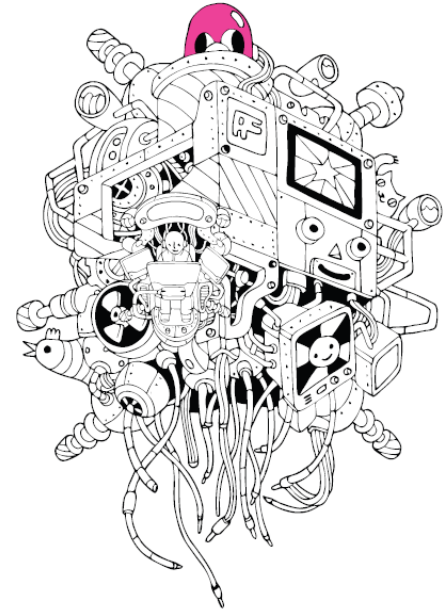


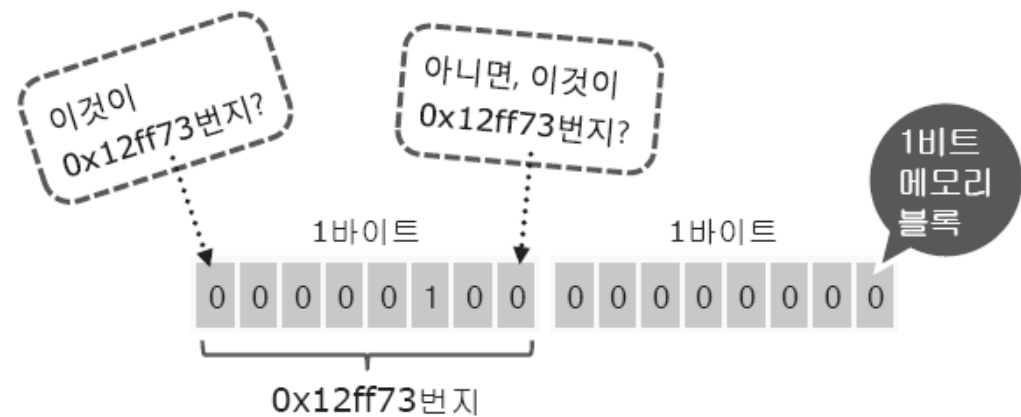
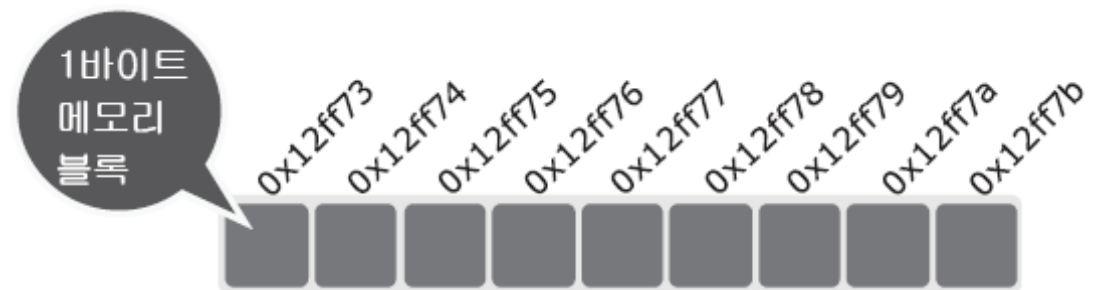
C 프로그래밍



포인터의 이해

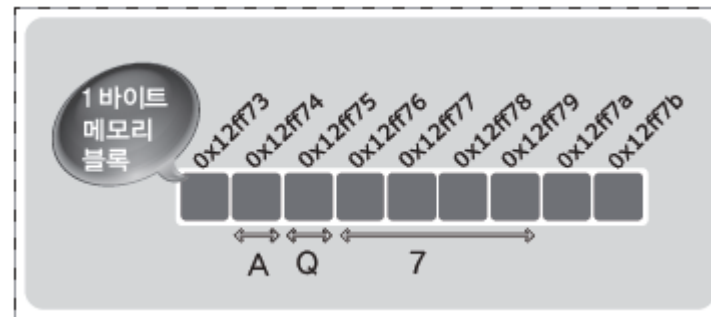
■ 메모리의 주소체계

- 주소 값은 1 바이트 단위로 할당이 된다.
- 비트 별 주소 값은 존재하지 않는다.



주소 값의 저장을 목적으로 선언되는 포인터 변수

```
int main(void)
{
    char ch1='A', ch2='Q';
    int num=7;
    . . . .
}
```



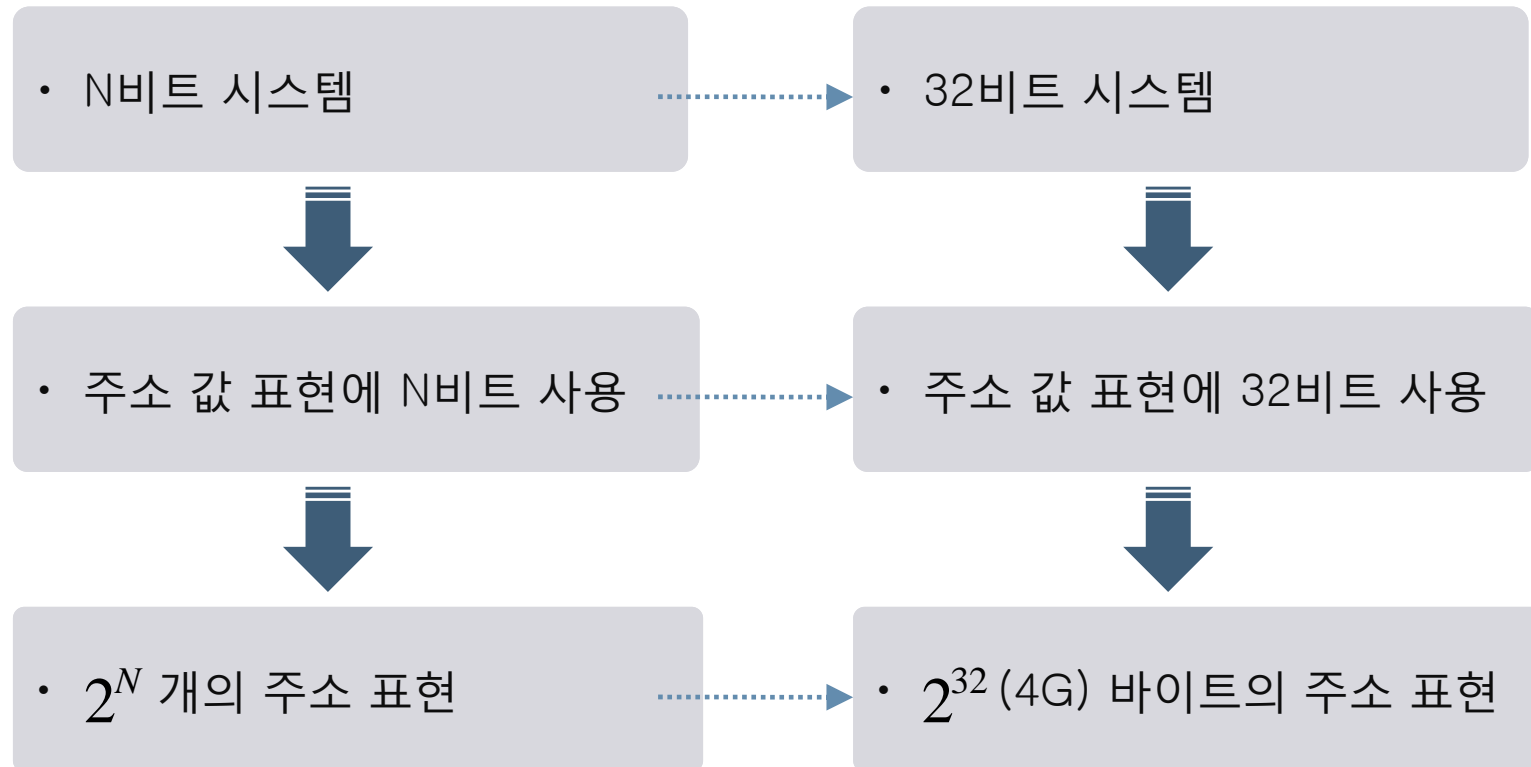
변수 num이 저장되기 시작한 주소 0x12ff76이 변수 num의 주소 값이다.

이러한 정수 형태의 주소 값을 저장하는 목적으로 선언되는 것이 포인터 변수이다.



■ 메모리의 주소 표현에 필요한 바이트 수

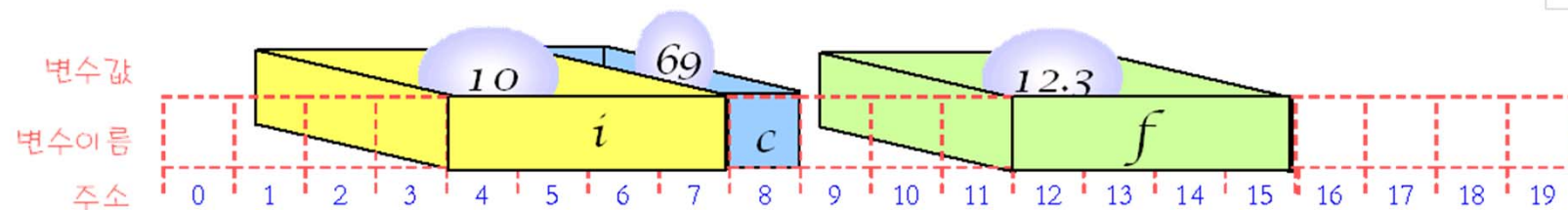
표현할 주소 값의 범위에 따라 필요한 바이트 수 달라짐



변수와 메모리

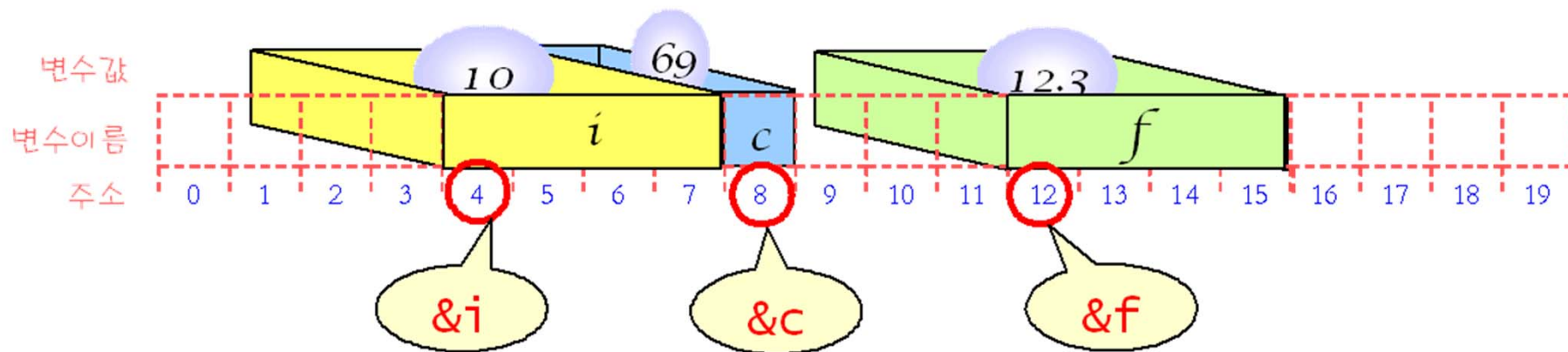
- ▶ 변수의 타입에 따라 차지하는 메모리 크기가 다름
- ▶ char형 변수: 1바이트, int형 변수: 4바이트,...

```
int main(void)
{
    int i = 10;
    char c = 69;
    float f = 12.3;
}
```



변수의 주소

- ▶ 변수의 주소를 계산하는 연산자: &
- ▶ 변수 *i*의 주소: &*i*



변수의 주소



```
int main(void)
{
    int i = 10;
    char c = 69;
    float f = 12.3;

    printf("i의 주소: %u\n", &i); // 변수 i의 주소 출력
    printf("c의 주소: %u\n", &c); // 변수 c의 주소 출력
    printf("f의 주소: %u\n", &f); // 변수 f의 주소 출력
    return 0;
}
```



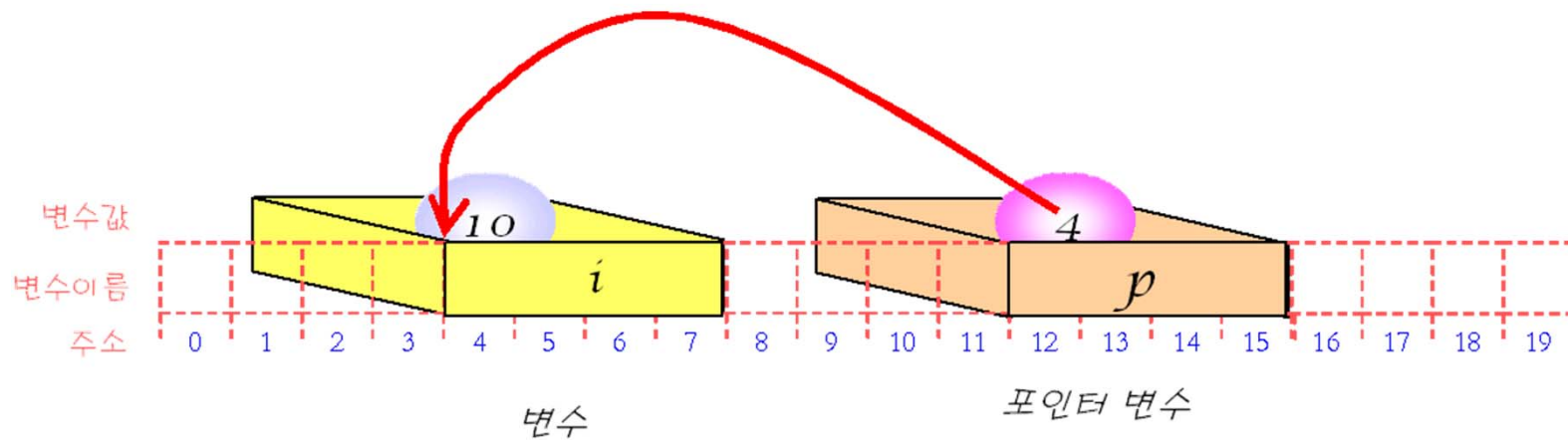
i의 주소: 1245024
c의 주소: 1245015
f의 주소: 1245000



포인터 변수의 선언 *

- ▶ 포인터: 변수의 주소를 가지고 있는 변수

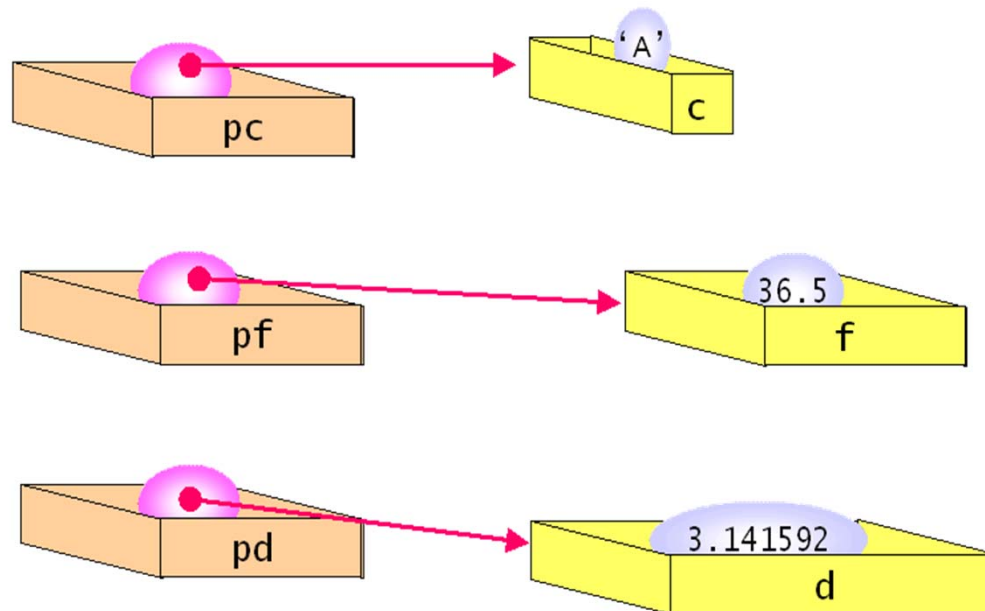
```
int i = 10;           // 정수형 변수 i 선언  
int *p = &i;          // 변수 i의 주소가 포인터 p로 대입
```



다양한 포인터의 선언

```
char c = 'A';           // 문자형 변수 c
float f = 36.5;          // 실수형 변수 f
double d = 3.141592;     // 실수형 변수 d

char *pc = &c;           // 문자를 가리키는 포인터 pc
float *pf = &f;           // 실수를 가리키는 포인터 pf
double *pd = &d;          // 실수를 가리키는 포인터 pd
```



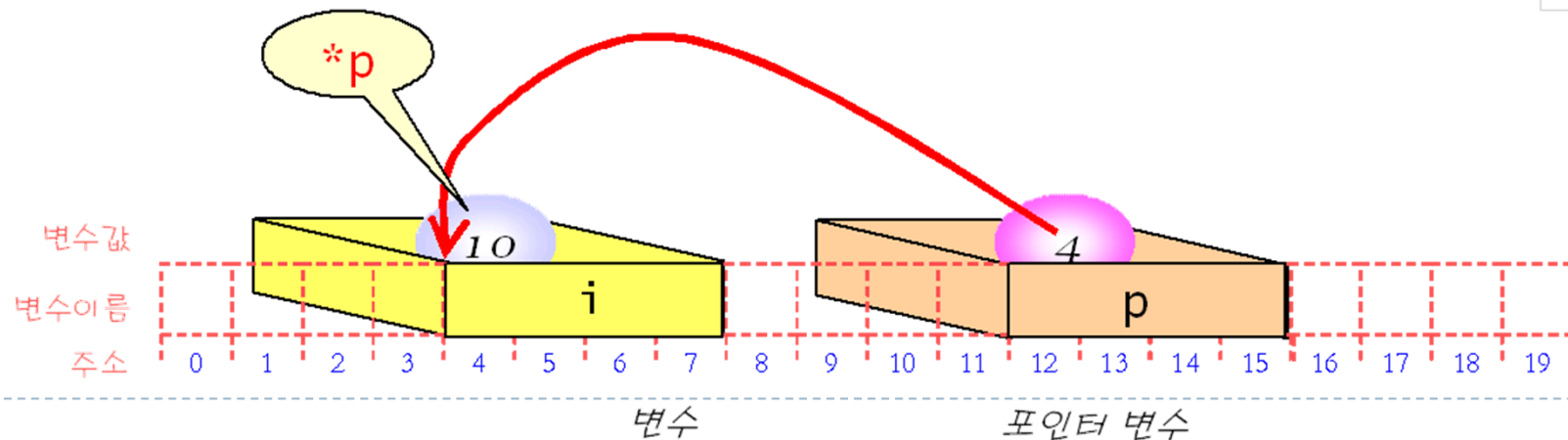
포인터

변수

간접 참조 연산자

- ▶ 간접 참조 연산자 *: 포인터가 가리키는 값을 가져오는 연산자

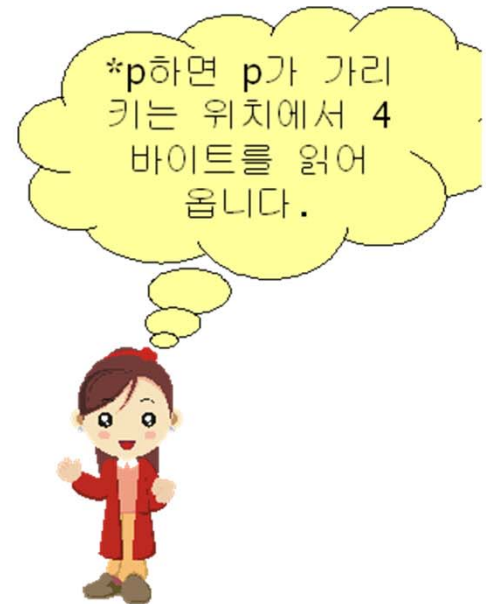
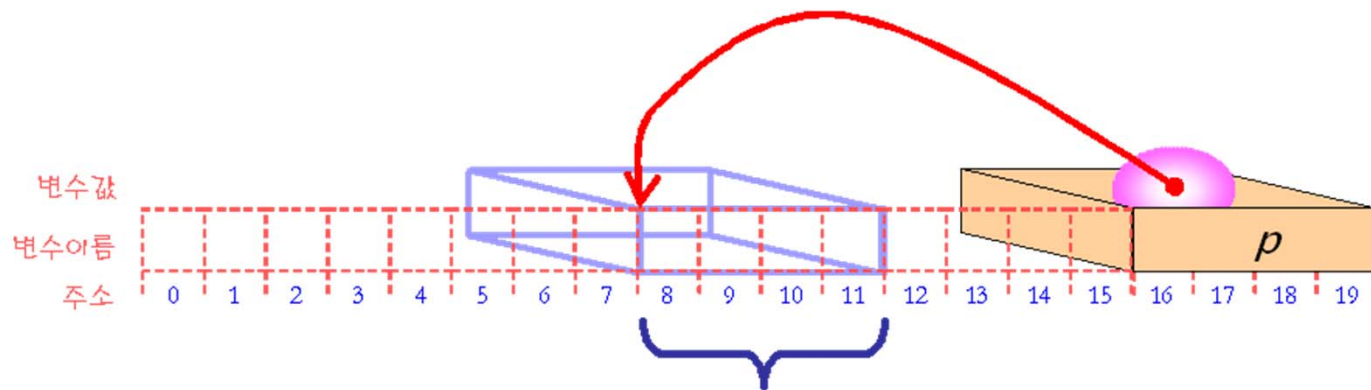
```
int i = 10;  
int *p = &i;  
int *q;  
  
printf("%d\n", *p); // 10이 출력된다.  
*p = 20;  
q = 20;  
printf("%d\n", *p); // 20이 출력된다.
```



간접 참조 연산자의 해석

- ▶ 간접 참조 연산자: 지정된 위치에서 포인터의 타입에 따라 값을 읽어 들인다.

```
int *p = 8;           // 위치 8에서 정수를 읽는다.  
char *pc = 8;         // 위치 8에서 문자를 읽는다.  
double *pd = 8;       // 위치 8에서 실수를 읽는다.
```



포인터 변수 *와 & 연산자 맛보기

“정수 7이 저장된 int형 변수 num을 선언하고 이 변수의 주소 값을 저장할 위한 포인터 변수 pnum을 선언하자. 그리고 나서 pnum에 변수 num의 주소 값을 저장하자.”



코드로 옮긴 결과

```
int main(void)
```

```
{
```

```
    int num=7;
```

```
    int * pnum; 포인터 변수 pnum의 선언  
    pnum = &num; num의 주소 값을 pnum에 저장  
    . . .
```

```
}
```

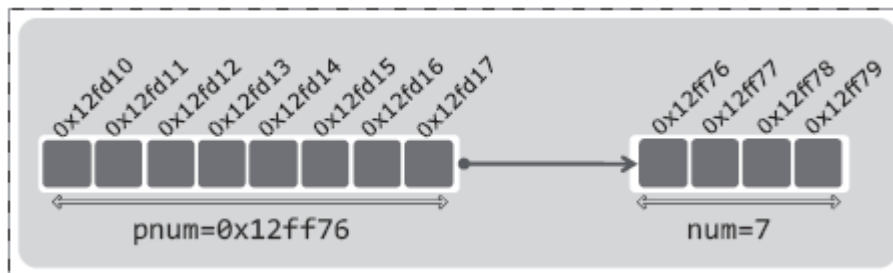
int * pnum 의 선언에서...

pnum 포인터 변수의 이름

int * int형 변수의 주소 값을 저장하는 포인터 변수의 선언



메모리의 저장상태



이 상태를 다음과 같이 표현한다.

포인터 변수 pnum이 변수 num을 가리킨다.

포인터 변수 선언하기

가리키고자 하는 변수의 자료형에 따라서 포인터 변수의 선언방법에는 차이가 있다.

포인터 변수에 저장되는 값은 모두 정수로 값의 형태는 모두 동일하지만, 그래도 선언하는 방법에 차이가 있다(차이가 있는 이유는 메모리 접근과 관련이 있다).

```
int * pnum1;
```

int * 는 int형 변수를 가리키는 pnum1의 선언을 의미함

```
double * pnum2;
```

double * 는 double형 변수를 가리키는 pnum2의 선언을 의미함

```
unsigned int * pnum3;
```

unsigned int * 는 unsigned int형 변수를 가리키는 pnum3의 선언을 의미함



일반화

```
type * ptr;
```

type형 변수의 주소 값을 저장하는 포인터 변수 ptr의 선언

포인터의 형(Type)

```
int *           int형 포인터  
int * pnum1;    int형 포인터 변수 pnum1  
  
double *        double형 포인터  
double * pnum2; double형 포인터 변수 pnum2
```



일반화

```
type *          type형 포인터  
type * ptr;     type형 포인터 변수 ptr
```

포인터 변수 선언에서 * 의 위치에 따른 차이는 없다. 즉, 다음 세 문장은 모두 동일한 포인터 변수의 선언문이다.

```
int * ptr;      // int형 포인터 변수 ptr의 선언  
int* ptr;      // int형 포인터 변수 ptr의 선언  
int * ptr;      // int형 포인터 변수 ptr의 선언
```

포인터가 가리키는 메모리를 참조하는 * 연산자

```
int main(void)
{
    int num=10;
    int * pnum=&num;
    *pnum=20;
    printf("%d", *pnum);
    . . .
}
```

pnum이 num을 가리킨다.

pnum이 가리키는 공간(변수)에 20을 저장

pnum이 가리키는 공간(변수)에 저장된 값 출력

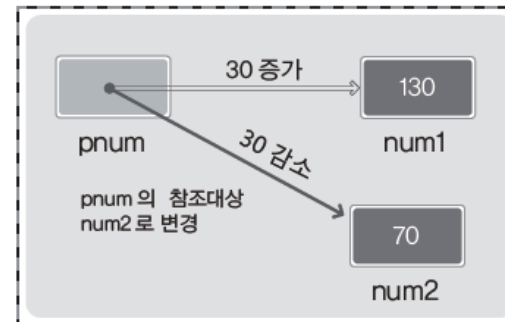
***pnum은 num을 의미한다.**
따라서 num을 놓을 자리에 *pnum을
놓을 수 있다.

```
int main(void)
{
    int num1=100, num2=100;
    int * pnum;

    pnum=&num1;    // 포인터 pnum이 num1을 가리킴
    (*pnum)+=30;    // num1+=30; 과 동일

    pnum=&num2;    // 포인터 pnum이 num2를 가리킴
    (*pnum)-=30;    // num2-=30; 과 동일

    printf("num1:%d, num2:%d \n", num1, num2);
    return 0;
}
```



실행결과

num1:130, num2:70

다양한 포인터 형이 존재하는 이유

포인터 형은 메모리 공간을 참조하는 방법의 힌트가 된다. 다양한 포인터 형을 정의한 이유는 * 연산을 통한 메모리의 접근기준을 마련하기 위함이다.

int형 포인터 변수로 * 연산을 통해 메모리(변수) 접근 시

4바이트 메모리 공간에 부호 있는 정수의 형태로 데이터를 읽고 쓴다.

double형 포인터 변수로 * 연산을 통해 메모리(변수) 접근 시

8바이트 메모리 공간에 부호 있는 실수의 형태로 데이터를 읽고 쓴다.

```
int main(void)
{
    double num=3.14;
    int * pnum=&num;
    printf("%d", *pnum);
    . . .
}
```

형 불일치! 컴파일은 된다.
pnum이 가리키는 것은 double형 변수인데, pnum이 int형 포인터 변수이므로 int형 데이터처럼 해석!

주소 값이 정수임에도 불구하고 int형 변수에 저장하지 않는 이유는 int형 변수에 저장하면 메모리 공간의 접근을 위한 * 연산이 불가능하기 때문이다.

잘못된 포인터의 사용과 널 포인터

```
int main(void)
{
    int * ptr;
    *ptr=200;
    . . . .
}
```

위험한 코드

ptr이 쓰레기 값으로 초기화 된다. 따라서 200이 저장 되는 위치는 어디인지 알 수 없다! 매우 위험한 행동!

```
int main(void)
{
    int * ptr=125;
    *ptr=10;
    . . . .
}
```

위험한 코드

포인터 변수에 125를 저장했는데 이곳이 어디인가? 역시 매우 위험한 행동!

```
int main(void)
{
    int * ptr1=0;
    int * ptr2=NULL;
    . . . .
}
```

안전한 코드

잘못된 포인터 연산을 막기 위해서 특정한 값으로 초기화하지 않는 경우에는 **널 포인터**로 초기화하는 것이 안전하다.

널 포인터 NULL은 숫자 0을 의미한다. 그리고 0은 0번지를 뜻하는 것이 아니라, 아무것도 가리키지 않는다는 의미로 해석이 된다.

포인터 예제 #1



```
#include <stdio.h>

int main(void)
{
    int i = 3000;
    int *p = &i;           // 변수와 포인터 연결

    printf("i = %d\n", i);   // 변수의 값 출력
    printf("&i = %u\n", &i);  // 변수의 주소 출력

    printf("p = %u\n", p);   // 변수의 값 출력
    printf("*p = %d\n", *p); // 포인터를 통한 간접 참조 값 출력

    return 0;
}
```



```
i = 3000
&i = 1245024
p = 1245024
*p = 3000
```



포인터 예제 #2



```
#include <stdio.h>
int main(void)
{
    int x=10, y=20;
    int *p;

    p = &x;
    printf("p = %d\n", p);
    printf("**p = %d\n\n", *p);

    p = &y;
    printf("p = %d\n", p);
    printf("**p = %d\n", *p);
    return 0;
}
```



```
p = 1245052
*p = 10
p = 1245048
*p = 20
```



포인터 예제 #3



```
#include <stdio.h>
int main(void)
{
    int i=10;
    int *p;

    p = &i;
    printf("i = %d\n", i);

    *p = 20;
    printf("i = %d\n", i);
    return 0;
}
```



```
i = 10
i = 20
```



포인터 예제 #3



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

```
int main(void)
```

```
{
```

```
    int i = 10000;
```

```
// 정수 변수 정의
```

```
    int *p, *q;
```

```
// 정수형 포인터 정의
```

```
    p = &i;
```

```
// 포인터 p와 변수 i를 연결
```

```
    q = &i;
```

```
// 포인터 q와 변수 i를 연결
```

```
    *p = *p + 1;
```

```
// 포인터 p를 통하여 1 증가
```

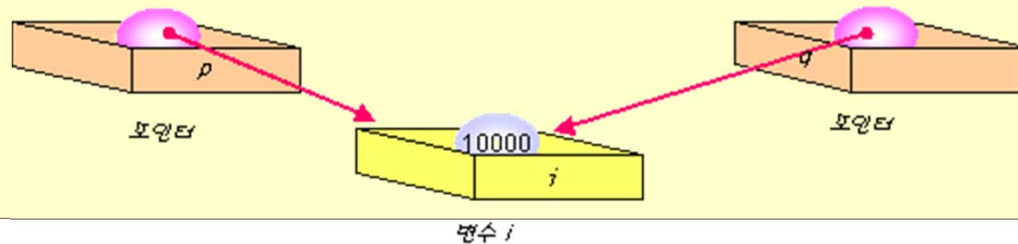
```
    *q = *q + 1;
```

```
// 포인터 q를 통하여 1 증가
```

```
    printf("i = %d\n", i);
```

```
    return 0;
```

```
}
```



```
i = 10002
```

포인터 사용시 주의점 #1

- ▶ 포인터의 타입과 변수의 타입은 일치하여야 한다.

```
#include <stdio.h>

int main(void)
{
    int i;
    double *pd;

    pd = &i;           // 오류! double형 포인터에 int형 변수의 주소를 대입
    *pd = 36.5;

    return 0;
}
```



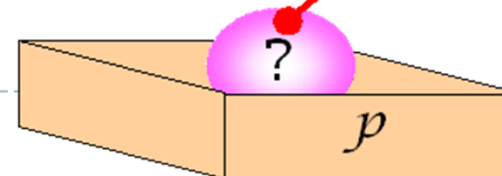
포인터 사용시 주의점 #2

- ▶ 초기화가 안된 포인터를 사용하면 안된다.

```
int main(void)
{
    int *p;           // 포인터 p는 초기화가 안되어 있음

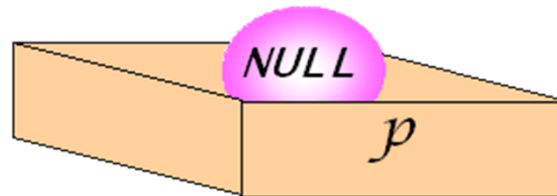
    *p = 100;         // 위험한 코드
    return 0;
}
```

주소가 잘못
된거 같은데...



포인터 사용시 주의점 #3

- ▶ 포인터가 아무것도 가리키고 있지 않는 경우에는 NULL로 초기화
- ▶ NULL 포인터를 가지고 간접 참조하면 하드웨어로 감지할 수 있다.
- ▶ 포인터의 유효성 여부 판단



포인터가 아무것
가리키지 않을 때
는 반드시 NULL
로 설정하세요.



포인터 연산

- ▶ 가능한 연산: 증가, 감소, 덧셈, 뺄셈 연산
- ▶ 증가 연산의 경우 증가되는 값은 포인터가 가리키는 객체의 크기

포인터 타입	++연산후 증가되는값
<i>char</i>	1
<i>short</i>	2
<i>int</i>	4
<i>float</i>	4
<i>double</i>	8

포인터의 증가는
일반 변수와는
약간 다릅니다.
가리키는 객체의
크기만큼
증가합니다.



증가 연산 예제



// 포인터의 증감 연산

#include <stdio.h>

int main(void)

{

char *pc;

int *pi;

double *pd;

pc = (char *)10000;

pi = (int *)10000;

pd = (double *)10000;

printf("증가 전 pc = %d, pi = %d, pd = %d\n", pc, pi, pd);

pc++;

pi++;

pd++;

printf("증가 후 pc = %d, pi = %d, pd = %d\n", pc, pi, pd);

return 0;

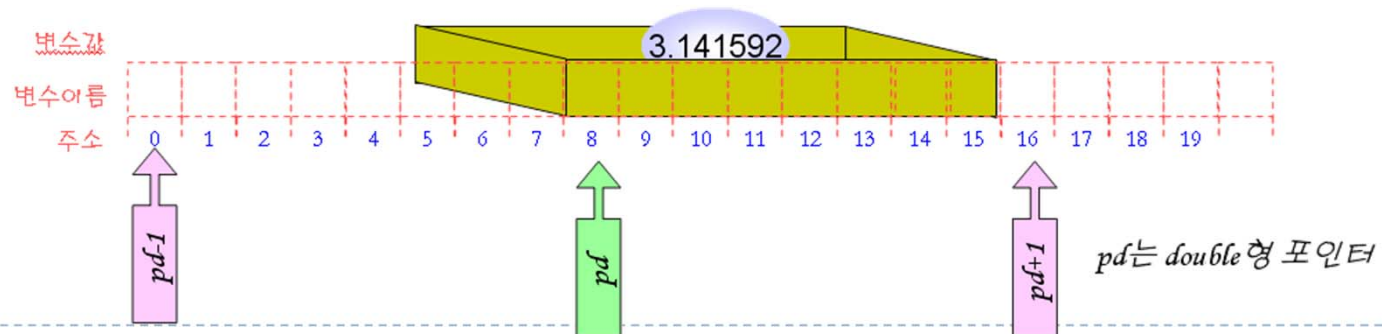
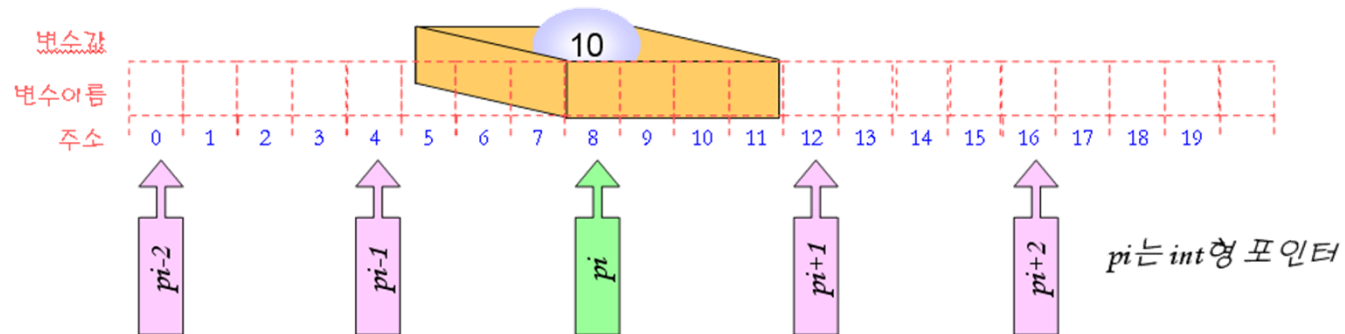
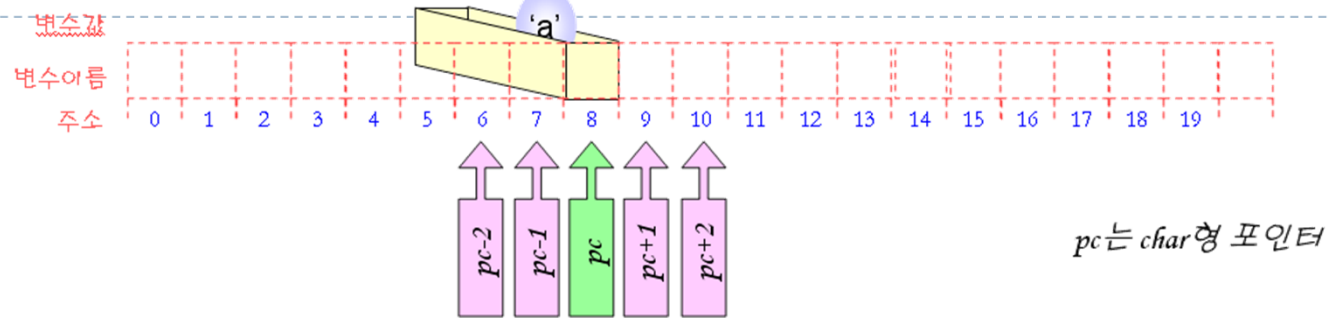
}



증가 전 pc = 10000, pi = 10000, pd = 10000

증가 후 pc = 10001, pi = 10004, pd = 10008

포인터의 증감 연산



간접 참조 연산자와 증감 연산자

수식	의미
<code>v = *p++</code>	<code>p</code> 가 가리키는 값을 <code>v</code> 에 대입한 후에 <code>p</code> 를 증가한다.
<code>v = (*p)++</code>	<code>p</code> 가 가리키는 값을 <code>v</code> 에 대입한 후에 가리키는 값을 증가한다.
<code>v = *++p</code>	<code>p</code> 를 증가시킨 후에 <code>p</code> 가 가리키는 값을 <code>v</code> 에 대입한다.
<code>v = ++*p</code>	<code>p</code> 가 가리키는 값을 가져온 후에 그 값을 증가하여 <code>v</code> 에 대입한다.



// 포인터의 증감 연산

#include <stdio.h>

int main(void)

{

int i = 10;

int *pi = &i;

printf("i = %d, pi = %p, *p= %08x\n", i, pi, *pi);

(*pi)++;

printf("i = %d, pi = %p, *p= %08x\n", i, pi, *pi);

*pi++;

printf("i = %d, pi = %p, *p= %08x\n", i, pi, *pi);

pi = &i;

printf("i = %d, pi = %p, *p= %08x\n", i, pi, *pi);

*++pi;

printf("i = %d, pi = %p, *p= %08x\n", i, pi, *pi);

++*pi;

printf("i = %d, pi = %p, *p= %08x\n", i, pi, *pi);

return 0;

}



i = 10, pi = 0012FF60

i = 11, pi = 0012FF60

i = 11, pi = 0012FF60

i = 11, pi = 0012FF64

포인터 맵 그려보기

```
int i = 0x0001;
int *pi1 = &i;
int *pi2 = &i;
int arr[3] = { 100, 2000, 3000 };
int j = 3;
char k = 'A';
char *pk1 = &k;
char *pk2 = &k + 1;
char str[] = "Hello World! 123!";
int z = 0;
unsigned char const *p = (unsigned char const *)&p;
int offset, dumpStart;
dumpStart = (unsigned int)p % 16;
for (offset = -dumpStart; (p+offset) < &i+1; offset++)
{
    if (0 == ((unsigned int)p + offset) % 16)
        printf("\n [%x] :", (p+offset));
    printf("%02X ", *(p + offset));
}
printf("\n");
}
```

결과를 적고,
각 변수의 위치에 표시하기

```
C:\Windows\system32\cmd.exe

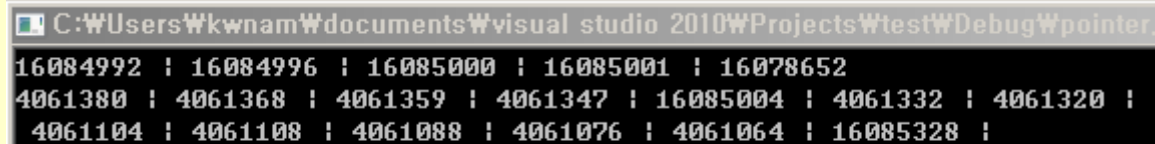
[3df9f0] :CC CC CC CC CC CC CC CC F8 F9 3D 00 CC CC CC CC
[3dfa00] :CC CC CC CC 00 00 00 00 CC CC CC CC CC CC CC CC
[3dfa10] :48 65 6C 6C 6F 20 57 6F 72 6C 64 21 20 31 32 33
[3dfa20] :21 00 CC CC CC CC CC CC CC CC CC CC 48 FA 3D 00
[3dfa30] :CC CC CC CC CC CC CC CC 47 FA 3D 00 CC CC CC CC
[3dfa40] :CC CC CC CC CC CC CC 41 CC CC CC CC CC CC CC CC
[3dfa50] :03 00 00 00 CC CC CC CC CC CC CC CC 64 00 00 00
[3dfa60] :D0 07 00 00 B8 0B 00 00 CC CC CC CC CC CC CC CC
[3dfa70] :88 FA 3D 00 CC CC CC CC CC CC CC CC 88 FA 3D 00
[3dfa80] :CC CC CC CC CC CC CC CC 01 00 00 00
계속하려면 아무 키나 누르십시오 . . .
```

변수의 실제 위치와 포인터

```
int xa = 1;
int xb = 13;
char xc = 2;
char xd = 15;
const int xe = 3;
```

```
int plus( int pa, int pb)
{
    int pc = 0;          int pd = 12;          int pe = 3;
    static int pf = 0;
    printf(" %u | %u | %u | %u | %u | %u \n", &pa, &pb, &pc, &pd, &pe, &pf );
    return pe;
}
```

```
void main(void)
{
    int ma = 5 ;          int mb = 11;          char mc = 6;
    char md = 14;
    static int me = 7;    const int mf = 8;
    int mg = 9;
    printf("%u | %u | %u | %u | %u \n", &xa, &xb, &xc, &xd, &xe);
    printf("%u | %u | %u | %u | %u | %u | %u \n", &ma, &mb, &mc, &md, &me, &mf, &mg);
    mg = plus( ma, mb );
}
```



C:\Users\Wkwnam\documents\visual studio 2010\Projects\Wtest\Debug\Wpointer.
16084992 | 16084996 | 16085000 | 16085004 | 16078652
4061380 | 4061368 | 4061359 | 4061347 | 16085004 | 4061332 | 4061320 |
4061104 | 4061108 | 4061088 | 4061076 | 4061064 | 16085328 |

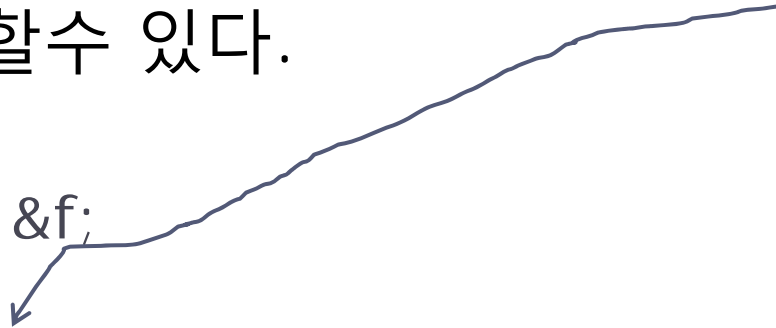
포인터의 형변환

- ▶ C언어에서는 꼭 필요한 경우에, 명시적으로 포인터의 타입을 변경할수 있다.

```
double *pd = &f;
```

```
int *pi;
```

```
pi = (int *)pd;
```



간접 참조 연산자와 증감 연산자

```
#include <stdio.h>
int main(void)
{
    char buffer[8];
    double *pd;
    int *pi;


    pd = (double *)buffer;
    *pd = 3.14;

    printf("%f\n", *pd);
    pi = (int *)buffer;
    *pi = 123;
    *(pi+1) = 456;

    printf("%d %d\n", *pi, *(pi+1));
    return 0;
}
```

char형 포인터를 double형 포인터로 변환, 배열의 이름은 char형 포인터이다.

char형 포인터를 int형 포인터로 변환



3.140000
123 456

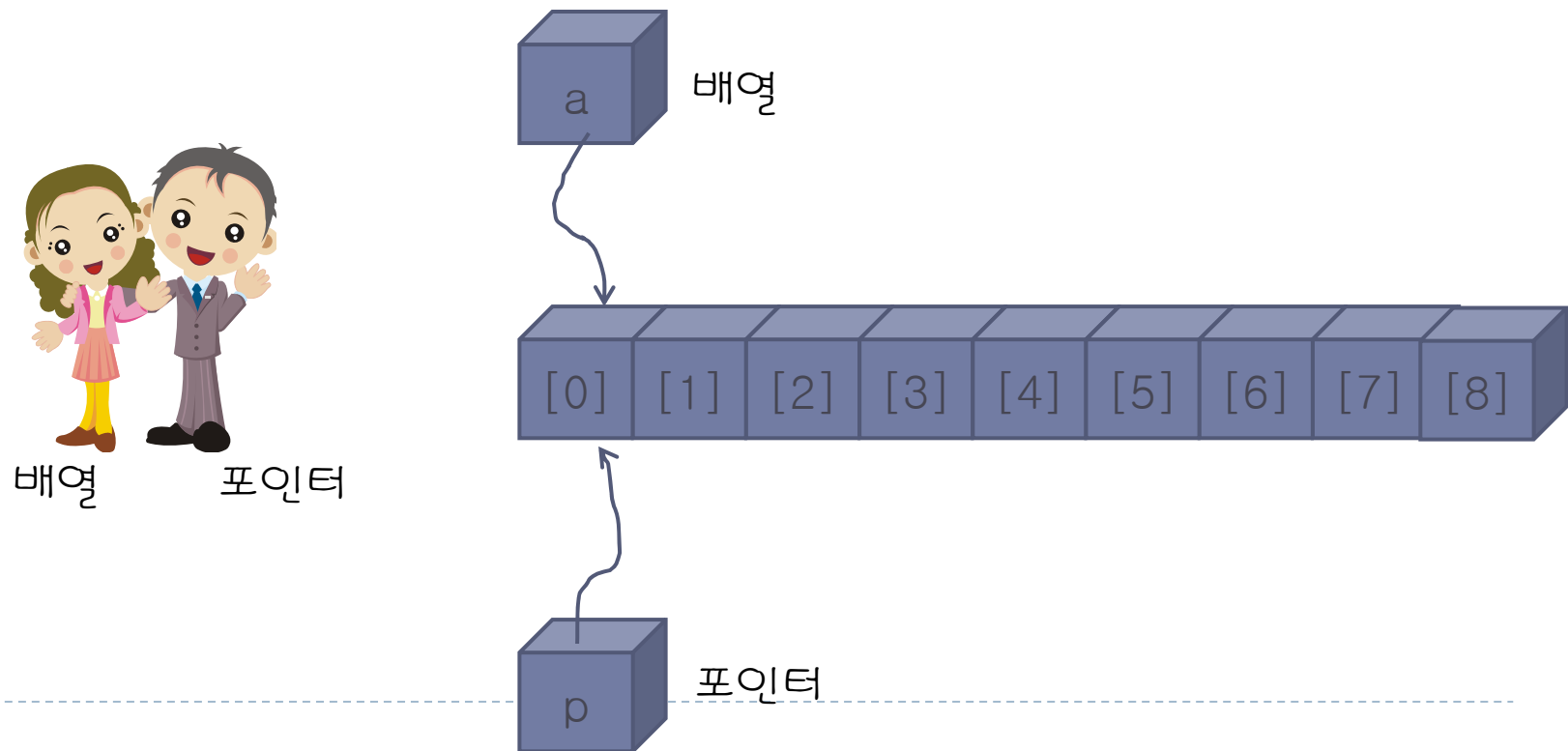
중간 점검

- ▶ 포인터에 대하여 적용할 수 있는 연산에는 어떤 것들이 있는가?
- ▶ int형 포인터 p가 80번지를 가리키고 있었다면 (p+1)은 몇 번지를 가리키는가?
- ▶ p가 포인터라고 하면 *p++와 (*p)++의 차이점은 무엇인가?
- ▶ p가 포인터라고 하면 *(p+3)의 의미는 무엇인가?



포인터와 배열

- ▶ 배열과 포인터는 아주 밀접한 관계를 가지고 있다.
- ▶ 배열 이름이 바로 포인터이다.
- ▶ 포인터는 배열처럼 사용이 가능하다.



포인터와 배열

// 포인터와 배열의 관계

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

```
int main(void)
```

```
{
```

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

```
    printf("&a[0] = %u\n", &a[0]);
```

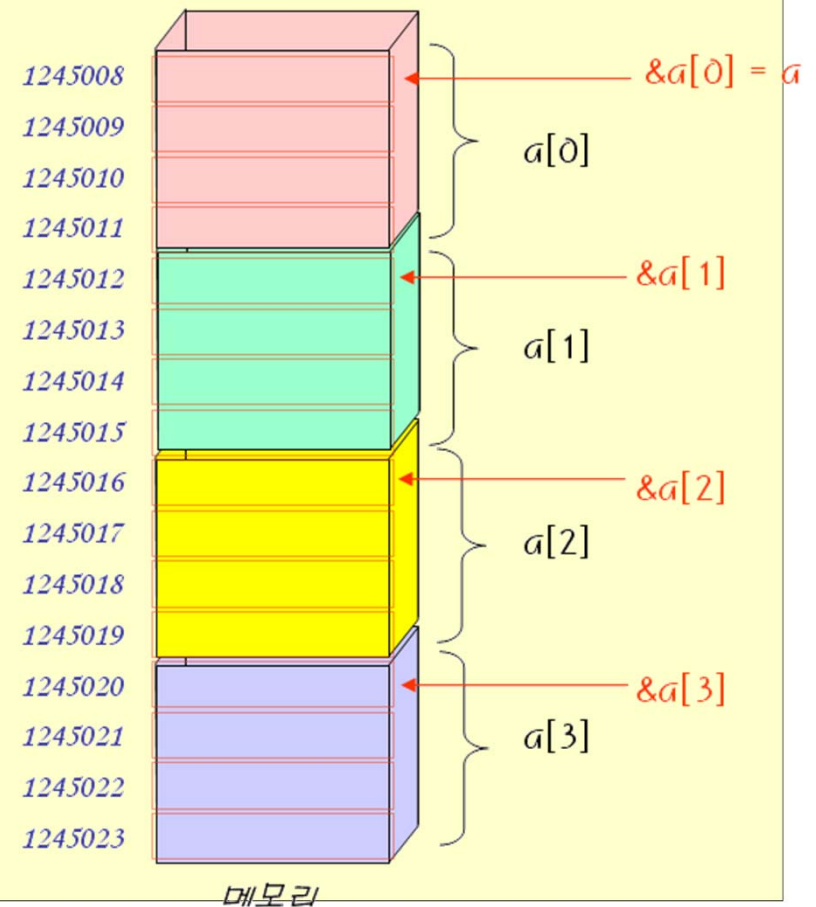
```
    printf("&a[1] = %u\n", &a[1]);
```

```
    printf("&a[2] = %u\n", &a[2]);
```

```
    printf("a = %u\n", a);
```

```
    return 0;
```

```
}
```



&a[0] = 1245008

&a[1] = 1245012

&a[2] = 1245016

a = 1245008

포인터와 배열

```
// 포인터와 배열의 관계
#include <stdio.h>
int main(void)
{
    int a[] = { 10, 20, 30, 40, 50 };

    printf("a = %u\n", a);
    printf("a + 1 = %u\n", a + 1);
    printf("*a = %d\n", *a);
    printf("*(a+1) = %d\n", *(a+1));

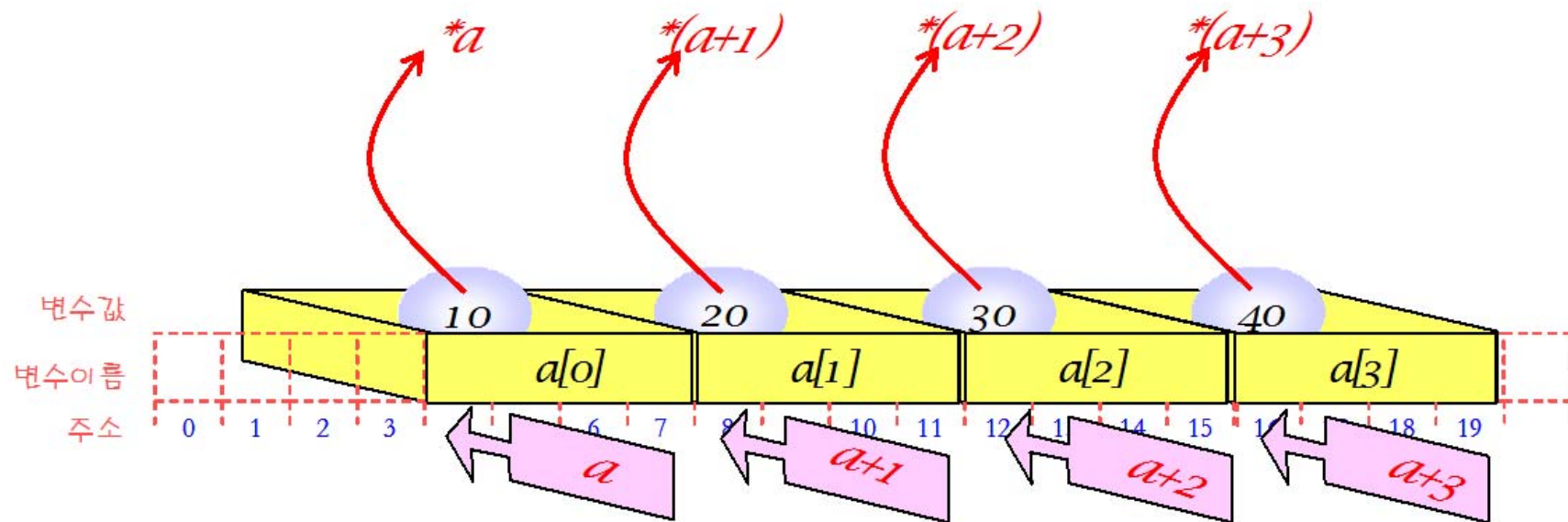
    return 0;
}
```



a = 1245008
a + 1 = 1245012
*a = 10
*(a+1) = 20

포인터와 배열

- ▶ 포인터는 배열처럼 사용할 수 있다.
- ▶ 인덱스 표기법을 포인터에 사용할 수 있다.



포인터를 배열처럼 사용

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

```
int main(void)
```

```
{
```

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

```
    int *p;
```

```
    p = a;
```

```
    printf("a[0]=%d a[1]=%d a[2]=%d \n", a[0], a[1], a[2]);
```

```
    printf("p[0]=%d p[1]=%d p[2]=%d \n\n", p[0], p[1], p[2]);
```

```
    p[0] = 60;
```

```
    p[1] = 70;
```

```
    p[2] = 80;
```

```
    printf("a[0]=%d a[1]=%d a[2]=%d \n", a[0], a[1], a[2]);
```

```
    printf("p[0]=%d p[1]=%d p[2]=%d \n", p[0], p[1], p[2]);
```

```
    return 0;
```

```
}
```

배열은 결국 포인터로
구현된다는 것을 알 수
있다.

포인터를 통하여 배열
원소를 변경할 수 있다.



```
a[0]=10 a[1]=20 a[2]=30
```

```
p[0]=10 p[1]=20 p[2]=30
```

```
a[0]=60 a[1]=70 a[2]=80
```

```
p[0]=60 p[1]=70 p[2]=80
```

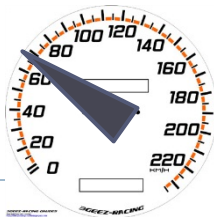
포인터를 사용한 방법의 장점

- ▶ 포인터가 인덱스 표기법보다 빠르다.
 - ▶ Why?: 인덱스를 주소로 변환할 필요가 없다.

```
int get_sum1(int a[], int n)
{
    int i;
    int sum = 0;

    for(i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

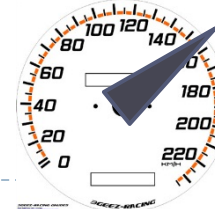
인덱스 표기법 사용



```
int get_sum2(int a[], int n)
{
    int i, sum = 0;
    int *p;

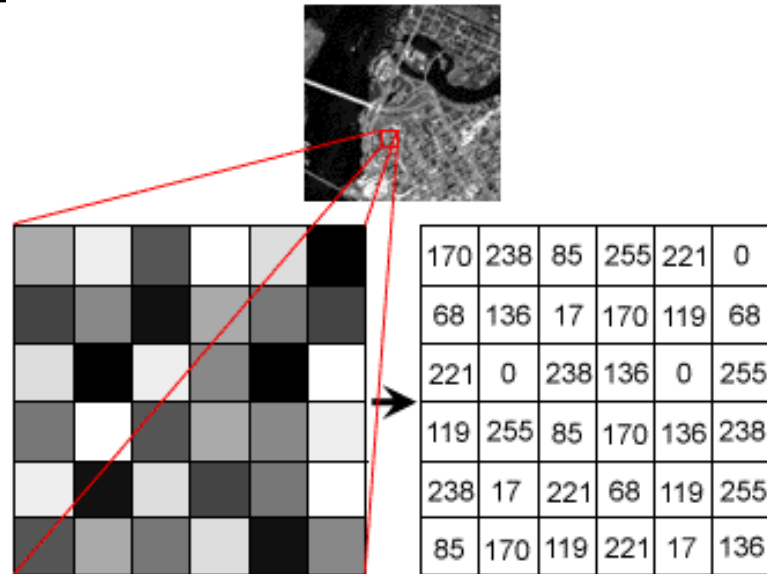
    p = a;
    for(i = 0; i < n; i++)
        sum += *p++;
    return sum;
}
```

포인터 사용

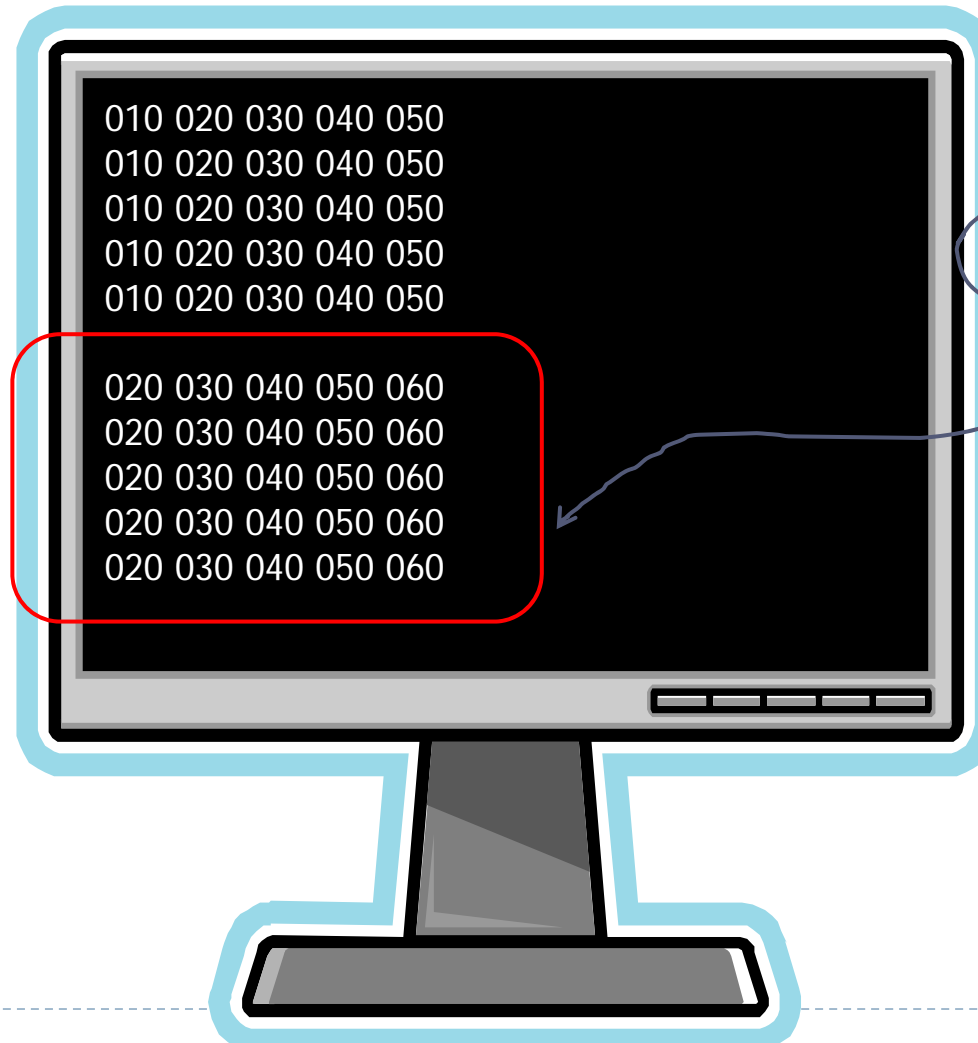


실습: 영상 처리

- ▶ 디지털 이미지는 배열을 사용하여 저장된다.
- ▶ 이미지 처리를 할 때 속도를 빠르게 하기 위하여 포인터를 사용한다.
- ▶ 이미지 내의 모든 픽셀의 값을 10씩 증가시켜보자.



실행 결과



모든 픽셀
의 값이
10씩 증가
되었다.

실습: 영상 처리

```
#include <stdio.h>
#define SIZE 5
void print_image(int image[][SIZE])
{
    int r,c;
    for(r=0;r<SIZE;r++){
        for(c=0;c<SIZE;c++){
            printf("%03d ", image[r][c]);
        }
        printf("\n");
    }
    printf("\n");
}
```



실습: 영상 처리

```
void brighten_image(int image[][SIZE])
{
    int r,c;
    int *p;
    p = &image[0][0];
    for(r=0;r<SIZE;r++){
        for(c=0;c<SIZE;c++){
            *p += 10;
            p++;
        }
    }
}
```



실습: 영상 처리

```
int main(void)
{
    int image[5][5] = {
        { 10, 20, 30, 40, 50},
        { 10, 20, 30, 40, 50},
        { 10, 20, 30, 40, 50},
        { 10, 20, 30, 40, 50},
        { 10, 20, 30, 40, 50}};

    print_image(image);
    brighten_image(image);
    print_image(image);
    return 0;
}
```



도전문제

- ▶ 포인터를 이용하지 않는 버전도 작성하여 보자. 즉 배열의 인덱스 표기법으로 위의 프로그램을 변환하여 보자.



배열의 원소를 역순으로 출력

```
#include <stdio.h>
void print_reverse(int a[], int n);

int main(void)
{
    int a[] = { 10, 20, 30, 40, 50 };

    print_reverse(a, 5);
    return 0;
}

void print_reverse(int a[], int n)
{
    int *p = a + n - 1;                // 마지막 노드를 가리킨다.

    while(p >= a)                      // 첫번째 노드까지 반복
        printf("%d\n", *p--);          // p가 가리키는 위치를 출력하고 감소
}
```



50
40
30
20
10

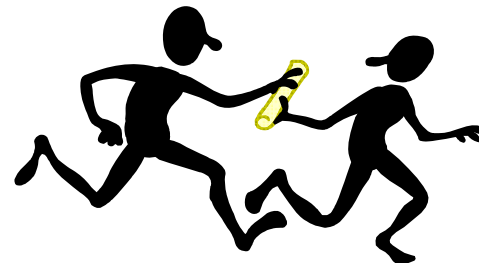
중간 점검

- ▶ 배열의 첫 번째 원소의 주소를 계산하는 2가지 방법을 설명하라.
- ▶ 배열 `a[]`에서 `*a`의 의미는 무엇인가?
- ▶ 배열의 이름에 다른 변수의 주소를 대입할 수 있는가?
- ▶ 포인터를 이용하여 배열의 원소들을 참조할 수 있는가?
- ▶ 포인터를 배열의 이름처럼 사용할 수 있는가?



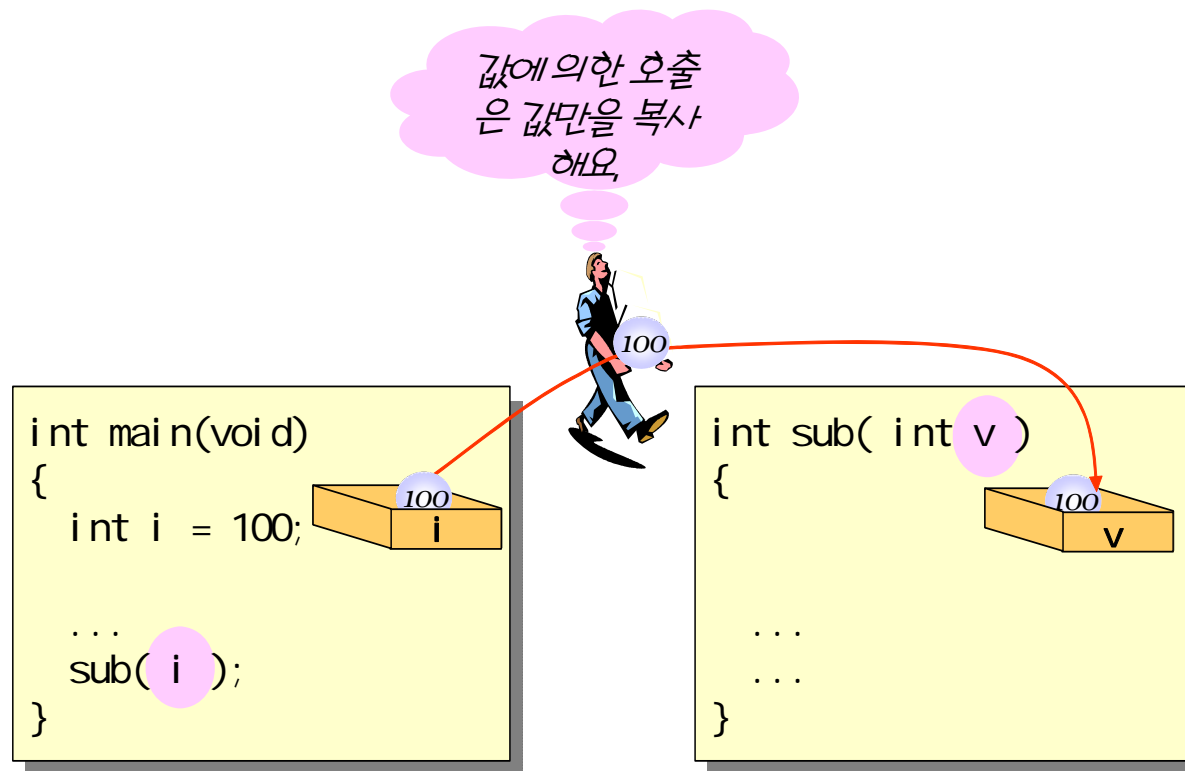
인수 전달 방법

- ▶ 함수 호출 시에 인수 전달 방법
 - ▶ 값에 의한 호출(call by value)
 - ▶ C에서 기본적인 방법
 - ▶ 참조에 의한 호출(call by reference)
 - ▶ C에서는 포인터를 이용하여 흉내 낼 수 있다.



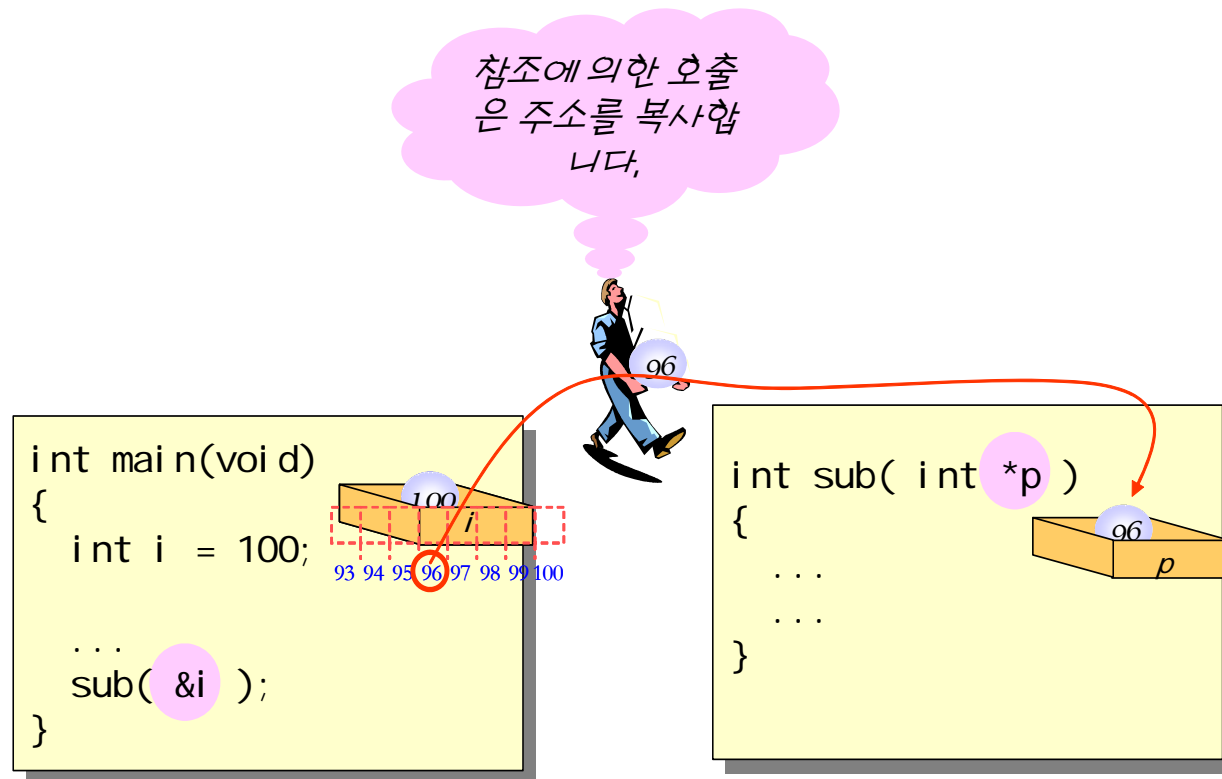
값에 의한 호출

- ▶ 함수 호출시에 변수의 값을 함수에 전달



참조에 의한 호출

- ▶ 함수 호출시에 변수의 주소를 함수의 매개 변수로 전달



swap() 함수 #1

- ▶ 변수 2개의 값을 바꾸는 작업을 함수로 작성

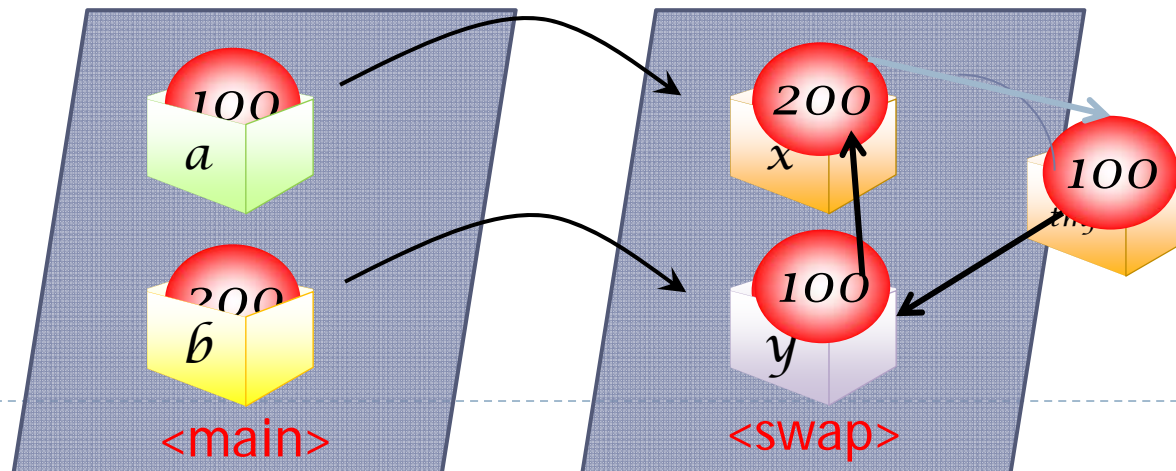
```
#include <stdio.h>
void swap(int x, int y);
int main(void)
{
    int a = 100, b = 200;

    swap(a, b);
    return 0;
}
```

```
void swap(int x, int y)
{
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}
```

함수 호출시에 값만 복사된다.



swap() 함수 #2

▶ 포인터를 이용

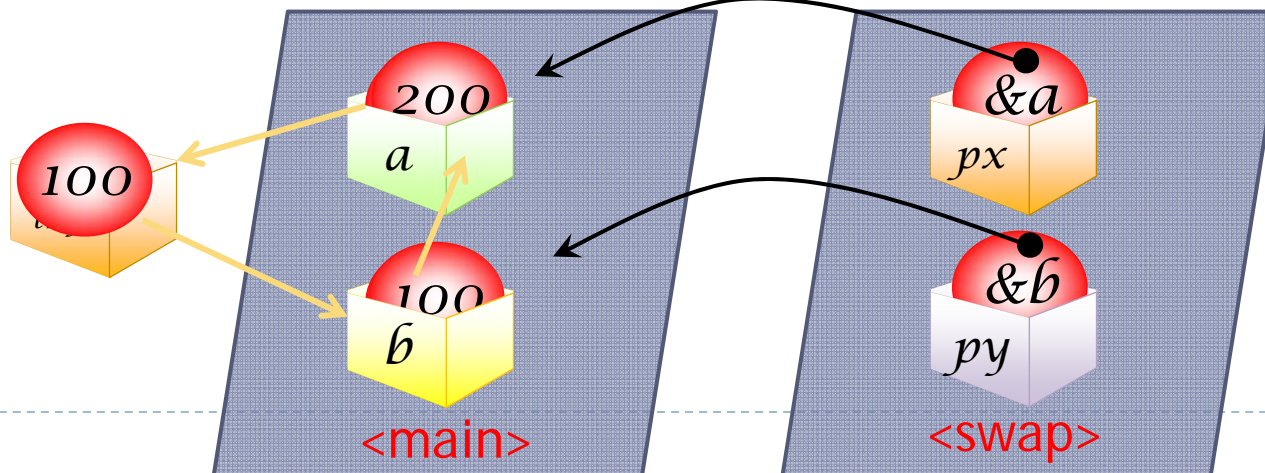
```
#include <stdio.h>
void swap(int x, int y);
int main(void)
{
    int a = 100, b = 200;
    swap(&a, &b);

    return 0;
}
```

```
void swap(int *px, int *py)
{
    int tmp;

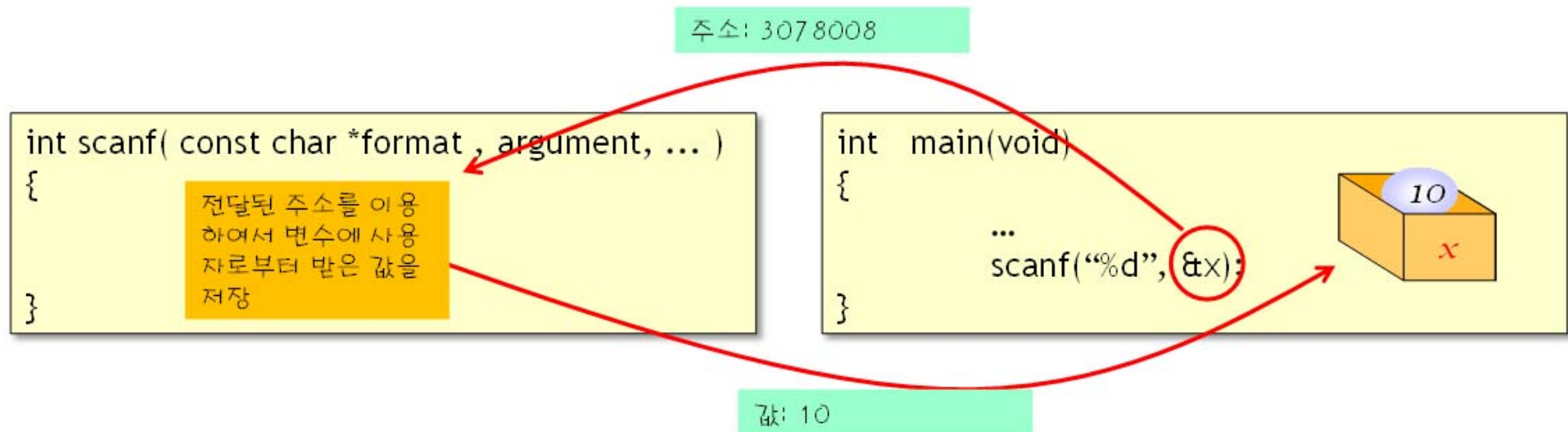
    tmp = *px;
    *px = *py;
    *py = tmp;
}
```

함수 호출시에 주소가 복사된다.



scanf() 함수

- ▶ 변수에 값을 저장하기 위하여 변수의 주소를 받는다.



2개 이상의 결과를 반환

```
#include <stdio.h>
// 기울기와 y절편을 계산
int get_line_parameter(int x1, int y1, int x2, int y2, float *slope, float *yintercept)
{
    if( x1 == x2 )
        return -1;
    else {
        *slope = (float)(y2 - y1)/(float)(x2 - x1);
        *yintercept = y1 - (*slope)*x1;
        return 0;
    }
}

int main(void)
{
    float s, y;
    if( get_line_parameter(3,3,6,6,&s,&y) == -1 )
        printf("에러\n");
    else
        printf("기울기는 %f, y절편은 %f\n", s, y);
    return 0;
}
```

기울기와 Y절편을 인수로 전달



기울기는 1.000000, y절편은 0.000000

배열 매개 변수

▶ 일반 매개 변수 vs 배열 매개 변수

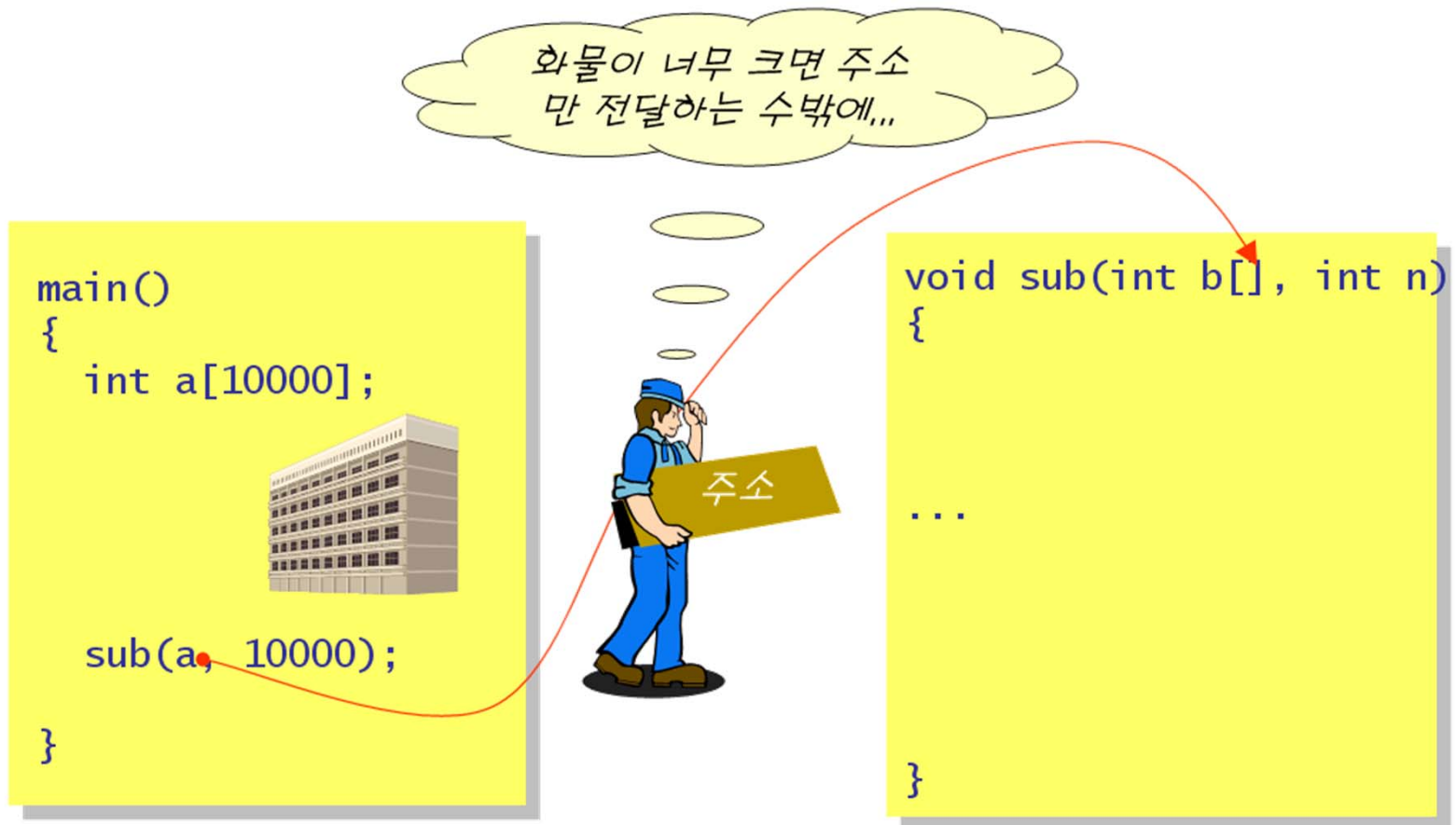
```
// 매개 변수 x에 기억 장소가 할당  
void sub(int x)  
{  
    ...  
}
```

```
// b에 기억 장소가 할당되지 않는다.  
void sub( int b[] )  
{  
    ...  
}
```

▶ Why? -> 배열을 함수로 복사하려면 많은 시간 소모



배열 매개 변수



예제

```
#include <stdio.h>
void sub(int b[], int n);
```

```
int main(void)
{
```

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

```
    printf("%d %d %d\n", a[0], a[1], a[2]);
```

```
    sub(a, 3);
```

```
    printf("%d %d %d\n", a[0], a[1], a[2]);
```

```
    return 0;
```

```
}
```

```
void sub(int b[], int n)
```

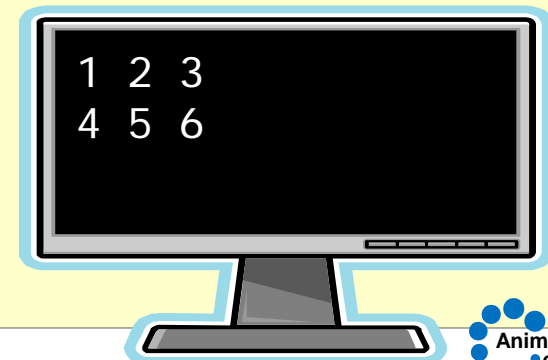
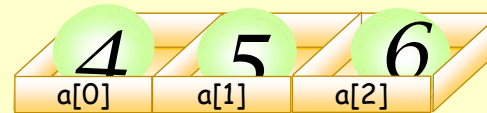
```
{
```

```
    b[0] = 4;
```

```
    b[1] = 5;
```

```
    b[2] = 6;
```

```
}
```



포인터를 반환할 때 주의점

- ▶ 함수가 종료되더라도 남아 있는 변수의 주소를 반환하여야 한다.
- ▶ 지역 변수의 주소를 반환하면, 함수가 종료되면 사라지기 때문에 오류

```
int *add(int x, int y)
{
    int result;

    result = x + y;
    return &result;
}
```

지역변수 result는
함수가 종료되면 소멸되므로 그
주소를 반환하면 안된다!



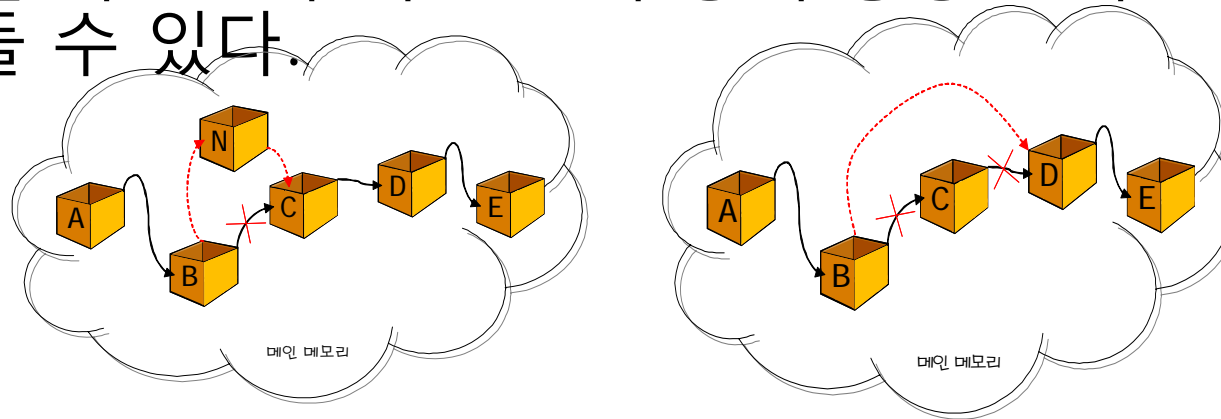
중간 점검

- ▶ 함수에 매개 변수로 변수의 복사본이 전달되는 것을 _____라고 한다.
- ▶ 함수에 매개 변수로 변수의 원본이 전달되는 것을 _____라고 한다.
- ▶ 배열을 함수의 매개 변수로 지정하는 경우, 배열의 복사가 일어나는가?



포인터 사용의 장점

- ▶ 연결 리스트나 이진 트리 등의 향상된 자료 구조를 만들 수 있다.



- ▶ 참조에 의한 호출
 - ▶ 포인터를 매개 변수로 이용하여 함수 외부의 변수의 값을 변경할 수 있다.
- ▶ 동적 메모리 할당
 - ▶ 17장에서 다룬다.