



# **C 언어 프로그래밍**

Part 03. C 언어 심화 학습

## **Chapter 10. 포인터 기초**

# 목차

1. 포인터란?
2. 포인터 사용
3. 포인터 연산자
4. void 포인터란?

[실전예제] memset 함수 내부 구현하기

# 학습목표

- 포인터의 필요성과 개념을 이해한다.
- 포인터 변수의 정의를 살펴본다.
- 포인터 연산자의 종류와 사용법을 살펴본다.
- void 포인터의 개념과 특징을 살펴본다.

01

포인터란?

# 01. 포인터란?

## I. 포인터의 개념

- 포인터

- 주소와 크기가 정해진 메모리의 특정 영역을 가리키고 해석하기 위해서 프로그래밍에 도입된 새로운 형태의 값
- 주소 : 메모리 영역의 시작 주소
- 타입 : 메모리 영역의 크기와 해석 방법

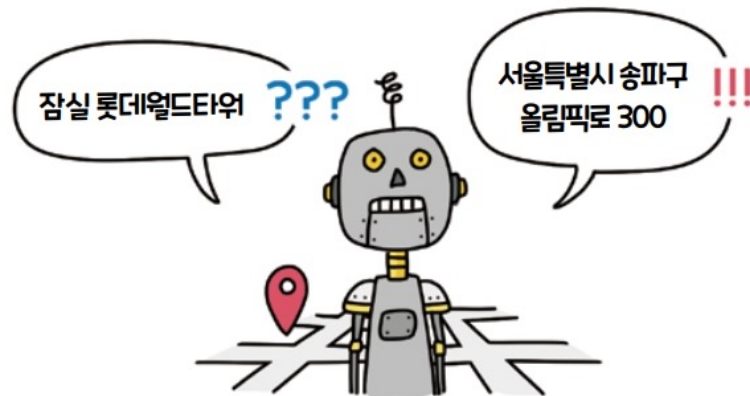


그림 10-1 건물명보다 주소가 더 편리한 경우

# 01. 포인터란?

## II. 포인터에 포함되는 정보

- 포인터를 이용한 예시

### [코드 10-1] 포인터를 이용한 메모리 영역 읽고 쓰기

```
01  int main( )  
02  {  
03      int a = 1;  
04      int* p = 0xABCD1234;  
05      *p += 1;  
06  }
```

# 01. 포인터란?

## 확인문제1

다음 빈칸에 들어갈 단어를 채우시오.

포인터가 포함하는 정보 두 가지는  와  으로서  는 메모리 영역의 위치를 나타내고,  은 메모리 영역의 크기와 해석 방법을 나타낸다.

02

포인터 사용



## 02. 포인터 사용

### I. 포인터 변수의 정의

- 포인터 변수를 정의하는 방법

`TYPE * Name;`  
타입    포인터의 이름

포인터가 가리키는 메모리 영역의 크기는 `sizeof(TYPE)`이 됩니다.

- TYPE : 포인터가 가리키는 대상의 타입으로, '대상 타입'이라고 부름
- 대상 주소 : 포인터가 가리키는 대상이 차지하는 메모리 영역의 시작 주소
- \* : 포인터 변수를 정의할 때 포인터임을 나타내는 기호  
TYPE과 Name 사이에 있어야 함

## 02. 포인터 사용

### I. 포인터 변수의 정의

- 포인터 변수를 정의하는 방법

하나 더 알기

포인터 타입(TYPE\*)

- 포인터 타입(대상타입이 아님)도 이름과 세미콜론만 생략해도 됨
- 일반적으로 TYPE에 \*를 붙인 TYPE\*이 포인터 타입

## 02. 포인터 사용

### I. 포인터 변수의 정의

- 포인터 변수를 정의하는 방법

[코드 10-2] 다양한 타입의 포인터 변수 정의

```
01 #include <stdio.h>
02
03 int main( )
04 {
05     char* pC;    // 대상 타입: char
06     int* pI;     // 대상 타입: int
07     double* pD;  // 대상 타입: double
08
09     printf("%d, %d, %d", sizeof(pC), sizeof(pI), sizeof(pD));
10 }
```

8, 8, 8

## 02. 포인터 사용

### I. 포인터 변수의 정의

- 포인터 변수를 정의하는 방법

하나 더 알기

포인터 변수를 정의할 때 주의할 점

#### [코드 10-2] 다양한 타입의 포인터 변수 정의

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int* p1, p2, *p3; → int* p1; int p2; int *p3;
06  }
```

- 포인터 변수와 int 변수를 구분하기 위해 콤마(,) 연산자가 적용되는 방식을 알아야 함

## 02. 포인터 사용

### II. 대상 타입과 객체 타입이 같은 포인터

[코드 10-4] int 변수를 가리키는 int\* 포인터

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int a
06
07      int* p = &a;
08      *p = 1;
09
10      printf("a: %d", a);
11  }
```

a: 1

## 02. 포인터 사용

### II. 대상 타입과 객체 타입이 같은 포인터

- 참조(&) 연산자

07	<code>int* p = &amp;a;</code>
----	-------------------------------

- &a는 변수 a가 나타내는 메모리 영역을 가리키는 포인터를 의미함
- &a는 변수 a가 나타내는 메모리 영역의 시작 주소가 대상 주소이고, 대상 타입은 int인 포인터
- 포인터 변수에 포인터를 저장할 때는 서로의 대상 타입이 일치해야 한다는 암묵적인 규칙이 있음

## 02. 포인터 사용

### II. 대상 타입과 객체 타입이 같은 포인터

- 간접(\*) 연산자

08	<code>*p = 1;</code>
----	----------------------

- 해당 메모리 영역을 대상 타입 변수처럼 사용할 수 있음

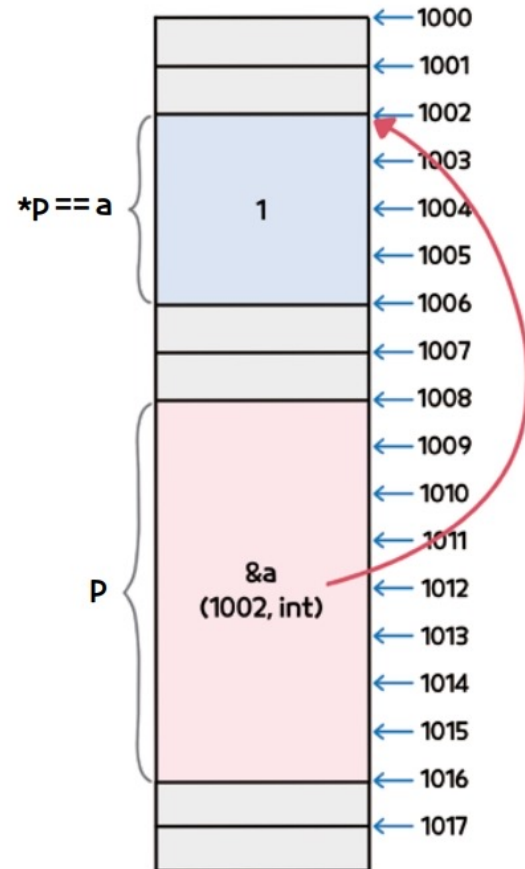
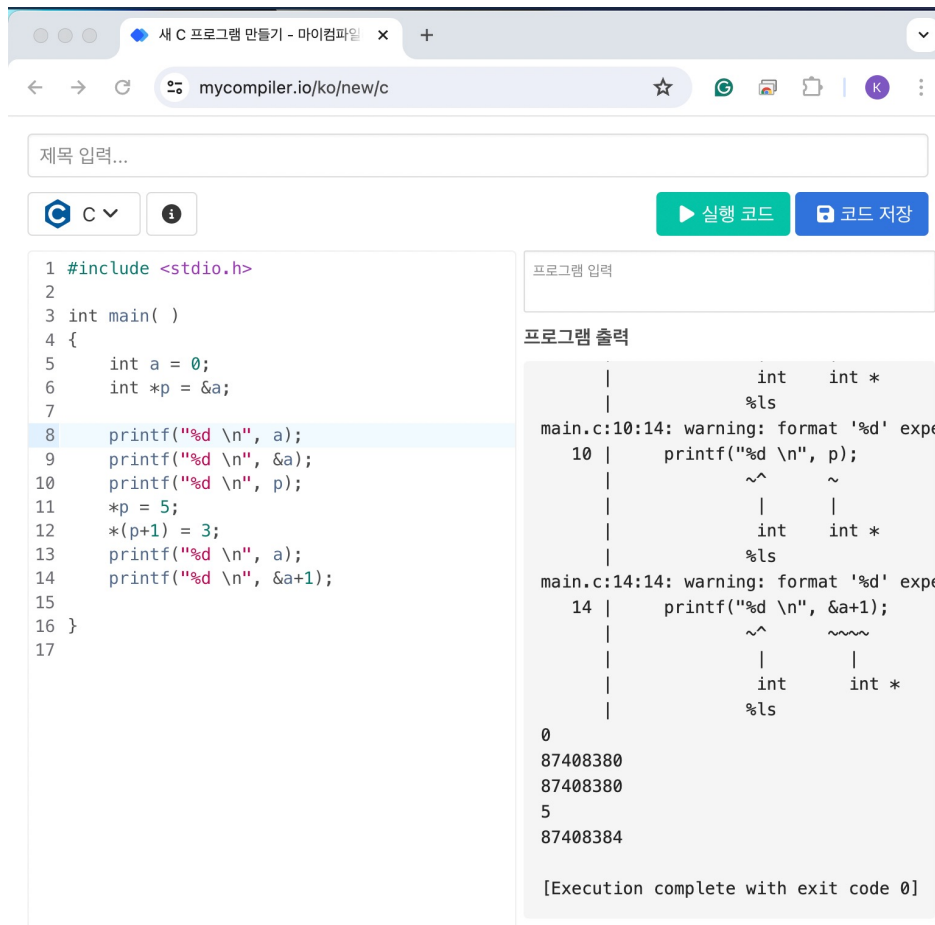


그림 10-2 int 변수 a를 가리키는 int\* 포인터 p

## 02. 포인터 사용

### II. 대상 타입과 객체 타입이 같은 포인터

- 간접(\*) 연산자



The screenshot shows a web browser with the URL `mycompiler.io/ko/new/c`. The code editor contains the following C code:

```
1 #include <stdio.h>
2
3 int main( )
4 {
5     int a = 0;
6     int *p = &a;
7
8     printf("%d \n", a);
9     printf("%d \n", &a);
10    printf("%d \n", p);
11    *p = 5;
12    *(p+1) = 3;
13    printf("%d \n", a);
14    printf("%d \n", &a+1);
15 }
16
17
```

The output section shows the following results:

```
0
87408380
87408380
5
87408384
```

Warnings are displayed for lines 10 and 14:

```
main.c:10:14: warning: format '%d' expects 'int' but 1 has type 'int*'
main.c:14:14: warning: format '%d' expects 'int' but 1 has type 'int*'
```

The execution status is "[Execution complete with exit code 0]".

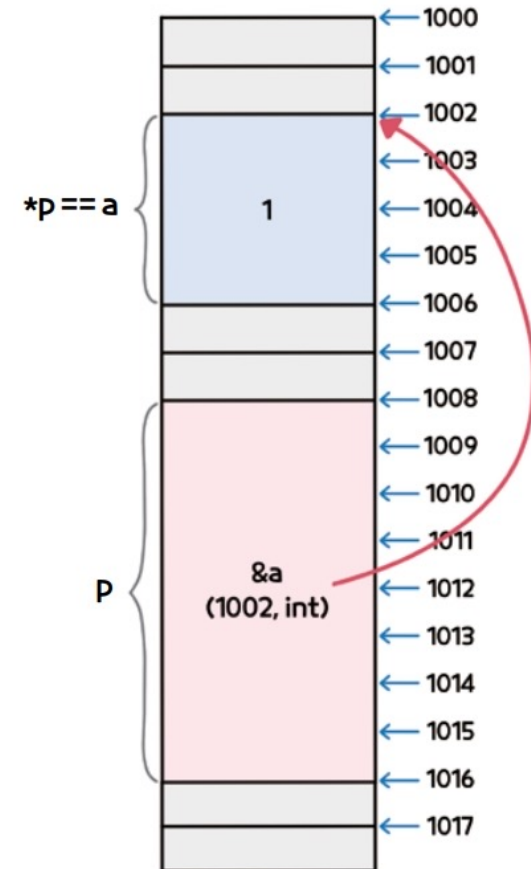


그림 10-2 int 변수 a를 가리키는 int\* 포인터 p



## 02. 포인터 사용

### III. 대상 타입과 객체 타입이 다른 포인터

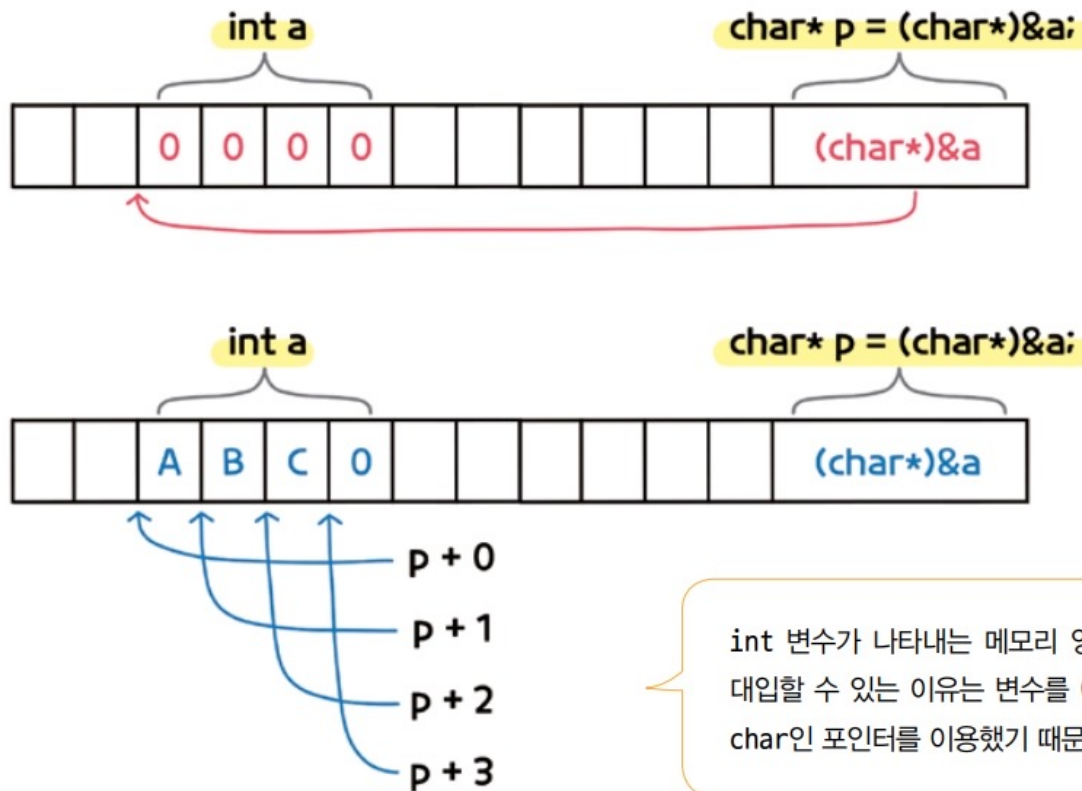
[코드 10-5] int 변수를 접근하는 char\* 포인터

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int a = 0;
06
07      //char* p = &a;           // 경고
08      char* p = (char*)&a;      // 문제 없음
09
10      *(p + 0) = 'A';
11      *(p + 1) = 'B';
12      *(p + 2) = 'C';
13      *(p + 3) = '\0';
14
15      printf("%s", &a);
16  }
```

ABC

## 02. 포인터 사용

### III. 대상 타입과 객체 타입이 다른 포인터



`int` 변수가 나타내는 메모리 영역에 1바이트씩 영문자를 대입할 수 있는 이유는 변수를 이용하지 않고 대상 타입이 `char`인 포인터를 이용했기 때문입니다.

그림 10-3 `int` 변수를 가리키는 `char*` 포인터

## 02. 포인터 사용

### 확인문제2

1. 대상 타입이 `float`인 포인터 변수 `pF`를 정의하시오.
2. `double` 변수 `d`가 있을 때 `&d`는 무엇인가?
3. 대상 타입이 `double`인 포인터 변수 `pD`가 `int` 변수 `a`를 가리키도록 정의하시오.

## 02. 포인터 사용

### LAB 10-1 int 변수를 -1로 설정하기

대상 타입이 char인 포인터를 이용하여 int 변수를 -1이 되도록 코드를 작성해봅시다.

int 변수 a는 4바이트인데, 각 1바이트를 가리키는 포인터 p+0, p+1, p+2, p+3에 대하여 간접(\*) 연산자를 사용하여 8비트를 모두 1로 채울 경우 변수 a는 -1이 된다.

## 02. 포인터 사용

### LAB 10-1

### 정답

```
01  #include <stdio.h>
02
03  int main()
04  {
05      int a = 0;
06      char* p = (char*)&a;
07
08      *(p + 0) = 0b11111111;
09      *(p + 1) = 0b11111111;
10      *(p + 2) = 0b11111111;
11      *(p + 3) = 0b11111111;
12
13      printf("%d", a);
14  }
```

03

포인터 연산자

# 03. 포인터 연산자

## I. 참조(&) 연산자

&a

→ a는 객체 또는 함수이다.

→ 연산 결과는 a가 나타내는 메모리 영역을 가리키는 포인터이다. 따라서 a가 객체인 경우 대상 주소는 객체의 주소이고 대상 타입은 객체의 타입이다.

- 피연산자가 객체(변수나 배열)일 경우 해당 객체를 가리키는 포인터를 도출함
- 주로 scanf를 호출할 때 콘솔로부터 입력 값을 받는 변수를 전달하기 위해 사용됨

```
int scanf(const char* format, ...);
```

# 03. 포인터 연산자

## II. 산술 연산자

- 주소 대입 연산

$p1 = p2$

→  $p1$ 의 대상 주소에  $p2$ 의 대상 주소를 설정한다.

→ 연산 결과는  $p1$ 이 된다.

- 포인터와 정수의 연산

$p + N, N + p, p - N$  (단,  $N - p$ 는 없다.)

→  $p$ 는 포인터이고,  $N$ 은 정수를 의미한다.

→ 연산 결과는  $p$ 의 대상 타입 크기의  $N$ 배수만큼 주소가 가감된 새로운 포인터가 된다.



# 03. 포인터 연산자

## II. 산술 연산자

- 포인터와 정수의 연산

[코드 10-6] 포인터와 정수의 덧셈

```
01 #include <stdio.h>
02
03 int main( )
04 {
05     char* pC = NULL;
06     printf("pC + 1 = %p\r\n", pC + 1);
07
08     int* pI = NULL;
09     printf("pI + 1 = %p\r\n", pI + 1);
10
11     double* pD = NULL;
12     printf("pD + 1 = %p", pD + 1);
13 }
```

```
pC + 1 = 0000000000000001
pI + 1 = 0000000000000004
pD + 1 = 0000000000000008
```

# 03. 포인터 연산자

## II. 산술 연산자

- 포인터와 정수의 연산

- $\text{int}^* \text{포인터} + 1$

- » 대상 타입인 `int`의 크기가 4바이트이므로 대상 주소에 4의 1배수인 4를 더함

- $\text{double}^* \text{포인터} + 1$

- » 대상 타입인 `double`의 크기가 8바이트이므로 대상 주소에 8의 1배수인 8을 더함

- $\text{char}^* \text{포인터} + 1$

- » 대상 타입인 `char`의 크기가 1바이트이므로 대상 주소를 1바이트 단위로 더함

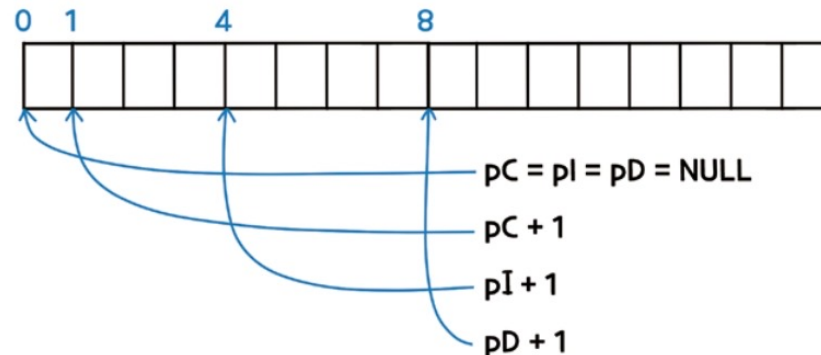


그림 10-4 포인터와 정수의 덧셈

# 03. 포인터 연산자

## II. 산술 연산자

- 포인터와 포인터의 연산

$p1 - p2$  (단,  $p1 + p2$ 는 없다.)

→  $p1, p2$ 는 포인터

→ 연산 결과는  $p1, p2$ 가 가리키는 대상 주소의 차를  $p1$ 의 대상 타입 크기로 나눈 값(정수)이 된다.

### [코드 10-7] 포인터와 포인터의 뺄셈

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int* p1 = 32;    // 대상 주소가 32
06      int* p2 = 4;     // 대상 주소가 4
07
08      printf("p1 - p2 = %d", p1 - p2);
09  }
```

$p1 - p2 = 7$

## 03. 포인터 연산자

### III. 증감 연산자

- 증감 연산자의 개념

- 포인터 변수의 대상 주소를 대상 타입 크기만큼 증감시키는 연산자
- 증감 연산자의 피연산자는 반드시 포인터 변수여야 함
- 증감 연산의 결과 피연산자인 포인터 변수의 대상 주소가 달라짐

`p++` → 연산 결과는 `p`이다. 연산 후 `p`의 대상 주소가 1배(대상 타입 크기) 증가한다.

`++p` → 연산 결과는 `p`의 대상 주소를 1배(대상 타입 크기) 증가한 후의 `p`이다.

`p--` → 연산 결과는 `p`이다. 연산 후 `p`의 대상 주소가 1배(대상 타입 크기) 감소한다.

`--p` → 연산 결과는 `p`의 대상 주소를 1배(대상 타입 크기) 감소한 후의 `p`이다.

(단, `p`는 포인터를 저장하는 포인터 변수이다.)

## 03. 포인터 연산자

### III. 증감 연산자

- 증감 연산자의 개념

#### [코드 10-8] 포인터의 증감 연산

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int* pI = NULL;
06      printf("++pI: %p\r\n", ++pI);    // pI는 4가 되고 4 도출
07      printf("--pI: %p\r\n", --pI);    // pI는 0이 되고 0 도출
08      printf("pI++: %p\r\n", pI++);    // 0 도출한 뒤 pI는 4
09      printf("pI--: %p", pI--);        // 4 도출한 뒤 pI는 0
10
11      int a;
12      //++(&a), --(&a), (&a)++, (&a)--;    // 오류 (&a는 포인터변수가 아님)
13  }
```

pI - a++pI: 0000000000000004  
--pI: 0000000000000000  
pI++: 0000000000000000  
pI--: 0000000000000004p2 = 7

## 03. 포인터 연산자

### IV. 간접 연산자

- 간접(\*) 연산자의 개념
  - 포인터를 통해서 간접적으로 메모리 영역을 객체처럼 나타냄

\*p

→ 연산 결과는 p가 가리키는 메모리 영역을 차지하는 대상 타입 객체가 된다.

# 03. 포인터 연산자

## IV. 간접 연산자

### [코드 10-9] 포인터의 간접(\*) 연산

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int q = 0;
06      int* p = &q;
07
08      *p = 1;
09      printf("q: %d, *p: %d", q, *p);
10  }
```

TYPE q;  
TYPE\* p = &q;  
\*p == q

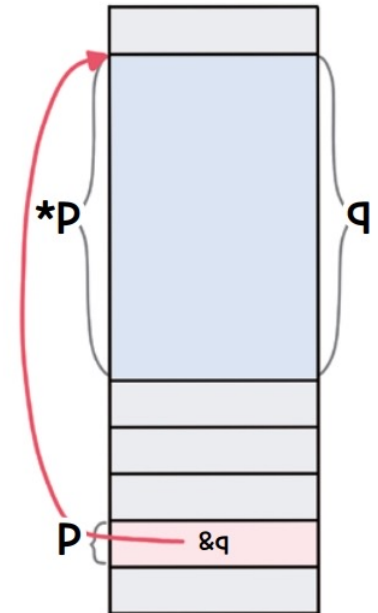


그림 10-5 간접(\*) 연산자

q: 1, \*p: 1

## 03. 포인터 연산자

### V. 간접 멤버 연산자

- 간접 멤버(->) 연산자의 사용
  - 구조체나 공용체의 객체를 가리킬 때 사용

`p->m`

→ `m`은 식별자이다.

→ 연산 결과는 `p`가 가리키는 구조체나 공용체 객체의 멤버 `m`이 된다.



## 03. 포인터 연산자

### VI. 첨자 연산자

- 첨자([]) 연산자의 사용
  - 배열에 주로 사용되지만 포인터에도 사용할 수 있음

`p[i]`

→ `p`는 포인터이다.

→ 연산 결과는 `*(p + i)`가 된다.

# 03. 포인터 연산자

## VI. 첨자 연산자

- 첨자([]) 연산자의 사용

### [코드 10-10] 포인터 첨자([]) 연산자

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int arr[3] = {1, 2, 3};
06      int* p = arr;
07
08      printf("%d, %d, %d", p[0], p[1], p[2]);
09  }
```

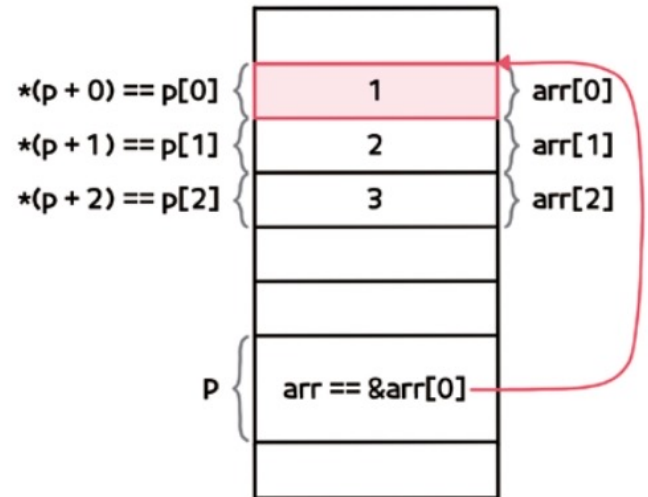


그림 10-6 첨자([]) 연산자

0, 1, 2

## 03. 포인터 연산자

### VII. 타입 변환 연산자

- 타입 변환 연산자의 사용
  - 대상 타입이 다른 포인터를 생성하기 위하여 사용됨

#### [코드 10-11] 타입 변환 연산자

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int*;
06
07      int s1 = sizeof(*(char*)p);
08      int s2 = sizeof(*p);
09
10      printf("s1: %d, s2: %d", s1, s2);
11  }
```

s1: 1, s2: 4

## 03. 포인터 연산자

### 확인문제3

1. 콘솔로부터 실수를 입력받아서 `double` 변수 `d`에 저장하는 `scanf`를 작성하시오.
2. `double* pD = 0x00001234;`일 때, 포인터 (`pD + 3`)의 대상 타입과 대상 주소는 무엇인가?  
(1) 대상 타입 : \_\_\_\_\_ (2) 대상 주소 : \_\_\_\_\_
3. 다음과 같이 선언되어 있을 때, 포인터 `pD++`의 대상 주소는 무엇인가?

```
double* pD = 0x00001234;
```

## 03. 포인터 연산자

4. 다음과 같이 선언되어 있을 때, d의 값은 무엇인가?

```
double d;  
double* p = &d;  
*p = 3.141592
```

5. 다음과 같이 선언되어 있을 때, p[2]는 무엇인가?

```
double arr[] = { 1.1, 2.2, 3.3 };  
double* p = arr;
```

6. `int *p;`일 때, p와 `(char*)p`의 차이를 설명하시오.

## 03. 전역 저장소 분류

### LAB 10-2

#### 배열의 모든 요소를 -1로 설정하기

`char* p`를 사용하여 `int arr[32];`의 모든 요소를 -1로 설정하는 코드를 작성해봅시다.

- ❶ `char* p`는 1바이트 단위로 배열 `arr`이 나타내는 메모리 영역에 접근할 수 있다.
- ❷ `*p`는 `p`가 가리키는 메모리 영역 1바이트를 나타내는 변수와 같다. 따라서 `*p = 0b11111111;`은 해당 영역의 모든 비트를 1로 설정한다.
- ❸ 반복문을 통해서 포인터 `p`의 대상 주소를 1바이트씩 증가시킨다.

## 03. 전역 저장소 분류

LAB 9-3

정답

```
01  int main()
02  {
03      int arr[32];
04      char* p = (char*)arr;
05
06      for(int i = 0; i < sizeof(arr); i++) // 128번수행 32*4
07      {
08          *p = 0b11111111;
09          p++;
10      }
11  }
```

04

void 포인터란?



## 04. void 포인터란?

### I. void 포인터의 개념

- void 포인터
  - 대상 타입이 없는, 따라서 대상 타입이 void인 포인터
  - void는 크기가 없으므로 void 변수를 정의할 수 없음
  - 보통 메모리의 주소만을 나타내기 위하여 사용됨

## 04. void 포인터란?

### II. void 포인터의 특징

- 범용성

- void 포인터가 변수로 사용될 경우
  - » 어떤 타입의 포인터라도 컴파일러 경고 없이 받을 수 있음
- void 포인터가 값으로 사용될 경우
  - » 컴파일러 경고 없이 어떤 포인터 타입으로도 암묵적으로 변환될 수 있음

## 04. void 포인터란?

### II. void 포인터의 특징

[코드 10-12] void\*

```
01  int main( )
02  {
03      int i;
04
05      int* pI = &i;
06      double* pD = &i;    // 경고
07      void* pV = &i;
08
09      pI = pD;            // 경고
10      pI = pV;
11  }
```

## 04. void 포인터란?

### II. void 포인터의 특징

- 함수의 인자로 사용되는 void 포인터

```
void* memset(void *dest, int c, size_t count);
```

- memset 함수 : 이미 정의된 배열의 모든 요소를 0으로 만들 때 사용됨
- dest : 대상 주소에서 count 바이트 크기의 메모리 영역에 1바이트씩 값 c를 채우는 함수이자, void 포인터인 매개변수

# 04. void 포인터란?

## II. void 포인터의 특징

- 함수의 인자로 사용되는 void 포인터

### [코드 10-13] memset 함수

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int arr[ ] = { 1, 2, 3 };
06
07      memset(arr, 0, sizeof(arr)); // arr, 0, 12
08
09      for(int i = 0; i < 3; i++)
10      {
11          printf("a[%d] = %d\r\n", i, arr[i]);
12      }
13  }
```

```
a[0] = 0
a[1] = 0
a[2] = 0
```

## 04. void 포인터란?

### III. void 포인터의 한계

- 대상 주소가 있다 해도 어떤 메모리 영역도 변수처럼 사용할 수가 없음

#### [코드 10-14] 간접(\*) 연산자의 피연산자 void 포인터

```
01  int main( )
02  {
03      int i = 0;
04      void* p = &i;
05
06      /*p = 1; ← 오류
07      *(int*)p = 1;    // 문제 없음
08  }
```

## 04. void 포인터란?

### III. void 포인터의 한계

- 다른 포인터 연산을 할 수 없음

[코드 10-15] 산술, 증감, 첨자 연산과 void 포인터

```
01  int main( )
02  {
03      int i = 0;
04      void* p = &i;
05
06      /*p = 1; ← 오류
07      *(int*)p = 1;    // 문제 없음
08
09      //p++; ← 오류
10      ((int*)p)++;    // 문제 없음
11
12      //p[0]; ← 오류
13      ((int*)p)[0];    // 문제 없음
14  }
```

## 04. void 포인터란?

### 확인문제4

1. 다음 코드를 실행하면 배열 `arr`의 모든 요소가 `-1`로 설정된다. 그 이유를 설명하시오.

```
int arr[4];  
memset(arr, -1, sizeof(arr));
```

2. `void* p`에 대하여 다음 표현식 중 올바른 것을 고르시오.

① `*p`

② `p++`

③ `p + 1`

④ `p = NULL`



## 04. void 포인터란?

### LAB 10-3

### 두 변수의 주소와 주소 차이 출력하기

두 변수의 주소와 주소 차이를 출력하는 함수(`VariableAddress`)를 작성해봅시다. 두 변수의 타입 제한은 없으며, `VariableAddress` 함수는 두 변수의 포인터를 입력받습니다.

모든 타입의 변수를 받기 위하여 함수 매개변수 타입은 `void*`가 되어야 한다. 또한 `void*`는 산술 연산을 할 수 없으므로 (`char*`)로 강제 타입 변환이 필요하다.

## 04. void 포인터란?

LAB 10-3

정답

```
01 void VariableAddress(void* p1, void* p2)
02 {
03     printf("%p, %p, %d", p1, p2, (char*)p1 - (char*)p2);
04 }
```

# [실전예제]

memset 함수 내부 구현하기

# [실전예제] memset 함수 내부 구현하기

## [문제]

[코드 10-13]에서 살펴본 memset 함수를 mymemset이란 함수로 내부를 구현해봅시다. 단, 반환 값은 dest 그대로입니다.

```
void* memset(void *dest, int c, size_t count);
```

## [해결]

1. 1바이트 단위로 메모리 영역에 값을 써야 하기 때문에 대상 타입이 char인 포인터 p를 dest로 초기화한다.
2. 메모리 영역 크기만큼 반복문으로 p[i]에 값 c를 대입한다.

# [실전예제] memset 함수 내부 구현하기

## [해결]

```
01 void* mymemset(void* dest, int c, size_t count)
02 {
03     char* p = dest;
04     for(int i = 0; i < count; i++)
05         p[i] = c;
06
07     return p;
08 }
```

# Thank you!