Ch.11 Pointers

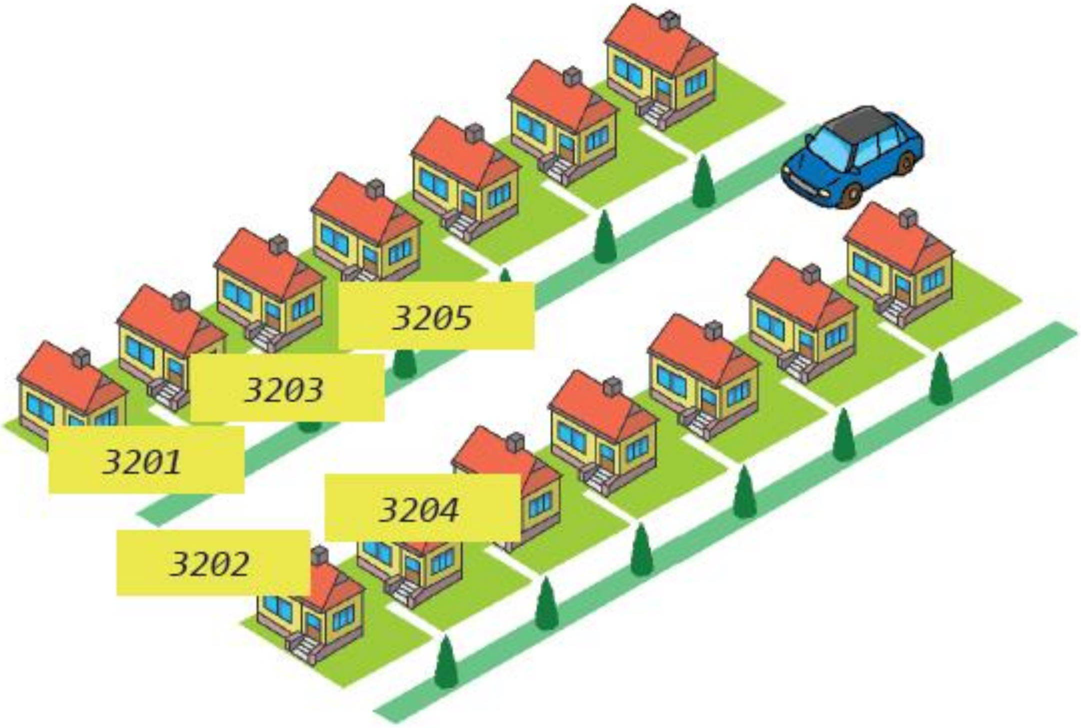
What you will learn in this chapter

- What is a pointer ?
- Address of variable
- Declaration of a pointer
- Indirect reference operator
- Pointer arithmetic
- Pointers and Arrays
- Pointers and functions

In this chapter
The basics of pointers
Learn knowledge .

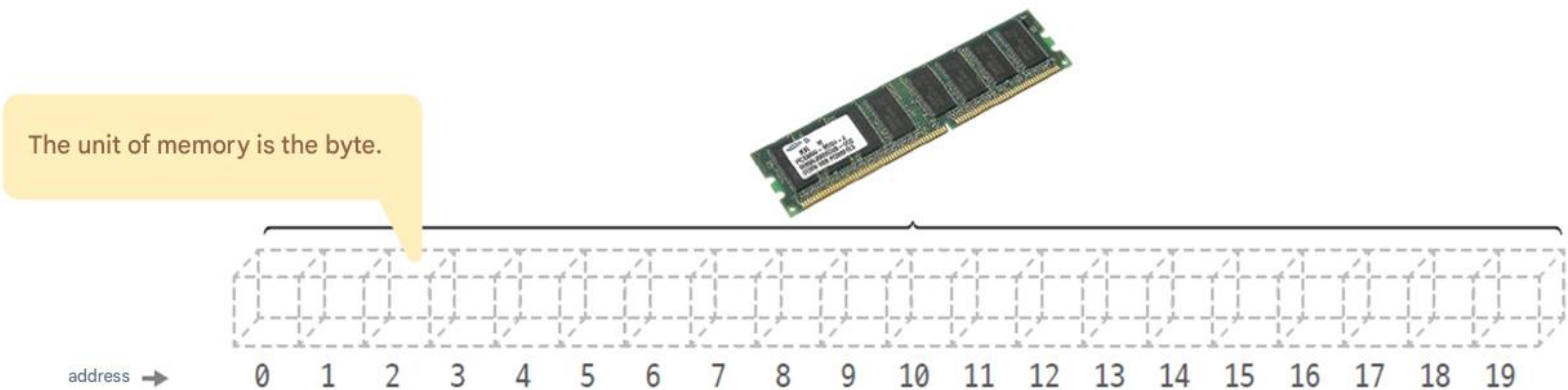
What is a pointer ?

- *Pointer :* A variable that has an address



Where is it stored in the variable ?

- Variables are stored in memory .
- Memory is accessed in bytes. (Minimum unit)
  - The address of the first byte is 0, the address of the second byte is 1, ...



Size of pointer

- Size of pointer depends on the system (architecture) not type of variables.

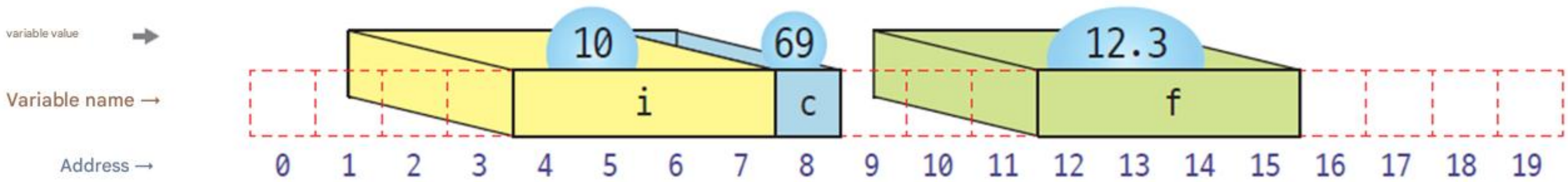
```
int *pi;
char *pc;
double *pd;

sizeof(pi) == sizeof(pc) == sizeof(pd);    // always same
```

System / Architecture	Address size	Pointer size
16-bit (old/embedded)	16 bits	2 bytes
32-bit systems (x86)	32 bits	4 bytes
64-bit systems (x86-64, ARM64)	64 bits	8 bytes
Windows 64-bit (LLP64)	64 bits	8 bytes
Linux/macOS 64-bit (LP64)	64 bits	8 bytes

Variables and Memory

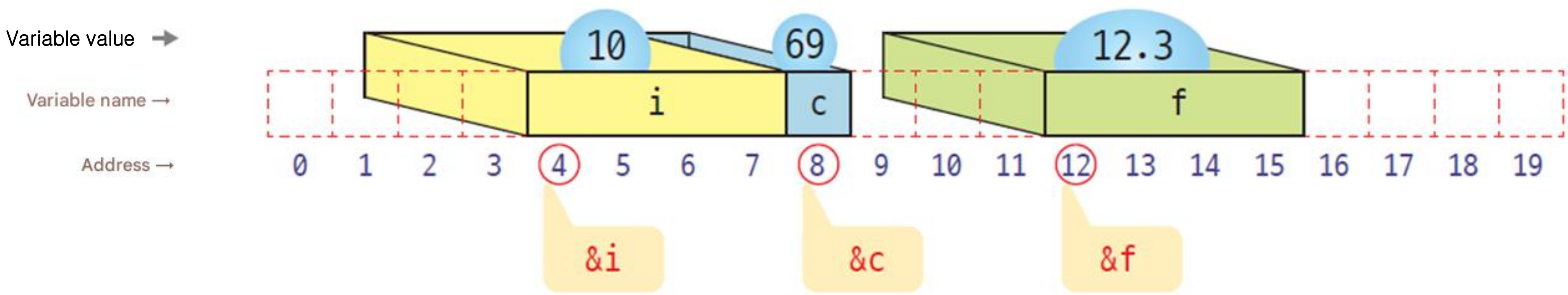
- The memory space occupied varies depending on the size of the variable.
- char type variable : 1 byte, int type variable : 4 bytes, …

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



Address of variable

- Operator used to obtain the address of a variable : &
- Address of variable i : &i



Address of variable

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

	printf ( "Address of i : %p\n" , &i ); // Print address of
	printf ( "Address of c : %p\n" , &c); // Print address of
	printf ( "Address of f : %p\n" , &f); // Print address of
	return 0;
}
```

The program The address will be different each time you run it .

Debugger

```
i 's address : 0000003D69DDF974
Address of c : 0000003D69DDF994
Address of f : 0000003D69DDF9B8
```

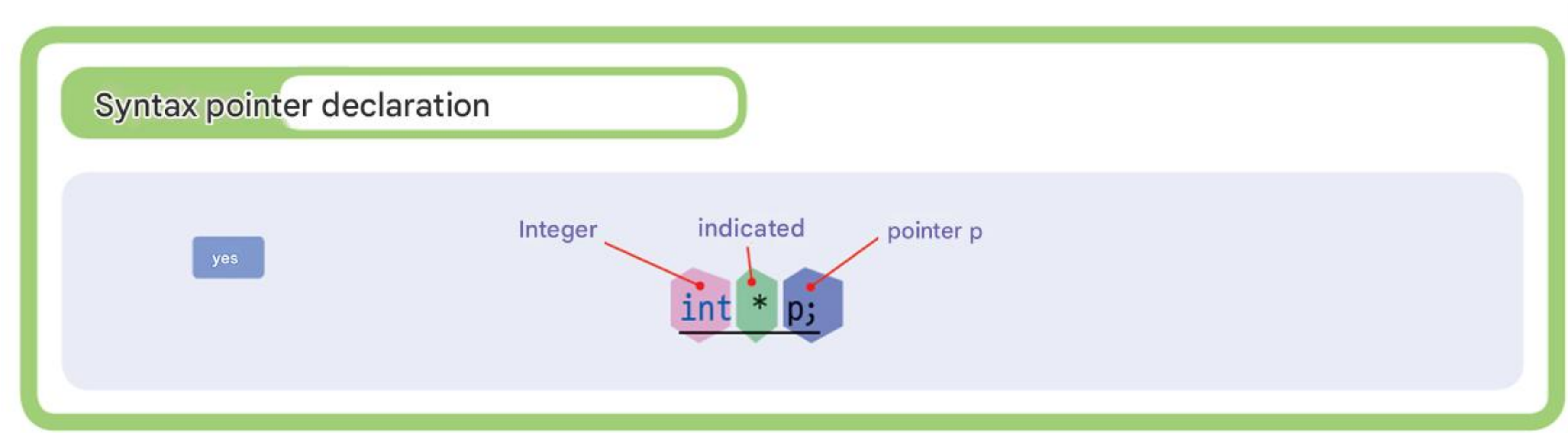
caution

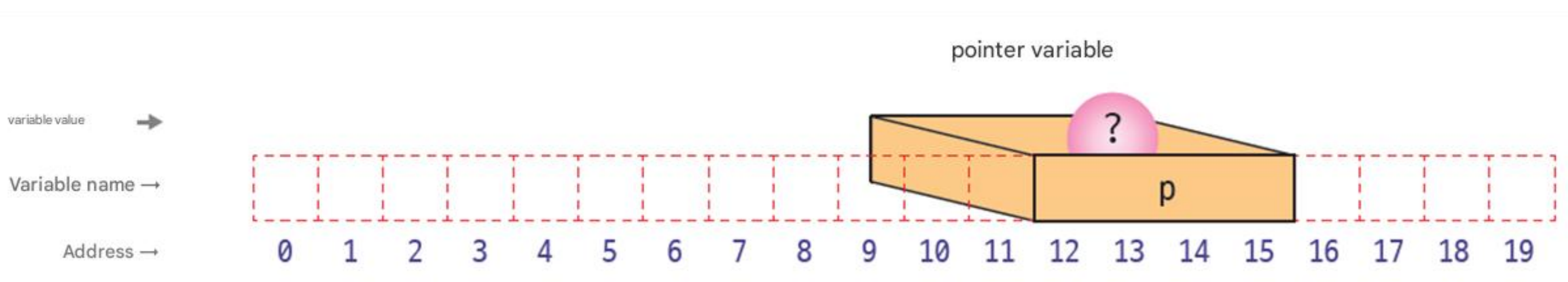
- Be careful when declaring multiple pointer variables on one line. Declaring them as follows is incorrect :
  - int *p1, p2, p3; // (×) p2 and p3 become integer variables .

- To declare correctly, you must do the following :
  - int *p1, *p2, *p3; // (○) p2 and p3 are pointer variables of integer type.

Declaration of a pointer

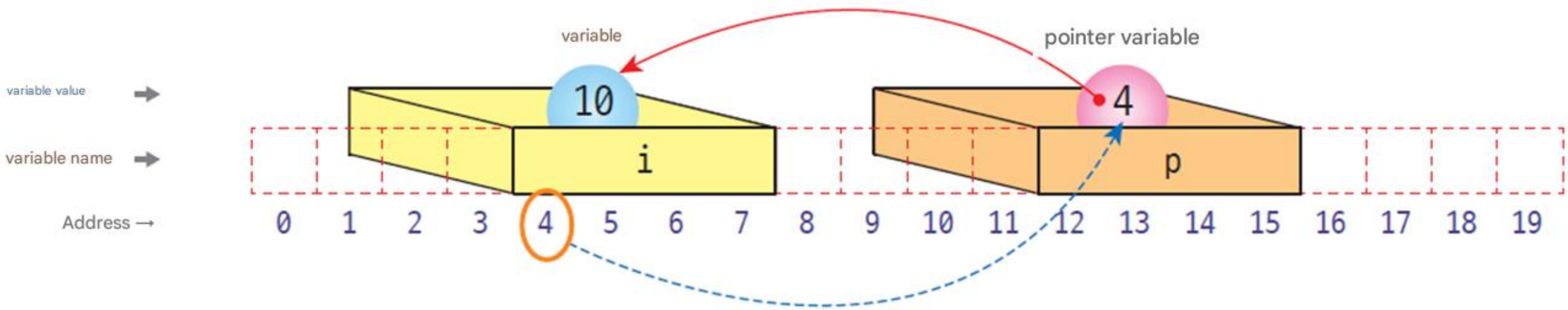
- A pointer is declared by specifying the data type it points to, followed by an asterisk (*), and then the pointer's name.





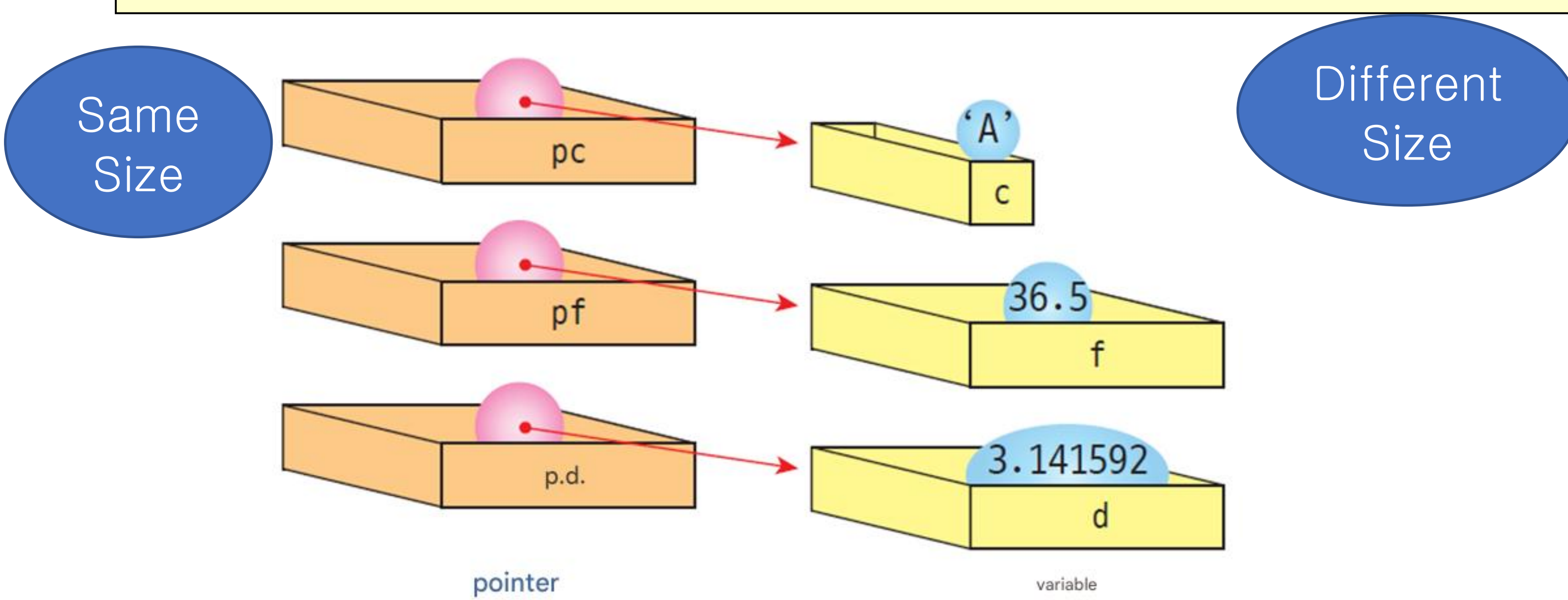
Assign the variables to pointers

```
int    i = 10; // declare of integer variable i
int    * p;     // declare of pointer variable p
p = & i ;       // assign the address of variable i to pointer p
```



Declaration of various pointers

```
char c = 'A’;              // character type Variable c
float f = 36.5;            // Real number variable f
double d = 3.141592; // Real number Variable d

char *pc = &c;              // characters indicated pointer pc
float * pf = &f;            // Real number indicated Pointer pf
Double * pd = &d;         // Real number indicated Pointer pd
```



Example

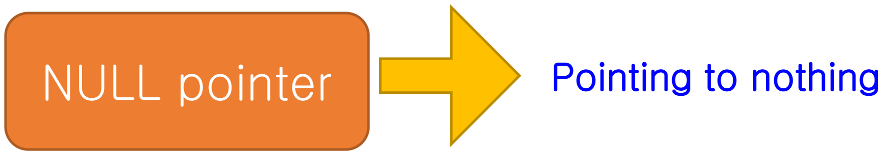
```
#include < stdio.h >

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
```

reference

- NULL is defined in stdio.h
- It represents address 0, and is a pointer constant defined in the header file as follows.
  - #define NULL ((void *)0)

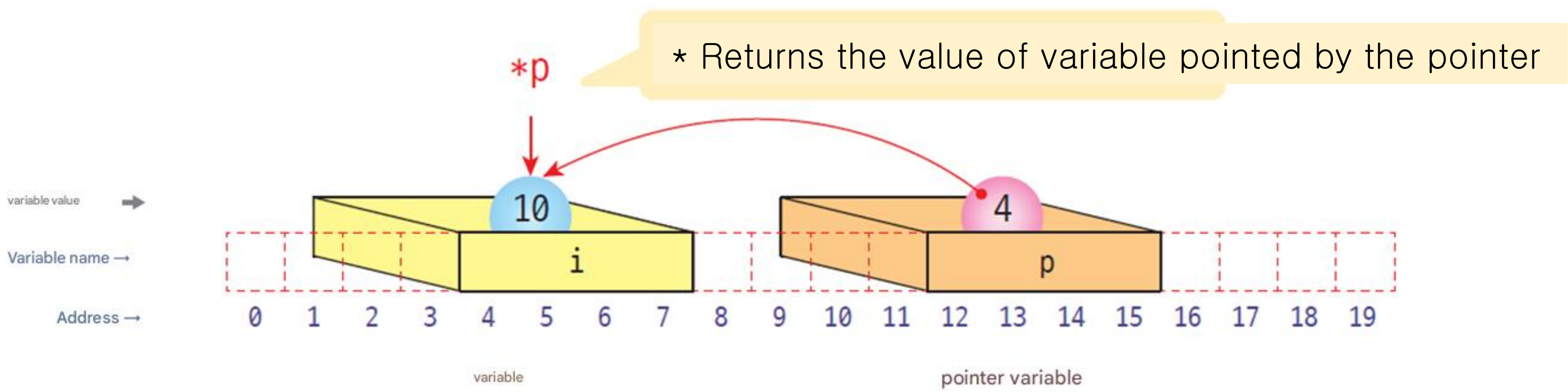
- Address 0 is generally unusable (CPU reserves it for interrupts). Therefore, if the value of a pointer variable is 0, we can assume that it is not pointing to anything .



Indirect reference operator

- Indirect reference operator * : Operator that retrieves the value pointed to by the pointer

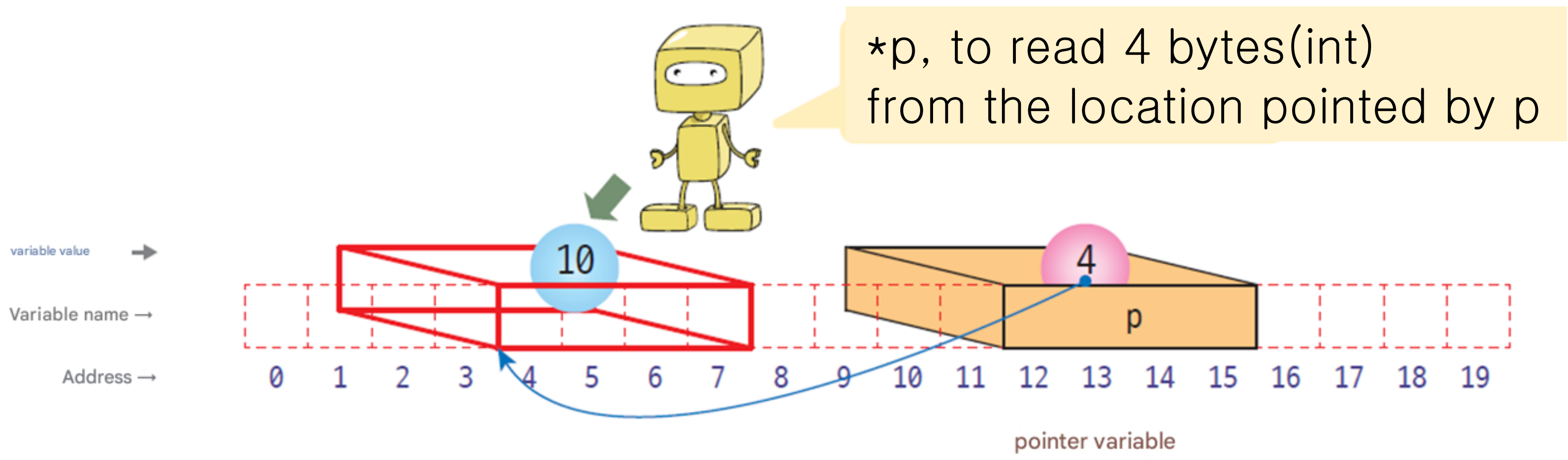
```
int i = 10;

int * p;
p = &i ;

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



Interpretation of indirect reference operators

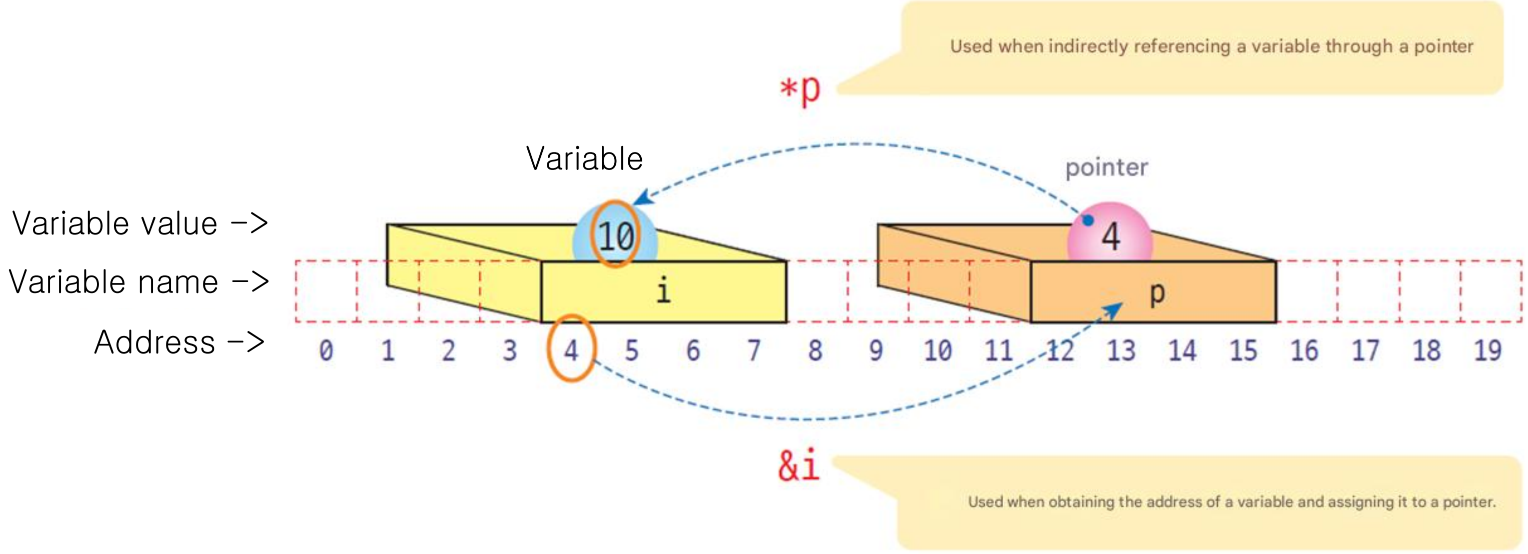
- Indirect reference operator : Reads a value based on the type of the pointer at the specified location.

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



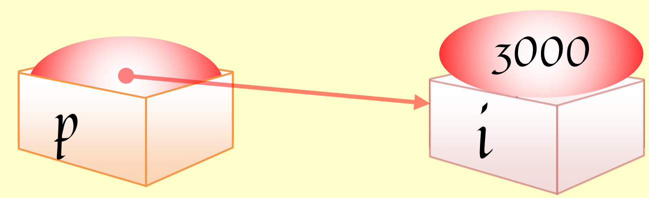
& operator and * operator

- & operator : Returns the address of a variable
- * Operator : Returns the contents of the location pointed by the pointer.



Pointer Example #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;
}
```



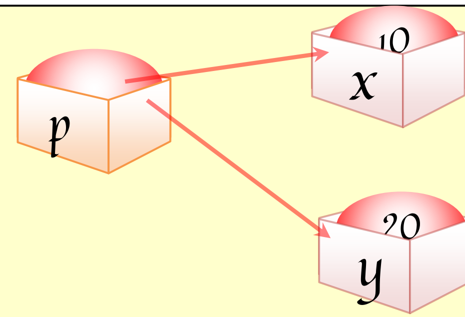

```
p = 0000006DEA0FFBD4
& i = 0000006DEA0FFBD4

i = 3000
*p = 3000
```

debugger

Practice
ch11-1.c

Pointer Example #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;
}
```

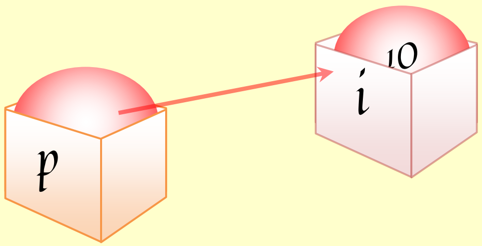



```
p = 0000007A8F3AF974
*p = 10

p = 0000007A8F3AF994
*p = 20
```

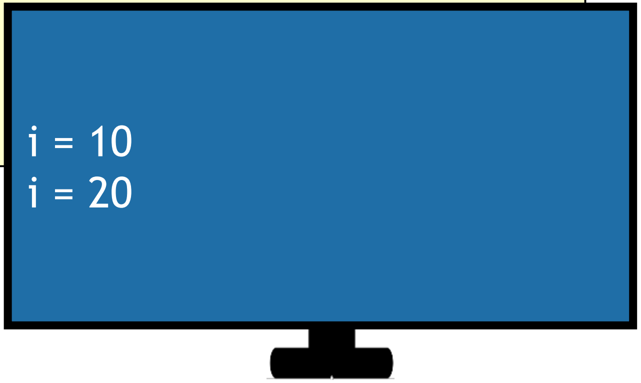
printf(“*p = %d\n, *p);

Pointer Example #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;
}
```



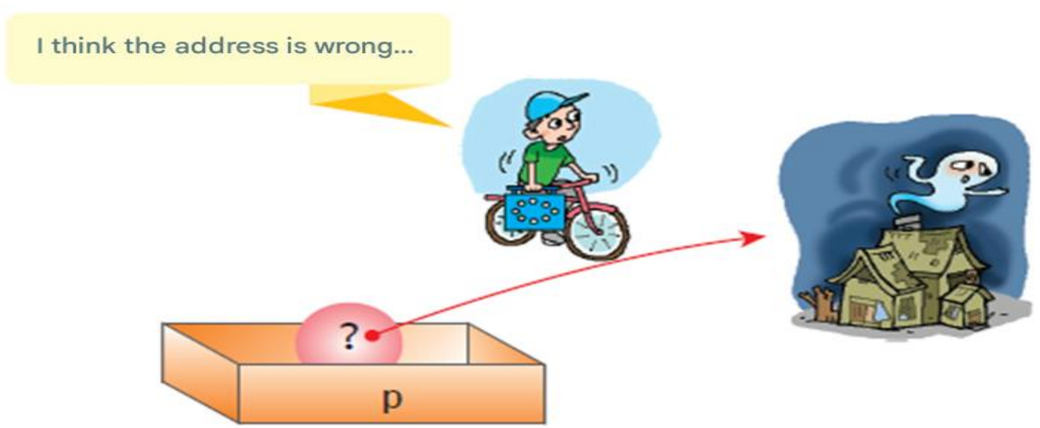
Change the value of a variable through a pointer.



Cautions when using pointers

- You should not use uninitialized pointers.
- The pointer contains an arbitrary memory address (a garbage value), so we have no way of knowing what it is pointing to.

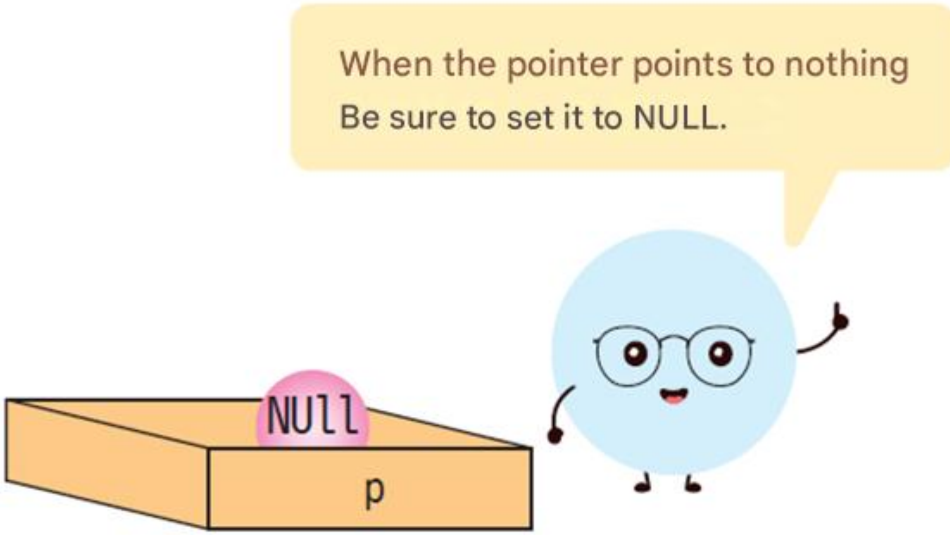
```c
int main( void )
{
	int *p;     // pointer p is not initialization.
	*p = 100;  // dangerous code
	return 0;
}
```



Cautions when using pointers

- If the pointer points to nothing, it is initialized to NULL .

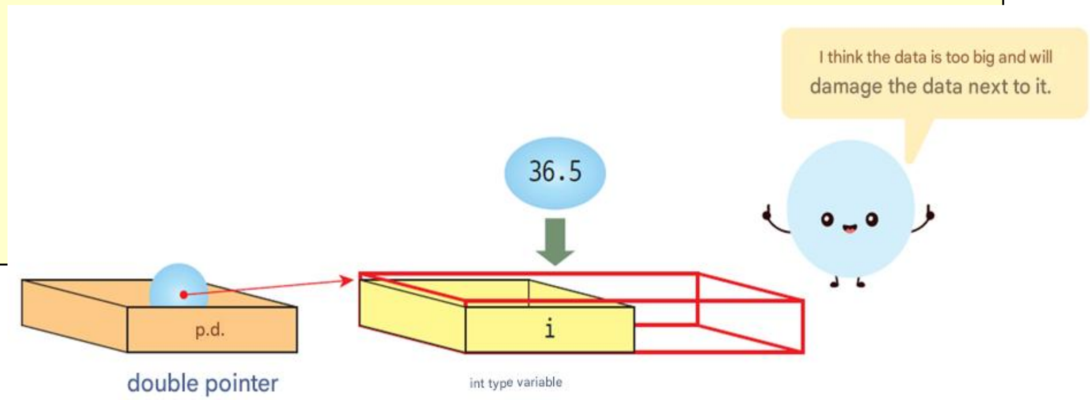
```
int *p = NULL;
```



Cautions when using pointers

- The type of the pointer and the type of the variable must match.

```c
#include < stdio.h >

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

      pd = &i ;    // error !
      *pd = 36.5;

    return 0;
}
```



Pointer arithmetic

- Possible operations : increment , decrement , addition , subtraction operations
- In the case of an increment operation, the value being increased is the size of the object pointed to by the pointer.

pointer type	++value that increases after operation
char	1
short	2
int	4
float	4
double	8

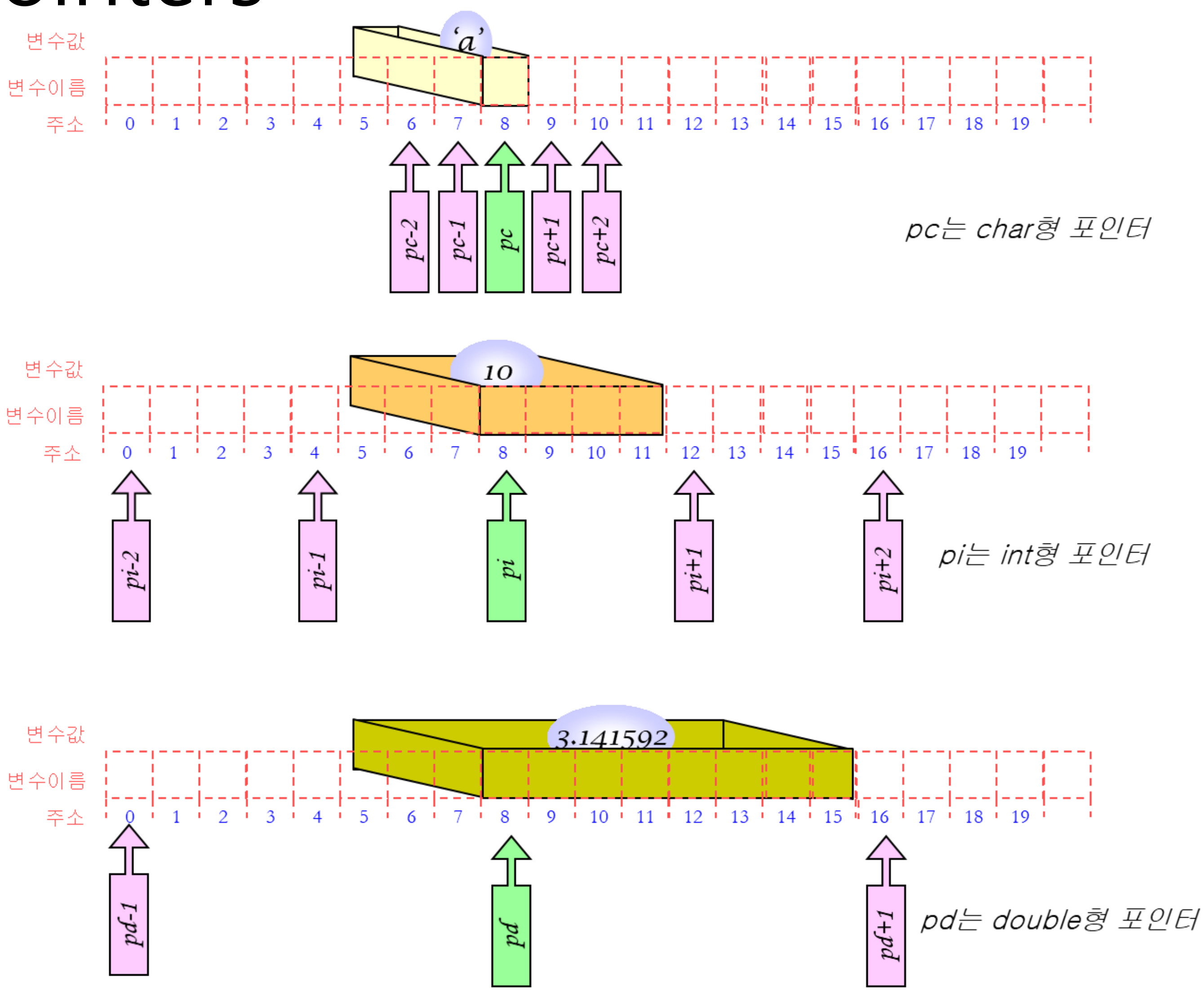
Increment operation example

Next page

```
// Increment/decrement operation of pointer
#include < stdio.h >

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

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

Increment and decrement operations of pointers



Indirect reference operator and increment/decrement operator

- (*p)++;
  - Increments the value stored at the location pointed to by p.
- *p++;
  - Increments p after retrieving the value from the location pointed to by p.

Indirect reference operator and increment/decrement operator

debugger

```
// Increment/decrement operation of pointer
#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);

 int j =  *pi++;
 printf ( " i = %d,  j = %d, pi = %p\n" , i , j, pi);
 return 0;
}
```

```
i = 10, pi = 000000FFEBCFF974
i = 11, pi = 000000FFEBCFF974
i = 11, j = 11, pi = 000000FFEBCFF978
```

Increments the value at the location pointed to pi.

After getting the value from the location pointed to by pi, increment pi.

What about read *pi after int j = *pi++? (Next value)

*pi++ means : Use the current value of p first, and then increment p afterward.

*++pi means : Increment pi first, then read from the new location.

Type conversion of pointers

- C language, you can explicitly change the type of a pointer when absolutely necessary.

```
double f = 3.14;
double *pd = &f;
int *pi;

pi = ( int *)pd;  // Casting a double* to an int* is dangerous
```

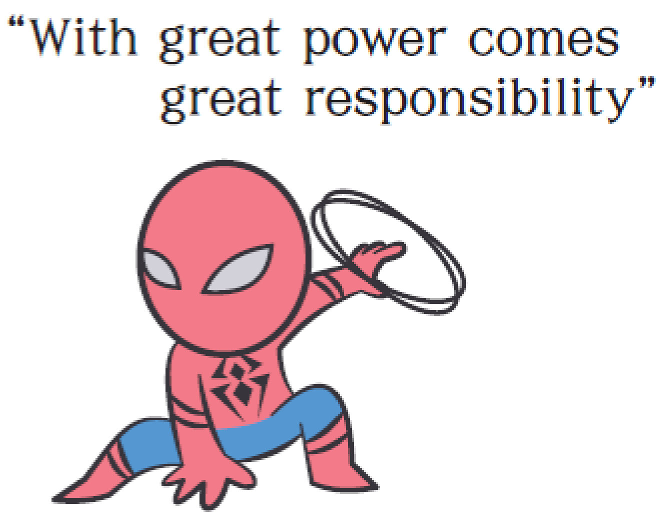
- This means that pi will now point to the same memory location as pd, but it will interpret that memory as an int instead of a double.
  This is generally unsafe because a double and an int have different sizes and alignment requirements.

- If your intention is to convert a double to an int, you should not cast pointers. You should convert the value itself:

```
double f = 3.14;
int i = (int)f;   // safe
```

Cautions when using pointers

- Pointers are both a strength and a weakness of the C language.
- Developers must use it responsibly .
- When using pointers, always remember the following quote from the Spider-Man movie :



https://www.youtube.com/shorts/y4gSN_XLyGw

Example

debugger

```c
#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
```

Little Endian

How to pass arguments to parameters

- Function How to pass arguments when calling
  - Call by value
    - Copy as a function It is delivered.
    - Basic methods in C.
  - Call by reference
    - The original is passed to the function.
    - In C, this can be emulated using pointers.

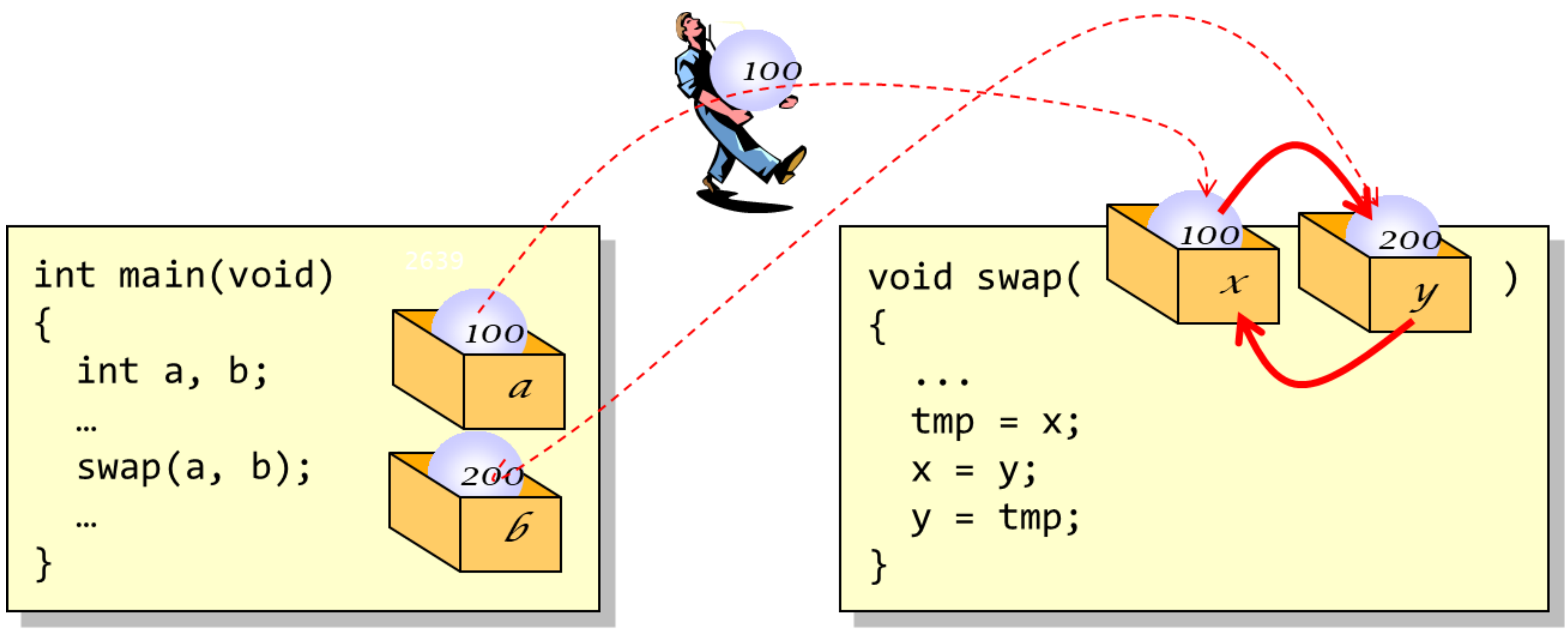
swap() function #1 (call by value)

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

	swap(a, b);

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

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

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

	printf(“x=%dy=%d\n”,x, y);
}
```

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

Call by value

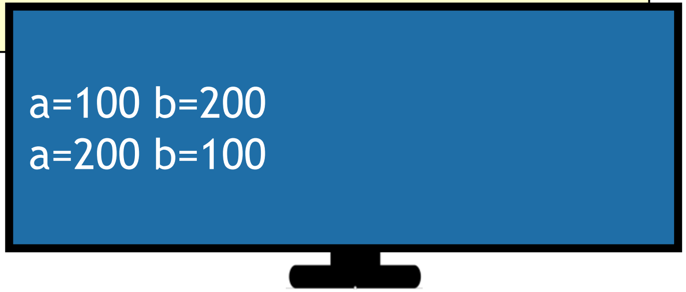


Practice
ch11-2.c

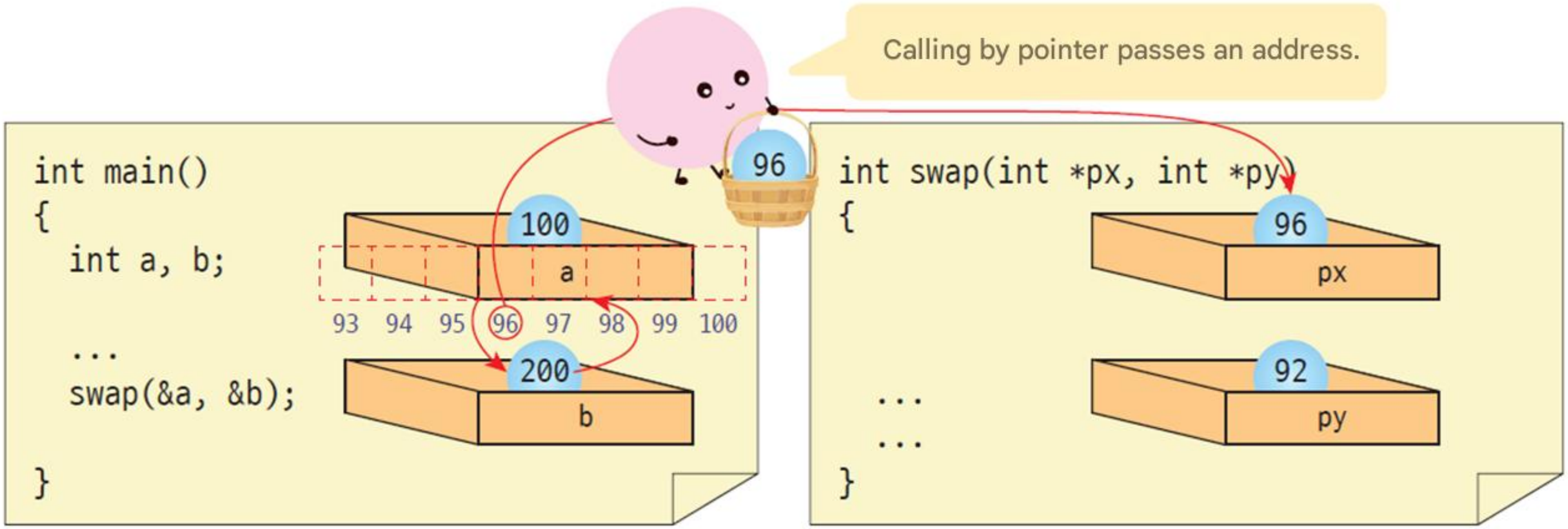
swap() function #2 (call by reference)

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

        swap(&a, &b);

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

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

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

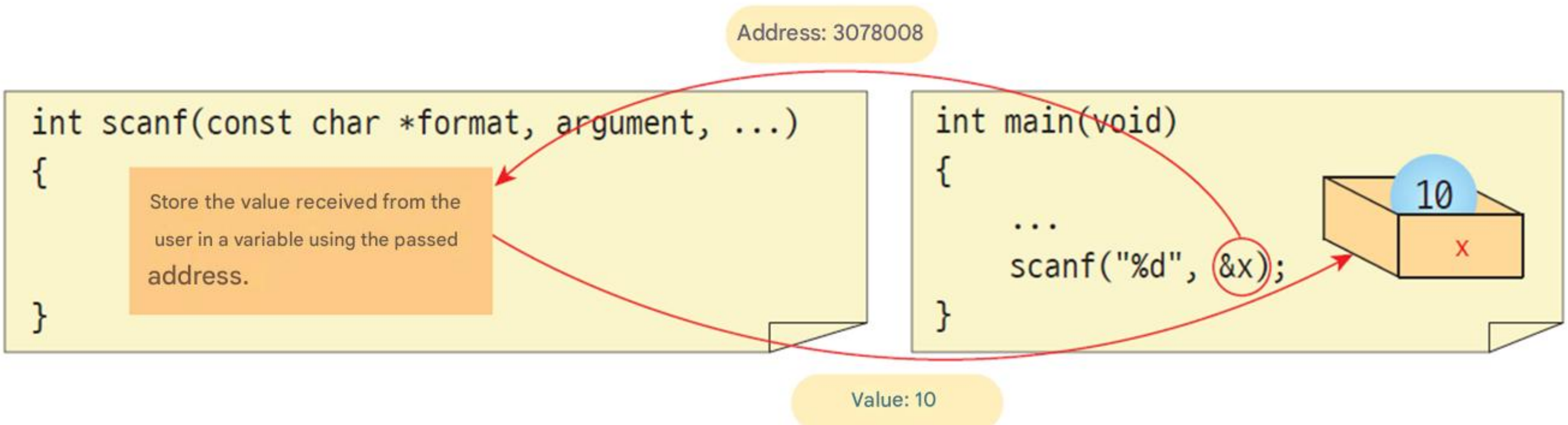


Call by reference



scanf () function

- Receives the address of a variable to store a value.



Note : How to prevent a function from changing a value through a pointer ?

- When declaring a function parameter, you can do so by adding const in front.
  Adding const in front means that the content pointed to pointer is a constant that cannot be changed.

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

Example

- If a function needs to return more than one value, one way to do this is to use pointers. Let's write a function that returns the result as parameter.

Return more than two results

```
#include <stdio.h>

// Addign two values, save the result
int add_numbers(int a, int b, int *result)
{
    if (result == NULL)    // 잘못된 포인터 검사
        return -1;

    *result = a + b;        // 계산 결과를 전달
    return 0;               // 성공
}

int main(void)
{
    int sum;

    if (add_numbers(3, 5, &sum) == -1)
        printf("Error!\n");
    else
        printf("Sum = %d\n", sum);

    return 0;
}
```

Return result as arguments

Cautions when returning a pointer

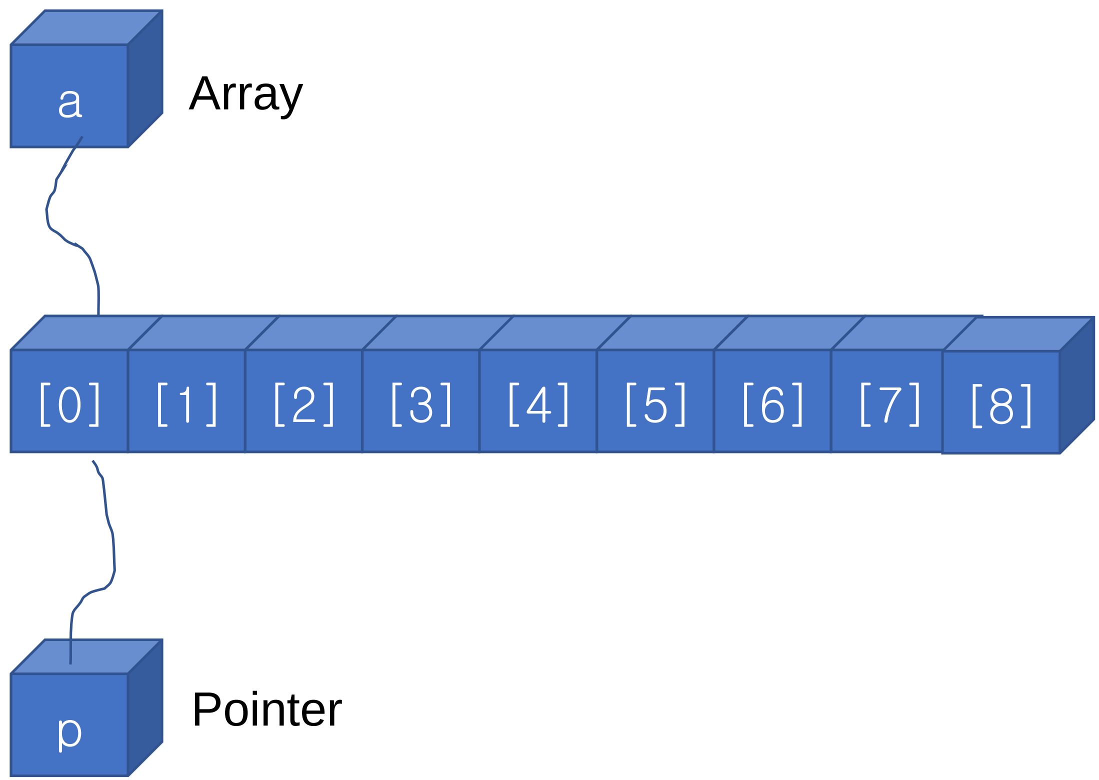
- The address of the variable that remains even after the function ends must be returned .
- If you return the address of a local variable, it will disappear when the function ends, so it is an error.

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

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

Local variables disappear when the function call ends, so you should not return the address of a local variable.

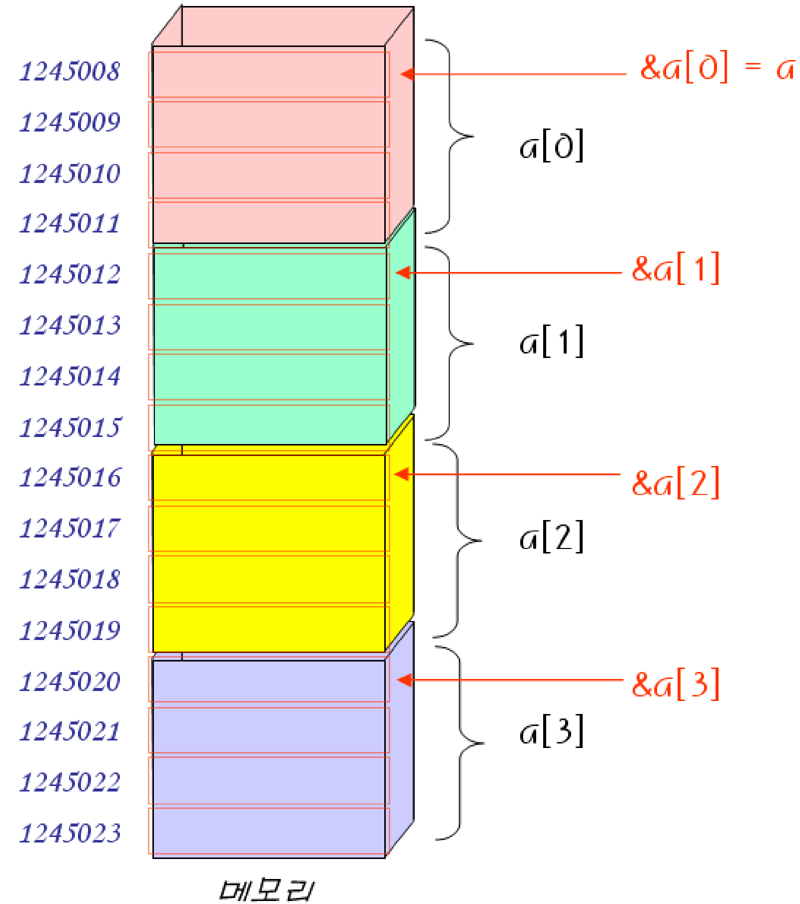
Pointers and Arrays

- Arrays and pointers have a very close relationship .
- The array name is actually a pointer .
- Pointers can be used like arrays .



Pointers and Arrays

```c
// Pointer and Array of relationship
#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
```

Example

Practice
ch11-3.c

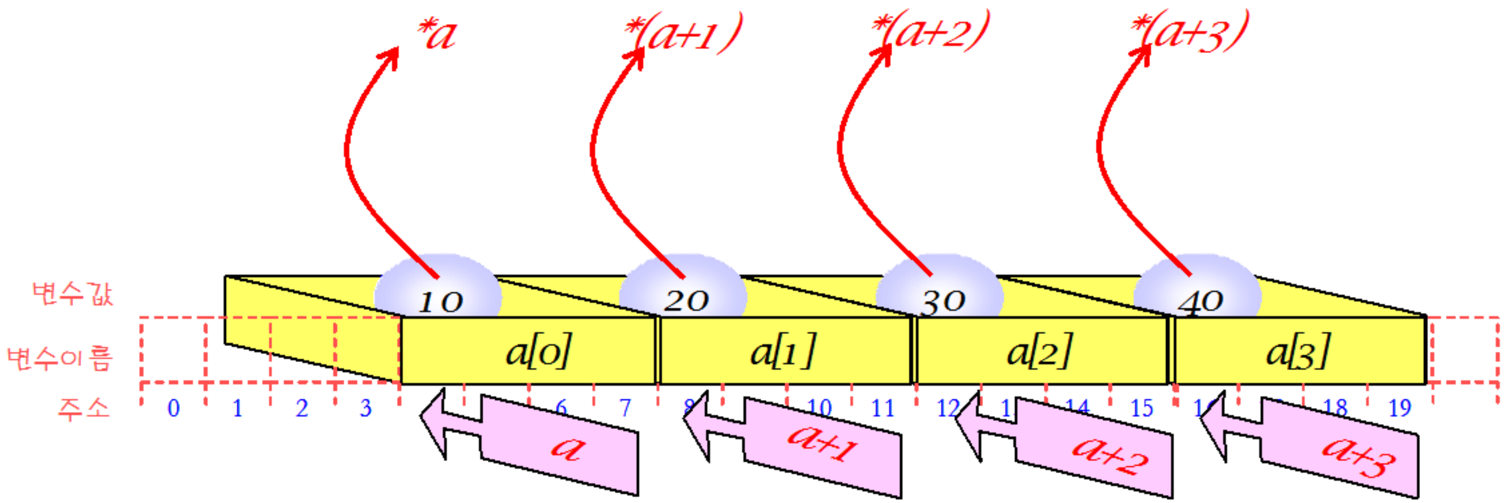
```c
// Relationship between pointers and arrays
#include < stdio.h >

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

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

	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
```

Use %p to print address.
Use %d to print integer.

Pointers and Arrays

- Pointers can be used like arrays .
- Index notation can be used with pointers .



Using pointers like arrays

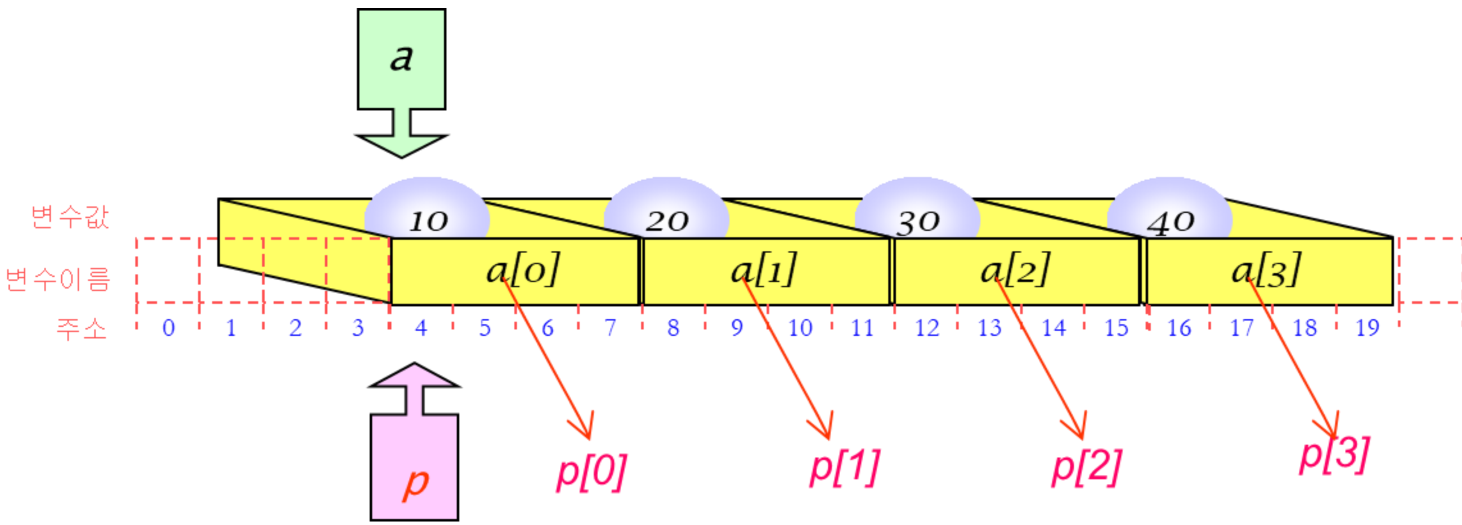
```c
#include < stdio.h >
int main( void )
{
	int a[] = { 10, 20, 30, 40, 50 };
	int *p;

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

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

	printf ( "a[0]=%da[1]=%da[2]=%d \n" , a[0], a[1], a[2]);
	printf ( "p[0]=%dp[1]=%dp[2]=%d \n" , p[0], p[1], p[2]);
	return 0;
}
```

You can see that arrays are ultimately implemented as pointers .

Array elements can be changed through pointers.

```
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
```

A pointer can also be used as an array name



Array parameters

- General parameters vs Array parameters

```
// Assign parameters variable x at
     memory place
void sub( int x)
{
…
}
```

```
// b does not have memory allocated to it
void sub( int b[] )
{
…
}
```

- Why? -> Copying an array to a function is time-consuming, so only pass the address of the array .

Array parameters

- Array parameters can be thought of as pointers .

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

        sub(a, 3);
}
```

The name of array is a pointer

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

You can change the original array through b.

```c
// Relationship between pointers and functions
#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;
}
```

The following two methods are completely equivalent :

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

-Array names and pointers are fundamentally the same.

-Accessing elements using array notation

```c
// pointer parameter
void sub(int *b, int size)
{
    *b = 4;
    *(b+1) = 5;
    *(b+2) = 6;
}
```

Accessing elements using pointer notation

Advantages of using pointers

When the compiler optimizes, the performance becomes almost similar

- Pointers are faster than index notation .
  - Why?: There is no need to convert index to address.

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

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

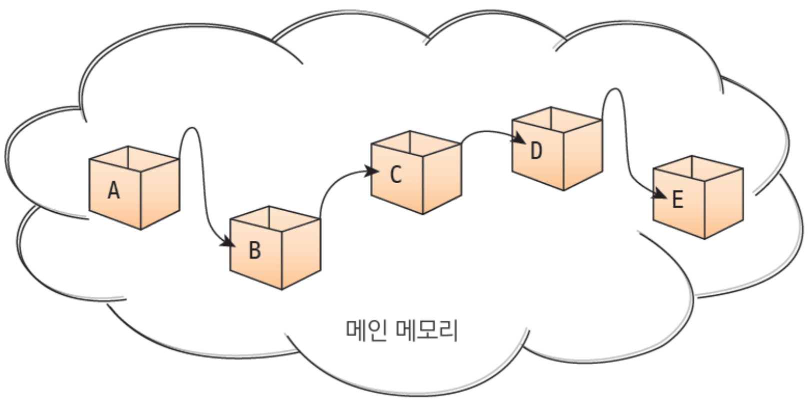
Using index notation

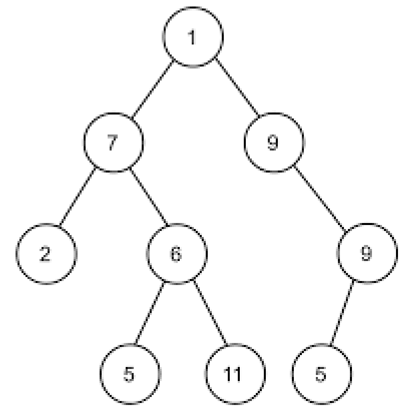
```
int get_sum2( int a[], int n)
{
        int i , sum =0 ;
        int *p;

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

Using pointers

Advantages of using pointers

- You can create advanced data structures such as linked lists and binary trees.





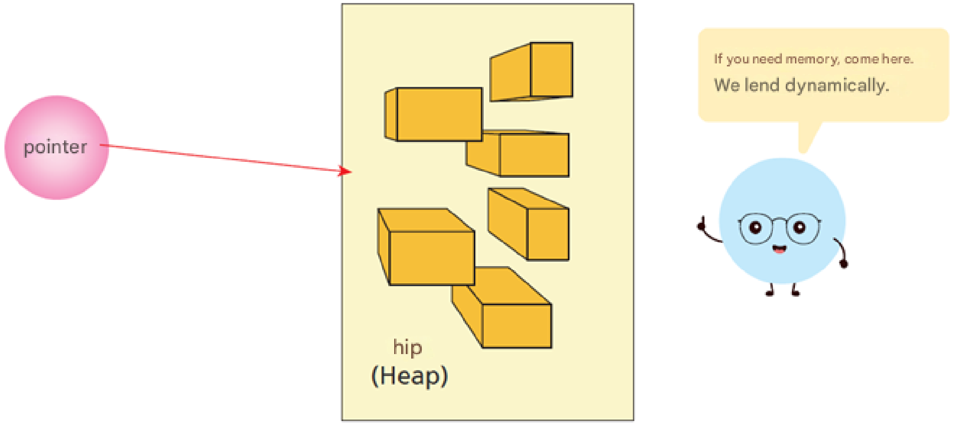
- Call by reference
  - You can change the value of a variable outside the function by using a pointer as a parameter.

Advantages of using pointers

- Memory mapping hardware
  - Memory-mapped hardware refers to hardware devices that can be accessed like memory .

```
volatile int * hw_address = (volatile int *)0x7FFF;
* hw_address = 0x0001; // Write value 0x0001 to the device at address 0x7FFF .
```

- Dynamic memory allocation
  - Covered in Chapter 17 .
  - To use dynamic memory, you must have a pointer .



Q & A