6장 배열, 포인터, 문자열

박 종 혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

목 차

1 차원 배열

- **배열 : 첨자가 붙은 변수를 사용하고 여러 개의 동질적 값을 표현할 수 있는 자료형**
- **예 (성적처리를 위한 변수 선언)**

```
int     grade0, grade1, grade2;
int     grade[3];
```

- **1차원 배열 선언**

```
int    a[size];
```

  - lower bound = 0
  - upper bound = size - 1
  - size = upper bound + 1

1 차원 배열

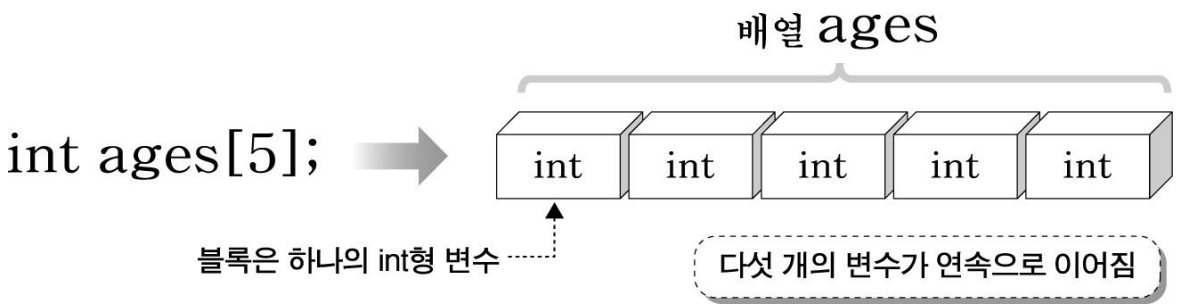
- **사용 예**

```
#define    N    100
int   a[N];
for (i = 0; i < N; ++i)
    sum += a[i];
```

1 차원 배열

- 배열은 배열명과 변수의 개수, 변수의 자료형으로 선언한다.

- **배열을 선언하면 변수의 개수만큼 연속된 기억공간을 할당한다.**

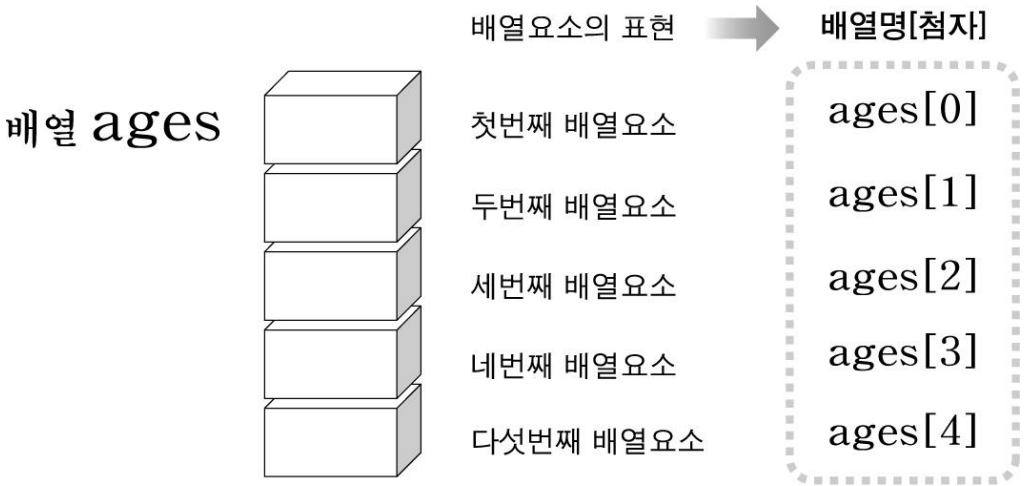


1 차원 배열

- 배열의 기억공간을 사용할 때는 각 기억공간이 배열에서 차지하는 위치를 사용한다.

기억공간의 사용 → 배열명 + 배열에서의 위치

- **배열을 구성하는 기억공간들을 배열의 요소(element)라고 하며 각 요소가 배열에서 차지하는 위치를 첨자(index)라고 한다.**
- **배열의 첨자는 0부터 시작한다.**



배열의 초기화

- 배열은 자동, 외부, 정적 기억영역 클래스는 될 수 있지만, 레지스터는 될 수 없음
- 전통적인 C에서는 외부와 정적 배열만 배열 초기자를 사용하여 초기화할 수 있음
- ANSI C에서는 자동 배열도 초기화될 수 있음

배열의 초기화

- **초기화 예제**

```
float   f[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

  - `f[0] = 0.0, f[1] = 1.0, . . .`

- **초기자 목록이 초기화되는 배열 원소 개수보다 적다면, 나머지 원소들은 0으로 초기화됨**

  int a[100] = {10} ;

  → **a[0] = 10, a[1] = 0, a[2] = 0, …., a[99] = 0**

  Cf) 배열 초기화 목록의 원소 개수가 배열 크기보다 많을 경우 → 오류로 간주

  int x[3] = {1, 3, 5, 7, 9}

배열의 초기화

- **외부와 정적 배열이 명시적으로 초기화되지 않았다면, 시스템은 디폴트로 모든 원소를 0으로 초기화함**

배열의 초기화

- **배열의 크기가 기술되어 있지 않고 일련의 값으로 초기화되도록 선언되어 있다면, 초기자의 개수가 배열의 암시적인 크기가 됨**

```
int a[] = {2, 3, 5, -7};
```

```
int a[4] = {2, 3, 5, -7};
```

  - 따라서, 위의 두 선언문은 같은 선언문임

배열의 초기화

- **문자열에서는 주의를 요함**

```
char    s[] = "abc";
```

  - 이 선언문은 다음과 같음

```
char    s[] = {'a', 'b', 'c', '\0'};
```

  - 즉, s 배열의 크기는 3이 아니라 4임
  - 문자열을 저장할때 끝을 표시하는 널문자('\0')를 넣어줘야 함.

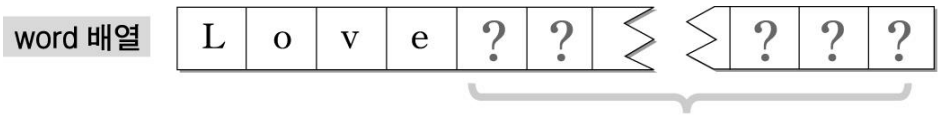
배열의 초기화

- 문자배열의 마지막에 널문자('\0')가 없으면 저장된 문자열의 끝을 알 수 없다.

  - printf함수로 출력하는 경우 쓰레기 값이 출력된다.

```
char word[50];        // 넉넉한 배열 선언
word[0]='L';          // 문자를 하나씩 입력하여 문자열 완성
word[1]='o';
word[2]='v';
word[3]='e';
printf("%s", word);  // 문자열 출력
```

Love徼 徼 徼 徼 徼 徼 徼 徼 徼 徼 徼 徼 徼 徼 徼 徼 徼

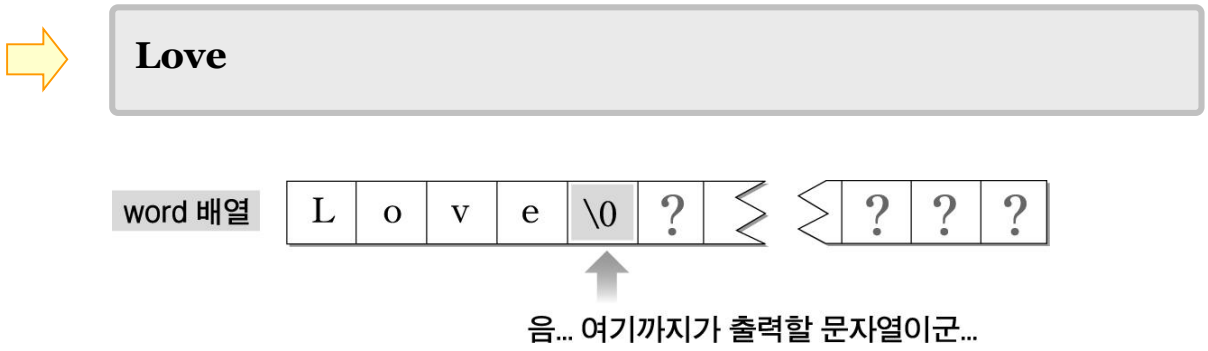


음... 여기에 있는 것도 출력할 문자들인가?

배열의 초기화

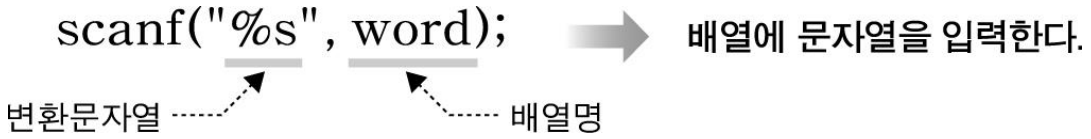
- 문자배열에 문자열을 저장할 때는 항상 마지막에 끝을 표시하는 널문자('\0')를 넣어줘야 한다.

```
char word[50];
word[0]='L';
word[1]='o';
word[2]='v';
word[3]='e';
word[4]='\0';          // 문자열의 끝을 널문자로 표시해 준다.
printf("%s", word);
```



▶ scanf함수를 사용한 문자열의 입력

- 문자배열에 문자열을 입력 받을 때는 %s변환문자열과 배열명을 scanf함수의 전달인자로 준다.

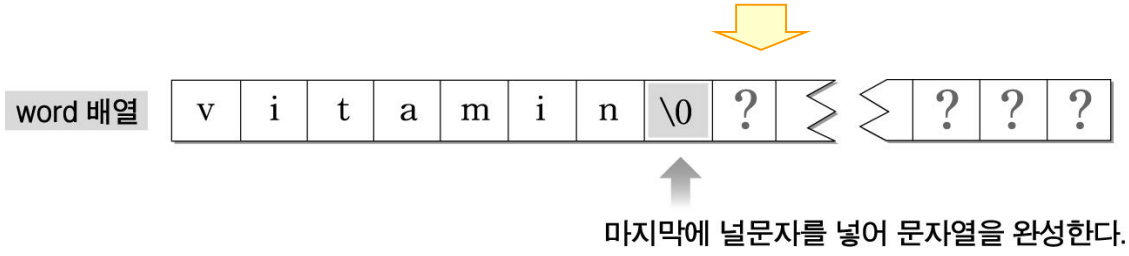


- **scanf함수로 문자열을 입력 받으면 널문자를 자동으로 채워준다.**

- word배열에 vitamin을 입력 받은 경우

```
printf("문자열을 입력하세요 : ");
scanf("%s", word);
```

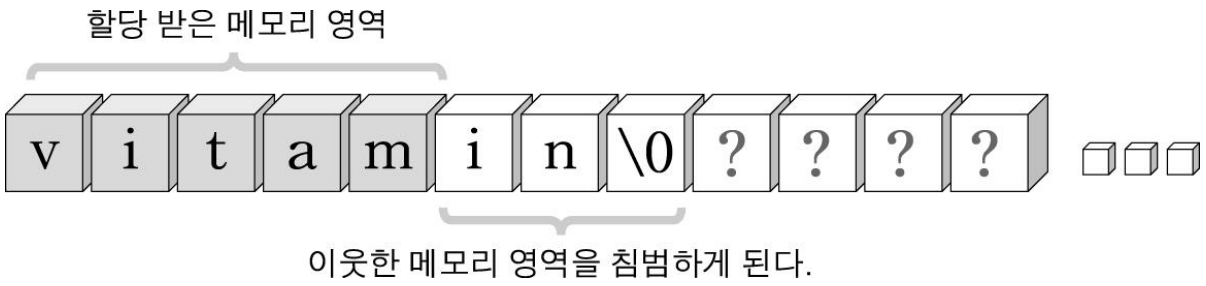
문자열을 입력하세요 : vitamin (엔터)



▶ scanf함수로 문자열을 입력할 때 주의할 점

- 배열의 크기보다 입력되는 문자열의 크기가 더 크면 할당되지 않은 기억공간을 침범하므로 주의해야 한다.

char word[5]; **// 이곳에 "vitamin"을 입력 받는다면...**



첨자

- **a가 배열이면, a의 원소를 접근할 때 a[expr]와 같이 씀**
  - 여기서, expr은 정수적 수식이고,
  - expr을 a의 첨자, 또는 색인이라고 함

첨자

- **사용 예**

```
int     i, a[N];
```

  - 여기서 N은 기호 상수이고,
  - 하나의 배열 원소를 참조하기 위해서는 첨자 i에 적당한 값을 할당한 후 a[i] 수식을 사용하면 됨
  - 이때, i는 0보다 크거나 같고 N - 1보다 작거나 같아야 함
  - i가 이 범위를 벗어나는 값을 갖는다면, a[i]를 접근할 때 실행시간 오류가 발생함

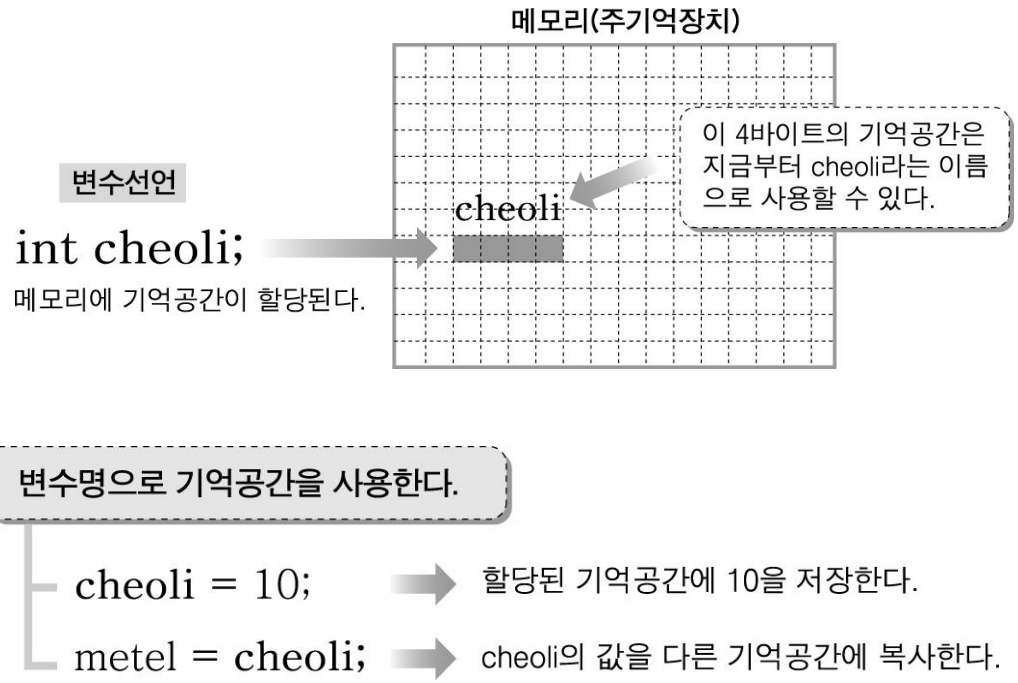
포인터

- **프로그램에서 메모리를 접근하고 주소를 다루기 위해 사용**
- **주소 연산자 &**
  - v가 변수라면, &v는 이 변수의 값이 저장된 메모리 위치, 또는 주소임
- **포인터 변수**
  - 값으로 주소를 갖는 변수
  - 선언 방법

    int *p;
  - 즉, 변수 이름 앞에 *를 붙여서 선언함

포인터

- 변수를 선언하는 것은 메모리에 기억공간을 할당하는 것이며 할당된 이후에는 변수명으로 그 기억공간을 사용한다.



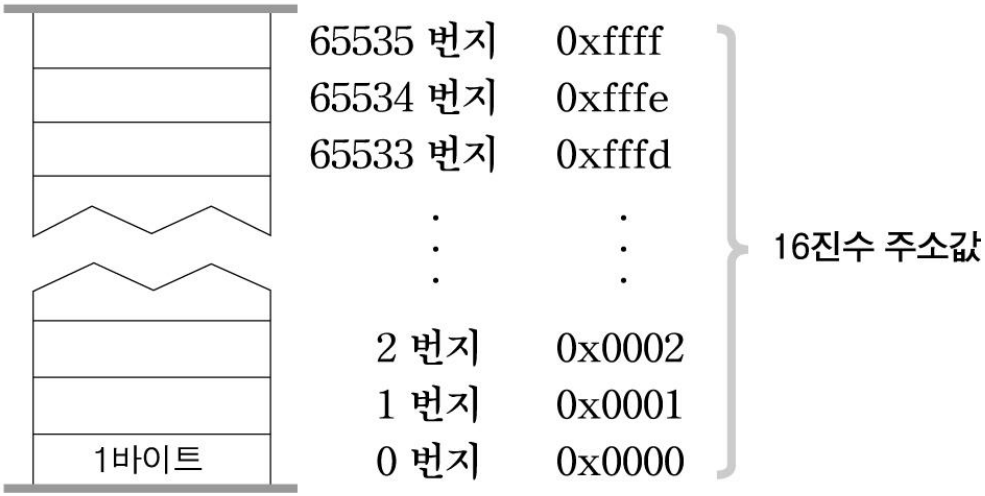
- **할당된 기억공간을 사용하는 방법에는 변수명 외에 메모리의 실제 주소값을 사용하는 것이다. 이 주소값을 포인터라고 한다.**

포인터

- 메모리에는 바이트(byte)단위로 그 위치를 식별할 수 있는 물리적인 주소값이 있다.
  - 메모리의 용량이 64kb라면 주소값은 0번지부터 65535번지까지 존재한다.

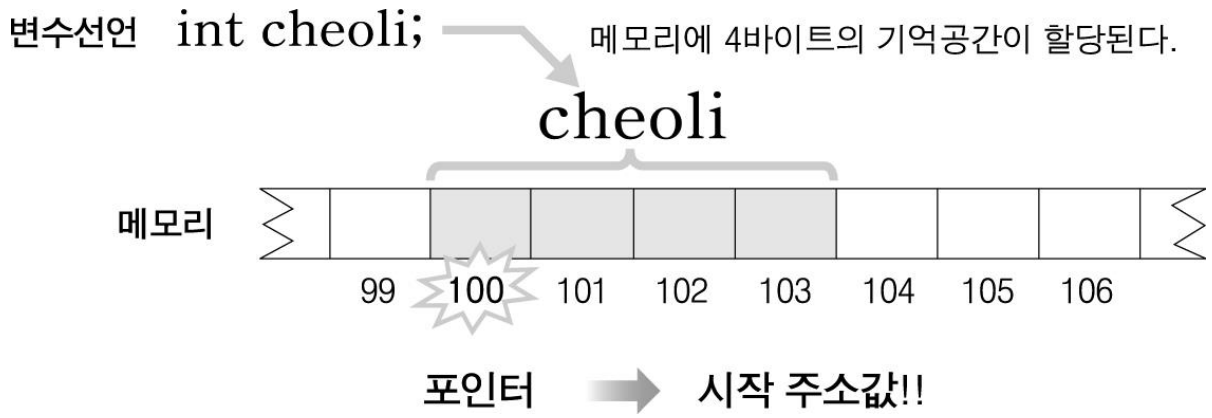
$$64\ \text{kbyte} = 64 \times 1024 = 65536\ \text{byte}\ (1\text{k} = 2^{10} = 1024)$$

메모리에는 바이트 단위로 위치를 식별할 수 있는 주소값이 있다.



포인터

- 변수를 선언하면 그 자료형의 크기만큼 메모리에 연속된 바이트의 기억공간이 할당되는데 그 첫번째 바이트의 주소값이 포인터이다.



- 이 포인터를 사용하여 4바이트의 기억공간에 값을 저장하거나 저장된 값을 꺼내어 쓸 수 있다.

포인터 변수

- 포인터의 유효한 값의 범위
  - 특정주소 0
  - 주어진 C 시스템에서 기계 주소로 해석될 수 있는 양의 정수 집합

포인터 변수

- **올바른 예제**

```
p = 0;
p = NULL;
p = &i;
p = (int *) 1776;
    /* an absolute address in memory */
```

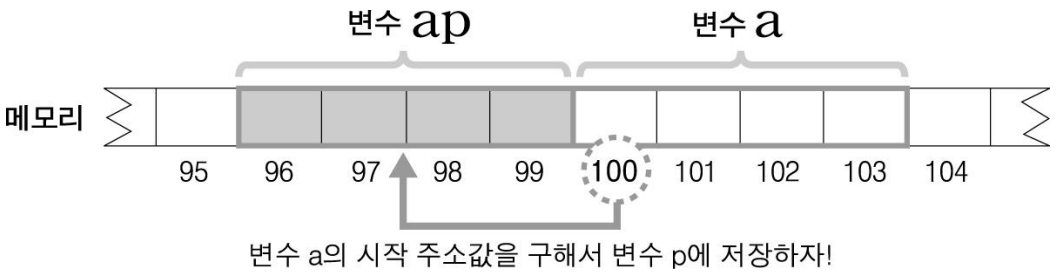
- **잘못된 예제**

```
p = &3;
p = &(i + 99);
p = &v                          /* register v; */
```

포인터 변수

- 포인터의 값 자체는 정수값이지만 가리키는 자료형에 대한 정보를 가지고 있으므로 정수형 변수에 저장할 수 없다.

```
int a;          // 포인터를 구할 변수
int ap;         // 포인터를 저장할 변수
ap = &a;        // a의 포인터를 구해서 ap에 저장한다.
```



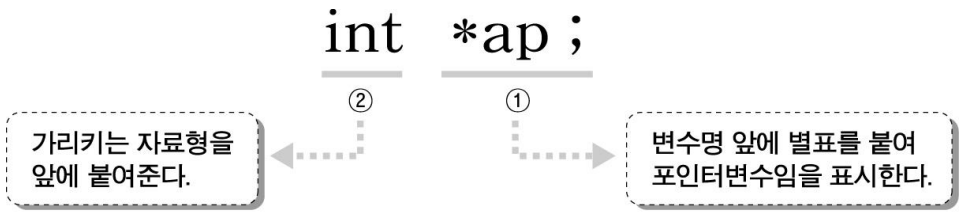
- 직관적으로는 충분히 가능할 듯 하지만 컴파일에러가 발생한다.

```
error C2440: '=' : cannot convert from 'int *' to 'int'
```

포인터 변수

- 포인터는 포인터가 가진 정보를 그대로 보존할 수 있도록 포인터변수에 저장해야 한다.

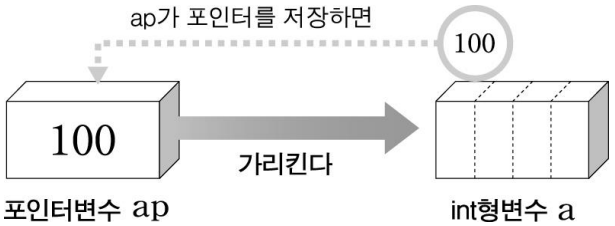
- 포인터변수는 변수명 앞에 '*'을 붙이고 가리키는 자료형을 앞에 적어준다.
  - int형 변수의 포인터를 저장하는 포인터변수의 선언



(제 이름은 ap이고요 포인터변수입니다. 저는 int형 변수의 시작주소값만을 저장할 수 있습니다. 그래서 int형 변수를 가리킨다고 말하지요.)

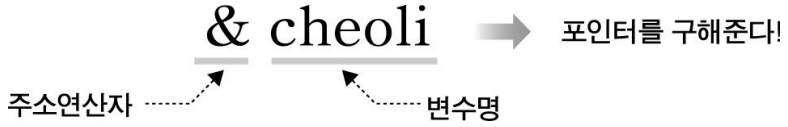
- 포인터변수가 포인터를 저장하면 포인터와 마찬가지로 기억공간을 가리킨다.

```
int a;
int *ap;
ap = &a;
```



주소 연산자 &

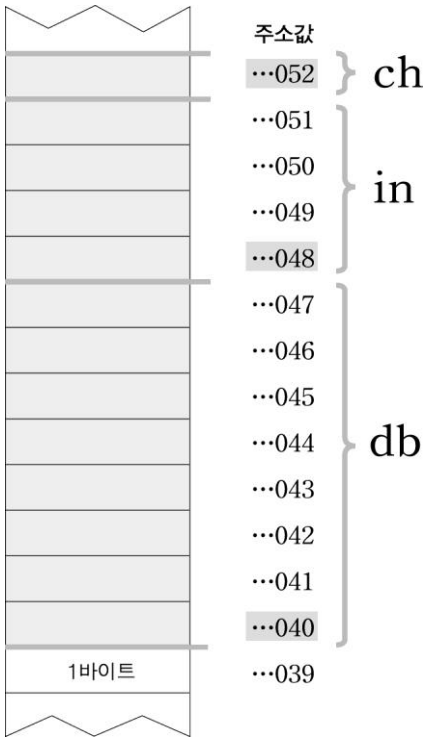
- 특정 변수의 포인터를 구하기 위해서는 주소연산자(&)를 사용한다.



- 포인터를 구하여 출력해 보자.

```
char ch;
int in;
double db;

printf("ch의 포인터 : %u\n", &ch);
printf("in의 포인터 : %u\n", &in);
printf("db의 포인터 : %u\n", &db);
```

```
ch의 포인터 : 1245052   // char형 변수의 주소값
in의 포인터 : 1245048   // int형 변수의 시작 주소값
db의 포인터 : 1245040   // double형 변수의 시작 주소값
```



역참조 연산자 *

- 간접지정 연산자라고도 함
- 단항 연산자, 우에서 좌로의 결합 법칙
- p가 포인터라면, *p는 p가 주소인 변수의 값을 나타냄

역참조 연산자 *

- 포인터를 통해서 기억공간을 사용하기 위해서는 참조연산자(*)를 사용한다.


```
char ch;
int in;
double db;
*&ch = 'P';        // 포인터 &ch가 가리키는 기억공간에 'P'를 저장한다.
*&in = 100;        // 포인터 &in이 가리키는 기억공간에 100을 저장한다.
*&db = 3.14;       // 포인터 &db가 가리키는 기억공간에 3.14를 저장한다.
printf("변수 ch에 저장된 문자 : %c\n", ch);
printf("변수 in에 저장된 값 : %d\n", in);
printf("변수 db에 저장된 값 : %lf\n", db);
```

```
변수 ch에 저장된 문자 : P
변수 in에 저장된 값 : 100
변수 db에 저장된 값 : 3.140000
```

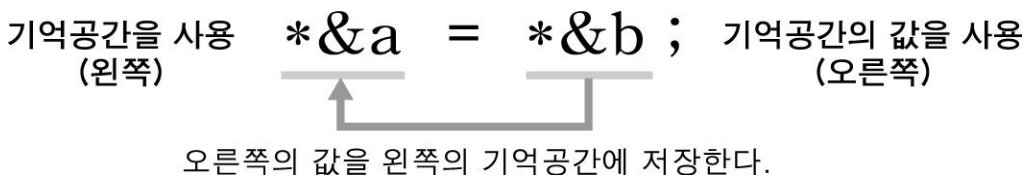
역참조 연산자 *

- “참조”는 기억공간 뿐만 아니라 기억공간에 저장된 값도 사용한다.

```
int a=100, b=0;
b = *&a;      // 포인터 &a가 가리키는 기억공간의 값을 b에 대입한다.
printf(“b의 값 : %d\n”, b);
```

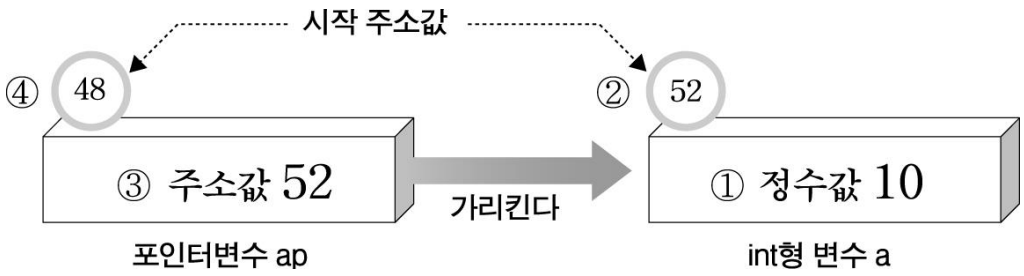
- 기억공간을 사용하는 것과 값을 사용하는 것은 대입연산자의 어디에 위치하느냐에 따라 결정된다.

```
int a=10, b=20;
*&a = *&b;          // 변수 b에 저장된 값을 변수 a의 기억공간에 저장한다.
printf(“a의 값 : %d\n”, a);      // a의 값은 20이 출력된다.
```



포인터 정리

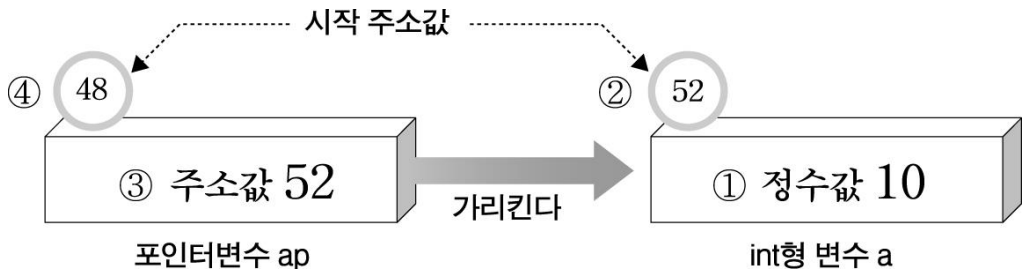
```
int a = 10;       // int형 변수 선언, 정수값 10으로 초기화
int *ap = &a;     // int 포인터변수 선언, a의 시작주소값으로 초기화
```




```
printf("%d", a);
printf("%d", *ap);
printf("%u", &a);
printf("%u", ap);
printf("%u", &ap);
```

포인터 정리

```
int a = 10;       // int형 변수 선언, 정수값 10으로 초기화
int *ap = &a;     // int 포인터변수 선언, a의 시작주소값으로 초기화
```



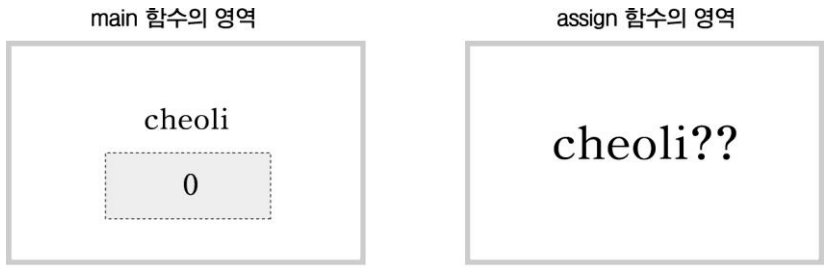

```
printf("%d", a);       // ①번 출력, a에 저장된 정수값 10
printf("%d", *ap);     // ①번 출력, ap가 가리키는 곳에 저장된 값 10
printf("%u", &a);      // ②번 출력, a의 시작주소값 52번지
printf("%u", ap);      // ③번 출력, ap에 저장된 주소값 52번지
printf("%u", &ap);     // ④번 출력, 포인터변수 ap의 시작주소값 48번지
```

포인터의 필요성

- 함수들은 독립된 기억공간을 가지므로 다른 함수에 선언된 변수를 사용할 수 없다.

  - assign함수를 호출하여 메인함수에 있는 cheoli변수에 값을 할당하는 예

```
#include <stdio.h>

void assign();

int main()
{
   int cheoli=0;
   assign();
   printf("함수가 호출된 후에 cheoli에 저장된 값 : %d\n", cheoli);
   return 0;
}

void assign()
{
   cheoli=100;
}
```



하나의 함수는 독립된 메모리 영역을 사용한다.

함수가 호출된 후에 cheoli에 저장된 값 : 0

포인터의 필요성

- **assign함수가 main함수의 cheoli변수를 사용하기 위해서는 메모리에서의 위치(포인터)를 알아야 한다.**

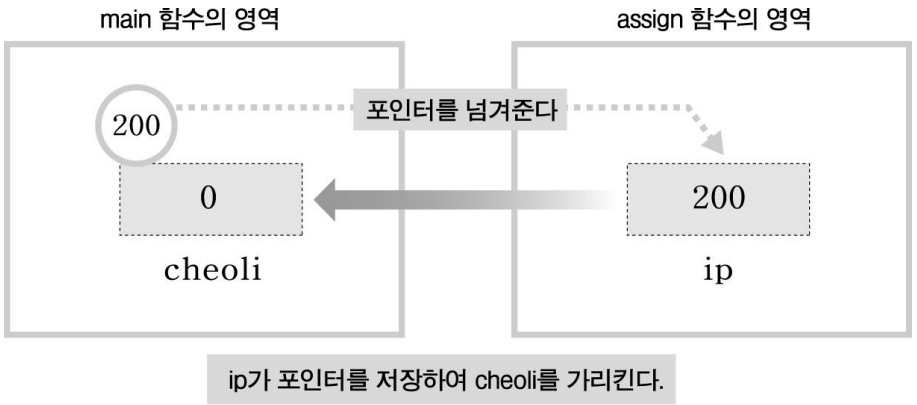
```
#include <stdio.h>

void assign(int *ip);

int main()
{
   int cheoli=0;
   assign(&cheoli);
   printf("함수가 호출된 후에 cheoli에 저장된 값 : %d\n", cheoli);  // 100출력
   return 0;
}

void assign(int *ip)
{
   *ip=100;
}
```

main 함수에서 호출할 때

```
assign( &cheoli );      // 포인터를 구해서 전달인자로 넘겨준다.
```

assign 함수를 선언할 때

```
void assign( int *ip );      // 포인터변수를 선언하여 포인터를 받는다.
```



ip가 포인터를 저장하여 cheoli를 가리킨다.

(main 함수에 있는 cheoli 변수의 시작주소값은 200이라고 가정합니다.)

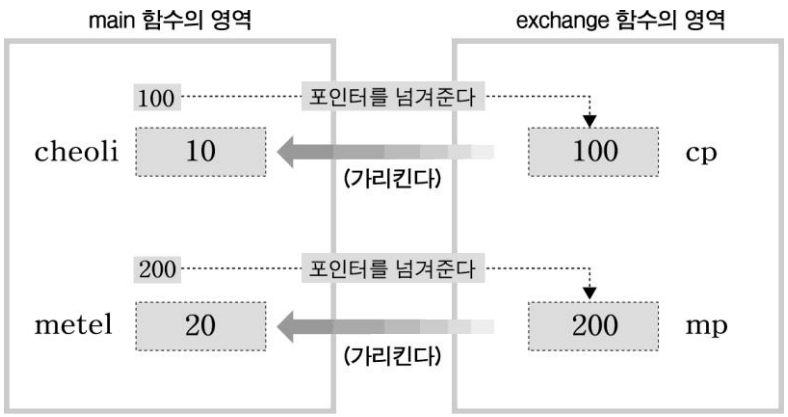
포인터의 필요성

- 함수는 전달인자가 많아도 리턴되는 값은 오직 하나이다. 따라서 메인함수에 있는 두 변수의 값을 바꾸는 함수는 포인터를 사용해야 한다.

```
#include <stdio.h>

void exchange(int *, int *);

int main()
{
   int cheoli=10, metel=20;
   exchange(&cheoli, &metel); //  [1]
   return 0;
}

void exchange(int *cp, int *mp)
{
   int temp;
   temp=*cp;   // [2]
   *cp=*mp;    // [3]
   *mp=temp;  // [4]
}
```

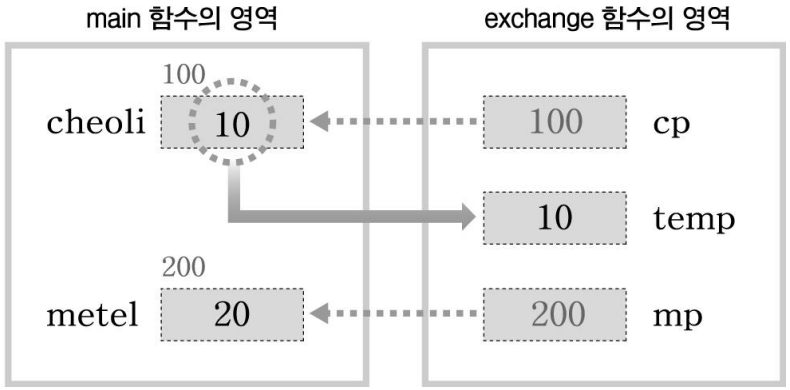
포인터의 필요성

[1]



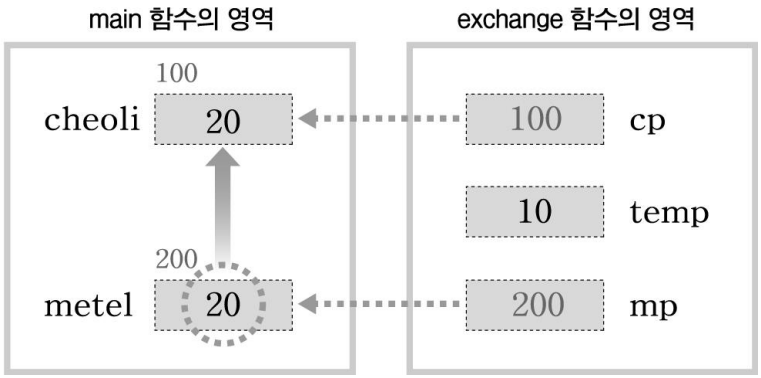
[2]

```
temp = *cp;
```



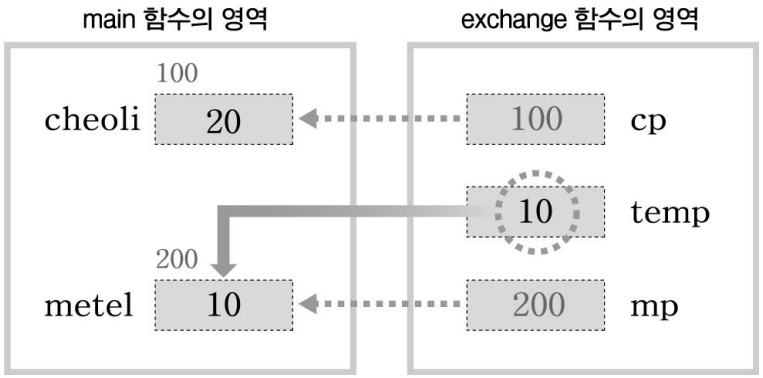
[3]

```
*cp = *mp;
```



[4]

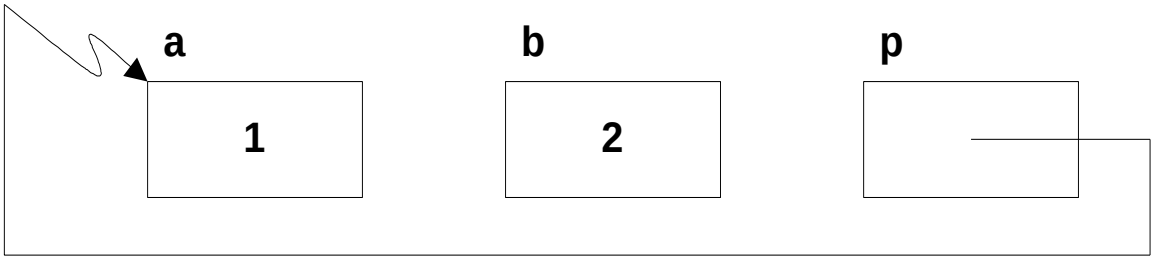
```
*mp = temp;
```



포인터 예제

int a = 1, b = 2, *p;

p = &a; // p에 a주소를 배정



b = *p; /* b에 p가 포인트하는 값을 배정, b = a; */

포인터 예제

```
#include   <stdio.h>
int main(void){
    int   i = 7, *p = &i;
    printf("%s%d\n%s%p\n",
     "     Value of i: ", *p, //그주소에 저장된 값
     "  Location of i: ", p); //주소 또는 위치
    return 0;
}
```

- **출력**

```
     Value of i: 7
  Location of i: effffb24
```

포인터 예제

선언 및 초기화		
int i = 3, j =5, *p = &i, *q = &j, *r; double x;		
수 식	등가 수식	값
p == & i	p == (& i)	1
* * & p	* (* (& p))	3
r = & x	r = (& x)	/* illegal */
7 * * p / * q + 7	(((7 * (* p))) / (* q)) + 7	11
* (r = & j) *= *p	(* (r = (& j))) *= (* p)	15

(주의) 7 * * p /* q + 7 ???

포인터 예제

선언	
int *p; float *q; void *v;	
올바른 배정문	잘못된 배정문
p = 0; p = (int *) 1; p = v = q; p = (int *) q;	p = 1; v = 1; p = q;

포인터변수를 사용한 참조

- 포인터를 저장한 포인터변수도 참조연산자로 그 것이 가리키는 기억공간 또는 그 기억공간의 값을 사용할 수 있다.

```
int a;              // int형 변수의 선언
int ap = &a;        // 포인터변수의 선언과 동시에 초기화, ap는 변수 a를 가리킨다.
*ap = 10;           // 포인터변수가 가리키는 기억공간에 10을 저장한다.
```

- 포인터변수도 하나의 변수이므로 주소연산자로 메모리에서의 위치를 구할 수 있다.
- 다음 출력값은 ? (a, ap의 시작주소는 각각 25, 48이라고 가정)

```
int a;
int ap = &a;
printf("ap에 저장된 값 : %u\n", ap);
printf("ap자체의 주소값 : %u\n", &ap);
```

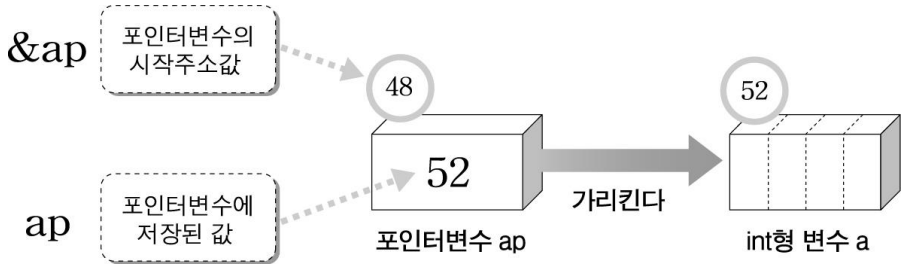
포인터변수를 사용한 참조

- 포인터를 저장한 포인터변수도 참조연산자로 그 것이 가리키는 기억공간 또는 그 기억공간의 값을 사용할 수 있다.

```
int a;              // int형 변수의 선언
int ap = &a;        // 포인터변수의 선언과 동시에 초기화, ap는 변수 a를 가리킨다.
*ap = 10;           // 포인터변수가 가리키는 기억공간에 10을 저장한다.
```

- 포인터변수도 하나의 변수이므로 주소연산자로 메모리에서의 위치를 구할 수 있다.

```
int a;              // int형 변수의 선언
int ap = &a;        // 포인터변수의 선언과 동시에 초기화, ap는 변수 a를 가리킨다.
printf("ap에 저장된 값 : %u\n", ap);      // 변수 a의 시작주소값 출력
printf("ap자체의 주소값 : %u\n", &ap);  // 포인터변수 ap의 시작주소값 출력
```



참조에 의한 호출

- C는 기본적으로 "값에 의한 호출" 메커니즘 사용
- "참조에 의한 호출"의 효과를 얻기 위해서는 함수 정의의 매개변수 목록에서 포인터를 사용해야 함
- 예제 프로그램

```
void swap(int *p, int *q){
   int tmp;
   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

```
void swap(int *, int *);
int main(void){
   int i = 3, j = 5;
   swap(&i, &j);
   printf("%d %d\n", i, j);
      /*  5   3  is  printed
*/
   return 0;
}
```

참조에 의한 호출

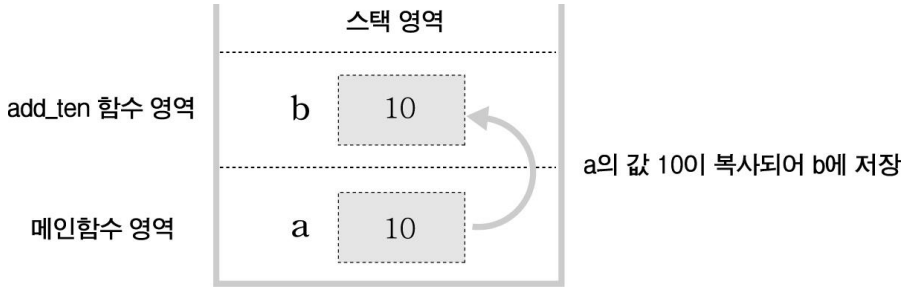
- **"참조에 의한 호출"의 효과를 얻는 방법**

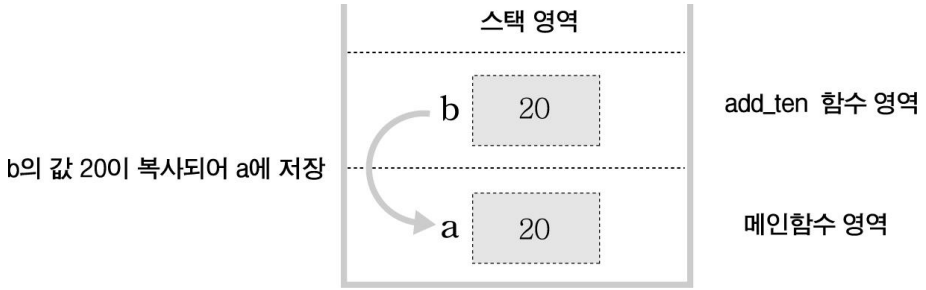
  1. 함수 매개변수를 포인터형으로 선언
  2. 함수 몸체에서 역참조 포인터 사용
  3. 함수를 호출할 때 주소를 인자로 전달

참조에 의한 호출

- 일반적인 함수의 호출 방법으로 호출함수의 전달인자가 피호출함수의 매개변수에 복사된다. 피호출함수는 리턴할 때 리턴값을 복사하여 호출함수로 전달한다.

```
#include <stdio.h>
int add_ten(int);
int main()
{
   int a=10;
   a=add_ten(a);
   printf("a : %d\n", a);
   return 0;
}

int add_ten(int b)
{
   b=b+10;
   return b;
}
```

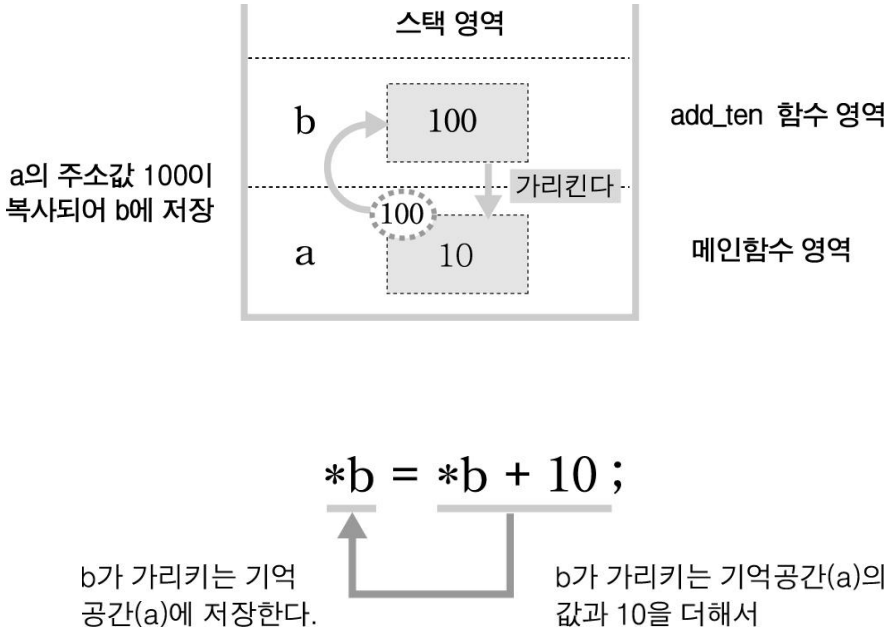




참조에 의한 호출

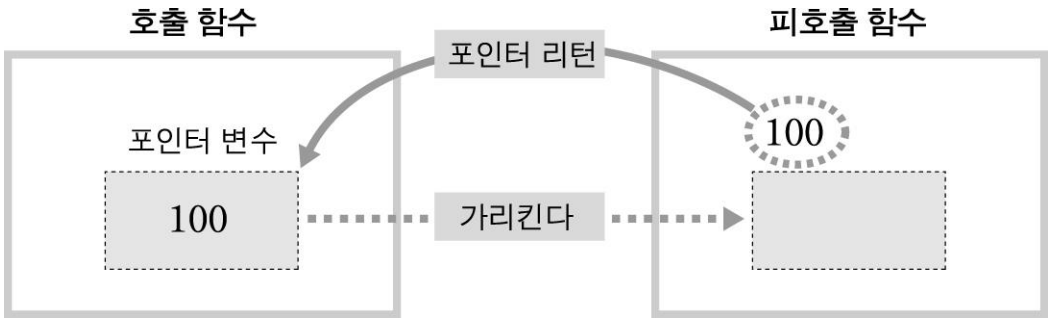
- 호출함수에서 변수의 포인터를 전달인자로 주고 피호출함수에서는 이 포인터를 받아 호출함수의 변수를 참조하는 방식이다.

```
#include <stdio.h>
int add_ten(int *);
int main()
{
   int a=10;
   a=add_ten(&a);
   printf("a : %d\n", a);
   return 0;
}

void add_ten(int *b)
{
   *b=*b+10;
}
```



참조에 의한 호출

- 피호출함수에서 포인터를 리턴하여 호출함수가 피호출함수의 기억공간을 참조할 수 있도록 할 수 있다.



- **포인터를 리턴하는 함수는 리턴값의 형태가 포인터형이 된다.**

  - int형 변수의 포인터를 리턴하는 경우

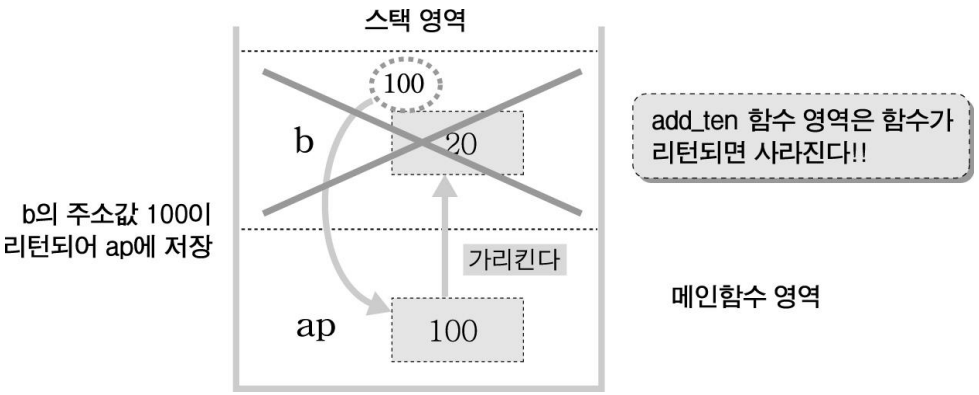
```
int * add_ten(int);
```

int형 기억공간의 포인터를 리턴한다!

참조에 의한 호출

- 자동변수의 포인터를 리턴하여 호출함수에서 다시 참조하는 것은 위험하다.

```
#include <stdio.h>
int *add_ten(int);
int main()
{
   int a=10;
   int *ap;
   ap=add_ten(a);
   printf("a : %d\n", *ap);   // 포인터 변수 ap로 add_ten함수의 변수를 참조한다.
   return 0;
}

int *add_ten(int b)
{
   b=b+10;
   return &b;    // add_ten함수의 자동변수 b의 포인터를 리턴한다.
}
```



참조에 의한 호출

- 포인터를 리턴하는 경우는 함수가 리턴된 후에도 그 기억공간이 계속 유지되는 경우만 가능하다.
  - 호출함수로부터 포인터를 받아서 다시 리턴하는 경우(문자열 처리 함수들의 예)

```
char *strcpy(char *A, char *B); // B의 문자열을 A에 복사하고 A를 리턴한다.
char *strcat(char *A, char *B); // B의 문자열을 A에 붙인 후에 A를 리턴한다.
char *gets(char *A);            // A에 문자열을 입력하고 A를 리턴한다.
```

- 포인터를 리턴하면 좀더 다양한 방식으로 프로그램을 작성할 수 있다.
  - 두 문자열을 붙인 후에 그 결과를 바로 확인하는 예

```
#include <stdio.h>
#include <string.h>
int main()
{
   char src[80]="빈대";
   printf("결과 : %s\n", strcat(src, "떡"));
   return 0;
}
```

배열과 포인터의 관계

- 배열 이름 그 자체는 주소 또는 포인터 값이고, 배열과 포인터에는 둘 다 첨자를 사용할 수 있음
- 포인터 변수는 다른 주소들을 값으로 가질 수 있음
- 반면에 배열 이름은 고정된 주소 또는 포인터임

배열과 포인터의 관계

- **예제**

```
int  * p, * q ;
int  a[4] ;
p = a;              /* p = &a[0]; */
q = a + 3;          /* q = &a[3]; */
```

배열과 포인터의 관계

- **a와 p는 포인터이고 둘 다 첨자를 붙일 수도 있음**

```
a[i] <==> *(a + i)
p[i] <==> *(p + i)
```

- **포인터 변수는 다른 값을 가질 수 있지만, 배열 이름은 안됨**

```
p = a + i ;
a = q ;          /* error */
```

배열과 포인터의 관계

- **예제 코드 (배열의 합 구하기)**

```
#define   N              100
int       * p, a[N], sum ;
```

  - **Version 1**

```
for (i = 0, sum = 0; i < N; ++i)
     sum += a[i] ;    /* 또는  sum += *(a + i) ; */
```

  - **Version 2**

```
for (p = a, sum = 0; p < &a[N]; ++p)
     sum += *p ;
```

  - **Version 3**

```
for (p = a, i = 0, sum = 0; i < N; ++i)
     sum += p[i] ;
```

배열과 포인터의 관계

- 배열의 모든 값을 출력하는 함수를 만들 때 배열요소의 값을 일일이 전달인자로 주는 것은 한계가 있다.

배열의 선언

```
int ary[5] = {10, 20, 30, 40, 50};
```

함수의 호출

```
ary_prn(ary[0], ary[1], ary[2], ary[3], ary[4]);
```

모든 배열요소를 일일이 전달인자로 줘야 한다.

함수의 정의

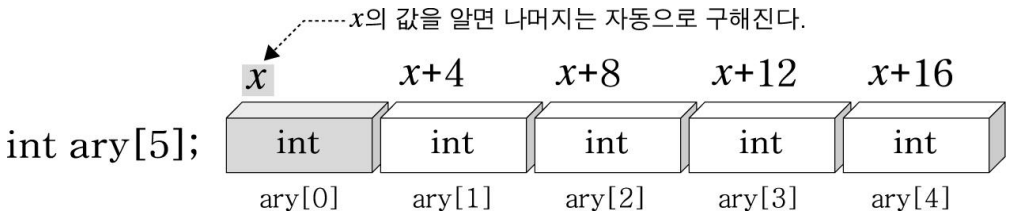
```
void ary_prn(int a, int b, int c, int d, int e)
{
    printf("%d, %d, %d, %d, %d\n", a, b, c, d, e);
}
```

매개변수도 배열요소의 개수만큼 있어야 한다!

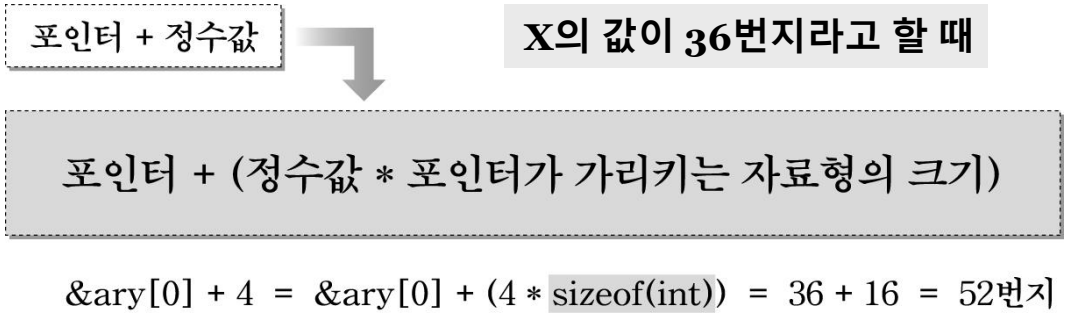
- **포인터를 사용하면 배열요소의 값을 간단히 처리할 수 있다.**

배열과 포인터의 관계

- 배열은 첫번째 배열요소의 포인터만 알면 나머지 배열요소의 포인터도 쉽게 알 수 있다.



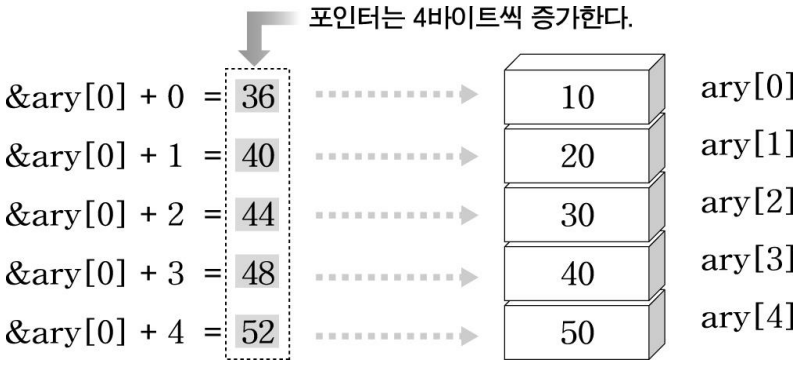
- **포인터에 정수값을 더할 때는 포인터가 가리키는 자료형의 크기를 곱해서 더해준다. 예를 들어 4를 더하면 마지막 배열요소의 포인터가 구해진다.**



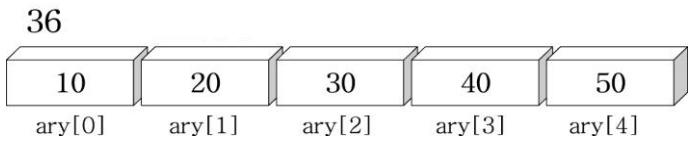
배열과 포인터의 관계

- 모든 배열요소의 포인터는 첫번째 배열요소의 포인터에 정수값을 차례로 더하면 구해진다.

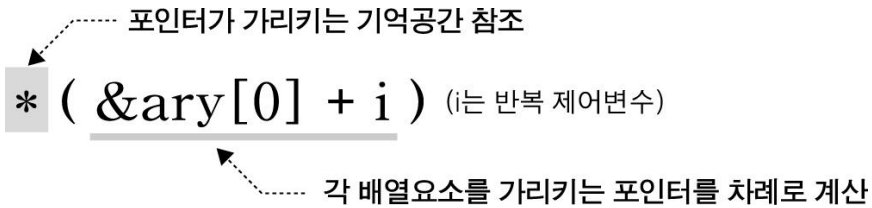
int ary[5] = {10, 20, 30, 40, 50};



첫번째 배열요소의 포인터 값이 36번지 일 때



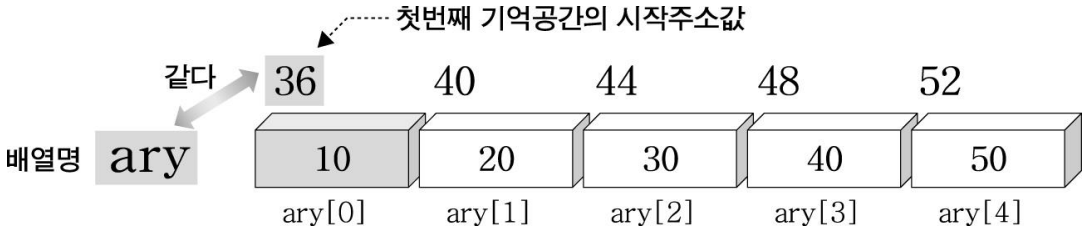
- **각 배열요소의 포인터에 참조연산자를 사용하면 모든 값을 참조할 수 있다.**




```
for(i=0; i<5; i++){
   printf("%d\n", *(&ary[0]+i));
}
```

배열과 포인터의 관계

- 배열명은 첫 번째 배열요소를 가리키는 포인터를 기호화한 것이다.



- **따라서 배열명으로 주소값을 계산하여 모든 배열요소를 참조할 수 있으며 의미상 이해하기 쉽게 배열표현을 주로 사용하는 것이다.**

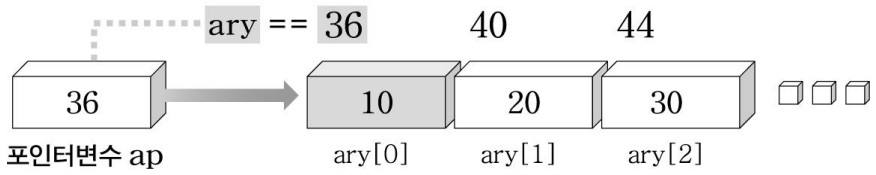
```
for(i=0; i<5; i++){
    printf("%d\n", *(ary+i));
}
```

배열표현		포인터표현
ary[0]		*(ary+0)
ary[1]		*(ary+1)
ary[2]	= =	*(ary+2)
ary[3]		*(ary+3)
ary[4]		*(ary+4)

배열과 포인터의 관계

- 배열명을 포인터변수에 저장하면 포인터변수도 배열명처럼 사용할 수 있다. 이 때 포인터변수는 첫번째 배열요소를 가리킨다.

```
int *ap = ary;  // 배열명을 포인터변수에 저장
```




```
*( ap + 1 )  ➡ 40번지에 있는 배열요소를 참조한다.
```

36 + (1 * sizeof(int)) = 36 + 4 = 40번지

```
int ary[5]={10,20,30,40,50};
int *ap=ary;
int i;

for(i=0; i<5; i++){
   printf("%5d", *(ap+i));  // ap[i]도 가능
}
```

배열표현		포인터표현
ap[0]	= =	*(ap+0)
ap[1]		*(ap+1)
ap[2]		*(ap+2)
ap[3]		*(ap+3)
ap[4]		*(ap+4)

배열과 포인터의 관계

- 포인터(변수)로 배열요소를 참조하는 방법은 다음과 같다.

① 배열명을 사용한 배열표현
② 배열명을 사용한 포인터표현
③ 배열명을 저장한 포인터변수를 사용한 포인터표현
④ 배열명을 저장한 포인터변수를 사용한 배열표현

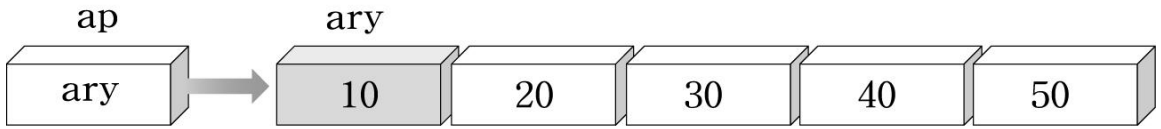
```
int ary[5] = {10, 20, 30, 40, 50};
int *ap=ary;

// 위 4가지 형태로 표현하면?
```

배열과 포인터의 관계

- 포인터(변수)로 배열요소를 참조하는 방법은 다음과 같다.

① 배열명을 사용한 배열표현
② 배열명을 사용한 포인터표현
③ 배열명을 저장한 포인터변수를 사용한 포인터표현
④ 배열명을 저장한 포인터변수를 사용한 배열표현

```
int ary[5] = {10, 20, 30, 40, 50};
int *ap=ary;

// 위 4가지 형태로 표현하면?
```



①	ary[0]	ary[1]	ary[2]	ary[3]	ary[4]
②	*(ary+0)	*(ary+1)	*(ary+2)	*(ary+3)	*(ary+4)
③	*(ap+0)	*(ap+1)	*(ap+2)	*(ap+3)	*(ap+4)
④	ap[0]	ap[1]	ap[2]	ap[3]	ap[4]

4가지 모두 첫번째 배열요소를 참조하는 표현식이다.

배열과 포인터의 관계

- 배열명은 포인터상수이므로 자신의 값을 바꿀 수 없다.

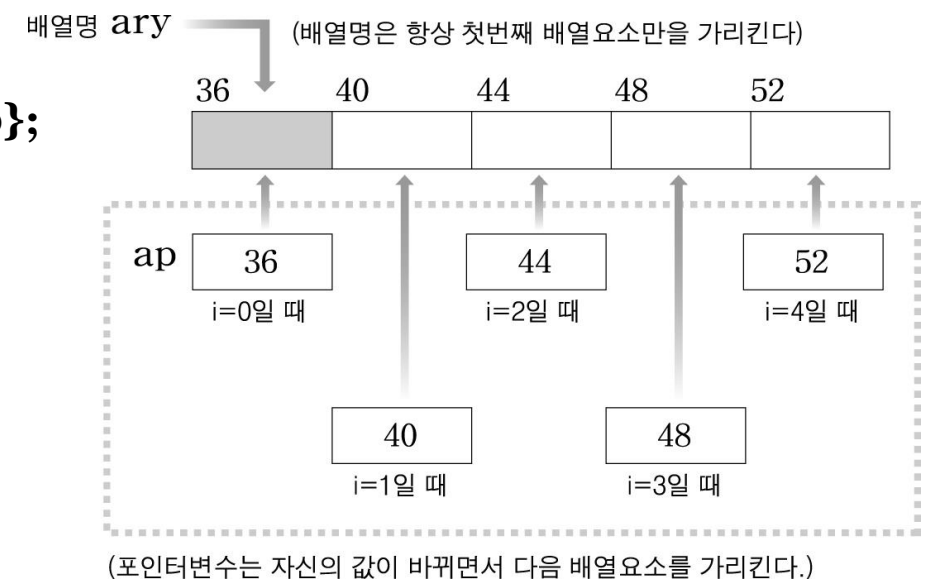
```
int ary[5] = {10, 20, 30, 40, 50};
```

```
ary = ary + 2;
ary++;
```

"배열명은 변수가 아니므로 자신의 값을 바꿀 수 없다"

- **포인터변수는 기억공간이므로 자신의 값을 바꿀 수 있다.**

```
int ary[5]={10,20,30,40,50};
int *ap=ary;
int i;

for(i=0; i<5; i++){
   printf("%5d", *ap);
   ap++;
}
```



(포인터변수는 자신의 값이 바뀌면서 다음 배열요소를 가리킨다.)

배열과 포인터의 관계

- 배열의 모든 요소는 포인터로 참조할 수 있으므로 배열을 처리하는 함수에는 그 시작위치인 배열명을 전달인자로 준다.
- 배열의 값을 출력하는 함수
- 배열에 값을 입력하는 함수
- 배열의 평균을 구하는 함수

배열과 포인터의 관계

- 배열명을 전달인자로 받으므로 매개변수는 포인터변수를 선언한다.

```
#include <stdio.h>
void ary_prn(int *);            // 함수의 선언
int main()
{
   int ary[5]={10,20,30,40,50};  // 배열의 선언과 초기화
   ary_prn(ary);                 // 배열명을 전달인자로 주고 호출한다.
   return 0;
}
void ary_prn(int *ap)            // 배열명을 저장할 포인터변수 선언
{
   int i;
   for(i=0; i<5; i++){
      printf("%5d", ap[i]);      // 포인터변수를 마치 배열명처럼 사용한다.
   }
}
```

배열과 포인터의 관계

- 포인터변수는 배열명이 아니므로 sizeof연산시 포인터변수의 크기만 계산된다. 따라서 포인터변수로 배열요소의 개수를 구할 수 없다.

```
for(i=0; i<sizeof(ap)/sizeof(ap[0]); i++){  // sizeof(ap)는 포인터변수의 크기만 계산
   printf("%5d", ap[i]);
}
```

- **배열의 크기가 바뀌어도 출력할 수 있는 함수를 만들 때는 배열요소의 개수를 전달인자로 받아야 한다.**

호출할 때 → `ary_prn ( ary, sizeof(ary)/sizeof(ary[0]) );`

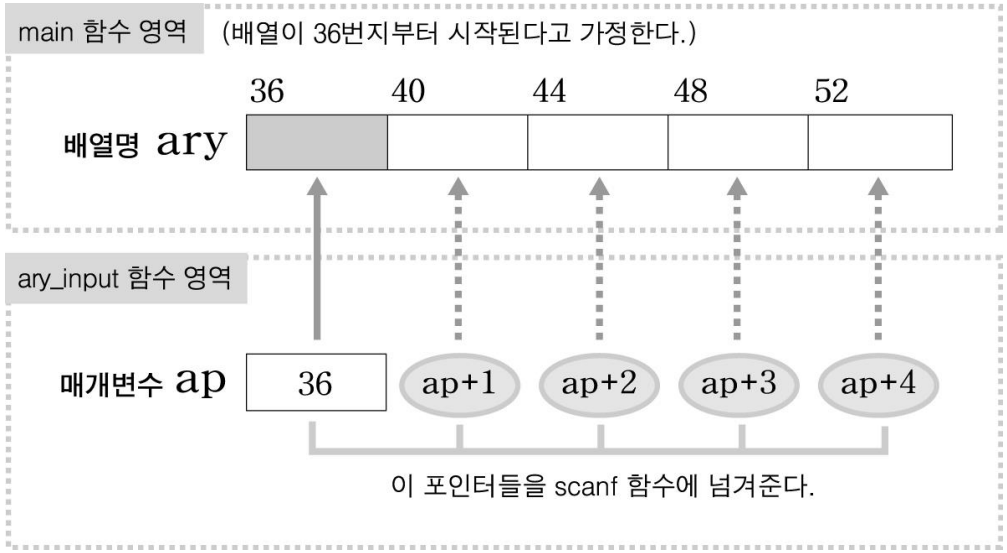
ary: 배열명 / sizeof(ary)/sizeof(ary[0]): 배열요소의 개수

```
void ary_prn(int *ap, int an)          // 배열요소의 개수를 받을 매개변수를 추가한다.
{
   int i;
   for(i=0; i<an; i++){                // 배열요소의 개수만큼 반복한다.
      printf("%5d", ap[i]);
   }
}
```

배열과 포인터의 관계

- 배열에 값을 입력할 때는 scanf함수에 각 배열요소의 포인터만을 전달인자로 준다(즉, 참조연산자를 사용하지 않는다).

```
void ary_input(int *ap)   // 배열명을 저장할 포인터변수 선언
{
   int i;
   for(i=0; i<5; i++){        // 배열요소의 개수만큼 반복한다.
      scanf("%d", ap+i);      // 각 배열요소의 포인터를 구해서 전달인자로 준다.
   }
}
```



배열과 포인터의 관계

- 모든 배열요소의 평균을 구해서 리턴하는 함수를 만들자.

```
#include <stdio.h>
double ary_avg(int *);                  // 함수의 선언
int main()
{
   int ary[5]={75,80,92,88,98};
   double res;                          // 리턴값을 저장할 변수
   res=ary_avg(ary);                    // 전달인자는 배열명, 리턴값은 res에 저장한다.
   printf("배열의 평균은 : %.2lf\n", res);
   return 0;
}
double ary_avg(int *ap)                 // 매개변수는 포인터변수
{
   int i, tot=0;                        // 제어변수와 합을 저장할 변수
   double average;                      // 평균을 저장할 변수
   for(i=0; i<5; i++) tot+=ap[i];       // 배열요소의 개수만큼 반복하면서 tot에 누적한다.
   average=tot/5.0;                     // 평균 계산
   return average;                      // 계산된 평균값 리턴
}
```

포인터 연산과 원소 크기

- 포인터 연산은 C의 강력한 특징 중 하나
- 변수 p를 특정형에 대한 포인터라고 하면, 수식 p + 1은 그 형의 다음 변수를 나타냄
- p와 q가 모두 한 배열의 원소들을 포인트하고 있다면, p - q는 p와 q 사이에 있는 배열 원소의 개수를 나타내는 int 값을 생성함

포인터 연산과 원소 크기

- **포인터 수식과 산술 수식은 형태는 유사하지만, 완전히 다름**

```
double   a[2], *p, *q;
p = a;              /* points to base of array */
q = p + 1;/* equivalent to q=&a[1] */
printf("%d\n", q - p);         /* 1 is printed */
printf("%d\n", (int) q - (int) p); /* 8 is printed */
```

함수 인자로서의 배열

- 함수 정의에서 배열로 선언된 형식 매개변수는 실질적으로는 포인터임
- 함수의 인자로 배열이 전달되면, 배열의 기본 주소가 "값에 의한 호출"로 전달됨
- 배열 원소 자체는 복사되지 않음
- 표기상의 편리성 때문에 포인터를 매개변수로 선언할 때 배열의 각괄호 표기법을 사용할 수 있음

함수 인자로서의 배열

- **예제 코드**

```
double sum(double a[], int n)  /* n is the size a[] */
{
   int     i;
   double  sum = 0.0;
   for (i = 0; i < n; ++i)
       sum += a[i];
   return sum;
}
```

함수 인자로서의 배열

- **함수 헤드를 다음과 같이 정의해도 됨**

```
double sum(double *a, int n)   /* n is the size a[] */
{
    .....
```

함수 인자로서의 배열

- **다양한 함수 호출 방법 및 의미**

호출	계산 및 리턴되는 값
sum(v, 100)	v[0] + v[1] + … + v[99]
sum(v, 88)	v[0] + v[1] + … + v[87]
sum(&v[7], k - 7)	v[7] + v[8] + … + v[k - 1]
sum(v + 7, 2 * k)	v[7] + v[8] + … + v[2 * k + 6]

calloc()과 malloc()

- **stdlib.h에 정의되어 있음**
  - calloc : contiguous allocation
  - malloc : memory allocation
- **프로그래머는 calloc()과 malloc()을 사용하여 배열, 구조체, 공용체를 위한 공간을 동적으로 생성함**

calloc()과 malloc()

- **각 원소의 크기가 el_size인 n 개의 원소를 할당하는 방법**

  calloc(n, el_size);

  malloc(n * el_size);

  - calloc()은 모든 원소를 0으로 초기화하는 반면 malloc()은 하지 않음

- **할당받은 것을 반환하기 위해서는 free()를 사용**

calloc()과 malloc()

- **예제 코드**

```
#include   <stdio.h>
#include   <stdlib.h>
int main(void){
   int   *a;            /* to be used as an array */
   int   n ;            /* the size of the array */
   .....                /* get n from somewhere,
  perhaps
                          interactively from the user
  */
   a = calloc(n, sizeof(int));      /* get space for a
  */
   .....
   free(a);
   .....
}
```

calloc()과 malloc()

- calloc함수는 배열을 할당 받고 초기화한다.

```
void *calloc(unsigned int, unsigned int);
```

- **첫 번째 배열요소의 개수, 두 번째는 배열요소의 크기를 전달인자로 준다.**
- **double형 변수 5개로 사용할 배열을 할당 받는 경우**

```
double *dp;
dp = (double *) calloc ( 5,  sizeof(double) );
```

배열요소의 개수 ------ double형 변수 하나의 크기

```
double *ap;
int i;
ap=(double *)calloc(5, sizeof(double));
for(i=0; i<5; i++){
   printf("%lf\n", ap[i]);
}
```

출력 결과

```
0.000000
0.000000
0.000000
0.000000
0.000000
```

// 모두 0으로
// 초기화 된다.

calloc()과 malloc()

```
#include <stdio.h>
#include <stdlib.h>      // malloc함수를 사용하기 위해서 포함시킨다.

int main()
{
   int *ip;              // int형을 가리킬 포인터변수
   double *dp;           // double형을 가리킬 포인터변수

   ip=(int *)malloc(sizeof(int));          // 기억공간을 동적으로 할당 받아서
   dp=(double *)malloc(sizeof(double));// 각 포인터 변수에 연결시킨다.

   *ip=10;               // 포인터변수로 각각 할당 받은 기억공간을
   *dp=3.4;              // 참조하여 값을 저장한다.

   printf("정수형으로 사용 : %d\n", *ip);    // 포인터변수로 저장된 값을 출력한다.
   printf("실수형으로 사용 : %lf\n", *dp);

   return 0;
}
```

출력 결과

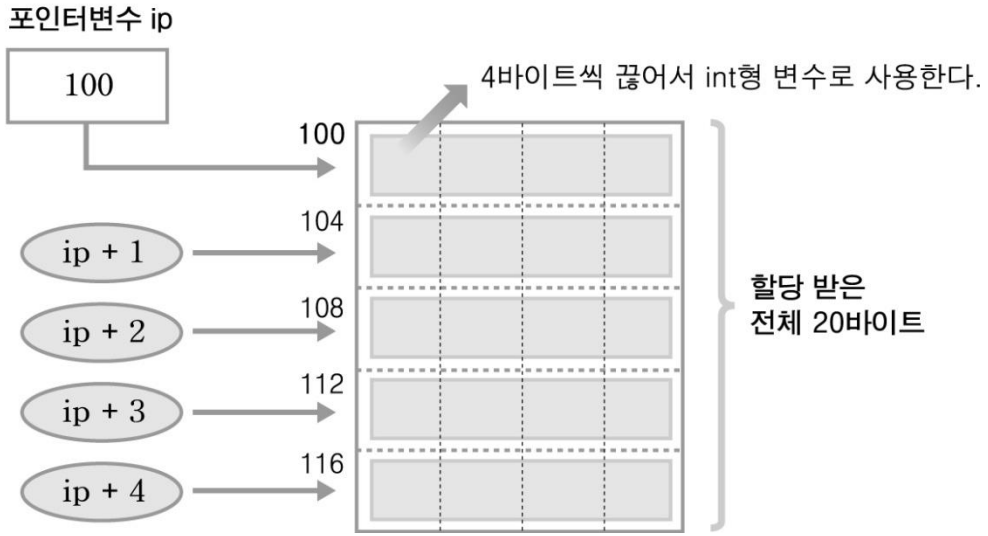
```
정수형으로 사용 : 10
실수형으로 사용 :
3.400000
```

calloc()과 malloc()

- 메모리의 동적할당은 많은 기억공간을 한꺼번에 할당 받아서 배열로 사용하는 것이 효율적이다.

  - int형 변수 5개를 동적으로 할당 받는 경우

```
int *ip;                    // 포인터변수 선언
ip = (int *)malloc(20);     // 20바이트를 한꺼번에 할당 받는다.
```



calloc()과 malloc()

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *ip;
   int i, sum=0;

   ip=(int *)malloc(5*sizeof(int));        // 전체 20바이트의 기억공간 할당
   if(ip==0){                              // 메모리가 할당 되었는지 확인하여
      printf("메모리가 부족합니다!\n");     // 메모리가 부족하면 메시지를 출력하고
      return 1;                            // 프로그램을 종료한다.
   }
   printf("다섯 명의 나이를 입력하세요 : ");
   for(i=0; i<5; i++){
      scanf("%d", ip+i);                   // 데이터를 저장할 포인터를 전달한다.
      sum+=ip[i];                          // 입력된 값을 참조하여 누적한다.
   }
   printf("다섯 명의 평균나이 : %.1lf\n", sum/5.0); // 평균나이 출력
   free(ip);                                        // 할당 받은 메모리 반환

   return 0;
}
```

출력 결과

```
다섯 명의 나이를 입력하세요 : 21 27 24 22 35 (엔터)
다섯 명의 평균나이 : 25.8
```

calloc()과 malloc()

- 메모리 동적 할당을 사용하면 입력되는 문자열의 길이에 맞게 기억공간을 할당할 수 있다.
  - ① **문자열을 입력 받기 전에는 그 길이를 알 수 없으므로 우선 충분한 크기의 문자배열이 필요하다.**

  - ② **문자배열에 문자열을 입력 받는다.**

  - ③ **문자열의 길이를 계산하여 그 크기에 맞게 기억공간을 동적으로 할당 받는다.**

  - ④ **동적으로 할당 받은 기억공간에 입력 받은 문자열을 복사한다.**

문자열

- **문자열**
  - char 형의 1차원 배열
  - 문자열은 끝의 기호인 \0, 또는 널 문자로 끝남
  - 널 문자 : 모든 비트가 0인 바이트; 십진 값 0
  - 문자열의 크기는 \0까지 포함한 크기

문자열

- **문자열 상수**
  - 큰따옴표 안에 기술됨
  - 문자열 예 : "abc"
    - 마지막 원소가 널 문자이고 크기가 4인 문자 배열
- **주의 - "a"와 'a'는 다름**
  - 배열 "a"는 두 원소를 가짐
  - 첫 번째 원소는 'a', 두 번째 원소는 '\0'

문자열

- **컴파일러는 문자열 상수를 배열 이름과 같이 포인터로 취급**

```
char  *p = "abc";
printf("%s %s\n", p, p + 1);   /* abc bc is
  printed */
```

  - 변수 p에는 문자 배열 "abc"의 기본 주소가 배정
  - char 형의 포인터를 문자열 형식으로 출력하면, 그 포인터가 포인트하는 문자부터 시작하여 \0이 나올 때까지 문자들이 연속해서 출력됨

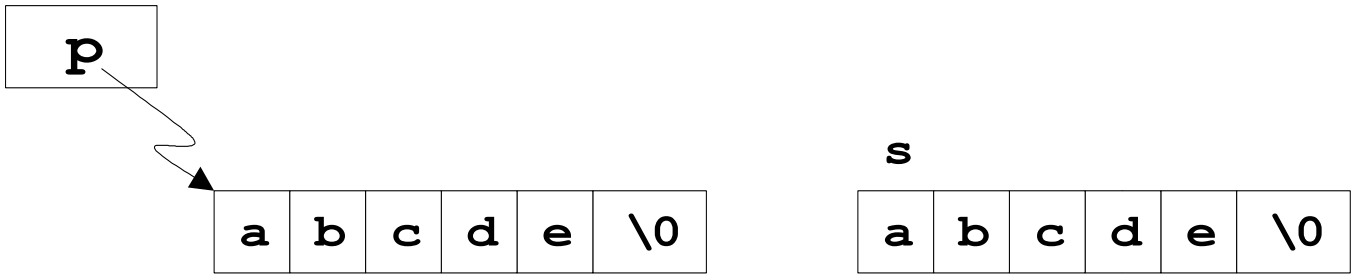
문자열

- **"abc"와 같은 문자열 상수는 포인터로 취급되기 때문에 "abc"[1] 또는 *("abc" + 2)와 같은 수식을 사용할 수 있음**

문자열

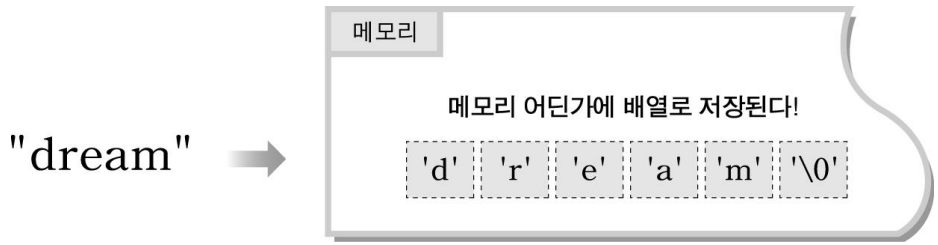
- **배열과 포인터의 차이**

```
char  *p = "abcde";
char  s[ ] = "abcde";
 // char  s[ ] = {'a', 'b', 'c', 'd', 'e', '\0'};
```



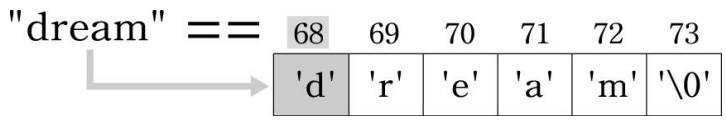
문자열

- 프로그램에서 사용된 모든 문자열은 메모리에 배열의 형태로 저장된다.



- **문자열이 컴파일되면 첫 번째 문자를 가리키는 포인터로 치환된다. 결국 문자열상수는 char형을 가리키는 포인터이다!**

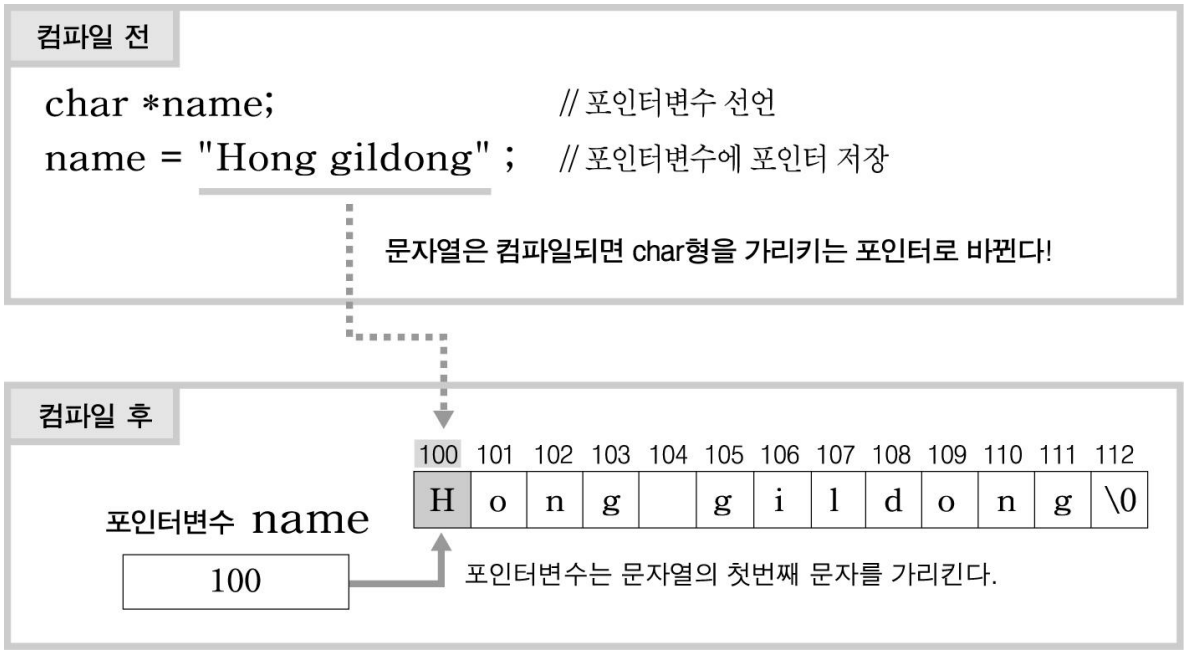
```
#include <stdio.h>

int main()
{
   printf("주소값을 출력 : %u\n", "dream");
   printf("첫번째 문자를 출력 : %c\n", *"dream");
   printf("세번째 문자를 출력 : %c\n", "dream"[2]);
   return 0;
}
```




```
주소값을 출력 : 4350068
첫번째 문자를 출력 : d
세번재 문자를 출력 : e
```

문자열

- 문자열상수가 포인터이므로 포인터변수에 저장하여 사용할 수 있다.



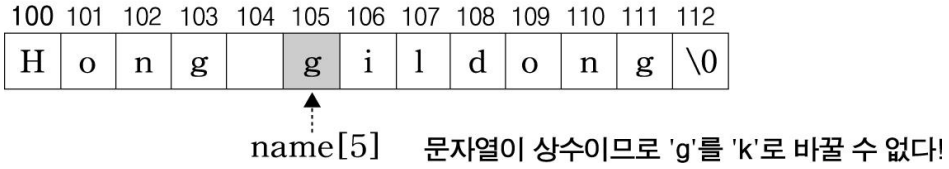
printf("이름 : %s\n", name);

printf("여섯번째 문자 : %c\n", name[5]);

문자열

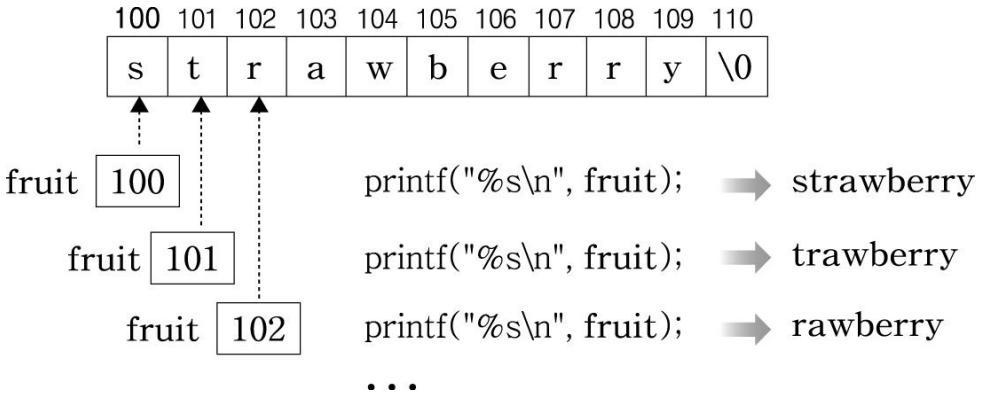
- 문자열상수는 상수이므로 포인터변수로 그 값을 바꿀 수 없다.

name[5] = 'k'



- **그러나 포인터변수의 값은 바꿀 수 있으므로 포인터변수가 문자열상수의 중간을 가리키게끔 할 수 있다.**

```
#include <stdio.h>

int main()
{
   char *fruit="strawberry";
   while(*fruit != '\0'){
      printf("%s\n", fruit);
      fruit++;
   }
   return 0;
}
```



문자열

- 포인터변수는 여러 개의 문자열을 선택하여 처리할 수 있다.

```
#include <stdio.h>

int main()
{
   int age;
   char *greeting;

   printf("나이를 입력하세요 : ");
   scanf("%d", &age);

   if(age>30) greeting="처음 뵙겠습니다.";
   else greeting="반가워요.";   // 나이에 따라 서로 다른 문자열을 저장한다.

   printf("인사말 : %s\n", greeting);
   return 0;
}
```

문자열

- **예제 코드**

```
#include   <ctype.h>
int word_cnt(const char *s){
   int   cnt = 0;
   while (*s != '\0') {
      while (isspace(*s))          // skip white space
         ++s;
      if (*s != '\0') {            // found a word
         ++cnt;
         while (!isspace(*s) && *s != '\0')
                                    // skip the word
            ++s;
      }
   }
   return cnt;
}
```

문자열 조작 함수

- char *strcat(char *s1, const char *s2);
  - 두 문자열 s1,s2를 결합하고, 결과는 s1에 저장
- int strcmp(const char *s1, const char *s2);
  - s1과 s2를 사전적 순서로 비교하여, s1이 작으면 음수, 크면 양수, 같으면 0을 리턴
- char *strcpy(char *s1, const char *s2);
  - s2의 문자를 \0이 나올 때까지 s1에 복사
- size_t strlen(const char *s);
  - \0을 뺀 문자의 개수를 리턴

문자열 조작 함수

선언 및 초기화	
char s1[ ] = "beautiful big sky country", s2[ ] = "how now brown cow";	
수식	값
strlen(s1) strlen(s2 + 8) strcmp(s1, s2)	25 9 음의 정수
문장	출력되는 값
printf("%s", s1+10); strcpy(s1 + 10, s2 + 8); strcat(s1, "s!"); printf("%s", s1);	big sky country beautiful brown cows!

문자열 조작 함수 strlen()

- 문자열상수나 배열에 저장된 문자열을 다른 배열에 복사한다.

```
unsigned int strlen(char *);  // 문자열의 길이 계산, string length
```

- 전달인자로 주어지는 문자열의 길이를 계산하여 리턴한다(널문자 제외).

```
char fruit[80] = "apple";
int len;
len = strlen(fruit);
printf("문자열의 길이 : %d\n", len);  // 문자열의 길이 : 5
```

- **배열에 저장된 문자열의 길이만을 계산해 준다.**

```
sizeof(fruit)  →  80   // 배열 전체의 크기
strlen(fruit)  →  5    // 배열에 저장된 문자열의 크기
```

- **문자열상수나 문자열을 가리키는 포인터변수를 사용할 수도 있다.**

```
char *strp = "apple";
strlen(strp);
strlen("banana");
```

문자열 조작 함수 strlen()

```
size_t strlen(const char *s){
    size_t   n;
    for (n = 0; *s != '\0'; ++s)
        ++n;
    return n;
}
```

문자열 조작 함수 – strcpy()

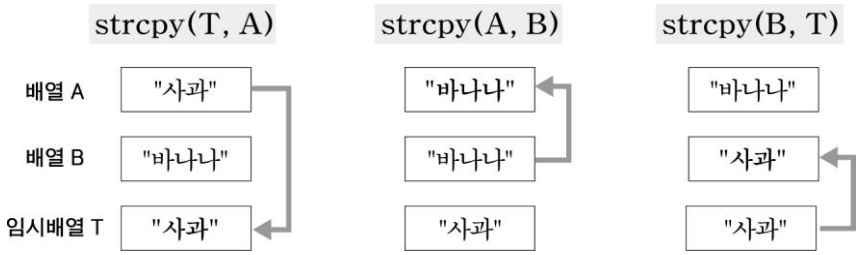
- 문자열상수나 배열에 저장된 문자열을 다른 배열에 복사한다.

```
char *strcpy(char *, char *);   // 문자열의 복사, string copy
```

- 두 번째 전달인자로 주어지는 문자열을 첫번째 전달인자의 위치에 복사한다.

- **두 배열에 저장된 문자열을 바꾸는 프로그램**

```
#include <stdio.h>
#include <string.h>   // 헤더파일 포함
int main()
{
   char str1[20]="apple";
   char str2[20]="banana";
   char temp[20];
   strcpy(temp, str1); // "apple" -> temp
   strcpy(str1, str2);   // "banana" -> str1
   strcpy(str2, temp); // "apple" -> str2
   printf("str1 : %s\n", str1);
   printf("str2 : %s\n", str2);
   return 0;
}
```



문자열 조작 함수 – strcpy()

```
char *strcpy(char *s1, register const char *s2)
{
    register char   *p = s1;
    while (*p++ = *s2++)
        ;
    return s1;
}
```

문자열 조작 함수 – strcmp()

- 두 문자열의 사전적 순서를 따진다.

```
int strcmp(char *str1, char *str2);  // 문자열의 비교, string compare
```

- 두 문자열을 비교하여 리턴하는 값(사전의 뒤에 나오는 것이 큰 문자열이다).

크기 비교	str1 > str2	str1 < str2	str1 == str2
리턴값	1	-1	0

- **두 문자열의 순서를 따져서 사전 순서로 바꾸는 예**

```
#include <stdio.h>
#include <string.h>

int main()
{
   char str1[20]="banana";
   char str2[20]="apple";
   char temp[20];
   int res;

   res=strcmp(str1, str2);
   if(res>0){          // str1이 더 크면 문자열을 바꾼다.
      strcpy(temp, str1);
      strcpy(str1, str2);
      strcpy(str2, temp);
   }
   printf("str1 : %s\n", str1);
   printf("str2 : %s\n", str2);
   return 0;
}
```

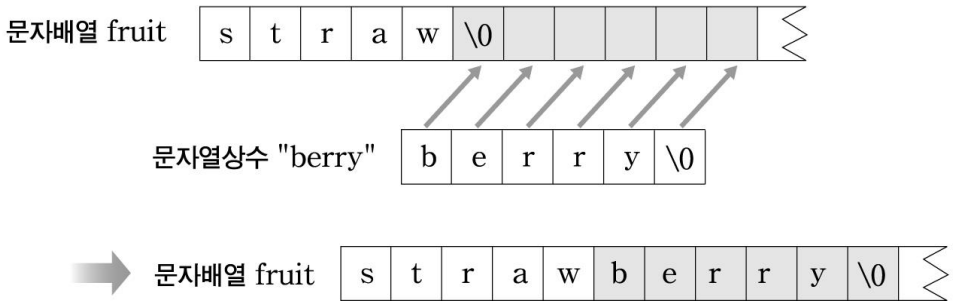
문자열 조작 함수 – strcat()

- 두 문자열을 붙여서 하나의 문자열을 만든다.

```
char *strcat(char *str1, char *str2);  // string concatenation
```

- str2의 문자열을 str1의 문자열 뒤에 붙인다.

```
#include <stdio.h>
#include <string.h>
int main()
{
   char fruit[80]="straw";
   strcat(fruit, "berry");
   printf("연결된 문자열 : %s\n", fruit);
   return 0;
}
```



gets(), puts()

- gets함수는 빈칸을 포함하여 한 줄을 입력할 수 있다.

```
char *gets(char *);   // 문자열의 입력, get string
```

- 전달인자는 입력 받을 배열의 주소이다.
- 입력될 문자열이 충분히 저장될 수 있도록 배열을 선언해야 한다.
- 데이터를 입력한 후에 마지막에 널문자를 붙여서 문자열을 완성한다.

```
char str[80];
printf("문자열을 입력하세요 : ");
gets(str);
printf("입력된 문자열 : %s\n", str);
```

```
문자열을 입력하세요 : 백번 보는 것보다 한번 짜보는 것이 낫다. (엔터)
입력된 문자열 : 백번 보는 것보다 한번 짜보는 것이 낫다.
```

gets(), puts()

- puts함수는 문자열만을 출력하는 전용 함수이다.

```
int puts(char *);  // 문자열의 출력, put string
```

- **문자열만을 출력하므로 printf함수보다 훨씬 작고 간편하다.**
- **출력할 문자열이 저장된 배열명을 전달인자로 준다.**
- **문자열을 출력한 후에는 자동으로 줄이 바뀐다.**

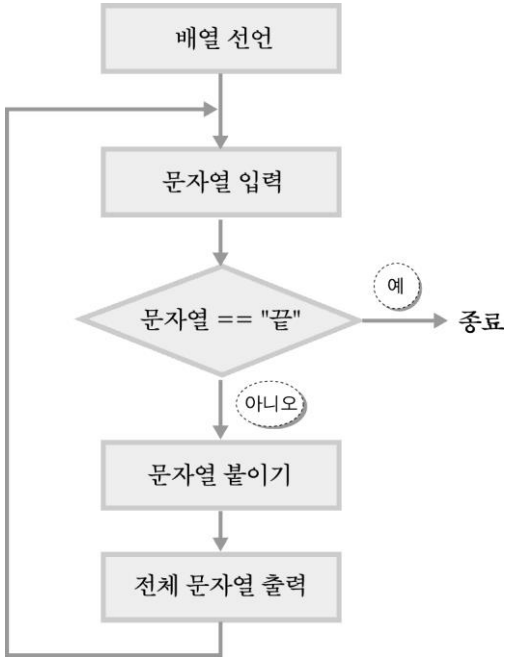
```
char str[80];
printf("문자열을 입력하세요 : ");
gets(str);
printf("입력된 문자열 : ");
puts(str)              // printf("%s\n", str);
```

```
문자열을 입력하세요 : 백번 보는 것보다 한번 짜보는 것이 낫다. (엔터)
입력된 문자열 : 백번 보는 것보다 한번 짜보는 것이 낫다.
_          // 커서의 위치!
```

gets(), puts()

- 키보드로 하나의 문장들을 입력 받을 때마다 이미 입력 받은 문장들과 연결하여 전체를 출력한다. “끝”이 입력되면 프로그램을 종료한다.

```
문자열을 입력하세요 : 문자열은 (엔터)
현재까지의 줄거리 : 문자열은

문자열을 입력하세요 : 컴파일 되면 (엔터)
현재까지의 줄거리 : 문자열은 컴파일 되면

문자열을 입력하세요 : 주소를 (엔터)
현재까지의 줄거리 : 문자열은 컴파일 되면 주소를

문자열을 입력하세요 : 남긴다 (엔터)
현재까지의 줄거리 : 문자열은 컴파일 되면 주소를 남긴다.

문자열을 입력하세요 : 끝 (엔터)          // 프로그램 종료
```



gets(), puts()

```
#include <stdio.h>
#include <string.h>

int main()
{
   char novel[800]={0};     // 전체 줄거리를 저장할 배열, 초기화가 필요하다!
   char str_in[80];         // 입력 문자열을 저장할 배열
   while(1){
      printf("문자열을 입력하세요 : ");
      _____________________________________ // 한 줄을 입력한다.
      if(_____________________) break;// 입력된 문자열이 "끝"이면 반복문을 빠져 나간다.
      strcat(_____________________);      // 입력된 문자열을 전체 줄거리에 붙인다.
      strcat(_____________________);      // 다음 문자열을 위해 한 칸 띄운다.
      printf("현재까지의 줄거리 : ");
      puts(_____________);                // 전체 줄거리 출력
      puts("\n");                         // 한 줄을 띄운다.
   }
   return 0;
}
```

// novel배열을 널문자로 초기화하지 않으면 쓰레기 문자들이 있으므로
// 처음에 문자열 붙일 때 쓰레기 문자들이 남을 가능성이 있다!!

gets(), puts()

```
#include <stdio.h>
#include <string.h>

int main()
{
   char novel[800]={0};    // 전체 줄거리를 저장할 배열, 초기화가 필요하다!
   char str_in[80];        // 입력 문자열을 저장할 배열

   while(1){
      printf("문자열을 입력하세요 : ");
      gets(str_in);                          // 한 줄을 입력한다.
      if(strcmp(str_in, "끝")==0) break;     // 입력된 문자열이 "끝"이면 반복문을 빠져 나간다.
      strcat(novel, str_in);                 // 입력된 문자열을 전체 줄거리에 붙인다.
      strcat(novel, " ");                    // 다음 문자열을 위해 한 칸 띄운다.
      printf("현재까지의 줄거리 : ");
      puts(novel);                           // 전체 줄거리 출력
      puts("\n");                            // 한 줄을 띄운다.
   }

   return 0;
}
```

// novel배열을 널문자로 초기화하지 않으면 쓰레기 문자들이 있으므로
// 처음에 문자열 붙일 때 쓰레기 문자들이 남을 가능성이 있다!!

다차원 배열

- C 언어는 배열의 배열을 포함한 어떠한 형의 배열도 허용함
- 2차원 배열은 두 개의 각괄호로 만듬
- 이 개념은 더 높은 차원의 배열을 만들 때에도 반복적으로 적용됨

배열 선언의 예	설명
int a[100]; int b[2][7]; int c[5][3][2];	1차원 배열 2차원 배열 3차원 배열

2차원 배열

- **2차원 배열은 행과 열을 갖는 직사각형의 원소의 집합으로 생각하는 것이 편리함**
  - 사실 원소들은 하나씩 연속적으로 저장됨
- **선언**

```
int a[3][5];
```

	1 열	2 열	3 열	4 열	5 열
1 행	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
2 행	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
3 행	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

형식 매개변수 선언

- **함수 정의에서 형식 매개변수가 다차원 배열일 때, 첫 번째 크기를 제외한 다른 모든 크기를 명시해야 함**

형식 매개변수 선언

- **예 (int a[3][5];으로 선언되어 있을 때)**

```
int sum(int a[][5]){    /* int sum(int a[3][5])
                           or int sum(int (*a)[5]) */
    int  i, j, sum=0;
    for (i = 0; i < 3; ++i)
        for (j = 0; j < 5; ++j)
            sum += a[i][j];
    return sum;
}
```

3차원 배열

- **3차원 배열 선언 예**

```
int    a[7][9][2];
```

  - a[i][j][k]를 위한 기억장소 사상 함수:

*(&a[0][0][0] + 9 * 2 * i + 2 * j + k)

- **함수 정의 헤더에서 다음은 다 같음**

```
int sum(int a[][9][12])
int sum(int a[7][9][12])
int sum(int (*a)[9][12])
```

초기화

- **다차원 배열 초기화 방법**

```
int     a[2][3] = {1, 2, 3, 4, 5, 6};
int     a[2][3] = {{1, 2, 3}, {4, 5, 6}};
int     a[ ][3] = {{1, 2, 3}, {4, 5, 6}};
```

  - 내부 중괄호가 없으면, 배열은 a[0][0], a[0][1], ..., a[1][2] 순으로 초기화되고, 인덱싱은 행 우선 임
  - 배열의 원소 수보다 더 적은 수의 초기화 값이 있다면, 남는 원소는 0으로 초기화됨
  - 첫 번째 각괄호가 공백이면, 컴파일러는 내부 중괄호 쌍의 수를 그것의 크기로 함
  - 첫 번째 크기를 제외한 모든 크기는 명시해야 함

초기화

- **초기화 예**

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

  - 이것은 다음과 같음

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

  - 모든 배열 원소를 0으로 초기화 하기

```
int  a[2][2][3] = {0};
       /* all element initialized to zero */
```

2차원배열 예제1)

/* 3명의 4과목 점수를 2차원 배열을 이용 저장 총점, 평균을 구하고 출력 하는 프로그램 */

```
#include <stdio.h>
int main()
{
   int score[3][4];                          // 3명의 4과목 점수를 저장할 2차원 배열 선언
   int i, j;                                 // 2중 for문을 위한 반복 제어변수
   int tot;                                  // 총점을 저장할 변수
   double avg;                               // 평균을 저장할 변수
    for(i=0; i<3; i++){                      // 3명이므로 3번 반복
        printf("네 과목의 점수를 입력하세요 : ");
        for(j=0; j<4; j++){                  // 4과목이므로 4번 반복
        scanf("%d", &score[i][j]);           // 점수 입력
        }
    }
    for(i=0; i<3; i++){
      tot=0;                                 // 각 학생의 점수를 새롭게 누적할 때마다 0으로 초기화
      for(j=0; j<4; j++){
       tot+=score[i][j];                     // 한 학생의 점수를 총점에 누적한다.
      }
      avg=tot/4.0;                           // 한 명의 총점을 모두 누적한 후에 평균 계산
      printf("총점 : %d, 평균 : %.2lf\n", tot, avg);  // 총점, 평균 출력
    }
   return 0;
}
```

2차원배열 예제2)

```
#include <stdio.h>

int main()
{
   char animal[][10]={ "monkey", "elephant", "dog", "sheep", "pig",
                       "lion", "tiger", "puma", "turtle", "fox" };
                                        // 2차원 문자배열의 선언과 초기화

   int i;                               // 반복 제어변수
   int count;                           // 문자열의 개수를 저장할 변수

   count=sizeof(animal)/sizeof(animal[0]); // 초기화된 문자열의 수를 계산한다.
   for(i=0; i<count; i++){                 // 문자열의 개수만큼 반복
      printf("%s\n", animal[i]);           // 저장된 문자열의 출력
   }

   return 0;
}
```

문자열의 개수 계산 ⇨ sizeof(animal) / sizeof(animal[0])

sizeof(animal): 전체 배열의 크기

sizeof(animal[0]): 부분배열 하나의 크기

typedef

- **사용 예**

```
#define   N   3
typedef   double  scalar;
typedef   scalar  vector[N];
typedef   scalar  matrix[N][N];
/* typedef vector matrix[N]; */
```

- **의미있는 이름을 사용하는 형 이름을 정의하여 가독성을 높임**

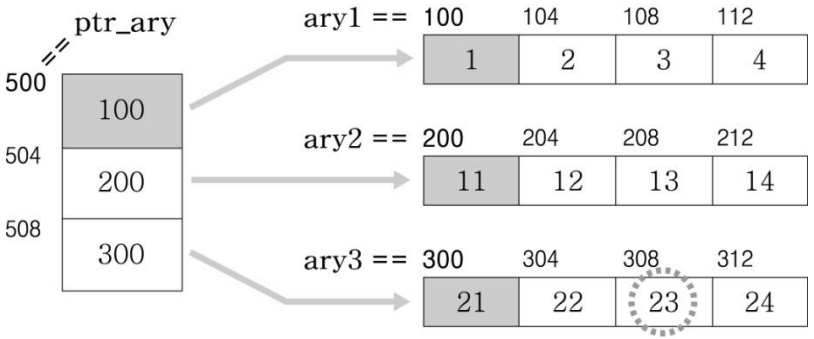
포인터 배열

- **배열의 원소의 형은 포인터형을 포함하여 임의의 형이 될 수 있음**
- **포인터 배열은 문자열을 다룰 때 많이 사용됨**

포인터배열

- 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 초기화한다.
```



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

- **ary3배열의 세 번째 배열요소(23값)를 참조하는 과정**

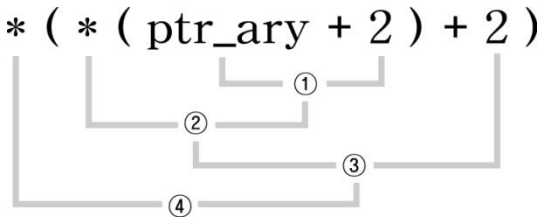
1. 먼저 ptr_ary 배열의 세 번 째 배열요소를 참조한다. ⇨ **ptr_ary[2]**

2. 참조된 배열요소 ptr_ary[2]는 배열명 ary3를 저장한 포인터변수이므로 배열명처럼 사용하여 ary3의 세 번째 배열요소를 참조한다.

   ⇨ **printf("%d", ptr_ary[2][2]);**

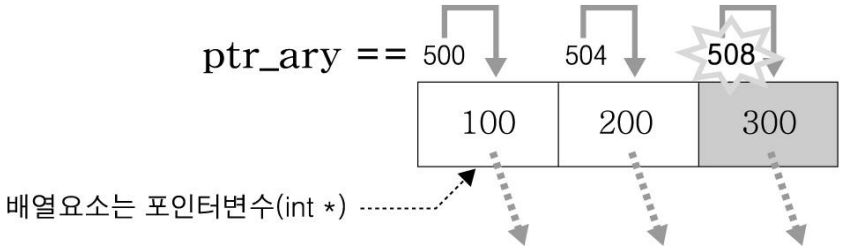
포인터배열

- ptr_ary[2][2]가 참조되는 과정의 주소값을 계산해보자.
  - **일단 포인터표현으로 바꾸고 연산순서를 따라간다.**



- **①번 연산 : 포인터배열의 세 번째 배열요소를 가리키는 포인터가 구해진다.**

ptr_ary+2 = ptr_ary+(2*sizeof(ptr_ary[0])) = 500+(2*4) = 508

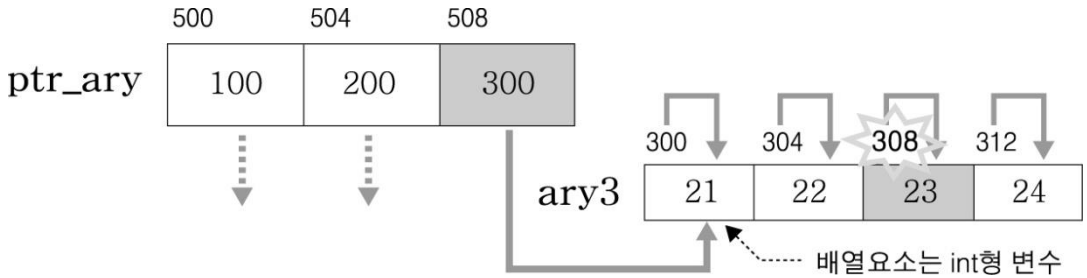


포인터배열

- ②번 연산 : 포인터배열의 세 번째 배열요소의 값 300번지가 구해진다.

- ③번 연산 : ary3배열의 세 번재 기억공간을 가리키는 포인터가 구해진다.

  300+2 => 300+(2*sizeof(ary3[0])) => 308



- ④번 연산 : 308번지는 포인터이므로 참조연산을 수행하면 값23이 참조된다.

포인터배열

```
#include <stdio.h>

int main()
{
   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}; // 포인터배열에 각 배열명을 초기화한다.
   int i, j;                           // 반복 제어변수

   for(i=0; i<3; i++){
      for(j=0; j<4; j++){
         printf("%5d", ptr_ary[i][j]);     // 3행 4열의 2차원 배열처럼 출력할 수 있다.
      }
      printf("\n");                        // 한 행을 출력한 후에 줄을 바꾼다.
   }

   return 0;
}
```

출력 결과

```
 1   2   3   4
11  12  13  14
21  22  23  24
```

main() 함수의 인자

- **main()은 운영체제와의 통신을 위해 argc와 argv라는 인자를 사용함**
- **예제 코드**

```
void main(int argc, char *argv[]) {
    int   i;
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
       printf("argv[%d] = %s\n", i, argv[i]);
}
```

  - argc : 명령어 라인 인자의 개수를 가짐
  - argv : 명령어 라인을 구성하는 문자열들을 가짐

main() 함수의 인자

- **앞의 프로그램을 컴파일하여 my_echo로 한 후, 다음 명령으로 실행:**

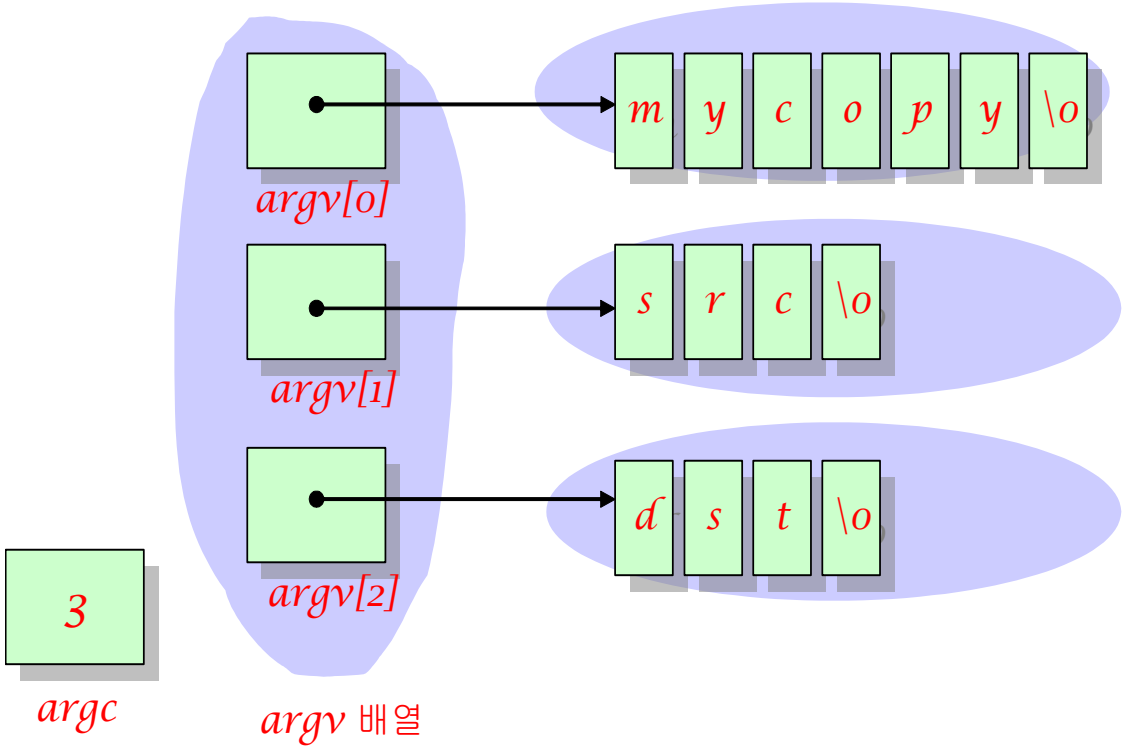
```
$ my_echo a is for apple
```

- **출력:**

```
argc = 5
argv[0] = my_echo
argv[1] = a
argv[2] = is
argv[3] = for
argv[4] = apple
```

인수 전달 방법
-다른 예제

```
C: \cprogram> mycopy src dst
```



main_arg.c

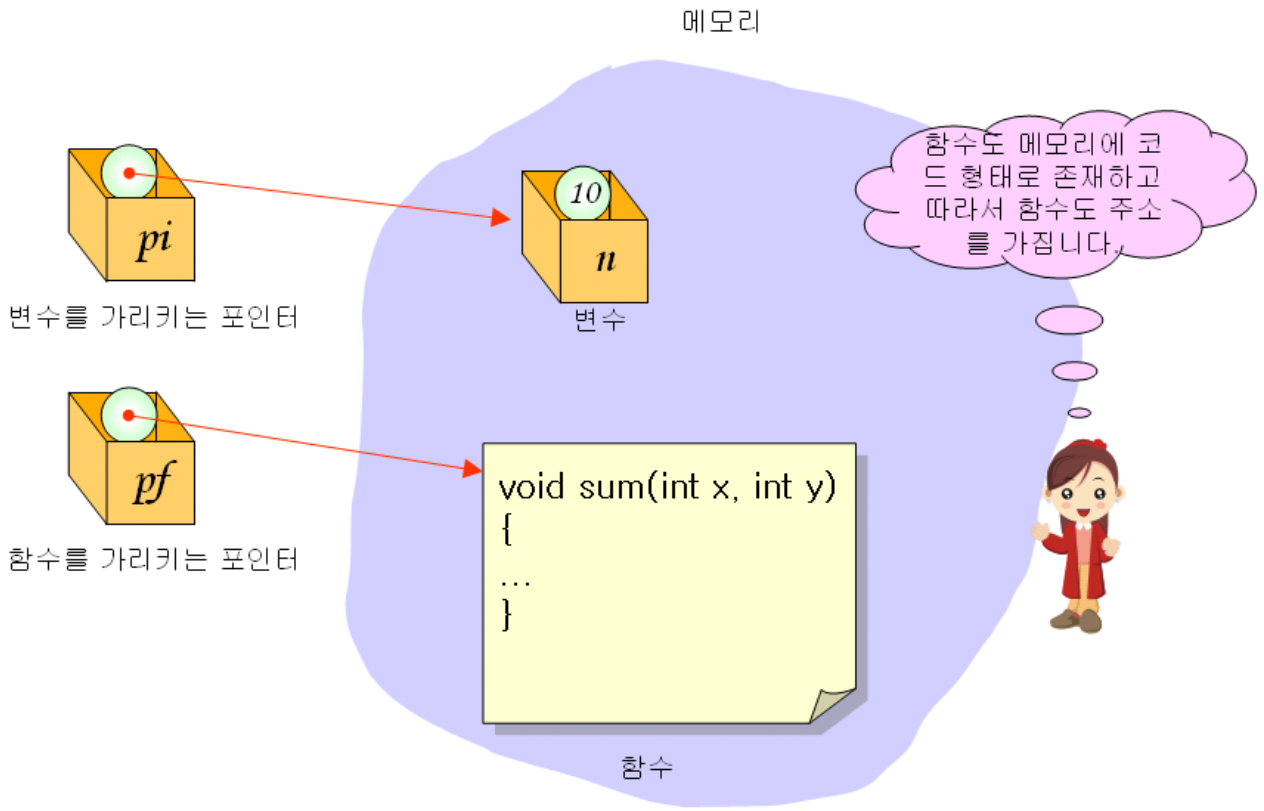
```c
#include <stdio.h>

int main(int argc, char *argv[])
{
	int i = 0;

	for(i = 0;i < argc; i++)
		printf("명령어 라인에서 %d번째 문자열 = %s\n", i, argv[i]);

	return 0;
}
```

```
c:\cprogram\mainarg\Debug>mainarg src dst
명령어 라인에서 0번째 문자열 = mainarg
명령어 라인에서 1번째 문자열 = src
명령어 라인에서 2번째 문자열 = dst
c:\cprogram\mainarg\Debug>
```

함수 포인터

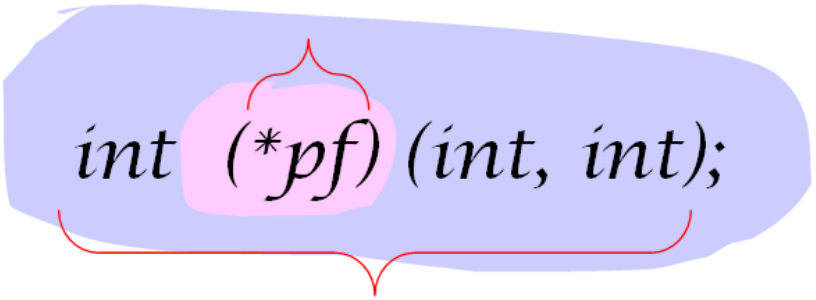
- 함수 포인터(function pointer): 함수를 가리키는 포인터

```
int (*pf)(int, int);
```



함수 포인터의 해석

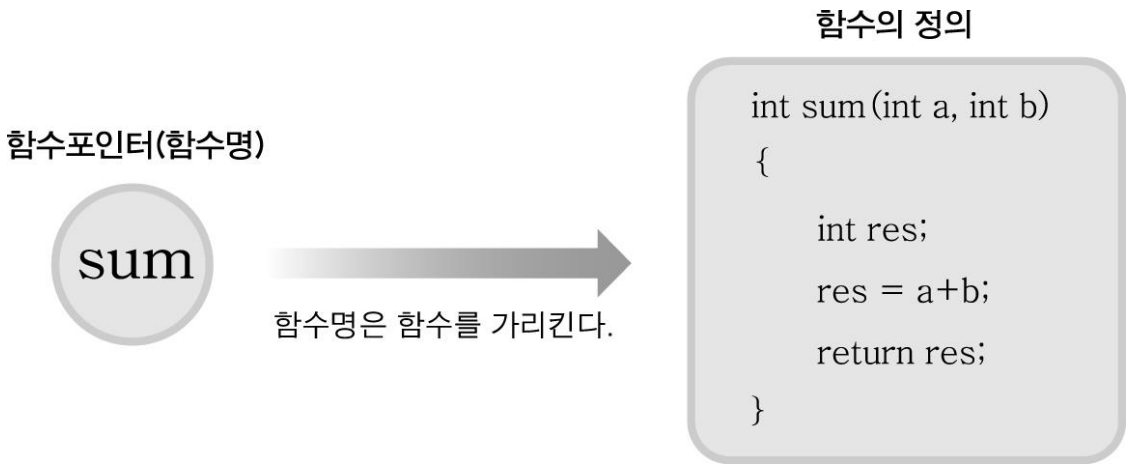
① 괄호에 의하여 () 연산자보다 * 연산자가 먼저 적용되어서 pf는 포인터가 된다.



② 어떤 포인터냐 하면 int f(int, int) 함수를 가리키는 포인터가 된다.

함수 포인터

- 함수포인터는 함수의 이름이다.
  - **함수명은 함수의 정의가 있는 메모리의 위치값이며 함수를 가리킨다.**



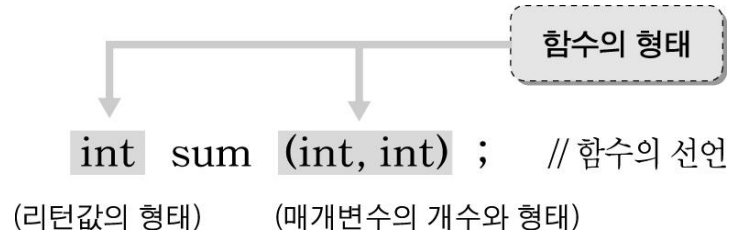
  - **함수명이 포인터라는 증거는 참조연산자를 사용하면 알 수 있다.**

```
(*sum)(10, 20);    // 10과 20을 전달인자로 주고 sum함수를 호출한다.
```

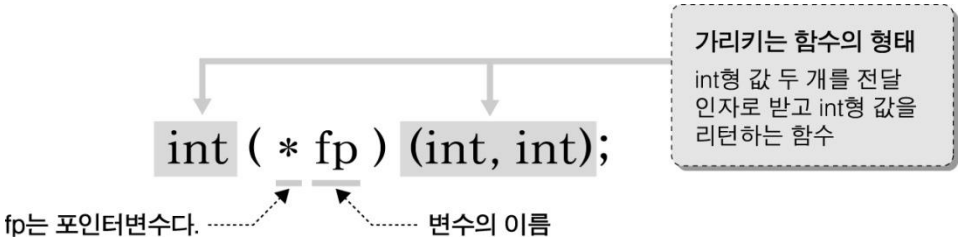
(함수를 참조한 후에 호출해야 하므로 참조연산에 괄호를 사용한다.)

함수 포인터

- 함수포인터가 가리키는 자료형은 함수의 형태이다.
  - **함수의 형태를 매개변수의 개수와 형태, 리턴값의 형태이다.**



- **함수포인터를 함수포인터변수에 저장하여 호출할 수 있다.**




```
int (*fp)(int, int);  // 함수포인터변수 선언
fp=sum;               // 함수명을 함수포인터변수에 저장한다.
fp(10, 20);           // 함수포인터변수로 함수 호출, (*fp)(10, 20)도 사용 가능
```

함수 포인터

```
#include <stdio.h>

int sum(int, int);                  // 함수의 선언

int main()
{
   int (*fp)(int, int);             // 함수 포인터변수 선언
   int res;                         // 리턴값을 저장할 변수

   fp=sum;                          // 함수명을 함수포인터변수에 저장한다.
   res=fp(10, 20);                  // 함수포인터변수로 함수를 호출한다.
   printf("result : %d\n", res);    // 리턴값 출력

   return 0;
}

int sum(int a, int b)               // 함수의 정의
{
   return a+b;
}
```

함수 포인터

- 함수포인터변수는 함수의 형태만 같으면 기능과 상관없이 모든 함수 포인터를 저장할 수 있다.

형태가 같다

```
int sum(int, int);   // 두 정수값을 더해서 리턴하는 함수
int mul(int, int);   // 두 정수값을 곱해서 리턴하는 함수
int max(int, int);   // 두 정수값 중에서 큰 값을 리턴하는 함수
...
```

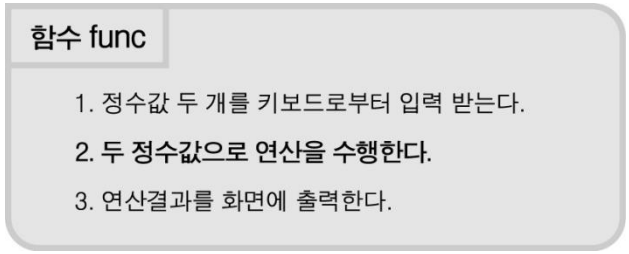
```
int (*fp) (int, int);  →  fp = sum;
                          fp = mul;
                          fp = max;
                          ...
```

모두 사용 가능하다.

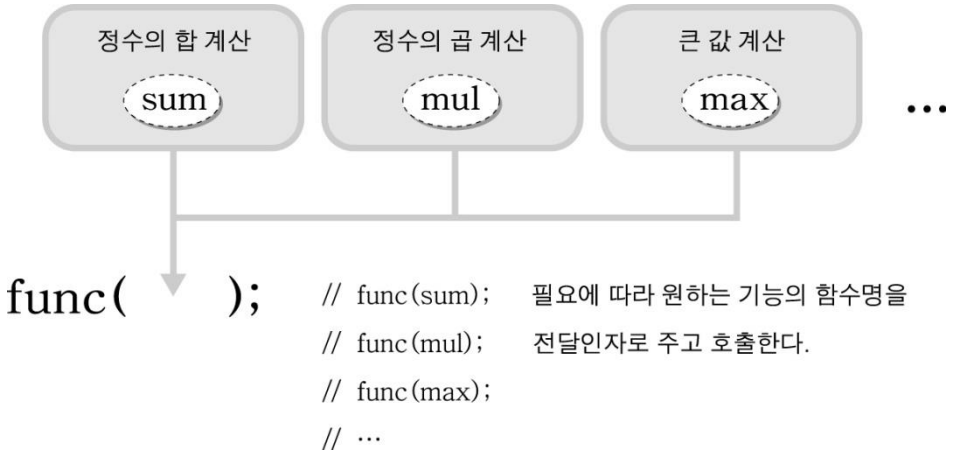
- **형태가 같은 다양한 기능의 함수를 선택적으로 호출하는데 사용할 수 있다.**

함수 포인터

- 다음과 같은 기능을 수행하는 함수를 만들자.



- func함수에서 2번 연산과정은 함수를 호출할 때 필요한 연산이 수행되도록 만들자.




```
void func(int (*fp)(int, int))
{
   …
   fp(a, b);
   …
}
```

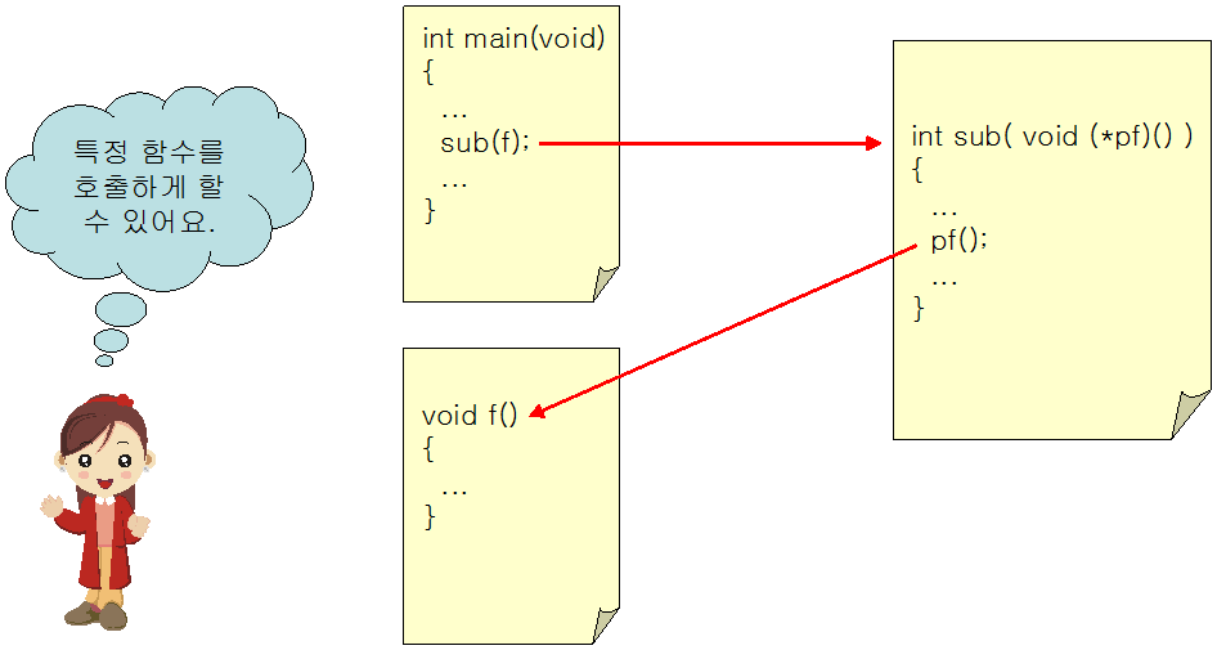
함수 포인터

```
#include <stdio.h>
void func(int (*)(int, int));
int sum(int, int);
int mul(int, int);
int max(int, int);
int main()
{
   int sel;
   printf("1. 두 정수의 합\n");
   printf("2. 두 정수의 곱\n");
   printf("3. 두 정수 중에서 큰 값 계산\n");
   printf("원하는 작업을 선택하세요 : ");
   scanf("%d", &sel);
   switch(sel){
   case 1: func(sum); break;
   case 2: func(mul); break;
   case 3: func(max); break;
   }
   return 0;
}
```

```
void func(int (*fp)(int, int))
{
   int a, b;
   int res;
   printf("두 정수값을 입력하세요 : ");
   scanf("%d%d", &a, &b);
   res=fp(a, b);
   printf("결과값은 : %d\n", res);
}

int sum(int a, int b)
{
   return a+b;
}

int mul(int a, int b)
{
   return a*b;
}
```

함수 인수로서의 함수 포인터

- 함수 포인터도 인수로 전달이 가능하다.



인자로서의 함수

- **함수의 포인터는 인자로서 전달될 수 있고, 배열에서도 사용되며, 함수로부터 리턴될 수도 있음**
- **예제 코드**

```
double sum_square(double f(double x), int m, int n)
  {
    int       k;
    double    sum = 0.0;
    for (k = m; k <= n; ++k)
       sum += f(k) * f(k);
    return sum;
}
```

인자로서의 함수

- **앞의 코드에서 식별자 x는 사람을 위한 것으로, 컴파일러는 무시함**
  - 즉, 다음과 같이 해도 됨

```
double sum_square(double f(double), int m, int n)
{
      . . . .
```

인자로서의 함수

- **포인터 f를 함수처럼 취급할 수도 있고, 또는 포인터 f를 명시적으로 역참조할 수도 있음**
  - 즉, 다음 두 문장은 같음:

```
sum += f(k) * f(k);
sum += (*f)(k) * (*f)(k);
```

인자로서의 함수

- **(*f)(k)**
  - f 함수에 대한 포인터
  - *f 함수 그 자체
  - (*f)(k) 함수 호출

Const 선언

const 선언하여 원본 데이터 변경 금지

const 선언하면 포인터 변수가 참조하는 값의 변경을 금지

형식	**const 타입 *포인터 변수;**
예	int x = 100; const int *pt = &x; *pt = 200; /* 컴파일 오류 */

원본 배열의 요소 내용을 변경되지 않게 하려면 함수를 정의(구현)한 부분에서 매개변수 앞에 const 키워드를 붙임

형식	**리턴타입 함수명(const 타입 배열명[ ])** **{** **몸체;** **}**

Const 선언

- ***const는 선언에서 기억영역 뒤와 형 앞에 옴***
- **사용 예**

```
static const int k = 3;
```

  - 이것은 "k는 기억영역 static인 상수 int이다"라고 읽음
- **const 변수는 초기화될 수는 있지만, 그 후에 배정되거나, 증가, 감소, 또는 수정될 수 없음**
- **변수가 const로 한정된다 해도, 다른 선언에서 배열의 크기를 명시하는 데는 사용될 수 없음**

const

- **예 1**

```
const int  n = 3;
int        v[n];
          /* any C compiler should complain */
```

- **예 2**

```
const int  a = 7;
int        *p = &a;  /* the compiler will complain */
```

  - p는 int를 포인트하는 보통의 포인터이기 때문에, 나중에 ++*p와 같은 수식을 사용하여 a에 저장되어 있는 값을 변경할 수 있음

- **예 3**

```
const  int  a = 7;
const  int  *p = &a;
```

  - 여기서 p 자체는 상수가 아님
  - p에 다른 주소를 배정할 수 있지만, *p에 값을 배정할 수는 없음

volatile 포인터

- volatile은 다른 프로세스나 스레드가 값을 항상 변경할 수 있으니 값을 사용할 때마다 다시 메모리에서 읽으라는 것을 의미

volatile

- **volatile 객체는 하드웨어에 의하여 어떤 방법으로 수정될 수 있음**

```
extern const volatile int real_time_clock;
```

  - 한정자 volatile은 하드웨어에 의해 영향을 받는 객체임을 나타냄
  - 또한 const도 한정자이므로, 이 객체는 프로그램에서 증가, 감소, 또는 배정될 수 없음
  - 즉, 하드웨어는 변경할 수 있지만, 코드로는 변경할 수 없음

void 포인터

- 순수하게 메모리의 주소만 가지고 있는 포인터
- 가리키는 대상물은 아직 정해지지 않음
  (예) `void *vp;`

- 다음과 같은 연산은 모두 오류이다.

```
*vp;            // 오류
*(int *)vp;     // void형 포인터를 int형 포인터로 변환한다.
vp++;           // 오류
vp--;           // 오류
```

참고문헌

- 열혈 C 프로그래밍, 윤성우, 오렌지미디어
- 쉽게 풀어쓴 C언어 Express, 천인국, 생능출판사
- 뇌를 자극하는 C 프로그래밍, 서현우, 한빛미디어
- 쾌도난마 C프로그래밍,강성수,북스홀릭
- C프로그래밍 기초와 응용실습, 고응남, 정익사

질의 및 응답