CHAPTER 2

# 포인터와 함수의 이해

# 학습 목표

- 함수의 인자로 배열 전달하기
- Call-by-value vs Call by reference
- 포인터 대상의 const 선언

# 함수의 인자로 배열 전달하기

# 2.1 인자전달의 기본방식은 값의 복사이다!

```
int SimpleFunc(int num) { . . . . }
int main(void)
{
     int age=17;
     SimpleFunc(age);
     . . . .
}
```

age에 저장된 값이 매개변수 num에 복사가 된다.

실제 전달되는 것은 age가 아니라 age에 저장된 값이다.

배열을 함수의 매개변수에 전달하는 이유는 함수 내에서 배열에 저장된 값을 참조하도록 하기 위함이다. 그런데 배열을 통째로 전달하지 않아도 이러한 일이 가능하다.

위의 코드에서 보이는 바와 같이, 배열을 함수의 인자로 전달하려면 배열을 통째로 복사할 수 있도록 배열이 매개변수로 선언되어야 한다. 그러나 C언어는 매개변수로 배열의 선언을 허용하지 않는다. 결론! 배열을 통째로 복사하는 방법은 C언어에 존재하지 않는다.

따라서 배열을 통째로 복사해서 전달하는 방식 대신에, 배열의 주소 값을 전달하는 방식을 대신 취한다.

# 2.2 배열을 함수의 인자로 전달하는 방식

```
int main(void)
{
     int arr[3]={1, 2, 3};
     int * ptr=arr;
     . . . .
}
```

배열의 이름은 int형 포인터!

배열의 이름은 int형 포인터! 따라서 int형 포인터 변수에 배열의 이름이 지니는 주소 값을 저장할 수 있다.

위의 예제를 통해서 다음과 같은 코드의 구성이 가능함을 유추할 수 있다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    SimpleFunc(arr);
    . . . .
}
```

배열이름 arr이 지니는 주소 값의 전달

```
void SimpleFunc(int * param)
{
    printf("%d %d", param[0], param[1]);
}
```

배열이름 arr은 int형 포인터이므로 매개변수는 int형 포인터 변수!

포인터 변수를 이용해서도 배열의 형태로 접근가능!

# 2.3 배열을 함수의 인자로 전달하는 예제

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}

int main(void)
{
    int arr1[3]={1, 2, 3};
    int arr2[5]={4, 5, 6, 7, 8};
    ShowArayElem(arr1, sizeof(arr1) / sizeof(int));
    ShowArayElem(arr2, sizeof(arr2) / sizeof(int));
    return 0;
}
```

실행결과

```
1 2 3
4 5 6 7 8
```

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}
void AddArayElem(int * param, int len, int add)
{
    int i;
    for(i=0; i<len; i++)
        param[i] += add;
}

int main(void)
{
    int arr[3]={1, 2, 3};
    AddArayElem(arr, sizeof(arr) / sizeof(int), 1);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));
    AddArayElem(arr, sizeof(arr) / sizeof(int), 2);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));
    AddArayElem(arr, sizeof(arr) / sizeof(int), 3);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));
    return 0;
}
```

실행결과

```
2 3 4
4 5 6
7 8 9
```

# 2.4 배열을 함수의 인자로 전달받는 함수의 또 다른 선언

```
void ShowArayElem (int * param, int len) { . . . . }
void AddArayElem (int * param, int len, int add) { . . . . }
```

동일한 선언

```
void ShowArayElem (int param[], int len) { . . . . }
void AddArayElem (int param[], int len, int add) { . . . . }
```

매개변수의 선언에서는 int * param과 int param[]이 동일한 선언이다. 따라서 배열을 인자로 전달받는 경우에는 int param[]이 더 의미있어 보이므로 주로 사용된다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    int * ptr=arr;    // int ptr[]=arr; 로 대체 불가능
    . . . .
}
```

하지만 그 이외의 영역에서는 int * ptr의 선언을 int ptr[]으로 대체할 수 없다.

# Call-by-value vs Call-by-reference

# 2.5 값을 전달하는 형태의 함수호출 : Call-by-value

함수를 호출할 때 단순히 값을 전달하는 형태의 함수호출을 가리켜 Call-by-value라 하고, 메모리의 접근에 사용되는 주소 값을 전달하는 형태의 함수호출을 가리켜 Call-by-reference라 한다. 즉, Call-by-value와 Call-by-reference를 구분하는 기준은 함수의 인자로 전달되는 대상에 있다.

```
void NoReturnType(int num)
{
    if(num<0)
        return;
    . . . .
}
```

*call-by-value*

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}
```

*call-by-reference*

call-by-value와 call-by-reference라는 용어를 기준으로 구분하는 것이 중요한 게 아니다.

중요한 것은 각 함수의 특징을 이해하고 적절한 형태의 함수를 정의하는 것이다.

call-by-value 형태의 함수에서는 함수 외부에 선언된 변수에 접근이 불가능하다. 그러나 call-by-reference 형태의 함수에서는 외부에 선언된 변수에 접근이 가능하다.

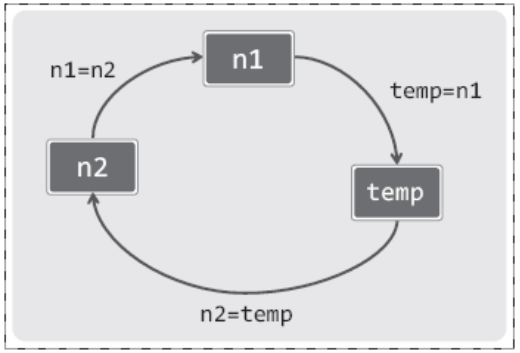
# 2.6 잘못 적용된 Call-by-value

```
void Swap(int n1, int n2)
{
    int temp=n1;
    n1=n2;
    n2=temp;
    printf("n1 n2: %d %d \n", n1, n2);
}

int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);

    Swap(num1, num2);   // num1과 num2에 저장된 값이 서로 바뀌길 기대!
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```

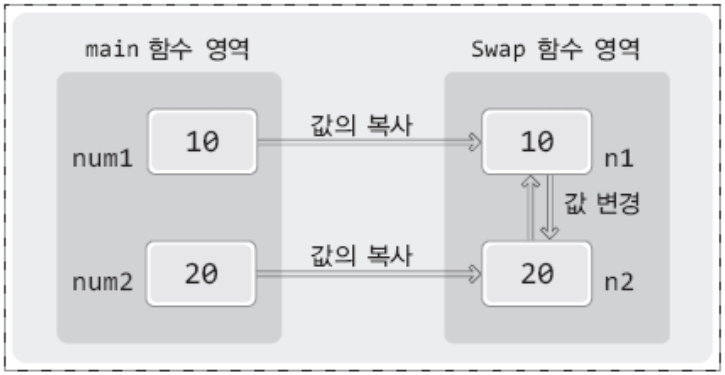
```
num1 num2: 10 20
n1 n2: 20 10
num1 num2: 10 20
```

실행결과

call-by-value가 적절치 않은 경우



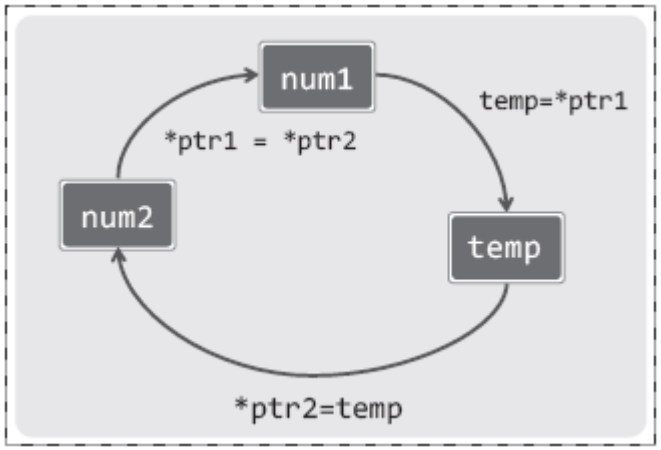

Swap 함수 내에서의 값의 교환




Swap 함수 내에서의 값의 교환은 외부에 영향을 주지 않는다.

# 2.7 주소 값을 전달하는 형태의 함수호출 : Call-by-reference

```
void Swap(int * ptr1, int * ptr2)
{
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);
    Swap(&num1, &num2);
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```




**Swap 함수 내에서 함수 외부에 있는 변수간 값의 교환**

```
num1 num2: 10 20
num1 num2: 20 10
```

실행결과

**Swap 함수 내에서의 *ptr1은 main 함수의 num1**
**Swap 함수 내에서의 *ptr2는 main 함수의 num2를 의미하게 된다.**

# 2.8 Scanf 함수호출 시 & 연산자를 붙이는 이유는?

```
int main(void)
{
    int num;
    scanf("%d", &num);
    . . . .
}
```

변수 num 앞에 & 연산자를 붙이는 이유는?

scanf 함수 내에서 외부에 선언된 변수 num에 접근 하기 위해서는 num의 주소 값을 알아야 한다. 그래서 scanf 함수는 변수의 주소 값을 요구한다.

```
int main(void)
{
    char str[30];
    scanf("%s", str);
    . . . .
}
```

배열 이름 str 앞에 & 연산자를 붙이지 않는 이유는?

str은 배열의 이름이고 그 자체가 주소 값이기 때문에 & 연산자를 붙이지 않는다. str을 전달함은 scanf 함수 내부로 배열 str의 주소 값을 전달하는 것이다.

# 포인터 대상의 const 선언

# 2.9 포인터 변수의 참조대상에 대한 const 선언

```
int main(void)
{
    int num=20;
    const int * ptr=&num;
    *ptr=30;    // 컴파일 에러!
    num=40;     // 컴파일 성공!
    . . . .
}
```

왼편의 const 선언이 갖는 의미

포인터 변수 ptr을 이용해서 ptr이 가리키는 변수에 저장된 값을 변경하는 것을 허용하지 않겠습니다!

그러나 변수 num에 저장된 값 자체의 변경이 불가능한 것은 아니다.
다만 ptr을 통한 변경을 허용하지 않을뿐이다.

# 2.10 포인터 변수의 상수화

```
int main(void)
{
    int num1=20;
    int num2=30;
    int * const ptr=&num1;
    ptr=&num2;      // 컴파일 에러!
    *ptr=40;        // 컴파일 성공!
    . . . .
}
```

왼편의 const 선언이 갖는 의미

포인터 변수 ptr에 저장된 값을 상수화 하겠다. 즉, ptr에 저장된 값은 변경이 불가능하다. ptr이 가리키는 대상의 변경을 허용하지 않는다.

```
const int * ptr=&num;

int * const ptr=&num;
```

```
const int * const ptr=&num;
```

두 가지 const 선언을 동시에 할 수 있다.

# 2.11 const 선언이 갖는 의미

```
int main(void)
{
     double PI=3.1415;
     double rad;
     PI=3.07;   // 실수로 잘못 삽입된 문장, 컴파일 시 발견 안됨
     scanf("%lf", &rad);
     printf("circle area %f \n", rad*rad*PI);
     return 0;
}
```

안전성이 높아진 코드

```
int main(void)
{
     const double PI=3.1415;
     double rad;
     PI=3.07;   // 컴파일 시 발견되는 오류상황
     scanf("%lf", &rad);
     printf("circle area %f \n", rad*rad*PI);
     return 0;
}
```

**const 선언은 추가적인 기능을 제공하기 위한 것이 아니라, 코드의 안전성을 높이기 위한 것이다. 따라서 이러한 const의 선언을 소홀히하기 쉬운데, const의 선언과 같이 코드의 안전성을 높이는 선언은 가치가 매우 높은 선언이다.**

# 질문 및 정리