
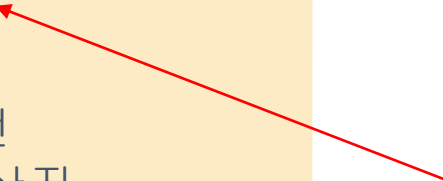


제 11장 포인터

이번 장에서 학습할 내용

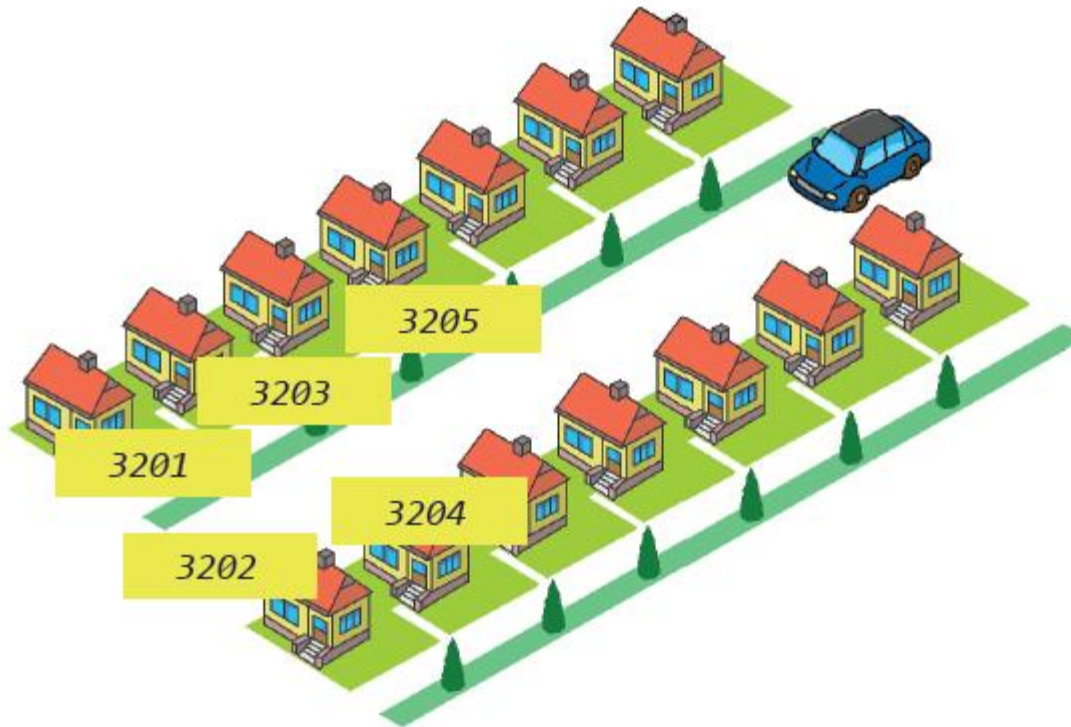
- 
- 포인터이란?
 - 변수의 주소
 - 포인터의 선언
 - 간접 참조 연산자
 - 포인터 연산
 - 포인터와 배열
 - 포인터와 함수
- 

이번 장에서는
포인터의 기초적인
지식을 학습한다.



포인터란?

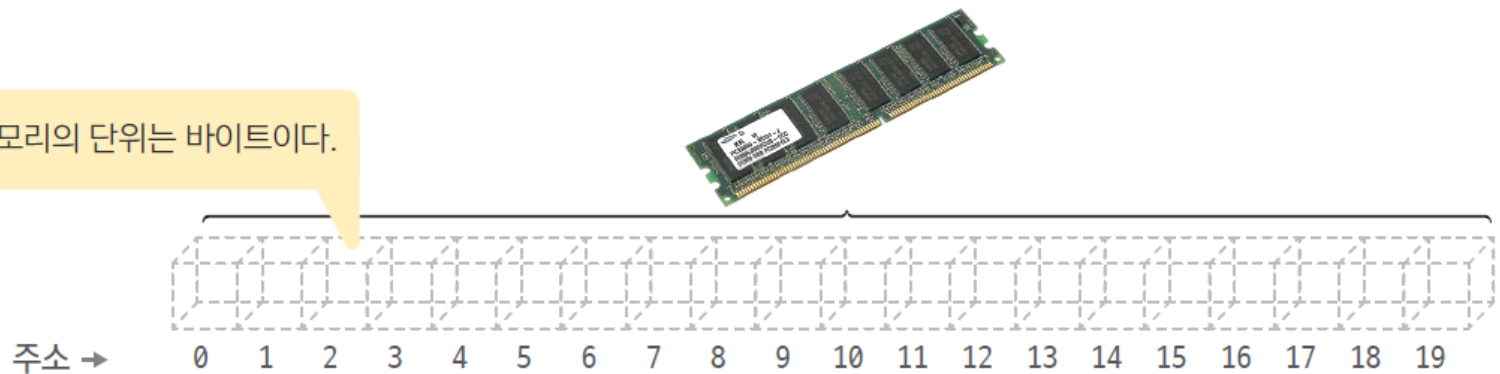
- *포인터(pointer)*: 주소를 가지고 있는 변수



변수에 어디에 저장되는가?

- 변수는 메모리에 저장된다.
- 메모리는 바이트 단위로 액세스된다.
 - 첫번째 바이트의 주소는 0, 두번째 바이트는 1,...

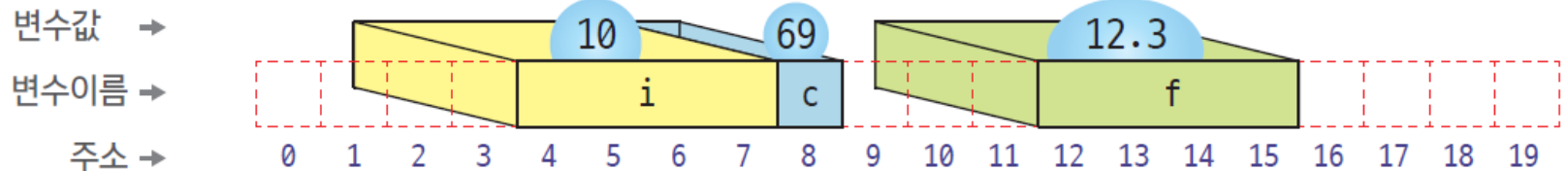
메모리의 단위는 바이트이다.



변수와 메모리

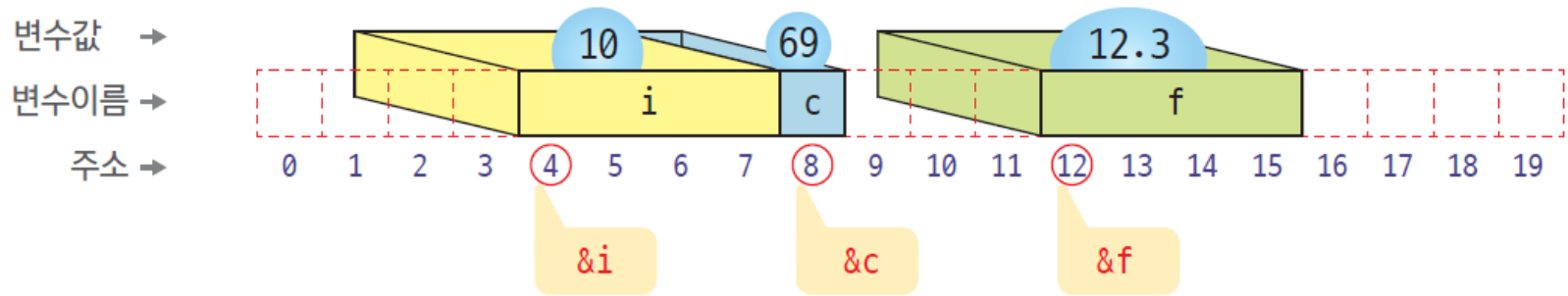
- 변수의 크기에 따라서 차지하는 메모리 공간이 달라진다.
- char형 변수: 1바이트, int형 변수: 4바이트,...

```
int main(void)
{
    int i = 10;
    char c = 69;
    float f = 12.3;
    return 0;
}
```



변수의 주소

- 변수의 주소를 계산하는 연산자: &
- 변수 i의 주소: &i



변수의 주소

```
int main(void)
```

```
{
```

```
    int i = 10;
```

```
    char c = 69;
```

```
    float f = 12.3;
```

```
    printf("i의 주소: %p\n", &i);
```

```
    printf("c의 주소: %p\n", &c);
```

```
    printf("f의 주소: %p\n", &f);
```

```
    return 0;
```

```
}
```

```
// 변수 i의 주소 출력
```

```
// 변수 c의 주소 출력
```

```
// 변수 f의 주소 출력
```

프로그램을 실행할 때
마다 주소는 달라집니
다.



```
i의 주소: 0000003D69DDF974  
c의 주소: 0000003D69DDF994  
f의 주소: 0000003D69DDF9B8
```

주의

- 여러 개의 포인터 변수를 한 줄에 선언할 때는 주의하여야 한다.
다음과 같이 선언하는 것은 잘못되었다.
 - `int *p1, p2, p3;` // (×) p2와 p3는 정수형 변수가 된다.
- 올바르게 선언하려면 다음과 같이 하여야 한다.
 - `int *p1, *p2, *p3;` // (○) p2와 p3는 int형 포인터 변수가 된다.

포인터의 선언

- 포인터: 변수의 주소를 가지고 있는 변수

Syntax

포인터 선언

예

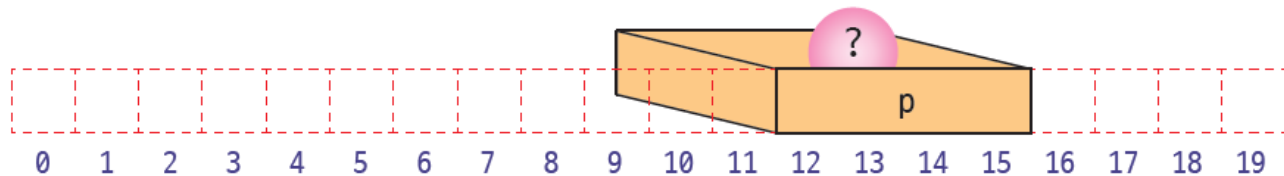
정수를 가리키는 포인터 p
`int *p;`

포인터 변수

변수값 →

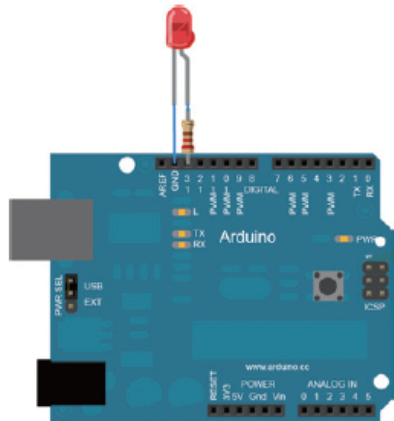
변수이름 →

주소 →



포인터의 초기화: 절대 주소로 초기화

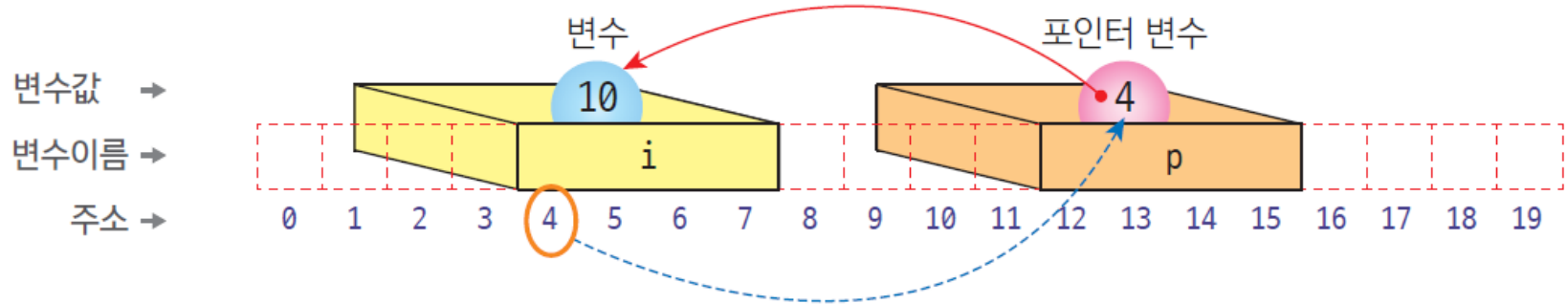
- 아두이노와 같은 엠베디드 시스템에서는 가능
- 윈도우에서는 안됨
 - 보안과 안정성을 위해 절대주소 포인터 초기화는 허용되지 않음
 - 반드시 운영체제가 할당해준 주소나, malloc, new 등을 통해 정상적으로 확보한 메모리 주소



```
char *p = (char *) 0x30000000;
```

포인터와 변수의 연결

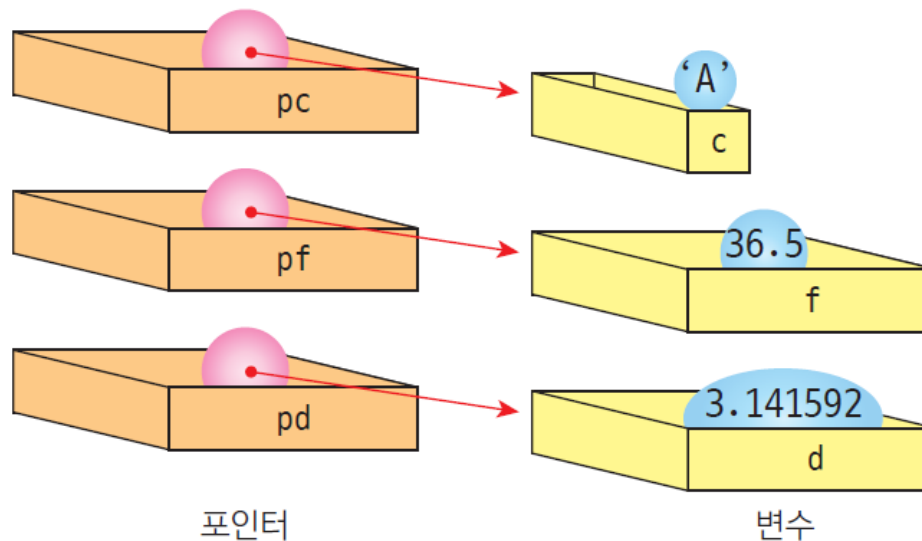
```
int i = 10;      // 정수형 변수 i 선언
int *p;          // 포인터 변수 p 선언
p = &i;          // 변수 i의 주소가 포인터 p로 대입
```



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
```



예제

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    double f = 12.3;
    int* pi = NULL;

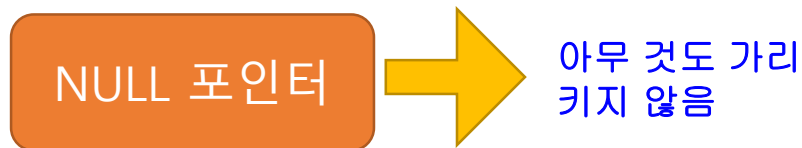
    double* pf = NULL;
    pi = &i;
    pf = &f;

    printf("%p %p \n", pi, &i);
    printf("%p %p \n", pf, &f);
    return 0;
}
```

```
0000002AFF8FFB24 0000002AFF8FFB24
0000002AFF8FFB48 0000002AFF8FFB48
```

참고

- NULL은 stdio.h 헤더 파일에 다음과 같이 정의된 포인터 상수로 0 번지를 의미한다.
 - #define NULL ((void *)0)
- 0번지는 일반적으로는 사용할 수 없다(CPU가 인터럽트를 위하여 사용한다). 따라서 포인터 변수의 값이 0이면 아무 것도 가리키고 있지 않다고 판단할 수 있다.



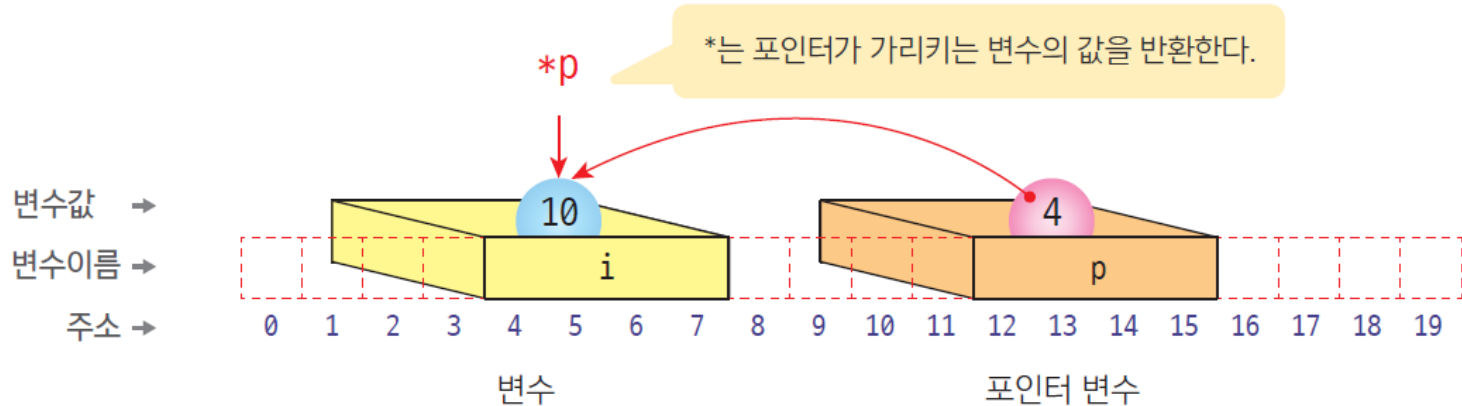
간접 참조 연산자

- 간접 참조 연산자 *: 포인터가 가리키는 값을 가져오는 연산자

```
int i = 10;
```

```
int* p;  
p = &i;
```

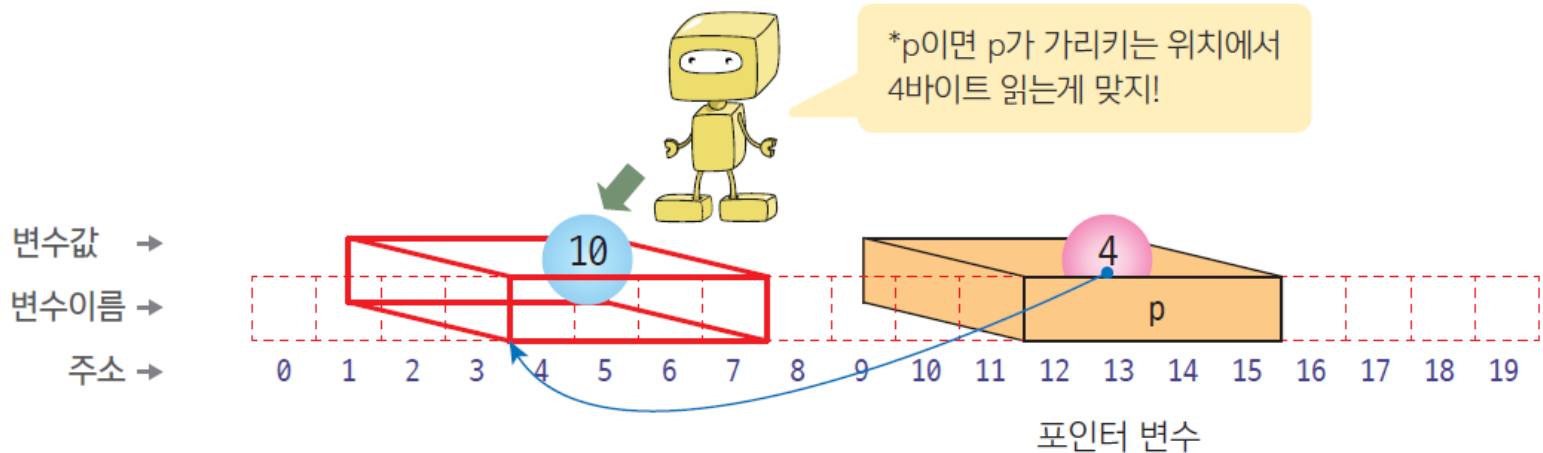
```
printf("%d \n", *p);
```



간접 참조 연산자의 해석

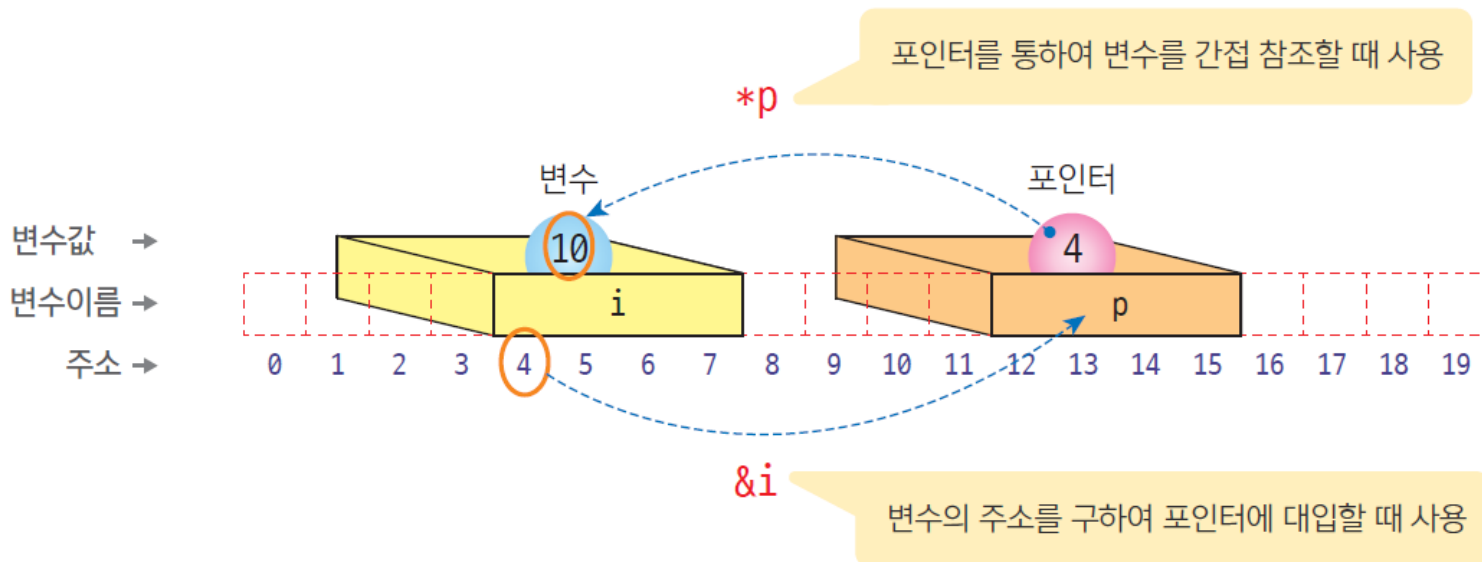
- 간접 참조 연산자: 지정된 위치에서 포인터의 타입에 따라 값을 읽어 들인다.

```
int *pi = (int *)10000;  
char *pc = (char *)10000;  
double *pd = (double *)10000;
```



& 연산자와 * 연산자

- & 연산자: 변수의 주소를 반환한다
- * 연산자: 포인터가 가리키는 곳의 내용을 반환한다.

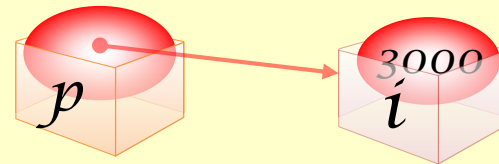


포인터 예제 #1

```
#include <stdio.h>
int main(void)
{
    int i = 3000;
    int *p=NULL;

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

    return 0;
}
```



p = 0000006DEA0FFBD4
&i = 0000006DEA0FFBD4

i = 3000
*p = 3000

포인터 예제 #2

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

```
int main(void)
```

```
{
```

```
    int x=10, y=20;
```

```
    int *p;
```

```
    p = &x;
```

```
    printf("p = %p\n", p);
```

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

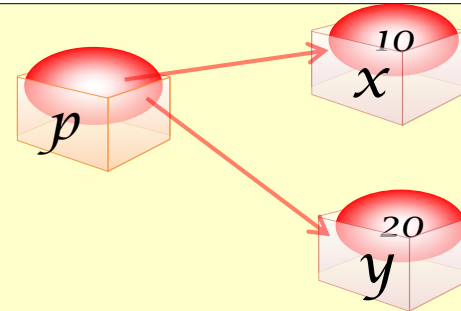
```
    p = &y;
```

```
    printf("p = %p\n", p);
```

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

```
    return 0;
```

```
}
```



```
p = 0000007A8F3AF974
```

```
*p = 10
```

```
p = 0000007A8F3AF994
```

```
*p = 20
```

포인터 예제 #3

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

```
int main(void)
```

```
{
```

```
    int i=10;
```

```
    int *p;
```

```
    p = &i;
```

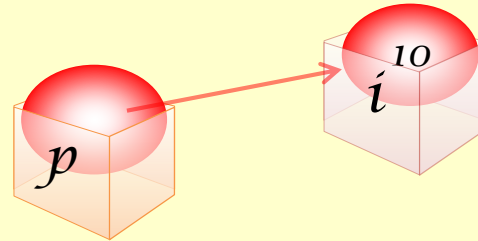
```
    printf("i = %d\n", i);
```

```
    *p = 20;
```

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

```
    return 0;
```

```
}
```



포인터를 통하여 변수의 값을 변경한다.

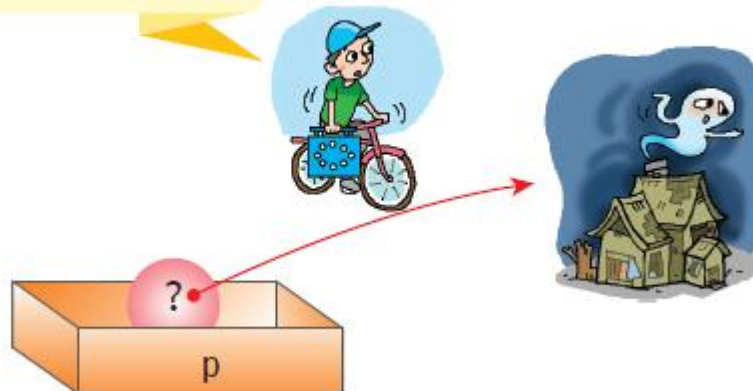
```
i = 10  
i = 20
```

포인터 사용시 주의점

- 초기화가 안된 포인터를 사용하면 안된다.

```
int main(void)
{
    int *p;           // 포인터 p는 초기화가 안되어 있음
    *p = 100;         // 위험한 코드
    return 0;
}
```

주소가 잘못된거 같은데...



포인터 사용시 주의점

- 포인터는 C언어의 강점이자 단점이다.
- 개발자가 책임감을 가지고 사용하여야 한다.
- 포인터를 사용할 때는 스파이더맨 영화에 나왔던, 다음과 같은 말을 항상 기억하자.

“With great power comes
great responsibility”

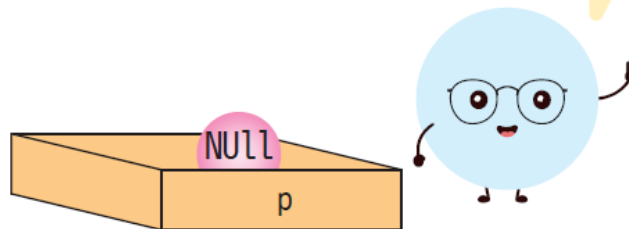


포인터 사용시 주의점

- 포인터가 아무것도 가리키고 있지 않는 경우에는 NULL로 초기화
- NULL 포인터를 가지고 간접 참조하면 하드웨어로 감지할 수 있다.

```
int *p = NULL;
```

포인터가 아무것 가리키지 않을 때는
반드시 NULL로 설정하세요.



포인터 사용시 주의점

- 포인터의 타입과 변수의 타입은 일치하여야 한다.

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

```
int main(void)
```

```
{
```

```
    int i;
```

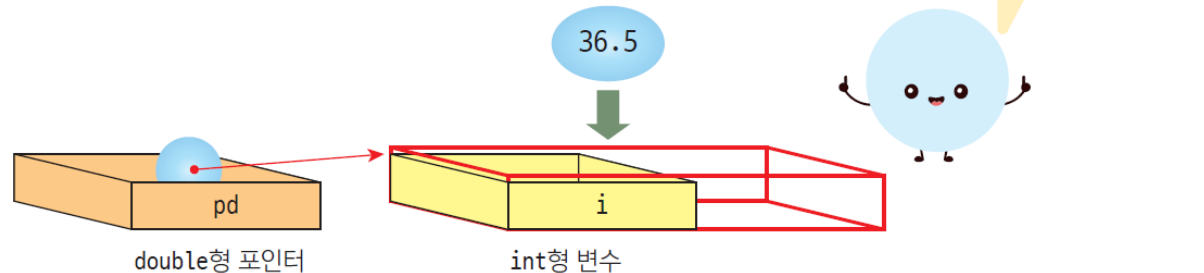
```
    double *pd;
```

```
    pd = &i;           // 오류!
```

```
    *pd = 36.5;
```

```
    return 0;
```

```
}
```



포인터 연산

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

포인터 타입	++연산후 증가되는값
char	1
short	2
int	4
float	4
double	8

증가 연산 예제

```
// 포인터의 증감 연산
#include <stdio.h>

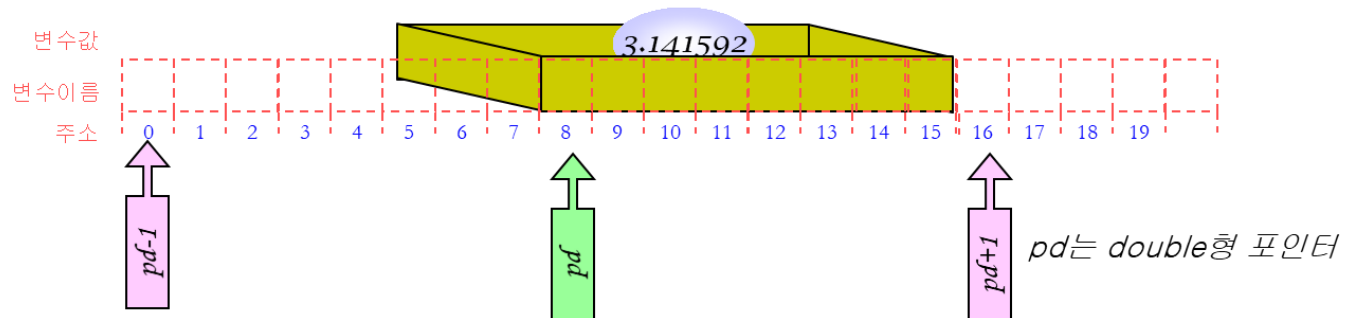
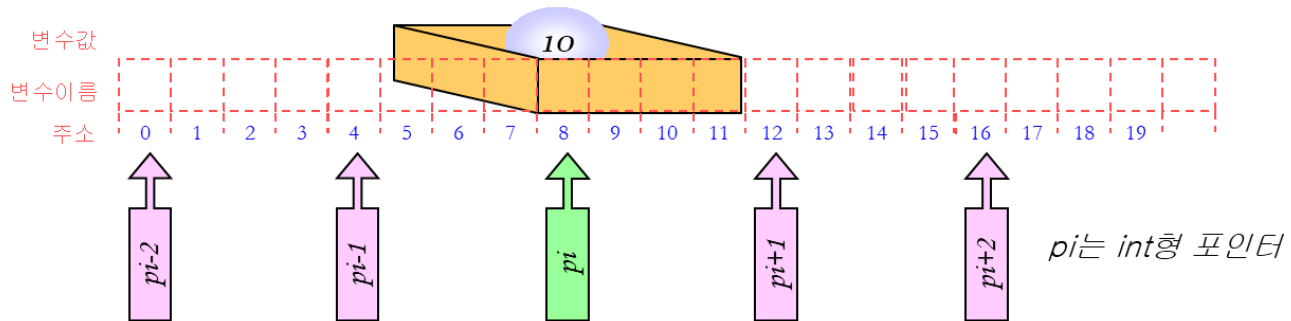
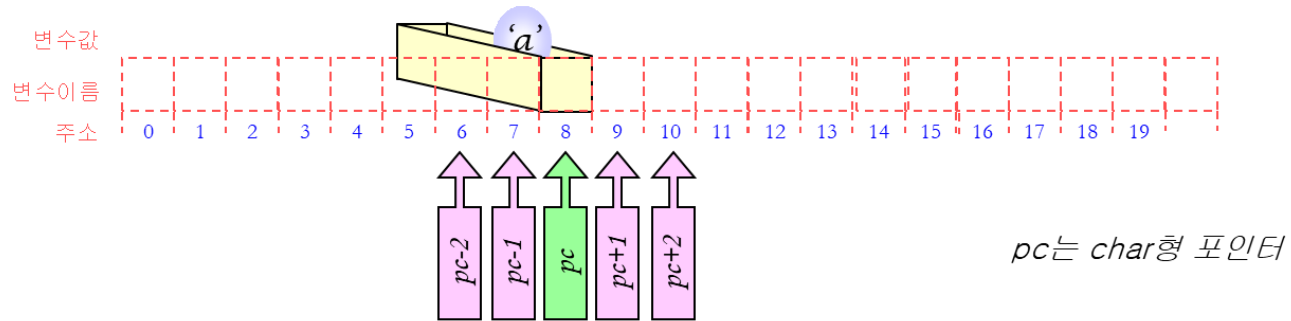
int main(void)
{
    char *pc;
    int *pi;
    double *pd;

    pc = (char *)10000;
    pi = (int *)10000;
    pd = (double *)10000;
    printf(" pc=%u, pc+1=%u, pc+2= %u\n", pc, pc + 1, pc + 2);
    printf(" pi=%u, pi+1=%u, pi+2= %u\n", pi, pi + 1, pi + 2);
    printf(" pd=%u, pd+1=%u, pd+2= %u\n", pd, pd + 1, pd + 2);

    return 0;
}
```

```
pc=10000, pc+1=10001, pc+2= 10002
pi=10000, pi+1=10004, pi+2= 10008
pd=10000, pd+1=10008, pd+2= 10016
```

포인터의 증감 연산



간접 참조 연산자와 증감 연산자

- `*p++`;
 - `p`가 가리키는 위치에서 값을 가져온 후에 `p`를 증가한다.
- `(*p)++`;
 - `p`가 가리키는 위치의 값을 증가한다.

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.

간접 참조 연산자와 증감 연산자

```
// 포인터의 증감 연산
```

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

```
int main(void)
```

```
{
```

```
    int i = 10;
```

```
    int *pi = &i;
```

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

```
    (*pi)++;
```

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

```
    *pi++;
```

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

```
    return 0;
```

```
}
```

pi가 가리키는 위치의 값을 증가한다.

pi가 가리키는 위치에서 값을 가져온 후에 pi를 증가한다.

i = 10, pi = 000000FFEBBCFF974

i = 11, pi = 000000FFEBBCFF974

i = 11, pi = 000000FFEBBCFF978

포인터의 형 변환

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

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

```
int* pi;
```

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

예제

```
#include <stdio.h>

int main(void)
{
    int data = 0x0A0B0C0D;
    char *pc;
    int i;

    pc = (char *)&data;
    for (i = 0; i < 4; i++)
        printf("*(pc + %d) = %02X \n", i, *(pc + i));
    return 0;
}
```

```
*(pc + 0) = 0D
*(pc + 1) = 0C
*(pc + 2) = 0B
*(pc + 3) = 0A
```

참고

- 포인터의 증감 연산에서 포인터의 위험성을 조금은 느낄 수 있다. 포인터는 우리가 마음대로 증감시킬 수 있지만 증감된 포인터가 잘못된 위치를 가리킬 수도 있다.
- 우리가 만든 데이터가 아닌 남의 데이터를 가리킬 수도 있고 운영체제가 사용하는 데이터 영역을 가리킬 수도 있다.
- 이런 경우, 포인터를 이용하여 값을 쓰거나 읽게 되면 심각한 오류가 발생할 수 있다.

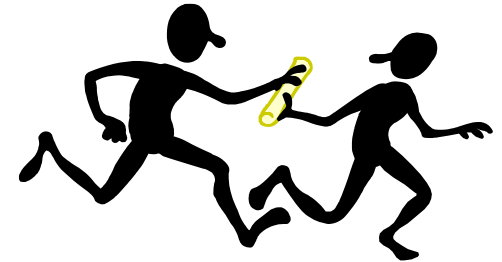
중간 점검

1. 포인터에 대하여 적용할 수 있는 연산에는 어떤 것들이 있는가?
2. `int`형 포인터 `p`가 80번지를 가리키고 있었다면 `(p+1)`은 몇 번지를 가리키는가?
3. `p`가 포인터라고 하면 `*p++`와 `(*p)++`의 차이점은 무엇인가?
4. `p`가 포인터라고 하면 `*(p+3)`의 의미는 무엇인가?



인수 전달 방법

- 함수 호출 시에 인수 전달 방법
 - 값에 의한 호출(call by value)
 - 함수로 복사본이 전달된다.
 - C언어에서의 기본적인 방법
 - 참조에 의한 호출(call by reference)
 - 함수로 원본이 전달된다.
 - C에서는 포인터를 이용하여 흉내 낼 수 있다.



swap() 함수 #1(값에 의한 호출)

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


    swap(a, b);

    printf("a=%d b=%d\n", a, b);
    return 0;
}
```

```
void swap(int x, int y)
{
    int tmp;
    printf("x=%d y=%d\n", x, y);

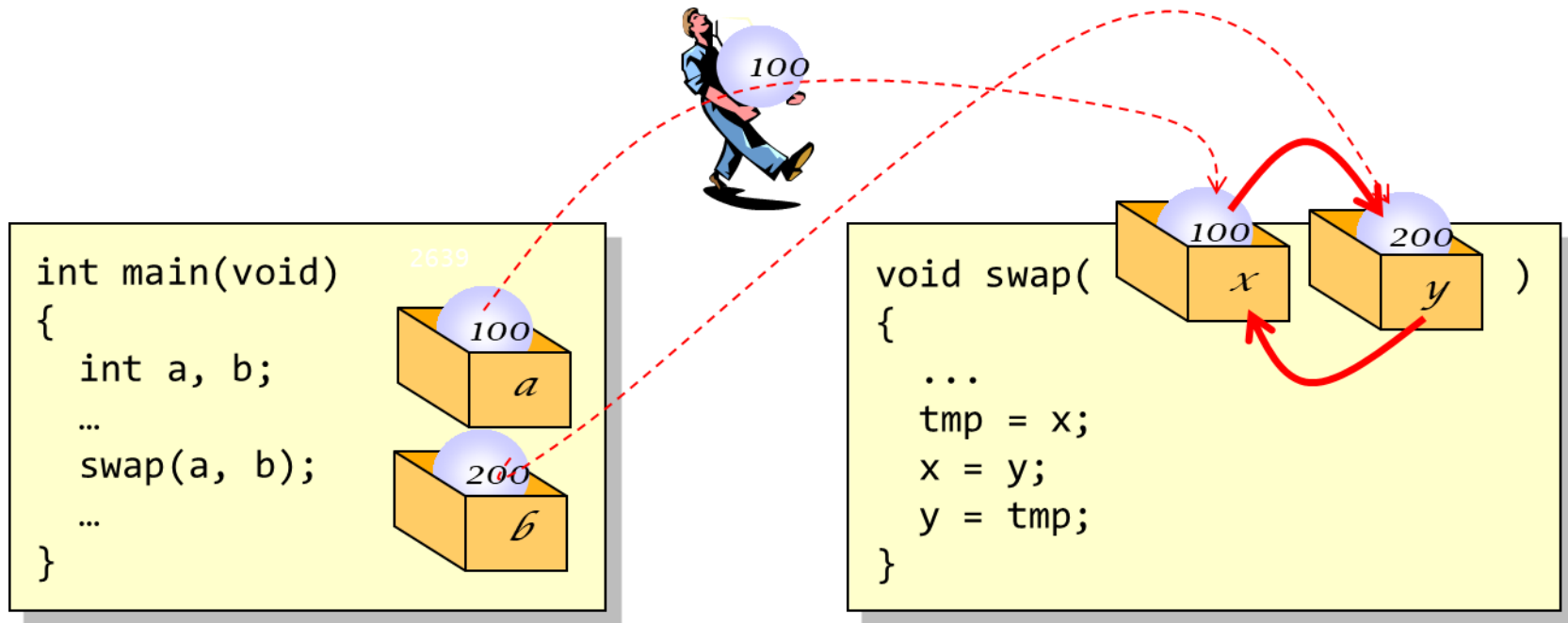
    tmp = x;
    x = y;
    y = tmp;

    printf("x=%d y=%d\n", x, y);
}
```



```
a=100 b=200
x=100 y=200
x=200 y=100
a=100 b=200
```

값에 의한 호출



swap() 함수 #2(참조에 의한 호출)

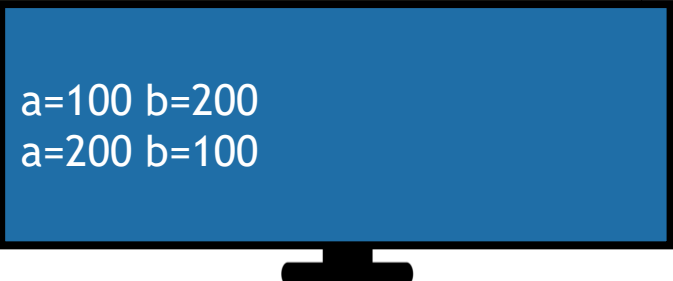
```
#include <stdio.h>
void swap(int x, int y);
int main(void)
{
    int a = 100, b = 200;
    printf("a=%d b=%d\n", a, b);

    swap(&a, &b);

    printf("a=%d b=%d\n", a, b);
    return 0;
}
```

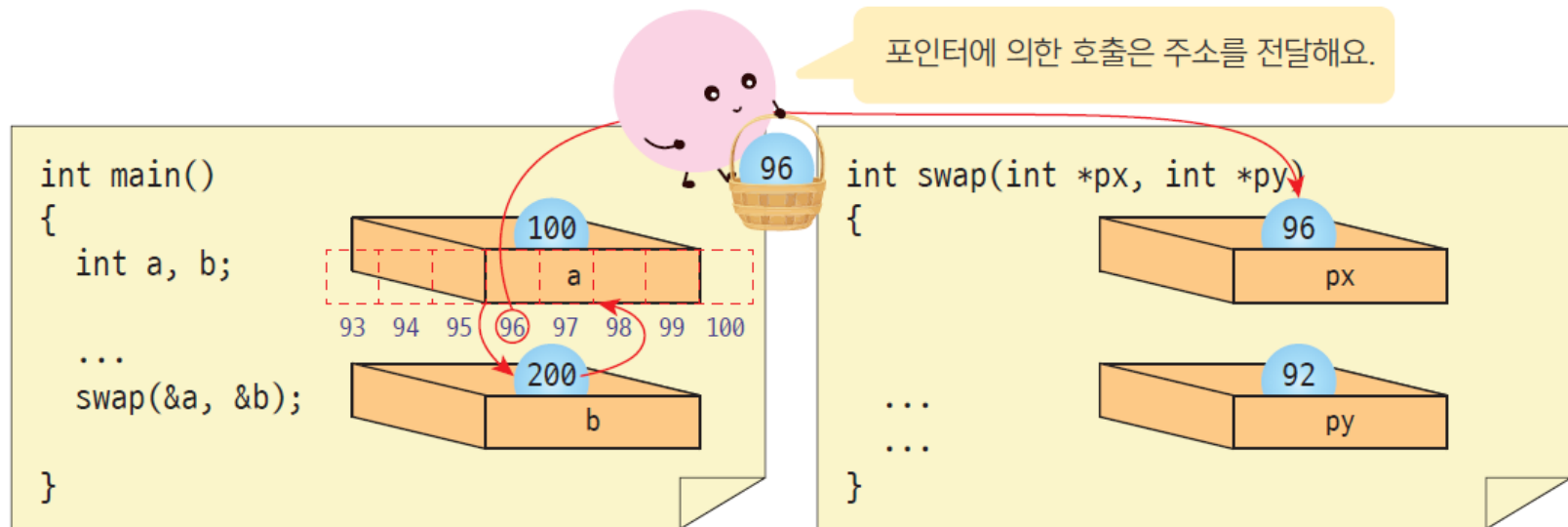
```
void swap(int *px, int *py)
{
    int tmp;

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



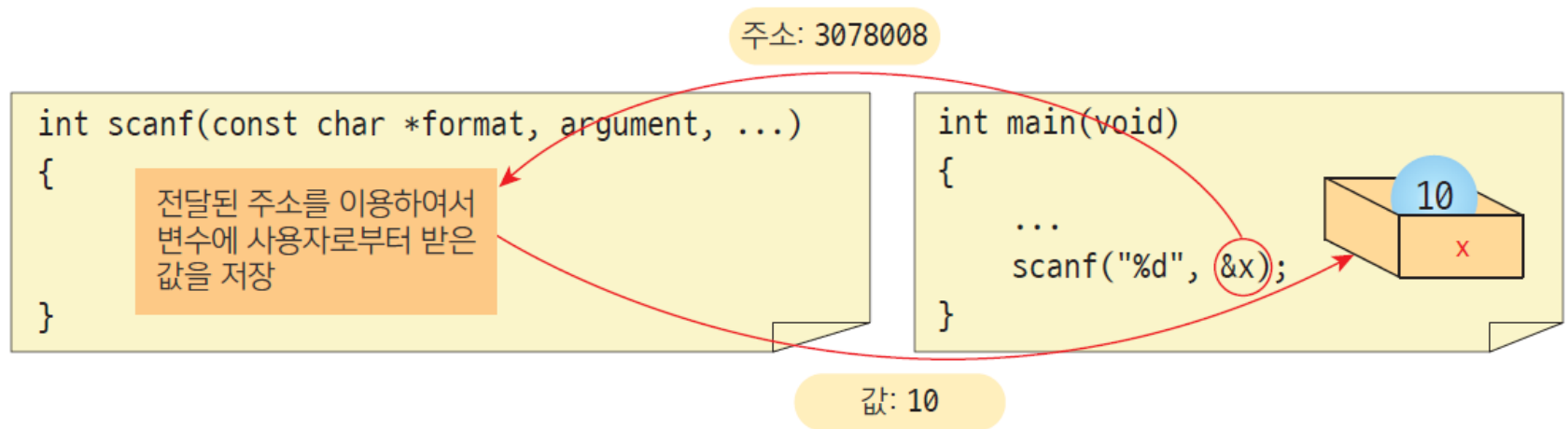
a=100 b=200
a=200 b=100

참조에 의한 호출



scanf() 함수

- 변수에 값을 저장하기 위하여 변수의 주소를 받는다.



참고:함수가 포인터를 통하여 값을 변경할 수 없게 하려면?

- 함수의 매개 변수를 선언할 때 앞에 const를 붙이면 된다. const를 앞에 붙이면 포인터가 가리키는 내용이 변경 불가능한 상수라는 뜻이 된다.

```
void sub(const int *p)
{
    *p = 0;    // 오류!!
}
```


예제

- 만약 함수가 하나 이상의 값을 반환하여야 한다면 포인터를 사용하는 것이 하나의 방법이다. 직선의 기울기와 y절편의 값을 동시에 반환하는 함수를 작성해보자.

A blue rectangular box with a black border, representing a computer monitor. Inside the box, the text "기울기는 1.000000, y절편은 0.000000" is written in white. Below the box is a small black horizontal bar representing the monitor's base.

기울기는 1.000000, y절편은 0.000000

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

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;

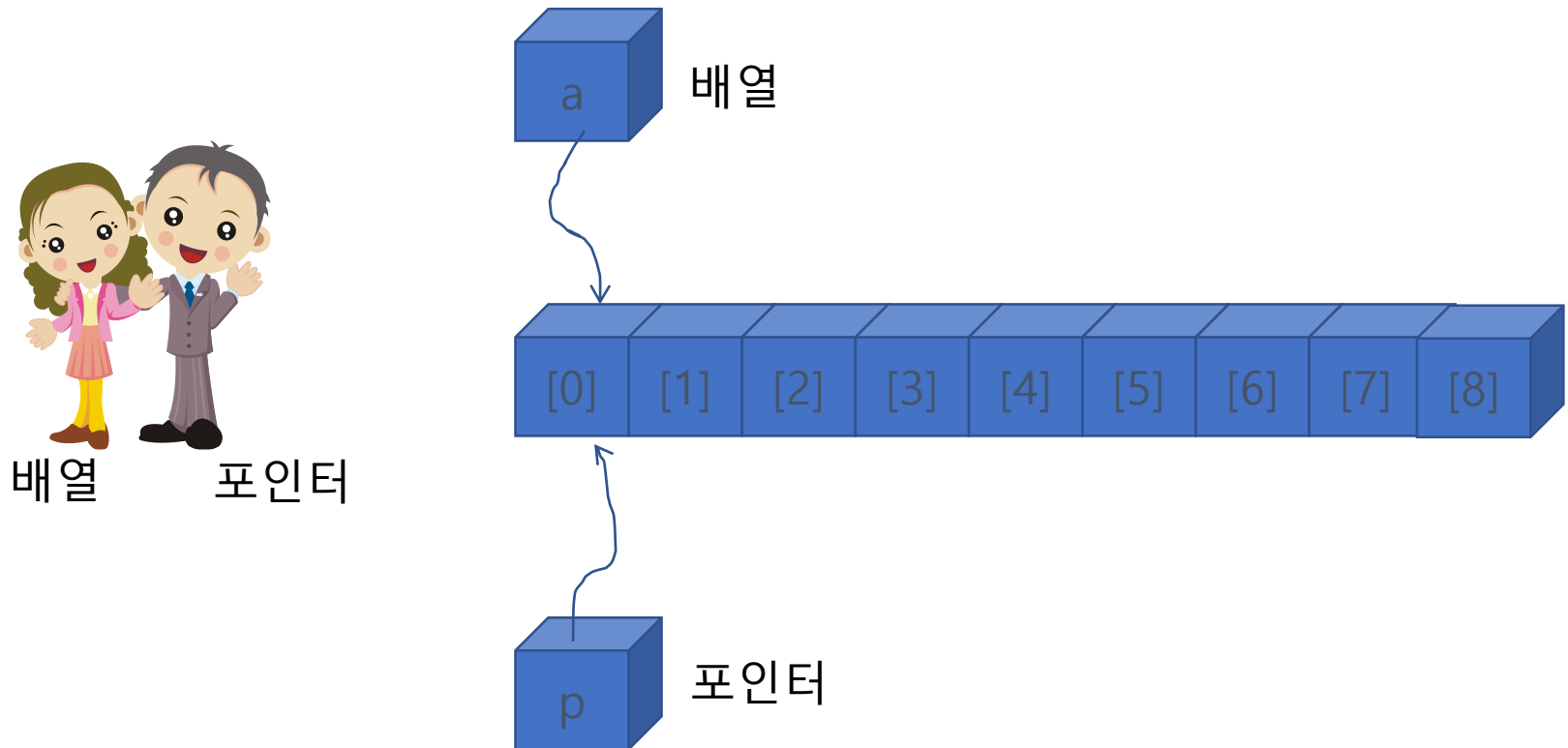
    result = x + y;
    return &result;
}
```

지역변수는 함수 호출이 종료되면 사라지므로 지역 변수의 주소를 반환하면 안됩니다.



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]);
```

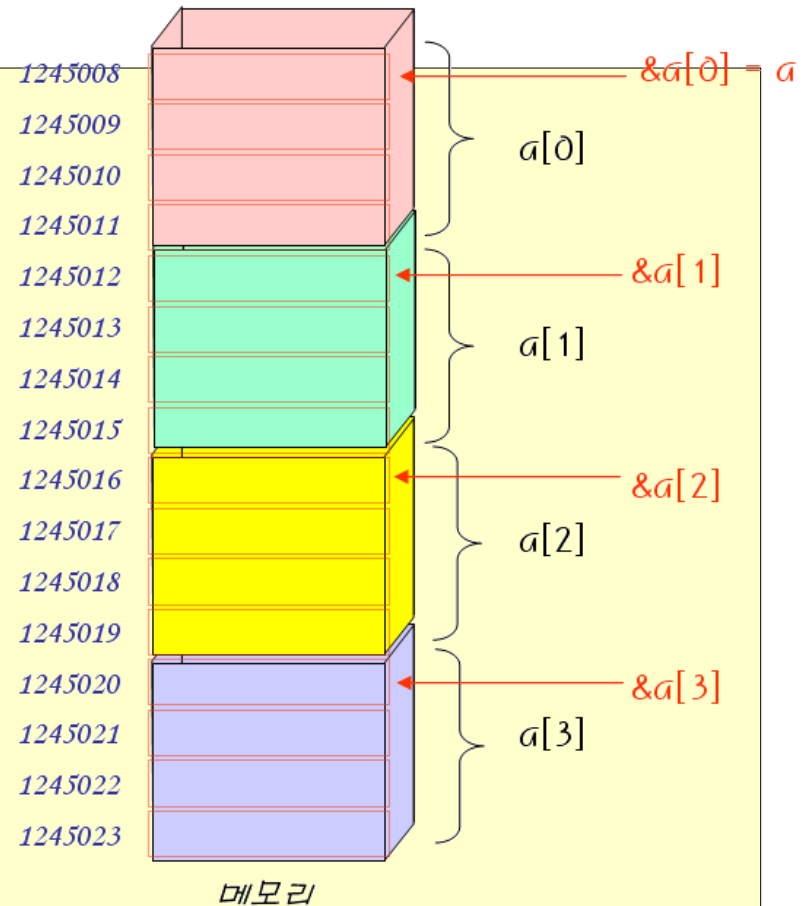
```
    printf("&a[1] = %u\n", &a[1]);
```

```
    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
```

예제

// 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

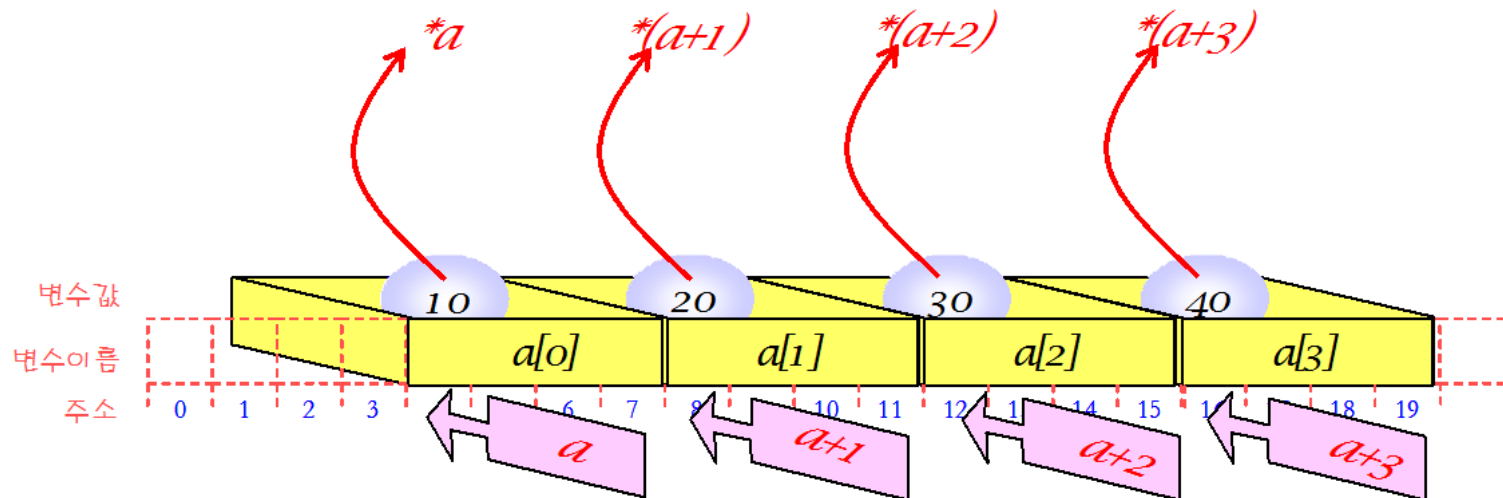
a + 1 = 1245012

*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;
}
```

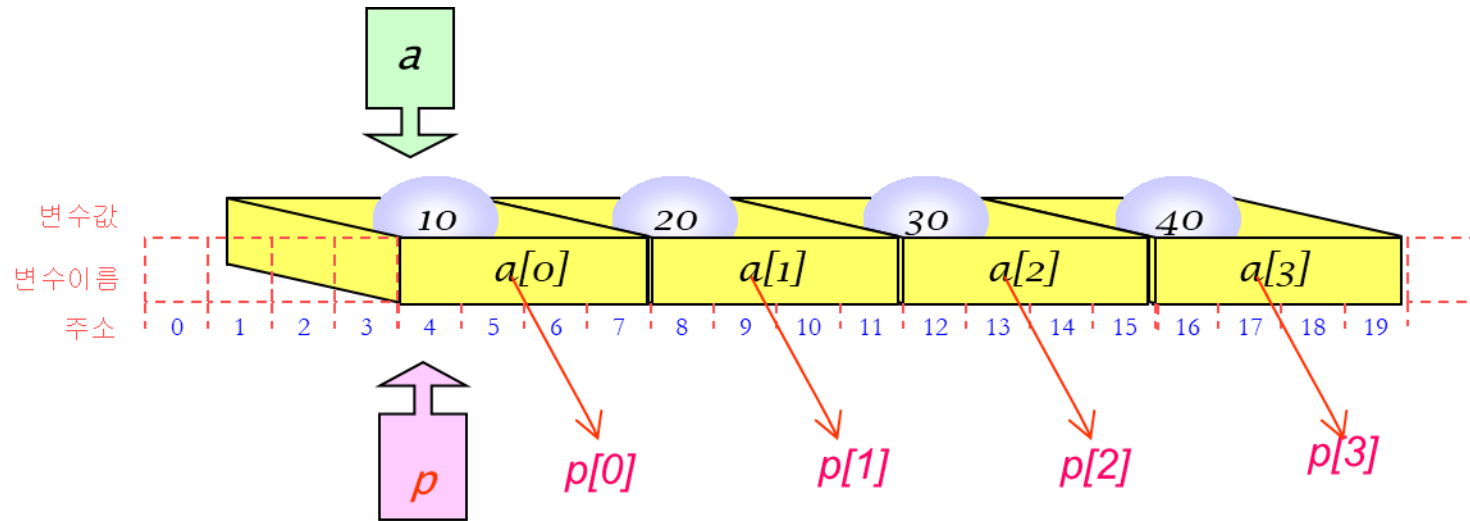
배열은 결국 포인터로
구현된다는 것을 알 수
있다.

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 배열의 이름처럼 사용할 수도 있다.



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 소모되므로 배열은 주소만을 전달한다.

배열 매개 변수

- 배열 매개 변수는 포인터로 생각할 수 있다.

```
int main(void)
{
    int a[3]={ 1, 2, 3 };

    sub(a, 3);
}
```

배열의 이름은 포인터이다.

```
void sub(int b[], int size)
{
    b[0] = 4;
    b[1] = 5;
    b[2] = 6;
}
```

b를 통하여 원본 배열을 변경할 수 있다.

// 포인터와 함수의 관계

#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;

}

1 2 3
4 5 6

다음 2가지 방법은 완전히 동일하다.

// 포인터 매개 변수

```
void sub(int *b, int size)
```

```
{
```

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

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

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

```
}
```

배열의 이름과 포인터
는 근본적으로 같다.

배열 표기법을 사용
하여 요소에 접근

// 포인터 매개 변수

```
void sub(int *b, int size)
```

```
{
```

```
    *b = 4;
```

```
    *(b+1) = 5;
```

```
    *(b+2) = 6;
```

```
}
```

포인터 표기법을 사
용하여 요소에 접근

포인터를 사용한 방법의 장점

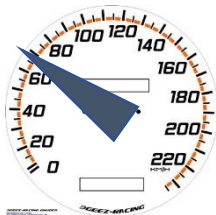
컴파일러가 최적화를 하면 성능은 거의 비슷해진다.

- 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;

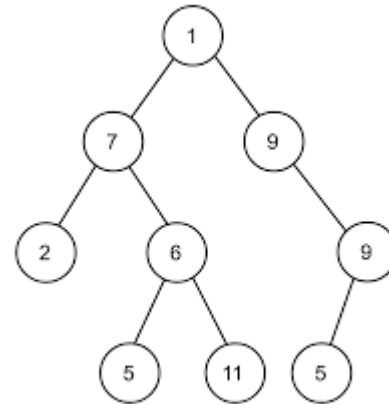
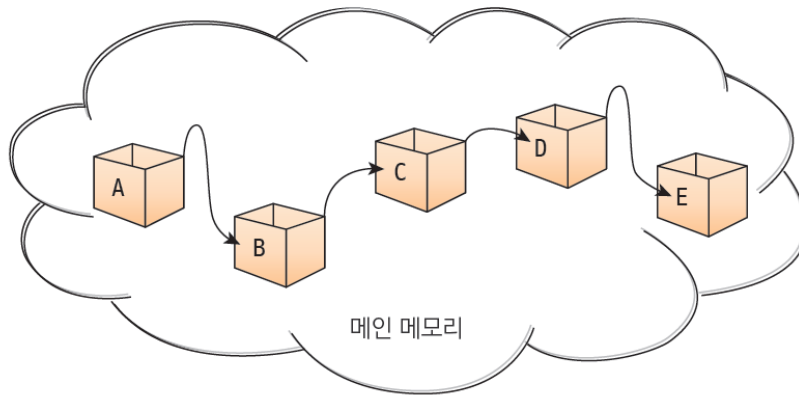
    p = a;
    for(i = 0; i < n; i++)
        sum += *p++;
    return sum;
}
```

포인터 사용



포인터 사용의 장점

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



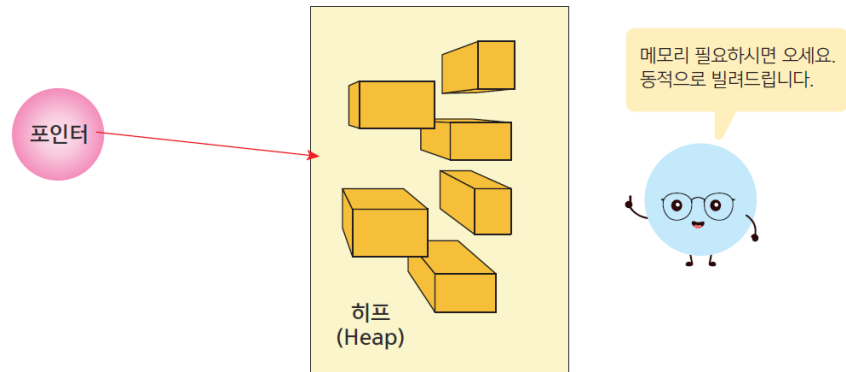
- 참조에 의한 호출
 - 포인터를 매개 변수로 이용하여 함수 외부의 변수의 값을 변경할 수 있다.

포인터 사용의 장점

- 메모리 매핑 하드웨어
 - 메모리 매핑 하드웨어란 메모리처럼 접근할 수 있는 하드웨어 장치를 의미한다.

```
volatile int *hw_address = (volatile int *)0x7FFF;  
*hw_address = 0x0001; // 주소 0x7FFF에 있는 장치에 0x0001 값을 쓴다.
```

- 동적 메모리 할당
 - 17장에서 다룬다.
 - 동적 메모리를 사용하려면 반드시 포인터가 있어야 한다.



Q & A

