
C 프로그래밍 및 실습

13. 연산자/함수/자료형 심화

세종대학교

목차

- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) main() 함수의 인자 (심화 내용)
- 5) const 키워드 (심화 내용)
- 6) 배열 포인터와 다차원 배열 (심화 내용)
- 7) void 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

1) 비트연산자

■ 비트 연산이란?

- 비트 단위로 처리되는 연산
- 비트 1은 참, 비트 0은 거짓을 의미
- 예) 두 이진수의 비트 단위 논리곱(bitwise AND)
 - ✓ 해당 자리의 비트가 모두 1인 경우에만 참

	0010	1010
AND	1010	1101

(결과)	0010	1000

- 비트 연산의 종류
 - ✓ 비트 단위 논리 연산
 - ✓ 비트 단위 이동 연산

1) 비트연산자

- 비트 단위 논리 연산

- 예) 비트 AND 연산자 : $x \ \& \ y$
 - ✓ 두 정수 x 와 y 의 비트단위 논리곱
- 비트 연산을 할 때는 16진수로 표현하는 것이 이해하기 쉬움

```
unsigned int x, y, z;
```

```
x = 0X2A;          // x = 0000 0000 ... 0010 1010
```

```
y = 0XAD;          // y = 0000 0000 ... 1010 1101
```

```
z = x & y;          // z = 0000 0000 ... 0010 1000
```

```
printf("%#X", z);   // #: 16진수임을 나타내는  
                    // '0X'를 앞에 출력
```

결과:

0X28 ➔ 16진수 28을 의미

1) 비트연산자

■ 비트 단위 이동 연산

- 예) 왼쪽 이동 연산자 : $x \ll k$
 - ✓ x를 비트 단위로 왼쪽으로 k 만큼 이동
 - ✓ 오른쪽 빈 자리는 k개의 0으로 채움

```
unsigned int x, z;
```

```
x = 0X2A01234C;           // x = 0010 1010 ... 0000 1100
```

```
z = x << 4;               // z = 1010 0000 ... 1100 0000  
                           (왼쪽으로 4칸 이동)
```

```
printf("%#X", z);
```

결과:

```
0XA01234C0
```

1) 비트연산자

■ C언어 비트 연산자

- 피 연산자는 정수형(char, int, long 등) 에 대해서만 연산 가능
- 특별한 이유가 없으면 unsigned 정수 사용
 - ✓ 이동 연산에서 빈 자리는 0으로 채우는 것이 기본이지만, signed 정수는 1로 채워지는 경우도 있음

연산자	연산자 기능	예시 (8비트 수라고 가정)
$x \& y$	x와 y의 비트 단위 <u>AND</u> 연산	0110 1100 & 0100 1010 → 0100 1000
$x y$	x와 y의 비트 단위 <u>OR</u> 연산	0110 1100 0100 1010 → 0110 1110
$x \wedge y$	x와 y의 비트 단위 <u>XOR</u> 연산	0110 1100 ^ 0100 1010 → 0010 0110
$\sim x$	x에 대한 비트 단위 <u>NOT</u> 연산	~ 0110 1100 → 1001 0011
$x \ll n$	x를 n 비트 <u>왼쪽 이동</u>	0100 1010 << 2 → 0010 1000
$x \gg n$	x를 n 비트 <u>오른쪽 이동</u>	0100 1010 >> 2 → 0001 0010

1) 비트연산자

- [예제 13.1] 다음을 작성하여 앞 슬라이드의 예시를 확인해보자.

- ① **unsigned char** 형 변수 3개 선언 (8비트 수 표현)
- ② 두 개의 변수에 **0110 1100**과 **0100 1010**을 16진수로 표현하여 대입
- ③ 각 논리 연산에 대해, 연산 결과를 나머지 하나의 변수에 대입하고 16진수로 출력

1) 비트연산자

- [예제 13.2] 어떤 정수 N을 이진수로 표기했을 때, 오른쪽에서 10번째 자리의 비트 값을 출력 (비트 연산의 응용)
 - ✓ 가장 오른쪽 비트를 0번째 비트라고 가정

```
unsigned int x = 0X71234567; // ... 0100 0101 0110 0111

x = x >> 10;           // 오른쪽에서 10번째 비트를
                        // 가장 오른쪽에 위치시킴
x = x & 1;              // x와 ... 0001 의 AND
                        // 두 문장을 하나로 합치면 (x >> 10) & 1

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

결과:

1 ➔ 0000 ... 0001

목차

- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) `main()` 함수의 인자 (심화 내용)
- 5) `const` 키워드 (심화 내용)
- 6) 배열 포인터와 다차원 배열 (심화 내용)
- 7) `void` 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

2) 재귀함수

- 재귀 함수란?

- 다음 프로그램은 정상적인 프로그램일까?
 - ✓ `dec()` 함수에서 자기를 다시 호출한다??
 - ✓ 그럼 현재 수행 중이던 `dec()` 함수는 어떻게 되는 거지??
 - ✓ 놀랍게도(?) 정상적으로 컴파일도 되고 실행도 된다.

```
void dec(int x) {  
    printf("x: %d\n", x);  
    if( x > 1)  
        dec(x-1); // 자기 자신 호출(?)  
}  
  
int main() {  
    dec(3);  
    return 0;  
}
```

출력 결과

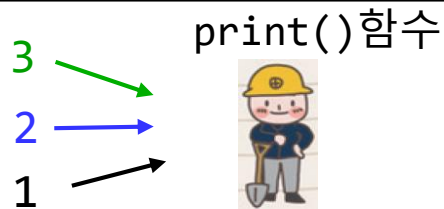
```
x: 3  
x: 2  
x: 1
```

2) 재귀함수

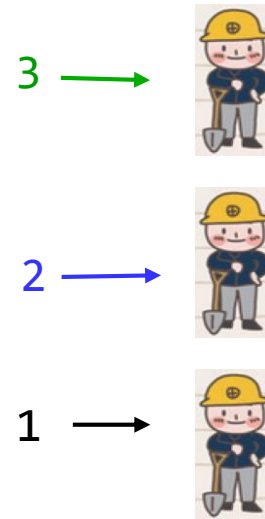
- 함수 호출 과정 들여다보기

- ✓ 아래 두 그림 중 어느 것이 `print()` 함수의 호출 과정을 더 정확하게 표현한 것일까?

```
void print(int x){  
    printf("x: %d\n", x);  
}  
void main(){  
    print(3);  
    print(2);  
    print(1);  
}
```



1명의 일꾼(함수)이
3건의 요청을 처리??



동일한 기능을 가진
3명의 일꾼(함수)이
각각 1건의 요청을 처리

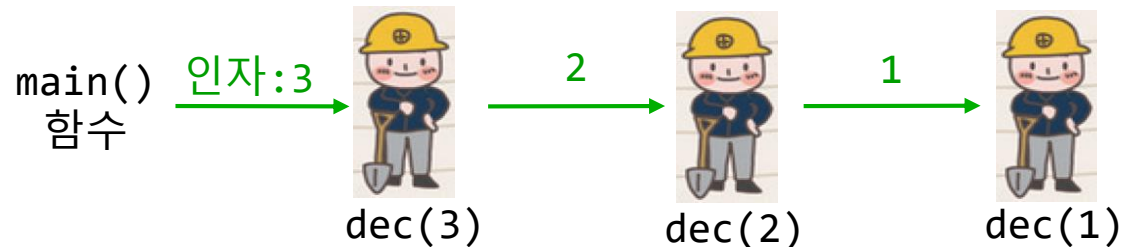
2) 재귀함수

- 이전 프로그램의 함수 호출 과정을 그림으로 표현하면?

```
void dec(int x)
{
    printf("x: %d\n", x);
    if( x > 1) dec(x-1);
}
void main()
{
    dec(3);
}
```



일꾼(함수) dec가
자신과 동일한 기능을 가진
다른 일꾼 dec에게 요청



2) 재귀함수

- 재귀함수

- 함수 내부에서 자기와 동일한 (이름의) 함수를 호출하는 함수
- 유사 개념: 점화식
 - ✓ $A_n = A_{n-1} + 2$ (등차 수열의 점화식)
 - ✓ A : 함수 이름에 해당
 - ✓ 첨자 n : 함수 인자에 해당

2) 재귀함수

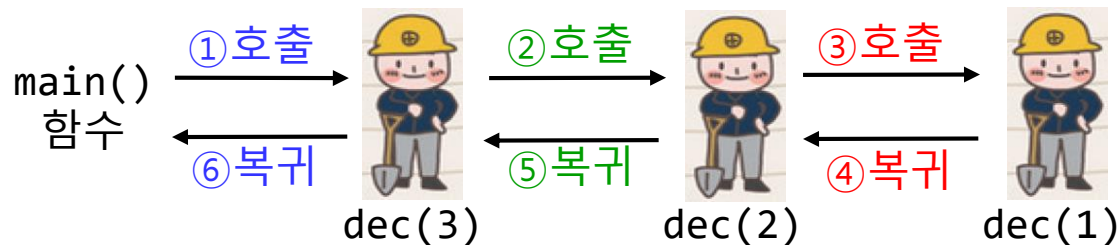
재귀 함수 동작 과정

- 동작 과정 파악을 위해 함수의 시작과 끝에 출력문 삽입

출력 결과

```
void dec(int x) {  
    printf("+start: x=%d\n",x);  
    if( x > 1) dec(x-1);  
    printf("-end: x=%d\n",x);  
}  
void main() {  
    dec(3);  
    return 0;  
}
```

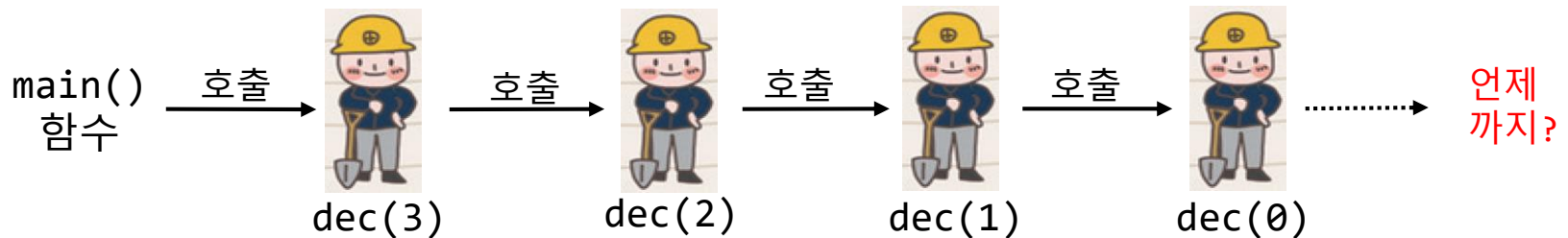
```
+start: x=3  
+start: x=2  
+start: x=1  
-end: x=1  
-end: x=2  
-end: x=3
```



2) 재귀함수

- 주의사항: 재귀 호출의 종료 조건이 없으면 제대로 동작하지 않음
 - 점화식에서 초기항이 없으면 정의되지 않는 것처럼
 - ✓ $A_n = A_{n-1} + 2, A_1 = 1$

```
void dec(int x){  
    printf("x: %d\n", x);  
    if(x > 1) dec(x-1); // 없으면 런타임 오류 발생  
}  
int main(){  
    dec(3);  
    return 0;  
}
```



2) 재귀함수

- [예제 13.3] 재귀함수를 사용하여 1부터 n까지의 합 계산
 - $\text{sum}(n) = 1+2+3+ \dots + (n-1) + n$ 을 점화식으로 표현하면?
 - ✓ $\text{sum}(n) = \text{sum}(n-1) + n$ ($n > 1$ 인 경우)
 - ✓ $\text{sum}(1) = 1;$ ($n==1$ 인 경우)

```
int sum(int n) {  
    int s;        // 합 저장  
    if( n == 1)   s = 1;  
    else         s = sum(n-1) + n;  
    return s;  
}  
int main() {  
    printf("%d", sum(10) );  
    return 0;  
}
```

```
// simple version  
  
int sum(int n) {  
    if( n == 1)   return 1;  
    return sum(n-1) + n;  
}  
void main() {  
    printf("%d", sum(10) );  
    return 0;  
}
```


2) 재귀함수

- [예제 13.4] 재귀함수를 이용하여 $n!$ 을 계산하는 프로그램을 작성하라.
 - $n! = n * (n-1) * (n-2) * \dots * 2 * 1$
 - 점화식으로 표현하면
 - ✓ $n! = n * (n-1)!$ ($n > 1$ 인 경우)
 - ✓ $1! = 1;$ ($n=1$ 인 경우)

목차

- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) `main()` 함수의 인자 (심화 내용)
- 5) `const` 키워드 (심화 내용)
- 6) 배열 포인터와 다차원 배열 (심화 내용)
- 7) `void` 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

3) 라이브러리 활용

- **난수(random number) 생성**
 - 난수란? 정의된 범위 내에서 임의로 추출되는 수
 - C언어에서는 난수를 생성하는 함수 제공
 - 관련 함수: rand() , srand() , time ()
- **실행 시간 측정**
 - C언어에서는 제공되는 시간 관련 함수를 사용하여 프로그램이 실행되는데 필요한 시간을 측정할 수 있음
 - 관련 함수: clock()

3) 라이브러리 활용

- 난수(random number) 생성
 - `rand()` 함수: `<stdlib.h>`에 선언되어 있음
 - ✓ 0~`RAND_MAX` 사이의 임의의 수 리턴
 - ✓ `RAND_MAX`는 `stdlib.h`에 정의된 상수 (`32767`)
 - `srand()` 함수: `<stdlib.h>`에 선언되어 있음
 - ✓ `rand()` 함수의 시드(seed) 변경
 - `time()` 함수: `<time.h>`에 선언되어 있음
 - ✓ 현재 시스템의 시간에 의해 결정되는 정수 리턴
 - ✓ 실행할 때 마다 시드를 바꾸기 위해서 사용

3) 라이브러리 활용

- 난수 생성 예제

- ✓ 0~ 32767 사이의 난수가 5개 출력됨

```
#include <stdio.h>
#include <stdlib.h>    // rand 함수 사용을 위해
#include <time.h>      // time 함수 사용을 위해

void main() {
    int i;
    srand( time(NULL) );    // 시드를 현재시간으로 지정
    for( i=0 ; i < 5 ; ++i)    // 5개의 난수 생성
        printf(" %d", rand() );
    return 0;
}
```

결과: 실행할 때 마다 달라짐

- ✓ srand() 함수를 빼고, 여러 번 수행시켜보자.
 - ✓ srand() 함수의 인자에 특정 정수를 넣어, 여러 번 수행시켜보자

3) 라이브러리 활용

- **[예제 13.5] 0부터 100사이의 난수를 5개 출력하는 프로그램 작성**
 - 실행할 때 마다 다른 값이 나와야 함
 - (hint) 난수로 생성된 값을 0~100 사이로 바꾸기 위해 나머지 연산자(%) 활용 (간단 버전)

3) 라이브러리 활용

- [min, max) 사이의 난수 하나 생성하기

✓ % 연산자를 사용하는 방식 보다 아래 방식 권장

```
int random_num(int min, int max) {  
    int rand_num;  
    srand( time(NULL) );  
    rand_num =  
        rand() / ((double)RAND_MAX + 1) * (max-min) + min;  
    return rand_num;  
}
```

- ✓ rand() : 0~RAND_MAX 사이의 임의의 정수 생성
- ✓ / ((double)RAND_MAX + 1) : 생성된 정수를 [0,1) 사이의 소수로 변환
- ✓ * (max-min) : [0,1) 사이의 소수를 [0,max-min) 사이의 값으로 변환
- ✓ + min : 수를 min 만큼 이동시켜, [min,max) 사이의 수로 변환

3) 라이브러리 활용

- 실행 시간 측정
 - `clock()` 함수 사용: `<time.h>`에 선언되어 있음
 - ✓ 호출 당시의 시스템 시각을 `CLOCKS_PER_SEC` (`time.h`에 정의된 상수) 단위로 반환
 - ✓ 초 단위의 시간을 얻기 위해서는 `clock()` 함수에 의해 측정된 시각을 `CLOCKS_PER_SEC`로 나누어야 함

3) 라이브러리 활용

- 실행 시간 측정 코드

```
#include <stdio.h>
#include <time.h>          // clock() 함수 사용을 위해

int main( void ) {
    clock_t start, finish;
    double duration;

    start = clock();        // 시작 시각
    ...                    // 수행시간을 측정하고 하는 코드
    finish = clock();       // 종료 시각

    duration = (double)(finish-start) / CLOCKS_PER_SEC;

    printf("실행 시간: %1f 초\n", duration);
    return;
}
```

목차

- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) **main() 함수의 인자 (심화 내용)**
- 5) `const` 키워드 (심화 내용)
- 6) 배열 포인터와 다차원 배열 (심화 내용)
- 7) `void` 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

4) main() 함수의 인자 (심화 내용)

- OS와 프로그램 사이의 정보 교환
 - main() 함수의 인자와 반환 값
 - 다만, 교환 형식이 정해져 있음
 - ✓ 반환: 정수 반환이 기본 형식 (main 함수에서 `return 0;`)
 - ✓ 인자는?
 - OS가 프로그램에 정보를 전달하는 방법
 - ✓ 프로그램(실행파일) 이름 뒤에 연달아 입력
- c:\W> show aaa bbb
- 3개의 문자열 "show" "aaa" "bbb"가 프로그램 show에 전달

4) main() 함수의 인자 (심화 내용)

▪ main() 함수의 인자

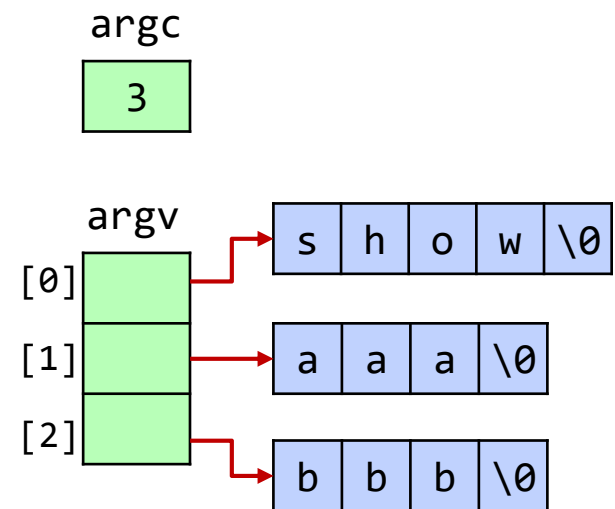
- 문자열 포인터 배열 형식으로 저장되어 전달
 - ✓ int argc : 전달된 문자열의 개수
 - ✓ char *argv[] : 전달된 문자열을 가리키는 포인터 배열 (10.4절)

```
/* 프로그램에 전달된 인자 출력 */
#include <stdio.h>

int main( int argc, char *argv[] ) {
    int i;

    for( i = 0; i < argc; ++i )
        printf("%d: %s\n", i, argv[i] );

    return 0;
}
```



목차

- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) main() 함수의 인자 (심화 내용)
- 5) **const 키워드 (심화 내용)**
- 6) 배열 포인터와 다차원 배열 (심화 내용)
- 7) void 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

3) const 상수 (심화 내용)

- **const 키워드: 변수의 상수화**
 - const 키워드가 붙은 변수는 도중에 값 대입 불가능
➔ 반드시 선언과 동시에 초기화해야 함!

```
const double pi = 3.14; // 변수의 상수화 (초기화 필수)
double r = 2.5;

printf("반지름이 %.1f인 원의 둘레는 %.2f이다.\n",
      r, 2*pi*r );
```

- 함수의 인자에도 자주 사용
 - ✓ char *strcpy(char *dest, const char *src);
 - 변경 가능성 있음
 - 변경되지 않음을 보장

3) const 상수 (심화 내용)

- 포인터 변수에 사용된 const

- const의 위치에 따라 의미가 달라짐
 - ✓ 선언 형태와 관련
- *p 를 상수화

```
int a = 10, b;
```

```
const int *p = &a ;    // *p 를 상수화
```

```
*p = 20;    // 컴파일 오류: *p 에 대입 불가능
```

```
a = 20;     // 정상: a 자체를 상수화 한 건 아니므로
```

```
p = &b;     // 정상: p 자체를 상수화 한 건 아니므로
```

*p 가 const int형 :
즉, *p (p가 가리키는 것)이
정수형이고
상수화된다는 의미

- » p를 통한 간접 참조로 값을 변경할 수 없다는 의미이지
- » p가 가리키는 변수 a 자체를 상수화 하라는 의미는 아님

3) const 상수 (심화 내용)

- p를 상수화

```
int * const p = &a ;    // 변수 p를 상수화

*p = 20;    // 정상: *p를 상수화 한 건 아니므로
p = &b;      // 컴파일 오류: 변수 p에 대입 불가능
```

- *p와 p 모두 상수화

```
const int * const p = &a ;    // p와 *p 모두 상수화

*p = 20;    // 컴파일 오류
p = &b;      // 컴파일 오류
```

- char *strcpy(char *dest, const char *src);
✓ src가 가리키는 문자열이 변경되지 않음을 보장

목차

- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) main() 함수의 인자 (심화 내용)
- 5) const 키워드 (심화 내용)
- 6) 배열 포인터와 다차원 배열 (심화 내용)**
- 7) void 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

5) 배열 포인터와 다차원 배열 (심화 내용)

▪ 개요

- `int a[5]`에서 `a`의 자료형은 `int *`, 그럼 `&a`의 자료형은?
- `int b[3][5]`에서 `b`의 자료형은?
- 아래 함수 인자의 자료형은?

```
void init(int x[][5]){ // 2차원 배열에 대한 인자
    . . .              (교재 8.5절에서 학습)
}
```

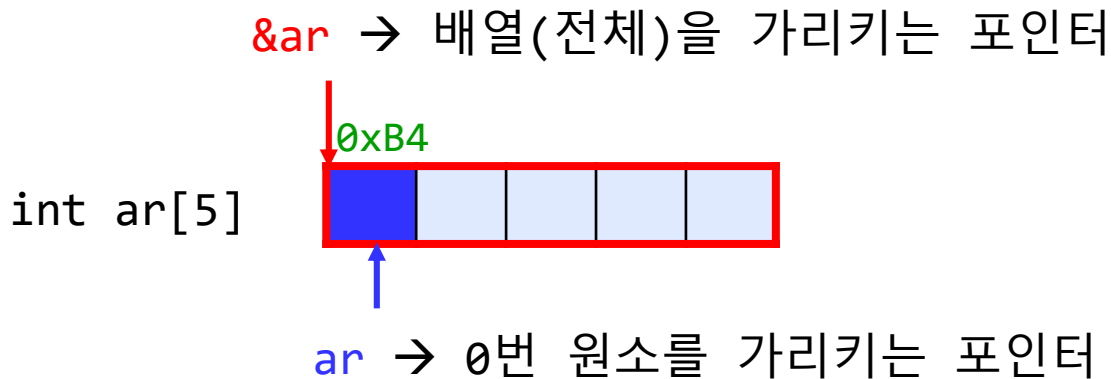
→ 답: `int (*)[5]` (??)

▪ 이 절에서 배울 내용

- 저 모양도 이상한 `int (*)[5]`는 뭔가요?
- 함수 인자의 `int x[][5]`의 정체는 뭔가요?
- 왜 첫 번째 크기는 비어 있고, 두 번째 크기인 `[5]`은 명시할까요?

5) 배열 포인터와 다차원 배열 (심화 내용)

- 일차원 int 배열 `ar[5]`의 이름에 대한 고찰
 - 배열의 이름 `ar == &ar[0]` (0번 원소의 시작 주소)
 - ✓ `ar[0]`이 int형이므로, `ar`은 int를 가리키는 포인터 '**int ***'
- 그럼, 배열 변수의 주소 **&ar** 은?
 - **&ar** 은 '배열 전체'의 시작 주소를 의미
 - ✓ **&ar**의 값은? `ar`과 동일 (아래 그림에서 **0xB4**)
 - ✓ **&ar**의 자료형은? "**크기가 5인 int 배열**"을 가리키는 포인터



5) 배열 포인터와 다차원 배열 (심화 내용)

- 배열 포인터: 배열을 가리키는 포인터

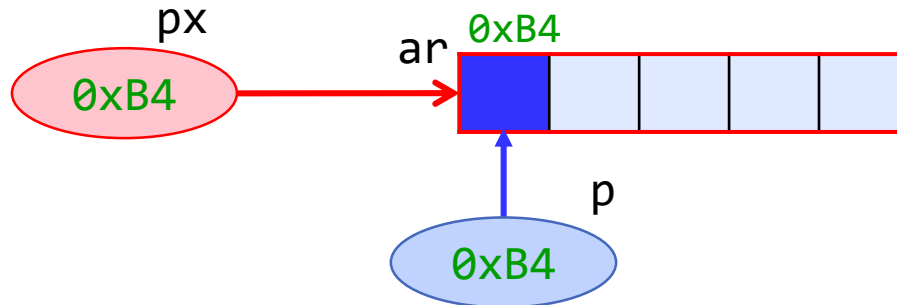
- 자료형 표기

- ✓ "크기가 5인 int 배열(int [5])"을 가리키는 포인터(*)

- ✓ **int (*) [5]**

```
int ar[5]={2, 3, 5, 7, -1}, *p;  
int (*px)[5]; // int [5]에 대한 포인터 선언
```

```
p = ar;    // 자료형 일치: int형에 대한 포인터  
px = &ar;  // 자료형 일치: int [5]형에 대한 포인터
```



5) 배열 포인터와 다차원 배열 (심화 내용)

- 주의!! 배열 포인터에서 소괄호를 빼면 다른 자료형을 의미
 - `int (*px)[5]`
 - ✓ `px`는 포인터(`*`), 가리키는 대상은 '`int [5]`' 형
 - ✓ 즉, 배열을 가리키는 포인터 (배열 포인터)
 - `int *par[5]`
 - ✓ `int *(par[5])` 와 동일 \leftarrow `[]` 의 우선 순위가 `*` 보다 높음
 - ✓ `par`는 크기가 5인 배열, 원소는 '`int *`' 형
 - ✓ 즉, 포인터들의 배열 (포인터 배열)
- 변수 이름이 **무엇과 짝을 이루냐**에 따라 의미하는 자료형이 달라짐(연산자의 의미와 적용 순서를 통해 이해하자)

5) 배열 포인터와 다차원 배열 (심화 내용)

- 포인터 값을 증가시켜보자. 얼마씩 커지는가?
 - 포인터를 1 증가시키면
→ '포인터가 가리키는 자료형'의 크기만큼 증가

```
int ar[5]={2, 3, 5, 7, -1}, *p = ar;  
int (*px)[5] = &ar;  
  
printf("%p %p %p\n", p, p+1, p+2);    // 'int *'의 증가량  
printf("%p %p %p\n", px, px+1, px+2); // 'int (*)[5]'의 증가량
```

실행 결과(예시)

001E40B4	001E40B8	001E40BC	→ 4씩 증가
001E40B4	001E40C8	001E40DC	→ 20씩 증가 (16진수임에 유의)


5) 배열 포인터와 다차원 배열 (심화 내용)


- 다차원 배열에 대한 이해
 - 1차원 배열 이름의 자료형은 (이미 학습)
 - ✓ `int a[5];`
 - ✓ 배열 이름 `a`의 자료형은 `int`형을 가리키는 **포인터** 즉, `int *`
 - 그럼 2차원 배열 이름의 자료형은?
 - ✓ `int b[3][5];`
 - ✓ 2차원 배열은 **1차원 배열의 배열**이므로,
 - ✓ 배열 이름 `b`의 자료형은 `int *`를 가리키는 **포인터**, 즉 이중 포인터 (`int **`) ?? NO!!
 - ✓ `b`의 자료형은 **`int (*)[5]`**


5) 배열 포인터와 다차원 배열 (심화 내용)

- 2차원 배열에 대한 고찰
 - 다차원 배열의 의미: 배열의 배열


`int a[3] ;` → a는 크기가 3인 배열, 원소는 'int'형
`int b[3][5] ;` → b는 크기가 3인 배열, 원소는 'int [5]'형
즉, '크기가 5인 int형 배열'이 3개 모인 배열


배열의 0번 원소: a[0] 


배열의 1번 원소: a[1] 

배열의 2번 원소: a[2] 

'int'형 원소

b[0] 

b[1] 

b[2] 

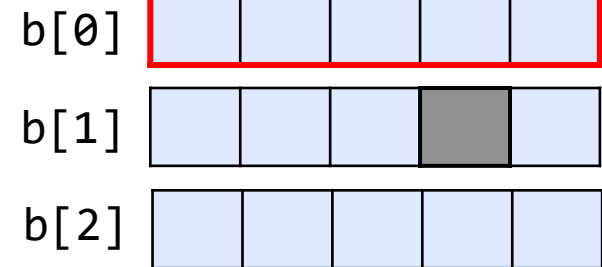
'int [5]'형 원소

5) 배열 포인터와 다차원 배열 (심화 내용)

▪ 2차원 배열과 자료형

- `int b[3][5]; == int (b[3])[5];`
 - ✓ `'int [5]'`에 대한 크기가 3인 배열

`'int [5]'`형 원소



- `b[0]`의 자료형 \rightarrow `int *`
 - ✓ 배열 이름은 그 배열의 0번 원소의 주소
 - ✓ 즉, `b[0]`의 자료형은 `&(b[0])[0] == &b[0][0]`의 자료형과 동일
 - ✓ 혼동되면, `b[0]`를 하나의 배열 변수 `X`라고 간주해보라.
즉, `int X[5]`에서 `X`의 자료형과 동일
- `b`의 자료형 \rightarrow `int (*)[5]`
 - ✓ `&b[0]`의 자료형과 동일 (배열의 0번 원소의 주소)
- `&b`의 자료형 \rightarrow `int (*) [3][5]`

5) 배열 포인터와 다차원 배열 (심화 내용)

- [예제 13.6] 이차원 배열 `b[3][5]`이 선언되었을 때, 적절한 포인터 변수를 선언하여 다음 값을 대입하는 코드를 작성해보자.

```
int b[3][5];  
  
... // 포인터 변수들 선언  
  
?? = &b[1][3];  
  
?? = b[2];  
?? = &b[2];  
  
?? = b;  
?? = &b;
```

- ✓ 잘못된 자료형의 변수를 대입했을 때 발생하는 컴파일 경고나 오류 메시지도 확인해보자.

5) 배열 포인터와 다차원 배열 (심화 내용)

- 자료형을 왜 알아야 할까?
 - 자료형은 C 프로그래밍에서 매우 중요한 요소
 - 자료형을 맞추지 않으면, 로직을 제대로 짜도 프로그램이 원하는 대로 동작하지 않을 수 있다.
 - 자료형에 대한 컴파일 경고나 오류를 보고 해석할 줄 알아야 한다.
 - 그럼, 배열의 자료형은 어디에 필요?
 - ✓ 대표적으로 함수 호출..

5) 배열 포인터와 다차원 배열 (심화 내용)

▪ 다차원 배열과 함수

- 2차원 배열을 함수의 인자로 전달하려면? (교재 8.5절에서 학습)

- ✓ 배열 이름을 인자로 전달

- ✓ 배열 이름의 자료형으로 함수 형식 인자 선언

- ✓ 형식인자의 첫 번째 첨자는 의미 없음 (보통 생략)

```
void init(int x[][5]) { // 형식인자
    ...
}
int main(){
    int b[3][5];

    init(b);    // 이차원 배열 이름 전달
    ...
}
```

5) 배열 포인터와 다차원 배열 (심화 내용)

▪ 2차원 배열과 함수 인자

- 'int x[][5]'의 정체는? 바로 배열 포인터
 - ✓ 아래 두 표기는 문법적으로나 의미적으로 완전히 동일한 자료형

```
void init(int x[][3]){  
    ...  
}
```

=

```
void init(int (*x)[3]){  
    ...  
}
```

- ✓ 비교) 1차원 배열

```
void init(int a[]){  
    ...  
}
```

=

```
void init(int *a){  
    ...  
}
```

- ✓ 배열의 차원이나 크기에 관계없이, 배열 이름을 전달한다는 것은 **주소를 전달**한다는 의미
- ✓ 배열 원소 접근에 필요한 배열 크기나 차원에 대한 정보는 **포인터 자료형에 표현**

5) 배열 포인터와 다차원 배열 (심화 내용)

- (참고) 배열 포인터 반환

- 앞의 프로그램에서 x를 반환하려면? (굳이 필요는 없지만..)
 - ✓ 함수 헤더에 반환 값의 자료형을 어떻게 표기할까?

```
int (*px)[5];                // 배열 포인터 변수 선언 형식

int (*func1())[5]{ ... }    // 인자는 없고,
                           // 배열 포인터를 반환하는 함수

int (*init( int (*x)[5] ))[5]{ ... }
                           // 인자는 하나의 배열 포인터,
                           // 배열 포인터를 반환하는 함수
```

목차

- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) main() 함수의 인자 (심화 내용)
- 5) const 키워드 (심화 내용)
- 6) 배열 포인터와 다차원 배열 (심화 내용)
- 7) void 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

6) void 포인터와 함수 포인터 (심화 내용)

- 주소는 모두 4 바이트로 표현되는 데, 왜 다양한 포인터 자료형이 필요할까?
 - 간접참조
 - ✓ 시작주소에서부터 **몇 바이트**를 읽어야 하는 지에 대한 정보 필요
 - 포인터 연산
 - ✓ 배열처럼 사용하기 위해서는 **증감 연산의 단위**가 정해져야 함
- void 포인터 (void *)
 - 자료형이 지정되지 않았음을 의미하는 특별한 포인터
 - 모든 자료형을 가리킬 수 있음
 - 하지만, 간접 참조나 증감 연산을 위해서는 **형 변환** 필요

6) void 포인터와 함수 포인터 (심화 내용)

- void 포인터 선언 및 연결
 - 기존 포인터와 동일

```
void *p;  
int i, *ip;  
char c, *cp;  
  
p = &i;           // 정상 컴파일  
p = &c;           // 정상 컴파일  
  
ip = &c;          // 컴파일 경고 또는 오류  
cp = &i;          // 컴파일 경고 또는 오류
```

6) void 포인터와 함수 포인터 (심화 내용)

- void 포인터 연산

- 간접 참조나 증감 연산 시 **형 변환** 필요

```
int x[2] = {4, 8};  
void *p = x ;  
  
printf("%d ", *p );           // 컴파일 오류  
p = p+1 ;                     // 컴파일 오류  
  
printf("%d ", *(int *)p );    // o.k.  
p = (int *)p + 1;             // o.k.  
printf("%d ", *(int *)p);
```

- 실사용 예: malloc 함수의 반환형은 void *

```
int *p = NULL;  
  
p = (int *) malloc( 5*sizeof(int) );
```

6) void 포인터와 함수 포인터 (심화 내용)

- 함수 이름은 포인터
 - 변수가 메모리에 할당되어 주소를 가지듯이
함수의 코드도 메모리를 차지하고 주소를 가짐
 - ➔ 함수를 가리키는 포인터(함수 포인터)도 가능
- 함수포인터
 - 함수의 형태에 따라 서로 다른 자료형으로 간주
 - ✓ 함수의 형태: 인자와 반환값의 자료형 형태로 정해짐
 - ✓ 예: `void print(int, int)`

6) void 포인터와 함수 포인터 (심화 내용)

- 함수포인터 선언 및 연결의 예

```
int add(int a, int b){
    return a+b;
}

int main() {
    int (*fp) (int,int); // 반환형 int, 인자가 int형 2개인
                        // 함수 포인터
    fp = add;           // 연결

    printf("%d\n", add(2,3)); // 결과 5: add() 함수의 반환값
    printf("%d\n", fp(1,5));  // 결과 6: fp() 함수의 반환 값
    return 0;
}
```

6) void 포인터와 함수 포인터 (심화 내용)

- 다양한 포인터 선언 예시 및 비교

일반 변수 및 함수 선언	포인터 선언
<code>int i;</code>	<code>→ int (*ip);</code>
<code>int a[10];</code>	<code>→ int (*ap)[10];</code>
<code>int a2[5][7];</code>	<code>→ int (*ap2)[5][7];</code>
<code>void f1();</code>	<code>→ void (*fp1)();</code>
<code>void f2(int, int);</code>	<code>→ void (*fp2)(int, int);</code>
<code>int f3(int, int *);</code>	<code>→ int (*fp3)(int, int *);</code>

6) void 포인터와 함수 포인터 (심화 내용)

- [예제 13.7] 함수 포인터를 사용한 덧셈, 뺄셈 연산

```
int add(int a, int b){    return a+b;    }
int sub(int a, int b){    return a-b;    }
int main() {
    char op;    int a, b;
    int (*fp) (int,int); // 함수 포인터 선언

    scanf("%c",&op);    // '+' or '-' 입력
    scanf("%d %d", &a, &b); // 피연산자 입력

    if( op == '+' ) fp = add;
    else fp = sub;    // 연산자에 따라 적절한 함수 연결

    printf("=>%d", fp(a,b)); // 간접 함수 호출
}
```

실행 예시

+
2 6
=>8

6) void 포인터와 함수 포인터 (심화 내용)

- void 포인터와 함수 포인터의 확장

- 일반적인 포인터와 마찬가지로 다양한 형태로 확장 가능
 - ✓ 포인터 배열, 구조체, 함수 인자 등
 - ✓ 예) 함수포인터의 배열

```
void (*fp1[5])( );
```

- void 포인터와 함수 포인터를 사용하는 이유

- 확장성, 편리성
 - ✓ 포인터를 사용하는 일반적인 이유임
- 지금은 이런 것도 있구나 정도만 알아두자.

목차

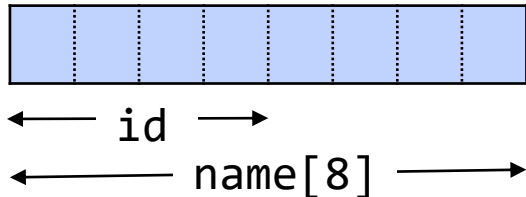
- 1) 비트연산자
- 2) 재귀함수
- 3) 라이브러리 활용
- 4) `main()` 함수의 인자 (심화 내용)
- 5) `const` 키워드 (심화 내용)
- 6) 배열 포인터와 다차원 배열 (심화 내용)
- 7) `void` 포인터와 함수 포인터 (심화 내용)
- 8) 공용체와 열거형 (심화 내용)

7) 공용체와 열거형 (심화 내용)

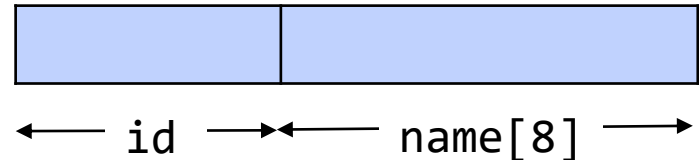
▪ 공용체

- 구조체와 비슷하게 멤버들로 구성되는 사용자 자료형
- 구조체와 차이점
 - ✓ 모든 멤버가 메모리를 공유

```
union student{  
    int id;  
    char name[8];  
};
```



```
struct student{  
    int id;  
    char name[8];  
};
```



7) 공용체와 열거형 (심화 내용)

- 공용체 초기화 및 사용

- 첫 번째 멤버만 초기화 가능
- 멤버의 값을 바꾸면 다른 멤버의 값에도 영향을 미침

```
union student st = {20001015};  
  
printf("id: %d\n", st.id);  
  
strcpy(st.name, "tom");  
  
printf("name: %s\n", st.name);  
printf("id: %d\n", st.id);
```

실행 결과

```
id: 20001015  
name: tom  
id: 7171956
```

7) 공용체와 열거형 (심화 내용)

▪ 열거형

- 이름이 부여된 정수 상수의 집합
- 코드의 가독성을 높여줌
- 내부적으로는 0부터 차례로 멤버들에게 정수가 부여됨

```
enum day_type {sun, mon, tue, wed, thu, fri, sat};

int main(){
    enum day_type day; // day_type 열거형 변수 day 선언
    day = mon;          // 변수에 값(상수) 대입
    if( day == mon)
        printf("Monday");
    return 0;
}
```

결과:

Monday