Chapter 03 - 포인터와 메모리 구조

생성일	@2025년 9월 11일 오전 9:37
태그	

01. 변수와 메모리 구조

```
#include <iostream>

int main()
{
        char char_value = 'A';
        int int_value = 123;
        double double_value = 123.456;

        return 0;
}
```

- 64 bit 운영체제에서 char, int, double 형은 각각 1byte, 4byte, 8byte의 크기를 차지함
- 변수를 선언하면 자료형의 크기에 맞게 공간을 확보한 후, 해당 공간에 데이터를 기록함
    - 확보하는 실제 공간의 주소는 메모리 상황에 따라 달라질 수 있음

포인터 연산자

- 포인터 : 메모리 주소를 저장하는 변수
    - 일반 변수를 지정할 때처럼 int나 char과 같은 자료형을 지정함
    - 다만. 다음처럼 자료형과 변수 이름 사이에 별표(*)를 추가함.

```
자료형 *(포인터_변수_이름)
```

포인터 변수 선언과 주소 대입하기

```
#include <iostream>

int main()
{
        char char_value = 'A';
        int int_value = 123;
        double double_value = 123.456;

        char *char_pointer_value = &char_value;
        int *int_pointer_value = &int_value;
        double *double_pointer_value = &double_value;

        return 0;
}
```

- 일반 변수 앞에 붙은 & ← 피연산자의 주소를 불러오는 주소 연산자

포인터 변수의 크기

- 포인터 변수의 크기는 데이터 형식과 관련이 없음

  → 모든 포인터 변수의 크기는 같음.

  → 포인터는 데이터 형식이나 변수의 크기와는 관련이 없음.
- 그럼에도 포인터 변수를 선언할 때 데이터 형식을 지정하는 이유

  → 해당 포인터가 가리키는 데이터의 형식을 명시하기 위함

```
int *ptr;  // int형 데이터를 가리키는 포인터
double *ptr2;  // double형 데이터를 가리키는 포인터
```

- 포인터 선언문에 지정한 데이터 형식으로, 해당 포인터가 가리키는 데이터의 크기와 해석 방법이 결정됨

  → 포인터를 대상으로 연산할 때 필요함

포인터 변수가 가리키는 데이터에 접근하기

- 포인터 변수에 ***역참조 연산자 **를 사용하면 해당 포인터 변수에 저장된 주소가 가리키는 데이터에 접근할 수 있음

```
#include <iostream>
using namespace std;

int main()
{
        char char_value = 'A';
        int int_value = 123;
        double double_vaue = 123.456;

        char *char_pointer_value = &char_value;
        int *int_pointer_value = &int_value;
        double *double_pointer_value = &double_value;

        // 일반 변수의 데이터 출력
        cout << "char_value: " << char_value << endl;
        cout << "int_value: " << int_value << endl;
        cout << "double_value: " << double_value << endl;
        cout << endl;

        // 역참조 연산자로 포인터 변수가 가리키는 데이터 출력
        cout << "*char_pointer_value: " << *char_pointer_va
lue << endl;
        cout << "*int_pointer_value: " << *int_pointer_valu
e << endl;
        cout << "*double_pointer_value: " << *double_pointe
r_value << endl;
        cout << endl;

        // 역참조 연산자로 원본 데이터 덮어쓰기
        *char_pointer_value = 'Z';
        *int_pointer_value = 321;
        *double_pointer_value = 654.321;

        // 일반 변수의 데이터 출력(업데이트 확인)
```

```
        cout << "char_value: " << char_value << endl;
        cout << "int_value: " << int_value << endl;
        cout << "double_value: " << double_value << endl;

        return 0;
}
```

실행 결과

```
char_value: A
int_value: 123
double_value: 123.456

*char_pointer_value: A
*int_pointer_value: 123
*double_pointer_value:123.456

char_value: Z
int_value: 321
double_value: 654.321
```

- 포인터 변수 앞에 역참조 연산자 *를 사용해, 포인터 변수가 가리키는 데이터에 직접 접근하는 것을 확인할 수 있음
- *char_pointer_value = 'z' 처럼 새로운 값을 넣으면 char_value 변수의 값이 'z'로 바뀌는 것도 볼 수 있음.

다중 포인터

- 포인터가 메모리 주소를 저장하는 '변수'라는 것은, 포인터가 차지하는 공간도 주소를 가지고 있음을 의미함

  → 포인터 변수의 주소를 저장하는 또 다른 포인터도 만들 수 있음.
- **다중 포인터** : 포인터를 가리키는 포인터

```
#include <iostream>

int main()
{
        int int_value = 123;

        int *int_pt_value = &int_value;
        int **int_pt_value = &int_pt_value;
        int ***int_pt_pt_pt_value = &int_pt_pt_value;

        return 0;
}
```

- 삼중 포인터 변수는 ***역참조 연산자 *를 3번, 이중 포인터 변수는 2번, 단일 포인터 변수는 1번 적용***해야 값에 접근할 수 있음

배열과 포인터

- 배열을 이용하면 간단하게 여러 값을 넣을 수 있음

```
자료형 배열_이름[크기] = { 값1, 값2, 값3, ..., 값n }
```

- **배열_이름[인덱스]** : [ ] 안에 인덱스라고 하는 차례 번호를 넣으면 해당 순서의 원소에 접근할 수 있음.
- 배열의 인덱스는 항상 0부터 시작함
- 배열에서 가장 첫 번째 원소에 접근하려면 → 배열_이름[0]
- 가장 마지막 원소에 접근하려면 → 배열_이름[n-1]

배열 선언과 원소에 접근하기

```
#include <iostream>
using namespace std;

int main()
```

```
{
        int lotto[45] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1
1, 12, 13, 14, 15
                          16, 17, 18, 19, 20, 21, 22, 23, 2
4, 25, 26, 27, 28, 29, 30,
                          31, 32, 33, 34, 35, 36, 37, 38, 3
9, 40, 41, 42, 43, 44, 45 };


         cout << "lotto[0] Address: " << &lotto[0] << endl;
         cout << "lotto[1] Address: " << &lotto[1] << endl;
         cout << "lotto[2] Address: " << &lotto[2] << endl;
         cout << "lotto[3] AddressL " << &lotto[3] << endl;
         cout << "lotto[4] Address: " << &lotto[4] << endl;
         cout << "lotto[5] AddressL " << &lotto[5] << endl;

         return 0;
}
```

실행 결과

```

```

포인터 연산으로 배열의 원소에 접근하기

- 포인터 연산으로 각 원소에 접근할 수 있음 (like 배열의 인덱스)

```
#include <iostream>
using namespace std;

int main()
{
        int lotto[45] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1
1, 12, 13, 14, 15
                          16, 17, 18, 19, 20, 21, 22, 23, 2
4, 25, 26, 27, 28, 29, 30,
                          31, 32, 33, 34, 35, 36, 37, 38, 3
```

```
9, 40, 41, 42, 43, 44, 45 };


        cout << "lotto[0] Address: " << &lotto[0] << endl;
        cout << "lotto[1] Address: " << &lotto[1] << endl;
        cout << "lotto[2] Address: " << &lotto[2] << endl;
        cout << "lotto[3] AddressL " << &lotto[3] << endl;
        cout << "lotto[4] Address: " << &lotto[4] << endl;
        cout << "lotto[5] AddressL " << &lotto[5] << endl;

        cout << "(lotto + 0) Address: " << lotto + 0 << en
dl;
        cout << "(lotto + 1) Address: " << lotto + 1 << en
dl;
        cout << "(lotto + 2) Address: " << lotto + 2 << en
dl;
        cout << "(lotto + 3) AddressL " << lotto + 3 << en
dl;
        cout << "(lotto + 4) Address: " << lotto + 4 << en
dl;
        cout << "(lotto + 5) AddressL " << lotto + 5 << en
dl;

        return 0;
}
```

실행 결과

→ 인덱스로 접근하나 포인터로 접근하나 ***결과는 똑같다.***

이러한 특징이 성립하는 이유

- 배열의 이름인 lotto가 첫 번째 원소의 주소 &lotto[0]를 가리키기 때문
- 포인터 연산에서 덧셈은 자료형의 크기를 곱한 만큼 덧셈을 수행함

배열과 포인터는 같을까?

→ 엄연히 다르다.

```
#include <iosteam>
using namespace std;

int main()
{
        int array[5] = { 1, 2, 3, 4, 5 };
        int *pointer_array = array;

        cout << "array: " << array << endl;
        cout << "pointer_array: " << pointer_array << endl
<< endl;

        cout << "sizeof(array): " << sizeof(array) << endl;
        cout << "sizeof(pointer_array): " << sizeof(pointer
_array) << endl;

        return 0;
}
```

- array와 pointer_array가 같은 주소를 가리키고 있지만, sizeof로 크기를 비교해보면 완전히 다르다.

  → array는 int[5] 형식이고, pointer_array는 int*형이기 때문이다.

  → array의 sizeof는 배열 전체 크기인 20byte, pointer_array는 포인터 변수의 크기인 8byte가 출력.
- 이처럼 배열과 포인터는 다르지만, 배열의 이름을 사용할 때 자동으로 첫 번째 원소를 가리키는 포인터가 되므로 마치 포인터와 배열이 같다고 생각할 수 있음.
- 배열과 포인터 관계의 핵심
  - **배열의 원소에 접근할 때 포인터 연산으로도 가능**하다는 것

```
#include <iostream>
using namespace std;

int main()
```

```
{
        int lotto[45] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1
1, 12, 13, 14, 15,
                                                16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
                                                31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45 };

        cout << "Today's lotto : "
                 << *lotto << ", " << *(lotto + 7) << ", "
<< *(lotto + 15) << ", "
                 << *(lotto + 27) << ", " << *(lotto + 36)
<< ", " << *(lotto + 44)
                 << endl;

        return 0;
}
```

실행 결과

```
Today's lotto : 1, 8, 16, 28, 37, 45
```

동적 메모리 할당

```
char char_arrray[10];
int int_array[500];
float float_array[1000];
```

- 배열의 크기가 고정되면 더 많은 원소가 필요할 때는 처리할 수 없고, 반대로 너무 큰 배열을 선언하면 메모리가 낭비되거나 프로그램이 강제 종료 될 수 있음.

동적 메모리 할당과 해제하기

- 동적 메모리 할당 : 프로그램 실행 중에도 필요한 크기의 메모리를 운영체제에 요청하여 사용할 수 있는 방법
- C++ 언어에서 동적 메모리를 할당하려면 new 키워드 사용

```
자료형 *변수_이름 = new 자료형;
```

- new로 할당된 메모리는 필요 없는 시점에 delete 키워드로 반드시 직접 해제해야 함.

```
delete 변수_이름;
```

```
#include <iostream>
using namespace std;

int main()
{
    int *pt_int_value = new int;    // 동적 메모리 할당
    *pt_int_value = 100;
    cout << *pt_int_value << endl;

    delete pt_int_value;    // 동적 메모리 해제

    return 0;
}
```

실행 결과

```
100
```

```
자료형 *변수_이름 = new 자료형[크기];
```

```
delete[ ] 변수_이름;
```

```
#include <iostream>
using namespace std;

int main()
{
    int* pt_int_array_value = new int[5];   // 동적 메모리 할당(배열)

    for (int i = 0; i < 5; i++)
    {
        pt_int_array_value[i] = i;  // 할당된 배열 변수에 0~4까지 순서대로 넣기
    }

    for (int i = 0; i < 5; i++)
    {
        cout << pt_int_array_value[i] << endl;
    }

    delete[] pt_int_array_value;    // 동적 메모리 해제(배열)

    return 0;
}
```

실행 결과

```
0
1
2
3
4
```

- **주의점**
  - 배열 형태로 메모리를 할당했다면, 똑같이 배열 형태로 메모리를 해제해야 함.

→ 그렇지 않으면 해제되지 않은 메모리 때문에 ***메모리 누수***가 발생함

동적 할당 메모리를 해제하는 이유

- 각각 사용하는 메모리의 영역이 다르기 때문
- 대부분의 일반 변수는 **스택**이라는 메모리 영역에 할당됨

  → 함수의 호출과 함께 할당되며, 함수가 반환 되면 자동으로 소멸함 (메모리 해제를 관리 필요 X)

  → 그러나, 스택 영역은 크기가 한정되어 있으며, 이 크기를 초과할 때는 운영체제가 해당 프로그램을 강제로 종료함

- 동적으로 할당된 변수는 **힙**이라는 메모리 영역에 존재함

  → 힙은 스택보다 훨씬 큰 메모리 풀이므로, 크기가 큰 메모리도 충분히 할당할 수 있음

  → 힙에 할당된 메모리는 명시적으로 해제하기 전에는 프로그램이 종료될 때까지 계속 유지됨

  → 메모리를 적절하게 해제하지 않으면 프로그램이 종료하면서 조금씩 메모리가 누수됨

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int customer_num = 0;

    cout << "오늘 방문 손님: ";
    cin >> customer_num;    // 손님 수 입력

    string* bread = new string[customer_num];   // 손님 수만큼 string 배열 생성

    for (int i = 0; i < customer_num; i++) // 입력받은 손님 수만큼 반복
    {
```

```
        bread[i] = "빵_" + to_string(i); // '빵_숫자' 형태로
배열에 저장
    }

    cout << endl << "--생산된 빵--" << endl;
    for (int i = 0; i < customer_num; i++)
    {
        cout << *(bread + i) << endl;   // 생산된 빵 출력(포
인터 연산 참고)
    }

    delete[] bread; // string 배열 삭제

    return 0;
}
```

실행 결과

```
오늘 방문 손님: 3

--생산된 빵--
빵_0
빵_1
빵_2
```

- **to_string(i)** → 숫자를 문자열로 변환하는 함수

포인터를 다룰 때 주의할 점

- 포인터를 역참조하기 전에 포인터가 유효한 메모리를 가리키는지 확인해야 함.

  → 포인터를 선언한다고 해서 자동으로 유효한 메모리 주소를 가리키는 건 아님
- 할당된 메모리의 범위를 벗어나는 포인터 연산은 피해야 함.
- 할당 해제 된 메모리를 역참조하지 말아야 함
-

02. 함수와 구조체

함수 만들기

- 함수는 '특정 작업을 수행하는 코드 집합'으로 정의할 수 있음

```
int func(int _arg1, int _arg2)
{
}
```

- 함수 선언의 필수 요소 4가지
  1. **반환 형식**
  2. **함수 이름**
  3. **매개 변수**
  4. **함수 몸체**

```
#include <iostream>
using namespace std;

int add(int _x, int _y) // add 함수 정의
{
    int result = _x + _y;
    return result;
}

int main()
{
    int add_result = add(2, 3);     // add 함수 호출
    cout << "add 함수 결과: " << add_result << endl;    //
함수 실행 결과

    return 0;
}
```

실행 결과

```
add 함수 결과: 5
```

함수의 매개변수 사용하기

```
#include <iostream>
using namespace std;

void change_negative(int _val)
{
    if (_val > 0)
    {
        _val = -_val;
    }
}

int main()
{
    int a = 3, b = -3;

    cout << "a : " << a << endl;
    cout << "b : " << b << endl;

    change_negative(a);
    change_negative(b);

    cout << "change_negative(a) : " << a << endl;
    cout << "change_negative(b) : " << b << endl;

    return 0;
}
```

실행 결과

```
a : 3
b : -3
change_negative(a) : 3
change_negative(b) : -3
```

- 함수를 호출할 때 전달하는 것 → 인자, 전달 받는 것 → 매개변수
- a의 값이 3으로 그대로 ⇒ 매개변수인 _val이 지역 변수이기 때문

포인터를 매개변수로 사용하기

```
#include <iostream>
using namespace std;

void change_negative(int *_val)
{
    if (*_val > 0)
    {
        *_val = -(*_val);
    }
}

int main()
{
    int a = 3, b = -3;

    cout << "a : " << a << endl;
    cout << "b : " << b << endl;

    change_negative(&a);
    change_negative(&b);

    cout << "change_negative(a) : " << a << endl;
    cout << "change_negative(b) : " << b << endl;
```

```
    return 0;
}
```

실행 결과

```
a : 3
b : -3
change_negative(a) : -3
change_negative(b) : -3
```

- 인자로 넘기는 a, b 변수 앞에 **주소 연산자 &**가 붙음

  → 변수 앞에 주소 연산자 &를 붙이면, 해당 변수의 주소를 불러옴
- ***역참조 연산자 *를 이용하면, 포인터 변수가 가리키는 데이터에 직접 접근할 수 있음***

배열을 매개 변수로 사용하기

```
#include <iostream>
using namespace std;

int average(int _array[], int _count)
{
    int sum = 0;
    for (int i = 0; i < _count; i++)
    {
        sum += _array[i];
    }
    return (sum / _count);
}ruct

int main()
{
    int score[5] = { 90, 75, 80, 100, 65 };
    cout << "평균 점수: " << average(score, 5) << endl;
```

```
    return 0;
}
```

실행 결과

```
평균 점수: 82
```

- 배열로 매개변수를 사용하면 포인터와 마찬가지로 실제로는 주소 값을 전달 받음

구조체 만들기

- 구조체 : 여러 형식의 데이터들을 묶어서 관리하는 방법

```
struct Person
{
        std::string name; // 이름
        int age;  // 나이
        float height;  // 키
        float weight;  // 몸무게
}
```

- Person 구조체는 자료형과 마찬가지로 형식만 정의된 형태
- 구조체는 하나 이상의 변수를 묶어 새로운 자료형으로 정의할 수 있음

```
Person adult;
```

```
Person adult;
adult.name = "Brain";
adult.age = 24;
adult.height = 180;
adult.weight = 70;
```

- 구조체 안의 개별적인 멤버에 접근하려면, 멤버 선택 연산자인 점(.)을 이용해야 함.

```
Person adult[3];

adult[0].name = "Brain";
adult[0].age = 24;
adult[0].weight = 180;
adult[0].height = 70;

adult[1].name = "Jessica";
adult[1].age = 22;
adult[1].weight = 165;
adult[1].height = 55;

adult[2].name = "James";
adult[2].age = 30;
adult[2].weight = 170;
adult[2].height = 65;
```

- 구조체 형식으로도 배열을 선언할 수 있음
- **adult[인덱스]** 처럼 인덱스로 각각의 Person을 구분하고, 각 Person의 정보가 담긴 멤버 변수는 .으로 접근할 수 있음.

구조체를 매개변수로 사용하기

```
#include <iostream>
using namespace std;

struct Person
{
    std::string name;    // 이름
    int age;    // 나이
    float height;   // 키
    float weight;   // 몸무게
};
```

```
void check_age(Person *_adult, int _count)
{
    for (int i = 0; i < _count; i++)
    {
        if (_adult[i].age >= 25)
        {
            cout << "name: " << _adult[i].name << endl;
            cout << "age: " << _adult[i].age << endl;
            cout << "height: " << _adult[i].height << endl;
            cout << "weight: " << _adult[i].weight << endl;
        }
    }
}

int main()
{
    Person adult[3] =
    {
        {"Brain", 24, 180, 70},
        {"Jessica", 22, 165, 55},
        { "Jessica", 30, 170, 65 },
    };
    check_age(adult, 3);

    return 0;
}
```

실행 결과

```
name: Jessica
age: 30
height: 170
weight: 65
```

- 구조체를 사용하면 **관련 값을 하나의 객체로 그룹화**하므로 코드를 더 읽기 쉽고 유지, 관리하기 좋게 만들 수 있음.
- 함수에 전달할 인자가 많을 때 특히 유용함.

- 그런데 구조체를 함수에 전달하면 ***복사본이 전달***되므로 구조체가 매우 크면 성능 문제가 발생할 수 있음

  → 복사본 대신 ***구조체에 대한 포인터(주소)를 전달***하면 해결할 수 있음.
- 앞의 예시에서는 구조체 배열의 시작 주소를 전달함.

구조체 초기화하기

```
Person adult = {"Brain", 24, 180, 70};
```

- 구조체 배열일 때는 중괄호 단위로 구분하여 반복해서 나열 해주면 됨.

```
Person adult[3] =
{
        { "Brain", 24, 180, 70 },
        { "Jessica", 22, 165, 55 },
        { "James", 30, 170, 65 },
};
```

03. 정적 변수와 상수 변수

정적 변수 선언하기 - static

- 지역 변수 : 선언된 지점에서 생성되고, 해당 블록이 끝나면 소멸함

  → 자동 지속
- 전역 변수 : 전역 범위에 생성된 변수, 해당 파일 전체에서 효력이 있음.

- 지역 변수에 static 키워드를 사용하면, 자동 지속에서 정적 지속으로 변수의 유효 범위가 바뀐다.

  → **static** 키워드는 **지역 변수를 정적 변수**로 바꾼다.

```
#include <iostream>
using namespace std;

void func()
{
    int a = 10;
    static int b = 10;

    a++;
    b++;

    cout << "a : " << a << " , b : " << b << endl;
}
int main()
{
    func();
    func();
    func();
    func();
    func();

    return 0;
}
```

실행 결과

```
a : 11 , b : 11
a : 11 , b : 12
a : 11 , b : 13
a : 11 , b : 14
a : 11 , b : 15
```

- 변수 a, b를 똑같이 1씩 증가 시켰는데 a는 항상 같은 값이고 b는 증가함

→ b는 static 키워드 때문에 정적 변수로 선언, 함수가 종료 되어도 사라지지 않음.

```
#include <iostream>
using namespace std;

int getNewID()
{
    static int ID = 0;
    return ++ID;
}

int main()
{
    cout << "ID: " << getNewID() << endl;
    cout << "ID: " << getNewID() << endl;
    cout << "ID: " << getNewID() << endl;
    cout << "ID: " << getNewID() << endl;
    cout << "ID: " << getNewID() << endl;

    return 0;
}
```

실행 결과

```
ID: 1
ID: 2
ID: 3
ID: 4
ID: 5
```

정적 변수와 수명 주기가 지역 변수와 다른 이유

- 지역 변수와 정적 변수는 메모리의 위치가 다름
- 지역 변수는 스택 (stack) 영역에 저장됨

    - 스택 영역의 변수는 함수가 호출될 때 메모리에 할당되며, 종료될 때 메모리에서 해제됨
- Static으로 선언된 정적 변수는 데이터 (data) 영역에 저장됨
- 지역 변수와 정적 변수는 할당되는 메모리 영역이 달라 수명 주기가 다름.

상수 변수 선언하기 - const

- 상수 : 변하지 않는 값
- 변수에 const 키워드를 사용하면 값을 변경할 수 없게 됨

```
const int a = 1;
int const b = 1;
```

- const 변수를 사용할 때는 반드시 초기화 해야함
- 초기화한 후 새로운 값을 넣으려고 해도 컴파일 오류 발생

```
#include <iostream>
using namespace std;

int main()
{
    const int a = 1;
    a = 2; // 컴파일 오류 발생

    return 0;
}
```

- 오류 발생

포인터 변수의 상수화

```
#include <iosteam>
using namespace std;
```

```
int main()
{
    int a = 0;
    const int *ptr = &a  // *ptr을 상수화

    a = 1;  // 컴파일 통과
    *ptr = 2;  // 컴파일 오류 발생

    return 0;
}
```

- *ptr이 상수로 지정되었으므로, 컴파일 오류 발생

```
#include <iosteam>
using namespace std;

int main()
{
    int a = 0;
    ibt b = 1;
    int *const ptr = &a  // *ptr을 상수화

    a = 1;  // 컴파일 통과
    ptr = &b;  // 컴파일 오류 발생

    return 0;
}
```

- 포인터 변수는 주소를 저장하는 변수
  - 변수 자체가 상수화 되어 다른 변수인 b의 주소로 변경할 수 없게 됨
  - ptr 변수는 오직 a 변수의 주소만 가지게 됨

  → **const는 어느 곳에 붙던 붙으면 해당 값은 변경할 수 없음.**

포인터 변수와 const 위치	요약
const int *ptr = &a	*ptr 상수화 = 포인터 변수가 가리키는 값을 상수화
int *const ptr = &a	ptr 상수화 = 포인터 변수 자체를 상수화

04. 레퍼런스 변수

레퍼런스 사용하기

- 일반 변수 : 값을 저장하는 변수
- 포인터 변수 : 메모리 주소를 저장하는 변수
- 레퍼런스 : 변수에 또 다른 이름, 별칭을 부여

```
#include <iostream>
using namespace std;

void swap(int a, int b)
{
    // swap 함수 내 교환 전 a, b 값
    cout << "[swap func] before swap, a : " << a << " b : "
<< b << endl;

    int temp = a;
    a = b;
    b = temp;

    // swap 함수 내 교환 후 a, b값
    cout << "[swap func] after swap, a : " << a << " b : "
<< b << endl;
}

int main()
{
    int a = 5;
```

```
    int b = 10;

    // swap 함수 호출 전 a, b 값
    cout << "[main] before swap, a : " << a << " b : " << b
<< endl << endl;

    swap(a, b);

    // swap 함수 호출 후 a, b값
    cout << endl << "[main] after swap, a : " << " b : " <<
b << endl;

    return 0;
}
```

실행 결과

```
[main] before swap, a : 5 b : 10

[swap func] before swap, a : 5 b : 10
[swap func] after swap, a : 10 b : 5

[main] after swap, a :  b : 10
```

- 값이 바뀌지 않는 이유 : main의 a, b와 swap 함수의 a, b 변수가 서로 다른 범위에 있기 때문
- 즉, 이름만 같을 뿐 메모리에서 서로 다른 공간에 저장되어 있음

  → **값에 의한 호출**
- 레퍼런스 변수를 이용하면 해당 문제를 간단하게 해결할 수 있음

  → 변수 이름 앞에 **& 기호**로 선언

```
자료형 &레퍼런스_변수_이름 = 대상_변수_이름;
```

- **참조에 의한 호출** : 매개변수가 참조자가 되므로 피호출자의 변수를 그대로 이용함

레퍼런스를 사용할 때 지켜야 할 점

1. **레퍼런스 변수는 선언한 후 반드시 참조할 원본 변수를 지정해야 함**
2. **참조할 대상이 지정된 레퍼런스 변수는 다른 변수를 참조하도록 변경할 수 없다.**
3. **레퍼런스 변수는 상수를 참조할 수 없다.**

세 가지 함수 호출 방식

1. **값에 의한 호출**
2. **참조에 의한 호출**
3. **주소에 의한 호출**