
pf = subtract;
```
함수 이름: 함수 코드가 위치한 메모리의 시작 주소

```
// 가장 일반적 방법
// 가장 일반적 방법
```

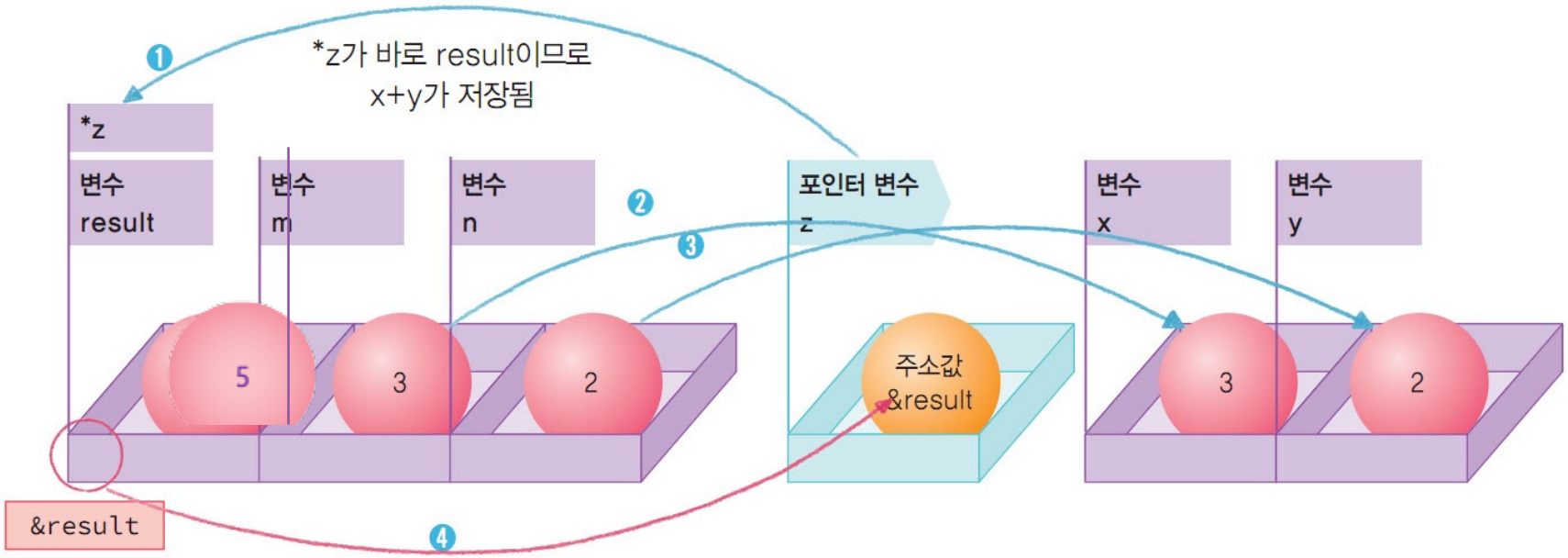
```
pf = &add;
pf = &subtract;
```
함수의 주소를 강조 - 내부적으로 동일하게 처리

```
// 가능 (주소 강조)
// 가능 (주소 강조)
```

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;
}
```



함수 주소를 저장하는 함수 포인터의 선언과 사용 (실습)

<08ptrfunction.c>

```c
#include <stdio.h>

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("연산 +, -를 수행할 실수 2개를 입력하세요. >> ");
    scanf("%lf %lf", &m, &n);

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

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

    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;
}
```

실행 결과

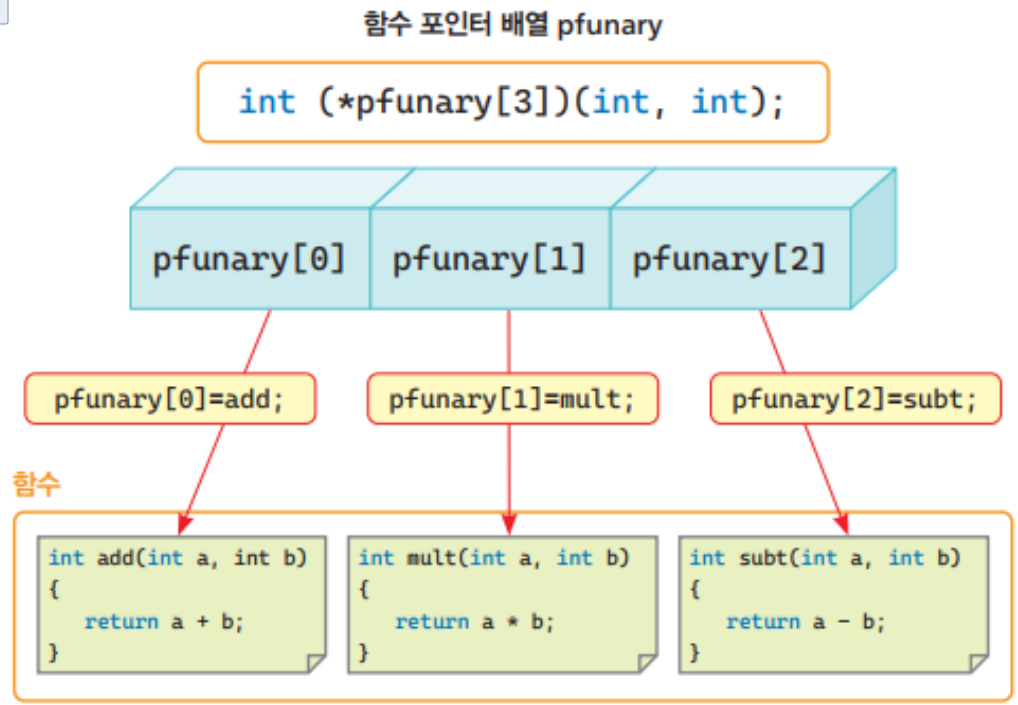
```
연산 +, -를 수행할 실수 2개를 입력하세요. >> 3 2

더하기 수행: 3.000000 + 2.000000 == 5.000000
0x104b93e9c 0x104b93e9c
  빼기 수행: 3.000000 - 2.000000 == 1.000000
0x104b93ec8 0x104b93ec8
```

함수 포인터 배열 개념

- **함수 포인터 배열(array of function pointer)**
  - 원소로 여러 개의 함수 포인터를 선언하는 배열
    - 크기가 3인 함수 포인터 배열 pfunary는 문장

```
int (*pfunary[3])(int, int);
```

    - 반환값이 int이고 인자목록이 (int, int)
    - 배열 pfunary의 각 원소가 각각 다른 함수를 가리킴



함수 포인터 배열 선언

- **함수 포인터 배열선언 구문**
  - 배열 fpary의 각 원소가 가리키는 함수
  - void (*fpary[4])(double*, double, double);
    - ➢ 반환값 void, 인자목록이 (double*, double, double)

```
반환자료형 (*배열이름[배열크기]) (자료형1 변수이름1, 자료형2 변수이름2, ...);
```

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

- **함수 포인터 초기화**
  - 배열 fpary을 선언한 이후에 함수 4개의 주소를 각각의 배열 원소에 저장

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

배열 선언과 함께 초기화

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

함수 포인터 배열의 선언과 사용 (실습)

<09aryptr.c>

```c
#include <stdio.h>

void add(double*, double, double);
void subtract(double*, double, double);
void multiply(double*, double, double);
void divide(double*, double, double);

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

    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 divide(double* z, double x, double y)
{
    *z = x / y;
}
```

```
사칙연산을 수행할 실수 2개를 입력하세요. >> 50.56 3.4
50.56 + 3.40 == 53.96
50.56 - 3.40 == 47.16
50.56 * 3.40 == 171.90
50.56 / 3.40 == 14.87
```

void 포인터 (void *)

- **포인터는 주소값을 저장하는 변수**
  - int *, double * 처럼 가리키는 대상의 구체적인 자료형을 포인터로 사용하는 것이 일반적
    - 자료형을 알아야 참조할 범위와 내용을 해석할 방법을 알 수 있음

- **void 포인터(void *)는 무엇일까?**
  - void 포인터는 자료형을 무시하고 주소만을 다루는 포인터
    - 대상에 상관없이 **모든 자료형의 주소를 저장할 수 있는 만능 포인터로 사용 가능**
    - void 포인터에는 일반 포인터는 물론 배열과 구조체, 함수 주소도 저장 가능

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

void * vp;

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

```
void* memcpy(void *dest, const void* src, size_t* count);
```

void 포인터 활용

- **void 포인터는 모든 주소를 저장 가능**
  - void 포인터가 가리키는 변수를 참조하거나 수정이 불가능
  - 주소만으로 변수를 참조하려면 자료형으로 참조범위를 알아야 함
    - ➢ void 포인터는 자료형의 정보가 전혀 없이 주소만을 담는 변수에 불과
  - void 포인터는 자료형 정보는 없이 임시로 주소만을 저장하는 포인터
    - ➢ void 포인터로 변수를 참조하기 위해서는 **자료형 변환이 필요**

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

void *p = &m;

int n = *(int *)p;          // int* 로 변환
n = *p;                     // 오류

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

type casting(형 변환) 과정이 반드시 필요

포인터 void *의 선언과 활용

<10voidptr.c>

```c
#include <stdio.h>

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

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

    void* p = &m;               // m의 주소 만을 저장
    printf("%p ", p);           // 주소 값 출력
    //printf("%d\n", *p);       // 컴파일 오류 발생
    printf("%d\n", *(int*)p); // int * 로 변환

    p = &d;
    printf("%p ", p);               // 주소 값 출력
    printf("%.2f\n", *(double*)p); // double * 로 변환
```

```c
void (*pf)(void);
pf = myprint;
pf()
```

```c
    p = myprint;
    //함수 포인터인 void(*)(void) 로 변환하여 호출 ()
    ((void(*)(void))p)();

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

    return 0;
}
```

실행 결과

```
0x16d3a6e68 10
0x16d3a6e60 3.98
void 포인터 신기하네요!
C 언어, 재미있네요!
C 언어, 재미있네요!
```

void pointer 예제

<11voidptr1.c>

```c
#include <stdio.h>
#define SIZE 10

void printArray(void *vPtr, int size, int type);

int main()
{
    int num[SIZE] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    float fractional[SIZE] = {1.1f, 1.2f, 1.3f, 1.4f, 1.5f,
                              1.6f, 1.7f, 1.8f, 1.9f, 2.0f};
    char characters[SIZE] = {'C', 'o', 'd', 'e',
                             'f', 'o', 'r', 'w', 'i', 'n'};

    printf("\nElements of integer array: ");
    printArray(num, SIZE, 1);

    printf("\nElements of float array: ");
    printArray(fractional, SIZE, 2);

    printf("\nElements of character array: ");
    printArray(characters, SIZE, 3);
    printf("\n");
    return 0;
}
```

```c
void printArray(void *vPtr, int size, int type)
{
    int i;

    for (i = 0; i < size; i++)
    {
        switch (type)
        {
        case 1:
            // printf("%d, ", *((int *)vPtr + i));
            // 배열 형태로 사용 가능
            printf("%d, ", ((int *)vPtr)[i]);
            break;

        case 2:
            printf("%.1f, ", *((float *)vPtr + i));
            break;

        case 3:
            printf("%c, ", *((char *)vPtr + i));
            break;
        }
    }
}
```

```
Elements of integer array: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
Elements of float array: 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0,
Elements of character array: C, o, d, e, f, o, r, w, i, n,
```

LAB 함수 포인터 배열의 활용

- 문자열 "*+-“ 연산 순서대로 수행하는 프로그램

```c
#include <stdio.h>

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;
}

int main(void)
{
    int (*pfunary[3])(int, int);
    pfunary[0] = add;
    pfunary[1] = mult;
    pfunary[2] = subt;

    int m = 8, n = 6;
    char* ops = "*+-";
    char op;
```

```c
    while (op = *ops++)
    {
        switch (op)
        {
        case '+':
            printf("%c 결과: %d\n", op, pfunary[0](m, n));
            break;
        case '-':
            printf("%c 결과: %d\n", op, pfunary[2](m, n));
            break;
        case '*':
            printf("%c 결과: %d\n", op, pfunary[1](m, n));
            break;
        }
    }
    return 0;
}
```

Questions?