

## 02. 포인터와 레퍼런스

2.1 포인터

2.2 레퍼런스(참조)

2.3 함수에서의 인자 전달

## 2.1 포인터

# 변수의 주소

## ■ 포인터(pointer)

- 변수의 주소를 나타냄

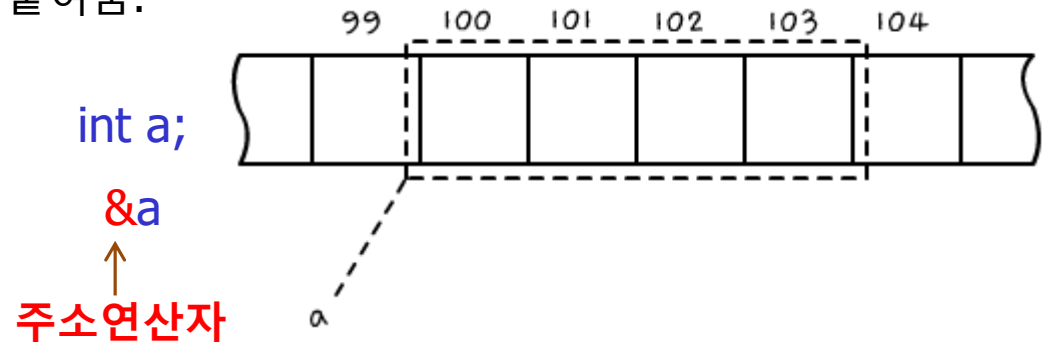
## ■ 변수의 주소

- 변수가 포함하고 있는 제일 첫 번째 바이트의 주소
- 변수의 주소는 변수 이름 앞에 &를 붙여줌.

즉 a의 주소 ⇔ &a

## ■ 포인터 변수

- 주소값을 저장하는 변수
- 포인터라고 약칭



# 포인터 변수의 선언 및 사용

## ■ 간접 참조 연산자 \*

- 포인터 변수가 가리키는 실제 변수의 값을 참조

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

int main() {
    int n=10, m;
    char c='A';
    double d;

    int *p= &n; // p는 n의 주소값을 가짐
    char *q = &c; // q는 c의 주소값을 가짐
    double *r = &d; // r은 d의 주소값을 가짐

    *p = 25; // n에 25가 저장됨
    *q = 'A'; // c에 문자 'A'가 저장됨
    *r = 3.14; // d에 3.14가 저장됨
    m = *p + 10; // p가 가리키는 값(n 변수값)+10을 m에 저장

    cout << n << ' ' << *p << "\n"; // 둘 다 25가 출력됨
    cout << c << ' ' << *q << "\n"; // 둘 다 'A'가 출력됨
    cout << d << ' ' << *r << "\n"; // 둘 다 3.14가 출력됨
    cout << m << "\n"; // m 값 35 출력
}
```

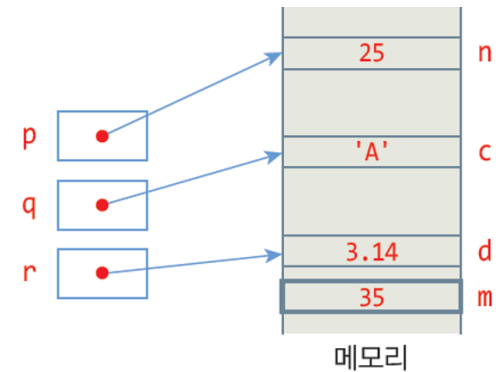
int m;

\*p = 25;

\*q = 'A';

\*r = 3.14;

m = \*p + 10;



# 포인터 기본 개념 리뷰

## ■ 문제)

- 아래 그림과 같이 int형으로 arr이란 배열이 이미 선언되고 초기화 되어져 있다고 가정하자.

주소	0	4	8	12	16	18
	1	2	3	4	5	6

<배열 arr의 메모리 상 표현>

- 다음 페이지의 소스코드를 보고, 빈칸에 알맞은 코드를 넣은 후, 프로그램을 완성하여 그 결과를 예측하여라.
- 프로그램을 실행시켜 예측한 결과와 비교하여라.
- 완성된 프로그램의 실행결과와 본인이 예측한 결과와 동일한지, 그렇지 않다면 그 이유가 무엇이었는지 생각해본다.

# 포인터 기본 개념 리뷰

```
int arr[6] = {1,2,3,4,5,6};
int _____; // 1) int에 대한 포인터 변수 chr_ptr을 선언하는 문장 작성

// chr_ptr이 arr 배열이 저장되어 있는 메모리 주소값을 갖도록 초기화.
chr_ptr = _____;

chr_ptr++; // chr_ptr의 값을 하나 증가

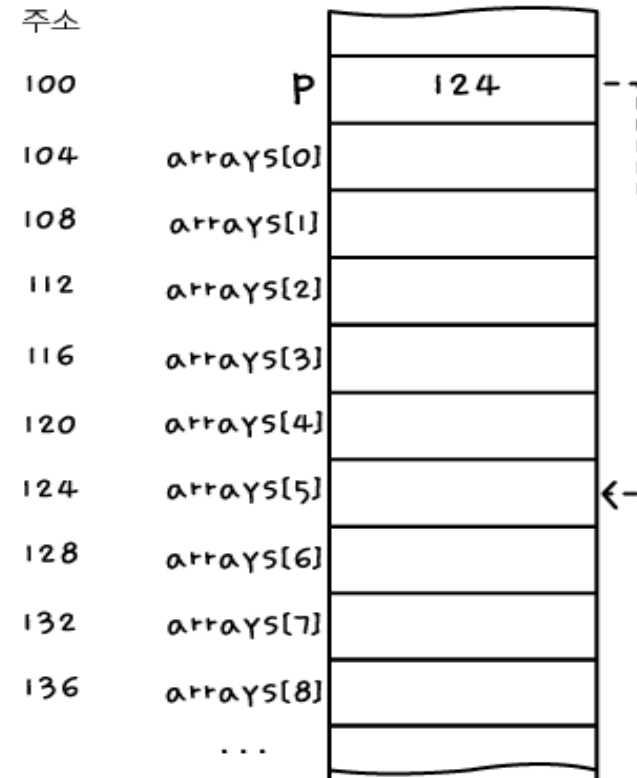
cout << chr_ptr << "\n"; // 3) 옆 문장이 실행되었을 때의 결과는?
cout << *chr_ptr << "\n"; // 4) 옆 문장이 실행되었을 때의 결과는?
cout << arr << "\n"; // 5) 옆 문장이 실행되었을 때의 결과는?
cout << arr+4 << "\n"; // 6) 옆 문장이 실행되었을 때의 결과는?
cout << &arr[3] << "\n"; // 7) 옆 문장이 실행되었을 때의 결과는?
cout << arr[4] << "\n";

// arr[3]의 값을 chr_ptr을 이용하여 프린트
cout << _____ << endl;
```

# 배열의 원소를 가리키는 포인터

- 포인터 변수를 사용해서 배열의 원소를 가리키는 예

```
int arrays[10];  
int* p = &arrays[5];
```



# 포인터를 사용한 원소의 탐색

## ■ 일반적인 방법을 사용한 원소의 탐색

```
int nArray[10];  
  
// 배열을 탐색하면서 값을 넣는다.  
for (int i = 0; i < 10; ++i)  
    nArray[i] = i;
```

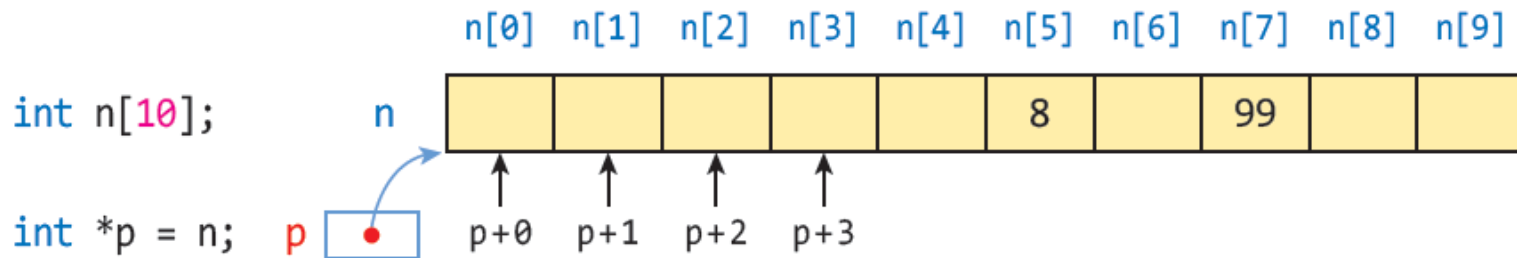
## ■ 포인터를 사용한 원소의 탐색

```
int nArray[10];  
int* p = &nArray[0];  
  
// 배열을 탐색하면서 값을 넣는다.  
for (int i = 0; i < 10; ++i)  
    *(p + i) = i;
```



# 배열과 포인터

- 배열 이름은 배열 메모리의 시작 주소로 다름



n[5] = 8	→ 배열 n의 시작 위치에서 5만큼 떨어진 주소에 8 기록
n + 5	→ n[5]의 주소
*(n + 5) = 8;	→ n[5]에 8기록
p = p + 7;	→ p는 n[7]의 주소
*p = 99;	→ n[7]에 99 기록

# 1차원 배열의 이름

## ■ 1차원 배열의 이름

- 1차원 배열의 첫 번째 원소의 주소값
- 포인터 상수(constant) : 바뀔 수 없는 값임

```
int nArray[10];  
if (nArray == &nArray[0])  
{  
    // 항상 이곳이 실행  
}  
nArray = &nArray[1]; // Fail  
nArray++; // Fail
```

## ■ 원소의 주소

- 주소 연산자 사용      **&nArray[2]**
- 배열이름 사용        **nArray+2 == &nArray[2]**

# 1차원 배열의 이름을 사용해서 배열 탐색

- 주소를 이용한 원소 참조

```
int nArray[10];
```

```
*(nArray + i) == nArray[i]
```

- 1차원 배열의 이름을 사용해서 배열을 탐색하는 예

```
int nArray[10];
```

```
// 배열을 탐색하면서 값을 넣는다.
```

```
for (int i = 0; i < 10; ++i)
```

```
    *(nArray + i) = i; // nArray[i] = i과 동일
```

# 2차원 배열과 포인터

## ■ 2차원 배열의 이름

- 배열의 시작 주소

**M = &M[0]          M+1 == &M[1]          // 주소**

**\*M == M[0]      \*(M+1) == &M[1][0] // 주소**

**\*\* (M    ) == M[0][0] // 값**

**\*\* (M+1) == M[1][0] // 값**

**int M[3][3];**

M[0][0]	M[0][1]	M[0][2]
M[1][0]	M[1][1]	M[1][2]
M[2][0]	M[2][1]	M[2][2]

## ■ 부분 배열의 이름

- 부분 배열의 시작 주소

**M[0]은 1행의 시작주소 , M[0] == \*M          // 주소**

**M[1]은 2행의 시작주소, M[1] == \*(M +1) // 주소**

- 포인터 자료형

**\*(M[1]    ) == M[1][0] // 값**

**\*(M[1]+1) == M[1][1] // 값**

**\*(M[1]+2) == M[1][2] // 값**

# 포인터 배열

- 포인터 배열은 포인터변수들을 배열요소로 갖는 배열이다.

```
char* ptr_ary[5];
```

```
char a='A', b='B', c='C', d='D', e='E';
```

```
ptr_ary[0] = &a;
```

```
ptr_ary[1] = &b;
```

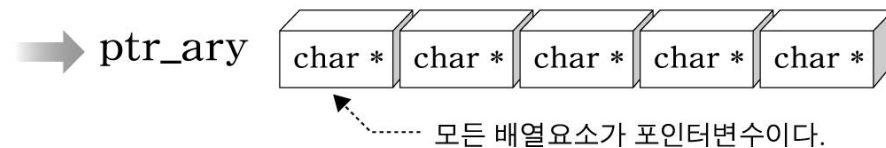
```
ptr_ary[2] = &c;
```

```
ptr_ary[3] = &d;
```

```
ptr_ary[4] = &e;
```

char \* ptr\_ary[5];

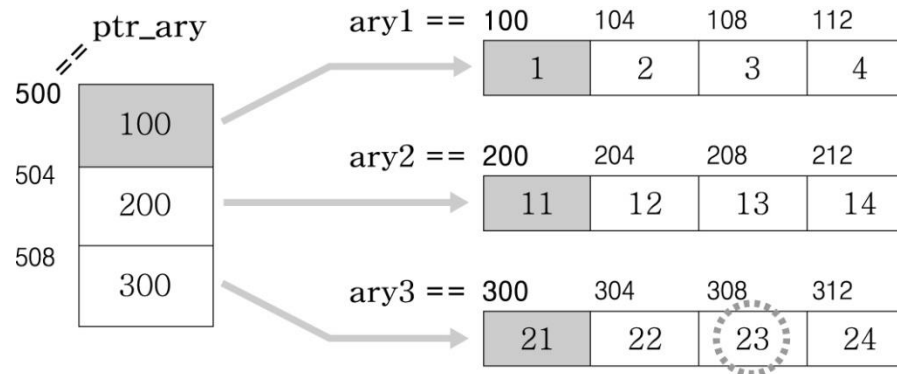
배열요소의 자료형      배열명      배열요소의 개수



## 2차원 배열과 포인터 배열

- 1차원 배열의 배열명을 포인터배열에 저장하면 포인터배열을 2차원 배열처럼 사용할 수 있다.

```
int ary1[4]={1, 2, 3, 4};  
int ary2[4]={11, 12, 13, 14};  
int ary3[4]={21, 22, 23, 24};  
int *ptr_ary[3]={ary1, ary2, ary3}; // 각 배열명을 포인터배열에 초기화
```



(각 기억공간의 주소값은 설명의 편의를 위해 임의로 붙인 것입니다.)

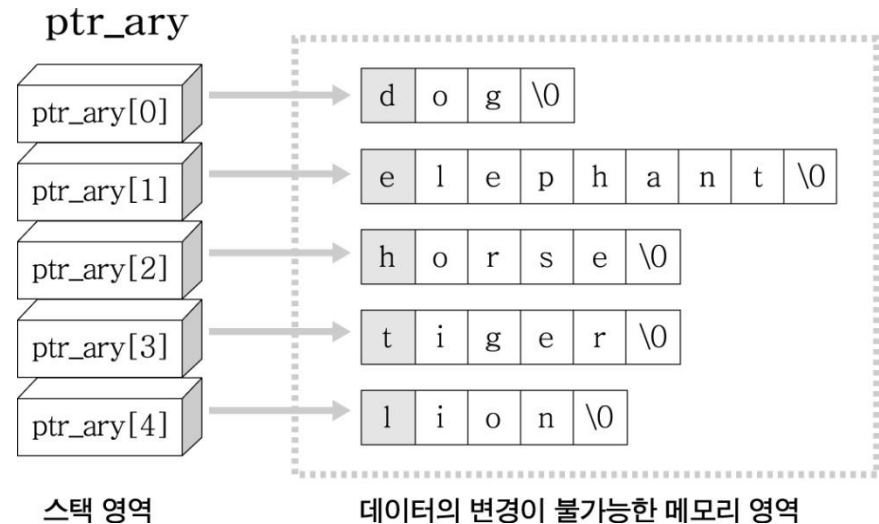
**ptr\_ary[2][2]**

# 문자열 포인터 배열

## ■ 문자열 포인터 배열

- 많이 사용되는 포인터 배열
- 문자열들을 효율적으로 저장
- 문자열 상수는 문자열의 시작주소

```
char* ptr_ary[ ] = {  
    "dog",  
    "elephant",  
    "horse",  
    "tiger",  
    "lion"  
};
```



# 문자열 포인터 배열과 문자열 상수

- 포인터 배열을 사용하여 여러 개의 문자열을 출력하는 예

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

int main()
{
    char *ptr_ary[5];
    int i;

    ptr_ary[0]="dog";
    ptr_ary[1]="elephant";
    ptr_ary[2]="horse";
    ptr_ary[3]="tiger";
    ptr_ary[4]="lion";

    for(i=0; i<5; i++) {
        cout << ptr_ary[i]<<"\n");
    }

    return 0;
}
```



# 문자열 포인터 배열과 2차원 문자 배열

## ■ 초기화

- 2차원 문자배열 : 모든 문자열이 배열에 복사
- 문자열 포인터 배열 : 주소값만 복사

```
char animal[5][10] = { "dog", "elephant", "horse", "tiger", "lion" };  
char *ptr_ary[5]   = { "dog", "elephant", "horse", "tiger", "lion" };
```

배열의 형태에 따라 문자열을 복사하거나 포인터를 저장한다.

초기화 방법은 같다.

## ■ 원소의 접근

- 2차원 문자 배열
- 문자열 포인터 배열

```
animal[0][1] == 'o'    animal[0] == "dog"  
ptr_ary[0][1] == 'o'   ptr_ary[0] == "dog"
```

## ■ 원소의 변경

- 2차원 문자 배열
- 문자열 포인터 배열

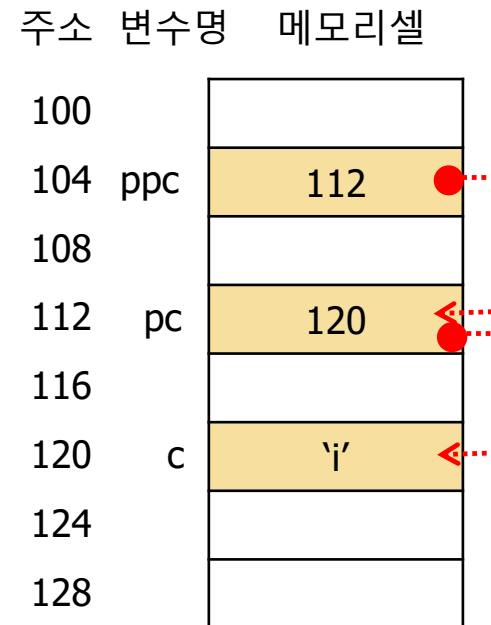
```
animal[0][1] = 'i'; // OK, dog가 dig로 바뀜  
ptr_ary[0][1] = 'i'; // Fail
```

# 이중 포인터

## ■ 이중 포인터

- 포인터 변수를 가리키는 포인터 변수

```
char c = '1';  
char* pc = &c;  
char** ppc = &pc;  
  
if ( *ppc == pc )  
{  
    // 이곳은 항상 실행된다.  
}  
  
if ( **ppc == c )  
{  
    // 이곳은 항상 실행된다.  
}
```



# 배열포인터

## ■ 배열포인터

- 배열을 가리키는 포인터

```
int arr[2][3] = { {11, 12, 13}, {21, 22, 23} };

int (*parr)[3]; // 배열포인터
int *p;    int **pt;

parr = arr;
cout << parr << arr;           // 주소 ptr == M
cout << parr+1 << arr+1;       // 주소 3 X 4byte 만큼 늘어남   parr+1 == arr+1
cout << *(parr+1) << parr[1] << *(arr+1) << arr[1];           // 주소
cout << ***(parr+1) << ***(arr+1) << *arr[1] << arr[1][0]; // 원소 값 : 21

p = arr[0];
cout << p << arr[0] << *arr; // p == arr[0] == *arr  각 원소의 시작주소
cout << *(p+1) << *(arr[0]+1) << *(*arr+1); // 원소 값 : 12

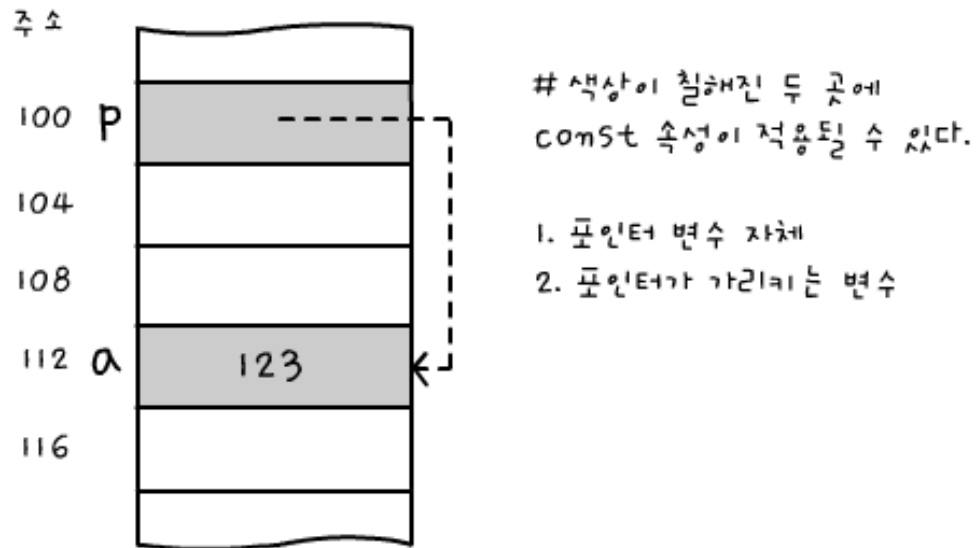
pt = &p; // pt = arr; (X)
cout << *pt << p; // *pt == p
cout << **pt << *p; // **pt == *p
```

- 2차원 배열을 포인터로 사용하기 위해서는 배열포인터로 사용하는 것이 좋다.

# const와 포인터

## ■ 포인터 변수에 const 속성을 부여할 때의 고려사항

- 포인터 변수 자체에 const 속성 부여
- 포인터 변수가 가리키는 변수에 const 속성 부여



# const와 포인터

## ■ 포인터 변수에 const 속성을 적용한 3가지 경우를 비교해보자.

- 포인터 변수가 가리키는 변수가 const인 경우 (포인터로 변수를 가리킬 수는 있지만 값을 변경할 수는 없음)

```
int i1 = 10;  
int i2 = 20;  
const int* p = &i1;  
  
p = &i2; // OK  
*p = 30; // FAIL
```

- 포인터 변수 자신이 const인 경우

```
int i1 = 10;  
int i2 = 20;  
int* const p = &i1;  
  
p = &i2; // FAIL  
*p = 30; // OK
```

- 두 변수 모두 const인 경우

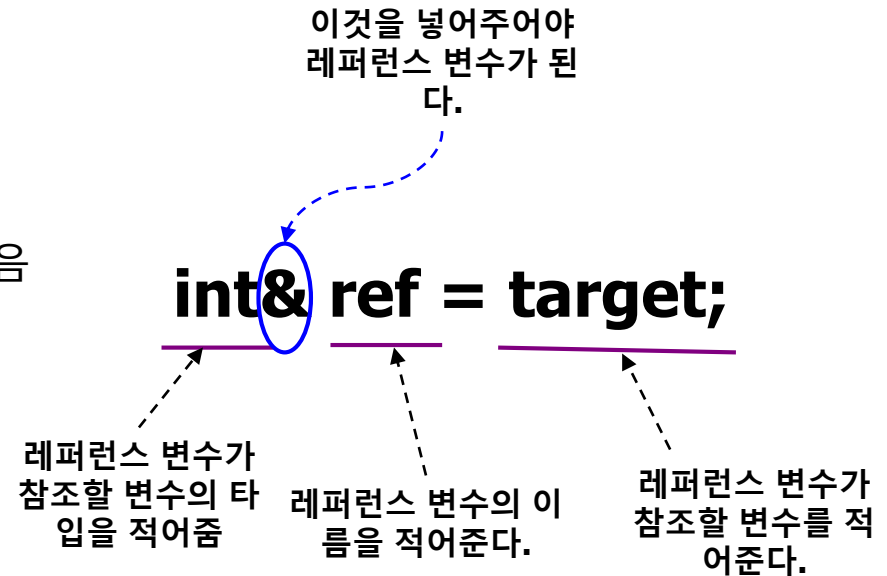
```
int i1 = 10;  
int i2 = 20;  
const int* const p = &i1;  
  
p = &i2; // FAIL  
*p = 30; // FAIL
```

## 2.2 레퍼런스

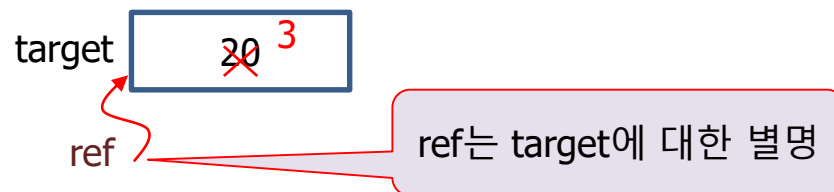
# 레퍼런스의 선언 및 사용

## ■ 레퍼런스(Reference) 선언

- 참조자 **&** 도입
- 이미 존재하는 변수에 대한 다른 이름 선언
  - 참조 변수는 이름만 생기며
  - 참조 변수에 새로운 공간을 할당하지 않음
  - 초기화로 지정된 기존 변수 공유



```
int target = 20;  
int& ref = target;  
ref = 3;
```



# 기본 타입 변수에 대한 참조

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

int main() {
    cout << "i" << '\t' << "n" << '\t' << "refn" << endl;
    int i = 1;
    int n = 2;
    int &refn = n; // 참조 변수 refn 선언. refn은 n에 대한 별명
    n = 4;
    refn++; // refn=5, n=5
    cout << i << '\t' << n << '\t' << refn << endl;

    refn = i; // refn=1, n=1
    refn++; // refn=2, n=2
    cout << i << '\t' << n << '\t' << refn << endl;

    int *p = &refn; // p는 n의 주소를 가짐
    *p = 20; // refn=20, n=20
    cout << i << '\t' << n << '\t' << refn << endl;
}
```

참조 변수 refn 선언

참조에 대한  
포인터 변수 선언

i	n	refn
1	5	5
1	2	2
1	20	20



# const 레퍼런스

## ■ const 레퍼런스

- 레퍼런스를 선언할 때 `const` 키워드를 함께 사용하면 레퍼런스가 참조하는 변수의 값을 변경할 수 없다.
- 레퍼런스는 자신이 참조하는 변수에 읽기 전용의 접근(read-only access)만 가능하다.

```
int data = 100;  
const int &ref = data;  
  
cout << ref;           // cout << data;의 의미  
ref = 200;           // const 레퍼런스는 변경할 수 없으므로 컴파일 에러  
data = 200;           // data를 직접 변경하는 것은 가능하다.
```

# 참조에 의한 호출

- 레퍼런스(참조) 변수를 가장 많이 활용하는 사례 : call by reference
  - 함수의 매개 변수를 레퍼런스 타입으로 선언
    - 참조 매개 변수(reference parameter)라고 부름
    - 참조매개 변수의 이름만 생기고 공간이 생기지 않음
    - 참조 매개 변수는 실인자 변수 공간 공유
    - 참조 매개 변수에 대한 조작은 실인자 변수 조작 효과

```
int main() {  
    int x = 10;          // main의 지역 변수 x는 foo 함수에서는 사용 불가  
  
    foo(x);              // 함수의 인자로 x를 전달한다.  
    cout << "x = " << x << endl;      // x = 11  
}  
  
void foo(int &ref) { // ref는 인자로 전달된 x의 별명이다.  
    ref++; // ref는 x의 별명이므로, main 함수의 x를 증가한다.  
}
```

## 2.3 함수에서의 인자전달

# 함수의 인자 전달

## ■ 함수의 인자 전달 방법을 결정하는 기준

- 함수를 호출할 때 넘겨준 인자를 함수 안에서 사용만 하고 변경하지 않을 때

→ 값에 의한 전달(**call by value**)

- 함수를 호출할 때 넘겨준 인자를 함수 안에서 변경해야 할 때

→ 포인터에 의한 전달이나 레퍼런스에 의한 전달 (**call by reference**)



(a) 값에 의한 호출



(b) 주소에 의한 호출

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

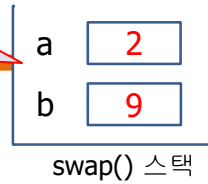
```
void swap(int a, int b) {
    int tmp;
```

```
    tmp = a;
    a = b;
    b = tmp;
```

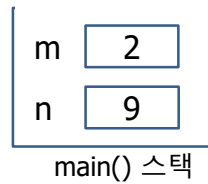
```
}
```

```
int main() {
    int m=2, n=9;
    swap(m, n);
    cout << m << " " << n;
}
```

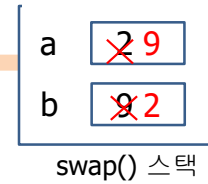
a, b에  
m, n의  
값 복사



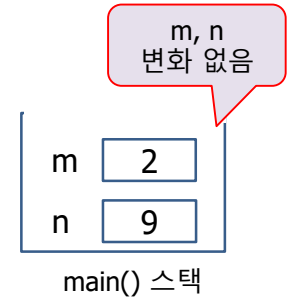
(1) swap() 호출 전



(2) swap() 호출 직후



(3) swap() 실행



(4) swap() 리턴 후

2 9

### 값에 의한 전달

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

```
void swap(int *a, int *b)
{
    int tmp;
```

```
    tmp = *a;
    *a = *b;
    *b = tmp;
```

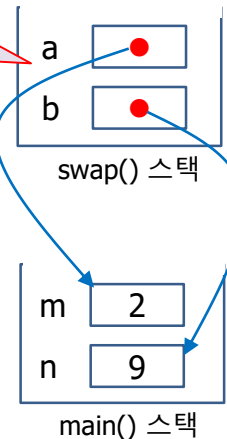
```
}
```

```
int main() {
    int m=2, n=9;
    swap(&m, &n);
    cout << m << " " << n;
}
```

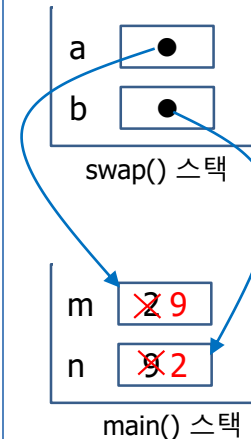
a, b에  
m, n의  
주소 전달



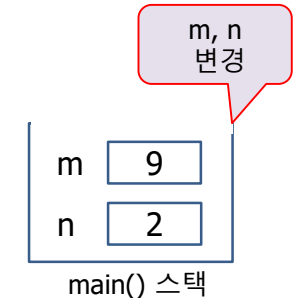
(1) swap() 호출 전



(2) swap() 호출 직후



(3) swap() 실행



(4) swap() 리턴 후

9 2

### 주소에 의한 전달

# 레퍼런스(참조)에 의한 호출 사례

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

```
void swap(int &a, int &b) {
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int m=2, n=9;
    swap(m, n);
    cout << m << " " << n;
}
```

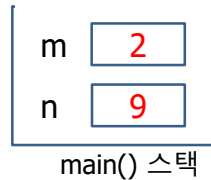
참조 매개 변수 a, b

참조 매개 변수를  
보통 변수처럼 사용

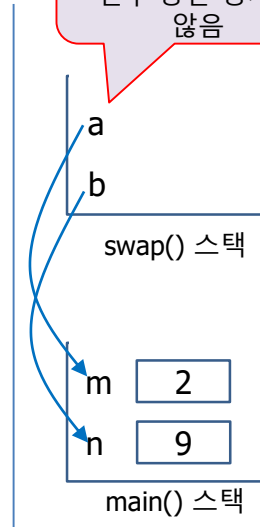
함수가 호출되면  
m, n에 대한 참조  
변수 a, b가  
생긴다.

9 2

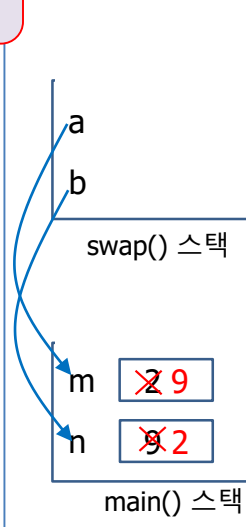
a, b는 m, n의 별명.  
a, b 이름만 생성.  
변수 공간 생기지  
않음



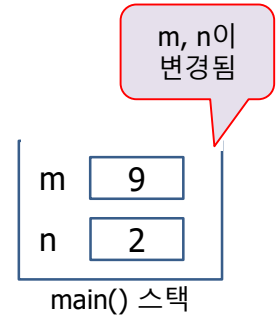
(1) swap() 호출 전



(2) swap() 호출 직후



(3) swap() 실행



(4) swap() 리턴 후

# const 레퍼런스의 전달

- 함수의 인자로 사용할 때의 const 레퍼런스
  - 구조체나 클래스 형을 넘길 때 유용하게 사용된다.
  - 구조체를 인자로 전달할 때 레퍼런스로 전달하면 구조체를 복사하지 않고 별명으로만 전달한다.

```
void Print(const STUDENT &s)
{
    s.grade[0] = 0; // 함수 안에서 s를 변경할 수 없으므로 컴파일 에러
}

int main()
{
    STUDENT s1 = {"장동건", 100, 90, 80, 99, 98};
    Print(s1);      // 레퍼런스에 의한 전달이므로
                   // Print 함수의 s는 s1의 별명임
}
```

# 함수의 인자전달 방법 – 포인터

## ■ 포인터 타입을 인자로 전달하는 방법의 정리

- 함수의 매개 변수는 포인터 타입으로 정의한다.
- 인자를 넘겨줄 때는 결과 값을 담고 싶은 변수의 주소를 넘겨준다.
- 함수 안에서 결과를 넘겨줄 때는 매개변수가 가리키는 곳에 값을 넣어준다.

```
void GCD_LCM(int a, int b, int* pgcd, int* plcm)
{
    ...
    *pgcd = 5;    // pgcd는 gcd를 가리키고 있으므로 gcd의 실제값이 바뀜
}

int main()
{
    ...
    GCD_LCM(28, 35, &gcd, &lcm);
    ...
}
```



# 함수의 인자전달 방법 – 레퍼런스

## ■ 레퍼런스 타입을 인자로 전달하는 방법의 정리

- 함수의 매개 변수는 레퍼런스 타입으로 정의한다.
- 인자를 넘겨줄 때는 결과 값을 담고 싶은 변수를 그대로 넘겨준다.
- 함수 안에서 결과를 넘겨줄 때는 매개 변수에 값을 넣어준다.

```
void GCD_LCM(int a, int b, int& pgcd, int& plcm)
{
    ...
    pgcd = 5;    // pgcd는 gcd의 별명이므로 gcd의 실제값이 바뀜
}

int main()
{
    ...
    GCD_LCM( 28, 35, gcd, lcm);
    ...
}
```

# 함수의 인자전달 방법 – 1차원 배열

## ■ 배열을 인자로 전달하는 방법의 정리

- 배열 전체를 한번에 보낼 수 없으므로, 배열의 시작 주소를 보냄
- 참조에 의한 호출(call-by-reference)를 사용하여 전달

## ■ 1차원 배열의 전달

- 배열의 주소를 넘겨준다
- 인자로 넘어온 배열을 사용할 때는 그냥 평범한 배열을 사용하듯이 하면 된다.

```
void UsingArray(int arr[]) { // 또는 int arr[3], int *arr
    ...
    cout << arr[0] << *(arr+1) << "\n";
}

int main() {
    int arr1[3] = {1,2,3};

    UsingArray(arr1);
    UsingArray(arr1+1);
    UsingArray(&arr[2]);
    ...
}
```

# 함수의 인자전달 방법 – 2차원 배열

## ■ 2차원 배열의 전달

- 1차원 배열의 경우와 같이 배열의 주소 전달
- 호출된 함수에서는 몇 개의 원소가 있는지 알 수 있도록 함.
  - 1차원 원소 개수는 절대로 생략하지 않는다.

```
void Using2DArray(int arr[][3]) { // 또는 int arr[5][3]
    ...
}

int main() {
    int array2[5][3] = {{ 1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}, {13, 14, 15}};

    Using2DArray(array2);

    ...
}
```

# 참조 리턴

## ■ C 언어의 함수 리턴

- 함수는 반드시 값만 리턴
  - 기본 타입 값 : int, char, double 등
  - 포인터 값

## ■ C++의 함수 리턴

- 함수는 값 외에 참조 리턴 가능
- 참조 리턴
  - 변수 등과 같이 현존하는 공간에 대한 참조 리턴
    - 변수의 값을 리턴하는 것이 아님

# 값을 리턴하는 함수 vs. 참조를 리턴하는 함수

## 문자 값을 리턴하는 get()

문자  
리턴

```
char c = 'a';  
  
char get() { // char 리턴  
    return c; // 변수 c의 문자('a') 리턴  
}  
  
char a = get(); // a = 'a'가 됨  
  
get() = 'b'; // 컴파일 오류
```

char 타입  
의 공간에  
대한 참조  
리턴

## char 타입의 참조(공간)을 리턴하는 find()

```
char c = 'a';  
  
char& find() { // char 타입의 참조 리턴  
    return c; // 변수 c에 대한 참조 리턴  
}  
  
char &ref = find(); // ref는 c에 대한 참조  
ref = 'M'; // c = 'M'  
  
find() = 'b'; // c = 'b'가 됨
```

find()가 리턴한 공  
간에 'b' 문자 저장

# 간단한 참조 리턴 사례

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

```
char& find(char s[], int index) {
    return s[index]; // 참조 리턴
}
```

s[index] 공간의 참조 리턴

```
int main() {
    char name[] = "Mike";
    cout << name << endl;
```

find()가 리턴한 위치에 문자 'm' 저장

```
    find(name, 0) = 'S'; // name[0]='S'로 변경
    cout << name << endl;
```

```
    char& ref = find(name, 2);
    ref = 't'; // name = "Site"
    cout << name << endl;
}
```

ref는 name[2] 참조

Mike  
Sike  
Site

(1) `char name[] = "Mike";`

M	i	k	e	\0
---	---	---	---	----

 name

(2) `return s[index];`

공간에 대한 참조, 즉 익명의 이름 리턴

M	i	k	e	\0
---	---	---	---	----

  
s[index] ↑

(3) `find(name, 0) = 'S';`

S	i	k	e	\0
---	---	---	---	----

(4) `ref = 't';`

S	i	t	e	\0
---	---	---	---	----

# 다음 수업

## ■ 개선된 함수 기능

- 1\_ 인라인 함수
- 2\_ 디폴트 인자
- 3\_ 함수 중복(오버로딩)
- 4\_ 함수 템플릿