

# C Basic

# 목차

1. C언어 Overview
2. 연산자(operator)
3. 제어문(control statement)
4. 배열(array)
5. 포인터(pointer)
6. 함수(function)
7. 변수의 기억류(Storage Class)
8. 구조체(struct)

# 1. C언어 Overview



- ◆ C언어의 특징 및 역사를 이해한다.
- ◆ C언어 Compile 과정을 이해한다.
- ◆ C언어 기본 구조를 이해한다.
- ◆ 변수/상수를 이해한다.
- ◆ Escape Sequence를 이해한다.
- ◆ 기본 데이터 Type을 이해한다.
- ◆ 표준 입출력을 이해한다.

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수 와 상수

1.5 Escape Sequence

1.6 기본 데이터 Type

1.7 표준 입/출력

---

# 1. C언어 Overview

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수 와 상수

1.5 Escape Sequence

1.6 기본 데이터 Type

1.7 표준 입/출력

---

- ➡ structured programming language
- ➡ UNIX 운영체제하에서 시스템 프로그래밍을 하기 위해 개발된 아주 강력한 기능을 가진 프로그래밍 언어
- ➡ 어셈블리어와 혼용해서 사용 가능
- ➡ 다른 고급언어와 연결(linking) 가능
- ➡ 이식성(portability) 강한 언어
- ➡ 빠른 수행속도를 가진 언어
- ➡ 풍부한 자료형/연산자/자체함수를 가진 언어
- ➡ ANSI 표준에 근거한 언어

➡ C 언어의 역사

1972년 미국의 AT&T Bell 연구소에서 Dennis Ritchie에 의해서 C language 이전에 B language의 자료형으로 기계 word 밖에 없었던 점을 보완하여 UNIX 운영체제를 위한 system 언어로 개발되었다.

➡ C 언어의 특징

- C 언어는 컴파일러 언어로써 효율적인 언어이다.
- C 언어는 이식하기 편리한 언어이다.
- High-level language & Low-level language compiler program용, system program용으로 사용 가능하다.

High-level language	Low-level language
<ul style="list-style-type: none"><li>• program의 제어구조</li><li>• data type</li></ul>	<ul style="list-style-type: none"><li>• bit operation</li><li>• address manipulation이 용이</li></ul>

➡ Why ANSI Standard C

Bell 연구소의 C 언어가 빠르게 보급되면서 각각의 프로그래머들은 자신만의 환경에 맞도록 언어를 수정하여, 다른 곳에서 작성된 C 프로그램을 정상적으로 실행하기 위해선 수정이 불가피하게 되었다.

이러한 문제점을 해결하기 위해 1983년, 미국의 국가표준협회(ANSI : American National Standard Institute) 에서 C에 대한 표준안을 발표하게 되었다.

# 1. C언어 Overview

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수 와 상수

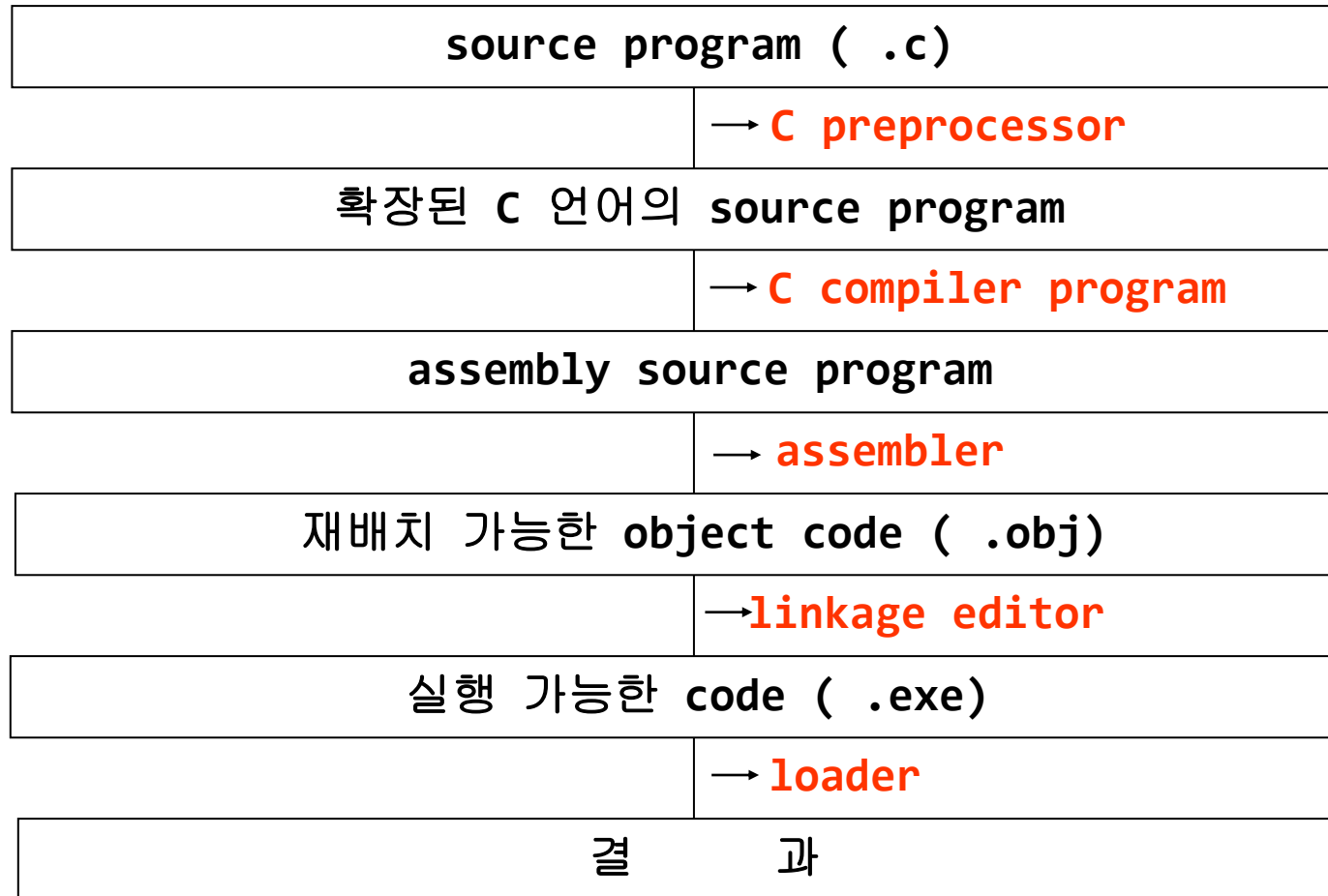
1.5 Escape Sequence

1.6 기본 데이터 Type

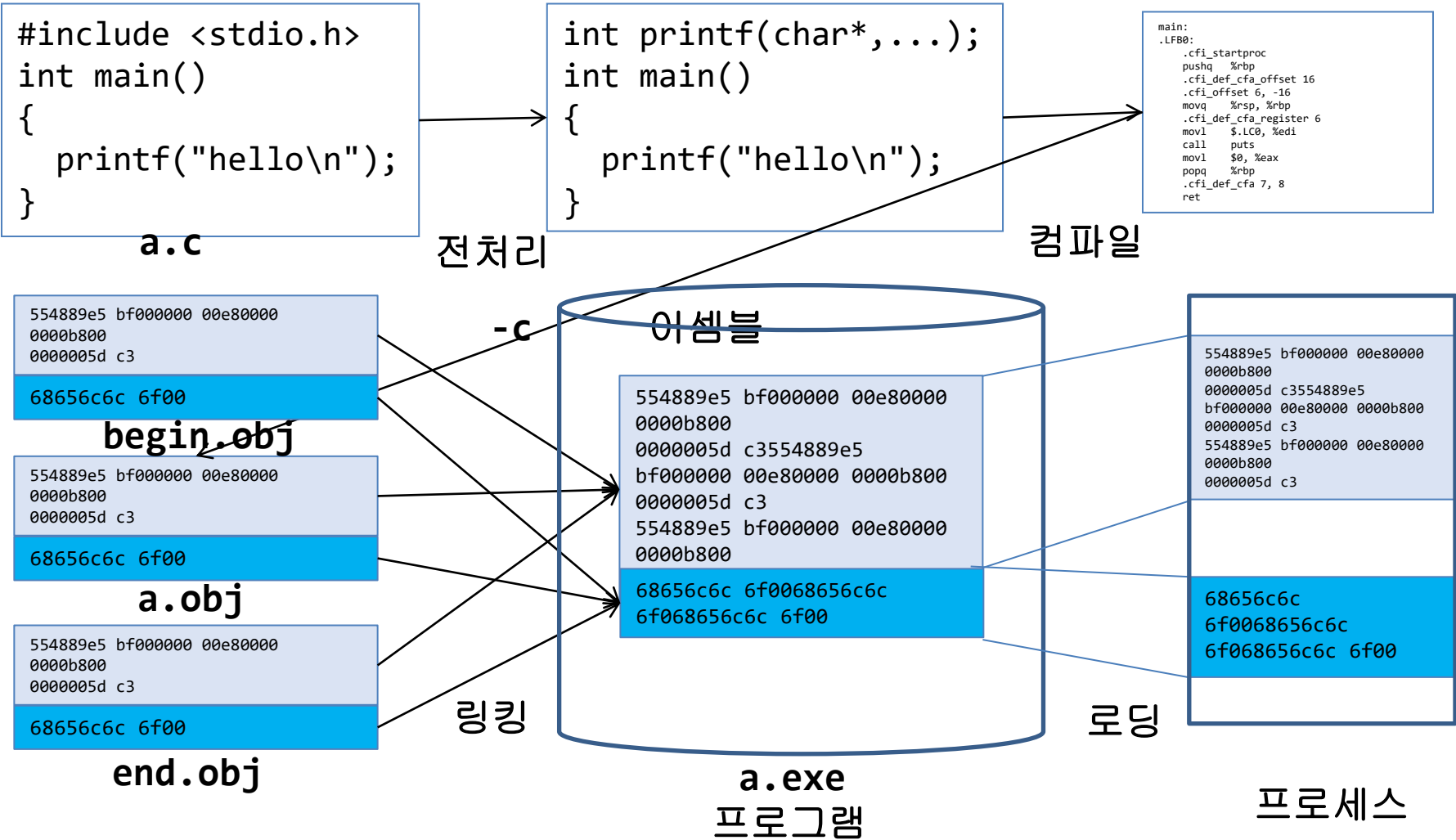
1.7 표준 입/출력

---





◆ 프로세스 빌드 과정



**C 프로그램을 실행시키기 위해서는 다음과 같은 절차가 필요하다.**

- ① editor로 source program을 작성한다.
- ② compile해서 object file을 만든다.
- ③ 다른 object file이나 library 등과 link해서 execute file을 만든다.

▪ **소스코드(source code)** : 프로그램의 내용 자체를 소스 코드라 한다.

▪ **소스파일(source file)** : 소스코드를 텍스트 파일에 기술하여 만들어진 파일로 C 언어의 소스파일 확장자는 .c 이다.

▪ **선행처리기지시어가 번역된 소스파일** : C 언어 프로그램 내에는 여러 가지 지시어들이 있는데 이걸 C 문법과는 별개로 처리되고 이것을 번역하는 프로그램을 선행처리기(preprocessor)라고 한다.

▪ **목적파일(object file)** : 지시어가 번역된 소스 파일은 다음으로 컴파일러(compiler)라는 프로그램에 의해 기계어로 번역된다. 이 번역된 파일을 목적 파일이라 하고, 확장자는 .obj 이다.

▪ **실행파일(executable file)** : 목적파일에 실행 부분이 들어있는 라이브러리를 덧붙여 주는 소프트웨어, 링커(linker)에 의해 실행파일을 작성하게 된다.

# 1. C언어 Overview

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수 와 상수

1.5 Escape Sequence

1.6 기본 데이터 Type

1.7 표준 입/출력

---

```
/* This is test */  
#include <stdio.h>  
  
int main()  
{  
    printf("Hello World!\n");  
    return 0;  
}
```

→ comment

→ preprocessor statement

→ function header

} → function body

**C 언어는 하나의 main 함수와 여러 개의 함수들로 구성된다.**

main() 이라는 특수한 함수는 C 프로그램의 처음 실행 위치를 나타내며 C 언어로 작성된 program은 반드시 함수 main()을 한 개만 보유해야 한다.

**C 언어는 프로그램 구성 시 문장이 기본 단위가 된다.**

문장은 프로그램을 구성하는 중요한 요소로 실행의 단위가 되고, 모든 문장은 반드시 ";"로 끝나야 한다.

**함수는 {로 시작하고 }로 끝난다.**

구조화 프로그램을 채택하고 있는 C 언어는 프로그램의 실행을 정의하는 문장들을 module로 정의하는데, 한 module에 속하는 문장들은 중괄호로 표시한다. 이를 block이라 한다.

**주석문은 /\* 와 \*/로 둘러싼다.**

프로그램의 어느 부분에서든지 사용이 가능한 것으로 program이나 문장의 내용을 설명할 때, debugging등 사용자 편의를 위하여 주로 사용한다.

**필요하다면 헤더파일(header file)을 지정한다.**

헤더파일에는 프로그램 작성을 편리하게 하고 프로그램을 안전하게 컴파일하기 위한 다양한 정보가 기술되어 있다. 여러 종류의 헤더 파일 중에서 stdio.h는 가장 기본이 되며 중요하다고 할 수 있다.

- C 프로그램은 다음과 같은 일련의 문자로 구성된다.

문      자	종                      류
lowercase letters	a, b, c, d, ..., x, y, z
uppercase letters	A, B, C, D, ..., X, Y, Z
digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
special characters	+, =, <u>_</u> , -, (, ), *, &, %, \$, #, !,  , <, >, ., [, ], ^, ~, ,, ;, :, “, ‘, ...
whitespace	blank, new line, tab

프로그램은 위의 문자들로 구성된 토큰(token)이라고 부르는 해석 상의 의미 있는 문자 또는 문자열로 분할된다.

토큰은 식별자, 예약어, 연산자, 분리자 등이 모두 포함된다.

### 식별자(identifier)

- 식별자 : 프로그래머가 프로그램에서 사용하는 변수, 함수, 상수 등에 부여한 이름
- 작성규칙
  - ① 영문자(a ~ z, A ~ Z), 숫자(0 ~ 9), 밑줄(underscore)의 조합으로 구성
  - ② 예약어는 사용하면 안되지만 예약어를 포함하는 것은 상관없다.
  - ③ 첫 글자는 영문자, underscore로 시작한다.
  - ④ 변수 명 안에는 공백이 있으면 안 된다.
  - ⑤ 대문자와 소문자는 서로 다른 문자로 처리된다.
  - ⑥ 통례적으로 변수 명은 소문자로, 기호 상수 명은 대문자를 사용한다.



예약어 (reserved word)

- 예약어 : compiler에 대해서 특별한 의미를 갖는 언어
- ANSI C 에서 예약어는 다음과 같다.

auto	do	goto	signed	while
break	double	if	sizeof	unsigned
case	else	int	static	void
char	enum	long	struct	volatile
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

연산자 (operator)

- +, -, \*, / 등 틀수한 의미를 갖는 문자로 각각 덧셈, 뺄셈, 곱셈, 나눗셈의 산술 연산을 의미한다.
- 이 외에도 C 언어에는 이항 연산자, 단항 연산자, 동등 연산자, 논리 연산자, 증가 연산자, 감소 연산자, bit 연산자, 주소 연산자, 배정 연산자 등 여러 가지 연산자가 있다.

### 문자열 (string)

- 문자열(string)은 원소 제일 끝에 null 문자(' 0')가 들어간 문자형 자료의 배열이며, 이중부호(" ")로 묶어 사용한다.

(예제)

“abcdef” →

내부형식 :

a	b	c	d	e	f	\0
---	---	---	---	---	---	----

- 문자열은 자동적으로 제일 끝에 ' 0'이 부가되므로 문자열의 끝을 판단할 수 있다.

### 선언문 (declaration statement)

- 프로그램에서 연산 대상으로 제공되는 것에는 변수와 상수가 있으며, 이 중 모든 변수는 사용하기 전에 반드시 선언되어야 한다.
- 선언문의 사용목적
  - ① compiler에게 미리 자료형을 알려줌으로써 기억 장소를 확보하고 효율적인 연산을 할 수 있도록 한다.
  - ② 사용자가 자료형이나 변수형을 무의식 중에 잘못 사용했을 경우 발생하는 오류를 알려준다.

# 1. C언어 Overview

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수와 상수

1.5 Escape Sequence

1.6 기본 데이터 Type

1.7 표준 입/출력

---

- ➡ 프로그램에서 사용되는 자료를 저장하기 위한 공간
- ➡ 할당 받은 메모리의 주소 대신 부르는 이름
  - 사용자가 변수 이름을 만들어 저장하기 위한 자료의 형태를 지정하면 compiler는 컴퓨터의 메인 메모리에 주소 값과 연결시켜준다.
- ➡ 프로그램 실행 중에 값 변경 가능
- ➡ 사용되기 이전에 선언되어야 함

### 변수의 선언

변수를 지정하는 것을 선언이라고 하는데, 변수의 선언은 프로그램에서 필요한 저장공간을 컴퓨터의 메인 메모리에 확보하라는 의미이다. 형식은

```
변수의 자료형태 변수 이름( = 초기값, 변수 이름= 초기값...);
```

의 형태로 이루어진다.

- 변수의 자료형태 : 변수에 저장할 값이 정수인지 소수인지 문자인지 등등을 정하는 키워드
- 변수명 : 그 변수를 상징하는 symbol 로, 식별자의 작성규칙에 맞추어 이름을 정하면 된다.
- 초기값 : 처음에 들어갈 값으로, 지정하지 않는 경우는 변수의 종류에 따라 쓰레기 값이 들어있는 것과 자동적으로 0 이 되는 것이 있다.

```
char ch = 'A';           // 문자형 변수 ch (1 byte 공간)
int a;                   // 정수형 변수 a (4 byte 공간)
int b = 10;              // 정수형 변수 b (4 byte 공간)
float f;                 // 실수형 변수 f (4 byte 공간)
double d = 3.14;        // 실수형 변수 d (8 byte 공간)
```

```
#include <stdio.h>

int main()
{
    int a;
    char ch;
    a = 10;
    ch = 'A';
    return 0;
}
```

} 변수 선언부

} 변수 처리부

변수는 선언과 동시에 초기화를 할 수 있다.

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

```
int main()
```

```
{
```

```
    int a = 10;
```

```
    char ch = 'A';
```

```
    return 0;
```

```
}
```

} 변수 선언부 및 초기화

### ➡ 프로그램 실행 중에 변하지 않는 고정된 수

#### ▪ 실제 값 상수

- 정수형 상수 (integer constant)
- 실수형 상수 (real constant)
- 문자형 상수 (character constant)
- 문자열 상수 (string constant)

### ➡ 기억 장소를 갖는 상수

- 선언하는 것은 변수 선언과 똑같지만 앞에 `const` 키워드를 써야 하고 반드시 초기값이 있어야 한다는 것이 다름.



## 정수형 상수

- 고정 소수점 상수 (fixed point constant) 라고도 한다.
- 부호(+ 또는 -) 및 0 ~ 9(10진수인 경우)로 구성한다. 단, 16진수인 경우는 0 ~ 9, a ~ f (또는 A ~ F)로 구성한다. 8진수는 0 ~ 7로 구성한다.
- 정수 값은 10진수, 8진수, 16진수로 표현 할 수 있다. 10진수 표현은 숫자 앞에 아무것도 붙지 않으며, 16진수 표현은 숫자 앞에 0x 또는 0X가 붙고, 8진수 표현은 숫자 앞에 0(zero)이 붙는다.
- 정수 값 다음에 대문자 L 또는 소문자 l을 붙일 수 있다. 이 때는 long data로 취급한다.  
사용 예) 36(10진수), 44(8진수), 24(16진수), 36L(10진수 36의 long 형태의 정수형 상수)

## 실수형 상수

- 부동 소수점 상수 (floating point constant) 라고도 한다.
- 십진형 : 10진수 형태의 실수. 정수부, 소수점, 소수부로 이루어진 실수의 기본형이다.
  - ① 부호 및 0 ~ 9로 구성되고, L 또는 l를 붙여 long형태의 실수로 표현할 수도 있다.
- 지수형 : 매우 큰 수나 작은 수를 표현할 때 10의 배수 형태로 표현하는 형태로, 소수부(fraction)와 지수부(exponential)로 구성된다.
  - ① 소수부와 지수부(E 또는 e)로 구성 한다.
  - ② 소수부는 10진수 형태의 소수점을 포함한 실수
  - ③ 지수부의 기호(E 또는 e)는 10의 거듭 제곱을 표현하기 위한 밑수(base)를 나타낸다.  
사용 예) 123.456(십진형 실수), 1.234560e+002(지수형 상수)

# 1. C언어 Overview

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수와 상수

1.5 Escape Sequence

1.6 기본 데이터 Type

1.7 표준 입/출력

---

문 자 열	코드 약호	기 능	
\a	07	BEL	내장 벨을 울린다(alert)
\b	08	BS	back space
\f	0C	FF	인쇄 시 종이 한 장을 넘김(form feed)
\n	0A	LF	new line
\r	0D	CR	carriage return
\t	09	HT	horizontal tab
\v	0B	VT	vertical tab
\\	5C		\ 자신
\'	2C		'(문자정수 대책)
\''	22		“(문자열정수 대책)
\0oo		Octal	8진수(o는 8진수 숫자를 의미)
\xhh		Hexa	16진수(h는 16진수 숫자를 의미)
\0	00	NULL	NULL character

### 문자형 상수

- C 언어에서 한 문자를 나타낼 때에는 단일 인용 부호 ( ' ')를 사용한다.
- 하나의 문자만을 가져야 하고, 문자 상수 값은 실제로 컴퓨터의 기억 장소에 기억시킬 때에는 ASCII 코드 값이 된다.  
[사용 예] 'A'                    /\* ASCII 코드 값 0x41을 의미 \*/
- 단일 인용부호 ' '내에 여러 문자로 표현 할 수 있는 경우는 \w(역슬래쉬)와 결합된 형태뿐이다.  
즉, escape 문자들이 이에 해당된다.  
[사용 예] '\wn'                /\* ASCII 코드 값 0xa 이며 '새로운 줄을 바꾸라'는 의미 \*/

### 문자열 상수

- C 언어에서 여러 문자를 나타낼 때에는 이중 인용 부호(" ")를 사용한다.
- 문자 배열의 형태로 저장된다.  
사용 예) "C is powerful language"

### 기억 장소를 갖는 상수

선언하는 방법, 사용법도 변수와 같지만, 선언 시에 앞에 const 키워드를 써야 하고, 안에 있는 값을 변경할 수 없다는 것이 다르다. 형식은

**const** 상수의 자료형태 상수 이름 = 초기값, (상수이름 = 초기값...);

의 형태로 이루어진다.                [사용 예] const int a = 1000;

# 1. C언어 Overview

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수와 상수

1.5 Escape Sequence

1.6 기본 데이터 Type

1.7 표준 입/출력

---

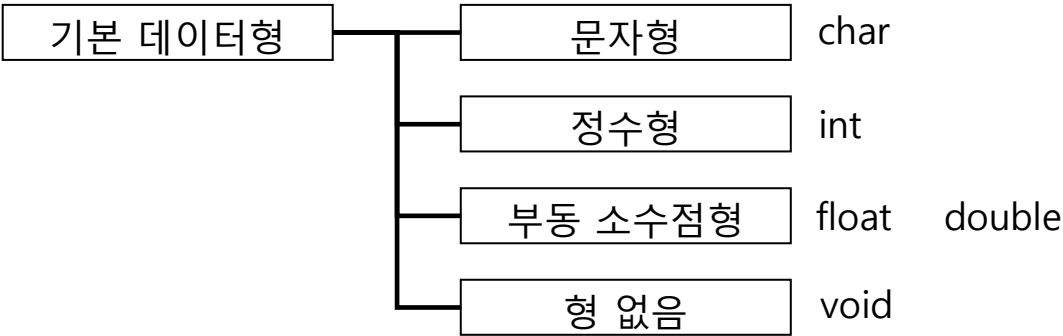
32bit CPU인 경우

구 분	자 료 형	범 위	바이트
문자형	char	-128 ~ 127 ( $-2^7 \sim 2^7-1$ )	1
	unsigned char	0 ~ 255 ( $0 \sim 2^8-1$ )	
정수형	short	-32768 ~ 32767 ( $-2^{15} \sim 2^{15}-1$ )	2
	unsigned short	0 ~ 65535 ( $0 \sim 2^{16}-1$ )	
	int	-32768 ~ 32767 ( $-2^{15} \sim 2^{15}-1$ ) -2147483648 ~ 2147483647 ( $-2^{31} \sim 2^{31}-1$ )	2 4
	unsigned int	0 ~ 65535 ( $0 \sim 2^{16}-1$ ) 0 ~ 4294967295 ( $0 \sim 2^{32}-1$ )	
	long	-2147483648 ~ 2147483647 ( $-2^{31} \sim 2^{31}-1$ )	4
	unsigned long	0 ~ 4294967295 ( $0 \sim 2^{32}-1$ )	
실수형	float	$\pm 3.4 * 10^{-38} \sim \pm 3.4 * 10^{38}$ (유효자릿수 7)	4
	double	$\pm 1.7 * 10^{-308} \sim \pm 1.7 * 10^{308}$ (유효자릿수 15)	8
	long double	$\pm 3.4 * 10^{-4916} \sim \pm 1.1 * 10^{4932}$ (유효자릿수 19)	10
무치형	void		

64bit CPU인 경우

구 분	자 료 형	범 위	바이트
문자형	char	-128 ~ 127 ( $-2^7 \sim 2^7-1$ )	1
	unsigned char	0 ~ 255 ( $0 \sim 2^8-1$ )	
정수형	short	-32768 ~ 32767 ( $-2^{15} \sim 2^{15}-1$ )	2
	unsigned short	0 ~ 65535 ( $0 \sim 2^{16}-1$ )	
	int	-32768 ~ 32767 ( $-2^{15} \sim 2^{15}-1$ ) -2147483648 ~ 2147483647 ( $-2^{31} \sim 2^{31}-1$ )	2 4
	unsigned int	0 ~ 65535 ( $0 \sim 2^{16}-1$ ) 0 ~ 4294967295 ( $0 \sim 2^{32}-1$ )	
	long	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,8087 ( $-2^{63} \sim 2^{63}-1$ )	8
	unsigned long	0 ~ 18,446,744,073,709,551,616 ( $0 \sim 2^{64}-1$ )	
실수형	float	$\pm 3.4 * 10^{-38} \sim \pm 3.4 * 10^{38}$ (유효자릿수 7)	4
	double	$\pm 1.7 * 10^{-308} \sim \pm 1.7 * 10^{308}$ (유효자릿수 15)	8
	long double	$\pm 3.4 * 10^{-4916} \sim \pm 1.1 * 10^{4932}$ (유효자릿수 19)	10
무치형	void		

기본 데이터형은 char, int, float, double로 구성되며 이에 특수한 데이터형인 void 형이 추가된다.



long과 short 는 기본 데이터의 크기를 수식하여 표준 이상인지 이하인지 지정한다.

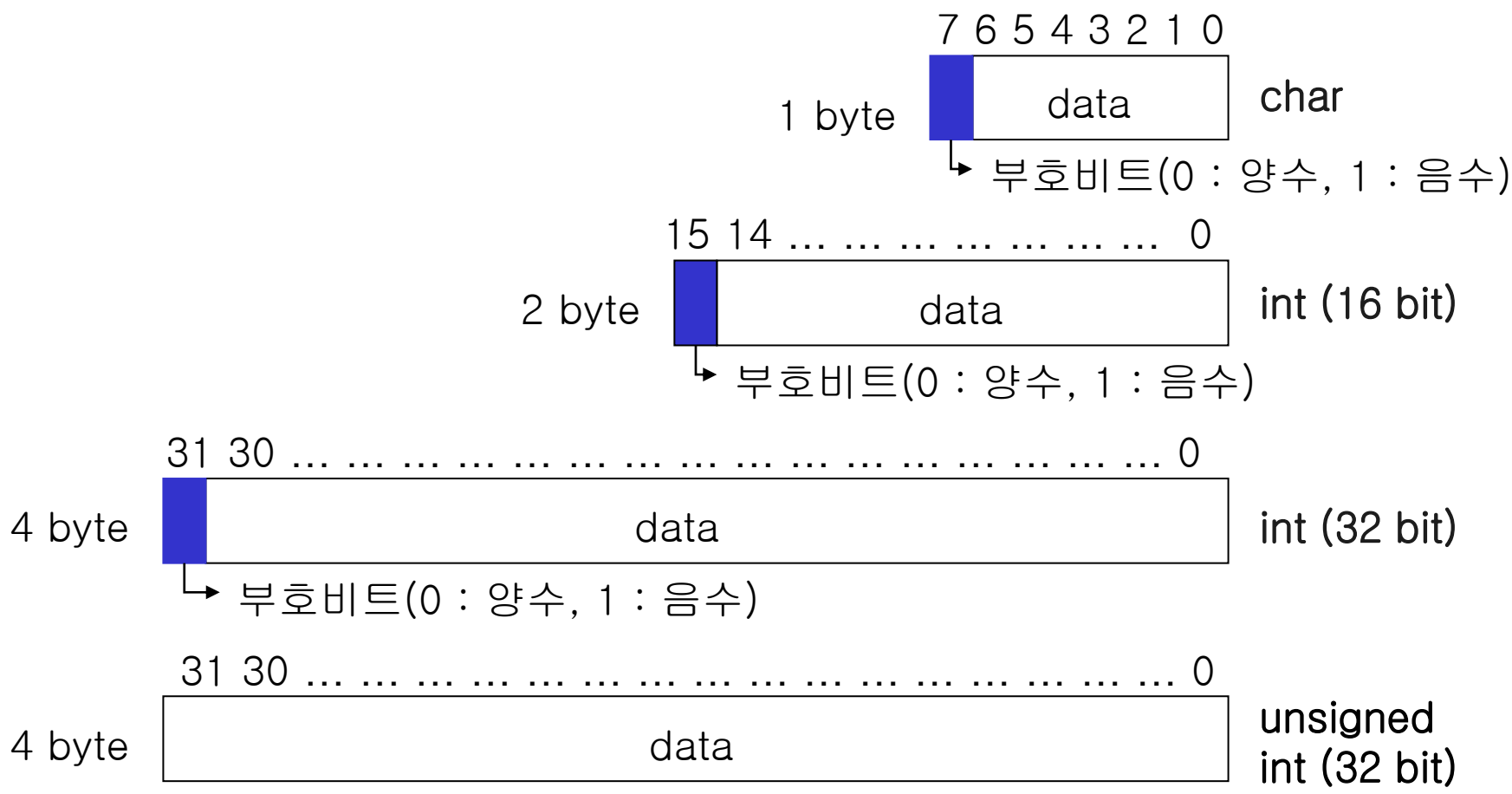
- 정수형의 대소관계 : short <= int <= long
- 부동소수점의 대소관계 : float <= double <= long double

char나 int 등 정수형의 변수는 부호 수식을 할 수 있다.

최상위 비트를 부호(0 이면 양수, 1 이면 음수)로 취급하여 나타내고, 부호지정이 없을 때는 부호가 있는 것(signed)로 간주된다.

void 형은 값을 갖지 않는 특수한 데이터형으로 값을 갖지 않음을 명시한다.





### 문자형의 크기 및 데이터

- 문자형인 char의 크기는 1 바이트이며 이것은 어떤 C 시스템이든 동일하다.

### 정수형의 크기 및 데이터

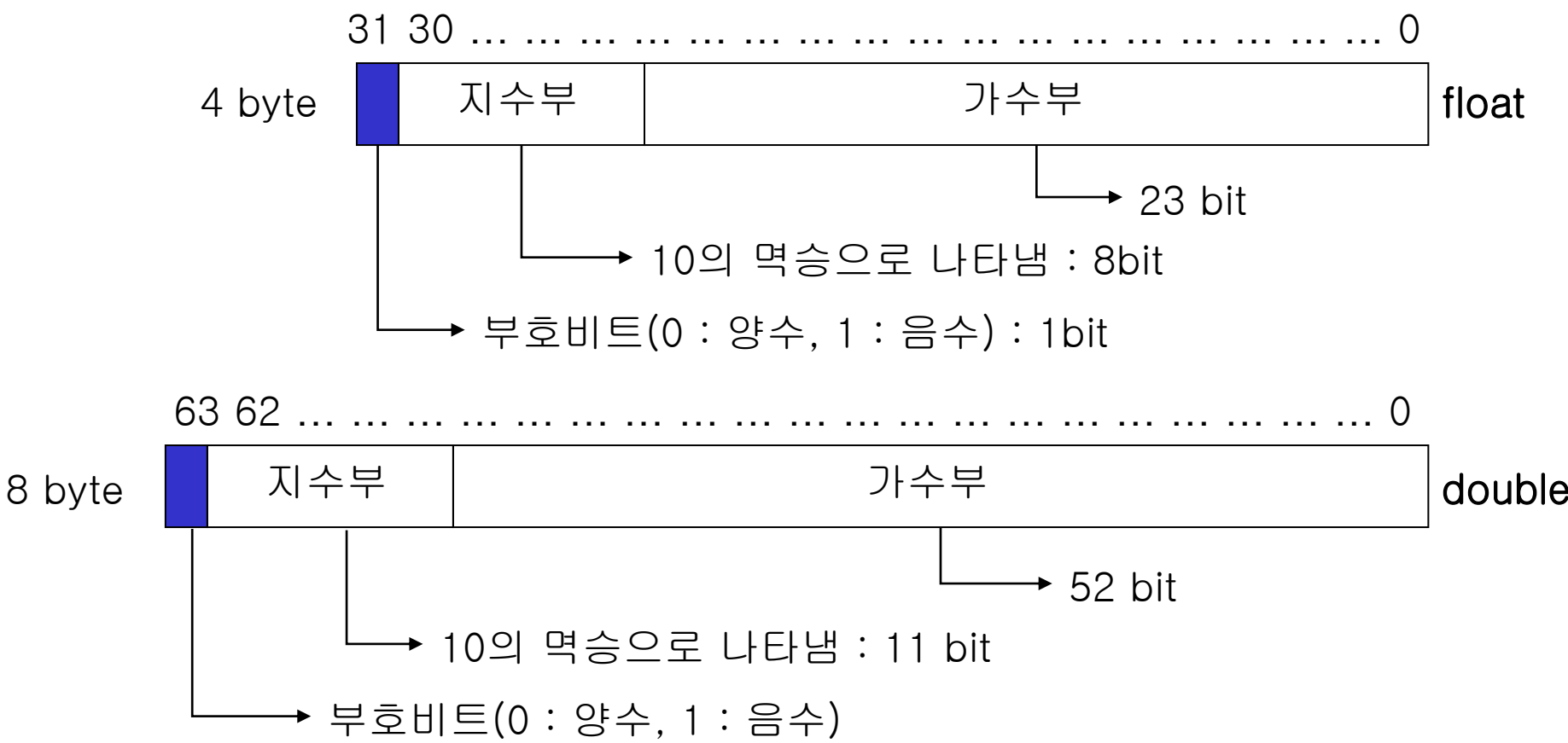
- short <= int <= long
- C 언어 시스템에 따라 달라 16비트 또는 32비트가 된다. 이것은 사용하는 컴퓨터의 환경상 가장 자연스럽게 효율적으로 처리할 수 정수 폭이라는 개념에 따라 결정된다. 일반적으로 종래의 MS-DOS 환경의 C 언어는 16 비트 int, Windows(상의 MS-DOS 프로그램을 포함)나 UNIX 환경의 C 언어는 32 비트 int 형으로 되어있다.

### char 형과 int 형은 그 기능은 동일하지만 표현하는 범위가 다르다.

- char 형은 문자형이라고 불리는데 이것은 ASCII 문자 하나를 표현할 수 있기 때문이지 특별히 문자를 취급하기위한 편리한 기능을 가지고 있어서는 아니다.

char 형	→ 1 바이트의 정수형
int 형	→ 표준 바이트의 정수형(16비트 또는 32비트)

- char 형은 1바이트의 정수이므로 256종류( $2^8$ )의 구별이 가능하여 영문이나 숫자를 표현하는데 충분하다. 즉, 문자를 표현하는 데 크기면에서 적당하기 때문에 문자형이라 불리는 것이다.



# 1. C언어 Overview

---

1.1 C언어 특징 및 역사

1.2 C언어의 Compile

1.3 C언어의 구조

1.4 변수와 상수

1.5 Escape Sequence

1.6 기본 데이터 Type

1.7 표준 입/출력

---

header file : <stdio.h>

표준 입/출력 함수	의 미
getchar()	fetch a character
putchar()	print a character
gets()	fetch a line
puts()	print a line
scanf()	fetch formatted input
printf()	fetch formatted output

**C 에서 입/출력 기능은 C 시스템 본체의 기능이 아니다.**

C 언어는 입/출력 기능을 언어의 기능에 포함하고 있지 않다. 따라서 입/출력 명령을 C 언어에서 가지고 있지 않기 때문에 library 함수 형태로 제공받고 있으며 C 언어에서의 모든 data 입/출력은 library 함수 호출 형태로 이루어진다.

**C compiler 개발자가 제공하는 library 함수를 표준 함수라고도 한다.**

### 콘솔 입/출력

- 「표준 입력」에서 입력하고 「표준 출력」에 출력한다.
- 표준 입력 (stdin) – 키보드  
표준 출력 (stdout) – 디스플레이(모니터)  
표준 에러 (stderr) – 디스플레이(모니터)

### 스트림(stream)

디스크나 그 외의 장치를 오가는 데이터의 모임

```
int putchar(int c);
```

기 능 ➤ 문자 `c`를 `stdout`에 출력한다.

리턴값 ➤ 정상시 : 출력한 문자, 에러시 : EOF

```
#include <stdio.h>

int main()
{
    putchar('T');
    putchar('\n');
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    char ch = 'A';
    putchar(ch);
    return 0;
}
```

### putchar 함수

- 지정된 문자를 표준 출력 스트림으로 출력하는 함수.
- 문자의 지정은 문자변수, 문자상수 혹은 ASCII code값 연산식이 가능하다.
- 반드시 하나의 인수를 가지고 있다.

[사용 예] `putchar('a');`

`/* 'a'를 출력한다. */`

```
int puts(const char *string);
```

기 능 ➤ 문자열 `string`을 `stdout`로 출력한 후, 개행을 행한다.

리턴값 ➤ 정상시 : 음수가 아닌 값, 에러시 : EOF

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

```
int main()
{
    puts("ABC");
    puts("DEF");
    return 0;
}
```

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

```
int main()
{
    puts("Hello World!\n");
    puts("Happy Birthday!\n");
    return 0;
}
```

### puts 함수

- 문자열 출력 함수로 버퍼의 문자열을 null이 나올 때까지 화면에 출력하는 함수
- null을 만나면 자동적으로 ‘\n’이 출력된다.

[사용 예] `puts("Hello");`                      `/* "Hello"라고 출력한다. */`



```
int printf(const *format[,argument],...);
```

기능

대응하는 인수의 내용을 `format`에서 지정한 서식에 따라 `stdout`에 출력한다.

리턴값

정상시 : 출력한 문자 수, 에러시 : 음의 값

```
#include <stdio.h>

int main()
{
    printf("%c, %d, %f\n", 'w', 10, 10.23);
    printf("%o, %x, %e\n", 10, 10, 10.23);
    return 0;
}
```

### printf 함수

- `format`(변환기호)에 의하여 출력 대상을 변환, 편집하여 표준 출력 스트림에 출력하는 함수
- 출력양식의 변환기호는 ‘%’로 시작한다.
- 출력 대상은 상수, 변수, 수식 등이며 ‘;’로 구분한다.
- `format`에는 다음과 같이 3 종류의 문자열을 쓸 수 있다.

종 류	변 환 문 자	의 미 ( % ➔ conversion specifiers )	
정수형	%d	decimal	인수를 10진수로 변환한다.
	%o	octal	인수를 8진수로 변환한다.
	%x 혹은 %X	hexa	인수를 16진수로 변환한다.
	%u	unsigned	인수를 부호 없는 10진수로 변환한다.
문자형	%c	character	인수를 한 문자로 출력한다.
문자열	%s	string	인수에 따라 지시되는 문자열을 출력한다.
부동 소수점형	%f	float	인수를 float 혹은 double 형으로 받아들여 십진형 소수로 출력한다.
	%e 혹은 %E	exponent	인수를 float 혹은 double 형으로 받아들여 지수형 소수로 출력한다.
	%g	generate	%e 또는 %f에 따른 변환 중 문자수가 적은 쪽의 변화를 취한다
주소형	%p	pointer	메모리의 주소 값을 16진수 형태로 출력한다.

### 정수의 덧셈과 출력

```
int main()
{
    int x, y, z;
    x = 123, y = 456;
    z = x + y;
    printf("x = %d\t", x);
    printf("y = %d\t", y);
    printf("z = x + y = %d\n", z);
    return 0;
}
```

### 부동 소수점의 덧셈과 출력

```
int main()
{
    float x = 123.456, y = 789.012;
    printf("x = %f\n", x);
    printf("y = %f\n", y);
    printf("x + y = %f\n", x + y);
    return 0;
}
```

수 정 자	의 미	사 용 예
-	오른쪽을 비워두고 왼쪽부터 채운다.	printf("%-5d",)
	확보할 field 폭을 m자 폭으로 취한다.	printf("%10d",)
.	field 폭과 문자 수 또는 소수점 이하의 자릿수와 구별	printf("%10.5f",)
	수치 : 소수점 이하의 자릿수 문자 : 취하는 폭	printf("%10.5s",) printf("%10.5s",)
l	대응하는 인수를 long 형으로 취한다.	printf("%ld",)

### 문자열과 문자의 출력

```
#include <stdio.h>

int main()
{
    char ch1, ch2, ch3, ch4;
    ch1 = 'Y', ch2 = 'e';
    ch3 = 's', ch4 = ',';
    printf("%c%c%c%c ", ch1, ch2, ch3, ch4);
    printf("%s\n", "I am a computer");
    return 0;
}
```

- : 왼쪽 정렬	0 : 공백 채움	+ : 부호 출력
-----------	-----------	-----------

```
#include <stdio.h>

int main()
{
    int a = 1234;
    printf("[%d]\n", a);           // 정수출력
    printf("[%8d]\n", a);         // 우측정렬
    printf("[% -8d]\n", a);       // 좌측정렬
    printf("[% +8d]\n", a);       // 부호출력
    printf("[%08d]\n", a);        // 공백채움
    printf("[% +08d]\n", a);
    return 0;
}
```

옵션 지정자

변환 문자열에는 옵션 지정자를 덧붙일 수 있는데 이 옵션 지정자는 말하자면 출력 모양을 보기 좋게 정돈해주는 보조 지정을 행한다고 할 수 있다. 이 중에서도 특히 필드 폭 지정자와 자릿수 지정자가 중요한데 이것을 사용할 때 변환 문자열은 다음과 같이 된다.



필드 폭 지정자

- %d        10 진수로 출력한다.
- %8d       8 자리의 10 진수로 출력한다.

▪ 필드 지정자의 예 (변수의 값이 1234 일 때)

지    정	출    력	설        명
%d	123	필요한 폭으로 출력된다.
%5d	_1234	앞에 스페이스 문자를 넣어 5문자로 출력된다.
%hd	1234	short 형으로 출력된다.
%2d	123	지정 폭이 작아도 필요한 폭은 확보된다.

```
#include <stdio.h>

int main()
{
    char b = 'A';
    float c = 3.14159f;
    printf("[%c]\n", b);           // 문자출력
    printf("[%8c]\n", b);          // 우측정렬
    printf("[% -8c]\n", b);        // 좌측정렬
    printf("[%f]\n", c);           // 실수출력
    printf("[%8.3f]\n", c);         // 우측정렬
    printf("[% -8.3f]\n", c);       // 좌측정렬
    printf("[%+8.3f]\n", c);        // 부호출력
    printf("[%08.3f]\n", c);        // 공백채움
    printf("[%+08.3f]\n", c);
    return 0;
}
```



자릿수 지정자

- %f        부동 소수점 수를 표준 자릿수로 출력한다.
- %10.2f    부동 소수점 수를 전체 10자리, 소수점 이하 2자리로 출력한다.

▪ 자릿수 지정자의 예(변수의 값이 12.34567일 때)

지    정	출    력	설            명
%f	12.345670	표준 폭으로 출력된다.
%12f	_ _ _ 12.345670	소수점을 넣어 12자리로 출력된다. 소수점 이하의 자릿수는 표준치가 된다.
%12.2f	_ _ _ _ _ _ _ 12.35	소수점을 넣어 9자리, 소수점 이하 2자리로 출력된다.
%12.0f	_ _ _ _ _ _ _ _ _ 12	소수점 이하 표현 없이 12자리로 출력된다.

➡ 자신의 이름을 출력하는 프로그램을 작성하시오.

➡ 1256을 8진수, 10진수, 16진수로 출력하는 프로그램을 작성하시오.

➡ This is the first line 이라고 출력한 후 2줄을 띄운 후 This is the second line 이라고 출력하는 프로그램을 작성하시오.

- 다음 프로그램의 실행 결과는 무엇인가?

```
#include <stdio.h>

int main()
{
    char ch1 = 'A', ch2 = 66, ch3 = 'A' + 2;
    printf("%c%c%c\n", ch1, ch2, ch3);
    return 0;
}
```

- x, y, z에 100을 배정한 후,  $x / 25$ ,  $y / 35$ ,  $z / 45$ 의 몫과 나머지를 구하는 프로그램을 작성하시오.
- <"That's great! You scored 99%">라고 출력하는 프로그램을 작성하시오.

```
int getchar(void);
```

기능 ➡ stdin에서 한 문자를 읽어 들인다.

리턴값 ➡ 정상시 : 읽은 문자, 파일 종료/에러시 : EOF

```
#include <stdio.h>

int main()
{
    char ch;
    printf("Enter the character : ");
    ch = getchar();
    printf("input character : %c\n", ch);
    return 0;
}
```

### getchar 함수

- 표준 입력 스트림으로부터 한 문자를 입력 받아 그 code 값을 되돌리는 함수.

[사용 예] c = getchar(); /\* 한 문자를 읽어 c에 넣는다 \*/

```
char *gets(char *buffer);
```

기능

stdin에서 문자열을 입력 받아 buffer에 넣는다.  
개행문자는 버린다.

리턴값

정상시 : 인수 buffer, 파일 종료/에러시 : NULL

```
int main()
{
    char str[10];
    printf("Enter the string : ");
    gets(str);
    printf("input string : %s\n", str);
    return 0;
}
```

### gets 함수

- 표준 입력 스트림으로부터 문자열을 입력 받는 함수.
- 문자열(string) 입력 함수로 입력의 끝은 null 이 부가되므로 buffer의 크기는 "예상되는 입력 최대 문자 수 + 1"을 확보해야 한다.
- 문자열 입력 시 공백을 허용한다.

[사용 예] gets(ss);                      /\* 1행을 입력하고 ss에 넣는다. \*/

```
int scanf(const *format[,argument],...);
```

기능

stdin에서 서식 `format`에 따라 데이터를 읽어와 해당 변수에 저장한다.

리턴값

정상시 : 입력 데이터의 개수, 파일 종료/에러시 : EOF

```
#include <stdio.h>

int main()
{
    char ch;
    printf("Enter the character : ");
    scanf("%c", &ch);
    printf("character : %c\n", ch);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int num;
    printf("Enter the number : ");
    scanf("%d", &num);
    printf("number : %d\n", num);
    return 0;
}
```

**scanf 함수**

- 입력 양식 제어 문자열에 의하여 입력 대상을 표준 입력 스트림에서 입력 받아 변수에 기억시키는 함수.
- 입력 양식의 변환 기호는 %로 시작한다. (%e, %g는 존재하지 않는다)  
%외의 문자열은 입력 시 구분 문자로 이용된다. 입력 구분 문자열이 특별히 지정되지 않은 경우는 공백문자가 쓰인다. 따라서 scanf에서 공백을 포함하는 문자열을 입력 할 수 없다.
- 서식 지정 문자열에서 기대한 입력 데이터와 실제로 입력된 입력 데이터 사이에 모순이 발생했을 때는 그 시점에서 scanf의 동작을 종료한다(입력의 실패). 예를 들어

```
scanf("%d", &num);           // 변수 num에 10진 숫자를 읽어 들인다.
```

에 대해 'a'라는 문자를 입력하면 입력 실패가 된다. 이 때 키보드에서 입력된 데이터는 영역에 그대로 남겨진다. 이 데이터는 다음의 scanf, getchar, gets 등의 입력 함수로 읽을 수 있다. 단 같은 기술의 scanf로 읽으면 재차 실패하므로 주의하여야 한다.

- 서식 지정 문자열

공백 : 제어문자열 안에 공백을 넣으면 입력 데이터 안의 공백문자를 읽고 버리게 한다.  
공백은 몇 개든 효과가 같다.  
일반 문자(% 제외) : 정해진 공백문자를 대신해 입력 구분 문자를 일반 문자로 지정한다.  
%로 시작하는 문자열 : 변환을 지시한다.

- 한 개의 scanf로 여러 개의 data를 입력할 수 있다. 이 때 입력 데이터는 공백, 탭, 개행으로 구분한다. 또한 특별한 문자로 구분하고 싶을 때는 서식 문자열 안에 그것을 기술한다. 예를 들어 'coma'를 넣으면 그것이 구분 기호가 된다.

지 정 서 식 예	입 력 데 이 터
scanf("%d %d %d", &a, &b, &c);	10 20 30
scanf("%d, %d, %d", &a, &b, &c);	10, 20, 30

scanf 함수 사용 시 주의점

- 데이터를 넣는 인수는 포인터이다. 그러나 포인터를 지정하지 않았을 때 예를 들어,

```
int a;  scanf("%d", a);
```

라고 해도 C compiler는 정상적으로 처리한다. 엄밀히 말하면 이것도 C의 규칙에 부합하기 때문이다. 그러나 프로그래머의 의도와는 다르게 되므로 이러한 오류를 일으키지 않으려면 사용자가 주의를 기울일 수밖에 없다.

[참고] 배열 명은 배열의 시작 번지를 가리키므로 배열 명 앞에는 &는 붙이지 않는다.

```
출력 : printf("format", variable constant);
입력 : scanf("format", variable address);
```



- 공백은 읽지 않는다. scanf의 "%s" 변환에서는 공백이 나오면 읽기를 중단한다. 따라서 "this is string"이라는 문자열은 읽을 수 없다. 이 입력에서는 「this」만 읽혀진다. 공백을 포함하는 문자열을 읽을 때는 gets를 이용한다. gets는 「Enter 키 입력」이 이루어 질 때까지 읽는다.
- 개행 문자가 남는다. scanf로 정상적으로 입력해도 최후의 개행 문자는 버퍼 안에 남기 때문에 이상 동작을 일으킬 때가 있다. 예를 들어 다음 프로그램은 의도한 대로 작동하지 않는다.

```
int main()
{
    int a, ch1, ch2;
    printf("number : "); scanf("%d", &a);           /* int형 입력 */
    printf("a = %d\n", a);
    printf("char1 : "); scanf("%c", &ch1);          /* char형 입력 */
    printf("ch1 = %c\n", ch1);
    printf("char2 : "); scanf("%c", &ch2);          /* char형 입력 */
    printf("ch2 = %c\n", ch2);
    return 0;
}
```

최초의 getchar가 실행되지 않고 비정상적으로 출력되었다. 이것은 최초의 수치를 입력할 때 입력된 개행 문자가 버퍼 안에 남아 있어 다음에 실행된 getchar가 그 개행 문자를 버퍼에서 읽어 들이기 때문이다. scanf나 getchar는 입력 데이터를 버퍼링하기 때문에 버퍼안이 비어 있지 않으면 키보드에서 새로 입력할 수 없다. 개행 문자를 읽어 버퍼가 비었기 때문에 두 번째 getchar의 경우 키보드에서 입력이 가능해졌다.

```
#include <stdio.h>

int main()
{
    char ch1, ch2;
    printf("한 문자 입력 : ");
    ch1 = getchar();           // scanf("%c", &ch1);
    fflush(stdin);             // getchar();

    printf("한 문자 입력 : ");
    ch2 = getchar();           // scanf("%c", &ch2);
    putchar(ch1);              // printf("%c", ch1);
    putchar(ch2);              // printf("%c", ch2);
    return 0;
}
```

```
int main()
{
    int num;
    long l_num;

    printf("8진수 입력 : ");
    scanf("%o", &num);
    printf("10진수 출력 : %d\n", num);

    printf("10진수 입력 : ");
    scanf("%d", &num);
    printf("10진수 출력 : %d\n", num);

    printf("16진수 입력 : ");
    scanf("%x", &num);
    printf("10진수 출력 : %d\n", num);

    printf("long 입력 : ");
    scanf("%ld", &l_num);
    printf("long 출력 : %ld\n", l_num);
    return 0;
}
```

```
int main()
{
    float f1, f2;
    printf("Input 2 real numbers >> ");
    scanf("%f %f", &f1, &f2);
    printf("addition      : %f\n", f1 + f2);
    printf("subtraction : %f\n", f1 - f2);
    return 0;
}
```

```
int main()
{
    float f;
    double d;

    printf("실수 입력 : ");
    scanf("%f", &f);
    printf("실수 출력 : %f\n", f);

    printf("double 입력 : ");
    scanf("%lf", &d);
    printf("double 출력 : %16lf\n", d);

    return 0;
}
```

```
int main()
{
    int num;
    printf("숫자를 하나 입력하시오 : ");
    scanf_s("%d", &num);
    printf("num = %d\n", num + 50);

    num = num + 50;
    printf("num = %d\n", num);
    return 0;
}
```

➡ 영문 2글자를 읽어 들여 한 줄에 하나씩 출력하는 프로그램을 작성하시오.

➡ 영문 5글자를 읽어 들여 역순으로 출력하는 프로그램을 작성하시오.

➡ 위의 문제를 getchar로 작성한 사람은 scanf로, scanf로 작성한 사람은 getchar로 바꾸어 작성하시오.

## 2. 연산자(operator)



- ◆ 연산자 우선순위와 결합성을 이해한다.
- ◆ 산술 연산자의 동작을 이해한다.
- ◆ 관계 연산자의 동작을 이해한다.
- ◆ 논리 연산자의 동작을 이해한다.
- ◆ 대입 연산자의 동작을 이해한다.
- ◆ 증가/감소 연산자의 동작을 이해한다.
- ◆ 비트 연산자의 동작을 이해한다.
- ◆ 조건 연산자의 동작을 이해한다.
- ◆ 기타(sizeof, typecast, 콤마) 연산자의 동작을 이해한다.

---

2.1 연산자 우선순위와 결합성

2.2 산술 연산자

2.3 관계 연산자

2.4 논리 연산자

2.5 대입 연산자

2.6 증가/감소 연산자

2.7 비트 연산자

2.8 조건 연산자

2.9 기타(sizeof, typecast, 콤마) 연산자

---

## 2. 연산자(operator)

---

### 2.1 연산자 우선순위와 결합성

2.2 산술 연산자

2.3 관계 연산자

2.4 논리 연산자

2.5 대입 연산자

2.6 증가/감소 연산자












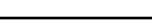
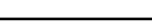


2.7 비트 연산자

2.8 조건 연산자

2.9 기타(sizeof, typecast, 콤마) 연산자

---



우선순위	형		연 산 자	결합성
1	일차식	primary	() [] . ->	
2	unary	단항	* & ! ++ -- (datatype) sizeof -	
3	binary	승법	* % /	
4		가법	+ -	
5		행 이동	<< >>	
6		관계	< > <= >=	
7		등가	== !=	
8		비트 곱	&	
9		비트 차	^	
10		비트 합		
11		논리 곱	&&	
12		논리 합		
13	trinary	조건	? :	
14	대입연산자	대입	= += -= *= %= /=	
15	순서연산자	순서	,	



### 연산자의 우선순위

한 행에 2개 이상의 연산자들이 나열되어 있을 때 먼저 처리해야 하는 연산자 간의 차례

가령,  $3 + 2 * 7$  의 계산에서 '+' 보다 '\*' 연산을 먼저 수행하는 것은 '+' 연산자 보다 '\*' 연산자의 우선순위가 높기 때문이다.

### 결합성

동일한 우선순위를 갖는 그룹내의 연산자들간에 연산자와 피연산자를 어느 방향으로 결합하는가를 결정하는 것

- ① 좌결합성(  ) : 연산자의 좌측에 있는 피연산과 결합하는 것
- ② 우결합성(  ) : 연산자의 우측에 있는 피연산과 결합하는 것

## 2. 연산자(operator)

---

2.1 연산자 우선순위와 결합성

2.2 산술 연산자

2.3 관계 연산자

2.4 논리 연산자

2.5 대입 연산자

2.6 증가/감소 연산자

2.7 비트 연산자

2.8 조건 연산자

2.9 기타(sizeof, typecast, 콤마) 연산자

---

```
#include <stdio.h>

int main()
{
    int a = 10, b = 3;    // 변수선언
    printf("%d + %d = %d\n", a, b, a + b);
    printf("%d - %d = %d\n", a, b, a - b);
    printf("%d * %d = %d\n", a, b, a * b);
    printf("%d / %d = %d\n", a, b, a / b);
    printf("%d %% %d = %d\n", a, b, a % b);
    return 0;
}
```

산술연산자

수치를 계산하는 것이며 결과로 수치를 얻는다. 정수 나눗셈을 하면 나머지는 버려지지만, 정수에 대해 % 연산자를 이용하면 나머지를 구할 수도 있다. 0으로 나누면 결과는 에러가 난다.

연산자	설 명	예
+	덧 셈	a = b + c;
-	뺄 셈	a = b - c;
*	곱 셈	a = b * c;
/	나눗셈	a = b / c;
%	나머지	a = b % c;

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    printf("%d\n", -4 + 6 * 5 + 3);
    printf("%d\n", 3 - 7 % 8 + 5);
    printf("%d\n", -5 * 3 % -2 / 4);
    printf("%d\n", (8 + 7) % 6 / 2);
    return 0;
}
```

## 2. 연산자(operator)

---

- 2.1 연산자 우선순위와 결합성
  - 2.2 산술 연산자
  - 2.3 관계 연산자
  - 2.4 논리 연산자
  - 2.5 대입 연산자
  - 2.6 증가/감소 연산자
  - 2.7 비트 연산자
  - 2.8 조건 연산자
  - 2.9 기타(sizeof, typecast, 콤마) 연산자
-

```
#include <stdio.h>

int main()
{
    int a = 15, b = 23;
    printf("%d < %d      : %d\n", a, b, a < b);
    printf("%d > %d      : %d\n", a, b, a > b);
    printf("%d == %d     : %d\n", a, b, a == b);
    printf("%d != %d     : %d\n", a, b, a != b);
    printf("%d + 8 >= %d : %d\n", a, b, a + 8 >= b);
    return 0;
}
```

관계 연산자

2개의 피연산자의 대소관계나 등치관계를 판정하여 진리 값을 얻는 것이다. 진리 값은 관계 연산식의 결과가 참이면 1, 거짓이면 0 이 된다.

연산자	설 명	예
<	작다	if(a < b)
<=	작거나 같다	if(a <= b)
>	크다	if(a > b)
>=	크거나 같아	if(a >= b)
==	같다	if(a == b)
!=	같지 않다	if(a != b)

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int i = 1, j = 2, k = -7;
    printf("%d\n", 'a' + 1 < j);
    printf("%d\n", -i - 5 * j >= k + 1);
    printf("%d\n", i + j + k == -2 * j);
    return 0;
}
```



## 2. 연산자(operator)

---

- 2.1 연산자 우선순위와 결합성
  - 2.2 산술 연산자
  - 2.3 관계 연산자
  - 2.4 논리 연산자
  - 2.5 대입 연산자
  - 2.6 증가/감소 연산자
  - 2.7 비트 연산자
  - 2.8 조건 연산자
  - 2.9 기타(sizeof, typecast, 콤마) 연산자
-

```
#include <stdio.h>

int main()
{
    char b = 'q';
    int i = 3, j = 4, k = 5;
    printf("!i : %d, !b : %d\n", !i, !b);
    printf("k>=3 && k<=7 : %d\n", k >= 3 && k <= 7);
    printf("i == 2 && j == 4 : %d\n", i == 2 && j == 4);
    printf("b == 'q' || b == 'Q' : %d\n", b == 'q' || b == 'Q');
    return 0;
}
```

논리 연산자

진리 값을 부정하거나 조건식을 여러 개 조합하는 것이다. 우선 순위는 !, &&, || 의 순서이다. 또한 &&와 ||는 왼쪽부터 오른쪽으로 평가되며 조건이 일치하면 거기서 조건 판정을 중지한다. 따라서 합치할 확률이 높은 조건을 맨 처음 기술하면 수행 속도가 빨라진다.

연산자	설 명	예
!	부 정	if(!a)
&&	논리 곱(그리고)	if(a == b && c == d)
	논리 합(또는)	if(a == b    c == d)

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int x = 1, y = 2, z = 4;
    printf("%d\n", x || y && z);
    printf("%d\n", x > y || z == y && x < z);
    printf("%d\n", x != y && y + 1 == !z + 4);
    printf("%d\n", y - 1 == 3 || y + 1 == 3);
    return 0;
}
```

## 2. 연산자(operator)

---

- 2.1 연산자 우선순위와 결합성
  - 2.2 산술 연산자
  - 2.3 관계 연산자
  - 2.4 논리 연산자
  - 2.5 대입 연산자
  - 2.6 증가/감소 연산자
  - 2.7 비트 연산자
  - 2.8 조건 연산자
  - 2.9 기타(sizeof, typecast, 콤마) 연산자
-

```
#include <stdio.h>

int main()
{
    int a = 1, b = 2, c = 3, d = 4;
    a /= 3 + 1;           // a = a / 4;
    b += a;               // b = b + a;
    c %= a + 1;          // c = c % (a + 1);
    d -= a * 2;          // d = d - (a * 2);
    printf("a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);
    return 0;
}
```

대입 연산자

단순 대입 연산자와 복합 대입 연산자의 두 종류로 나뉘어진다.  
단순 대입 연산자는 `a = 100;` 과 같이 일반적 대입 처리에 이용되는 것이다.  
복합 대입 연산자는 `a += 100;` ( $\rightarrow a = a + 100$ ) 과 같이 연산, 대입 처리를 하는 생략 기술된 형태이다.

단순 대입 연산자	=
복합 대입 연산자	+=   -=   *=   /=   %=   &=   ^=    =   <<=   >>=

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int i, j, k;
    i = j = k = 3;
    printf("%d\n", k %= i = 1 + k / 2);
    printf("%d\n", i += j += k);
    printf("%d\n", k = (i / 2) + (j / 2));
    return 0;
}
```

## 2. 연산자(operator)

---

- 2.1 연산자 우선순위와 결합성
  - 2.2 산술 연산자
  - 2.3 관계 연산자
  - 2.4 논리 연산자
  - 2.5 대입 연산자
  - 2.6 증가/감소 연산자
  - 2.7 비트 연산자
  - 2.8 조건 연산자
  - 2.9 기타(sizeof, typecast, 콤마) 연산자
-

```
#include <stdio.h>

int main()
{
    int a, b = 10;
    printf("%d\n", a = ++b);

    b = 10;
    printf("%d\n", a = b++);
    return 0;
}
```

```
b = b + 1;
a = b;
```

```
a = b;
b = b + 1;
```



## 전치형

## 후치형

의 두 가지 형태를 보유하고 있기 때문에 단순히 「 $a = a + 1$ 」의 대체 기능에 머물지 않는 강력한 기능을 갖고 있다.

변수  $a$ 의 값을 1 증가하려면 보통

```
a = a + 1;
```

이라고 하지만 증가 연산자를 이용하면

**++a; 또는 a++;**

라고 할 수 있다. 여기서 ++이 변수의 앞에 있는 것을 전치형, 뒤에 있는 것을 후치형이라고 한다. 전치형은 그 식 전체를 처리하기 전에 1증가(감소)를 행한다. 후치형은 그 식 전체를 처리하고 나서 1가산(감산)을 행한다.

연산자		설 명	일반 작성법
++	++a;	a를 1 증가 후 사용	a = a + 1;
	a++;	a를 사용 후 1 증가	
--	--a;	a를 1 감소 후 사용	a = a - 1;
	a--;	a를 사용 후 1 감소	

```
#include <stdio.h>

int main()
{
    int a = 30, b = 30;
    printf("%d, %d\n", a, b);
    printf("%d\n", ++a);
    printf("%d, %d\n", a, b);
    printf("%d\n", a++);
    printf("%d, %d\n", a, b);
    printf("%d\n", ++a + ++b);
    printf("%d, %d\n", a, b);
    printf("%d\n", b++ + b++);
    printf("%d, %d\n", a, b);
    return 0;
}
```

증감 연산자 사용상의 주의점

- ① 한 변수가 어떤 함수의 실매개 변수로 두 번 이상 쓰이면 절대로 이러한 변수에 증감연산자를 사용하지 않는다.
- ② 한 변수가 수식 내에서 두 번 이상 쓰이면 절대로 이러한 변수에 증감 연산자를 쓰지 않는다.

컴파일러는 연산자의 우선 순위에 의해서 식을 평가하는 것이 아니라 컴파일러 편의대로 식을 평가하기 때문에 위의 경우 해당되는 상황일 경우 증감 연산자의 사용보다는 식에 괄호를 적절히 사용하거나 임시 변수를 사용하는 것이 좋다.

옳은 예		틀린 예	
prefix	<code>++count</code> <code>count++</code>	<code>777++</code>	<i>/* 상수는 증가될 수 없음 */</i>
postfix	<code>--count</code> <code>count--</code>	<code>++ (count/index)</code>	<i>/* 수식은 증가될 수 없음 */</i>

```
#include <stdio.h>

int main()
{
    int a = 10, b = 2;
    a += b++;
    printf("a = %d, b = %d\n", a, b);
    {
        int a = 2;
        a += b;    b -= a;
        printf("a = %d, b = %d\n", a, b);
        {
            int b = 5;
            b *= a + 1;    ++a;
            printf("a = %d, b = %d\n", a, b);
        }
        printf("a = %d, b = %d\n", a, b);
    }
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

```
int main()
{
    int a = 3, b = 2, c = 1;
    printf("%d\n", -(--a));
    printf("%d\n", b - (--c));
    printf("%d\n", a++ - b);
    printf("%d\n", a++ / ++b * --c);
    printf("%d\n", ++c / b++ * --a);
    return 0;
}
```

```
int main()
{
    printf("%d\n", -2 + 3 * 4 % 5);
    printf("%d\n", 2 + -3 * 4 % -5);
    printf("%d\n", (2 - 3) * -4 % -5);
    printf("%d\n", -4 % -5);
    printf("%d\n", 12 % -5);
    return 0;
}
```

## 2. 연산자(operator)

---

- 2.1 연산자 우선순위와 결합성
  - 2.2 산술 연산자
  - 2.3 관계 연산자
  - 2.4 논리 연산자
  - 2.5 대입 연산자
  - 2.6 증가/감소 연산자
  - 2.7 비트 연산자
  - 2.8 조건 연산자
  - 2.9 기타(sizeof, typecast, 콤마) 연산자
-

```
#include <stdio.h>

int main()
{
    int a = 0x5555, b = 0x00ff;
    printf("  a = %x,   b = %x\n", a, b);
    printf("~a = %x, ~b = %x\n", ~a, ~b);
    printf("a & b = %x\n", a & b);
    printf("a | b = %x\n", a | b);
    printf("a ^ b = %x\n", a ^ b);
    return 0;
}
```

비트 연산자

비트 단위로 데이터에 대해 논리 처리, 반전, shift 등의 조작을 하는 것이다. 이 연산자를 이용하면 어셈블러에 가까운 처리를 할 수 있다.

비트 연산자의 대상은 정수여야 하며 float 이나 double 형에 비트 연산을 할 수는 없다.

연산자	설 명	예
~	비트의 반전(1의 보수)	<code>a = ~a;</code>
&	비트 단위 AND (bit 곱)	<code>a = b &amp; 0x7FFF;</code>
	비트 단위 OR (bit 합)	<code>a = b   0x8000;</code>
^	비트 단위 XOR (bit 차)	<code>a = b ^ 0x000F;</code>
<<	왼쪽 shift	<code>a = a &lt;&lt; 2;</code>
>>	오른쪽 shift	<code>a = a &gt;&gt; 2;</code>

- & (and) : 양쪽 비트가 1일 때만 결과가 1이 된다.
- | (or) : 적어도 하나의 비트가 1이면 결과가 1이 된다.  
양쪽 비트가 모두 0일 때만 결과가 0이 된다.
- ^ (xor) : 양쪽 비트가 다를 때만 결과가 1, 같으면 0이 된다.
- ~(보수) : 비트의 반전



```
#include <stdio.h>

int main()
{
    int a = 30;
    printf("a << 1 = %d\n", a << 1);
    printf("a << 2 = %d\n", a << 2);
    printf("a << 3 = %d\n", a << 3);
    printf("a << 4 = %d\n", a << 4);
    return 0;
}
```

### shift 연산자

변수의 비트를 왼쪽 또는 오른쪽으로 shift 시키는 것이다.

shift 대상은 정수여야 하고, shift 의 방향과 부호의 유무에 의해 동작이 달라진다.

- 왼쪽 shift 연산자

왼쪽 shift 를 하면 왼쪽에서 벗어난 비트는 버려지고 오른쪽에서 0 이 shift-in 된다.

왼쪽으로 1 비트 shift 할 때마다 값은 2 배가 된다.

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int a = 5, b = 3, c = 7, d = 6;
    printf("%d\n", a & b);
    printf("%d\n", c ^ d);
    printf("%d\n", a | d);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int a = 30;
    printf("a >> 1 = %d\n", a >> 1);
    printf("a >> 2 = %d\n", a >> 2);
    printf("a >> 3 = %d\n", a >> 3);
    printf("a >> 4 = %d\n", a >> 4);
    return 0;
}
```

- 오른쪽 shift 연산자

부호가 없는 정수의 오른쪽 shift는 왼쪽 shift의 반대 동작이 된다. 즉 오른쪽에서 벗어난 비트는 버려지고 왼쪽에서 0이 shift-in 된다. 단 음수의 오른쪽 shift는 다른 방식으로 행해진다.

오른쪽으로 1 비트 shift 할 때마다 값은 1/2 배가 된다.

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int a = 03, b = 02, c = 01;
    printf("%d\n", a | b & c);
    printf("%d\n", a | b & ~c);
    printf("%d\n", a ^ b & ~c);

    a = 1, b = -1;
    printf("%d\n", a << 3);
    printf("%d\n", b << 3);
    return 0;
}
```

## 2. 연산자(operator)

---

2.1 연산자 우선순위와 결합성

2.2 산술 연산자

2.3 관계 연산자

2.4 논리 연산자

2.5 대입 연산자

2.6 증가/감소 연산자

2.7 비트 연산자

2.8 조건 연산자

2.9 기타(sizeof, typecast, 콤마) 연산자

---

```
#include <stdio.h>

int main()
{
    int x = 1, y = 2, z = 3;
    printf("%d\n", x > y ? x : y);
    printf("%d\n", x < y ? x++ : y++);
    z += x + y ? x++ : y++;
    printf("%d, %d, %d\n", x, y, z);
    return 0;
}
```

## 조건 연산자

조건식 ? 식1 : 식2

조건식을 이용해 연산을 실행하므로 조건 연산자라고 하지만, 3개의 항목을 평가하기 때문에 삼항 연산자로 불리는 경우도 있다.

조건식이 참이면 식1, 그렇지 않으면 식2를 값으로 한다.

조건 연산자는 if문에 의한 조건 처리의 대체로서 간결하게 기술할 수 있다. 또한 식의 값을 결과로 취하므로 값을 나타내는 함수와 같은 독특한 용법이라 할 수 있다.

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int i = 1, j = 2, k = 3;
    printf("%d\n", i == j ? i : j);
    printf("%d\n", k % 3 == 0 ? (i = 5) : (i = 10));
    printf("%d\n", i = k % 3 == 0 ? 5 : 10);
    return 0;
}
```

## 2. 연산자(operator)

---

- 2.1 연산자 우선순위와 결합성
  - 2.2 산술 연산자
  - 2.3 관계 연산자
  - 2.4 논리 연산자
  - 2.5 대입 연산자
  - 2.6 증가/감소 연산자
  - 2.7 비트 연산자
  - 2.8 조건 연산자
  - 2.9 기타(sizeof, typecast, 콤마) 연산자
-



```
#include <stdio.h>

int main()
{
    int a = 3, b = 5;
    printf("char      : %d byte\n", sizeof(char));
    printf("int       : %d byte\n", sizeof(int));
    printf("float      : %d byte\n", sizeof(float));
    printf("double     : %d byte\n", sizeof(double));
    printf("sizeof a    : %d byte\n", sizeof(a));
    printf("sizeof a + b : %d byte\n", sizeof(a + b));
    return 0;
}
```

## sizeof 연산자

sizeof (식)  
sizeof (datatype)

인수의 크기를 byte 수로 보고한다. 주어진 대상이 식일 때는 식의 값, 형 표기 일 때는 그 형이 갖는 어떤 대상의 값을 기억하는데 필요한 byte 수를 수치로 표현한다. 시스템에 따라 다른 결과 값이 나올 수 있다.

sizeof 연산자는 대응하는 식의 크기를 해석하지만 그 식을「수행」하지는 않는다.

```
int main()
{
    int a;
    long b = 100;
    a = sizeof(++b);           // ++b는 수행되지 못한다.
    printf("a = %d, b = %ld\n", a, b); // 출력은 a = 4, b = 100
    return 0;
}
```

라는 기술에서 b의 값이 101이 되지 않는다. 따라서 sizeof와 함께「부가적인 작용을 기대하는 식」을 쓸 수는 없다.

```
#include <stdio.h>

int main()
{
    int a = 5, b = 2;
    double f;
    printf("%.2f\n", f = a / b);
    printf("%.2f\n", (double)f = a / b);
    printf("%.2f\n", f = (double)a / b);
    printf("%.2f\n", f = (double)(a / b));
    return 0;
}
```

### typedef 연산자

(형) 단항식

식의 결과를 일시적으로 지정한 데이터 형으로 변경하는 것이다.

implicit type conversion 은 자동적으로 이루어지지만 의도대로 explicit type conversion 을 하기 위한 경우에 "cast" 연산자를 사용한다.

- 묵시적 형 변환 (implicit type conversion)

데이터 타입이 서로 다른 피연산들끼리의 혼합 연산 시 기억장소의 크기가 작은 쪽에서 큰 쪽으로 변환되어 연산

**char < int < unsigned int < long < unsigned long < float < double**

- 명시적 형 변환 (explicit type conversion)

데이터 타입을 강제로 변환

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    printf("%c\n", (char)(65 + 32));
    printf("%d\n", (int)((int)3.14 + (float)3 + 1));
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int a, b;
    a = (b = 0, b + 2); // b = 0, a = b + 2;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

### 콤마 연산자

열거된 순서대로 수행되고, 여러 번의 처리를 하나의 식에서 행하면 효과가 있을 때 이용한다.

➡ 사용자 입력으로 두 실수를 입력 받아 평균을 구하는 프로그램을 작성하시오.

➡ 사용자 입력으로 초(second)를 입력 받아 몇 분 몇 초인가를 계산하는 프로그램을 작성하시오. 산술 연산자와 /와 %를 이용하여 연산하시오.

➡ 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int a, b;
    a = (7 % 5);
    b = (a = a + 5) + 6;
    printf("%d, %d\n", a, b);
    return 0;
}
```

```
int main()
{
    int a = 10, b = 0;
    b = (a < 3) ? 5 : 8;
    printf("%d, %d\n", a, b);
    a = (b = 10, b++);
    printf("%d, %d\n", a, b);
    return 0;
}
```

➡ 사용자 입력으로 원의 반지름을 입력 받은 후, 원의 넓이를 구하는 프로그램을 작성 하시오.

➡ 사용자 입력으로 4개의 정수를 입력 받아 그 중 최대값을 구하는 program을 조건 연 산자를 사용하여 작성하시오.



➡  $a = 20, b = 50$  일 때 각각을 2로 나눈 값을  $a, b$ 의 값으로 하고 그 합을 구하는 program을 작성하시오.

➡  $a = 710$  인 경우 오른쪽으로 4 bit 이동한 값을 구하는 program을 작성하시오.

➡  $x = 0xabcd$  일 때  $!x$ 의 값을 구하는 program을 작성하시오.

➡  $x = 0xff00$  일 때  $x$ 의 1의 보수를 구하는 program을 작성하시오.

### 3. 제어문(control statement)



- ◆ 제어문의 종류 및 특징을 이해한다.
- ◆ if문의 동작을 이해한다.
- ◆ switch case문의 동작을 이해한다.
- ◆ while 문의 동작을 이해한다.
- ◆ do while 문의 동작을 이해한다.
- ◆ for 문의 동작을 이해한다.
- ◆ break, continue 문의 동작을 이해한다.

---

### 3.1 제어문의 종류 및 특징

#### 3.2 if 문

#### 3.3 switch case 문

#### 3.4 while 문

#### 3.5 do while 문

#### 3.6 for 문

#### 3.7 break, continue 문

---

# 3. 제어문(control statement)

---

## 3.1 제어문의 종류 및 특징

### 3.2 if 문

### 3.3 switch case 문

### 3.4 while 문

### 3.5 do while 문

### 3.6 for 문

### 3.7 break, continue 문

---

➡ 컴퓨터의 각 실행동작을 규정하는 것

➡ 형태별 문의 종류

- 공문 / 단문 / 복문

➡ 기능별 문의 종류

- 선언문 / 치환문 / 제어문 / 함수호출문

### 형태별 문의 종류

- 공문 : ';'으로만 되어있는 것으로 아무것도 실행하지 않는다.
- 단문 : 하나의 문장으로만 구성된 문
- 복문 : 한 개 이상의 문들의 나열이 '{'와 '}'로 둘러싸인 문장

### 기능별 문의 종류

- 선언문
  - ① 함수의 외부에서 외부변수의 선언이나 정의를 하는 문
  - ② 함수의 서두에서 매개변수의 형 선언을 하는 문
- 치환문 : 대상 참조식에 값을 지정하는 문
- 제어문 : 프로그램 제어의 흐름에 관한 문
  - ① 조건문 : if문, if-else문, nested if문, 다중 if문
  - ② 선택문 : switch-case문
  - ③ 반복문 : while문, do-while문, for문
  - ④ 보조 제어문 : break문, continue문, return문, goto문
- 함수 호출문 : 이미 정의된 함수가 수행되도록 함수를 호출하는 문

# 3. 제어문(control statement)

---

3.1 제어문의 종류 및 특징

3.2 if 문

3.3 switch case 문

3.4 while 문

3.5 do while 문

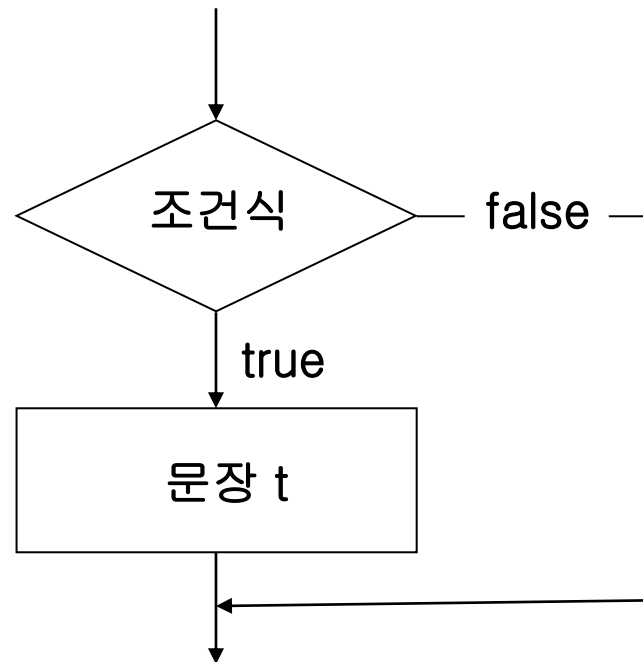
3.6 for 문

3.7 break, continue 문

---

- ➡ 조건식이 참이면 문장 t를 실행하고 거짓이면 실행하지 않는다.
- ➡ 처리할 문장이 한 개 이상일 때는 { }를 이용한다.

```
if(조건식)
{
    문장 t;
}
```





**if문**

만약 조건이 만족되면 {}안의 하나 이상의 문장들이 실행이 된다.

대부분의 조건식은 보통 관계식이나 논리식이 사용이 된다.

C 는 0이 아닌 모든 값은 참으로 인식하고, 0만 거짓으로 인식하기 때문에 조건식의 값이 0 이면 논리적 거짓, 0 이 아니면 논리적 참으로 간주한다.

```
if(a == 10)
```

조건식 :  $a == 10$

참 : a의 값이 10일 때 「참」이 된다.

거짓 : a의 값이 10이 아닐 때 「거짓」이 된다.

```
if(a)
```

조건식 :  $a$

참 : a의 값이 0 이 아닐 때 「참」이 된다.

거짓 : a의 값이 0 일 때 「거짓」이 된다.

```
#include <stdio.h>

int main()
{
    int x = 4, y = 5, z = 6;
    if (x < y && y < z)
        y += 1;
    printf("%d, %d, %d\n", x, y, z);
    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int x = 0, y = 10;
    if (x != 0 && x < y)
        x = 10;
    printf("%d, %d\n", x, y);
    return 0;
}
```

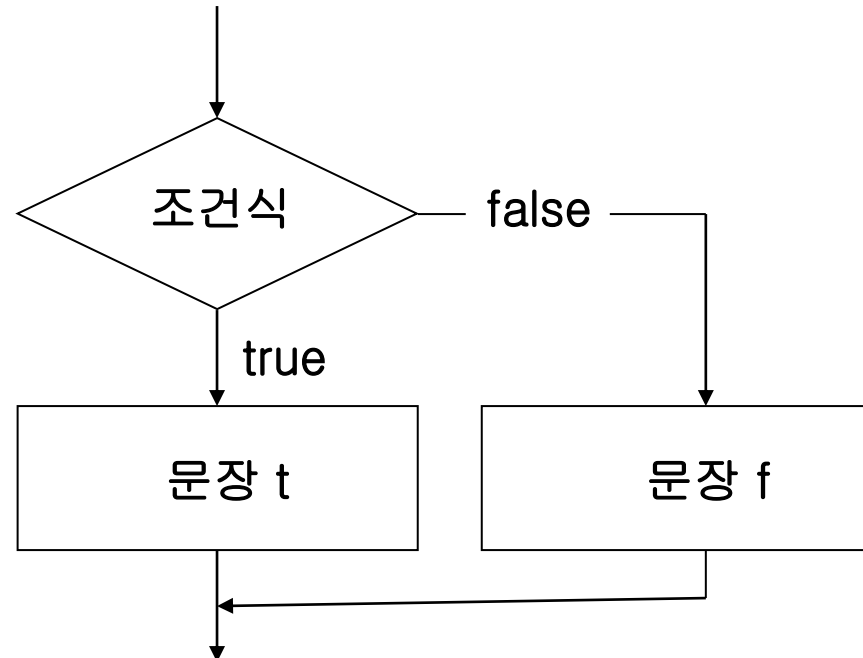
- 다음 프로그램의 오류를 지적하십시오.

```
int main()
{
    int a = 10;
    if (a = 100)
        printf("a is 100\n");
    return 0;
}
```

'='와 '=='는 완전히 구분하여 사용된다. C compiler는 이것을 '올바른 프로그램'이라고 간주하기 때문에 compile error로 처리되지 않는다. 단, 의도적으로 '='를 사용하여 프로그램을 작성하기도 한다.

➡ 조건식이 참이면 문장 t를 수행하고 거짓이면 문장 f를 수행한다.

```
if(조건식)
{
    문장 t;
}
else
{
    문장 f;
}
```



**if-else문**

else절은 조건을 표현할 수 없는데, 이는 match되는 if절의 조건이 만족되지 않으면 else절의 문장이 수행되기 때문이다.

else절은 위치시킬 문장이 없다면 생략할 수 있다.

- 입력한 정수 값이 0 이상이면 plus, 그렇지 않으면 minus라고 출력하라.

```
#include <stdio.h>

int main()
{
    int num;
    scanf("%d", &num);
    if (num >= 0)    printf("plus\n");
    else            printf("minus\n");
    return 0;
}
```

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

```
int main()
```

```
{
```

```
    int a, x = 3, y = 4;
```

```
    if (x < y)
```

```
        a = 5;
```

```
    else
```

```
        a = 7;
```

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

```
    return 0;
```

```
}
```



```
a = (x > y) ? x : y;
```

다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int a, x = 3, y = 4;
    if (x == y)
        a = 5;
    else
        a = 7;
    printf("a = %d\n", a);
    return 0;
}
```

다음과 같이 명령문이 실행된 후에 변수 num에 저장되는 값은 무엇인가?

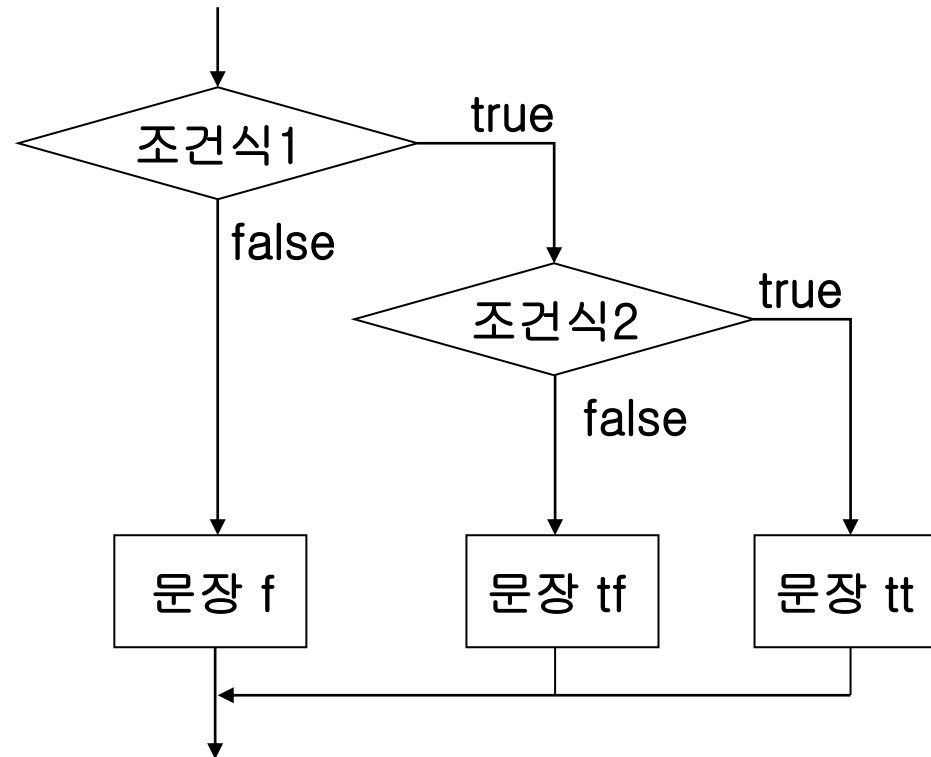
```
#include <stdio.h>

int main()
{
    int num, data = 0;
    if (!data)
        num = 10;
    else
        num = 1;
    return 0;
}
```

- ➡ if문 안에 if문이 중첩되는 형태이다.
- ➡ nested if-else의 else는 가장 가까운 if와 match된다.

```

if(조건식1)
{
    if(조건식2) 문장 tt;
    else 문장 tf;
}
else
{
    문장 f;
}
    
```





### nested if-else문

일치할 확률이 높은 조건을 먼저 제시하는 편이 처리가 빨라질 수 있는데 조건이 일치하면 그 부분에서 if문 전체의 처리를 중단하기 때문이다.

- 입력한 정수 값이 0 이면 zero, 0 이상이면 plus, 그렇지 않으면 minus라고 출력하라.

```
int main()
{
    int num;
    scanf("%d", &num);
    if (num >= 0)
    {
        if (num == 0)
            printf("zero\n");
        else
            printf("plus\n");
    }
    else
        printf("minus\n");
    return 0;
}
```

or

```
int main()
{
    int num;
    scanf("%d", &num);
    if (num > 0)
        printf("plus\n");
    else
    {
        if (num == 0)
            printf("zero\n");
        else
            printf("minus\n");
    }
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int x = 4, y = 5, z = 6;
    if (x < y) {
        if (y < z)    y += 1;
        else        y -= 1;
    }
    else
        y = 0;
    printf("%d, %d, %d\n", x, y, z);
    return 0;
}
```

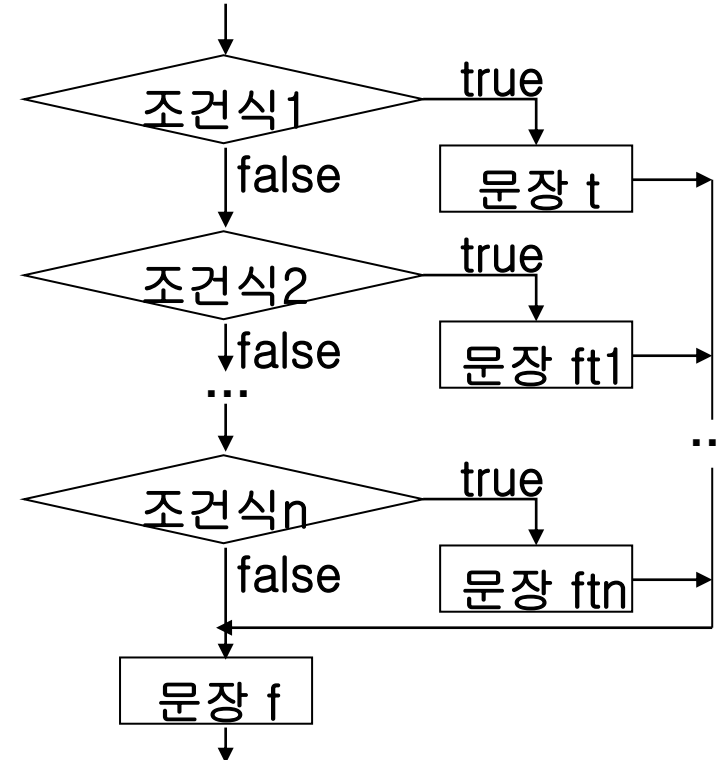
```
#include <stdio.h>

int main()
{
    int x, y, z = 7, w = 6;
    if (z > 0) {
        if (w > 10) {
            x = 20;
            y = x + 10;
        }
        else {
            x = 30;
            y = x + 10;
        }
    }
    else {
        x = 30; y = x;
    }
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

- ➡ 조건식 1이 참이면 문장 t를 실행한다.
- ➡ 그렇지 않고 조건식 2가 참이면 문장 ft1를 실행한다.
- ➡ 그렇지 않고 조건식 n이 참이면 문장 ftn을 실행한다.
- ➡ 어떤 조건에도 만족하지 않으면 문장 f를 실행한다

```

if(조건식1)
    문장 t;
else if(조건식 2)
    문장 ft1;
    ...
else if(조건식 n)
    문장 ftn;
else
    문장 f;
    
```



### 다중 if-else문

역시 일치할 확률이 높은 조건을 먼저 제시하는 편이 처리가 빨라질 수 있는데 조건이 일치하면 그 부분에서 if문 전체의 처리를 중단하기 때문이다.

- 입력한 정수 값이 0 이면 zero, 0 이상이면 plus, 그렇지 않으면 minus라고 출력하라.

```
#include <stdio.h>

int main()
{
    int num;
    scanf("%d", &num);

    if (num == 0)
        printf("zero\n");
    else if (num > 0)
        printf("plus\n");
    else
        printf("minus\n");
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int a, x = 5, y = 4, z = 6;

    if (x < y && y < z)
        a = 5;
    else if (x < z && y < z)
        a = 6;
    else
        a = 7;
    printf("a = %d\n", a);
    return 0;
}
```

- 입력한 정수 값이 0 이면 zero, 0 이상이면 plus, 그렇지 않으면 minus라고 출력하라.

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

```
int main()
```

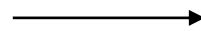
```
{
```

```
    int a, x = 20, y = 30;
```

```
    if (x > y) a = 3;
```

```
    else if (x < y) a = 4;
```

```
    else a = 5;
```



```
a = (x > y) ? 3 : (x < y) ? 4 : 5;
```

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

```
    return 0;
```

```
}
```

```
#include <stdio.h>

int main()
{
    int age;
    printf("Enter the your age : ");
    scanf_s("%d", &age);

    if (age >= 60) printf("60대 이상\n");
    else if (age >= 50) printf("50대 \n ");
    else if (age >= 40) printf("40대 \n ");
    else if (age >= 30) printf("30대 \n ");
    else if (age >= 20) printf("20대 \n ");
    else if (age >= 10) printf("10대 \n ");
    else if (age >= 0) printf("유소년 \n ");
    else printf("해당사항 없습니다 \n ");

    return 0;
}
```



- 복잡한 정수 판단식

예를 들어, 변수 num이 0 과 100 사이에 들어가는가를 판단하는 조건식으로

```
0 <= num <= 100
```

라고 기술하면 안되고 다음과 같이 작성해야 한다.

```
0 <= num && num <= 100
```

문자 변수 ch의 내용이 알파벳인지를 판단하는 조건식은 다음과 같다.

```
('A' <= ch && ch <= 'Z') || ('a' <= ch && ch <= 'z')
```

즉, &&는 연산자 AND에 해당하고, ||는 연산자 OR에 해당한다.

# 3. 제어문(control statement)

---

3.1 제어문의 종류 및 특징

3.2 if 문

3.3 switch case 문

3.4 while 문

3.5 do while 문

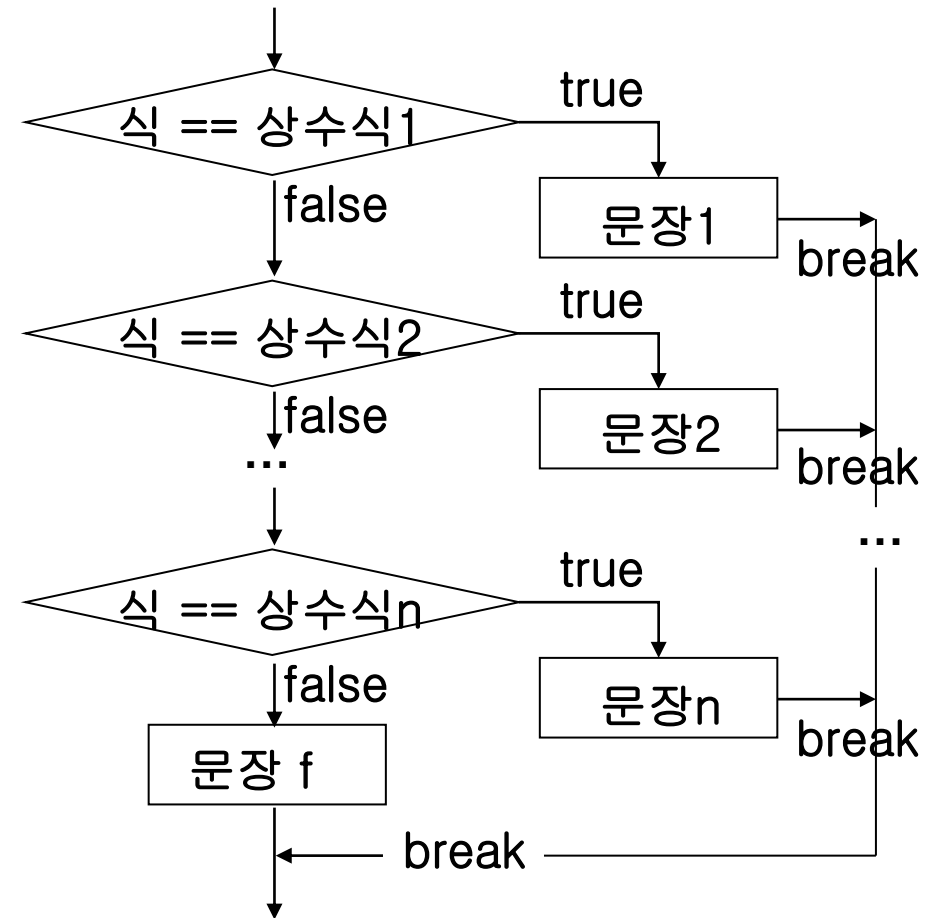
3.6 for 문

3.7 break, continue 문

---

- 식의 값에 따라 여러 방향으로 분기될 수 있다.

```
switch(식)
{
    case 상수식1 : 문장1;
                  break;
    case 상수식2 : 문장2;
                  break;
    ...
    case 상수식n : 문장n;
                  break;
    default : 문장;
             break;
}
```



**switch문**

switch문은 (식)의 값이 일치하는 곳의 case 문으로 제어가 옮겨간 후에는 그 이후 아래의 문을 모두 순차적으로 실행한다. 따라서 다음의 case 문으로 넘어가서는 안될 곳에는 필요 시마다 "break;"로 switch문을 벗어나게 한다.

다음은 switch문을 사용할 때의 주의점이다.

- 식은 결과가 반드시 정수적 표현이어야 한다.
- 식의 값과 어떤 case 다음의 식의 결과의 값을 비교하여 서로 일치하는 곳의 case 라벨 문장을 수행한다.
- 그 처리는 break 문을 만나거나 switch문이 끝나면 종료된다.
- 각 case마다 break 문을 쓰는 것이 일반적이지만 반드시 그래야 하는 것은 아니다. 의도적으로 break 문을 쓰지 않을 수도 있다.
- 식의 결과값과 일치되는 case 문이 없을 때는 default 로 분기한다.
- default는 필요 없으면 생략할 수 있다.
- case의 기능은 단순한 라벨이다.
- 하나의 실행 그룹에 여러 개의 case를 붙여도 된다.
- 같은 값을 갖는 2개의 case 라벨을 사용해서는 안 된다.

- 정수를 입력 받아 1을 입력 받으면 "one", 2를 입력 받으면 "two", 3을 입력 받으면 "three", 다른 수를 입력 받으면 "error"라고 출력하라.

```
#include <stdio.h>

int main()
{
    int a;
    scanf("%d", &a);
    if (a == 1)
        printf("one\n");
    else if (a == 2)
        printf("two\n");
    else if (a == 3)
        printf("three\n");
    else
        printf("error\n");
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int a;
    scanf("%d", &a);
    switch (a)
    {
        case 1: printf("one\n");
                break;
        case 2: printf("two\n");
                break;
        case 3: printf("three\n");
                break;
        default: printf("error\n");
    }
    return 0;
}
```

- break의 의미를 파악해보자.

```
#include <stdio.h>

int main()
{
    int a;
    scanf("%d", &a);
    switch (a)
    {
        case 1: printf("one\n");
        case 2: printf("two\n");
        case 3: printf("three\n");
        default: printf("error\n");
    }
    return 0;
}
```

a == 1 이면  
출력 : onetwothreeerror

a == 2 이면  
출력 : twothreeerror

a == 3 이면  
출력 : threeerror

이외  
출력 : error

```
#include <stdio.h>

int main()
{
    int x, y;
    char ch;
    scanf("%d %d", &x, &y);
    switch (x + y) {
        case 100: ch = 'A'; break;
        case 200: ch = 'B'; break;
        case 300: ch = 'C'; break;
        default: ch = 'Z';
    }
    printf("%c\n", ch);
    return 0;
}
```

- 다음과 같이 명령문이 실행된 후에 변수 num에 저장되는 값은?

```
#include <stdio.h>

int main()
{
    int num, data = 1;
    data += 2;

    switch (data) {
        case 1:  num = 60;
        case 2:  num = 70;
        case 3:  num = 80;
        case 4:  num = 90;
        default: num = 100;
    }

    printf("%d\n", num);
    return 0;
}
```



```
#include <stdio.h>

int main()
{
    int age;
    printf("Enter the your age : ");
    scanf("%d", &age);
    switch (age / 10) {
        case 6: printf("60대 \n"); break;
        case 5: printf("50대 \n "); break;
        case 4: printf("40대 \n "); break;
        case 3: printf("30대 \n "); break;
        case 2: printf("20대 \n "); break;
        case 1: printf("10대 \n "); break;
        case 0: printf("유소년 \n "); break;
        default: printf("해당사항 없습니다 \n ");
    }

    return 0;
}
```

- 식과 결과값은 정수 값(상수, 변수, 수식)을 결과로 하는 것이어야 한다.  
종래의 C에서는 int형을 결과로 하는 것으로 제한되어 있었지만,  
ANSI C에서는 정수면 되므로 short int 든 long int든 관계없다.  
그러나 float이나 double등의 부동 소수점형을 이용할 수는 없다.
- case 부분에는 정수형을 쓴다. 부동소수점수, 변수, 문자열은 이용할 수 없다.
- 다음 프로그램의 오류를 지적하시오.

```
int main()
{
    double a;
    scanf("%lf", &a);
    switch (a) {
        case 1: printf("1111\n");
        break;
        case 3: printf("3333\n");
        break;
        default: printf("9999\n");
    }
    return 0;
}
```

# 3. 제어문(control statement)

---

3.1 제어문의 종류 및 특징

3.2 if 문

3.3 switch case 문

3.4 while 문

3.5 do while 문

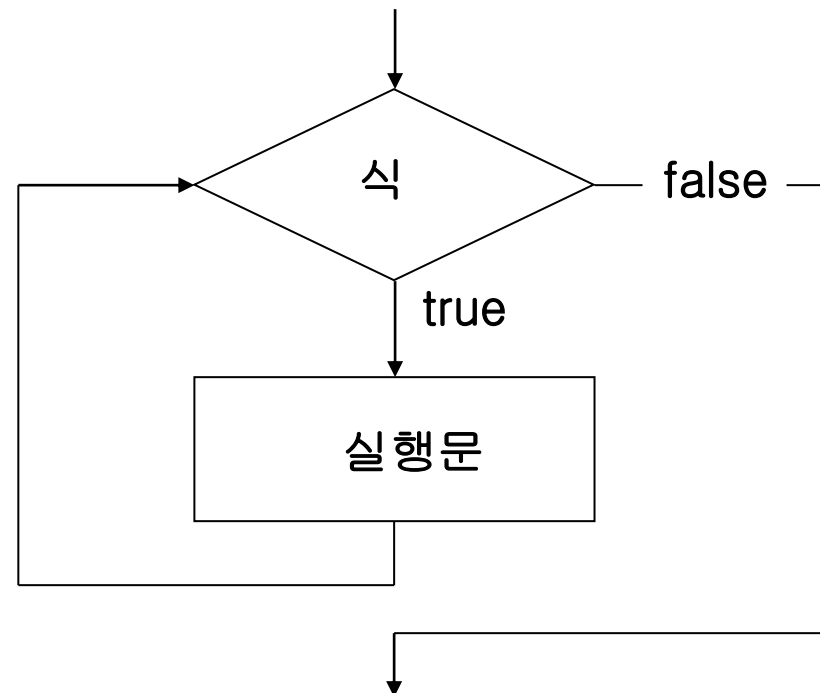
3.6 for 문

3.7 break, continue 문

---

- 조건이 만족되는 동안 반복하는 반복문이다.
- 먼저 수식을 평가한 후 결과가 참인 동안만 반복 문장을 수행한다.

```
while(식)  
{  
    실행문;  
}
```



**while문 ( 비교→수행 : 거짓이면 한번도 수행 안 함 )**

while문은 조건이 처음에 있으므로 처음 실행 시 조건이 참이면 루프 처리를 행한다. while문의 조건식은 「다음 루프를 행할 것이냐」이며 식이 처음부터 거짓일 경우는 한 번도 실행되지 않는다.

반복 수행할 문장 내부에 반복을 계속할 것인지 판단하기 위해 조건식의 결과값이 변경될 수 있게 하는 문장을 기술한다. 또한 while문의 조건식 차제에 식의 값이 변경될 수 있도록 기술할 수도 있다.

- 식이 참일 때 루프 처리를 반복한다.
- 식이 처음부터 거짓일 때는 실행문을 한번도 실행하지 않는다.
- 반복 수행 문장이 여러 개일 경우는 { }을 이용한다.

```
#include <stdio.h>

int main()
{
    int a = 1;
    while (a <= 10) {
        printf("hello\n");
        a++;
    }

    return 0;
}
```

```
int main()
{
    int a = 1;
    while (a > 5)
    {
        printf("%d\n", a);
        a++;
    }
    return 0;
}
```

- 반복 수행 문장이 여러 개일 경우는 {}을 이용한다.

```
int main()
{
    int data = 1, num = 1;
    while (num < 6)
        data += (num++);
    printf("%d, %d\n", data, num);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int age, level = 0;
    printf("Enter the your age : ");
    scanf("%d", &age);

    while (age >= 10)
    {
        age = age - 10;
        level = level + 1;
    }
    printf("%d대\n", level * 10);
    return 0;
}
```

# 3. 제어문(control statement)

---

3.1 제어문의 종류 및 특징

3.2 if 문

3.3 switch case 문

3.4 while 문

3.5 do while 문

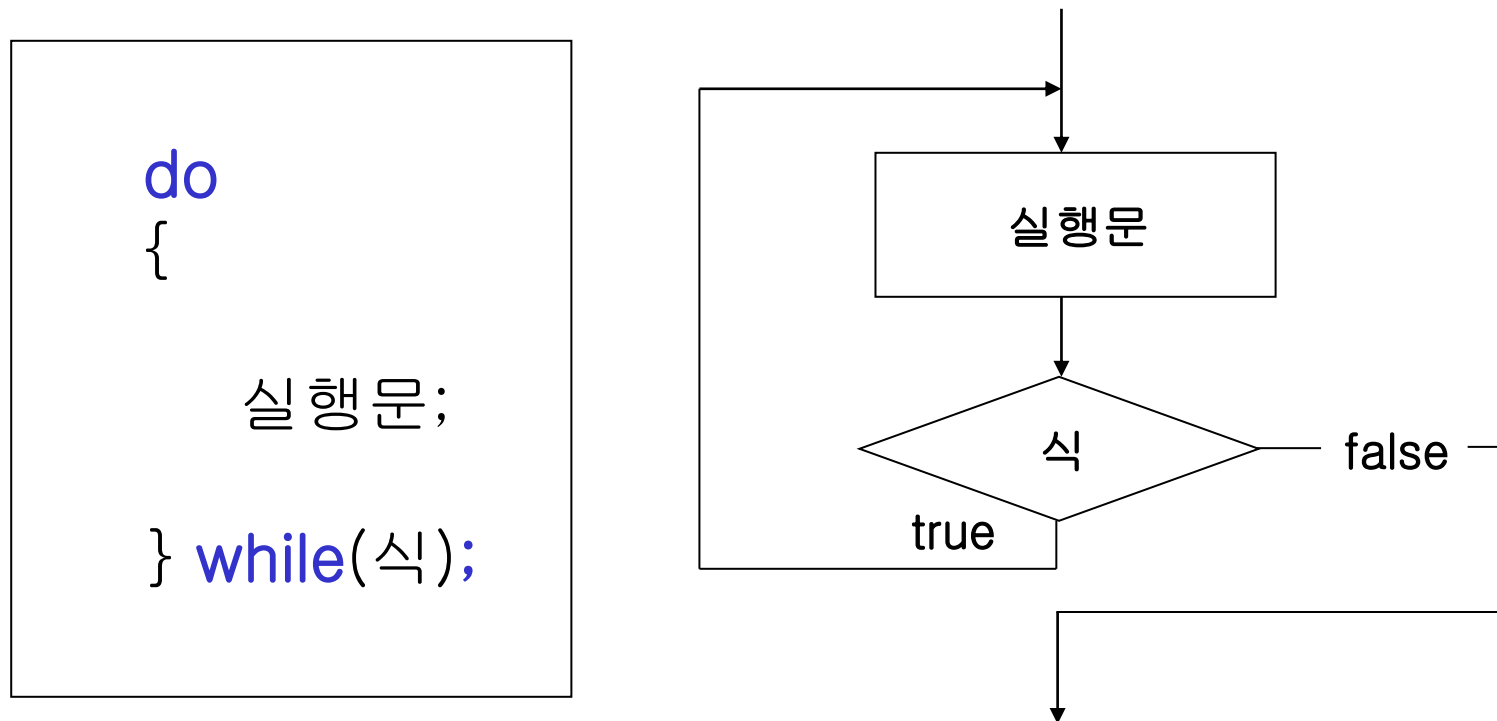
3.6 for 문

3.7 break, continue 문

---



- 조건이 만족되는 동안 반복하는 while문과 유사하다.
- 단, 실행문을 한번 수행한 후, 조건을 검사하여, 결과가 참인 동안만 반복 문장을 수행한다.



**do-while문 ( 수행 → 비교 : 거짓이라도 한번은 수행함 )**

do-while문은 while문과 마찬가지로 반복 루프를 행하지만, 반복을 계속할지 묻는 시점이 다르다. do-while문은 반복 수행할 문장을 실행한 다음 조건식을 판단하므로 while문과 달리 최소한 한 번은 do문 내의 문장을 실행한다.

- 식이 참일 때 루프 처리를 반복한다.
- 식이 처음부터 거짓일 때도 실행문을 적어도 한 번은 실행한다.
- 실행문이 여러 개일 때는 { }을 이용한다.
- 마지막에 ';'이 필요하다.

```
#include <stdio.h>

int main()
{
    int a = 1;
    do {
        printf("hello\n");
        a++;
    } while (a <= 10);
    return 0;
}
```

```
int main()
{
    int a = 1;
    do
    {
        printf("%d\n", a);
        a++;
    } while (a > 5);
    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int data = 7, num = 6;
    do {
        data += 2;
        ++num;
    } while (num > 10);
    printf("%d, %d\n", data, num);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int age, level = 0;
    printf("Enter the your age : ");
    scanf("%d", &age);
    do
    {
        age = age - 10;
        level++;
    } while (age >= 10);
    printf("%d대\n", level * 10);
    return 0;
}
```

# 3. 제어문(control statement)

---

3.1 제어문의 종류 및 특징

3.2 if 문

3.3 switch case 문

3.4 while 문

3.5 do while 문

3.6 for 문

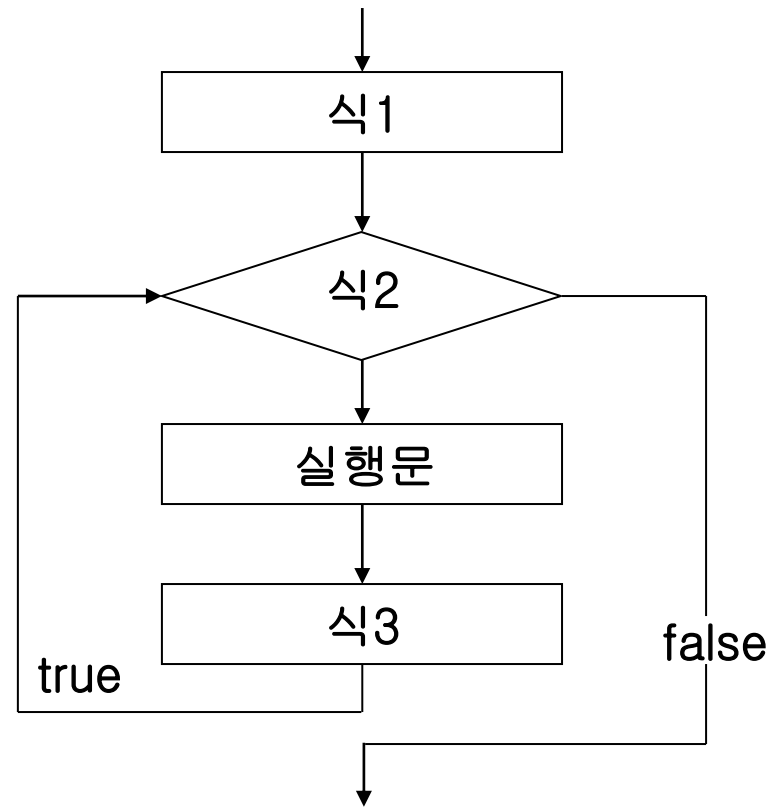
3.7 break, continue 문

---

- 조건이 만족되는 동안 반복시키는 반복문이다.

```
for(식1; 식2; 식3)  
{  
    실행문;  
}
```

식1 : initialize  
식2 : expression  
식3 : in/decrement



**for문**

- 식이 참일 때 루프 처리를 반복한다.
- 루프 처리로 들어가기 전에 우선 식1(초기값 설정)을 실행한다.
- 그 후 식2(계속조건)가 참일 때 문장을 계속 실행한다.
- 각 루프 처리 때마다 「문장」의 실행 후 식3(증감식)을 실행한다.
- 식2가 처음부터 거짓일 때는 실행문을 한번도 실행하지 않는다.
- 반복 수행할 문장이 여러 개일 때는 { }를 이용한다.
- 「초기화, 조건식, 증감식」은 일부 또는 전부를 생략할 수 있다.

```
for( ; n <= 100; n += 2)    // 초기값 생략 가능
for(n = 1; ; ++n)          // 조건식 생략 가능
for(n = 3; n < 100; )       // 증감식 생략 가능
for( ; ; )                 // 전부 생략 가능, 이 때는 무한 반복됨
```

- for문의 loop(루프)내에 또 다른 for문의 loop가 포함된 다중 for문 사용이 가능하다.

```
#include <stdio.h>

int main()
{
    int x;
    for (x = 9; x >= 1; x -= 2)
    {
        if (x > 1)
            printf("%d, ", x);
        else
            printf("%d\n", x);
    }
    return 0;
}
```



```
int main()
{
    int a;
    for (a = 1; a <= 10; a++) {
        printf("hello\n");
    }
    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int i, j, data;
    for (i = 1, data = 0; i <= 6; i++) {
        for (j = 1; j <= 1; j++)
            data += j;
    }
    printf("%d\n", data);
}
```

```
int main()
{
    int age, level;
    printf("Enter the your age : ");
    scanf("%d", &age);

    for (level = 0; age >= 10; level++)
    {
        age -= 10;
    }
    printf("%d대\n", level * 10);
    return 0;
}
```

- 다음 프로그램의 오류를 지적하시오.

```
int main()
{
    int i;
    for (i = 0; i <= 10; i++;)
        printf("%d\n", i);
    return 0;
}
```

# 3. 제어문(control statement)

---

3.1 제어문의 종류 및 특징

3.2 if 문

3.3 switch case 문

3.4 while 문

3.5 do while 문

3.6 for 문

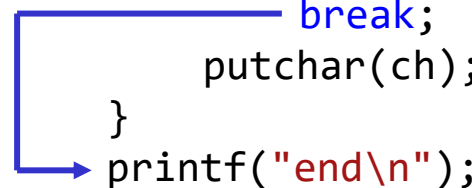
3.7 break, continue 문

---

- switch문 또는 for, while, do~while의 loop에서 탈출시키기 위한 문이다.
- 흐름이 반전 또는 jump된다.

```
int main()
{
    char ch;
    while (1)
    {
        if ((ch = getchar()) == '/')
            break;
        putchar(ch);
    }
    printf("end\n");

    return 0;
}
```

A blue line with an arrow originates from the 'break;' statement inside the while loop and points to the closing brace of the while loop, indicating that the loop is exited immediately.

**break문**

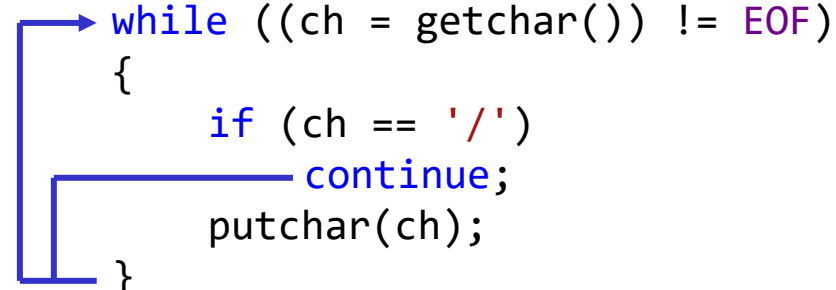
- switch문 안에서 이용되면 특정한 case 라벨부터 시작된 실행을 중단하고 그 switch문 전체를 종료한다.
- for, while, do-while 루프 안에서 이용되면 그 루프 처리를 도중에 중지한다.
- 중첩된 반복문에서 break를 사용하면 반복구주를 한 단계 벗어난 후 바깥쪽 반복문을 계속 수행한다. 즉, break를 기준으로 하여 가장 가까운 루프만 빠져 나온다.
- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int sum = 0, k = 0;
    while (1) {
        k++;
        if (k > 5)
            break;
        sum = sum + k;
    }
    printf("%d, %d\n", k, sum);
    return 0;
}
```

- for, while, do-while의 loop 안에서 처리를 건너뛰기 위한 문이다.
- 조건 판단 부분으로 제어가 옮겨진다.

```
#include <stdio.h>

int main()
{
    int ch;
    while ((ch = getchar()) != EOF)
    {
        if (ch == '/')
            continue;
        putchar(ch);
    }
    return 0;
}
```



**continue문**

continue문은 일반적으로 if문과 함께 쓰인다. 반복문 안에서 어떤 조건을 취하면 그 회의 처리를 건너뛴다. continue가 쓰여진 위치에서 반복문 끝까지 모두 건너뛴다.

break는 반복문 전체를 종료 시키지만, continue는 그 회의 처리를 건너뛴 뿐 다음 회 이후의 반복문은 정상적으로 처리된다.

- for, while, do-while안에서 이용된다.
  - 루프 처리를 일시적으로 skip한다.
  - 지정한 위치에서 반복문의 끝으로 제어를 옮긴다.
- 
- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int x, y;
    for (x = y = 0; y < 10; y++) {
        if (y % 2 == 0)
            continue;
        x += y;
    }
    printf("%d, %d\n", x, y);
    return 0;
}
```

➡ 1에서 100까지 차례대로 출력하고, 합을 구하는 프로그램을 while, for 문을 사용하여 작성하시오.

➡ 2, 4, 6, ..., 100 까지 출력하고, 합을 구하는 프로그램을 while, for문을 사용하여 작성하시오.

➡ 5명 학생에 대한 점수를 입력 받아 학점으로 변경시키고, 합계와 평균을 구하는 프로그램을 작성하시오.

( 90 ~ 100점 : A학점, 80 ~ 89점 : B학점, 70 ~ 79점 : C학점, 그 외 : F학점 )



➡ 정수( $n$ )를 입력 받아  $n!$ 을 구하는 프로그램을 작성하시오.

➡  $n$ 개의 숫자를 읽어 들여 이들 중 최대값과 최소값, 그리고 합과 평균을 계산하여 출력하는 프로그램을 작성하시오.

➡ 10-100까지 의 양의 정수들 중에서 4의 배수의 개수와 그 배수의 합을 구하는 프로그램을 작성하시오.

- ➡ 두개의 숫자를 입력 받아 최대 공약수와 최소 공배수를 구하는 프로그램을 작성하시오.
- ➡ 문자열(1 line – 배열 사용하지 않음)을 입력 받아 space를 dot로 출력하는 프로그램을 작성하시오.
- ➡ 문자열을 EOF가 입력될 때까지 입력 받도록 만들고 입력 받은 문자의 수, 단어의 수, 라인의 수를 출력하는 프로그램을 작성하시오.

➡ 구구단을 출력하는 프로그램을 작성하시오.

➡ 아래의 예와 같은 다이아몬드 도형을 출력하는 프로그램을 작성하시오.

예)

```
  *  
 ***  
*****  
*****  
*****  
 ***  
  *
```

## 4. 배열(array)



- ◆ 배열의 개념을 이해한다.
- ◆ 일차원 배열을 이해한다.
- ◆ 문자 배열을 이해한다.
- ◆ 이차원 배열을 이해한다.

---

4.1 배열의 개념

4.2 일차원 배열

4.3 문자 배열

4.4 이차원 배열

---

## 4. 배열(array)

---

4.1 배열의 개념

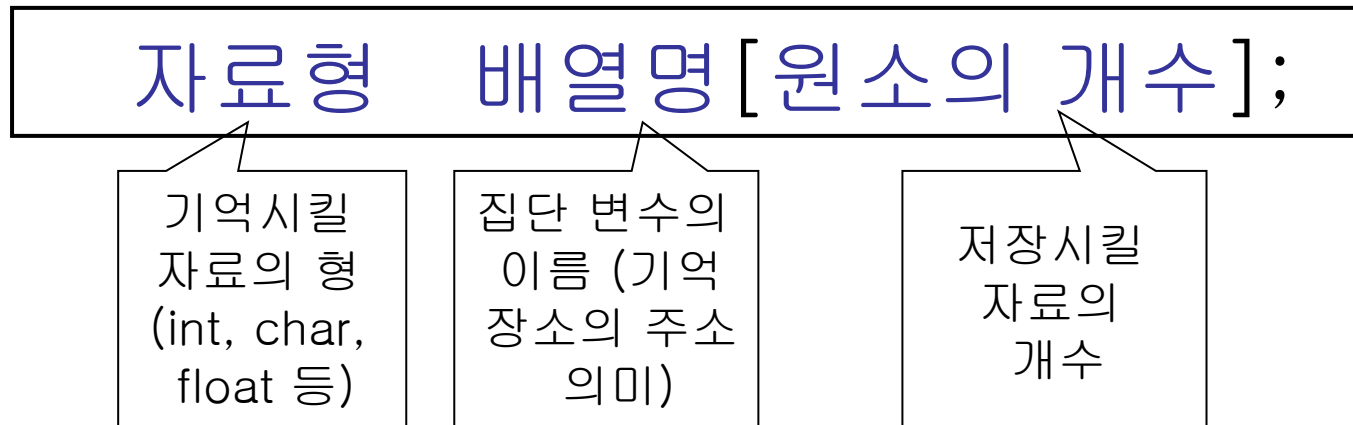
4.2 일차원 배열

4.3 문자 배열

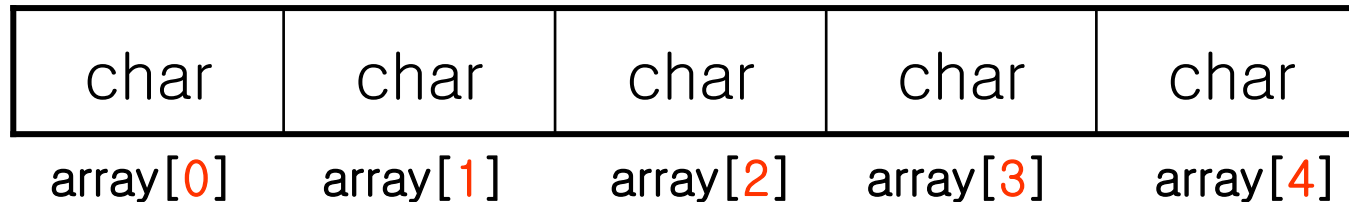
4.4 이차원 배열

---

- 같은 형태의 구조화된 data 집단



char array[5];



## 배열

배열은 하나의 변수를 만들어 사용한 단순 변수와는 달리 첨자를 이용하여 하나의 변수명으로 여러 개의 변수를 표현할 수 있는 형태이다.

- 배열의 구조 : 1차원적인 직선형태와 2차원적인 평면형태 그리고 다차원적인 입체적 구조로 사용될 수 있다. (배열의 차원에는 제한이 없고 단지 기억장치의 크기와 관련이 있다. 그러므로 배열의 형태는 자료구조의 개념이지 실제 memory 상의 구현 형태와는 별개의 문제이다.)
- 배열도 집단적인 변수의 형태이기 때문에 변수와 마찬가지로 프로그램에서 사용 이전에 반드시 선언되어야 하고, 그 구조는 아래와 같이 표현할 수 있다.

```
char c_array[80];    // char형 변수 80 개로 구성되는 배열 c_array
int i_array[10];     // int형 변수 10 개로 구성되는 배열 i_array
float f_array[20];   // float형 변수 20 개로 구성되는 배열 f_array
```

- 첨자는 0부터 시작된다.  
사용할 수 있는 요소의 개수는 선언할 때 적어준 첨자 수이다.  
따라서 「 int array[5]; 」 이라고 선언했을 때 이용할 수 있는 배열은  
array[0] ~ array[4]  
의 5개 이다. array[5]는 없으므로 주의한다.



## 4. 배열(array)

---

4.1 배열의 개념

4.2 일차원 배열

4.3 문자 배열

4.4 이차원 배열

---

```
int array[5] = {1, 2, 3, 4, 5};
```

1	2	3	4	5
array[0]	array[1]	array[2]	array[3]	array[4]

- array[5]는 int형 공간 5개를 의미한다.
- 배열의 변수(첨자)들은 0에서부터 시작한다.
- $\text{sizeof(array)} = 4\text{bytes} * 5 = 20\text{bytes}$ .
- $\text{array} == \&\text{array}[0]$

### 일차원 배열

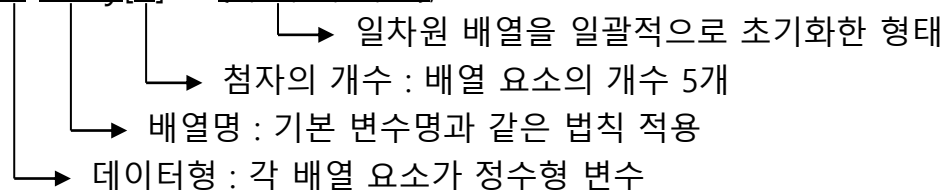
첨자가 있는 [ ] (대괄호)의 개수가 1개인 배열 형태를 1차원 배열이라 한다.

배열명은 변수명과 같은 법칙이 적용된다.

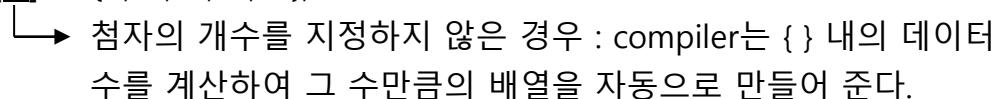
즉, 예를 들면 정수형 배열인 경우 정수형 데이터만 기억된다.

#### ■ 배열의 초기화

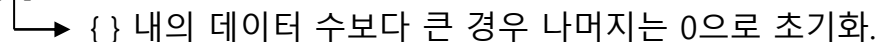
[예 1] `int array[5] = {1, 2, 3, 4, 5};`



[예 2] `int array[_] = {1, 2, 3, 4, 5};`



[예 3] `int array[5] = {1, 2, 3};`



[예 4] `int array[3] = {1, 2, 3, 4, 5}; // compiler는 error 처리`



```
int main()
{
    int i, array[5] = { 1,2,3,4,5 }; // 배열에 초기값 부여
    for (i = 0; i < 5; i++)
        printf("%d\t", array[i]);
    printf("\n");
    return 0;
}
```

```
int main()
{
    int i, array[5];
    for (i = 0; i < 5; i++) {
        array[i] = i + 1; // 배열에 대입
        printf("%d\t", array[i]);
    }
    printf("\n");
    return 0;
}
```

- 일차원 정수 배열의 초기화

```
int main()
{
    int a[3] = { 1, 2, 3 };
    int b[] = { 10, 20, 30 };
    int sum1, sum2, sum3;

    sum1 = 0;
    sum2 = 0;
    sum3 = 0;

    sum1 = a[0] + b[0];
    sum2 = a[1] + b[2];
    sum3 = a[2] + b[2];

    printf("sum1 = %d, sum2 = %d, sum3 = %d\n", sum1, sum2, sum3);
    return 0;
}
```

## 4. 배열(array)

---

4.1 배열의 개념

4.2 일차원 배열

4.3 문자 배열

4.4 이차원 배열

---

- 문자열을 표현하기 위해서는 char 형 배열을 사용
- 문자열의 끝에는 NULL(' 0')이 자동으로 추가  
➔ 배열의 크기 : 필요한 문자 수 + 1

```
char array[5] = "come";  
// char array[] = "come";  
// char array[5] = {'c', 'o', 'm', 'e'};  
// char array[] = {'c', 'o', 'm', 'e', '\0'};
```

'c'	'o'	'm'	'e'	'\0'
array[0]	array[1]	array[2]	array[3]	array[4]

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

```
int main()
```

```
{
```

```
    int number[4];
```

```
    char alpha[8];
```



정수 및 문자배열을 선언하며  
배열 사용 전에 선언해야 한다.

```
    /* 자료의 저장 */
```

```
    number[0] = 5;
```

```
    number[1] = 10;
```

```
    number[2] = 15;
```

```
    number[3] = 20;
```



정수 배열은 각 공간에  
원하는 정수 값 대입

```
    alpha[0] = 'a';
```

```
    alpha[1] = 'b';
```

```
    alpha[2] = 'c';
```

```
    alpha[3] = 'd';
```

```
    alpha[4] = '\0';
```



문자배열에 abcd를 할당하며  
한 문자를 단일 인용으로 묶는다.

```
    return 0;
```

```
}
```

문자열을 배열에 할당할 경우 문자열  
마지막에 NULL('\0')을 삽입한다.



## 예제 I

```
#include <stdio.h>

int main(void)
{
    int i, num[10];
    for (i = 0; i < 10; i++)
    {
        printf("Input %d : ", i + 1);
        scanf("%d", &num[i]);    // 배열의 정수 값 입력
    }
    printf("\nOUTPUT\n");

    for (i = 0; i < 10; i++)
        printf("%d\t", num[i]);    // 배열의 정수 값 출력
    printf("\n");
    return 0;
}
```

## 예제 I

```
/* 정수 값 10개를 입력 받아 역순으로 출력하는 프로그램 */
#include <stdio.h>

int main(void)
{
    int i, num[10];
    for (i = 0; i < 10; i++) {
        printf("Input %d : ", i + 1);
        scanf("%d", &num[i]);    // 배열의 정수 값 입력
    }
    printf("\nOUTPUT\n");
    for (i = 9; i >= 0; i--)
        printf("%d\t", num[i]);    // 배열의 정수 값 출력
    printf("\n");
    return 0;
}
```

for(i = 0; i < 10; i++)  
printf("%d\t", num[10 - i - 1]);

## 예제 II

```
/* 각 달의 날짜 수를 출력 */
#include <stdio.h>
#define MON 12

int main(void)
{
    int days[MON] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
    int index;

    for (index = 0; index < MON; index++)
    {
        printf("Month %d has %d days.\n", index + 1, days[index]);
    }
    return 0;
}
```

## 예제 II

- 다음 프로그램의 실행 결과는 무엇인가?

```
#include <stdio.h>

int main(void)
{
    char str1[7] = { 'K', 'O', 'R', 'E', 'A', '\0' };
    char str2[6] = "KOREA";
    char str3[] = "KOREA";
    char alpha[] = { 'a', 'b', 'c', '\0', '1', '2', '3', '\0' };

    printf("%s\n", alpha);
    printf("%c\n", alpha[4]);
    printf("%s\n", &alpha[4]);
    printf("%d\n", alpha);
    printf("%s %s %s\n", str1, str2, str3);

    return 0;
}
```

## 4. 배열(array)

---

4.1 배열의 개념

4.2 일차원 배열

4.3 문자 배열

4.4 이차원 배열

---

```
int ar[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

	ar[0][0]	ar[0][1]	ar[0][2]	ar[0][3]
ar[0]	1	2	3	4
ar[1]	5	6	7	8
ar[2]	9	10	11	12
	ar[2][0]	ar[2][1]	ar[2][2]	ar[2][3]

- `sizeof(ar) = sizeof(int) * 12 = 48 bytes.`
- `ar == ar[0] == &ar[0][0]`

### 이차원 배열

첨자가 있는 [ ] (대괄호)의 개수가 2개인 배열 형태를 2차원 배열이라 한다.

정수형 자료의 저장을 ar라는 이름의 2차원 배열에 대한 배열구조의 의미는 아래와 같다.

```
int ar[3][4];
```

ar[0][0]	ar[0][1]	ar[0][2]	ar[0][3]
ar[1][0]	ar[1][1]	ar[1][2]	ar[1][3]
ar[2][0]	ar[2][1]	ar[2][2]	ar[2][3]

ar : 배열 ar의 시작 번지를 의미

ar[0] : 첫 번째 행의 시작번지를 의미

ar[1] : 두 번째 행의 시작번지를 의미

ar[2] : 세 번째 행의 시작번지를 의미

ar[2][2] : 세 번째 행의 세 번째 열에 저장된 자료 값을 의미

```
/* 3×4 배열에 정수 값을 입력하고 출력 */
#include <stdio.h>

int main(void)
{
    int i, j, array[3][4];
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++) {
            printf("Input number : ");
            scanf("%d", &array[i][j]); // 입력
        }
    }
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 4; j++)
            printf("%d\t", array[i][j]); // 출력
        printf("\n");
    }
    printf("\n");
    return 0;
}
```



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

```
int main(void)
```

```
{
```

```
    int num[3][4];
```

```
    char month[3][4];
```

```
    num[0][0] = 1;    num[0][1] = 2;
```

```
    num[0][2] = 3;    num[0][3] = 4;
```

```
    num[1][0] = 5;    num[1][1] = 6;
```

```
    num[1][2] = 7;    num[1][3] = 8;
```

```
    num[2][0] = 9;    num[2][1] = 10;
```

```
    num[2][2] = 11;   num[2][3] = 12;
```

```
    month[0][0] = 'J'; month[0][1] = 'a';
```

```
    month[0][2] = 'n'; month[0][3] = '\0';
```

```
    month[1][0] = 'F'; month[1][1] = 'e';
```

```
    month[1][2] = 'b'; month[1][3] = '\0';
```

```
    month[2][0] = 'M'; month[2][1] = 'a';
```

```
    month[2][2] = 'r'; month[2][3] = '\0';
```

```
    return 0;
```

```
}
```

정수 및 문자배열을  
선언하며 배열 사용  
전에 선언해야 한다.

정수 배열에  
1 ~ 12 의  
수를 저장

문자배열에 Jan, Feb,  
Mar을 할당하며  
한 문자를 단일  
인용으로 묶는다.  
문자열을 배열에 할당할  
경우 문자열 마지막에  
NULL('\0')을 삽입한다.

- ➡ 1차원 배열에 정수 10개를 입력 받아 배열의 요소 중 최소값, 최대값, 전체 합을 구하여 화면에 출력하는 프로그램을 작성하시오.
  
- ➡ 2차원 배열을 사용하여 3명 학생에 대한 KOREAN, ENGLISH, MATH의 성적을 입력 받아 아래의 결과와 같은 성적표를 출력하는 프로그램을 작성하시오.

KOREAN	ENGLISH	MATH	TOTAL
=====			
77	66	88	231
90	80	100	270
65	75	85	225

➡ 아래의 출력 결과는 1 ~ 200 사이의 소수를 구한 결과이다. 아래와 같은 출력 결과를 나타내는 프로그램을 작성하시오.

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199				

Total : 46 개

➡ 10행 10열의 이차원 배열을 사용하여 아래와 같은 출력결과를 나타내는 프로그램을 각각 작성하시오.

출력결과 1)

									1
								2	3
							4	5	6
						7	8	9	10
					11	12	13	14	15
				16	17	18	19	20	21
			22	23	24	25	26	27	28
		29	30	31	32	33	34	35	36
	37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54	55

출력결과 2)

1	2	4	7	11	16	22	29	37	46
	3	5	8	12	17	23	30	38	47
		6	9	13	18	24	31	39	48
			10	14	19	25	32	40	49
				15	20	26	33	41	50
					21	27	34	42	51
						28	35	43	52
							36	44	53
								45	54
									55

➡ 사용자에게 임의의 동일한 행 / 열의 정수 값을 입력 받아 아래와 같은 출력형식을 나타내는 프로그램을 각각 작성하시오.(단 홀수만 입력 가능)

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

나비모양

1	3	6	10	15
2	5	9	14	19
4	8	13	18	22
7	12	17	21	24
11	16	20	23	25

45도 방향 규칙성

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

달팽이 배열

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

마방진

## 5. 포인터(pointer)



- ◆ 포인터의 개념을 이해한다.
- ◆ 포인터와 배열의 관계를 이해한다.
- ◆ 포인터 연산을 이해한다.
- ◆ 포인터와 다차원 배열의 관계를 이해한다.
- ◆ 포인터와 문자열의 관계를 이해한다.
- ◆ 포인터 배열에 대해 이해한다.
- ◆ 다중 포인터에 대해 이해한다.

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

5.5 포인터와 문자열

5.6 포인터 배열

5.7 다중 포인터

---

# 5. 포인터(pointer)

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

5.5 포인터와 문자열

5.6 포인터 배열

5.7 다중 포인터

---



- ➡ 어떤 데이터가 차지하는 기억 장소의 주소를 저장하기 위한 변수
- ➡ 원시적인 데이터 유지
  - 1000 번지에 33을 넣는다.
- ➡ 변수에 의한 데이터 관리
  - 1000번지라는 메모리 영역을 data라고 부른다.
  - data에 33을 넣는다.

포인터

포인터는 기계어 level의 기능을 언어 사양에 반영시킨 것으로, 언어인 동시에 기계어 레벨의 세세한 처리를 가능하게 해준다.  
변수명으로 특정 번지를 지정할 수 있는 것은 compiler 내부에 아래와 같은 변수명과 특정 번지를 연결하는 변환표(대응표)가 있기 때문이다.

변환표

변수명	해당 번지
data	1000
num	2000

이 테이블을 참조하면 「 data = 33;」을 명령했을 때  
“1000번지에 33을 넣는다”  
와 같은 구체적인 번지 정보를 알 수 있다.

- 포인터명 앞에 \* 을 붙인다.
- 데이터형은 그 포인터 변수가 가리키고자 하는 지점의 데이터형에 맞게 선언한다.
- sizeof(포인터명) = 2 bytes(ANSI C)

## 데이터형 \* 포인터명;

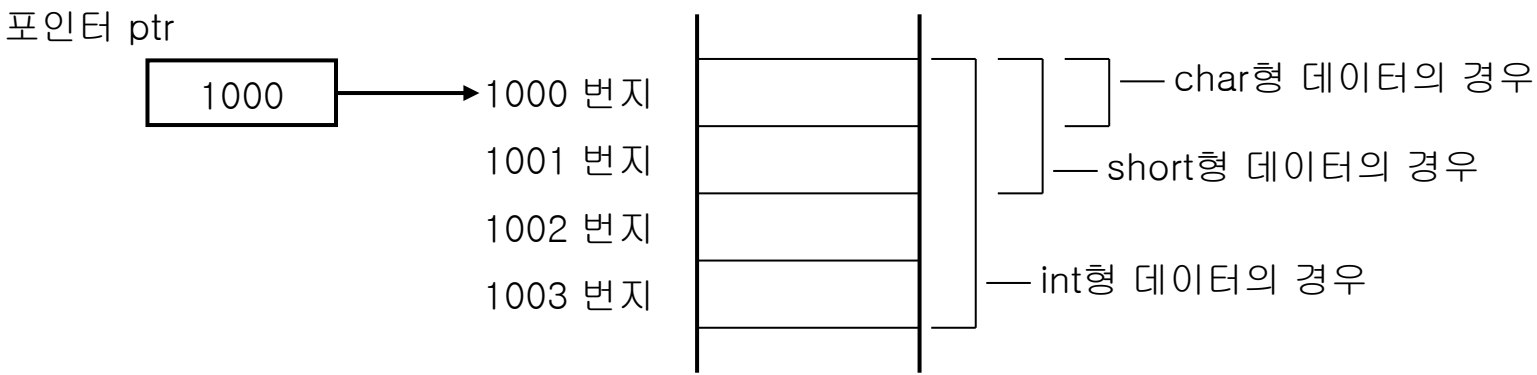
```
char *cptr;    // char 형을 가리키는 포인터 cptr 선언
int *iptr;     // int 형을 가리키는 포인터 iptr 선언
float *fptr;   // float 형을 가리키는 포인터 fptr 선언
```

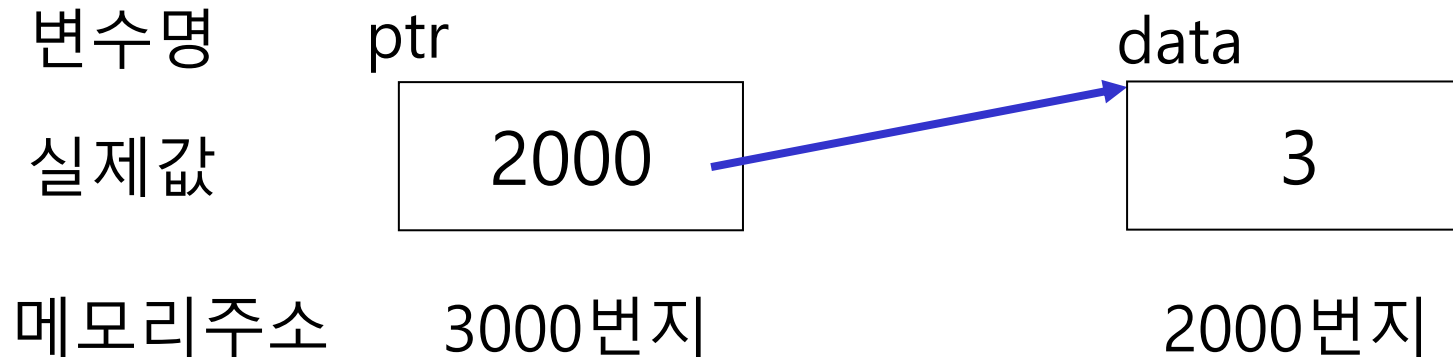
포인터 연산자

- 주소연산자(&) : & 바로 다음에 변수가 오면 그 변수가 저장된 주소를 나타낸다.
  - ① &는 수식, 상수, 레지스터 변수에는 사용 하지 못한다.
  - ② 단순변수, 배열요소, 포인터변수, 구조체변수는 가능하다.
- 값 연산자 : 실행문일 경우 \* 바로 다음에 pointer가 오면 그 pointer가 가리키는 주소에 저장된 값을 나타낸다.

포인터는 변수와 정보 교환을 할 수 있어야 하기 때문에 변수의 data type인 char, int, float 등 다양한 data type에 맞는 포인터 변수의 선언이 이루어져야 한다.

포인터 ptr이 1000일 때 char형 데이터를 가리키면 1000번지의 내용을 추출하지만, short형 데이터를 추출할 경우는 1000 ~ 1001번지의 2bytes 데이터를 추출할 필요가 있다. 그리고 int 형 데이터를 추출할 경우는 1000번지 ~ 1003번지의 4bytes의 데이터를 추출해야 한다.





```
int data;  
int *ptr;    // ptr이 정수형 pointer 변수임을 의미  
data = 3;  
ptr = &data; // data의 주소
```

```
ptr = &x; // 단순변수 x가 차지하는 실제 값의 주소를 ptr에 대입
y = *ptr; // 변수 y에 포인터 변수 ptr이 가리키는 곳의 값 즉, 단순 변수 x의 값을 대입
```

```
ptr = &x;
y = *ptr;
```

```
int num1, num2;
int *ptr;           // 포인터 변수의 선언
num2 = 10;
ptr = &num2;         // 단순변수 num2가 차지하는 실제 값의 주소를 ptr에 저장
num1 = *ptr;         // 변수 num1에 포인터 변수 ptr이 가리키는 곳의 값을 대입
```

선언	데이터 참조	주소 참조
int a;	a	&a
int *ptr	*ptr	ptr

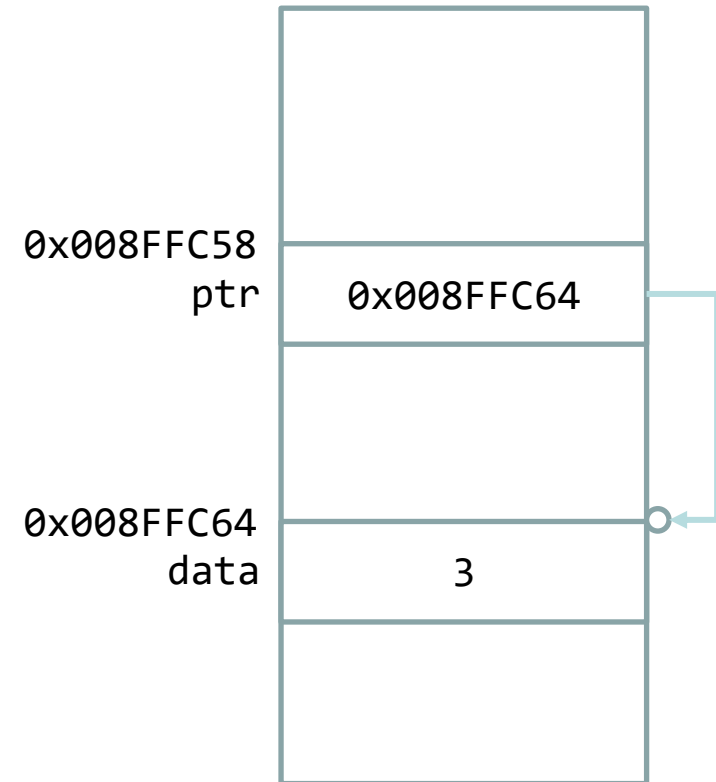
## 실제 메모리 상태

```
#include <stdio.h>

int main()
{
    int data;
    int* ptr;
    data = 3;
    ptr = &data;
    printf("&data = %p\n", &data);
    printf("ptr = %p\n", ptr);
    printf("&ptr = %p\n", &ptr);
    return 0;
}
```

## 출력결과

```
&data = 008FFC64
ptr    = 008FFC64
&ptr   = 008FFC58
```



- 포인터 선언 시 값을 초기화할 수 있다.
- 초기화는 포인터 선언과 주소 값 설정을 동시에 행하는 것이다.

**데이터형 \*포인터명 = 초기값 주소;**

```
char ch = 'A';  
char *cptr;  
cptr = &ch;
```

```
char ch = 'A';  
char *cptr = &ch;
```



### 포인터 초기화의 예

[예 1] 변수의 주소를 & 연산자로 설정한다.

```
int d;  
int *ptr = &d;
```

[예 2] 배열의 시작 주소 값을 설정한다.

```
char str[80];  
char *ptr = str;
```

[예 3] 배열의 요소에 & 연산자를 취하여 얻은 주소 값을 포인터에 설정한다.

```
char str[80];  
char *ptr = &str[10];
```

[예 4] char형 포인터에 문자열의 시작 주소 값을 설정한다. 우선 문자열 "abcde"를 메모리 상의 어딘가에 확보하고, 그 시작 주소 값을 포인터 ptr에 설정한다. 문자열 "abcde"가 ptr에 들어가는 것은 아니다.

```
char *ptr = "abcde";
```

[예 5] 프로그램 실행 개시 후 동적으로 메모리를 확보하고 그 시작 주소 값을 설정한다.

```
char *ptr = (char *)malloc(100);
```

```
#include <stdio.h>

int main()
{
    int num1=0, num2=0;
    int* ptr = &num1;

    *ptr = 10;
    printf("%d, %d, %d\n", num1, num2, *ptr);

    num2 = *ptr;
    printf("%d, %d, %d\n", num1, num2, *ptr);

    ptr = &num2;
    *ptr = 5;
    printf("%d, %d, %d\n", num1, num2, *ptr);

    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
#include <stdio.h>

int main()
{
    int num1 = 0, num2 = 0;
    int* ptr = &num1;

    printf("%d, %d, %d\n", num1, num2, *ptr);

    num2 = *ptr;
    printf("%d, %d, %d\n", num1, num2, *ptr);

    ptr = 0;
    printf("%d, %d, %d\n", num1, num2, *ptr);

    return 0;
}
```

# 5. 포인터(pointer)

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

5.5 포인터와 문자열

5.6 포인터 배열

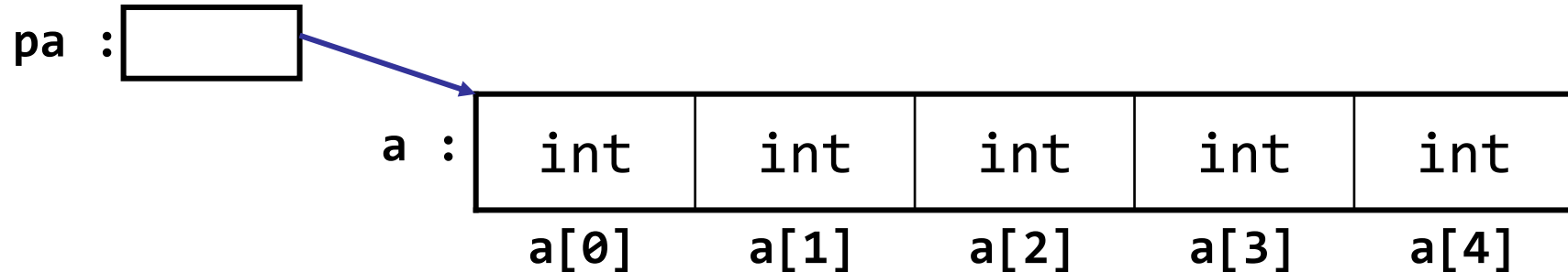
5.7 다중 포인터

---

- C에서 포인터와 배열은 밀접한 관계가 있다.

```
int a[5];  
int *pa;  
pa = &a[0];
```

```
int a[5];  
int *pa;  
pa = a;
```



포인터와 배열

배열에 의한 처리는 포인터를 사용해 처리할 수 있으며 포인터를 사용하는 것이 프로그램의 실행속도를 증가시킬 수 있다.

예를 들어 정수형 배열 `int array[10]`을 선언하고 이 배열과 포인터와의 관계를 살펴보면,

`int array[10];`

배열 array

--	--	--	--	--	--	--	--	--	--

`array[0]; array[1]; array[2]; array[3]; array[4]; array[5]; array[6]; array[7]; array[8]; array[9];`

이 선언으로 `array[0]`, `array[1]`, ..., `array[9]`와 같은 배열 변수를 사용할 수 있으며, `par` 를 정수형 포인터라고 하고 다음과 같이 선언한다.

```
int *par;
```

이와 같은 상태에서 배열의 첫 번째 주소 값을 포인터에 할당하면

```
par = &array[0];
```

`par` 포인터 변수에는 배열 `array`의 주소 값을 갖게 된다.

포인터에 주소 값을 할당할 경우 주소 연산자 `&`를 사용하면 된다. 그리고

```
x = *par;
```

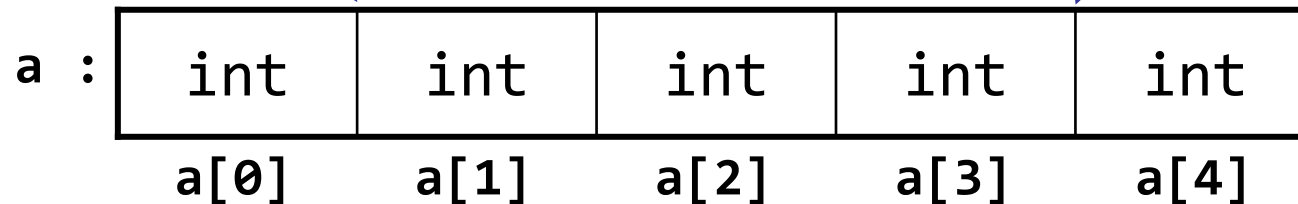
와 같이 하면 `x` 변수에는 배열 `array[0]`에 저장된 자료와 같은 값을 갖게 된다.

```
int a[5];  
int *pa;  
pa = &a[1];
```

```
int a[5];  
int *pa;  
pa = &a[4];
```

pa : &a[1]

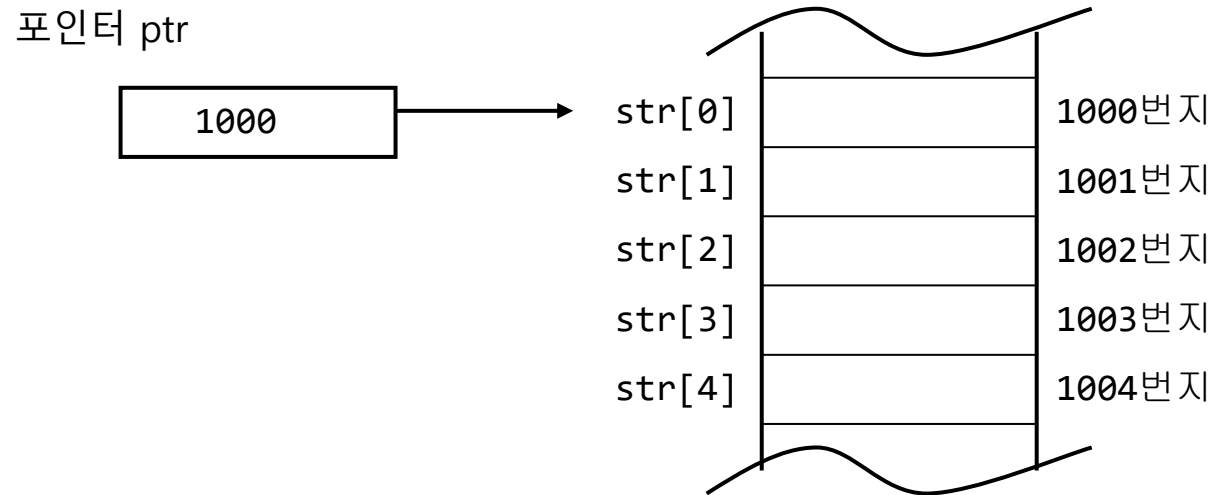
pa : &a[4]



**배열명은 포인터이다.**

```
char str[5];
```

의 선언에서 str은 배열 `str[0]` ~ `str[4]`의 시작 주소 값을 나타내는 정수이다.



그러므로 배열명은 포인터의 일종이다.

단 배열의 시작 주소 값을 고정적으로 가리키고 있는 정수로서의 포인터로 쓰인다. 따라서 배열명 `str`을 이용해 배열의 시작 주소 값을 다른 포인터에 대입할 수 있다. 그러나 반대로 배열명 `str`에 무언가를 대입하는 등의 변경 행위는 허용되지 않는다.



# 5. 포인터(pointer)

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

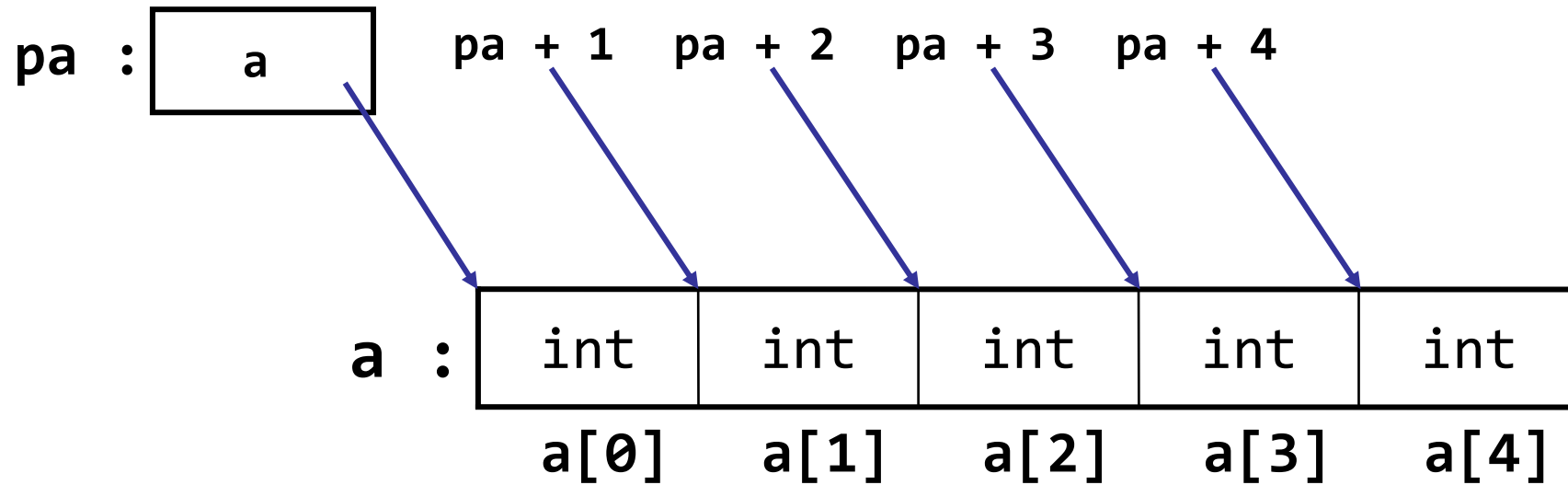
5.5 포인터와 문자열

5.6 포인터 배열

5.7 다중 포인터

---

```
int a[5];  
int *pa = a;
```



### 포인터 연산

포인터 `par`이 배열 `array[i]`의 포인터라면, `par + 1`은 `array[i + 1]`의 포인터가 되고 `par - 1`은 `array[i - 1]`의 포인터가 된다.

배열과 포인터와의 관계에서 포인터에 1을 더한다는 것은 1 byte를 더한다는 의미가 아니고 그 다음 배열 요소를 의미하는 것이다.

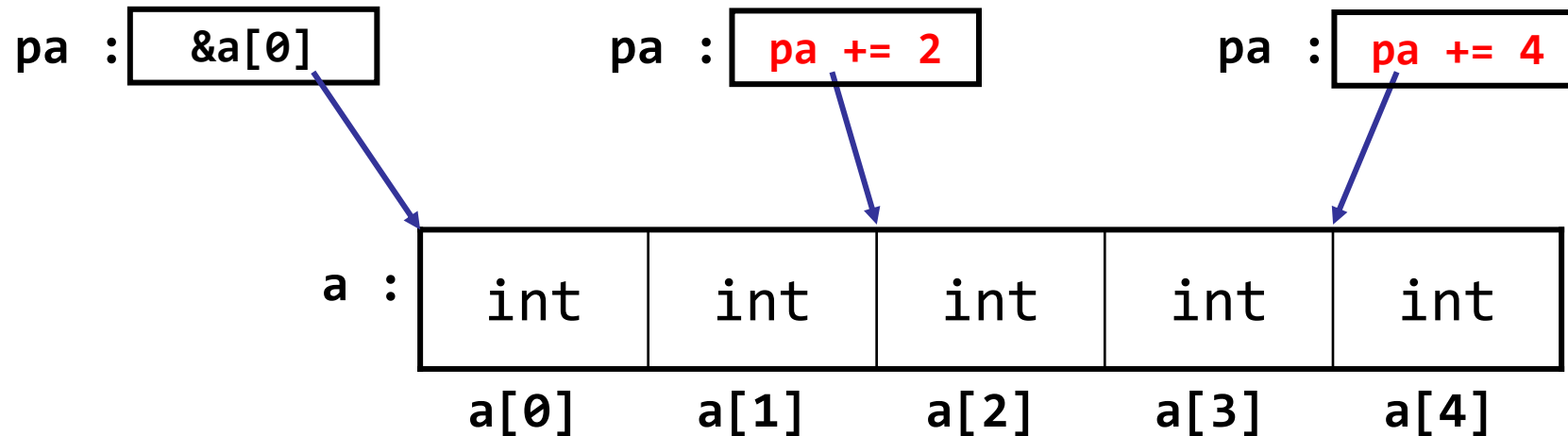
배열의 이름은 그 배열의 첫 번째 요소의 위치 값 즉 주소 값을 의미하므로 아래의 두 문장은 같은 의미로 해석될 수 있다.

```
int array[10], *par;  
par = array;  
par = &array[0];
```

```
int a[5];  
int *pa;  
pa = a;
```

```
int a[5];  
int *pa = a;  
pa += 2;
```

```
int a[5];  
int *pa = a;  
pa += 4;
```



```
#include <stdio.h>

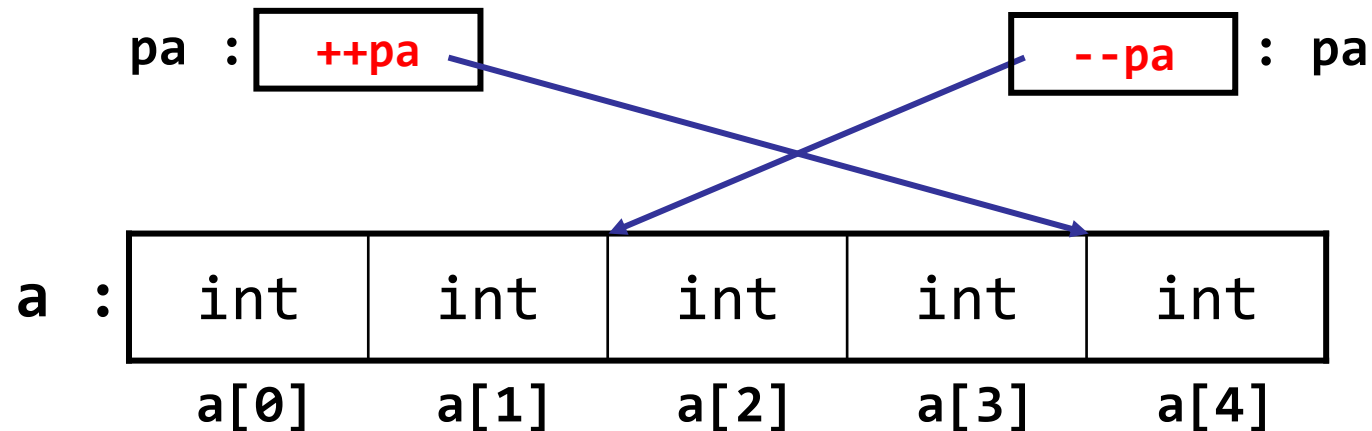
int main()
{
    char str[10];
    char* ptr;
    str[0] = 'A';
    str[1] = 'B';
    str[2] = 'C';
    ptr = str;
    printf("%c, %c, %c\n", str[0], str[1], str[2]);
    printf("%c, %c, %c\n", *ptr, *(ptr + 1), *(ptr + 2));

    return 0;
}
```

포인터 ptr을 통해  $*(ptr + i)$ 라고 표현하면  $str[i]$ 와 동일해진다.  
여기서 값  $ptr + i$ 는 포인터를 배열 str의 요소 +i 개 만큼 진행시킨다는 의미이다.  
포인터는 변수이므로 일시적으로 그 값을 늘릴 수도 있고, 값 자체를 가산할 수도 있다.

```
int a[5];  
int *pa = &a[3];  
++pa;
```

```
int a[5];  
int *pa = &a[3];  
--pa;
```



```
int array[5] = {1, 2, 3, 4, 5};  
int *ptr = array;
```

1	2	3	4	5
---	---	---	---	---

array[0]      array[1]      array[2]      array[3]      array[4]

\*(ptr+0)      \*(ptr+1)      \*(ptr+2)      \*(ptr+3)      \*(ptr+4)

ptr[0]      ptr[1]      ptr[2]      ptr[3]      ptr[4]

\*(array+0)    \*(array+1)    \*(array+2)    \*(array+3)    \*(array+4)

포인터의 주소 계산

포인터 연산은 실제로 포인터의 데이터 형에 의해 가/감산 되는 주소 값이 달라진다.  
다시 말하면 포인터는 1 가산할 때마다

sizeof(데이터형)  
만큼, 즉 「 즉 그 포인터가 나타내는 데이터형의 바이트 수 」만큼 번지가 진행되는 것이다. 따라서  
ptr = ptr + 4;  
를 행하면 새로운 포인터는 4요소만큼 앞의 주소를 가리키게 된다. 「 4 \* sizeof(데이터형) 」만큼 주소가 앞으로 진행된다.

- 포인터의 진행방법(char, int, long이 각각 1, 2, 4 byte인 경우)

데이터형	크 기	처 리	결 과
char	1	++ptr;	1번지 진행한다.
short	2	++ptr;	2번지 진행한다.
int	4	++ptr;	4번지 진행한다.

$$p + n \Rightarrow p + \text{sizeof}(*p) * n$$



- 포인터는 연산하거나 비교할 수 있다.
- 포인터 변수  $\pm$  정수
- 포인터 변수  $-$  포인터 변수

연산자	사용 예
+	<code>ptr = ptr + 2;</code>
-	<code>ptr = ptr - 2;</code>
++	<code>++ptr; 또는 ptr++;</code>
--	<code>--ptr; 또는 ptr--;</code>

## 포인터의 연산

포인터는 정수와의 덧셈, 뺄셈만이 허용되며 곱셈과 나눗셈은 할 수 없다. 즉, 주소를 따라가는 일만이 허용된다.

포인터간의 연산은 뺄셈만 가능하다.

```
char str[10], array[10], *sptr1, *sptr2, *sptr3;
int i, *iptr;
float f;
sptr1 = &str[3]; sptr2 = &str[5]; sptr3 = array;
iptr = i;
```

### ▪ 정수와의 덧셈과 뺄셈

```
iptr = iptr + 2;           // 2개 뒤의 요소를 가리킨다.
--iptr;                   // 앞의 요소를 가리킨다.
ip = ip + f;               // 이것은 오류, f는 정수가 아니다.
```

### ▪ 2개의 포인터가 동일한 배열의 임의의 요소를 가리키고 있을 때의 뺄셈

```
i = sptr2 - sptr1;        // i는 2가 된다.
i = sptr2 - cpstr3;       // 다른 배열을 가리키고 있으므로 에러는 아니지만 의미가 없다.
```

포인터 연산자 우선순위

기 술	해 석	주 석 문
*ptr + 1;	(*ptr) + 1;	값을 +1 한다.
*(ptr + 1);	*(ptr + 1);	번지를 +1 한 후 그 값을 가져온다.
*ptr += 1;	(*ptr) += 1;	*ptr을 +1 한다.
*ptr++;	*(ptr++);	번지를 1만 나중에 더한다.
(*ptr)++;	(*ptr)++;	값을 1만 나중에 더한다.
*++ptr;	*(++ptr);	번지를 1만 먼저 더한다.
++*ptr;	++(*ptr);	값을 1만 먼저 더한다.

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int count[5] = { 100, 200, 300, 400, 500 };
    int* ptr = count;
    printf("%d\n", *ptr++);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int days[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    int index, * ptr;

    ptr = days;

    for (index = 0; index < 12; ptr++, index++)
        printf("Month %d has %d day\n", index + 1, *ptr);
    // for(index = 0; index < 12; index++)
    // printf("Month %d has %d day\n", index + 1, *(ptr + index));
    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int menu[5] = { 0, 10, 20, 30, 40 };
    int* ptr, i;
    ptr = &menu[1];
    i = *ptr++;
    printf("i = %d, *ptr = %d\n", i, *ptr);
    return 0;
}
```

```
int main()
{
    int num, array[] = { 1, 2, 3, 4, 5 };
    num = *array + 3;
    printf("%d, ", num);
    num = *(array + 3) + 5;
    printf("%d\n", num);
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int data[2] = { 100, 300 };
    int* p1, * p2, * p3;

    p1 = p2 = p3 = data;

    printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",
        *p1++, *++p2, (*p3)++);
    printf("*p1 = %d, *p2 = %d, *p3 = %d\n",
        *p1, *p2, *p3);
    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    int i;
    char str[20], * ptr;
    for (i = 0; i < 20; i++)
        str[i] = 65 + i;
    ptr = str + 19;
    for (i = 0; i < 20; i++)
        printf("%c, %c\n", str[i], *(ptr - i));
    return 0;
}
```

```
int main()
{
    char city[] = "Korea Seoul";
    int i = 0;
    while (*(city + i) != '\0')
        i++;
    printf("%d\n", i);
    return 0;
}
```

# 5. 포인터(pointer)

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

5.5 포인터와 문자열

5.6 포인터 배열

5.7 다중 포인터

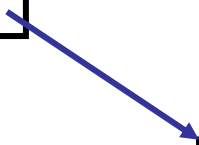
---



```
int a[3][4];  
int *pa;  
pa = &a[0][0];
```

pa **&a[0][0]**

a :



int	int	int	int
int	int	int	int
int	int	int	int

포인터와 이차원 배열

포인터와 이차원 배열은 서로 호환성이 있다. 즉, 포인터 선언 시에 이차원 배열을 사용할 수 있고 역으로 이차원 배열 선언 시에 포인터 사용이 가능하다.

```
int array[2][2] = {1, 2, 3, 4};    // 정수형을 저장할 수 있는 2차원 배열 선언
int *ptr;                          // 정수형 변수의 번지 값을 기억할 수 있는 포인터 변수
ptr = array[0];                    // ptr = &array[0][0];과 같은 의미
```

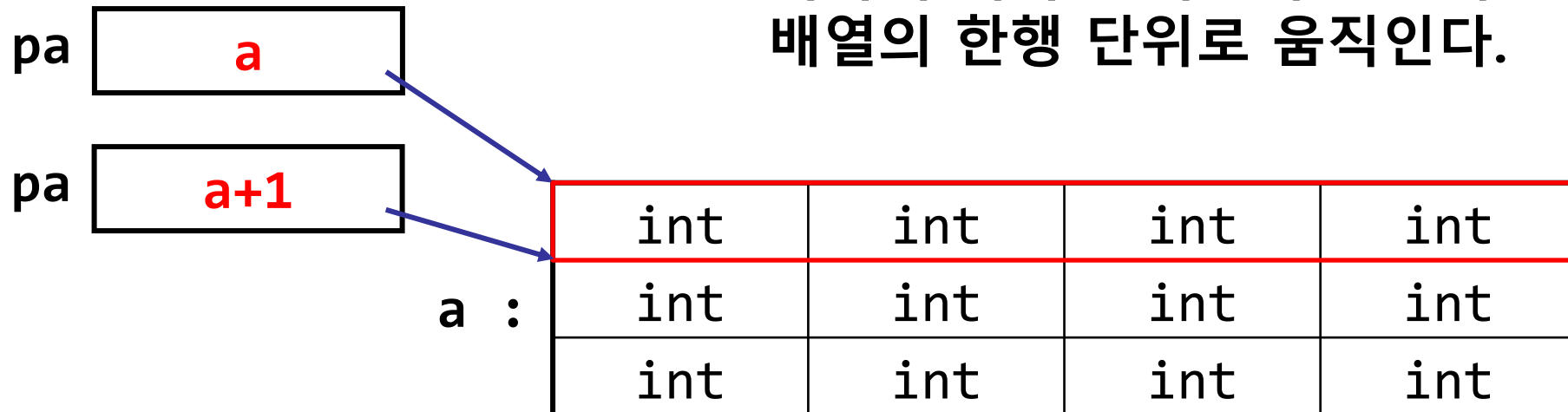
이차원 배열의 각 요소와 같은 표현은 다음 그림과 같다.

1	2	3	4
array[0][0]	array[0][1]	array[1][0]	array[1][1]
*(ptr + 0)	*(ptr + 1)	*(ptr + 2)	*(ptr + 3)
ptr[0]	ptr[1]	ptr[2]	ptr[3]
*(*(array + 0) + 0)	*(*(array + 0) + 1)	*(*(array + 1) + 0)	*(*(array + 1) + 1)

```
int a[3][4];  
int (*pa)[4];  
pa = a;
```

```
pa + 1  
=> pa + sizeof(*p)*1  
=> pa + sizeof(int[4])*1  
=> pa + 16*1
```

따라서 이차원 배열의 포인터는  
배열의 한행 단위로 움직인다.



```
#include <stdio.h>

int main()
{
    int array[2][2] = { 1,2,3,4 };
    int(*ptr)[2] = array;
    int i, j;

    ptr[0][0] = 10;
    ptr[0][1] = 20;
    ptr[1][0] = 30;
    ptr[1][1] = 40;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
            printf("array[%d][%d] = %d\n", i, j, array[i][j]);
    }

    return 0;
}
```

# 5. 포인터(pointer)

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

5.5 포인터와 문자열

5.6 포인터 배열

5.7 다중 포인터

---



포인터와 문자열

C에는 문자열을 표현하는 데이터형이 없으므로 char형 배열을 이용하여 문자열을 표현하는 것이 보통이다. 따라서 포인터와 배열과의 관계를 알아두면 포인터로 문자열을 조작하는 것을 이해할 수 있다.

포인터에 의한 문자열 처리      → 문자열의 시작 주소만 저장한다.

char형 배열에 의한 문자열 처리      → 문자열 정수 자체를 저장한다.

	문자열의 배열	포인터 변수
선언	char str[];	char *str;
초기화	char str[] = "string";	char *str = "string";
데이터 길이	고정	가변
각 요소 참조	str[i] (i = 0, 1, 2, 3, ...)	*(str + i) (i = 0, 1, 2, 3, ...)
값 할당	str[0] = 'c';	str = "hello world";
시작번지	str	str
번지연산	불가능	가능 ( 예 : str++ )
처리속도	첨자 계산을 수행함으로 포인터 변수보다 느리다.	빠르다.

# 5. 포인터(pointer)

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

5.5 포인터와 문자열

5.6 포인터 배열

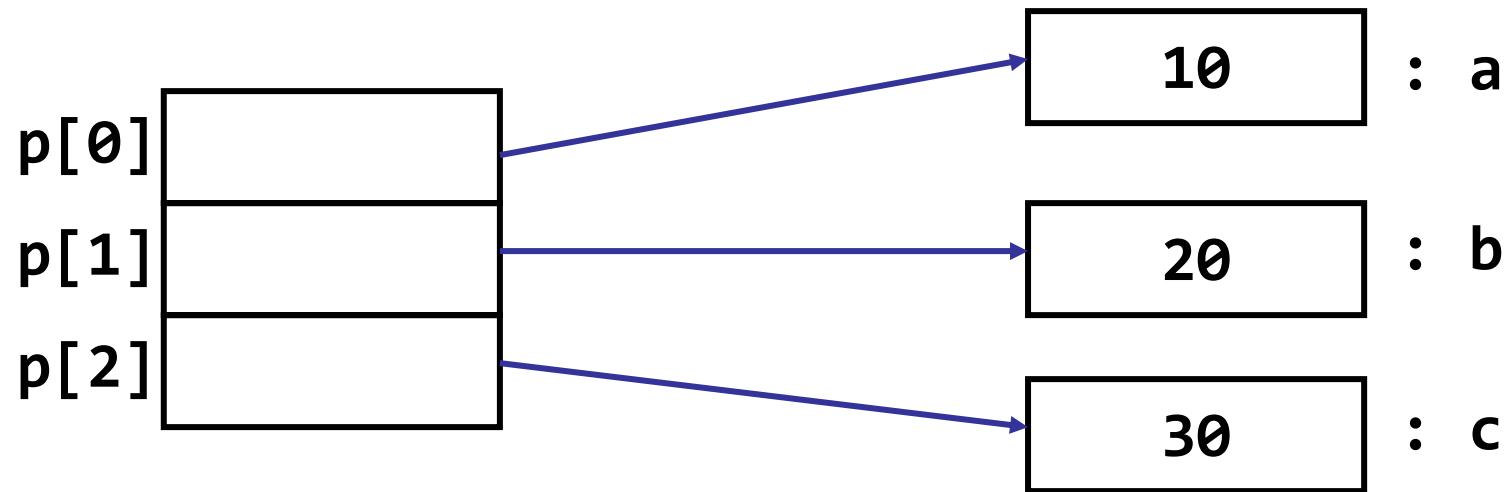
5.7 다중 포인터

---



- 포인터 변수들의 배열

```
int a = 10, b = 20, c = 30;  
int *p[3] = {&a, &b, &c};
```



## 포인터 배열

포인터도 변수이기 때문에 포인터들로 이루어진 배열도 가능하다.

주소 값을 여러 개 저장하기 위해서 포인터 변수를 여러 개 선언하지 않고 포인터로 배열을 선언할 수 있으며 이것은 주로 문자열을 표현할 때 사용한다.

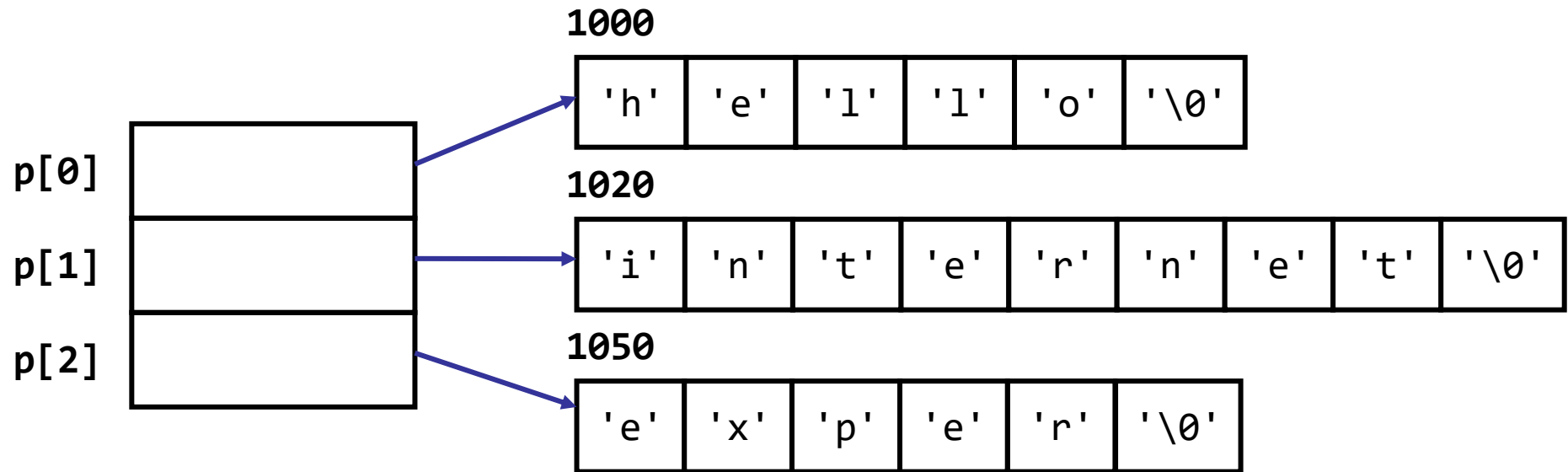
- 여러 개의 포인터 변수들로 구성된 배열을 포인터 배열이라고 한다.

```
char *n_city[3];    // *n_city[0], *n_city[1], *n_city[2] 의 배열요소로  
                   // 구성된 배열
```

- 포인터 배열은 선언과 초기화를 일괄적으로 할 수 있다.

```
char n_city[4] = {"korea", "seoul", "japan", "tokyo"};  
    // n_city[0] : 문자열 "korea"가 저장된 장소의 시작 주소를 가리킨다.  
    // n_city[1] : 문자열 "seoul"이 저장된 장소의 시작 주소를 가리킨다.  
    // n_city[2] : 문자열 "japan"이 저장된 장소의 시작 주소를 가리킨다.  
    // n_city[3] : 문자열 "tokyo"가 저장된 장소의 시작 주소를 가리킨다.
```

```
char *p[3] = {"hello", "internet", "world"};
```



- 포인터 배열은 길이가 일정치 않은 문자열들의 배열을 나타낼 때 유용하다. char형의 배열에서는 다차원 문자열을 다음과 같이 선언한다.

```
char array[3][9] = {"hello", "internet", "world"};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
array[0]	'h'	'e'	'l'	'l'	'o'	'\0'			
array[1]	'i'	'n'	't'	'e'	'r'	'n'	'e'	't'	'\0'
array[2]	'w'	'o'	'r'	'l'	'd'	'\0'			

포인터 배열과 다차원 배열 내 문장의 선언은 메모리 안이 데이터 확보 방법이 다르다. 배열에서는 각 배열 요소에 데이터의 실체가 들어간다. 한편 포인터에서는 3개의 문자열을 메모리 상의 어딘가에 확보하고 그 시작 번지를 각 포인터에 넣는다.

또한 배열의 경우는 설정하는 문자열이 짧을 때도 지정한 길이만큼 메모리를 소비한다. 그러나 포인터의 경우는 문자열의 길이만큼만 메모리를 사용하여 그 대신 포인터 자체를 기억하는 메모리 영역이 필요하다.

```
int main()
{
    int i;
    char* week[] = { "Sunday", "Monday", "Tuesday",
                     "Wednesday", "Thursday", "Friday", "Saturday" };
    for (i = 0; i < sizeof(week)/sizeof(week[0]); i++)
    {
        printf("week[%d] = %c, %s\n", i, *week[i], week[i]);
    }
    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
int main()
{
    char* fruit[] = { "apple", "grape", "banana", "orange" };
    printf("%s\n", fruit[1]);
    return 0;
}
```

# 5. 포인터(pointer)

---

5.1 포인터의 개념

5.2 포인터와 일차원배열

5.3 포인터 연산

5.4 포인터와 다차원 배열

5.5 포인터와 문자열

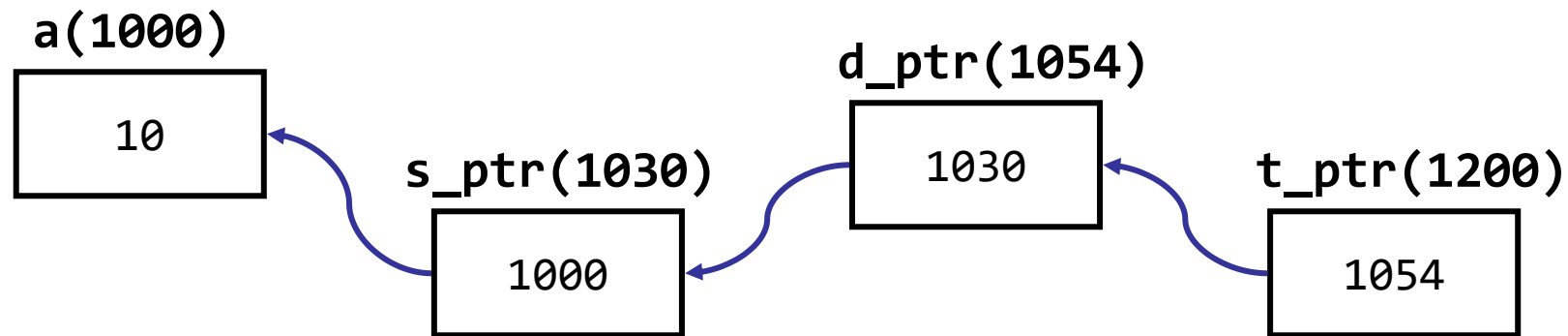
5.6 포인터 배열

5.7 다중 포인터

---

- 포인터의 포인터

```
int a = 10, *s_ptr = &a;  
int **d_ptr = &s_ptr, ***t_ptr = &d_ptr;
```



## ■ 포인터의 포인터

선 언	설 명
<code>char ch;</code>	<code>ch</code> 는 <code>char</code> 형 변수이다.
<code>char *s_ptr;</code>	<code>s_ptr</code> 은 <code>char</code> 형 값(의 위치)을 가리키는 포인터이다. 「 <code>*s_ptr</code> 」의 표현이 <code>char</code> 형 이다.
<code>char **d_ptr;</code>	<code>d_ptr</code> 은 <code>char</code> 형 값(의 위치)을 가리키는 포인터(의 위치)를 가리키는 포인터이다. 「 <code>**d_ptr</code> 」의 표현이 <code>char</code> 형 이다. 「 <code>*d_ptr</code> 」은 「 <code>**d_ptr</code> 」라는 값의 위치를 가리키는 포인터이다.
<code>char ***t_ptr;</code>	<code>t_ptr</code> 은 <code>char</code> 형 값(의 위치)을 가리키는 포인터(의 위치)를 가리키는 포인터(의 위치)를 가리키는 포인터이다. 「 <code>***t_ptr</code> 」의 표현이 <code>char</code> 형 이다. 「 <code>**t_ptr</code> 」은 「 <code>***t_ptr</code> 」라는 값의 위치를 가리키는 포인터이다. 「 <code>*t_ptr</code> 」은 「 <code>**t_ptr</code> 」라는 값의 위치를 가리키는 포인터이다.



➡ 10개의 정수를 입력 받아 오름차순으로 정렬하는 프로그램을 작성하시오.

(배열의 요소자체를 정렬시키시오.)

➡ 문자열을 입력 받아 소문자를 대문자로 변환하는 프로그램을 작성하시오.

➡ 문자열을 입력 받아 't' 문자가 문장에서 몇 번째 위치하는지 찾아내는 프로그램을 작성하시오.

➡ 포인터 배열을 사용하여 5개의 문장을 입력 받아 오름차순으로 정렬하는 프로그램을 작성하시오.

➡ 연도와 일수를 입력 받아 평년과 윤년을 구분하여 그 해의 월, 일로 출력하는 프로그램을 작성하시오. (4의 배수이지만 100의 배수는 아니며 400의 배수인 연도는 윤년이다.)

예) 연도를 입력하세요 : 2020

날짜정보를 입력하세요 : 205

2020, July, 24

- 다음 프로그램의 실행 결과는 무엇인가?

```
#include <stdio.h>

int main()
{
    char* language[] = { "visual c++", "java", "basic", "delphi" };
    char** ptr = language;

    printf(" [ %c ] ", *language[3]);
    printf(" [ %s ] ", *language);
    printf(" [ %s ]\n ", *++ptr);
    return 0;
}
```

- 다음 프로그램의 실행 결과는 무엇인가?

```
#include <stdio.h>

int main()
{
    char* com[] = { "monitor", "printer", "speaker", "mouse" };
    char** ptr[] = { com + 3, com + 2, com + 1, com };
    char*** root = ptr;

    printf(" [ %s ] ", **++root);
    printf(" [ %s ] ", *--*++root + 2);
    printf(" [ %s ] ", *root[-2] + 2);
    printf(" [ %s ] \n", root[1][1]);
    return 0;
}
```

## 6. 함수(function)



- ◆ 함수의 개념을 이해한다.
- ◆ 함수의 구성과 구현을 이해한다.
- ◆ 함수 매개변수의 전달 기법을 이해한다.
- ◆ 재귀함수를 이해한다.

---

6.1 함수의 개념

6.2 함수의 구성과 구현

6.3 함수 매개변수의 전달

6.4 재귀 함수

---

## 6. 함수(function)

---

6.1 함수의 개념

6.2 함수의 구성과 구현

6.3 함수 매개변수의 전달

6.4 재귀 함수

---

- ➡ 함수는 인수를 전달 받아 일련의 작업을 수행한 후 생성된 결과를 전달하는 프로그램이다.
- ➡ C 프로그램은 일반적으로 작은 여러 개의 함수들이 모여서 이루어진 구조로 이루어져 있다.
- ➡ 함수는 많은 연산 작업들을 작은 여러 개의 연산 작업들로 세분하여 전체적으로 프로그램이 명확해지게 되고 수정을 용이하게 해준다.



## 함수

C 프로그램은 일반적으로 한 개 이상의 파일로 구성되어 있으며 반드시 main 함수를 포함한 여러 개의 함수로 구성되어진다. 이렇게 함수 단위로 시작해서 프로그램의 처리용도에 따라 각각 파일단위로 만들 수도 있고 하나의 파일에 전체 함수를 다 만들 수도 있다.

그러나 C 언어를 이용한 프로그램 기법에 있어서 일반적인 작성방법은 해결하고자 하는 문제를 부분적으로 분해하여 처리하는 것이 전체 문제를 쉽게 이해할 수 있을 뿐만 아니라 프로그램 작성시 쉽게 처리할 수 있는데 이러한 방법을 구조적 프로그램 기법이라 하며 다음과 같은 특징이 있다.

- C 언어에서의 함수는 각 함수를 서로 자유로이 호출할 수 있으며 자기자신을 호출할 수도 있다.
- C 언어에서의 함수는 모두가 동등한 단위 프로그램이다.
- 입출력 기능을 비롯한 대다수의 기능들이 표준함수로 제공된다.
- C 프로그램은 반드시 main() 이라는 이름의 함수를 갖는데 이 main() 함수는 C 프로그램의 시작점을 지정한다.

## 6. 함수(function)

---

6.1 함수의 개념

6.2 함수의 구성과 구현

6.3 함수 매개변수의 전달

6.4 재귀 함수

---

- 함수를 정의하는 일반적인 방식

```
[기억류] [함수의형] 함수명(가인수의 선언, ...)  
{  
    함수 내 변수의 선언;  
    실행문;  
}
```

함수의 종류

표준 함수 : C 언어 compiler 제작회사에서 미리 만들어 제공하는 함수  
사용자 정의 함수 : 프로그래머가 임의로 만들어 정의할 수 있는 함수

함수의 구조

- 함수명 : 만들어질 함수의 이름을 의미하며 함수를 호출할 때 사용하는 이름을 의미한다.
- 기억류 : 함수의 기억류는 다음 두 가지로 구성된다.

기억류	의 미
extern	함수를 다른 파일에서 참조할 수 있다.(default)
static	함수를 다른 파일에서 참조할 수 없다.

여기서 「다른 파일」이란 다른 compiler 단위(module)를 의미한다. static을 지정하면 compile가 함수명을 linker에 건네주지 않기 때문에 함수를 다른 module에서 사용할 수 없게 된다. 여러 명의 프로그래머가 협력하여 프로그램을 만들 경우 static으로 선언된 함수를 이용하면 그것은 "자신만의 함수(타인이 작성함 함수와 이름이 충돌하지 않음)"가 되는 것이다.  
함수의 기억류는 생략되는 것이 보통이며 특별한 목적이 있을 때에만 지정한다.

- 인수(argument) : 실인수 / 가인수  
↳ 함수끼리 서로 주고받는 data, 실인수 / 가인수

- 함수의 형

만일 호출된 함수에서 호출한 함수로 값을 돌려주어야 할 필요가 있을 때에는 return 문을 사용하여 값을 되돌려준다.

**return** 식;

**return** 이후에 나오는 문장은 무시하고 함수를 빠져나온다.

- 인수(parameter) : 호출함수가 피호출 함수에게 처리해야 할 데이터를 넘겨주기 위해서 인수가 필요하다. 인수에는 실인수(actual parameter)와 가인수(formal parameter)의 2종류가 있다.

실인수(actual parameter) : 호출 함수에 나타나는 인수

가인수(formal parameter) : 피호출 함수에 나타나는 인수

가인수 리스트는 실인수의 값을 전달 받기 위한 인수에 대한 선언으로 호출시의 인수와 그 순서나 개수, 각 인수의 자료형이 반드시 일치해야 한다. 일치하지 않으면 C의 형변환 규칙에 의해 변형된다.

- 함수의 형(return 형) : compiler에게 이 함수가 실행되었을 때의 결과값이 어떤 자료형인지 알려주는 역할을 하는 것으로 int, char, float 등의 형식이 사용된다.  
생략 시는 자동적으로 int형이 되며 함수의 결과값이 없는 경우는 void라고 규정한다.  
return 문은 피호출 함수의 계산 결과를 호출 함수에 돌려주기 위해서도 필요하고, 피호출 함수의 중간에서 제어를 호출함수로 옮기려고 할 때도 return문이 사용된다.

## ■ 함수 원형(prototype) 선언

```
int func(int x, int y);    // 세미콜론은 반드시!
```

- 이 프로그램에서는 func이라는 함수를 사용한다.
- 함수 func은 int형의 값을 return 하므로 적절히 처리하기 바란다.
- 함수 func의 첫 번째 인수와 두 번째 인수는 int 형인데 적합하지 않은 인수를 사용하면 경고하기 바란다.
- 변수명은 생략할 수 있다.

### 함수의 정의와 선언

함수를 사용하기 위해서는 함수의 정의와 함수의 선언이 선행되어야 한다. 어떤 함수를 사용하기 위해서는 먼저 그 함수를 만들어야 하고 이 함수가 어떤 형태로 구성되어 있는지에 대한 정보를 compiler에게 알려주고 그런 다음 만들어진 함수를 호출하여 임의의 처리에 이용한다.

함수의 정의 : 함수를 만드는 일

함수의 선언 : 만들어진 함수를 compiler에게 알리는 것( == 함수의 prototype)

만들어진 함수는 함수 사용이전에 compiler에게 선언으로 함수의 사용정보를 알려주어야만 하며 함수의 정의가 함수를 사용하는 위치보다 먼저 선행되어 있다면 함수의 정의와 선언이 동시에 이루어진다.

#### ▪ 사용자 정의 함수의 prototype

함수의 정의 header 부분에 ';'을 추가한 것으로 이것을 함수의 호출부분 이전(#include와 main() 함수 사이)에 위치시킨다. 호출한 쪽에서는 인수명을 자유롭게 사용하기 때문에 prototype 선언 안의 가인수명은 생략해도 되는 것이지만, 변수명을 써주는 것이 어떤 함수인지 알아보기가 쉽다.

#### ▪ 표준 함수의 prototype

모든 표준 함수는 지정된 이름의 header file에 prototype이 준비되어 있다.



```
/* 프로그램명, 화일명, 날짜, version 등의 일반적인 정보 */
#include < ....>    /* 표준 헤더 파일 */
#include " .... "   /* 사용자 정의 헤더 파일 */

#define .....      /* 매크로 상수나 매크로 함수를 정의 */

int function1( .... );      /* user define function 프로토타입 */
extern int n1, n2; /* 외부 변수 선언 */
/* 윗부분을 별도의 user define header file로 작성해도 무방함 */

int main(void) { ... }      /* main function 정의 */
int function1( ... ) { ... } /* user define function 정의 */
```

### 함수의 장점

- 반복되는 작업을 함수 화하여 coding 양을 줄일 수 있다.
- 구조적 프로그램 방식으로 module화 시킴에 따라 프로그램 작성 및 수정이 쉽다.
- 다른 프로그래머들에게 표준 library를 제공한다.
- 전문 지식을 요하는 분야에서는 각 내용별로 sub program을 작성하여 프로그램 작성시 도움을 준다.

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

```
int main(void)
```

```
{
```

```
    printf("*****\n");
```

```
    printf("hello\n");
```

```
    printf("*****\n");
```

```
    printf("hi\n");
```

```
    printf("*****\n");
```

```
    printf("good_bye\n");
```

```
    printf("*****\n");
```

```
    return 0;
```

```
}
```

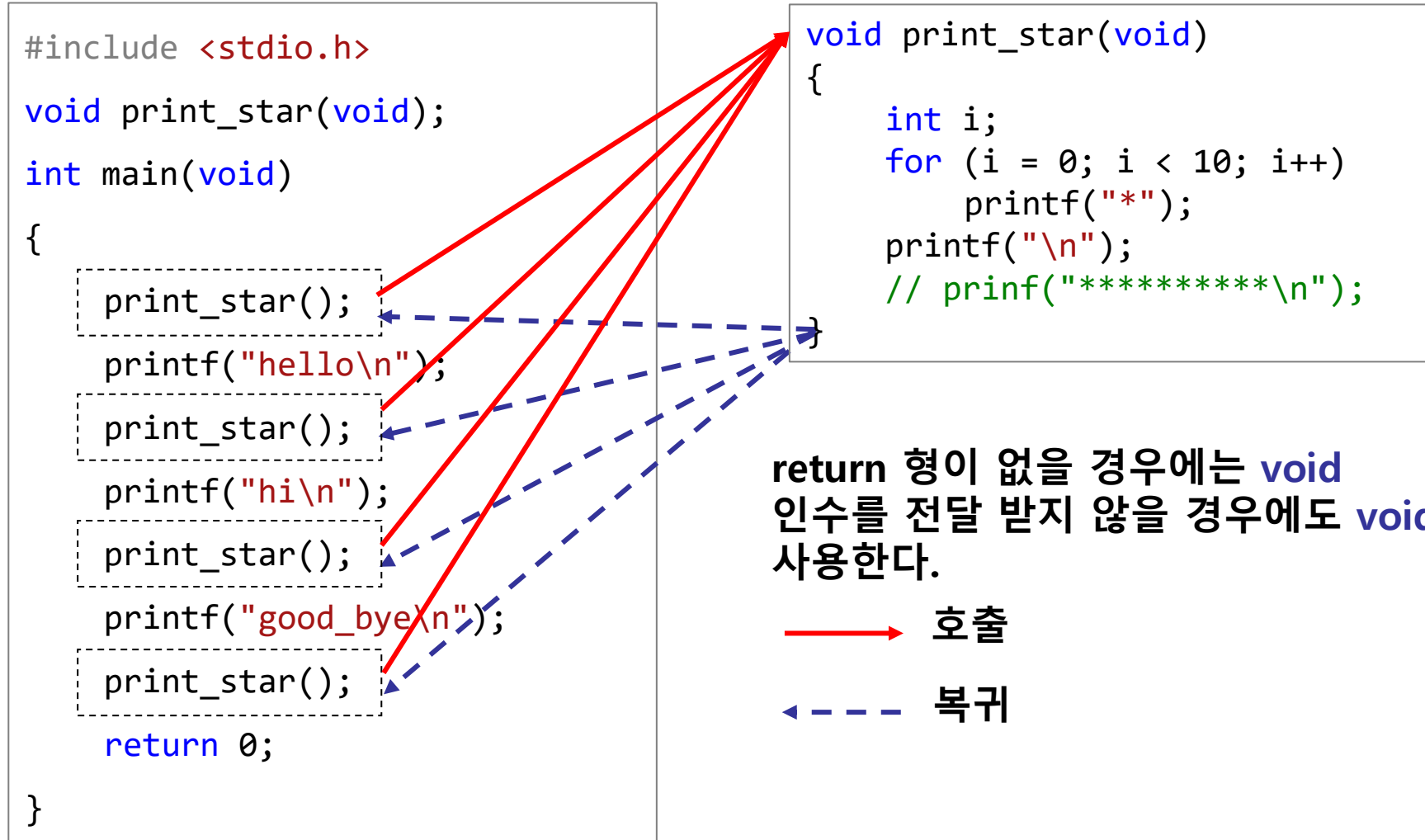
중복코드 :  
수정에서 비효율



**print\_star()**함수로 재구성

사용자 정의 함수를 사용하지 않는 경우

- main()함수 내의 printf() 함수를 이용하여 화면에 출력한다.



```
#include <stdio.h>

void print_star(void);
int main(void)
{
    int num;
    scanf("%d", &num);
    print_star(num);
    printf("hi\n");
    print_star(num);
    printf("good_bye\n");
    print_star(num);
    return 0;
}
```

```
void print_star(int num)
{
    int i;
    for (i = 0; i < num; i++)
        printf("*");
    printf("\n");
}
```

return 형이 없을 경우에는 void  
'\*'를 출력 개수를 조정할 num의  
실제 값이 인수(int num)로 전달됨.

→ 호출

← - - - 복귀

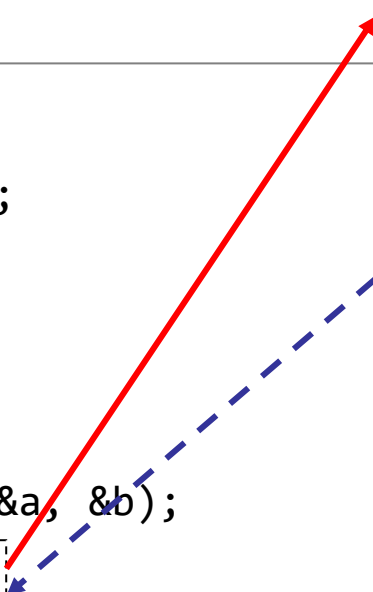
```
#include <stdio.h>

int main(void)
{
    int a, b, sum;
    scanf_s("%d %d", &a, &b);
    sum = a + b;
    printf("%d + %d = %d\n", a, b, sum);
    return 0;
}
```

a, b의 두 정수를  
더하는 명령행을  
하나의 함수로 작성

```
#include <stdio.h>
int add(int a, int b);
int main(void)
{
    int a, b, sum;
    scanf_s("%d %d", &a, &b);
    sum = add(a, b);
    printf("%d + %d = %d\n", a, b, sum);
    return 0;
}
```

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



피호출함수의 임의의 값을 호출함수에 전달하고 싶을 경우에는 return 문을 사용

# 6. 함수(function)

---

6.1 함수의 개념

6.2 함수의 구성과 구현

6.3 함수 매개변수의 전달

6.4 재귀 함수

---

- Call by value (값에 의한 전달)
  - C의 일반적인 함수 호출 방식
  - 변수의 값을 복사해서 전달하는 호출 방식
  - 값을 넘겨준 함수의 변수에는 아무런 영향을 미치지 않는다.
  
- Call by address (주소에 의한 전달)
  - 변수의 메모리 주소를 전달하는 호출 방식
  - 값을 넘겨준 함수의 변수에 영향을 미칠 수 있다.
  - 여러 개의 결과값을 한꺼번에 전달하는데 유용하게 사용되고, 기억장소의 절약 효과도 있다.



**call by value(값에 의한 전달)**

값에 의한 전달은 C에서는 가장 일반적인 인수 전달 방법이다. 호출하는 함수는 값을 설정하여 함수를 호출하고, 호출 받는 함수는 가인수에 그 값을 대입하고 나서 처리가 시작된다. 이때 실인수의 값은 메모리 상에 준비된 가인수에 복사된다.

즉, 가인수와 실인수가 별도의 기억장소를 유지하는 방법이다.

데이터 전달이 끝나면 실인수와 가인수는 서로 관계가 없어지므로 피호출 함수 측에서 조작한 가인수 값이 실인수의 값에는 영향을 미치지 않지만, 각 함수 내 변수의 독립성과 안정성에는 유익하다.

**call by address(주소에 의한 전달)**

주소에 의한 전달은 변수의 값이 아닌 변수가 존재하는 주소를 건네주는 방법이다.

각 함수 내의 변수가 불안정하긴 하지만, 연산에 의한 가인수 변경 값이 실인수에 영향을 미칠 수 있다.

실제로 일반적인 프로그래밍을 할 때에는 값에 의한 전달을 사용하고, 주소 전달 방법은 그 장점을 활용할 수 있을 때 이용하도록 한다.

예를 들어 배열의 모든 데이터를 값 전달하면 작업이 번거로워지지만, 주소 전달을 이용하면 배열의 시작 주소 값을 알려줌으로써 함수 측에서는 전달 받은 시작 주소 값에 의지하여 배열을 조작하기 때문에 작업이 용이해진다. 또한 주소 전달은 함수 측에서 여러 개의 리턴 값을 얻기 위해서도 유용하게 이용된다.

```
// call by value
#include <stdio.h>

void func(int num);

int main(void)
{
    int num = 10;
    func(num);
    return 0;
}

void func(int num)
{
    printf("num : %d\n", num);
}
```

```
// call by address
#include <stdio.h>

void func(int *num);

int main(void)
{
    int num = 10;
    func(&num);
    return 0;
}

void func(int *num)
{
    printf("num : %d\n", *num);
}
```

```
#include <stdio.h>
// call by value
void swap(int a, int b);

int main(void)
{
    int x = 20, y = 50;
    printf("[1] : %d, %d\n", x, y);
    swap(x, y);
    printf("[4] : %d, %d\n", x, y);
    return 0;
}

void swap(int a, int b)
{
    int temp;
    printf("[2] : %d, %d\n", a, b);
    temp = a;
    a = b;
    b = temp;
    printf("[3] : %d, %d\n", a, b);
}
```

```
#include <stdio.h>
// call by address
void swap(int* a, int* b);

int main(void)
{
    int x = 20, y = 50;
    printf("[1] : %d, %d\n", x, y);
    swap(&x, &y);
    printf("[4] : %d, %d\n", x, y);
    return 0;
}

void swap(int* a, int* b)
{
    int temp;
    printf("[2] : %d, %d\n", *a, *b);
    temp = *a;
    *a = *b;
    *b = temp;
    printf("[3] : %d, %d\n", *a, *b);
}
```

```
/* 반지름을 입력해 원의 넓이를 구함 */  
#include <stdio.h>  
  
float circle_area(int r);  
  
int main(void)  
{  
    int radius;  
    float area;  
    printf("Input radius : ");  
    scanf("%d", &radius);  
    area = circle_area(radius);  
    printf("area = %.2f\n", area);  
    return 0;  
}  
  
float circle_area(int r)    /* 원 넓이 */  
{  
    return 3.14f * r * r;  
}
```

```
/* 함수의 인자로 일차원 배열 넘겨주기 */
#include <stdio.h>
void input(int* ar);

int main(void)
{
    int i, array[5];
    input(array);
    for (i = 0; i < 5; i++)
        printf("%3d\n", array[i]);
    return 0;
}

void input(int* ar)
{
    int i;
    for (i = 0; i < 5; i++) {
        printf("[%d's] Number >> ", i + 1);
        scanf("%d", &ar[i]);
    }
}
```

```
/* 함수의 인자로 이차원 배열 넘겨주기 */
#include <stdio.h>

void func1(int(*a)[2]);

int main(void)
{
    int array[2][2] = { {1, 2}, {3,4} };
    func1(array);
    return 0;
}

void func1(int(*a)[2])
{
    int i, j;
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++)
            printf("a[%d][%d]=%3d\n", i,j,a[i][j]);
    }
}
```

# 6. 함수(function)

---

6.1 함수의 개념

6.2 함수의 구성과 구현

6.3 함수 매개변수의 전달

6.4 재귀 함수

---



- 동일한 작업을 단순하게 처리하기 위해 함수 내에서 자신의 함수를 호출하는 재귀적 호출 (recursive call) 방법
  - 프로그램은 매우 간결해진다.
  - 메모리를 많이 소비하고 처리속도가 느려진다.
- 함수 안에서 사용하는 변수는 자동변수
- 함수로의 인수 전달은 call by value
- 반드시 종료 조건을 명시

## 재귀함수

C에서는 함수를 재귀적으로 불러올 수 있다.

- 직접 순환(direct recursion) : 어떤 함수가 자기 자신의 함수를 직접 호출하는 경우
- 간접 순환(indirect recursion) : 어떤 함수가 다른 함수를 호출하면 그 함수에서 자기를 호출하거나, 혹은 일련의 함수를 통해서 간접적으로 자기 자신을 호출하는 경우이다.

## 장단점

순환을 사용하면 프로그램이 보기 쉽고, 간결하며, 오류 수정이 용이하다는 장점을 갖고 있지만 지나친 함수의 호출로 인하여 컴퓨터 수행 시간을 많이 할 우려가 있으며 함수 호출 시에 값들을 stack에 저장하므로 stack이 모자라거나 낭비를 가져올 우려가 있으므로 주의해서 사용해야 한다.

즉, 함수 안에서 선언된 자동 변수는 그 함수가 불러갈 때마다 메모리 상에 새로운 자동 변수로서 설정된다. 새로 설정된 자동 변수는 전 함수의 자동 변수 set과는 아무런 상관이 없다. 이처럼 재귀적 호출에서는 불러진 횟수만큼 필요 메모리를 소비하므로 stack 영역을 넘어가지 않도록 주의해야 한다.

- 자주 이용되는 예로 factorial 계산이 있다.

```
// 비재귀로 구현한 factorial
#include <stdio.h>

int main(void)
{
    int i, fact = 1;
    for (i = 1; i <= 5; i++)
        fact = fact * i;
    printf("5! = %d\n", fact);
    return 0;
}
```

$$5! = 5 * 4 * 3 * 2 * 1$$



$$5! = 5 * 4!$$



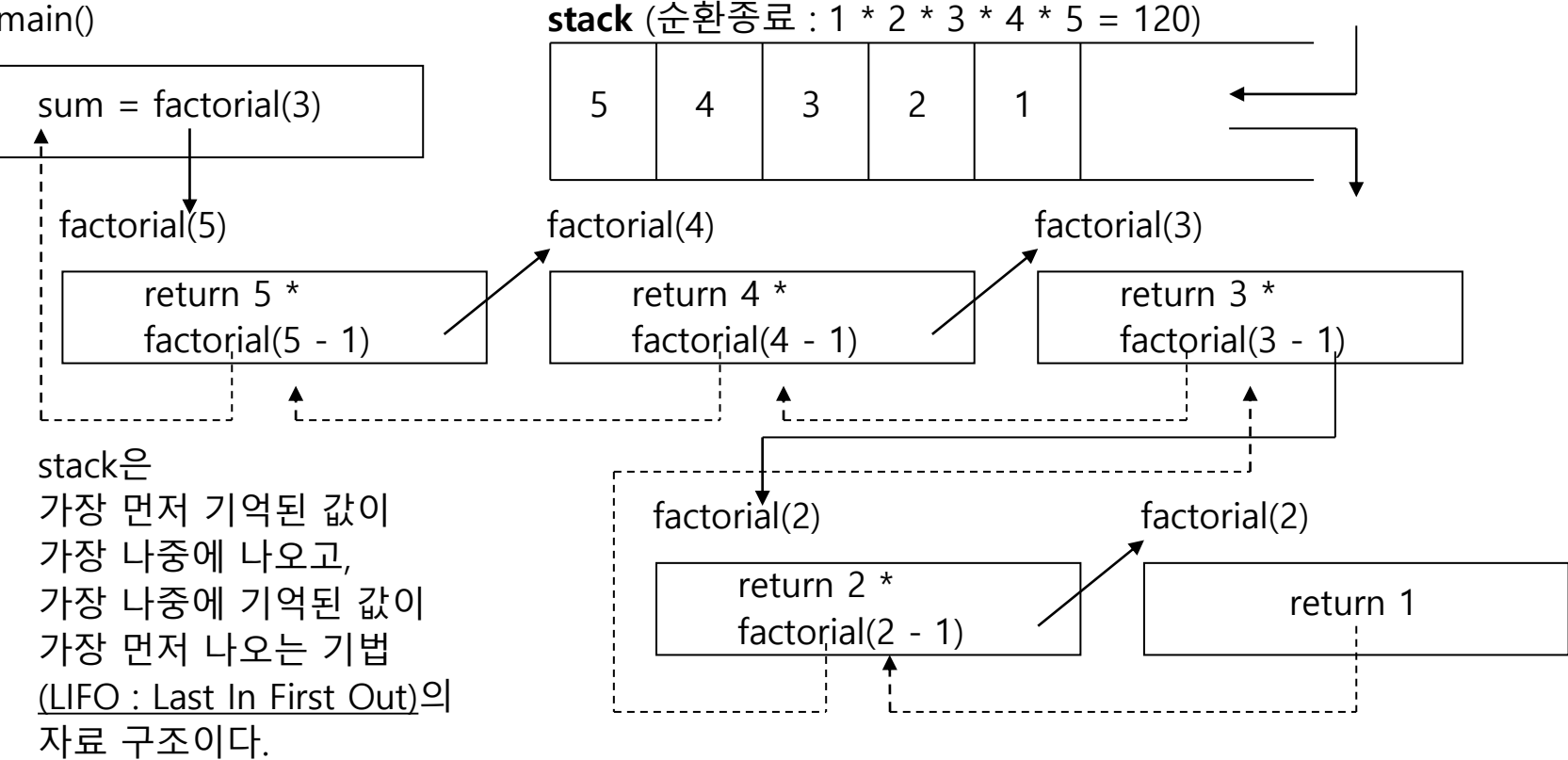
$$n! = n * (n - 1)!, 1! = 1$$

```
#include <stdio.h>

int factorial(int f);

int main(void)
{
    int fact = 1;
    fact = factorial(5);
    printf("5! = %d\n", fact);
    return 0;
}
// 재귀함수로 구현한 factorial
int factorial(int f)
{
    if (f == 1)
        return 1;           // 종료조건
    else
        return (f * factorial(f - 1));
                                // n! = n * (n - 1)!
}
```

함수의 순환을 이용해서 factorial 값을 구한다. 함수의 순환 기법을 이용하면 컴퓨터의 임시 기억 장소인 stack이라는 기억 장소에 factorial(f - 1)의 값이 1씩 감소하면서 자기 자신을 다시 호출하여 아래의 그림(stack) 형태로 기억되어 있다가 순환을 종료하면서  $f * \text{factorial}(f - 1)$ 의 값을 계산하여 호출 프로그램으로 return 한다.



- 연속된 숫자의 합을 구하는 함수를 만들고 이를 이용하여 합을 구하시오.

예) sum from 1 to 10 is 55

- 배열내의 원소들의 합을 구하는 함수를 만들고 이를 이용하여 배열 안에 들어있는 값들의 합을 구하시오.

예) {1,2,3,4,10,6,7,8,9,10} - sum of array is 60

- $(1 + 2 + \dots + 10) + (1 + 2 + \dots + 10 + 11) + \dots + (1 + 2 + \dots + 10 + 11 + \dots + 20)$ 의 합을 구하는 프로그램을 작성하시오.

- $^{\circ}\text{C} = (5/9)(^{\circ}\text{F} - 32)$ 라는 공식을 사용해서 화씨 온도를 섭씨 온도로 바꾸어 주는 프로그램을 작성하시오.
- 구구단을 출력하는 프로그램을 작성하시오.  
(함수의 인자로 이차원 배열, 일차원 배열을 전달하여)
- 영문 대문자로 이루어진 문장을 영문 소문자로 바꾸어 주는 `void my_tolower(char *str);` 함수를 작성하고 이를 이용하여 문자열을 변경하는 프로그램을 작성하시오.

- 영문 소문자로 이루어진 문장을 영문 대문자로 바꾸어 주는 `void my_toupper(char *str);` 함수를 작성하고 이를 이용하여 문자열을 변경하는 프로그램을 작성하시오.
- 문자열 `s`를 입력 받아 문자 't'의 가장 오른쪽 위치를 나타내는 함수 `strindex(s)`를 작성하시오. 만약 't'가 없다면 -1을 리턴하는 프로그램을 작성하시오.
- 하나의 문장을 입력 받아 거꾸로 출력해주는 `reverse` 함수를 작성하시오.

예) I am a student → tneduts a ma I



- fibonacci 수열을 구할 수 있는 재귀함수, 비재귀 함수를 작성하고 이를 이용하여 수열을 구하는 프로그램을 작성하시오.

(fibonacci 수열 :  $n\text{항} = (n - 1)\text{항} + (n - 2)\text{항}$ )

예) 1 1 2 3 5 8 13 19 31 ...

## 7. 변수의 기억류(Storage Class)



- ◆ 변수의 `scope`에 대해 이해한다.
- ◆ 기억의 영역 분류에 대해 이해한다.
- ◆ 동적메모리 할당/해제를 이해한다.

---

7.1 변수의 scope

7.2 기억의 영역 분류

7.3 동적메모리 할당/해제

---

# 7. 변수의 기억류(Storage Class)

---

7.1 변수의 scope

7.2 기억의 영역 분류

7.3 동적메모리 할당/해제

---

- 지역변수(local variable)  
함수 안에서 정의되어 그 함수 안에서만 참조가능  
예) auto, register, 함수 내 선언의 static
- 전역변수(global variable)  
함수 밖에서 정의되어 어떤 함수에서든 참조가능  
예) extern, 함수 외 선언의 static

☞ 지역 변수의 할당과 회수는 하나의 함수 내에서도 발생함

- 함수 내에서 새로운 블록을 열고 변수를 선언하면 그 블록이 끝날 때 변수가 사라짐

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

```
int main()
```

```
{
```

```
    int a=10, b=20;
```

```
    printf("바꾸기 전 a: %d, b: %d\n", a, b);
```

```
    {
```

```
        int temp;
```

```
        temp=a;
```

```
        a=b;
```

```
        b=temp;
```

```
    }
```

```
    printf("바꾼 후 a: %d, b: %d\n", a, b);
```

```
    return 0;
```

```
}
```

변수 a, b 의 사용범위

변수 temp의 사용범위

- 중첩된 블록에서 같은 이름의 변수를 선언하면 가장 가까운 블록에 선언된 변수에 우선권이 있음

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

```
int main()
```

```
{
  int i; ①
  i = 5;
```

```
{
  int i; ②
  i = 3;
```

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

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

```
}
```

①번 변수 범위

②번 변수 범위

가장 가까이 선언된  
2번 변수를 사용한다.

출력 결과

```
i=3 // ② 번 출력
i=5 // ① 번 출력
```

☞ 아래 코드의 결과를 예상해 보고, 실행 결과를 확인해 보자.

```
#include <stdio.h>

void func(int a);
int a, b; // 전역 변수 선언

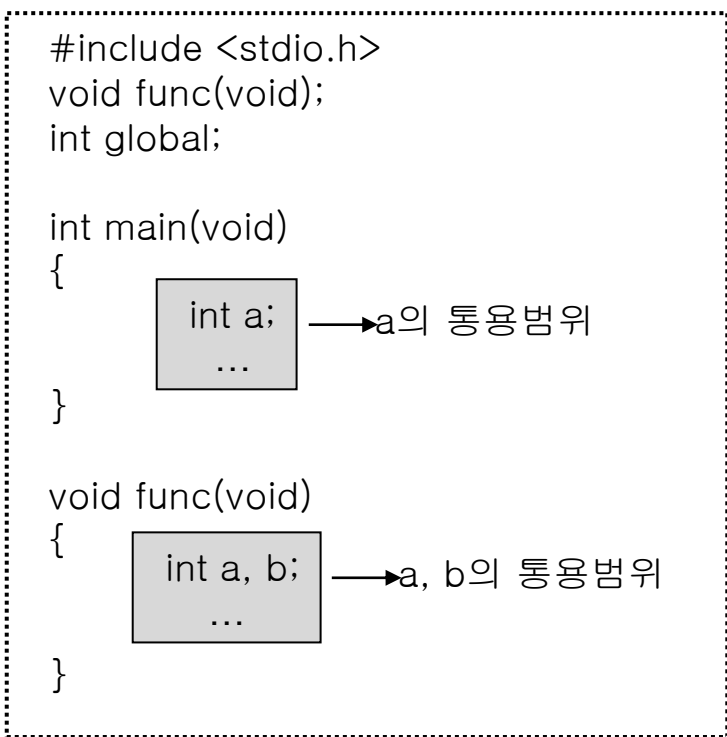
int main()
{
    a=5;
    func(a);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}

void func(int a)
{
    b=2;
    a+=2;
}
```



### 변수

- 지역변수 : 프로그램내의 함수가 실행될 때마다 필요한 공간이 할당되고 routine의 실행종료와 더불어 소멸된다.
- 전역변수 : compile할 때 기억장소가 할당되며, 할당된 기억장소는 프로그램의 실행에서 종료 시까지 계속 존재 하도록 기억 공간에 할당하는 것으로 compile시 미리 main memory 공간에 확보하는 것이다. 이렇게 확보된 공간은 프로그램 종료 시 소멸된다.



→ global의 통용범위

- global은 전역 변수.
- main()의 a, func()의 a,b는 지역 변수.
- 전역 변수 global는 main()에서도 func()에서도 참조가능.
- 지역 변수 main()의 a는 main()에서만 참조가능.
- 지역 변수 func()의 a,b는 func()에서만 참조가능.
- 지역 변수 main()의 a와 func()의 a는 충돌하지 않음.

# 7. 변수의 기억류

---

7.1 변수의 scope

7.2 기억의 영역 분류

7.3 동적메모리 할당/해제

---

➡ C 프로세스는 메모리를 용도 및 특성에 따라 나누어 사용

➡ 메모리를 나누는 기준

- 텍스트 영역 : 실행할 프로그램 코드 및 상수를 올려 놓을 공간  
.text 읽기 전용 영역; 프로그램 코드  
.rodata 읽기 전용 영역; global const, 문자열("...")
- 데이터 영역 : 프로그램 종료 시까지 유지할 데이터 저장 공간  
.data RW 영역; 전역/정적 변수 중 초깃값을 갖는 변수  
.bss RW 영역; 전역/정적 변수 중 초깃값을 갖지 않는 변수  
➔ main() 함수 실행 전 일괄 초기화
- 스택 영역 : 잠시 사용하고 버릴 데이터 저장 공간; 지역 변수
- 힙 영역 : 언제든지 할당하고 사용 후 해제할 수 있는 데이터 저장 공간;  
malloc() 함수로 할당 & free() 함수로 해제

메모리  
전체 영역

텍스트 영역

데이터 영역

힙 영역

스택 영역

- ➡ 정적 변수는 함수가 리턴된 후에도 기억 공간이 존재함
- ➡ 정적 변수는 자료형 앞에 **static** 예약어를 사용하여 선언(예: static int num;)
- ➡ 프로그램이 실행되는 시점에 메모리 공간에 할당 및 초기화가 이루어짐
- ➡ 정적 지역 변수와 정적 전역 변수로 사용이 가능함

```
#include <stdio.h>
void PrintCount();
int main()
{
    int i;
    for(i=0; i<5; i++) {
        PrintCount();
    }
    return 0;
}
```

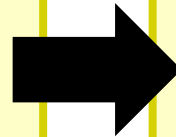
```
void PrintCount()
{
    static int count=0;
    count++;
    printf("%d번째 호출\n", count);
}
```

### 출력 결과

```
1번째 호출
2번째 호출
3번째 호출
4번째 호출
5번째 호출
```

- 정적 지역 변수는 전역 변수처럼 계속해서 유지해야 하지만, 언제 어디서나 접근 가능한 것이 아니라 하나의 함수 내에서만 접근하도록 제약할 때 사용
- 정적 전역 변수는 전역 변수로 선언해서 잘못된 접근의 위험에 노출시키기보다는 하나의 파일 내에서만 접근이 가능하도록 제약할 때 사용; 정적 함수도 마찬가지임

```
void bubbleSort(int value[], int count)
{
    int i=0, j=0;
    int temp = 0;
    for( i = 0 ; i < count; i++ )
    {
        for( j=0; j < count - (i+1); j++ )
        {
            if(value[j] > value[j+1])
            {
                temp = value[j+1];
                value[j+1] = value[j];
                value[j] = temp;
            }
        }
        printf("Step-%d", i);
        printArray(value, count);
    }
}
```



```
void bubbleSort(int value[], int count)
{
    int i=0, j=0;
    for( i = 0 ; i < count; i++ )
    {
        for( j=0; j < count - (i+1); j++ )
        {
            if(value[j] > value[j+1])
            {
                static int temp=0;
                temp = value[j+1];
                value[j+1] = value[j];
                value[j] = temp;
            }
        }
        printArray(value, count);
    }
}
```

< count.c >

// count.c 파일 내에서만 접근하도록 함  
**static int cnt = 0;**

```
void AddCnt()
{
    cnt++;
}
```

```
void SubCnt()
{
    cnt--;
}
```

```
int GetCnt()
{
    return cnt;
}
```

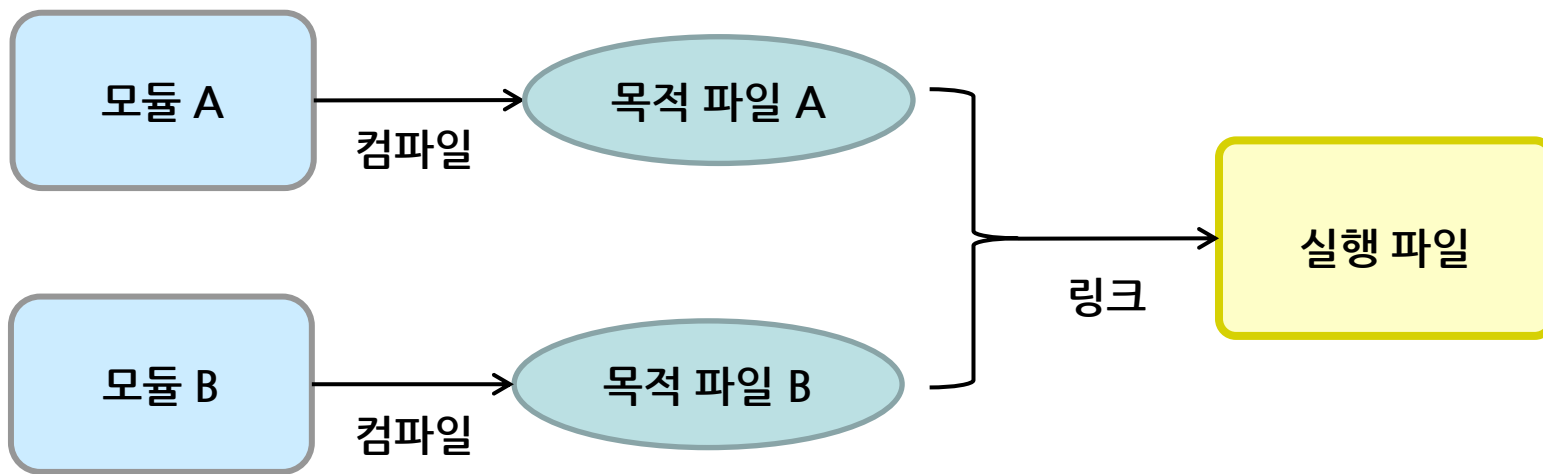
< main.c >

#include <stdio.h>

```
extern void AddCnt(); // 다른 파일에 있는 함수임
extern void SubCnt();
extern int GetCnt();
```

```
int main()
{
    AddCnt();
    AddCnt();
    printf("Count 값: %d\n", GetCnt());
    SubCnt();
    printf("Count 값: %d\n", GetCnt());
    return 0;
}
```

- ➡ 하나의 프로그램은 기능에 따라 많은 함수들을 포함
- ➡ 이 함수들을 여러 개의 파일 단위로 묶어서 작성할 수 있으며 각각을 모듈이라고 함
- ➡ 하나의 프로그램을 여러 개의 모듈로 나누어서 작성하는 것을 분할 컴파일이라고 함



- ➡ 다른 모듈(파일)에 있는 전역 변수를 자신의 변수인 것처럼 사용
- ➡ 예약어 extern을 사용하여 외부 변수를 사용함
- ➡ extern을 사용한 외부 변수는 실제로 기억 공간이 할당되는 것이 아니며, 다른 모듈에 있음을 컴파일러에 알려주는 역할만 함

```
extern int a, b;
```



➡ 하나의 파일에 아래 코드를 작성해 보자.

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

```
int num;
```

```
void Increment()  
{  
    num++;  
}
```

```
int GetNum()  
{  
    return num;  
}
```

```
int main()  
{  
    printf("num: %d\n", GetNum());  
    Increment();  
    printf("num: %d\n", GetNum());  
    Increment();  
    printf("num: %d\n", GetNum());  
  
    return 0;  
}
```

출력 결과

```
num : 0  
num : 1  
num : 2
```

☞ 앞에서 작성한 파일을 총 3개의 파일로 나누어 컴파일해 보자. (분할 컴파일 적용)

< num.c >

```
int num = 0;
```

< func.c >

```
void Increment()
{
    num++;
}
```

```
int GetNum()
{
    return num;
}
```

< main.c >

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

```
int main()
{
    printf("num: %d\n", GetNum());
    Increment();
    printf("num: %d\n", GetNum());
    Increment();
    printf("num: %d\n", GetNum());

    return 0;
}
```

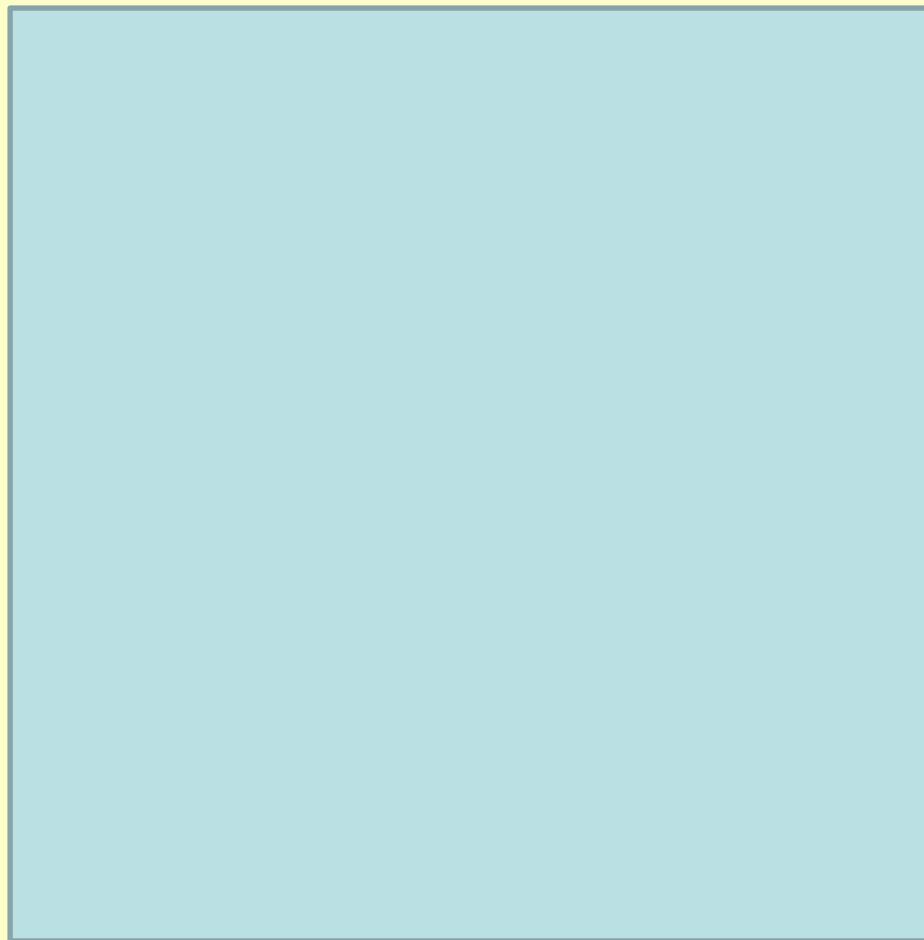
컴파일 결과는?

컴파일 에러를  
수정해보자!!

☞ 다음 변수들이 사용하는 메모리 영역은 어디인가?

```
int a1;           // ?
int a2 = 10;      // ?
static int b1;    // ?
static int b2 = 10; // ?

int main(void)
{
    int x1;        // ?
    int x2 = 100;  // ?
    static int y1;  // ?
    static int y2 = 200; // ?
    int *p = (int *)malloc(100); // ?
    b1 = 10;
    return 0;
}
```



# 7. 변수의 기억류

---

7.1 변수의 scope

7.2 기억의 영역 분류

7.3 동적메모리 할당/해제

---

- 실행시간에 할당되어 사용되는 메모리 블록
- 필요할 때 메모리를 할당 받고, 필요 없을 때 해제  
➔ heap 영역에 할당
  - 메모리 블록 할당 : `malloc()`, `calloc()`
  - 메모리 블록 해제 : `free()`
- 단순히 매개변수에 지정된 크기만큼의 메모리를 할당하고 그 메모리의 주소(**pointer**)를 반환
- 프로세스의 힙 영역에 더 이상 쓸 수 있는 메모리 공간이 없으면 **NULL**을 return

#### 메모리 확보의 문제점

C에서 데이터를 저장하기 위해 필요로 하는 메모리 영역은 일반적으로 변수 선언으로 확보된다. 이 때 외부 변수와 정적 변수용 영역은 프로그램의 실행 중 항상 고정적으로 확보되어 있다. 이러한 영역의 확보는 C 시스템에 의해서 효율적으로 처리가 되긴 하지만,

프로그램의 개시 전에 필요로 하는 메모리 크기를 결정해 뒀야 한다.

는 조건이 있다. 또한

여유 있는 크기의 기억 영역을 확보해 뒀야 한다.

확보한 기억 영역보다 1 byte라도 더 사용하면 이상 작동을 일으킬 가능성이 있다.

는 점을 고려해야 한다.

#### 동적 메모리 할당

위와 같은 문제점의 해결은 동적 메모리 할당함으로써 해결할 수 있다.

동적 할당을 이용하면 프로그램이 시작되고 나서 필요한 메모리를 필요할 때마다 비어 있는 자유 메모리 공간에서 확보할 수 있다. 또한 그 메모리가 불필요해지면 그 때마다 해제할 수 있고, 해제한 메모리 공간은 다음 번 동적 메모리 확보에 재이용할 수 있다.

```
void *malloc( size_t size );  
void *calloc( size_t num, size_t size );  
void free( void *memblock );
```

```
int *ptr;  
ptr = (int *)malloc(100 * sizeof(int)); // 메모리 할당 library func.  
if(ptr == NULL) {                       // 메모리 확보에 실패하면  
    printf("메모리 할당 실패\n");        // 메시지를 출력하고  
    exit(0);                           // 프로그램을 중단한다.  
}  
...  
free(ptr);                             // 메모리 해제 library func.
```

malloc()의 return 값은 void \*이므로 사용하고 싶은 데이터 형으로 type cast !!

#### 동적 메모리 할당시의 유의점

- `'free'`에 전달되는 포인터는 반드시 `'malloc'`이나 `'calloc'`, `'realloc'`에 의해 할당 받은 메모리의 첫 시작 주소를 나타내는 값이어야 한다.
- 10 byte 크기의 블록을 할당 받았다면 반드시 10 byte 크기를 해제해야 한다.
- `'free'`를 쓸 때에 이미 해제된 블록의 포인터를 또 쓸 경우, 어떠한 일이 벌어질지 보장할 수 없다. 한 번 할당해놓고, 두 번 해제하는 실수를 범하지 말아야 한다.
- 일단 `'free'`를 불러서 해제한 메모리는 어떠한 데이터를 저장하고 있을지 보장할 수 없다. 즉, 데이터가 필요하다면 다른 곳에 저장 시킨 후 `'free'`를 호출해야 한다.



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

int main(void)
{
    char* ptr;
    int size;
    printf("memory bytes >> ");
    scanf("%d", &size);
    if ((ptr = (char*)malloc(size)) == NULL)
    {
        // 메모리 할당
        printf("메모리 할당에 실패하였습니다.\n");
        exit(0);
    }
    getchar();
    printf("string >> ");
    gets(ptr);
    printf("%s\n", ptr);
    free(ptr); // 메모리 해제
    return 0;
}
```

## 8. 구조체(struct)



- ◆ 구조체의 개념을 이해한다.
- ◆ 구조체의 기본 문법을 이해한다.
- ◆ 구조체 배열/포인터를 이해한다.
- ◆ 구조체의 다양한 문법을 이해한다.

---

**8.1 구조체의 개념**

**8.2 구조체의 기본 문법**

**8.3 구조체 배열과 포인터**

**8.4 구조체의 다양한 문법**

---

## 8. 구조체(struct)

---

8.1 구조체의 개념

8.2 구조체의 기본 문법

8.3 구조체 배열과 포인터

8.4 구조체의 다양한 문법

---

- 여러 개의 서로 다른 데이터 형 변수들을 묶어서 하나의 단위로 처리할 수 있도록 구조화 시킨 자료 구조의 형태

```
struct tag {  
    데이터형 멤버명1;  
    데이터형 멤버명2;  
    ...  
    데이터형 멤버명n;  
};  
struct tag identifiers;
```

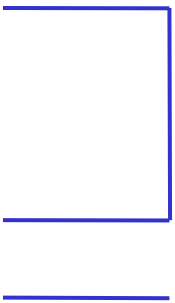
```
struct tag {  
    데이터형 멤버명1;  
    데이터형 멤버명2;  
    ...  
    데이터형 멤버명n;  
} identifiers;
```

구조체의 선언과 변수의 선언 방법

구조체로 선언된 변수는 일반 변수와 동일하게 취급된다. 일반 변수에 기억 클래스를 지정할 수 있는 것처럼 구조체 변수도 필요하다면 기억 클래스를 앞부분에 붙일 수 있다.

- 구조체의 구조를 선언하고 나서 그 구조체를 갖는 변수를 선언할 수도 있고 동시에 기술할 수도 있다.

```
struct student {  
    int student_id;  
    char student_name[10];  
    int age;  
}stu1;  
struct student stu2;
```

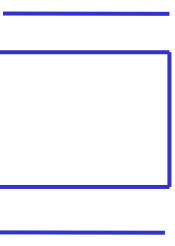


구조체 정의/  
stu1변수 선언

stu2변수 선언

- tag명을 다시 사용하지 않는다면 그것을 생략하고 「tag명은 없지만 그 구조체 형태로 변수」를 선언할 수 있다.

```
struct {  
    int student_id;  
    char student_name[10];  
    int age;  
}stu;
```



tag명은 없지만  
이 구조체 형태로  
stu변수 선언

## 8. 구조체(struct)

---

8.1 구조체의 개념

8.2 구조체의 기본 문법

8.3 구조체 배열과 포인터

8.4 구조체의 다양한 문법

---

- 각 멤버에 대응하는 초기값을 { } 안에 나열

```
struct student {  
    int stu_id;           // 학 번  
    char stu_name[10];    // 이 름  
    int stu_age;          // 나 이  
}stu1 = {1, "김철수", 15};  
  
struct student stu2 = {2, "김영수", 14};
```

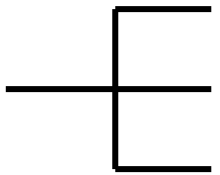


구조체의 메모리 확보

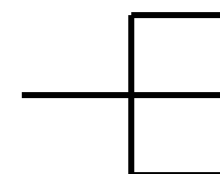
구조체 선언은 student 형이라는 새로운 틀을 만들뿐이다. 즉, 구조체 형식 선언은 실제로 기억 장소를 확보하는 것이 아니라 데이터의 구성 형식을 선언하는 것에 불과하다. 실제로 사용하기 위해서는 student 형을 갖는 구체적인 변수명을 선언하고 필요한 메모리 영역을 확보해야 한다.

	stu_id(int)	stu_name(char)	stu_age(int)
stu1	1	김철수₩0	15
	stu_id(int)	stu_name(char)	stu_age(int)
stu2	2	김영수₩0	14

- "."(membership operator)에 의해 지정 :  
구조체 변수의 멤버를 지정한다.

struct student stu;		stu.stu_id
		stu.stu_name
		stu.stu_age

- "->"(right arrow operator)에 의해 지정 :  
구조체 포인터 변수의 멤버를 지정한다.

struct student *stu;		stu->stu_id == (*stu).stu_id
		stu->stu_name
		stu->stu_age

**membership operator (도트 연산자)**

이 연산자는 구조체 변수 자체의 멤버를 지정할 때 이용한다. 예를 들어 변수 `stu`의 멤버인 `student_id`에 3을 대입하고, 그 값을 출력하는 방법은 아래와 같다.

```
struct student stu;  
stu.stu_id = 3;  
printf("%d\n", stu.stu_id);
```

**right arrow operator (화살표 연산자)**

이 연산자는 구조체 포인터에 의해 멤버를 참조할 때 이용한다. 구조체 변수는 일반 변수와 마찬가지로 포인터로 조작할 수 있다.

```
struct student stu;  
struct student *ptr = &stu;
```

라고 선언하면 포인터 `ptr`은 구조체 변수 `stu`의 시작 주소 값을 가리킨다. 이 포인터 `ptr`을 이용해 멤버를 지정하는 경우 도트 연산자를 사용할 수도 있다. 「`*ptr`」라고 하면 값을 나타내므로 도트 연산자로 멤버 참조를 할 수 있다.(단, 우선순위에 주의해야 함)

```
ptr->stu_id = 3;           // (*ptr).stu_id = 3;  
printf("%d\n", ptr->stu_id); // printf("%d\n", (*ptr).stu_id);
```

```
#include <stdio.h>

struct time
{
    int hours;
    int minutes;
    int seconds;
};

int main(void)
{
    struct time birth = { 8, 45, 23 };
    struct time* ptr = &birth;
    printf("birth time - %d : %d : %d\n",
        ptr->hours, ptr->minutes, ptr->seconds);
    return 0;
}
```

```
#include <stdio.h>

typedef struct time
{
    int hours;
    int minutes;
    int seconds;
} time;

int main(void)
{
    time birth = { 8, 45, 23 };
    time* ptr = &birth;
    printf("birth time - %d : %d : %d\n",
        ptr->hours, ptr->minutes, ptr->seconds);
    return 0;
}
```

```
#include <stdio.h>

typedef struct time
{
    int hours;
    int minutes;
    int seconds;
} time;

int main(void)
{
    time birth = { 8, 45, 23 };
    time* ptr = &birth;
    printf("birth time - %d : %d : %d\n",
        ptr->hours, ptr->minutes, ptr->seconds);
    return 0;
}
```

## 8. 구조체(struct)

---

8.1 구조체의 개념

8.2 구조체의 기본 문법

8.3 구조체 배열과 포인터

8.4 구조체의 다양한 문법

---

- 구조체 형태로 배열을 선언할 수 있다.

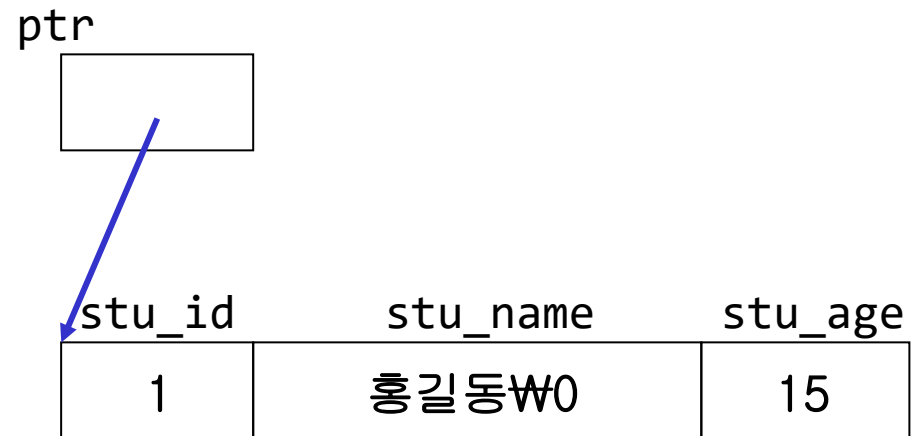
예) `struct student stu[4];`

<code>stu[0].stu_id</code>	<code>stu[0].stu_name</code>	<code>stu[0].stu_age</code>	<code>stu[0]</code>
<code>stu[0].stu_id</code>	<code>stu[0].stu_name</code>	<code>stu[0].stu_age</code>	<code>stu[1]</code>
<code>stu[0].stu_id</code>	<code>stu[0].stu_name</code>	<code>stu[0].stu_age</code>	<code>stu[2]</code>
<code>stu[0].stu_id</code>	<code>stu[0].stu_name</code>	<code>stu[0].stu_age</code>	<code>stu[3]</code>



- 구조체 포인터 변수를 선언할 수 있다.

```
struct student {  
    int stu_id;  
    char stu_name[10];  
    int stu_age;  
}stu = {1, "홍길동", 15};  
struct student *ptr = &stu;
```



## 8. 구조체(struct)

---

8.1 구조체의 개념

8.2 구조체의 기본 문법

8.3 구조체 배열과 포인터

8.4 구조체의 다양한 문법

---

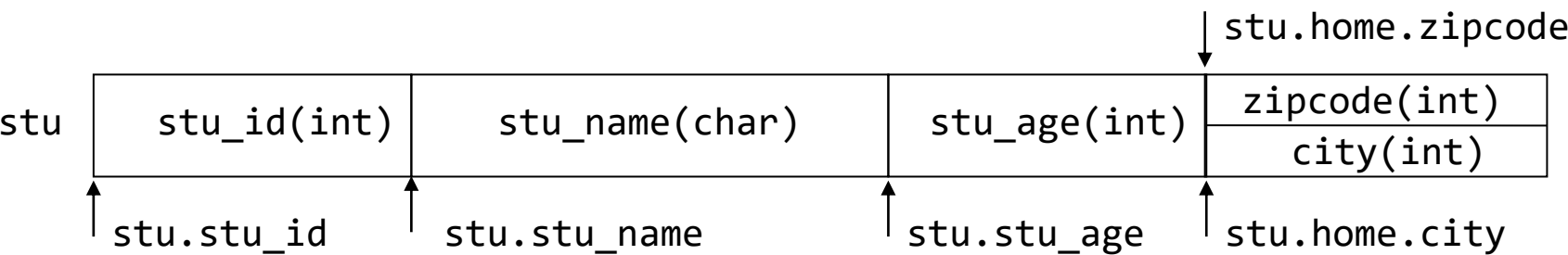
- 구조체 안에서 다시 구조체를 선언할 수 있다.

```
struct address
{
    int zipcode;
    int city;
};
```

```
struct student
{
    int stu_id;
    char stu_name[10];
    int stu_age;
    struct address home;
};
struct student stu;
```

nested 구조체

구조체는 멤버에 다른 구조체를 포함할 수 있다. 이러한 내부 구조체의 최종 멤버를 참조하려면 멤버 연산자를 여러 개 사용하면 접근할 수 있다.



- 아래의 입력 데이터는 학생 4명의 학번과 이름 그리고 각 학생들의 국어, 영어, 수학 점수이다. 아래 입력 데이터를 바탕으로 아래와 같은 출력 결과를 산출하는 프로그램을 작성하시오.  
(단, 등급은 A : 90-100, B : 80-89, C: 70-79, F: 0-59 이며, 함수의 인수는 반드시 구조체 포인터를 이용할 것)

9200313	임꺽정	90	85	90
9200311	김영철	90	85	80
9200314	이영희	65	70	100
9200312	홍길동	50	75	90

*****							
번호	이름	국어	영어	수학	총점	평균	등급
*****							
9200311	김영철	90	85	80	255	85	B
9200312	홍길동	50	75	90	215	71	C
9200313	임꺽정	90	85	90	265	88	B
9200314	이영희	65	70	100	235	78	C
*****							