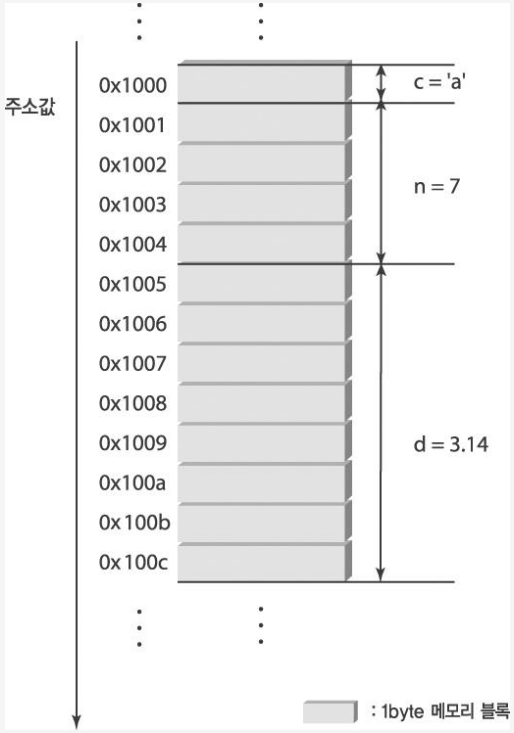
# 강05. 포인터

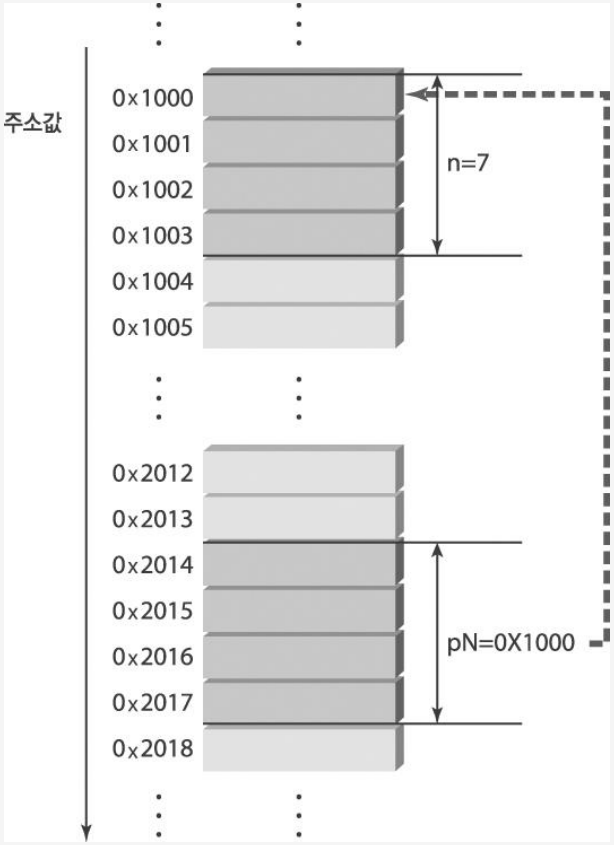
# 13. 포인터의 이해

- 포인터와 포인터 변수
  - 메모리의 주소 값을 저장하기 위한 변수
  - "포인터"를 흔히 "포인터 변수"라 한다.
  - 주소 값과 포인터는 다른 것이다.

```
int main(void)
{
    char c='a';
    int n=7;
    double d=3.14;
    . . . . .
```



- 그림을 통한 포인터의 이해
  - 컴퓨터의 주소 체계에 따라 크기가 결정
  - 32비트 시스템 기반 : 4 바이트

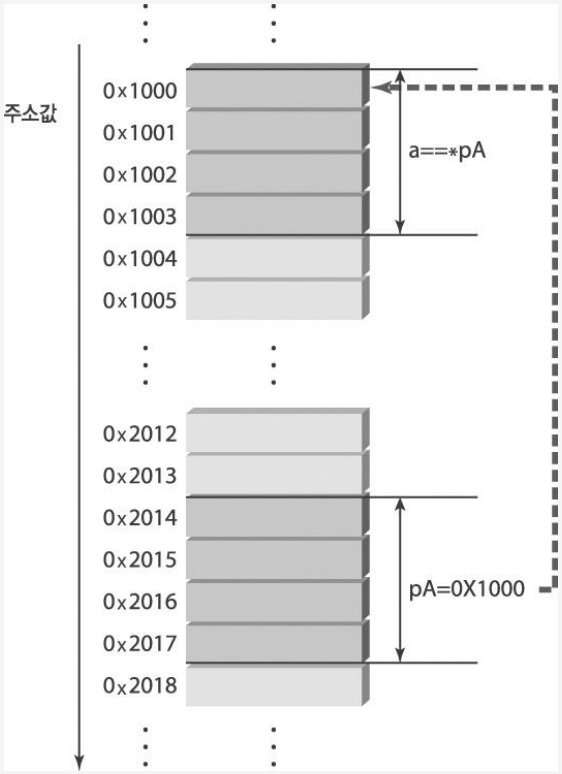


- 포인터의 타입과 선언
  - 포인터 선언 시 사용되는 연산자 : *
  - A형 포인터(A*) : A형 변수의 주소 값을 저장

```
int main(void)
{
   int *a;              // a라는 이름의 int형 포인터
   char *b;             // b라는 이름의 char형 포인터
   double *c;// c라는 이름의 double형 포인터
    . . . . .
```

- 주소 관련 연산자
  - & 연산자 : 변수의 주소 값 반환
  - * 연산자 : 포인터가 가리키는 메모리 참조

```
int main(void)
{
   int a=2005;
   int *pA=&a;
   printf("%d", a);   //직접 접근
   printf("%d", *pA);   // 간접 접근
   . . . . .
```



```
/* pointer1.c */
#include <stdio.h>

int main(void)
{
    int a=2005;
    int* pA=&a;

    printf("pA : %d ₩n", pA);
    printf("&a : %d ₩n", &a);

    (*pA)++;            //a++와 같은 의미를 지닌다.

    printf("a   : %d ₩n", a);
    printf("*pA : %d ₩n", *pA);

    return 0;
}
```

- 포인터에 다양한 타입이 존재하는 이유
  - 포인터 타입은 참조할 메모리의 크기 정보를 제공

```
#include <stdio.h>
 int main(void)
 {
     int a=10;
     int *pA = &a;
     double e=3.14;
     double *pE=&e;

     printf("%d %f", *pA, *pE);
     return 0;
 }
```

- 오류 1

```
int main(void)
{
   int *pA;       // pA는 쓰레기 값으로 초기화 됨
   *pA=10;
   return 0;
}
```

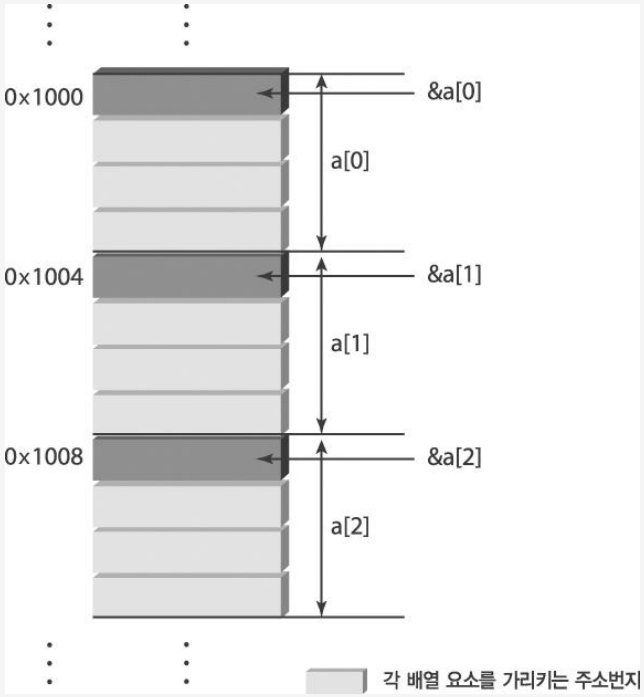
- 오류 2

```
int main(void)
{
   int* pA=100;   // 1000이 어딘 줄 알고???
   *pA=10;
   return 0;
 }
```

# 14. 포인터와 배열! 함께 이해하기

- 배열의 이름의 정체
  - 배열 이름은 첫 번째 요소의 주소 값을 나타낸다.

```
int a[5]={0, 1, 2, 3, 4}
```



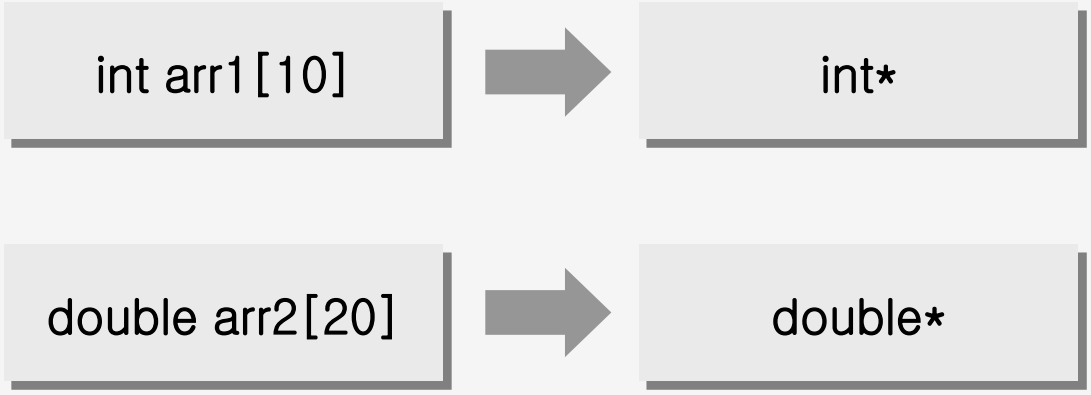
```
/* pointer_array1.c */
#include <stdio.h>

int main(void)
{
    int a[5]={0, 1, 2, 3, 4};
    double *b = {1.1,2.2,3.3,4.4,5.5};

    printf("%d, %d ₩n", a[0], a[1]);
    printf("%d 번지 , %d 번지 ₩n", &a[0], &a[1]);
    printf("%d 번지 , %d 번지 ₩n", a, a+1);

    printf("배열 이름 : %d ₩n", a);

    return 0;
}
```

- 배열 이름과 포인터 비교

| 비교 조건 \ 비교 대상 | 포인터 | 배열 이름 |
|---|---|---|
| 이름이 존재하는가 | 물론 있다. | 당연히 있다. |
| 무엇을 나타내는가 | 메모리의 주소 | 메모리의 주소 |
| 변수인가 상수인가 | 변수 | 상수 |

```
int main(void)
{
    int a[5]={0, 1, 2, 3, 4};
    int b=10;
    a=&b;   //a는 상수이므로 오류, a가 변수였다면 OK!
}
```

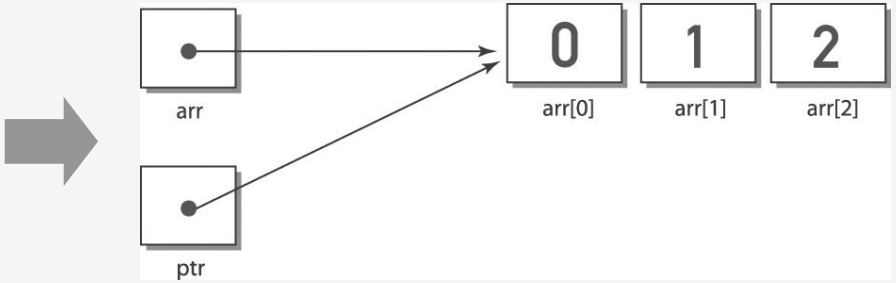
- 배열 이름의 타입
  - 배열 이름도 포인터이므로 타입이 존재
  - 배열 이름이 가리키는 배열 요소에 의해 결정



- 키보드에서 5개의 정수를 입력받아서
- 그중 가장 작은 수를 찾아서 출력하시오
- (단 배열과 함수를 이용해서 구성할 것.
- 함수는 배열을 매개변수로 받아서
- 그중 가장 작은 수를 되돌려 주는 함수임)

- 배열 이름의 활용
  - 배열 이름을 포인터처럼, 포인터를 배열 이름처럼 활용하는 것이 가능!

```
/* pointer_array2.c */
#include <stdio.h>

int main(void)
{
    int arr[3]={0, 1, 2};
    int *ptr;

    ptr=arr;

    printf("%d, %d, %d ₩n", ptr[0], ptr[1], ptr[2]);
    return 0;
}
```



- 포인터 연산이란?
  - 포인터가 지니는 값을 증가 혹은 감소시키는 연산을 의미

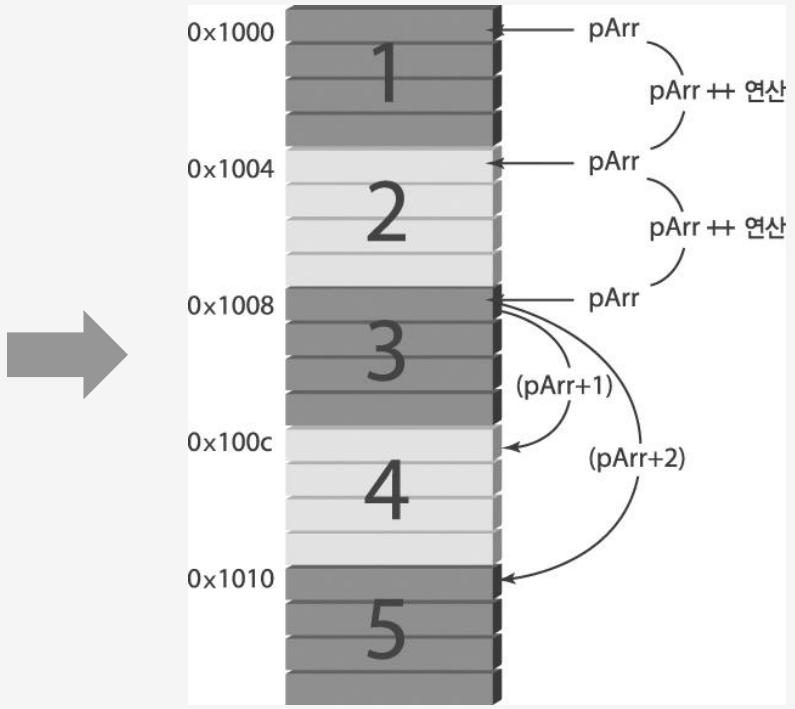
```
ptr1++;
ptr1 += 3;
--ptr1;
ptr2=ptr1+2;
```

- 포인터 연산
  - 포인터가 가리키는 대상의 자료형에 따라서 증가 및 감소되는 값이 차이를 지님

```
/* pointer_op.c */
#include <stdio.h>

int main(void)
{
    int* ptr1=0;                    // int* ptr1=NULL; 과 같은 문장
    char* ptr2=0;                   // char* ptr2=NULL; 과 같은 문장
    double* ptr3=0;                 // double* ptr3=NULL; 과 같은 문장

    printf("%d 번지, %d 번지, %d 번지 ₩n", ptr1++, ptr2++, ptr3++);
    printf("%d 번지, %d 번지, %d 번지 ₩n", ptr1, ptr2, ptr3);

    return 0;
}
```

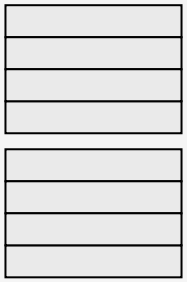
- 포인터 연산을 통한 배열 요소의 접근

```
/* pointer_array3.c */
#include <stdio.h>

int main(void)
{
   int arr[5]={1, 2, 3, 4, 5};

   int* pArr=arr;
   printf("%d ₩n", *pArr);

   printf("%d ₩n", *(++pArr));
   printf("%d ₩n", *(++pArr));

   printf("%d ₩n", *(pArr+1));
   printf("%d ₩n", *(pArr+2));

   return 0;
}
```



- 포인터와 배열을 통해서 얻을 수 있는 중대한 결론

```
/* two_same.c */
#include <stdio.h>

int main(void)
{
   int arr[2]={1, 2};
   int* pArr=arr;

   printf("%d, %d ₩n", arr[0], *(arr+1));

   printf("%d, %d ₩n", pArr[0], *(pArr+1));

   return 0;
}
```

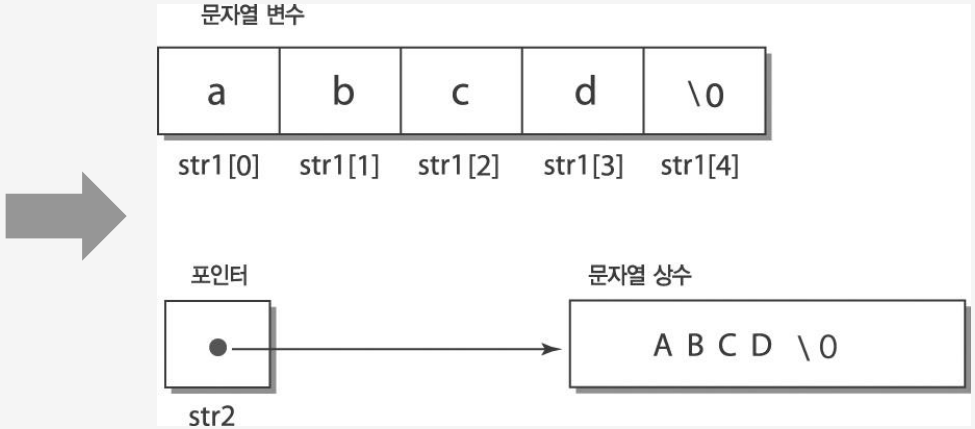


arr[i] == *(arr+i)

arr이 "포인터"이거나 "배열 이름"인 경우

- 문자열 표현 방식의 이해
  - 배열 기반의 문자열 변수
  - 포인터 기반의 문자열 상수

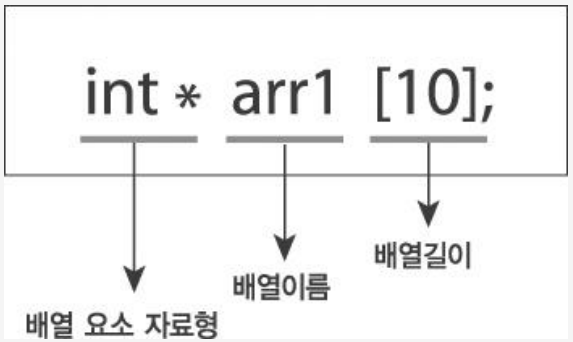
```
char str1[5]="abcd";
char *str2="ABCD";
```



```
/* str_prn.c*/
#include <stdio.h>

int main()
{
    char str1[5]="abcd";
    char *str2="ABCD";

    printf("%s ₩n", str1);
    printf("%s ₩n", str2);

    str1[0]='x';
    //  str2[0]='x'; //Error ?!

    printf("%s ₩n", str1);
    printf("%s ₩n", str2);

    return 0;
}
```

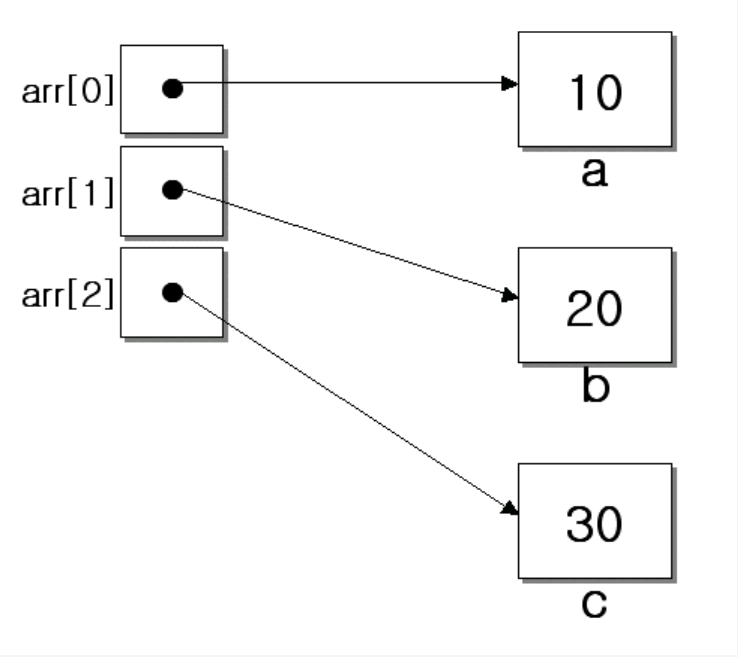
- 포인터 배열
  - 배열의 요소로 포인터를 지니는 배열

```
int* arr1[10];
double* arr2[20];
char* arr3[30];
```



- 포인터 배열의 예#1

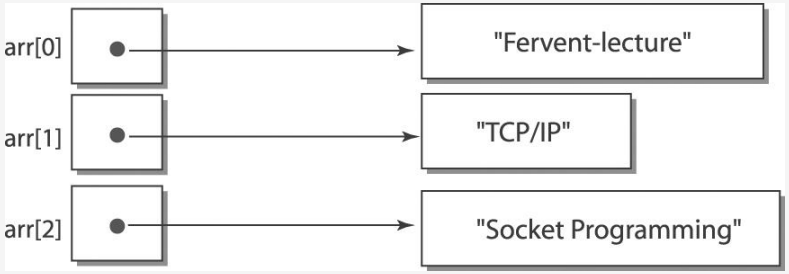
```
/* ptr_arr.c */
#include <stdio.h>

int main(void)
{
   int a=10, b=20, c=30;
   int* arr[3]={&a, &b, &c};

   printf("%d ₩n", *arr[0]);
   printf("%d ₩n", *arr[1]);
   printf("%d ₩n", *arr[2]);

   return 0;
}
```



- 포인터 배열의 예#2

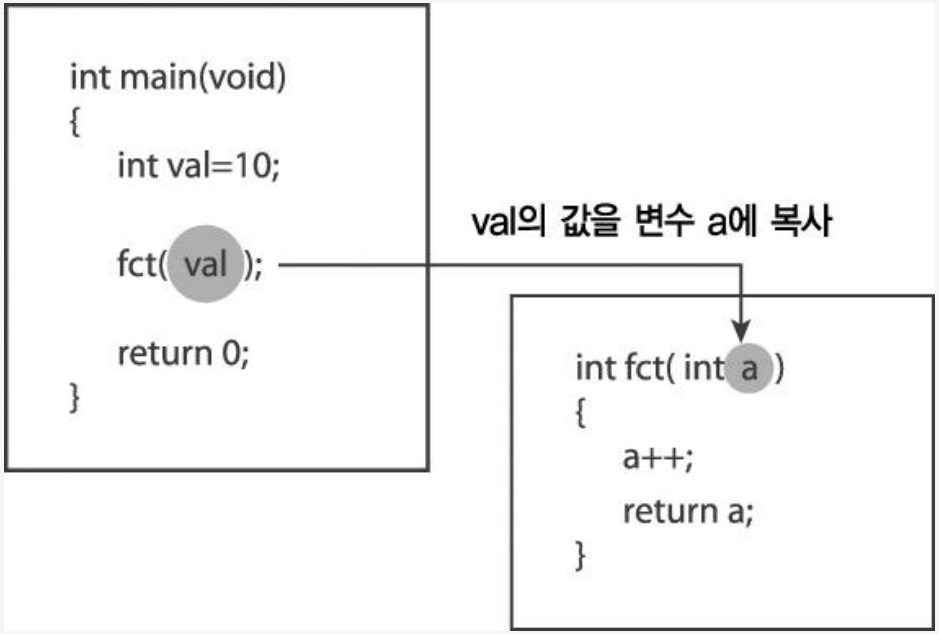
```
/* str_arr.c */
#include <stdio.h>

int main(void)
{
   char* arr[3]={
          "Fervent-lecture",
          "TCP/IP",
          "Socket Programming"
   };
    printf("%s ₩n", arr[0]);
   printf("%s ₩n", arr[1]);
   printf("%s ₩n", arr[2]);

   return 0;
}
```

```
char * arr[3] = {"Fervent-lectur","TCP/IP","Socket Programming"};
                          ↓
char * arr[3] = {0x1000, 0x2000, 0x3000};
```



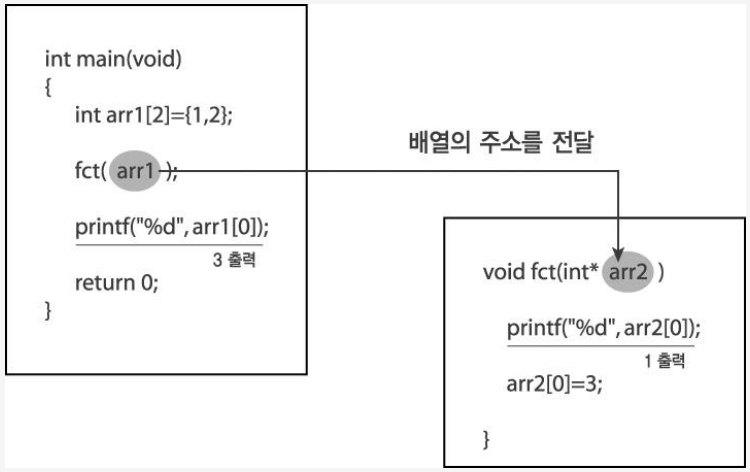
# 15장. 포인터와 함수에 대한 이해

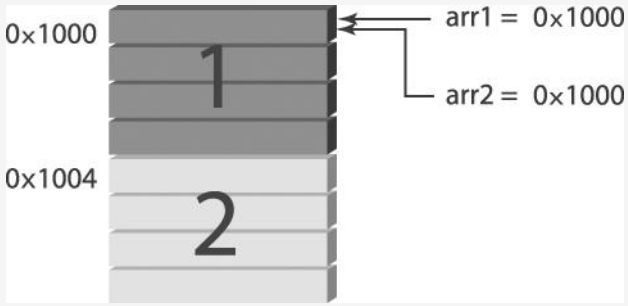
- 기본적인 인자의 전달 방식
  - 값의 복사에 의한 전달



- 배열의 함수 인자 전달 방식
  - 배열 이름(배열 주소, 포인터)에 의한 전달

```
#include <stdio.h>
void fct(int *arr2);

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

    fct(arr1);
    printf("%d ₩n", arr1[0]);
    return 0;
}
void fct(int *arr2)
{
    printf("%d ₩n", arr2[0]);
    arr2[0]=3;
}
```



- 배열 이름, 포인터의 sizeof 연산
  - 배열 이름 : 배열 전체 크기를 바이트 단위로 반환
  - 포인터 : 포인터의 크기(4)를 바이트 단위로 반환

```
#include <stdio.h>

int main(void)
{
   int arr[5];
   int* pArr=arr;

   printf("%d ₩n", sizeof(arr) );          // 20 출력
   printf("%d ₩n", sizeof(pArr) );         // 4 출력
   return 0;
}
```

- "int * pArr" vs. "int pArr[ ]"
  - 둘 다 같은 의미를 지닌다.
  - 선언 "int pArr[ ]"은 함수의 매개 변수 선언 시에만 사용 가능

```
int function(int pArr[])
{
   int a=10;
   pArr=&a;          // pArr이 다른 값을 지니게 되는 순간
   return *pArr;
}
```

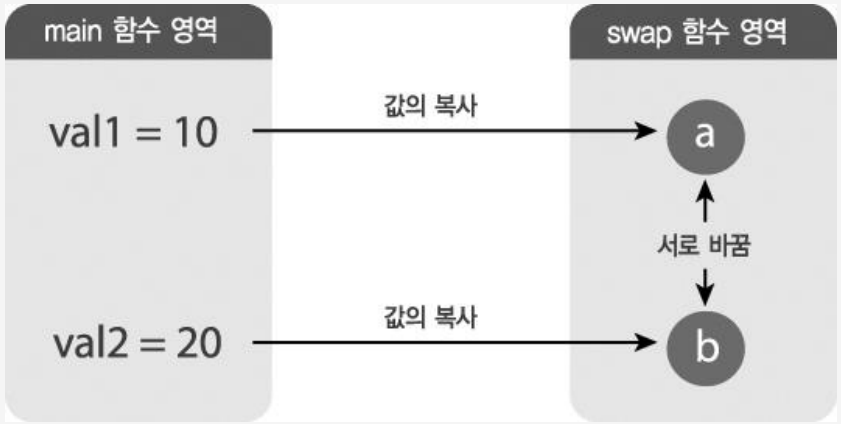
- Call-By-Value
  - 값의 복사에 의한 함수의 호출
  - 가장 일반적인 함수 호출 형태

```
#include <stdio.h>
int add(int a, int b);

int main(void)
{
    int val1=10;
    int val2=20;
    printf(" 결  과 : ", add(val1, val2);

    return 0;
}
int add(int a, int b)
{
    return a+b;
}
```

# Call-By-Value와 Call-By-Reference

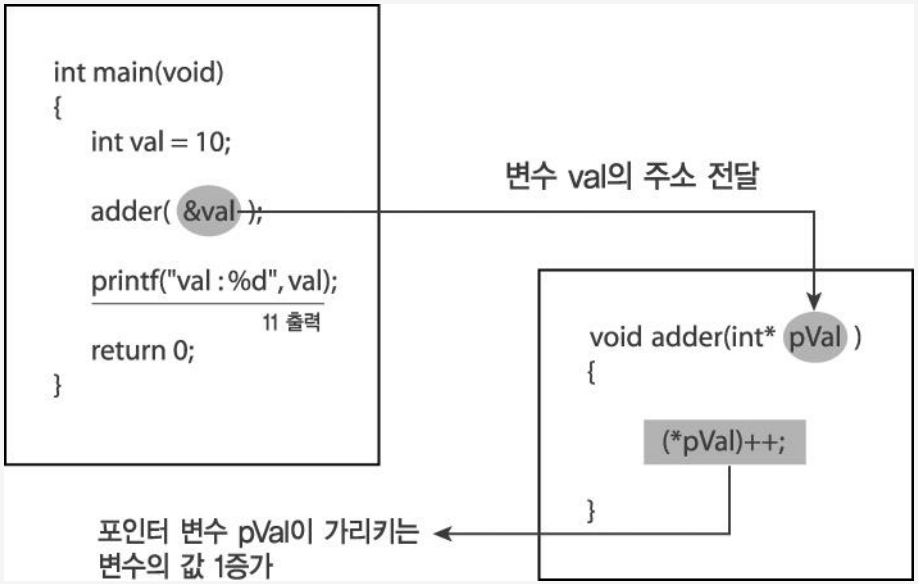
- Call-By-Value에 의한 swap

```
int main(void)
{
    int val1=10;
    int val2=20;
    swap(val1, val2);

    printf("val1 : %d ₩n", val1);
    printf("val2 : %d ₩n", val2);
    return 0;
}
void swap(int a, int b)
{
    int temp=a;
    a=b;
    b=temp;

    printf("a : %d ₩n", a);
    printf("b : %d ₩n", b);
}
```



# Call-By-Value와 Call-By-Reference

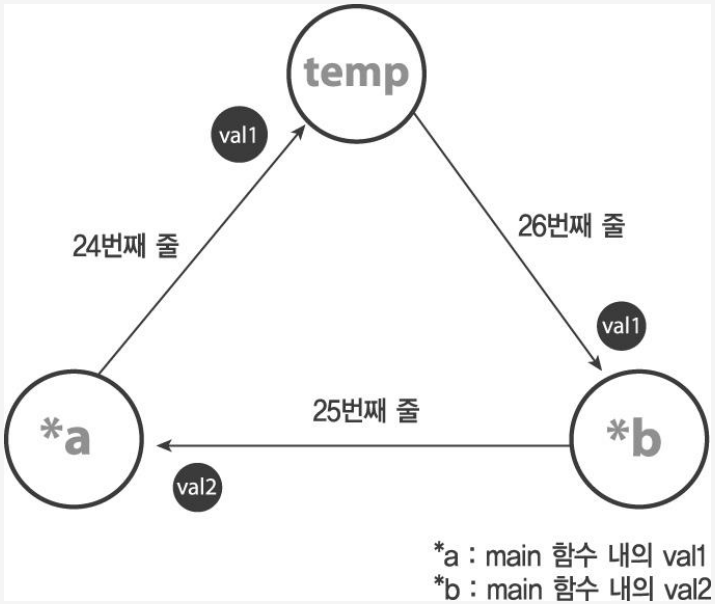
- Call-By-Reference
  - 참조(참조를 가능케 하는 주소 값)를 인자로 전달하는 형태의 함수 호출



# Call-By-Value와 Call-By-Reference

- Call-By-Reference에 의한 swap

```
int main(void)
{
    int val1=10;
    int val2=20;
    printf("Before val1 : %d ₩n", val1);
    printf("Before val2 : %d ₩n", val2);
    swap(&val1, &val2);      //val1, val2 주소 전달

    printf("After val1 : %d ₩n", val1);
    printf("After val2 : %d ₩n", val2);
    return 0;
}
void swap(int* a, int* b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
```



# Call-By-Value와 Call-By-Reference

- scanf 함수 호출 시 &를 붙이는 이유
  - case 1

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

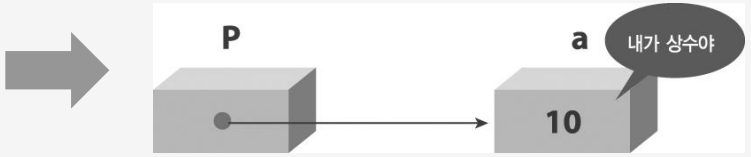
  - case 2

```
int main(void)
{
    char str[100];
    printf("문자열 입력 : ");
    scanf("%s", str);
    . . . . .
```

# 포인터와 const 키워드

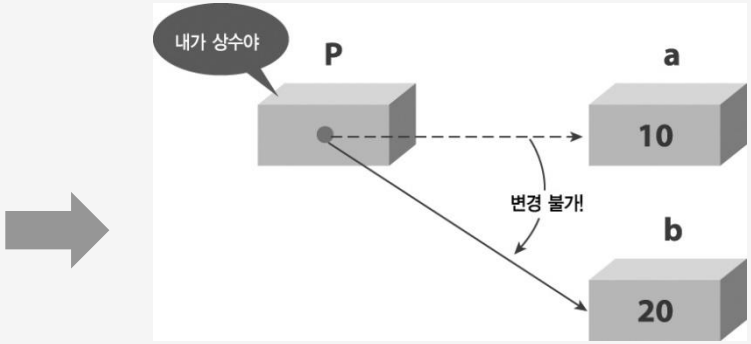
- 포인터가 가리키는 변수의 상수화

```
int a = 10;
const int * p = &a;
*p=30            // Error!
a=30             // OK!
```




- 포인터 상수화

```
int a=10;
int b=20;
int * const p = &a;
p=&b             // Error!
*p=30            // OK!
```



# 포인터와 const 키워드

- const 키워드를 사용하는 이유
  - 컴파일 시 잘못된 연산에 대한 에러 메시지
  - 프로그램을 안정적으로 구성

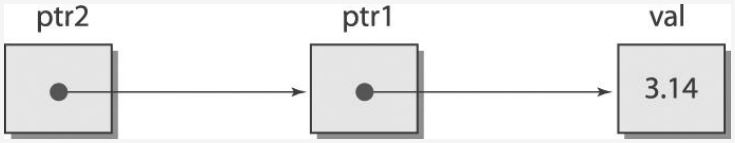
```
#include <stdio.h>
float PI=3.14;

int main(void)
{
    float rad;
    PI=3.07;      // 분명히 실수!!

    scanf("%f", &rad);
    printf("원의 넓이는 %f ₩n", rad*rad*PI);
    return 0;
}
```

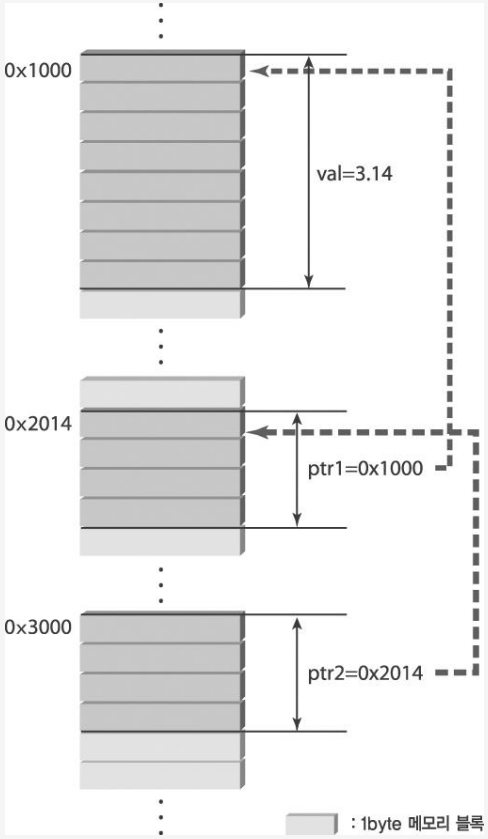
```
#include <stdio.h>
const float PI=3.14;

int main(void)
{
    float rad;
     PI=3.07;     // Compile Error 발생!

    scanf("%f", &rad);
    printf("원의 넓이는 %f ₩n", rad*rad*PI);
    return 0;
}
```

# 16장. 포인터의 포인터

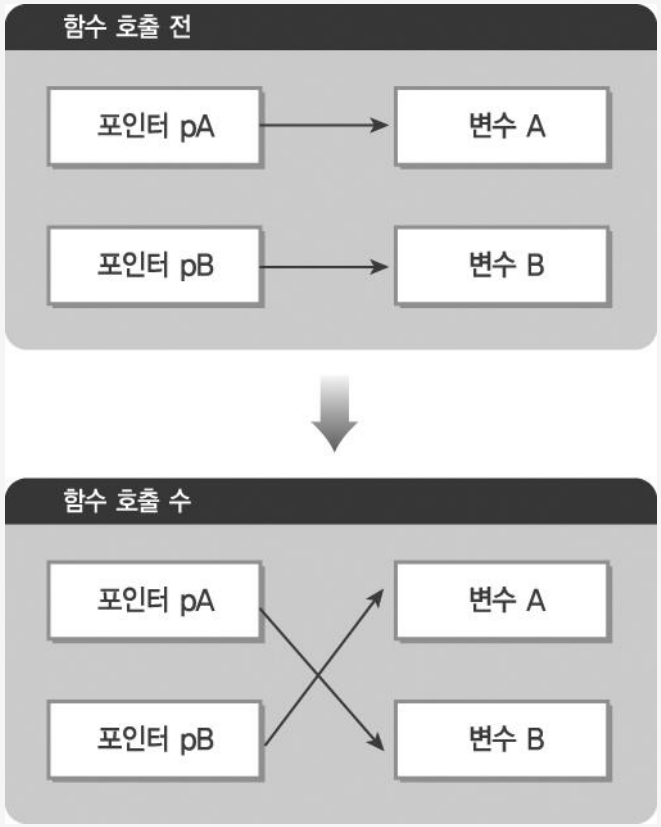
- 포인터의 포인터
  - 더블 포인터라고 불린다.
  - 싱글 포인터의 주소 값을 저장하는 용도의 포인터

```
int main(void)
{
   double val=3. 14;
   double *ptr1 = &val;    // 싱글 포인터
   double **ptr2 = &ptr1; // 더블 포인터
   . . .
```



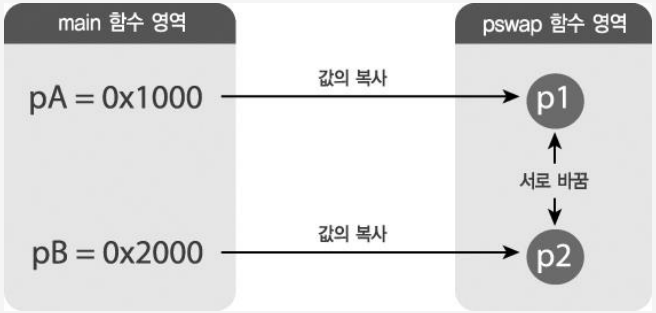
- 더블 포인터의 의한 Call-By-Reference
  - 다음 그림이 제시하는 프로그램의 구성을 통한 이해



- 구현 사례 1 : 효과 없는 swap 함수의 호출

```
/* ptr_swap1.c */
#include <stdio.h>

void pswap(int *p1, int *p2);

int main(void)
{
    int A=10, B=20;
    int *pA, *pB;
    pA=&A, pB=&B;

    pswap(pA, pB);

    // 함수 호출 후
    printf("pA가 가리키는 변수 : %d ₩n", *pA);
    printf("pB가 가리키는 변수 : %d ₩n", *pB);

    return 0;
}
```

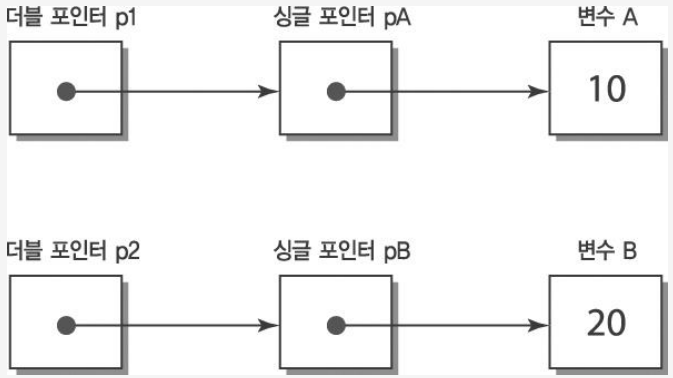
```
void pswap(int *p1, int *p2)
{
    int *temp;
    temp=p1;
    p1=p2;
    p2=temp;
}
```



- 구현 사례 2 : 더블 포인터 입장에서의 swap

```
/* ptr_swap2.c */
#include <stdio.h>

void pswap(int **p1, int **p2);

int main(void)
{
    int A=10, B=20;
    int *pA, *pB;
    pA=&A, pB=&B;

    pswap(&pA, &pB);

    //함수 호출 후
    printf("pA가 가리키는 변수 : %d ₩n", *pA);
    printf("pB가 가리키는 변수 : %d ₩n", *pB);

    return 0;
}
```

```
void pswap(int **p1, int **p2)
{
    int *temp;
    temp=*p1;
    *p1=*p2;
    *p2=temp;
}
```



- 포인터 배열과 포인터 타입
  - 1차원 배열의 경우 배열이름이 가리키는 대상을 통해서 타입이 결정된다.
  - 포인터 배열이라고 하더라도 마찬가지!

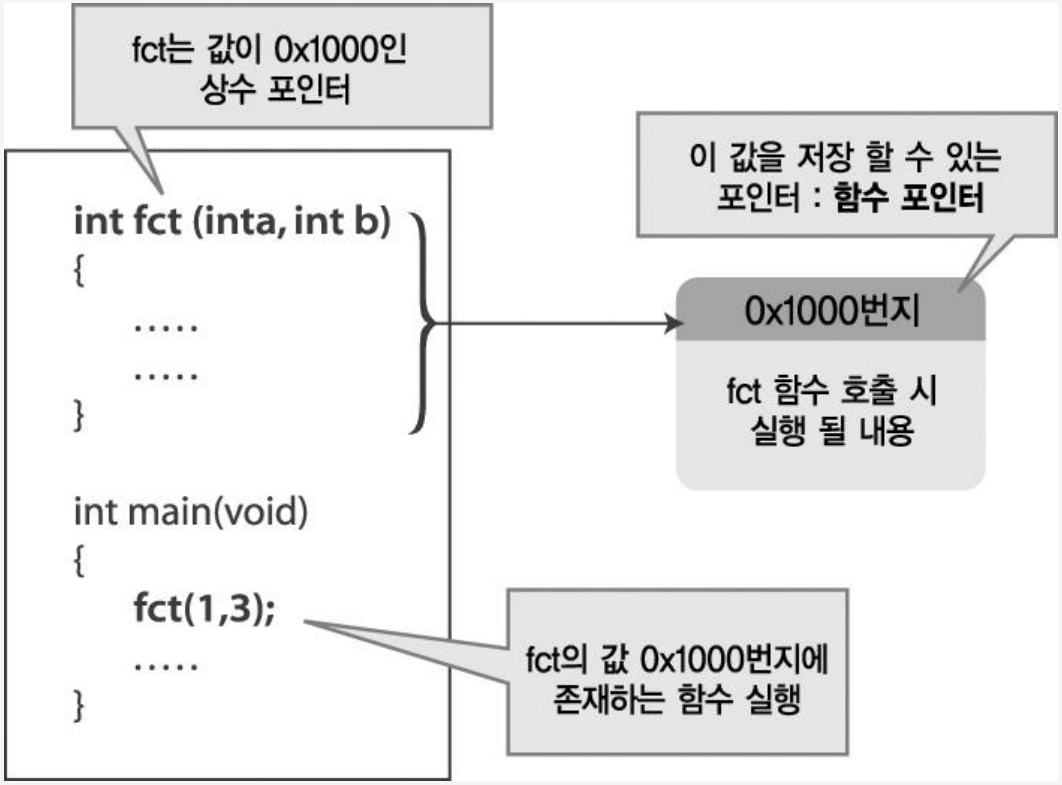
```
int* arr1[10];
double* arr2[20];
char* arr3[30];
```

- 지금까지...
  - swap 함수와 같이 함수 내에서 데이터의 조작을 하기 위해서...

- 앞으로...
  - 메모리 동적 할당
  - 자료구조의 구현

# 17. 함수 포인터와 void 포인터

- 함수 포인터의 이해


fct는 값이 0x1000인
상수 포인터
이 값을 저장 할 수 있는
포인터 : 함수 포인터
int fct (inta, int b)
{
.....
.....
}
0x1000번지
fct 함수 호출 시
실행 될 내용
int main(void)
{
fct(1,3);
.....
}
fct의 값 0x1000번지에
존재하는 함수 실행

- 함수 이름의 포인터 타입을 결정짓는 요소
  - 리턴 타입 + 매개 변수 타입

```
int fct1 (int a)
{
    a++
    return a;
}
```

```
int (*fPtr1) (int);
```

```
double fct2 (double a, double b)
{
    double add=a+b;
    return add;
}
```

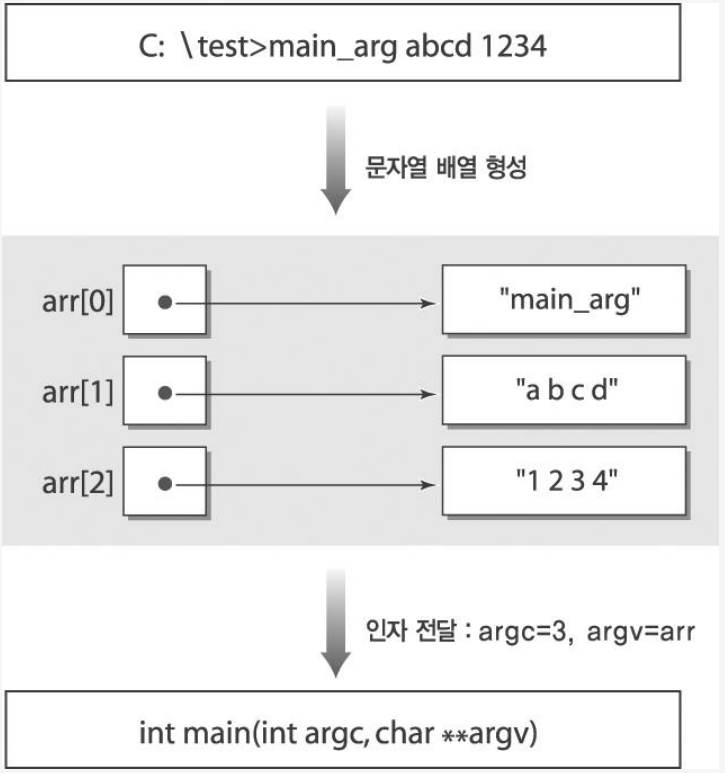
```
double (*fPtr2) (double, double
```

- void형 포인터란 무엇인가?
  - 자료형에 대한 정보가 제외된, 주소 정보를 담을 수 있는 형태의 변수
  - 포인터 연산, 메모리 참조와 관련된 일에 활용 할 수 없다.

```
int main(void)
{
    char c='a';
    int n=10;
    void * vp;    // void 포인터 선언
    vp=&c;
    vp=&n;
    . . . . .
```

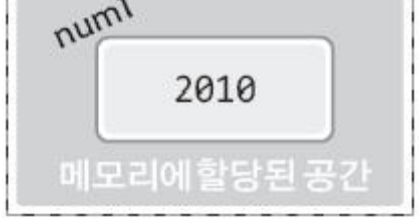
```
int main(void)
{
    int n=10;
    void * vp=&n;
    *vp=20;                 // Error!
    vp++;                   // Error!
    . . . . .
```

```c
/* main_arg.c */
#include <stdio.h>

int main(int argc, char **argv)
{
    int i=0;
    printf("전달된 문자열의 수 : %d ₩n", argc);

    for(i=0; i<argc; i++)
        printf("%d번째 문자열 : %s ₩n", i+1, argv[i]);

    return 0;
}
```



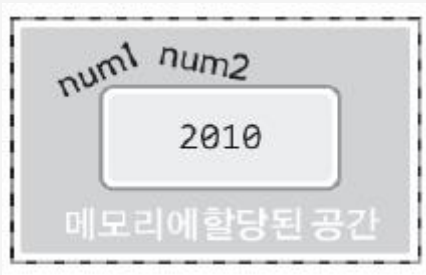
# Reference 의 이해

```
int num1=2010;
```




변수의 선언으로 인해서 num1 이라는 이름으로 메모리 공간이 할당된다 .

```
int &num2=num1;
```




reference의 선언으로 인해서 num1 의 메모리 공간에 num2 라는 이름이 추가로 붙게 된다 .

reference는 기존에 선언된 변수에 붙이는 ' 별칭 ' 이다 . 그리고 이렇게 reference가 만들어지면 이는 변수의 이름과 사실상 차이가 없다 .

# reference 관련 예제와 reference의 선언

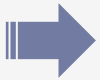
```
int main(void)
{
    int num1=1020;
    int &num2=num1;
    num2=3047;
    cout<<"VAL: "<<num1<<endl;
    cout<<"REF: "<<num2<<endl;
    cout<<"VAL: "<<&num1<<endl;
    cout<<"REF: "<<&num2<<endl;
    return 0;
}
```

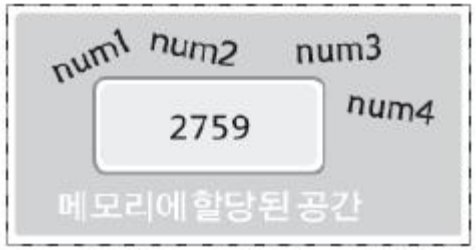
num2 는 num1 의 reference이다 . 따라서 이후부터는 num1 으로 하는 모든 연산은 num2 로 하는것과 동일한 결과를 보인다 .

실행결과

```
VAL: 3047
REF: 3047
VAL: 0012FF60
REF: 0012FF60
```

```
int num1=2759;
int &num2=num1;
int &num3=num2;
int &num4=num3;
```






reference의 수에는 제한이 없으며 , reference 를 대상으로 reference를 선언하는 것도 가능하다 .

# C/C++ reference의 선언 가능 범위

```
int &ref=20;        (×)
```

상수 대상으로의 reference 선언은 불가능하다 .

```
int &ref;           (×)
```

reference는 생성과 동시에 누군가를 참조해야 한다 .

```
int &ref=NULL;      (×)
```

포인터처럼 NULL 로 초기화하는 것도 불가능하다 .

불가능한 reference의 선언의 예

정리하면 , reference는 선언과 동시에 누군가를 참조해야 하는데 , 그 참조의 대상은 기본적으로 변수가 되어야 한다 . 그리고 참조자는 참조의 대상을 변경할 수 없다 .

```
int main(void)
{
    int arr[3]={1, 3, 5};
    int &ref1=arr[0];
    int &ref2=arr[1];
    int &ref3=arr[2];

    cout<<ref1<<endl;
    cout<<ref2<<endl;
    cout<<ref3<<endl;
    return 0;
}
```

변수의 성향을 지니는 대상이라면 참조자의 선언이 가능하다 .
배열의 요소 역시 변수의 성향을 지니기 때문에 reference의 선언이 가능하다 .

실행결과

```
1
3
5
```

# 포인터 변수 대상의 reference 선언

```
int main(void)
{
    int num=12;
    int *ptr=&num;
    int **dptr=&ptr;

    int &ref=num;
    int *(&pref)=ptr;
    int **(&dpref)=dptr;

    cout<<ref<<endl;
    cout<<*pref<<endl;
    cout<<**dpref<<endl;
    return 0;
}
```

ptr 과 dptr 역시 변수이다 . 다만 주소 값을 저장하는 포인터 변수일 뿐이다 . 따라서 이렇듯 reference의 선언이 가능하다 .

실행결과

```
12
12
12
```

# C/C++ Call-by-value & Call-by-reference

```
void SwapByValue(int num1, int num2)
{
    int temp=num1;
    num1=num2;
    num2=temp;
}   // Call-by-value
```

값을 전달하면서 호출하게 되는 함수이므로 이 함수는 Call-by-value 이다 . 이 경우 함수 외에 선언된 변수에는 접근이 불가능하다 .

```
void SwapByRef(int * ptr1, int * ptr2)
{
    int temp=*ptr1;
    *ptr1=*ptr2;
    *ptr2=temp;
}   // Call-by-reference
```

값은 값이되 , 주소 값을 전달하면서 호출하게 되는 함수이므로 이 함수는 Call-by-reference 이다 . 이 경우 인자로 전달된 주소의 메모리 공간에 접근이 가능하다 !

# C/C++ Call-by-address? Call-by-reference!

```
int * SimpleFunc(int * ptr)
{
    return ptr+1;
}
```

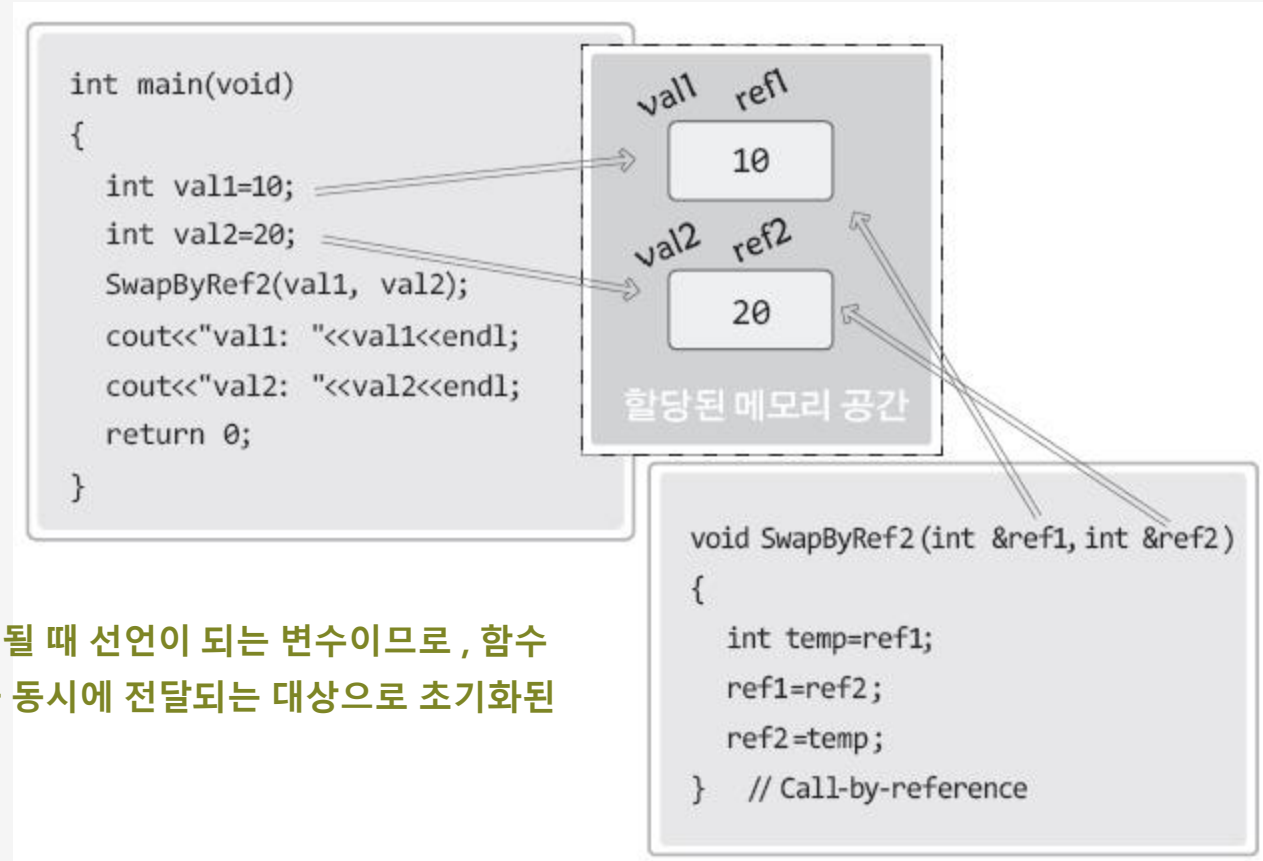
포인터 ptr 에 전달된 주소 값의 관점에서 보면
이는 Call-by-value 이다 .

```
int * SimpleFunc(int * ptr)
{
    if(ptr==NULL)
        return NULL;
    *ptr=20;
    return ptr;
}
```

주소 값을 전달 받아서 외부에 있는 메모리 공간
에 접근을 했으니 이는 Call-by-reference 이다 .

C++ 에는 두 가지 형태의 Call-by-reference 가 존재한다 . 하나는 주소 값 을
이용하는 형태이며 , 다른 하나는 reference 를 이용하는 형태이다 .

# reference를 이용한 Call-by-reference




매개변수는 함수가 호출될 때 선언이 되는 변수이므로 , 함수 호출의 과정에서 선언과 동시에 전달되는 대상으로 초기화된다 .

즉 , 매개변수에 선언된 reference는 여전히 선언과 동시에 초기화된다 .

reference 기반 의 Call-by-reference !

# const reference

함수의 호출 형태

```
int num=24;
HappyFunc(num);
```

함수의 정의 형태

```
void HappyFunc(int &ref) { . . . . }
```

함수의 정의형태와 함수의 호출형태를 보아도 값의 변경유무를 알 수 없다 ! 이를 알려면 HappyFunc 함수의 몸체 부분을 확인해야 한다 . 그리고 이는 큰 단점이다 !

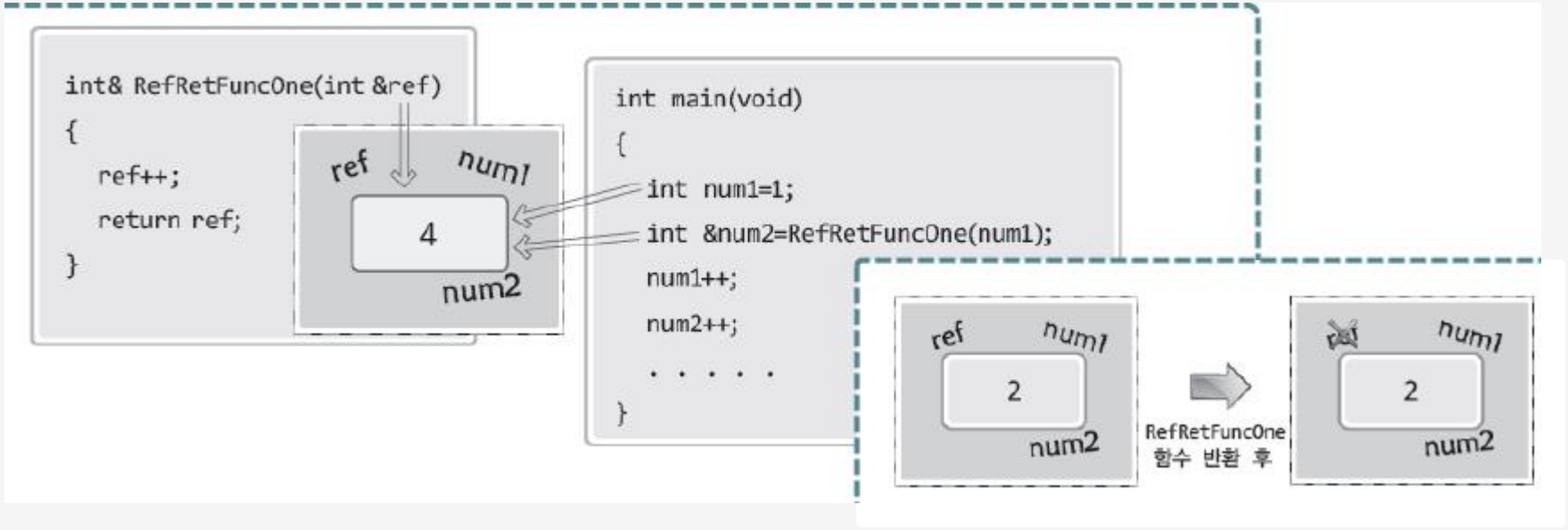
```
void HappyFunc(const int &ref) { . . . . }
```

함수 HappyFunc 내에서 reference ref 를 이용한 값의 변경은 허용하지 않겠다 ! 라는 의미 !

함수 내에서 reference를 통한 값의 변경을 진행하지 않을 경우 reference를 const 로 선언해서 , 다음 두 가지 장점을 얻도록 하자 !

1. 함수의 원형 선언만 봐도 값의 변경이 일어나지 않음을 판단할 수 있다 .
2. 실수로 인한 값의 변경이 일어나지 않는다 .

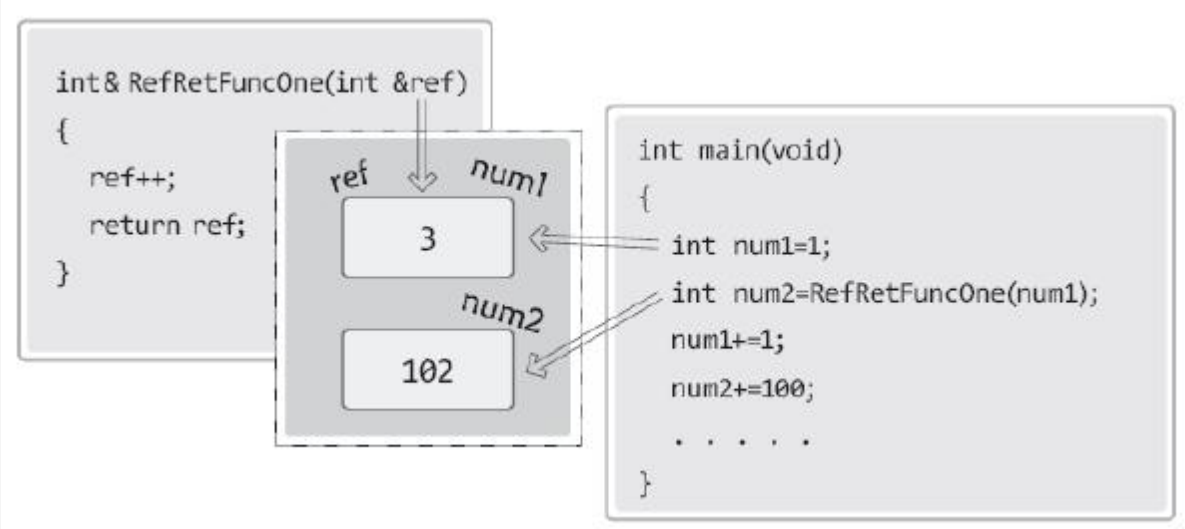
# C/C++ 반환형이 참조이고 반환도 참조로 받는 경우




반환의 과정에서 일어나는 일은 다음의 경우와 같다 .

```
int num1=1;
int &ref=num1;      // 인자의 전달과정에서 일어난 일
int &num2=ref;      // 함수의 반환과 반환 값의 저장에서 일어난 일
```

# C/C++ 반환형은 참조이되 반환은 변수로 받는 경우




반환의 과정에서 일어나는 일은 다음의 경우와 같다 .

```
int num1=1;
int &ref=num1;      // 인자의 전달과정에서 일어난 일
int num2=ref;       // 함수의 반환과 반환 값의 저장에서 일어난 일
```

# 참조를 대상으로 값을 반환하는 경우

```
int RefRetFuncTwo(int &ref)
{
    ref++;
    return ref;
}
```

```
int main(void)
{
    int num1=1;
    int num2=RefRetFuncTwo(num1);
    num1+=1;
    num2+=100;
    cout<<"num1: "<<num1<<endl;
    cout<<"num2: "<<num2<<endl;
    return 0;
}
```

reference를 반환하건 , 변수에 저장된 값을 반환하건 , 반환형이 참조형이 아니라면 차이는 없다 ! 어차피 reference가 참조하는 값이나 변수에 저장된 값이 반환되므로 !

- `int num2=RefRetFuncOne(num1);` (○)
- `int &num2=RefRetFuncOne(num1);` (○)

반환형이 참조형인 경우에는 반환되는 대상을 reference로 그리고 변수로 받을 수 있다 .

- `int num2=RefRetFuncTwo(num1);` (○)
- `int &num2=RefRetFuncTwo(num1);` (×)

그러나 반환형이 값의 형태라면 , reference로 그 값을 받을 수 없다 !

# 잘못된 참조의 반환

```
int& RetuRefFunc(int n)
{
    int num=20;
    num+=n;
    return num;
}
```

이와 같이 지역변수를 참조의 형태로 반환하는 것은 문제의 소지가 된다 . 따라서 이러한 형태로는 함수를 정의하면 안 된다 .

에러의 원인 ! ref 가 참조하는 대상이 소멸된다 !

```
int &ref=RetuRefFunc(10);
```

# C/C++ const reference의 또 다른 특징

```
const int num=20;
int &ref=num;
ref+=10;
cout<<num<<endl;
```

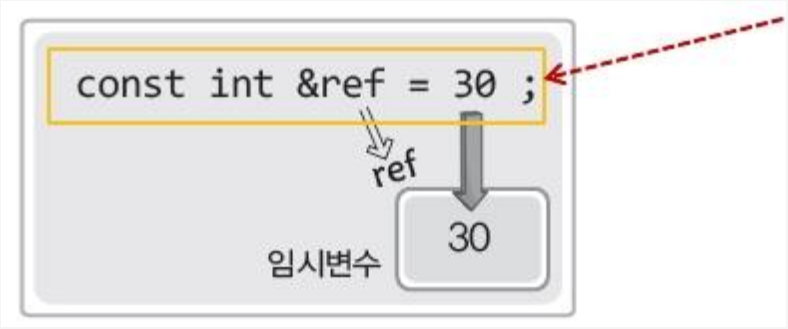
에러의 원인 ! 이를 허용한다는 것은 ref 를 통한 값의 변경을 허용한다는 뜻이 되고 , 이는 num 을 const 로 선언하는 이유를 잃게 만드는 결과이므로 !

해결책 !

```
const int num=20;
const int &ref=num;
const int &ref=50;
```

따라서 한번 const 선언이 들어가기 시작하면 관련해서 몇몇 변수들이 const 로 선언되어야 하는데 , 이는 프로그램의 안정성을 높이는 결과로 이어지기 때문에 , const 선언을 빈번히 하는 것은 좋은 습관이라 할 수 있다 .

# C/C++ reference의 상수 참조




const reference는 상수를 참조할 수 있다 .
이유는 ,
이렇듯 , 상수를 const reference로 참조할 경우 , 상수를 메모리 공간에 임시적으로 저장하기 때문이다 ! 즉 , 행을 바꿔도 소멸시키지 않는다 .

이러한 것이 가능하도록 한 이유 !

```
int Adder(const int &num1, const int &num2)
{
    return num1+num2;
}
```

이렇듯 매개변수 형이 reference인 경우에 상수를 전달할 수 있도록 하기 위함이 바로 이유이다 !