Pointers

adopted from KNK C Programming : A Modern Approach

요약

순서

- Binary to Byte
- 주소
- 변수의 주소 확인
- Scanf에서 & 연산자의 활용
- 포인터의 2개의 연산자
- 포인터 변수와 사용 예
- 활동: 인간 포인터 연습
- void 타입과 변수 크기와의 관계
- 포인터에서의 캐스팅
- 포인터와 배열

Binary to Byte

- 1203의 2진수 표현

$$1203 = 1024 + 128 + 32 + 16 + 2 + 1$$

$$= 2^{10} + 2^{7} + 2^{5} + 2^{4} + 2^{1} + 2^{0}$$

2진수 => 0100 1011 0011

16진수 => 4 B 3

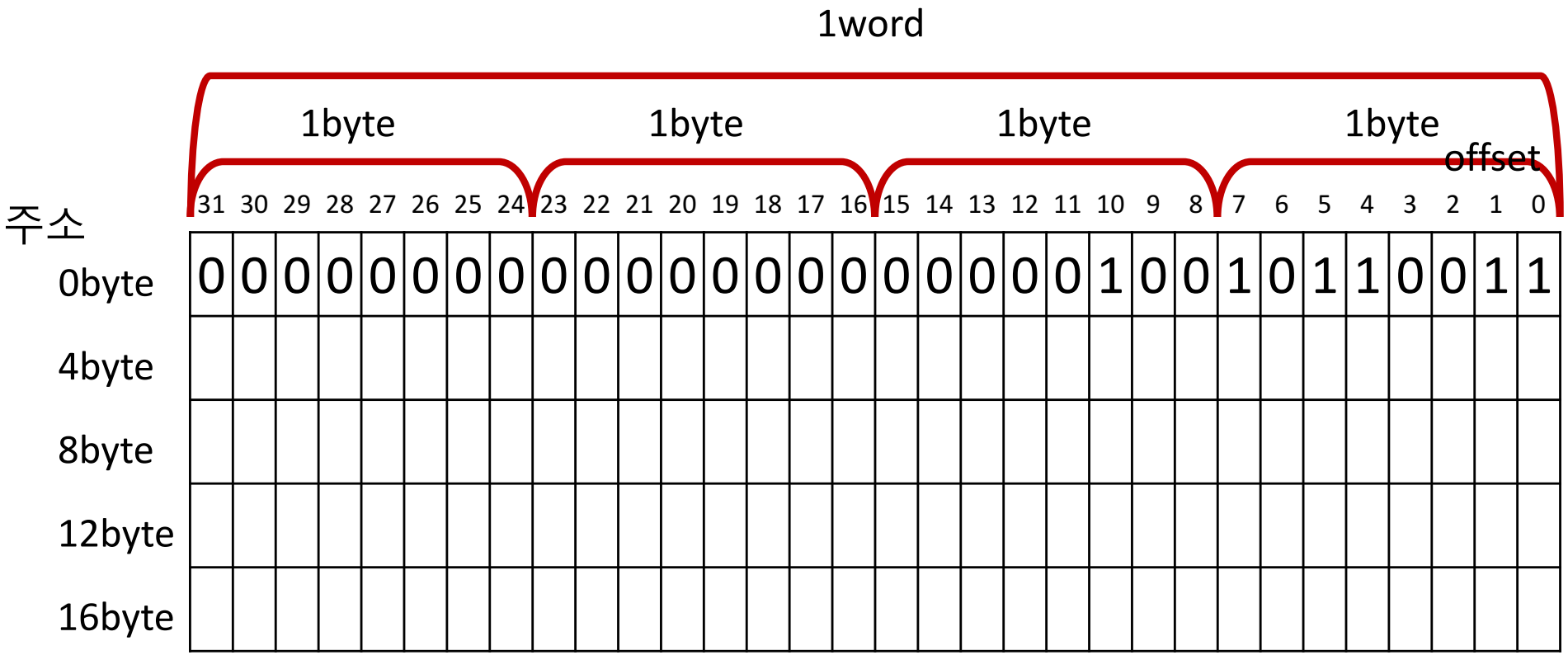
Binary(2진수)는 각 자리 또는 bit에 0과 1만 표현 가능
1203을 2진수로 표현하기 위해 11bit가 필요함

8 bit = 1 byte

위치	10	9	8	7	6	5	4	3	2	1	0
값	1	0	0	1	0	1	1	0	0	1	1

주소

- 32bit 주소 체계를 따르는 컴퓨터의 경우 정보의 표현
  - 64bit인 경우 한 줄에 64개 2진수 표현 가능

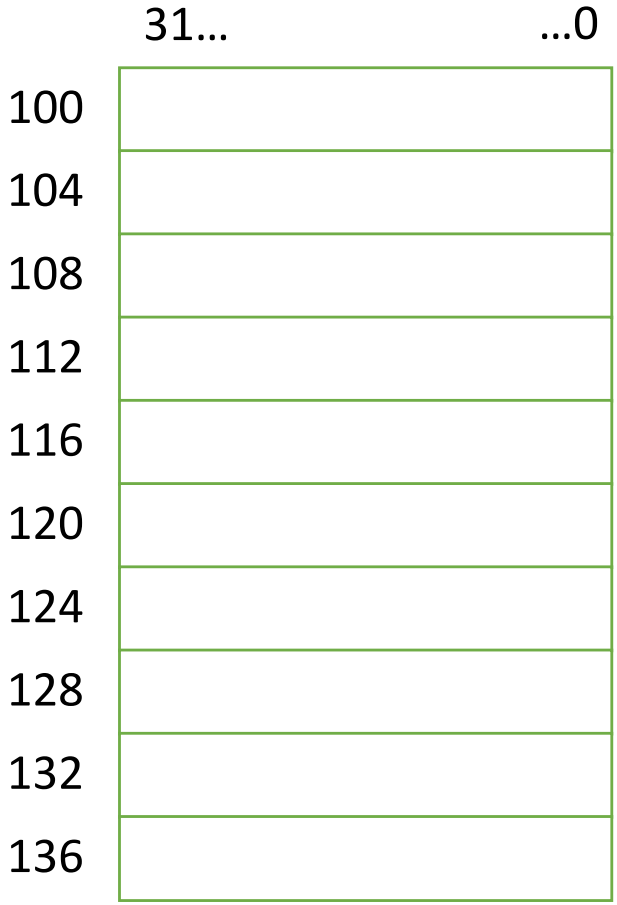


주소

- 모든 객체 (변수, 함수 등)은 고유의 주소를 갖음

예시

```
int a = 100;
int b = 200;
int c = 300;
```

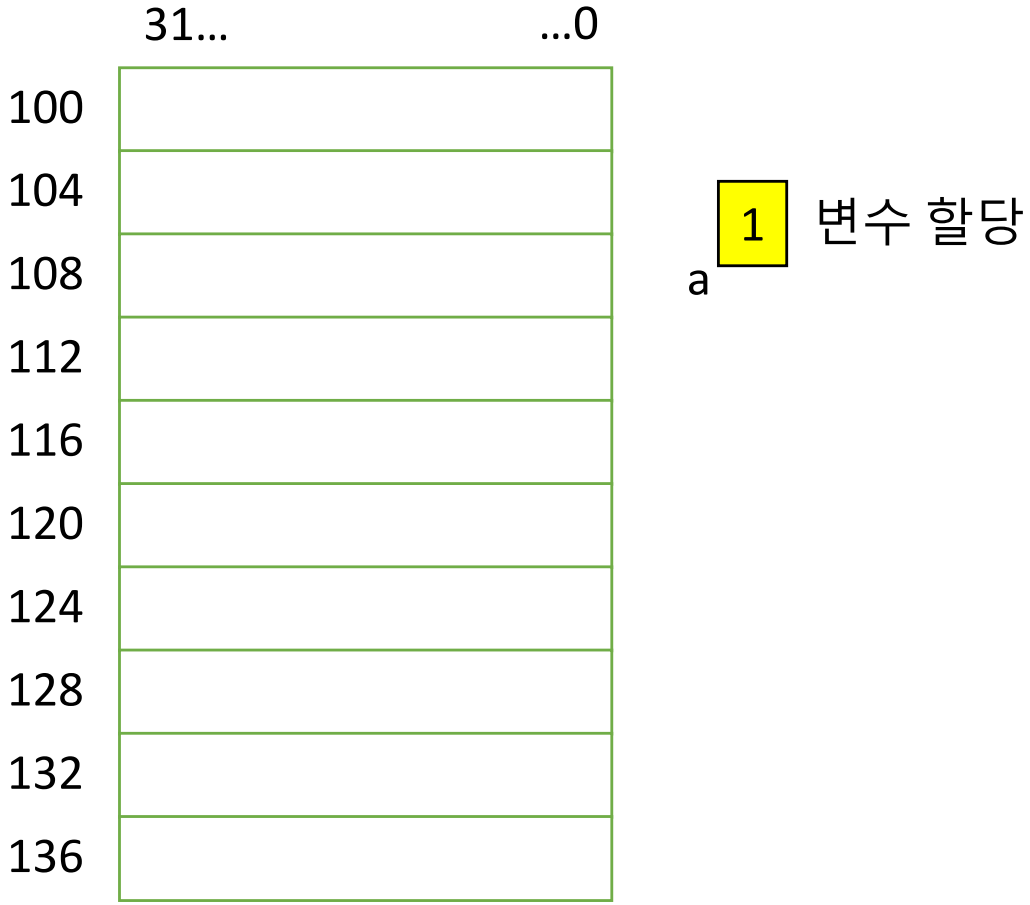


주소

- 모든 객체 (변수, 함수 등)은 고유의 주소를 갖음

예시

```
int a = 100;
int b = 200;
int c = 300;
```

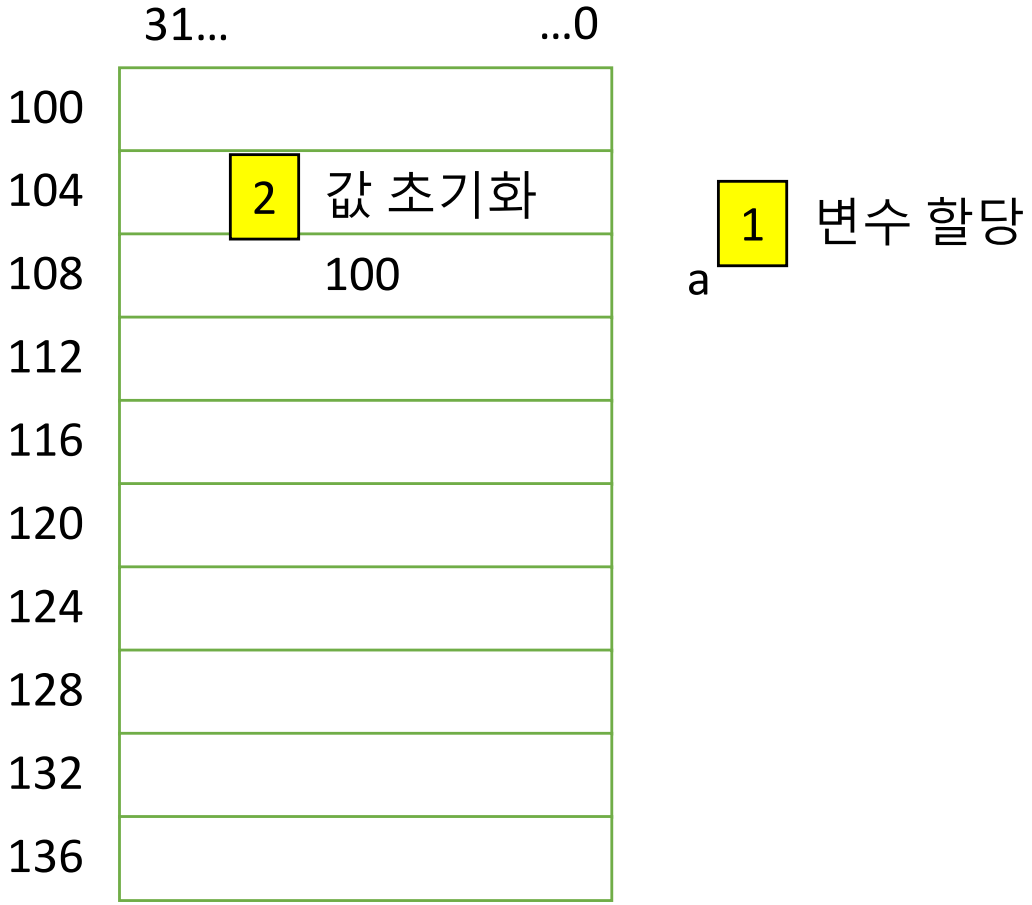


주소

- 모든 객체 (변수, 함수 등)은 고유의 주소를 갖음

예시

```
int a = 100;
int b = 200;
int c = 300;
```

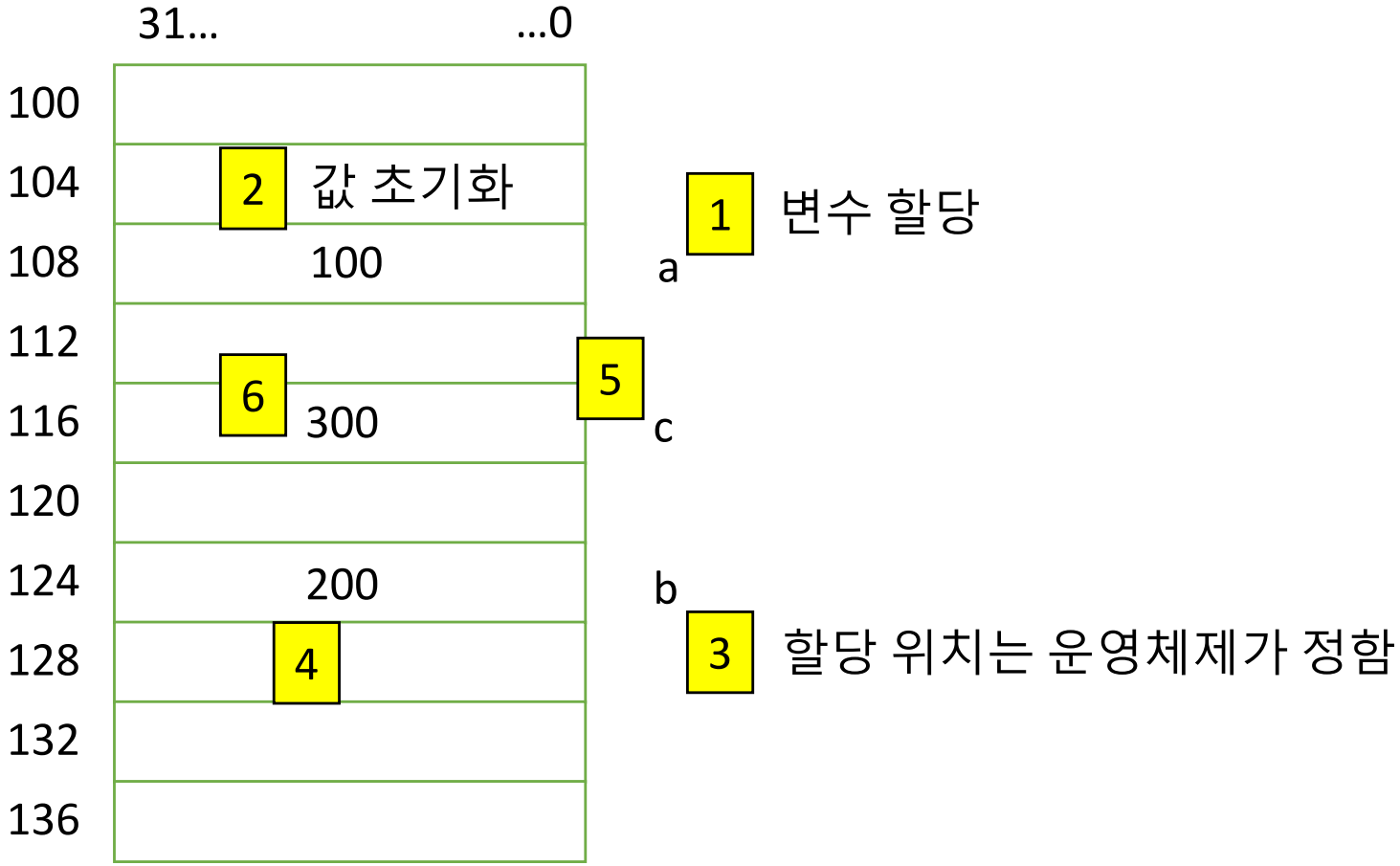


주소

- 모든 객체 (변수, 함수 등)은 고유의 주소를 갖음

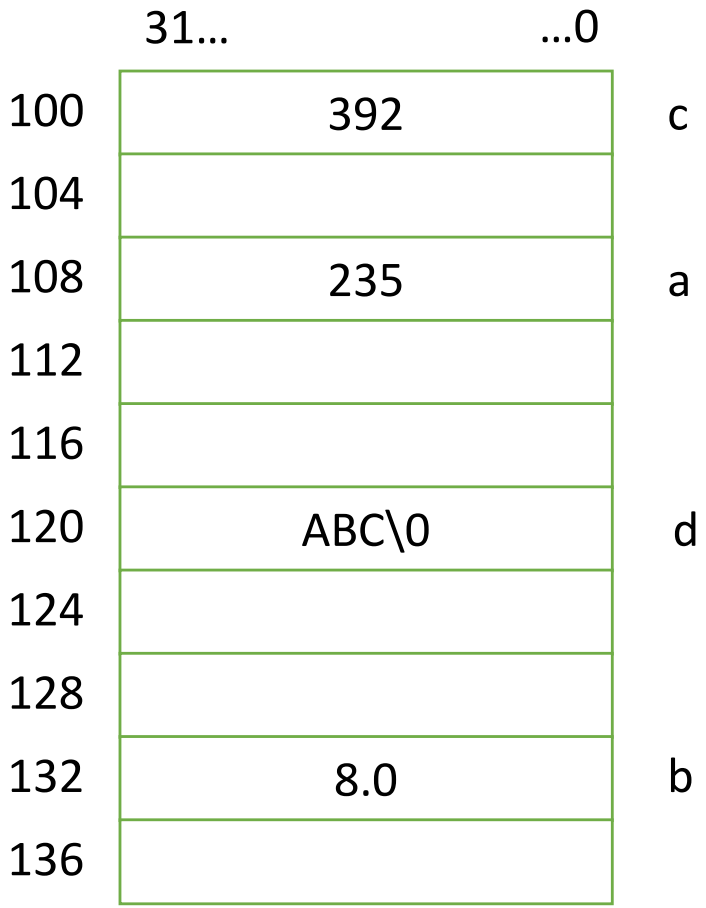
예시

```
int a = 100;
int b = 200;
int c = 300;
```



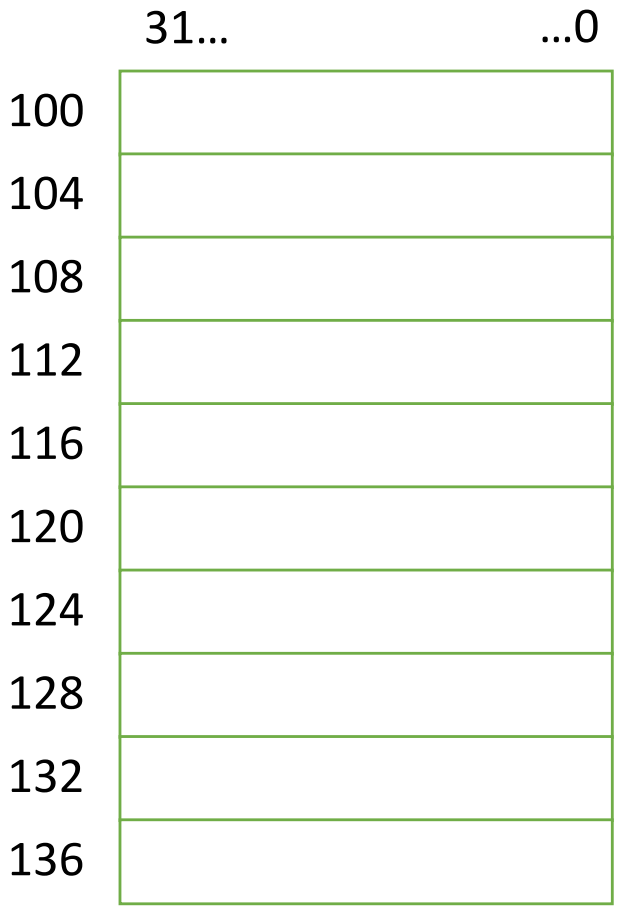
문제

- 다음의 메모리 상태를 참고로, 변수 초기화 문장을 작성



문제

- 다음의 메모리 초기화 문장을 토대로 메모리 배치도 작성




```
int a = 1;
float b = 39.2;
double c = 400201;
char d[5] = { 1, 2, 3, 4, 5};
```

변수의 주소 확인

```
#include <stdio.h>
int main(){
    int c[5] = { 3, 2, [3] = 8, [4] = 9};
    for (int k = 0; k < 5; k++)
        printf("value of c[%d] = %d; address is %p\n", \
                k, c[k], &c[k]);
    return 0;
}
```

16진수 주소 형식

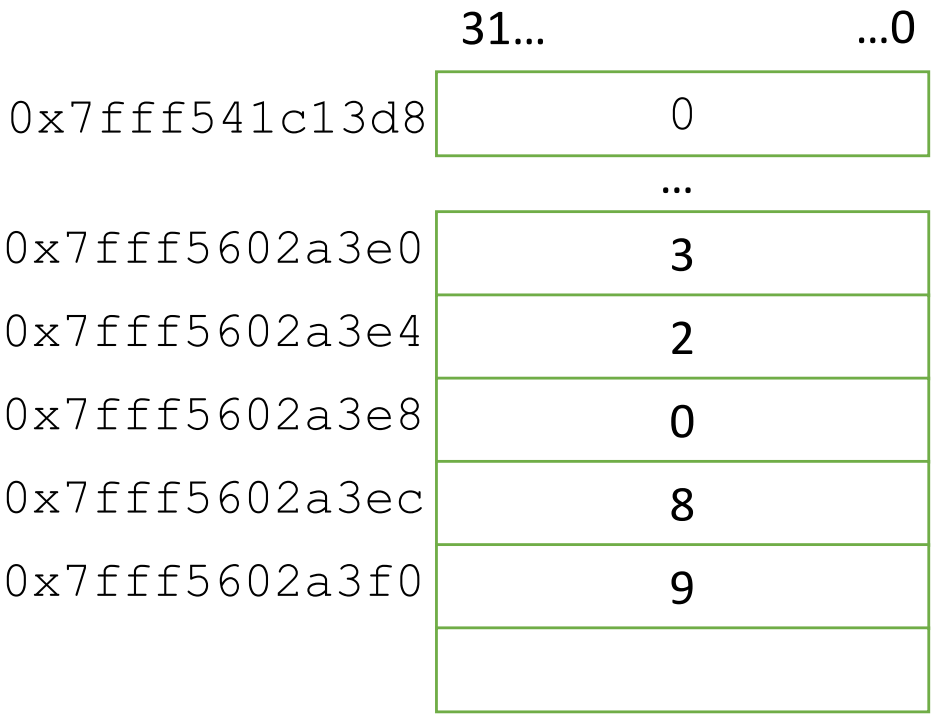
주소 연산자: 변수의 저장된 주소

1. 변수 c 할당
2. 변수 c 초기화
3. 변수 k 할당
4. 변수 k 초기화
5. c[k]의 값과 주소 출력
6. 변수 k의 값 증가

변수의 주소 확인

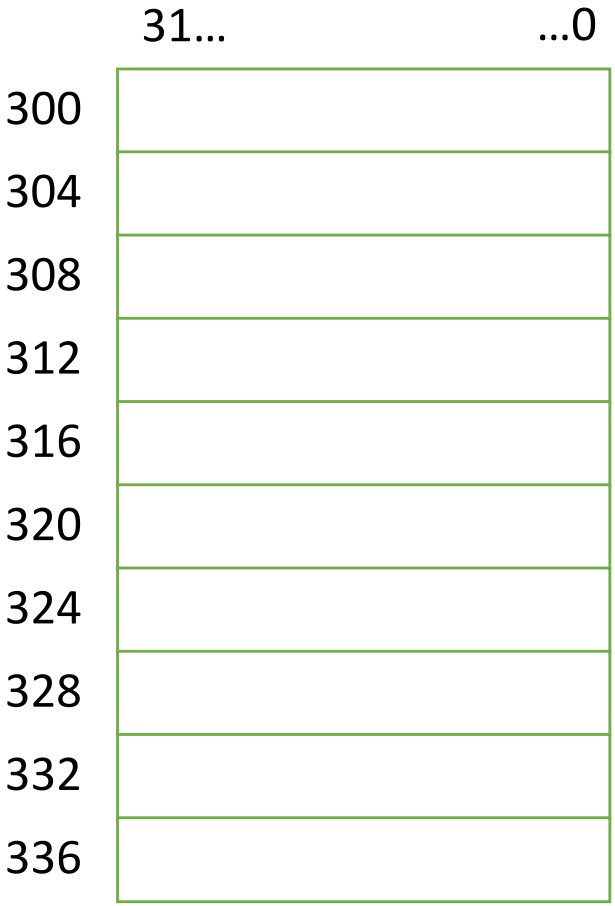
실행 결과

```
value of c[0] = 3; address is 0x7fff5602a3e0
value of c[1] = 2; address is 0x7fff5602a3e4
value of c[2] = 0; address is 0x7fff5602a3e8
value of c[3] = 8; address is 0x7fff5602a3ec
value of c[4] = 9; address is 0x7fff5602a3f0
```



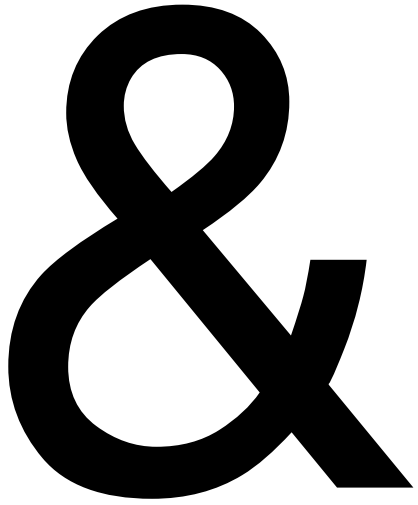
Scanf에서 &연산자의 활용

```
1   float a; // 4byte
2   scanf(“%f”, &a);
```



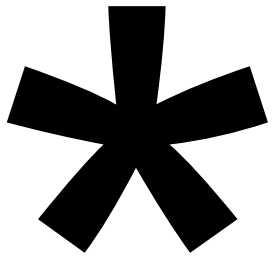
포인터의 2개의 연산자

포인터 변수
저장하는 것은 주소



참조 연산자(reference operator)

"~의 주소"로 이해



역참조 연산자(dereference operator)

"~가 가리키는 값"
으로 이해

*Dereference는 역참조 또는 간접참조라 부름

포인터 변수

- 선언 – 역참조 연산자를 사용하여 주소를 저장하는 변수임을 표시

```
타입      역참조 연산자   포인터 변수명
double *pointer_variable;
```

- 할당 – 참조 연산자를 사용하여 주소를 포인터 변수에 저장

```
포인터 변수                참조 연산자  변수명
pointer_variable = &variable;
```

- 읽기 – 역참조 연산자를 사용하여 저장된 주소가 가리키는 곳의 값을 가져옴

```
역참조 연산자   포인터 변수명
*pointer_variable;
```

포인터 변수의 사용 예

- 다른 변수의 주소를 포인터 변수에 저장하는 방법

- Type I – 포인터 변수 선언시 주소 할당

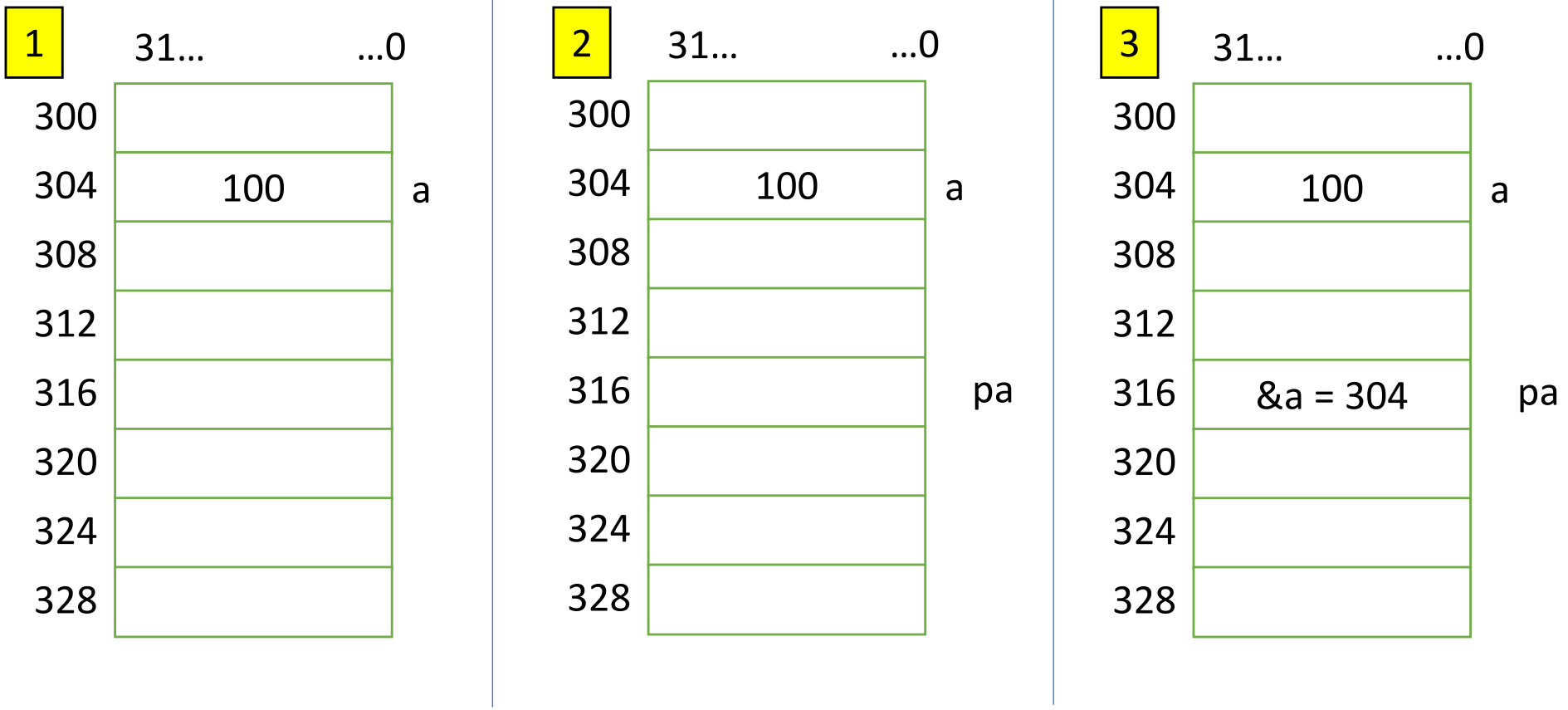
```
1   int a = 100;    // a 의 주소는 256이라 가정
2   int *pa = &a;   // 포인터 변수 선언 및 주소 할당
```

- Type II – 일반적인 주소 할당 방법

```
1   int a = 100;    // a 의 주소는 256이라 가정
2   int *pa;        // 포인터 변수 선언
3   …               // 다른 문장들 실행
4   pa = &a;        // 포인터 변수에 a의 주소 저장
```

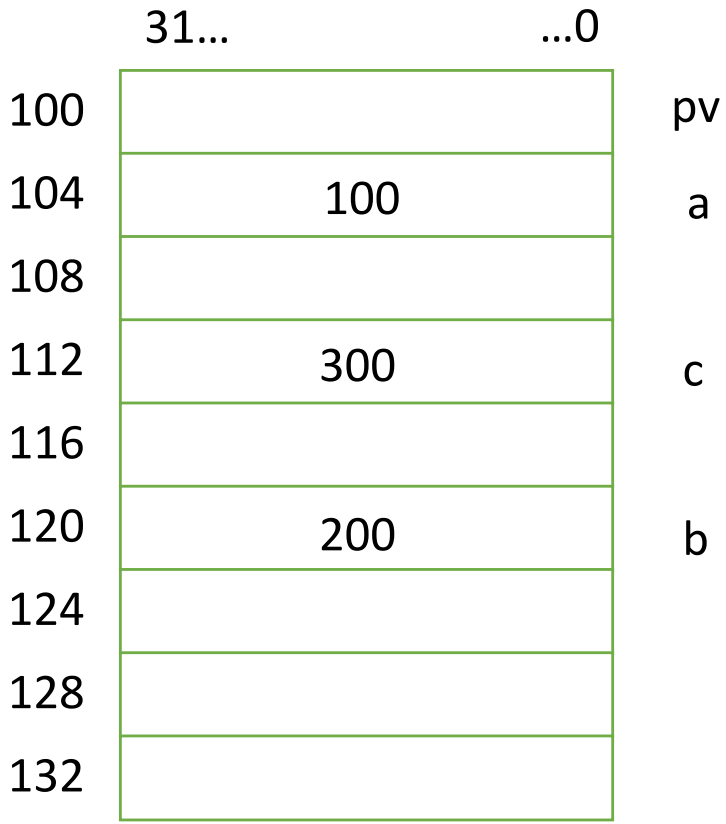
포인터 변수의 사용 예

```
1   int a = 100;      // a 의 주소는 256이라 가정
2   int *pa;          // 포인터 변수 선언
3   …                 // 다른 문장들 실행
4   pa = &a;          // 포인터 변수에 a의 주소 저장
```



예시

```
1   int a = 100, b = 200, c = 300;
2   int *pv;
3   printf("%p\n", pv = &a);
4   printf("%p\n", pv = &b);
5   printf("%p\n", pv = &c);
```



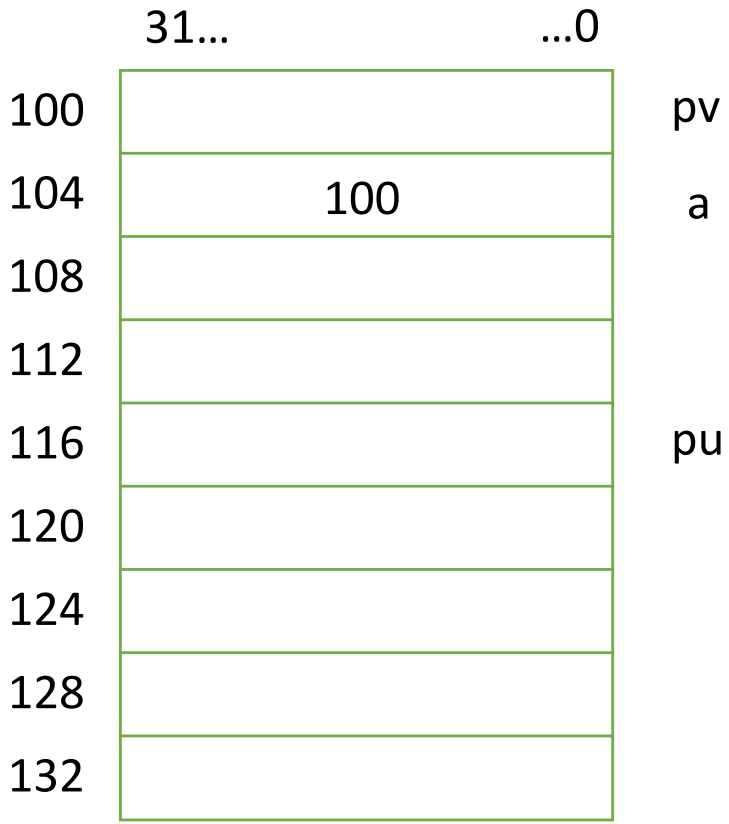
1 pv 값은 a의 주소 104

2 pv 값은 b의 주소 120

3 pv 값은 c의 주소 112

예시

```
1   int a;
2   int *pv = &a, *pu = &a);  // pv와 pv는 a의 주소 104를 저장
3   printf("%d\n", *pu = 400);
4   printf("%d, %d\n", a, *pv);
```



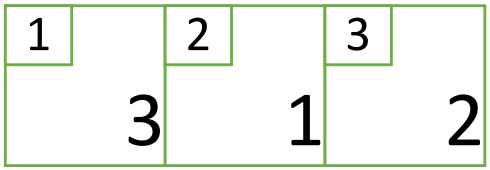
1 pu가 가리키는 a에 400 저장

2 pv가 가리키는 a의 값 400 읽음

3 pv 값은 c의 주소 112

활동: 인간 포인터 연습

- 전체 인원을 반으로 나누어 2 팀으로 나눔
- 제비뽑기를 통해 모든 사람은 두 개의 숫자를 갖음
  - 그 중 한 숫자는 주소 다른 한 숫자는 값임
  - 주소는 1에서 부터 40까지, 값도 1에서 부터 40까지임
  - 주소와 값이 동일한 수이면 왼쪽 사람의 값과 교환
- 1번 주소부터 40번 주소까지 모두 순회하는 데 걸리는 시간을 비교 함
- 1번 주소의 사람은 자신의 값을 활용하여 역참조(*연산)를 하여 값을 찾아 부름
- 역참조된 값을 갖고 있는 주소로 반복
  - 예시:



포인터 연습 문제

```
int a; // a의 주소는 1
int *p = &a; // p의 주소는 20
*p = 12;
```

문제:

1	2	3	4	5
12	3	4	11	14
6	7	8	9	10
20	17	15	10	19
11	12	13	14	15
18	9	8	16	13
16	17	18	19	20
7	2	6	5	1

void 타입과 변수 크기와의 관계

void는 타입 미지정

필요에 따라 void 타입 변수를
다른 타입으로 캐스팅(casting)하여 사용

- 캐스팅 방법 (새로운 타입) 변수명

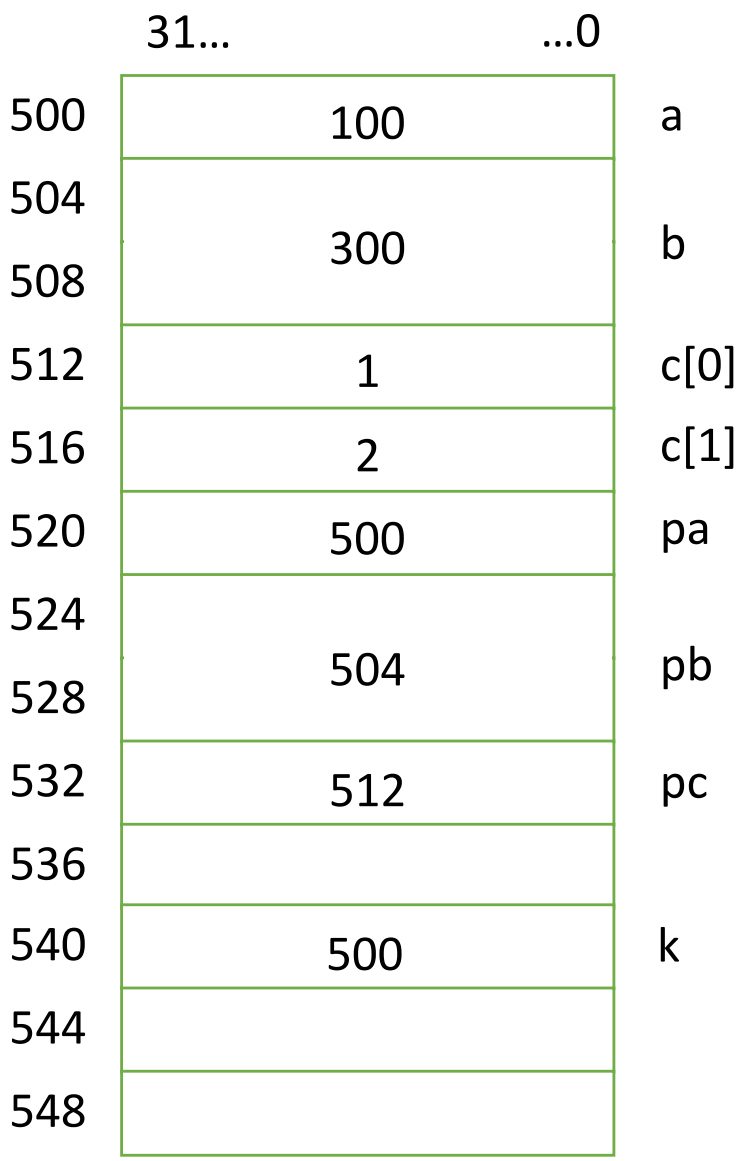
```
float a = 3.9, b = 7.2
int sum;
sum = (int)b % (int)a;
```

float형 변수 a와 b를 int형으로 캐스팅
int형 변수 sum에 결과를 저장

포인터에서의 캐스팅

```
int a = 100;
double b = 300;
int c[2] = {1, 2};

int *pa = &a;
double *pb = &b;
int *pc = &c[0];

void *k;

k = &a;
```

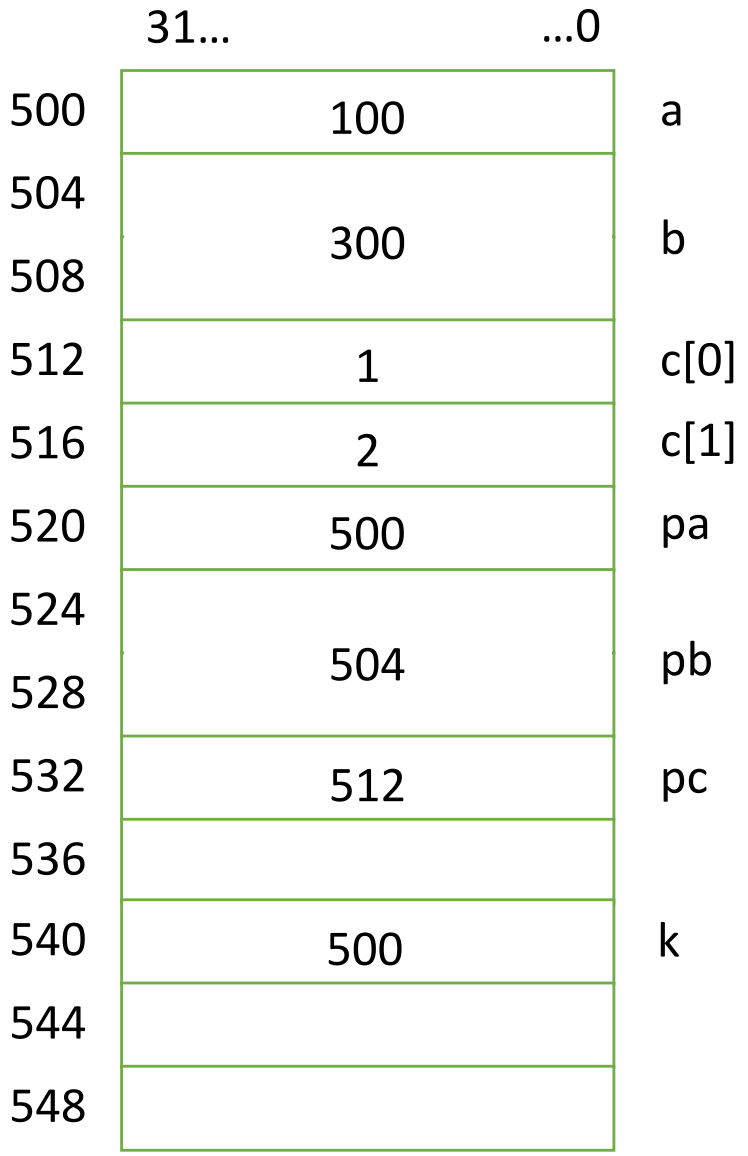
1 k 값은 a의 주소 500



포인터에서의 캐스팅

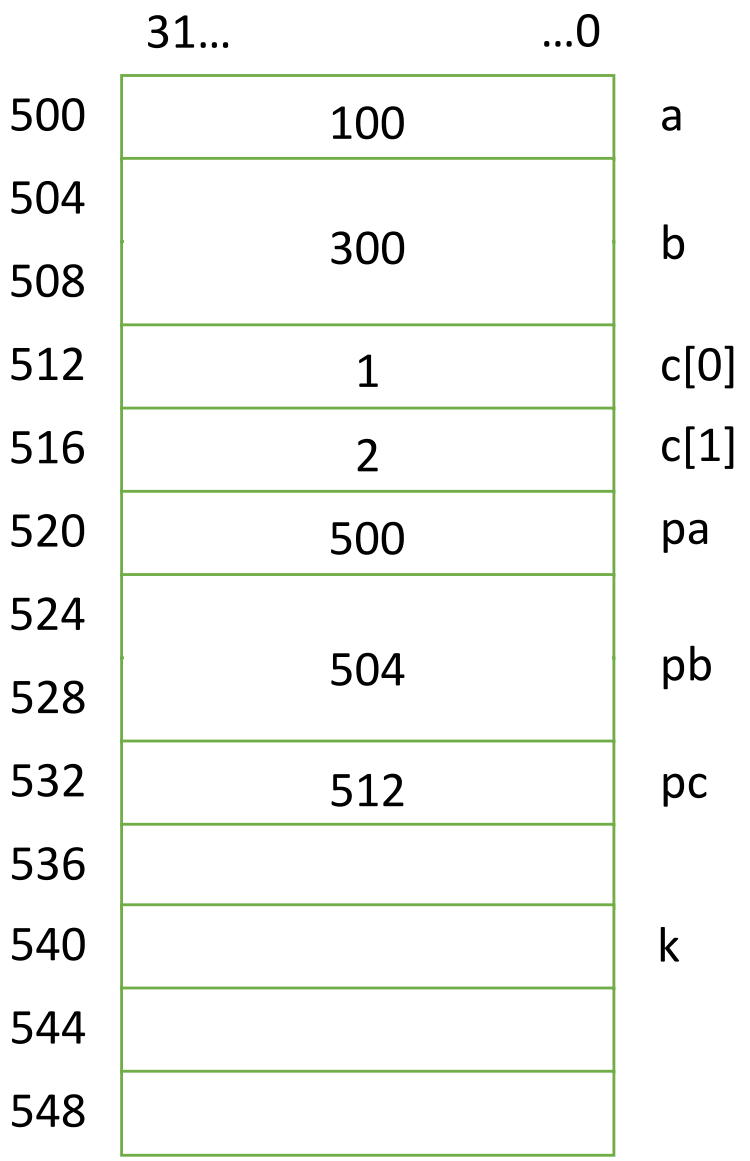
```
int a = 100;
double b = 300;
int c[2] = {1, 2};

int *pa = &a;
double *pb = &b;
int *pc = &c[0];

void *k;

k = &a;
printf("%d\n", *(int*)k);
```

1. k 값은 a의 주소 500
2. 읽을 바이트의 길이는 int 타입의 길이 4바이트



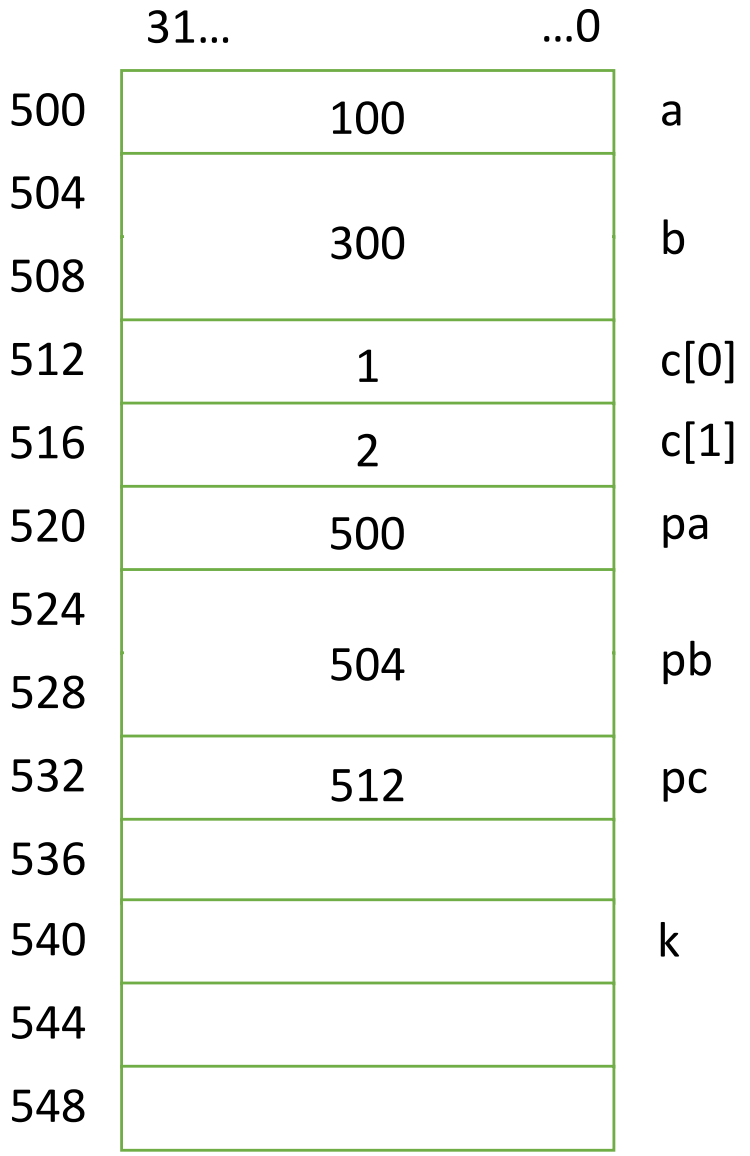
포인터에서의 캐스팅

```
int a = 100;
double b = 300;
int c[2] = {1, 2};

int *pa = &a;
double *pb = &b;
int *pc = &c[0];

void *k;

k = &a;
printf("%d\n", *(int*)k);

k = &b;
printf("%d\n", *(double*)k);
```



포인터에서의 캐스팅

```
int a = 100;
double b = 300;
int c[2] = {1, 2};

int *pa = &a;
double *pb = &b;
int *pc = &c[0];

void *k;

k = &a;
printf("%d\n", *(int*)k);

k = &b;
printf("%d\n", *(double*)k);

k = &c[0];
printf("%d\n", *(int*)k);
```



포인터와 배열

```
int c[4] = {1, 2, 3, 4};
int *pc;
```

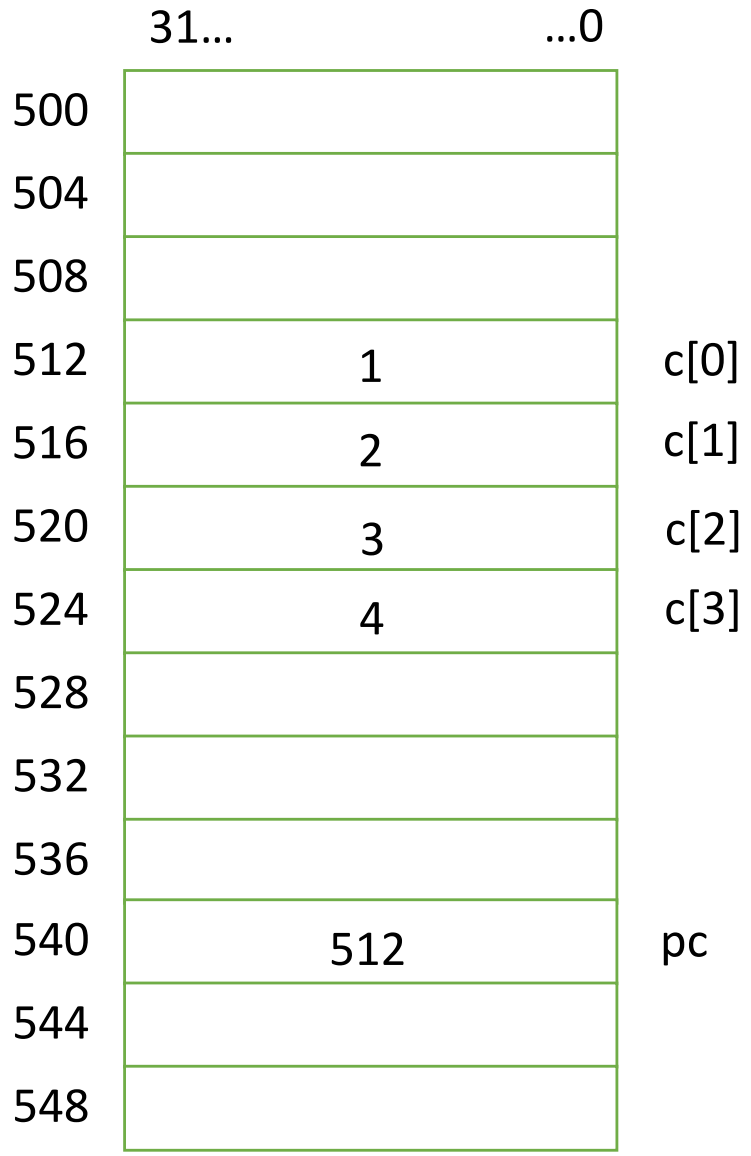
포인터를 이용한 배열 접근

```
pc = &c[0]; // c[0]의 주소
pc = &c[1]; // c[1]의 주소
pc = &c[2]; // c[2]의 주소
pc = &c[3]; // c[3]의 주소
```

포인터 연산을 이용한 배열 접근

```
pc; // c[0]의 주소
pc+1; // c[1]의 주소
pc+2; // c[2]의 주소
pc+3; // c[3]의 주소
```

시작 주소+(변수의 타입)*배열인덱스
c[2]의 주소 = 512 + 4*2 = 520

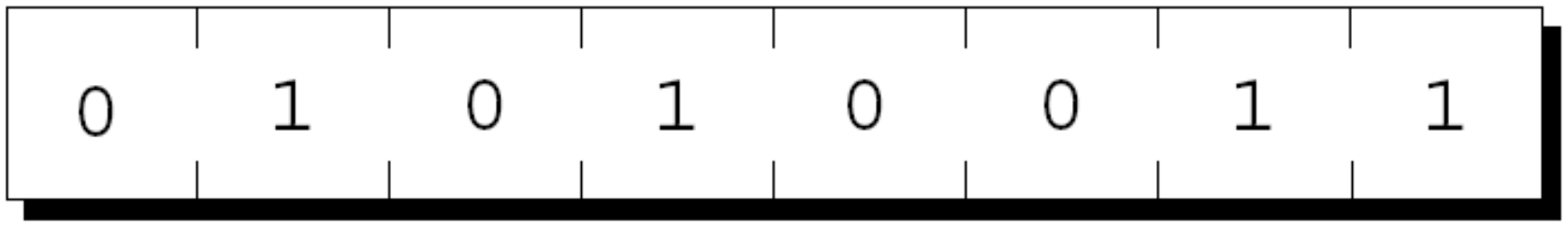


Pointer 자료

Pointer Variables

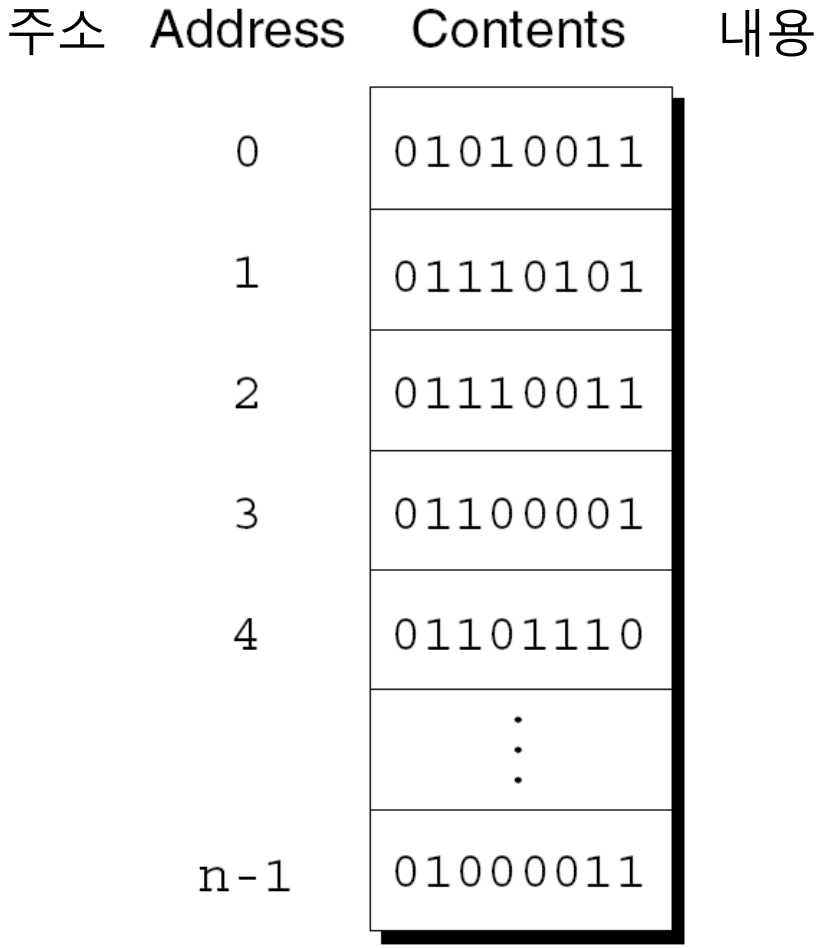
- The first step in understanding pointers is visualizing what they represent at the machine level.
  포인터를 이해하는 첫 단계는 기계레벨에서 어떻게 표현되는지 가시화하는 것
- In most modern computers, main memory is divided into ***bytes,*** with each byte capable of storing eight bits of information:
  메인메모리는 바이트로 구분되어 있고 각 바이틑 8bit 정보를 저장할 수 있음

- Each byte has a unique ***address.***
  각 바이트는 유일한 **주소**를 갖고 있음



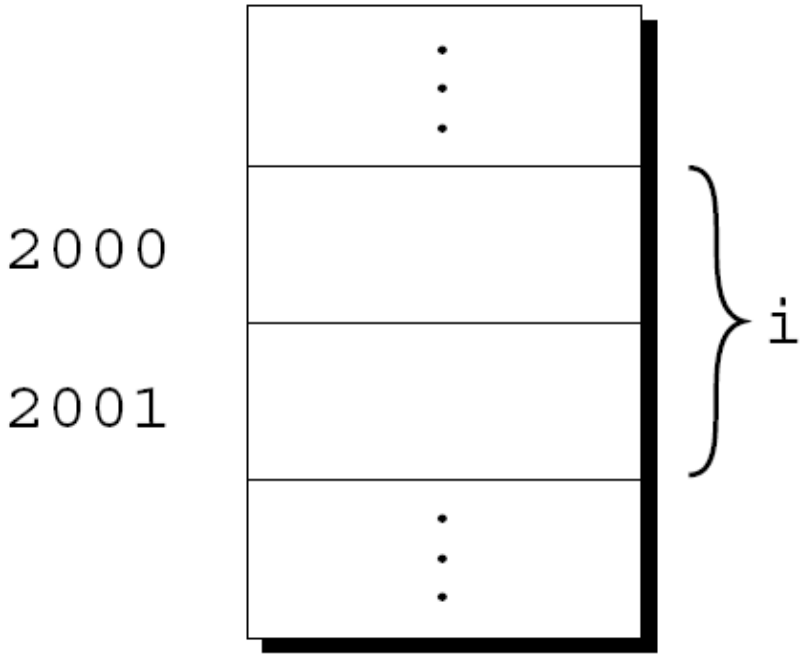
Pointer Variables

- If there are *n* bytes in memory, we can think of addresses as numbers that range from 0 to *n* – 1:
  n 바이트의 메모리가 있다면 주소는 0부터 n-1까지라고 가정



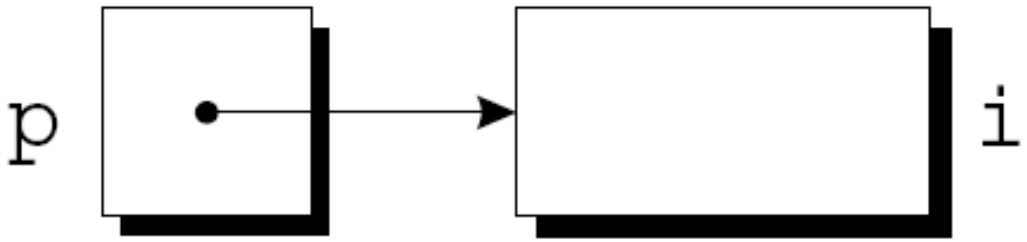
Pointer Variables

- Each variable in a program occupies one or more bytes of memory. 각 변수는 1 또는 2 바이트의 메모리를 차지함
- The address of the first byte is said to be the address of the variable. 변수의 주소는 첫번째 바이트의 주소임
- In the following figure, the address of the variable `i` is 2000:
  변수 i의 주소는 2000



Pointer Variables

- Addresses can be stored in special ***pointer variables.***
  주소는 포인터 변수라는 특별한 변수에 저장됨
- When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`.
  변수 i의 주소를 포인터 변수 p에 저장한다면, p는 i를 가리킨다(포인트한다)라고 표현함
- A graphical representation: 도식화 해보자



Declaring Pointer Variables

- When a pointer variable is declared, its name must be preceded by an asterisk:
  포인터 변수를 선언할 때, 변수 명 앞에 별표를 해야 함

```
int *p;
```

- `p` is a pointer variable capable of pointing to ***objects*** of type `int`.
  해석하면 포인터 변수 p는 int 타입 객체를 포인트 할 수 있다는 의미임
- We use the term *object* instead of *variable* since `p` might point to an area of memory that doesn't belong to a variable.
  변수 대신 객체라 부르는 이유는 p가 변수가 아닌 다른 메모리 영역을 가리킬 수도 있기 때문

Declaring Pointer Variables

- Pointer variables can appear in declarations along with other variables: 포인터 변수는 다른 변수들과 같이 선언 될 수 있음

```
int i, j, a[10], b[20], *p, *q;
```

- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):
  모든 포인터 변수는 특정 타입(참조 타입)의 객체만 포인트할 수 있음

```
int *p;     /* points only to integers   */
double *q;  /* points only to doubles    */
char *r;    /* points only to characters */
```

- There are no restrictions on what the referenced type may be. 참조 타입에 대한 제한 조건은 없음

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers. 포인터에 활용할 수 있는 연산자를 c에서 제공함
  - To find the address of a variable, we use the `&` (address) operator.
    변수의 주소는 &(주소) 연산자를 통해 얻음
  - To gain access to the object that a pointer points to, we use the `*` (***indirection***) operator.
    포인터가 포인트하는객체를 접근하기 위해서는 *(간접참조, 간접)연산자를 씀

The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:
  포인터 변수를 선언하는 것은 변수 자체를 위한 공간만 할당; 객체 참조안함

```
int *p;  /* points nowhere in particular */
```

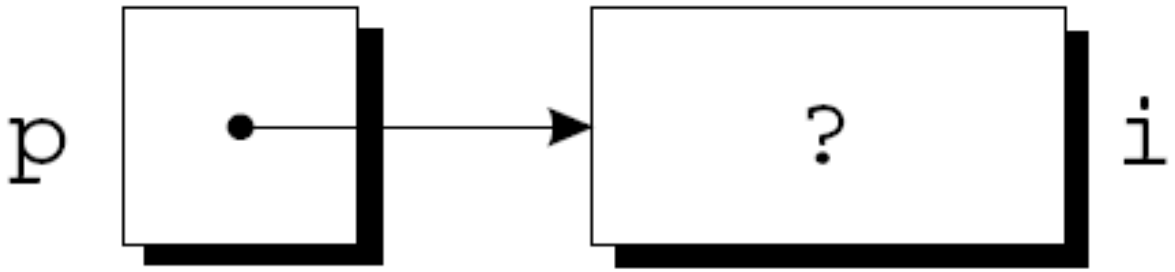
- It's crucial to initialize `p` before we use it.
  포인터 변수 p를 사용전에 초기화하는 것이 중요!!

The Address Operator

- One way to initialize a pointer variable is to assign it the address of a variable:
  포인터 변수를 초기화하는 방법 중 하나는 다른 변수의 주소를 할당하는 것

```
int i, *p;
…
p = &i;
```

- Assigning the address of `i` to the variable `p` makes `p` point to `i`:
  변수 i의 주소를 (포인터)변수 p에 저장하면 (포인터)변수 p가 i를 포인팅함



The Address Operator

- It's also possible to initialize a pointer variable at the time it's declared:
  또 다른 방법: 선언하면서 포인터 변수 초기화

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

- The declaration of `i` can even be combined with the declaration of `p`:
  변수 i를 선언하면서 포인터 변수 p를 i의 주소를 포인팅하도록 초기화 할 수 있음

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

The Indirection Operator

- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.
  포인터 변수가 어떤 객체를 포인팅하면 * (간접참조) 연산자를 사용해서 객체에 저장된 것에 접근할 수 있음
- If `p` points to `i`, we can print the value of `i` as follows:
  p가 i를 가리킬 때 i의 값을 다음처럼 출력할 수 있음

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

- Applying `&` to a variable produces a pointer to the variable.
  Applying `*` to the pointer takes us back to the original variable:
  어떤 변수에 &를 쓰면 그 변수를 가리킬 수 있는 주소를 알려줌
  다시 *를 쓰면 주소에 저장된 값을 알려줌

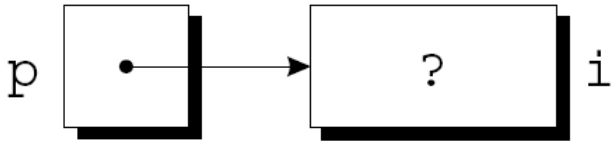
```
j = *&i;    /* same as j = i; 서로 동치 */
```

The Indirection Operator

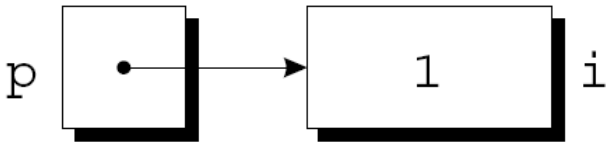
- As long as `p` points to `i`, `*p` is an ***alias*** for `i`.
  p가 i를 가리키는 동안에는 *p는 i에 대한 또 다른 이름임
  - `*p` has the same value as `i`. *p는 i와 동일한 값을 갖음
  - Changing the value of `*p` changes the value of `i`.
    *p의 값을 변경하면 i의 값도 변경됨
- The example on the next slide illustrates the equivalence of `*p` and `i`.
  다음 슬라이드의 예제를 살펴보자.

The Indirection Operator

```
p = &i;
```

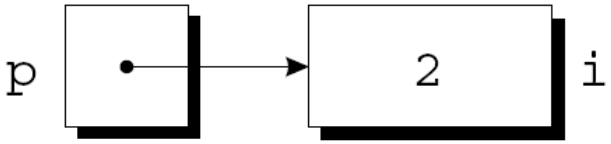



```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
```

```
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:
  간접 참조 연산자를 초기화하지 않고 사용하면 오동작을 일으킴

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assigning a value to `*p` is particularly dangerous:
  포인터 변수 *p에 상수를 저장하는 것은 특히 위험!! 왜일까?

```
int *p;
*p = 1;    /*** WRONG ***/
```

Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
  할당 연산자를 통해 동일한 타입에 대한 포인터를 복사가능
- Assume that the following declaration is in effect:
  다음과 같이 선언했다고 가정

```
int i, j, *p, *q;
```

- Example of pointer assignment: 포인터 할당 예제

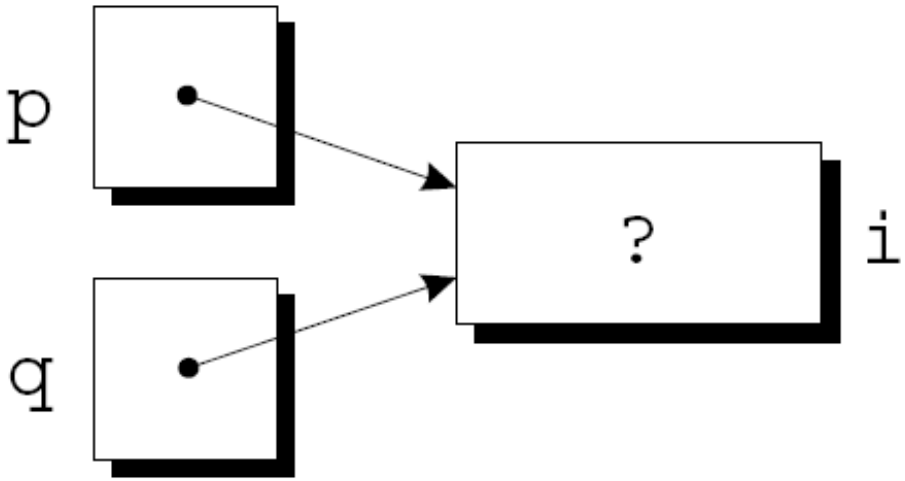
```
p = &i;
```

Pointer Assignment

- Another example of pointer assignment:
  또 다른 포인터 할당 예제

  `q = p;`

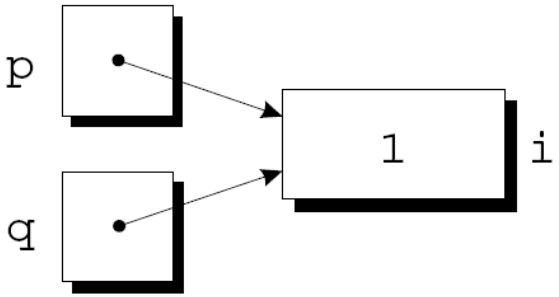
  `q` now points to the same place as `p`:
  q가 이제 p가 가리키는 곳을 똑같이 가리킴



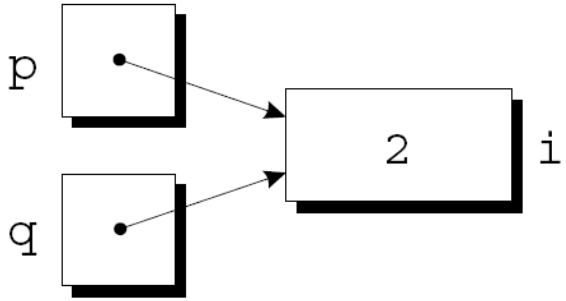
Pointer Assignment

- If `p` and `q` both point to `i`, we can change `i` by assigning a new value to either `*p` or `*q`:
  p와 q가 i를 가리킨다면, *p 또는 *q의 값 변경으로 i의 값을 변경 가능

```
*p = 1;
```




```
*q = 2;
```



- Any number of pointer variables may point to the same object. 포인터 변수가 몇 개든 상관없이 동일한 객체를 가리킬 수 있음

Pointer Assignment

- Be careful not to confuse

  ```
  q = p;
  ```

  with

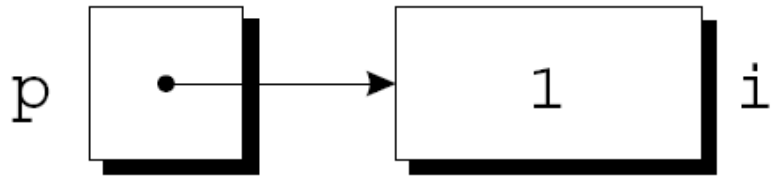
  ```
  *q = *p;
  ```

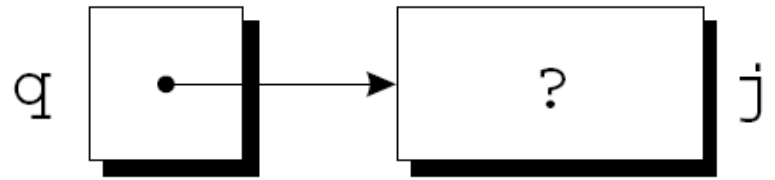
  위의 두 문장이 서로 갖다고 생각하지 말것

- The first statement is a pointer assignment, but the second is not.
  첫 번째 문장은 포인터 할당이지만, 두 번째 문장은 아님
- The example on the next slide shows the effect of the second statement. 다음 슬라이드에서 두 번째 문장의 의미를 살펴보자

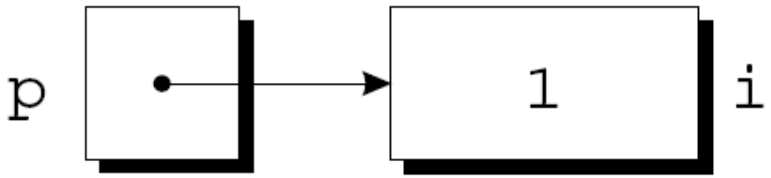
Pointer Assignment

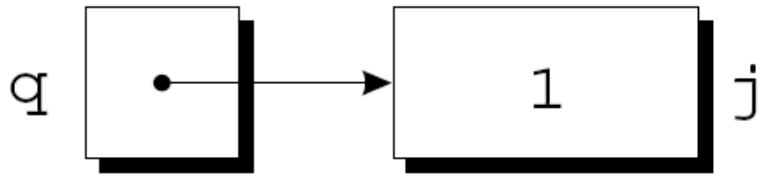
```
p = &i;
q = &j;
i = 1;
```






```
*q = *p;
```





Pointers as Arguments

- In Chapter 9, we tried—and failed—to write a `decompose` function that could modify its arguments.
  9장에서 decompse 함수가 실수 부분과 소수 부분을 분리하는 것을 실패하였음
- By passing a *pointer* to a variable instead of the *value* of the variable, `decompose` can be fixed.
  변수의 값 대신 변수에 대한 포인터를 전달하는 것으로 문제를 해결할 수 있음

Pointers as Arguments

- New definition of `decompose`: 새로운 decompose 함수의 정의

```
void decompose(double x, long *int_part,
               double *frac_part)
{
  *int_part = (long) x;
  *frac_part = x - *int_part;
}
```

- Possible prototypes for `decompose`: 이 함수의 프로토타입

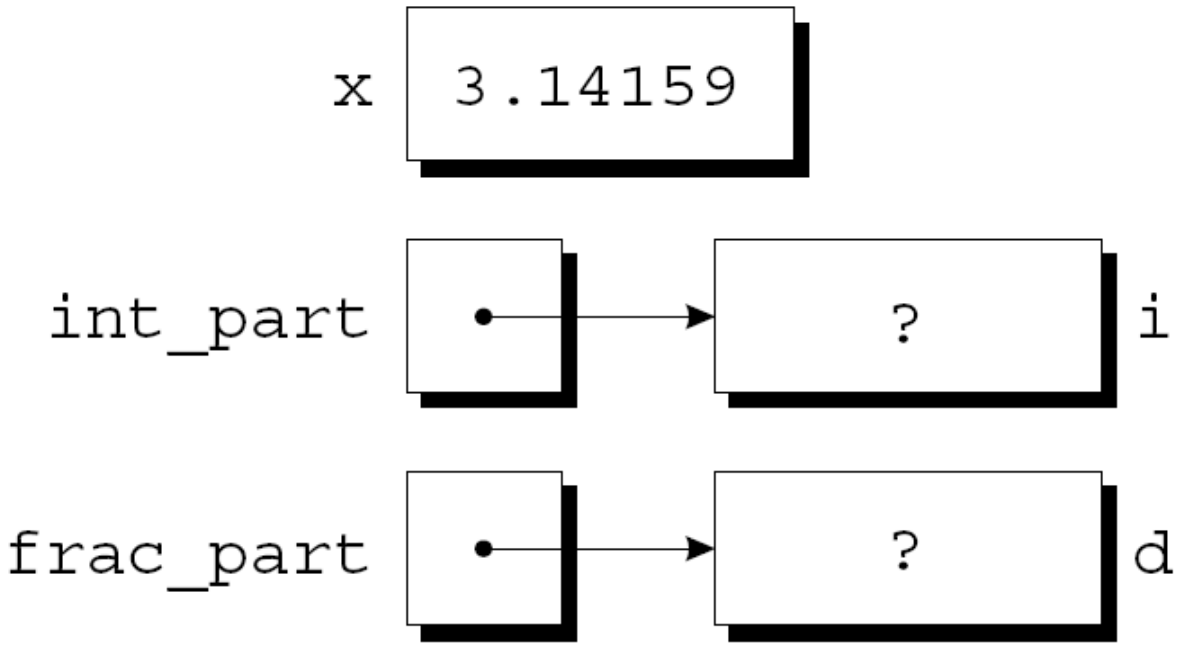
```
void decompose(double x, long *int_part,
               double *frac_part);
void decompose(double, long *, double *);
```

Pointers as Arguments

- A call of `decompose`: 함수 호출 방법

```
decompose(3.14159, &i, &d);
```

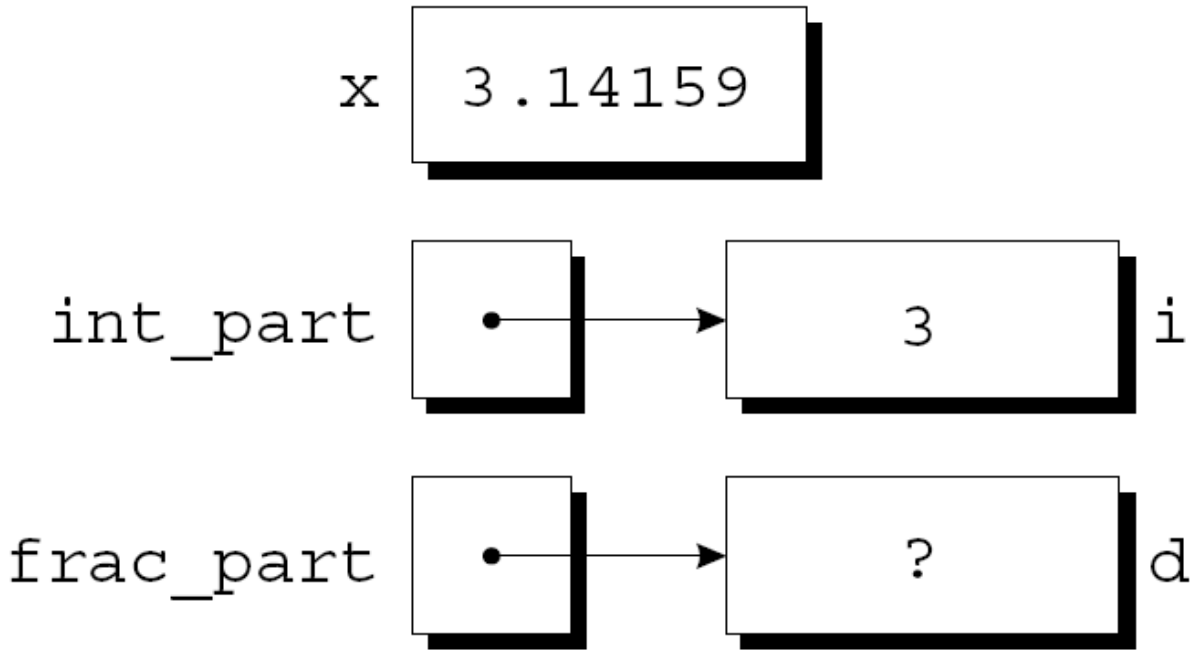
- As a result of the call, `int_part` points to `i` and `frac_part` points to `d`:
  호출의 결과로 int_part는 i를 가리키고 frac_part는 d를 가리킴



Pointers as Arguments

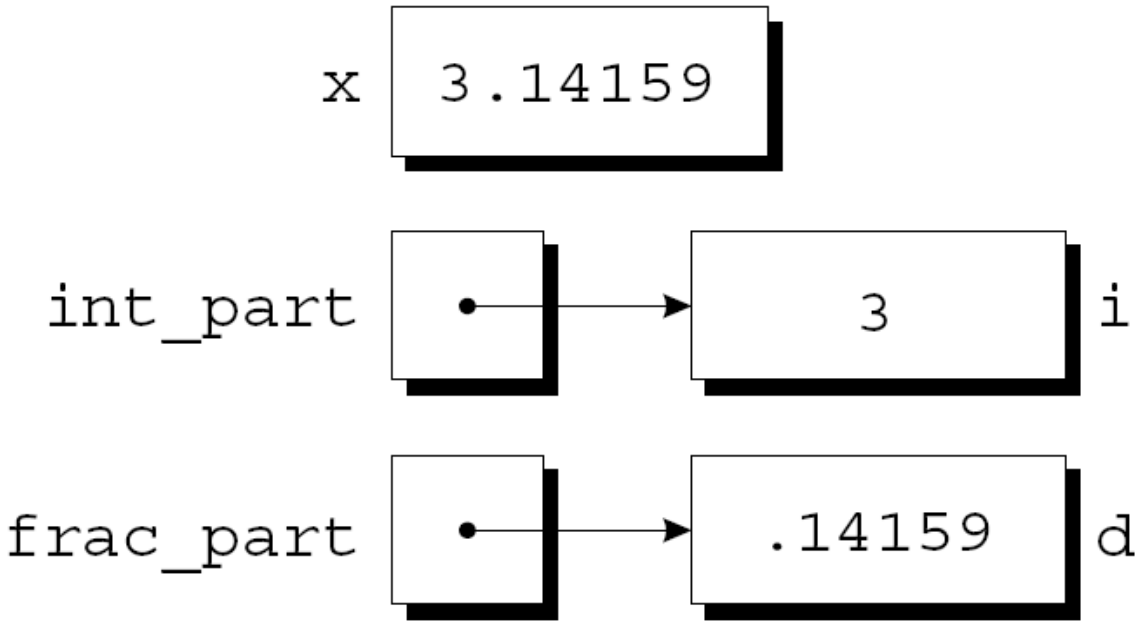
- The first assignment in the body of `decompose` converts the value of `x` to type `long` and stores it in the object pointed to by `int_part`:

  decompse 함수의 첫 할당문은 x의 값을 long 타입으로 변경한 후 int_part가 가리키는 객체에 저장함



Pointers as Arguments

- The second assignment stores `x – *int_part` into the object that `frac_part` points to:
  두 번째 할당문은 x - *int_part 의 계산 결과를 frac_part가 가리키는 객체에 저장함



Pointers as Arguments

- Arguments in calls of `scanf` are pointers:
  scanf 호출의 인자에도 포인터가 포함되어 있음

```
int i;
…
scanf("%d", &i);
```

  Without the `&`, `scanf` would be supplied with the *value* of `i`.
  & 없이는 scanf는 i의 값을 전달 받게 됨

Pointers as Arguments

- Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator:
  scanf의 인자가 포인터야 하지만, 모든인자 값이 &연산자가 있어야 하는 것은 아님

```
int i, *p;
…
p = &i;
scanf("%d", p);
```

- Using the `&` operator in the call would be wrong:
  이 경우 & 연산자를 호출하는 것은 잘못된 것임

```
scanf("%d", &p);    /*** WRONG ***/
```

Pointers as Arguments

- Failing to pass a pointer to a function when one is expected can have disastrous results.
  함수에 포인터 전달이 실패 된다면 결과가 이상할 수 있다.
- A call of `decompose` in which the `&` operator is missing:
  decompe의 호출에 &연산자를 안 쓴다면

```
decompose(3.14159, i, d);
```

- When `decompose` stores values in `*int_part` and `*frac_part`, it will attempt to change unknown memory locations instead of modifying `i` and `d`.
  decompose는 i 와 d의 값을 변경하는 대신 *int_part와 *frac_part이 가리는 임의의 주소의 값을 변경함
- If we've provided a prototype for `decompose`, the compiler will detect the error.
  decompse의 프로토타입을 선언했었다면 오류를 검출 했을 것임
- In the case of `scanf`, however, failing to pass pointers may go undetected.
  단, scanf의 경우 포인터를 사용 안더라도 검출 안될 수 있음

Program: Finding the Largest and Smallest Elements in an Array

- The `max_min.c` program uses a function named `max_min` to find the largest and smallest elements in an array.
  max_min.c이라는 프로그램은 max_min 함수를 사용하여 배열에서 가장 큰 수와 작은 수를 찾음
- Prototype for `max_min`: 프로토타입은 다음과 같음

```
void max_min(int a[], int n, int *max, int *min);
```

- Example call of `max_min`: 호출 예제

```
max_min(b, N, &big, &small);
```

- When `max_min` finds the largest element in `b`, it stores the value in `big` by assigning it to `*max`. 이 함수가 배열 b에서 가장 큰 요소를 찾으면 *max를 통해서 big에 할당
- `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`. 이 함수가 배열 b에서 가장 작은 요소를 찾으면 *min을 통해서 small에 할당

Program: Finding the Largest and Smallest Elements in an Array

- `max_min.c` will read 10 numbers into an array, pass it to the `max_min` function, and print the results:
  사용자로부터 10개의 수를 받아 들이고 max_min함수에 전달함. 이후 결과를 출력

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

maxmin.c

```
/* Finds the largest and smallest elements in an array */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
  int b[N], i, big, small;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &b[i]);
```

```
  max_min(b, N, &big, &small);

  printf("Largest: %d\n", big);
  printf("Smallest: %d\n", small);

  return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

Using **`const`** to Protect Arguments

- When an argument is a pointer to a variable `x`, we normally assume that `x` will be modified:
  어떤 인자가 변수 x에 대한 포인터라 할 때 x가 변경될 것을 가정함

  ```
  f(&x);
  ```

- It's possible, though, that `f` merely needs to examine the value of `x`, not change it.
  때로는 x의 값을 변경하는 것이 아니라 확인만하고자 할 수 있음

- The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage.
  이 때 포인터를 쓰는 것은 효율성 때문임: 값으로 전달하면 변수를 복사하는 과정에서 공간과 시간이 낭비 될 수 있음, 특히 변수가 많은 저장 공간을 필요로 한다면 문제는 더 심각함

Using **`const`** to Protect Arguments

- We can use `const` to document that a function won't change an object whose address is passed to the function.
  이 경우 const라는 키워드를 사용하여 함수가 전달 받은 변수의 주소가 가리키는 객체가 변경되지 않을 것을 명시 할 수 있음
- `const` goes in the parameter's declaration, just before the specification of its type:
  const는 매개 변수를 선언할 때 기록하고 타입 앞에 붙여야 함

```
void f(const int *p)
{
  *p = 0;    /*** WRONG ***/
}
```

Attempting to modify `*p` is an error that the compiler will detect.
*p를 변경하려고 시도하면 컴파일러가 해당 오류를 검출함

Pointers as Return Values

- Functions are allowed to return pointers:
  함수는 포인터를 리턴할 수 있음

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

- A call of the `max` function: max를 호출하는 방법

```
int *p, i, j;
…
p = max(&i, &j);
```

  After the call, `p` points to either `i` or `j`. 호출 결과, p는 i 또는 j를 가리킴

Pointers as Return Values

- Although `max` returns one of the pointers passed to it as an argument, that's not the only possibility.
  max가 인자로 전달 받은 포인터 중 하나를 리턴하지만 다른 것도 가능함
- A function could also return a pointer to an external variable or to a static local variable.
  함수가 external 변수나 정적 지역 변수에 대한 포인터도 리턴할 수 있음
- Never return a pointer to an *automatic* local variable:
  절대로 자동 지역 변수에 대한 포인터를 리턴하지 말 것

```
int *f(void)
{
  int i;
  …
  return &i;
}
```

The variable `i` won't exist after `f` returns.
변수 i는 f가 리턴하면 소멸됨

Pointers as Return Values

- Pointers can point to array elements.
  포인터는 배열의 원소를 포인트 할 수 있음
- If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`.
  만약 a가 배열이면 &a[i]는 a의 i번째 요소에 대한 포인터임
- It's sometimes useful for a function to return a pointer to one of the elements in an array.
  때로는 배열의 요소 중 하나에 대한 포인터를 리턴하는데 유용함
- A function that returns a pointer to the middle element of `a`, assuming that `a` has `n` elements:
  다음 함수는 배열 a의 가운데 요소에 대한 포인터를 리턴함, a는 n개 요소가 있음

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```