

# 목차

## 1. 프로그래밍 기초

1. 운영체제 기초
2. 프로그래밍 기초

## 2. 리눅스 기초

1. vi 편집
2. 리눅스 파일시스템

## 3. c 언어 기초

1. c 언어 기초
2. c 프로그램의 구성요소

## 4. 저장

1. 자료형과 변수 선언
2. 정수의 저장
3. 문자의 저장
4. 논리값의 저장
5. 실수의 저장
6. 복소수의 저장
7. 형변환

## 5. 수식과 연산자

1. 수식과 연산자

## 6. 제어의 흐름

1. 조건문
2. 반복문
3. 점프문

## 7. 배열과 정렬

1. 배열
2. 다차원 배열
3. 함수와 배열
4. 정렬

## 8. 함수

1. 함수
2. 함수의 정의, 호출, 선언
3. 함수의 매개변수 전달법
4. c 의 여러가지 함수들

## 9. 변수

1. 지역변수와 전역변수
2. 스토리지 클래스

## 10. 문자열

1. 문자열
2. 문자열 처리 함수

## 부록 1. 파일 입출력

1. 리다이렉션
2. c 의 파일 입출력 함수

# 1. 프로그래밍 기초

## 1. 운영체제 기초

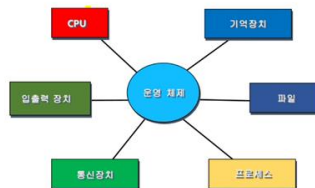
### 1. 운영체제(Operating System, OS)란?

사용자와 컴퓨터 하드웨어 사이에서 인터페이스 역할을 하는 프로그램.  
컴퓨터를 켜는 순간부터 종료할 때까지 계속 동작함  
운영체제가 존재해야 컴퓨터를 사용할 수 있음



### 2. 운영체제가 하는 일

사용자 프로그램, 응용 프로그램을 수행함  
여러 기능들을 이어줌.  
컴퓨터 자원을 관리함



### 3. 운영체제의 종류와 특징

#### 1) UNIX

1969년 벨 연구소에서 켄 톰슨 등이 만들

현재 사용되는 대부분의 운영체제에 영향을 줌

c언어로 작성됨

계층적 파일 시스템

일관적인 파일 포맷(바이트 스트림)과 디바이스 접근(?)

다중 사용자와 다중 작업

**메모 포함[이1]:** 데이터를 바이트 단위로 주고받는 것을 말함

#### 2) LINUX

1991년 헬싱키 대학의 리누스 토르발스가 만들

UNIX의 영향을 크게 받음

오픈 소스 소프트웨어 운영체제임

무료임

기본적으로 CLI 기반임

서버, 임베디드 시스템, IOT 디바이스에서 보편적으로 사용됨

1992년에는, 1984년에 리처드 스톨만이 만든 GNU프로젝트의 커널로 선택됨

**메모 포함[이2]:** Command Line Interface.  
텍스트를 통해 상호작용하는 방식.

**메모 포함[이3]:** 제어를 위한 특정 기능을 수행하는 컴퓨터 시스템.

**메모 포함[이4]:** Internet Of Things. 사물인터넷.  
무선 통신을 통해 각종 사물을 연결하는 기술.

+ windows 는 LINUX 와는 달리 GUI 기반이다.

**메모 포함[이5]:** Graphical User Interface.  
그래픽을 통해 상호작용하는 방식.

#### + Cygwin

LINUX를 설치하지 않아도 Windows에서 CLI기반의 LINUX를 사용할 수 있게 해주는 공개 소프트웨어  
GNU 프로젝트가 개발한 다양한 툴들을 Windows API 환경에 포팅하는 것이 최종 목표이다.

cygwin에서는 리눅스 명령어와 배시 셸을 사용할 수 있음.

윈도우에서 리눅스 환경을 구축할 수 있게 해주지만, 그렇다고 리눅스와 똑같지는 않음.

**메모 포함[이6]:** LINUX에서 사용할 수 있는 GUI가 개발되긴 했지만, 개발과 관련된 일들에 있어서는 모든 기능을 전부 다룰 수도 없고 효율도 떨어지기 때문에 개발자라면 CLI기반의 LINUX를 다룰 줄 알아야 한다.

**메모 포함[이7]:** Application Programming Interface.  
응용 프로그램이 사용할 수 있도록  
운영 체제나 프로그래밍 언어가 제공하는 기능을 제어할 수 있게 만든 인터페이스.

**메모 포함[이8]:** = 이식.

## 2. 프로그래밍 기초

### 1. 프로그래밍이란?

프로그래밍이란 알고리즘을 프로그래밍 언어로 구현하는 일. 프로그램을 만드는 행위를 말함.

### 2. 프로그램이란?

컴퓨터와 대화하는 수단.

### 3. 프로그래밍 언어란?

CPU는 처리할 수 있는 고유의 명령어 집합이 있고, 그것은 **기계어**로 이루어져 있음.  
하지만 기계어로 코딩하는 것은 쉽지 않기에 프로그래밍 언어가 개발됨.  
다양한 프로그래밍 언어가 있고, 그 용도가 가지각색임.

**메모 포함[이9]:** 이진수로 이루어진, 컴퓨터가 알아들을 수 있는 언어.

### 4. 프로그래밍 환경이란?

프로그래밍 개발에 필요한 문서 편집기, **컴파일러**, 디버거 등 도구들의 집합.  
프로그래밍 도구들이 하나로 통일된 것을 통합 프로그래밍 환경이라고 함. (비주얼 스튜디오 등)

**메모 포함[이10]:** 컴파일러는 프로그래밍 언어를 기계어로 번역해 주는 역할을 함.  
(컴파일러는 소스 파일을 실행 파일로 번역함.)  
다양한 종류의 컴파일러가 존재함

## 5. 프로그래밍의 단계

프로그램 작성 -> 컴파일 -> 실행

### 1) 프로그램 작성

문서 편집기로 작성

텍스트 파일로 저장해야 함 (컴파일러가 이해할 수 없는 내용이 추가로 들어갈 수 있기 때문.)

c언어 컴파일러에게 읽히려면 해당 파일의 확장자는 .c 로 해야함.

알아보기 용이한 파일명으로 작성해야 함

### 2) 컴파일

통합 환경이라면 컴파일 메뉴 사용. 우분투에서는 컴파일 명령어를 입력.

실행 파일이 생성됨.

오류가 발생했다면 실행 파일이 생성되지 않고 오류 메시지가 뜸

여러 개의 파일로 구성된 프로그램을 컴파일할 때에는 해당 파일들을 gcc의 인자로 모두 작성하면 됨.

### 3) 실행

통합 환경이라면 실행 메뉴 사용, 그렇지 않다면 명령 입력.

실행 파일의 경로를 입력하면 실행됨.

gcc <file>  
(우분투에서 컴파일하는 방법)

**메모 포함[이11]:** 실행 파일명은  
리눅스에서는 a.out으로,  
씨그윈에서는 a.exe로,  
윈도우에서는 파일명.exe로 생성됨

**메모 포함[이12]:** 즉, \*.c 인 파일들만 인자로 작성할 수 있다.  
파일 입출력으로 다루는 파일은 이곳에 작성하는 것이 아니다.

## 6. 순서도 (Flowchart)










문제 처리 과정을 다이어그램으로 표현한 것.

모든 프로그래밍 언어는 순서도의 흐름을 표현할 수 있는 구문들을 제공함.

-> 순서도를 보면 프로그램을 쉽게 작성할 수 있음

프로그램 작성 전에, 문제를 분석하고 해결 방법을 찾은 후 순서도로 그려보는 것이 좋은 프로그램을 작성하는 데에 도움을 줄 수 있음.

**메모 포함[이13]:** + 연결자는 순서도가 길어질 경우 끊어서 쓸 수 있게 한 도형이다.  
+ 판단은 조건문과 대응되는 듯.  
+ draw.io같은 순서도 그리기 프로젝트도 있다.

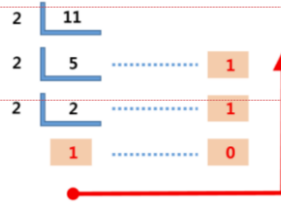
도형	이름	설명	도형	이름	설명
	종단	순서도의 시작 또는 끝		연결자	한 페이지 내에서 연결자
	프로세스	처리문		다른 페이지 연결자	다른 페이지로의 연결자
	판단	주어진 조건에 따른 예 또는 아니오 판단		주석	설명이 필요한 경우 설명
	입력/출력	데이터 입력 또는 출력		흐름선	다음 순서를 알려줌
	미리 정의된 프로세스	이미 만들어진 순서도			

## 7. 진법의 변환

### 1) 10 진수 -> 2 진수

나름의 계산방법을 사용함. (나누기)

8 진수, 16 진수로 바꿀 때처럼 거듭제곱으로 표현해도 됨.



**메모 포함[이14]:** 2, 8, 10, 16진수가 있는데 이들끼리 변환하는 경우의 수는 12가지임. 여기에 전부 정리했음.

**메모 포함[이15]:** 원리가 무엇일까?

### 2) 10 진수 -> 8 진수, 16 진수

해당 10 진수 값을 8 또는 16 의 거듭제곱수로 표현함.

가장 큰 거듭제곱수로 나누고, 나머지를 그 다음으로 큰 거듭제곱수로 나눔.

각각의 몫이 8, 16 진수의 각 자리에 들어가는 수임.

$$\begin{aligned} 1011 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\ &= 8 + 0 + 2 + 1 \\ &= 11 \end{aligned}$$

### 3) 2 진수, 8 진수, 16 진수 -> 10 진수

각 자리 숫자와 자리 값들을 곱해서 더함.

자리 값은 자릿수를  $n$  이라고 했을 때, 2진수는  $2^{(n-1)}$ 이고 8진수는  $8^{(n-1)}$ 이고 16진수는  $16^{(n-1)}$ 임.

### 4) 8 진수 <-> 2 진수

2 진수 세 자리와 8 진수 한 자리가 대응됨.

2 진수는 세 자리씩 끊어서 8 진수로 바꾸고, 8 진수는 한 자리씩 끊어서 2 진수로 바꿈.

(ex. 8 진수 075 는 2 진수로 111101 임.)

**메모 포함[이16]:** 직관적으로 이해가 안 될 수도 있는데, 8진수를 10진수로 바꾼 후에 2진수로 바꿔봐도 이 방법과 결과가 같다.

### 5) 16 진수 <-> 2 진수

2 진수 네 자리와 16 진수 한 자리가 대응됨.

2 진수는 네 자리씩 끊어서 16 진수로 바꾸고, 16 진수는 한 자리씩 끊어서 2 진수로 바꿈.

### 6) 16 진수 <-> 8 진수

그냥 10 진수로 먼저 바꾼 후에 전환하자.

## 2. 리눅스 기초

# 1. vi 편집기

### 1. vi (vim) 편집기

UNIX의 대표적 문서 편집기 (visual editor)

명령 모드와 입력 모드가 있음

마지막 행 모드도 있지만 마지막 행 모드는 명령 모드에 포함시켜 이야기함.

"vi 파일명.확장자"  
-> vi 편집기 실행.

**메모 포함[이17]:** vim은 vi improved로, vi에 여러가지 기능이 추가된 것이다.  
이 수업에서는 vim과 vi모두 통틀어 vi라고 부르기로 했다.

**메모 포함[이18]:** :, /, ? 등을 사용하는 특별 명령 처리기.  
ex 편집기의 명령.

### 2. 명령 모드와 입력 모드의 전환

- 명령모드->입력모드 : i, a, l, A, o, O 중 하나를 입력

i: 커서 위치에 문자 입력 시작

a: 커서 위치 다음 칸에 문자 입력 시작

l (대문자 아이): 커서가 위치한 행의 맨 앞에 문자 입력 시작 (행의 첫 칼럼으로 이동하여 입력)

A: 커서가 위치한 행의 맨 끝에 문자 입력 시작 (행의 마지막 칼럼으로 이동하여 입력)

o: 커서가 위치한 행 다음 행에 문자 입력 시작

O: 커서가 위치한 행의 이전 행에 문자 입력 시작

- 입력모드->명령모드: Esc 키 입력

**메모 포함[이19]:** iao로 암기.

+ 처음 vi를 실행시켰을 때는 명령 모드로 시작한다.

+ 명령 모드가 기본이고 입력 모드라는 기능이 있는 것이라고 생각하면 편함.

즉, 입력을 위해 특정한 전환 키를 사용하여 해당하는 기능을 수행하게 하는 것

### 3. 명령

#### - 저장 및 종료 명령

w : 현재 파일을 문서로 저장  
:w <filename> : 해당 파일명으로 새로운 문서 저장  
:wq, :wq!, ZZ : 문서 저장 후 vi 종료  
:q! : 문서를 저장하지 않고 vi 종료  
:q : 작업한 것이 없을 때 단순 종료  
:e! : 수정된 사항을 취소. (마지막 저장 명령 직후의 상태로 복귀. vi가 닫히지는 않음.)

#### - 커서 이동 명령

h : 커서를 왼쪽으로 이동  
j : 커서를 아래로 이동  
k : 커서를 위로 이동  
l (소문자 엘) : 커서를 오른쪽으로 이동  
w : 커서를 오른쪽 단어 첫번째 글자로 이동  
e : 커서를 오른쪽 마지막 글자로 이동 (현재 단어의 마지막부터 시작함)  
b : 커서를 왼쪽 단어 맨 앞 글자로 이동 (back)  
^, 0, <home 키> : 커서를 현재 행의 맨 왼쪽으로 이동  
\$, <end 키> : 커서를 현재 행의 맨 오른쪽으로 이동  
+, <enter 키> : 커서를 다음 행의 맨 앞으로 이동  
- : 커서를 이전 행의 맨 앞으로 이동  
H : 커서를 화면의 맨 위 행 맨 앞으로 이동 (맨 위는 아닐 수 있음. 커서가 갈 수 있는 곳까지 감) (High)  
M : 커서를 화면의 중간 행 맨 앞으로 이동. 행의 개수가 짝수라면 두 행 중 위의 행으로 이동. (Middle)  
L : 커서를 화면의 맨 아래 행 맨 앞으로 이동 (맨 아래는 아닐 수 있음. 커서가 갈 수 있는 곳까지 감) (Low)  
G : 커서를 문서의 마지막 행 맨 앞 글자로 이동 (Go)  
<인수>G : 커서를 인수에 해당하는 행으로 이동

**메모 포함[이20]:** 줄임말들.

w : write  
q : quit  
y : yank  
p : put  
d : delete  
c : change

**메모 포함[이21]:** G: 문서의 끝 의미

0: 행의 맨 처음 의미  
\$, 대문자 하나: 행의 맨 마지막 의미  
w: 단어 의미. 현재 단어에서는 커서 위치부터 문자 끝까지를 의미. -> 앞에 숫자 명시할 수 있음.  
동일한 문자 2개 반복: 하나의 행 전체를 의미 -> 앞에 숫자 명시할 수 있음.  
문자 하나: 해당 문자가 가지는 의미 -> 앞에 숫자 명시할 수 있음.  
<block> 문자: 해당 블록에 문자의 의미 적용  
!: 중단하는 것을 의미

**메모 포함[이22]:** 왼손잡이용이라고 암기. 왼쪽이 내려가는 것임.

**메모 포함[이23]:** -를 사용할 때는 +와 다르게 shift 안 눌러도 되는 거 유의하자!



## - 화면 이동 명령

표 3-7 화면 이동 명령 키

기존 명령 키	기능	추가 명령 키
<sup>^</sup> u ( <b>Ctrl</b> +u)	반 화면 위로 이동한다.	
<sup>^</sup> d ( <b>Ctrl</b> +d)	반 화면 아래로 이동한다.	
<sup>^</sup> b ( <b>Ctrl</b> +b)	한 화면 위로 이동한다.	Page Up
<sup>^</sup> f ( <b>Ctrl</b> +f)	한 화면 아래로 이동한다.	Page Down
<sup>^</sup> y ( <b>Ctrl</b> +y)	화면을 한 행만 위로 이동한다.	
<sup>^</sup> e ( <b>Ctrl</b> +e)	화면을 한 행만 아래로 이동한다.	

(up, down, back, front)

## - 삭제 명령 (x, d)

x, #x : 커서 위치 문자 하나 삭제, #에는 삭제할 글자 수를 작성할 수 있음

X : 커서 위치 이전 칸 문자 삭제

dw, #dw : 커서가 위치한 단어를 삭제. 현재 위치한 단어는 커서 뒤부터만 삭제함. #에는 삭제할 단어 수를 작성할 수 있음

dd, #dd : 커서 위치 행 삭제(잘라두기), #에는 삭제(잘라두기)할 행 수를 작성할 수 있음

d\$, D, <shift>+d : 커서 위치부터 행의 끝까지 삭제

d0 : 현재 행의 처음부터 커서위치 앞 문자까지 삭제

dG : 현재 행 위치부터 문서의 끝까지 삭제

:<block>d : 지정된 <block> 삭제

## - 수정 명령 (rectify, change)

r : 커서가 위치한 문자를 삭제하고 하나의 문자를 입력 받은 후 다시 명령모드로 전환.

R : 수정모드인 입력모드로 전환

cw, #cw : 커서 위치부터 단어 끝까지 삭제한 후 입력모드로 전환, #에 수정할 단어의 개수를 작성할 수 있음.

cc : 커서가 위치한 행 전체 삭제한 후 입력모드로 전환

c0 : 현재 행의 처음부터 커서 위치 앞 문자까지 제거 후 입력모드로 전환

c\$, C : 커서 위치부터 행 끝까지 제거 후 입력모드로 전환

s, #s : 커서 위치의 문자를 하나 지우고 입력모드로 전환. #에는 삭제할 글자의 수를 작성할 수 있음.

+ 수정 명령에서 r 빼고는 모두 사용 후 입력모드로 바뀜

+ 한 문장이 길어져 아래 행까지 내려갈 경우 아래 행이 아니라 한 문장 취급됨

**메모 포함[이24]:** 해당 내용을 삭제하고 입력모드로 바꾸는 것이 수정하는 것이다.

**메모 포함[이25]:** 삭제 명령과 형식이 유사함. d대신 c를 쓰면 입력모드가 되는 것!

**메모 포함[이26]:** 여기에서 수정모드를 사용할 수 있게 하는 것은 R 명령어 하나뿐이다.

수정모드 : 글자를 타이핑할 경우 기존의 문자가 밀리는 것이 아니라 삭제되는 모드이다.

삽입모드 : 글자를 타이핑할 경우 기존의 문자가 밀리는 모드이다.

워드 프로그램에서는 보통 insert키로 이 모드를 전환하게 되어있다.

## - 복사, 붙여넣기 명령 (y, p)

yy, #yy : 커서 위치 행 복사, #에 복사할 행의 수를 작성할 수 있음.

y0 : 현재 행의 처음부터 커서 위치 앞 칸까지 복사

y\$ : 커서 위치부터 행의 끝까지 복사

yw : 커서 위치부터 단어 끝까지 복사

yG : 현재 행 위치부터 문서의 끝까지 복사

:<block> y : <block> 복사

p (소문자) : 복사한 내용을 커서 위치 다음 행에 붙여넣기

P (대문자) : 복사한 내용을 커서 위치 위 행에 붙여넣기

:<block> co <인수> : <block>을 복사하여 <인수> 행 다음에 붙여넣기 (copy)

## - 문자열 검색, 대체 명령

/<인수> : 커서 위치부터 아래 방향으로 인수(문자열) 검색

?<인수> : 커서 위치 위 방향으로 인수(문자열) 검색

n : 원래 찾던 방향으로 이전 인수 계속 검색 (next)

N : 원래 찾던 반대 방향으로 이전 인수 계속 검색 (Next)

:s/인수1/인수2 : 커서가 위치한 행의 첫번째 인수1을 인수2로 변환

:%s/인수1/인수2/g : 문서 내의 모든 인수1을 인수2로 변환 (global)

:<block>s/인수1/인수2/ : <block>의 모든 행에서 제일 앞 인수1 하나를 인수2로 변환

:<block>s/인수1/인수2/g : <block>의 모든 행에서 인수1을 인수2로 변환 (global)

:<block>s/인수1/인수2/gc : <block>의 모든 행에서 인수1을 인수2로 변환하는데 매번 수정 여부를 물어봄.

(global check)

:g/인수1/s//인수2/gc : 문서 전체에서 인수1을 탐색하고, 선택에 따라 인수2로 변환할 수 있는 기능 실행

+ 인수로 입력한 문자열이 한 단어로 존재하지 않아도 그 문자의 배열이라면 검색이 됨.

+ 검색 시 대문자 소문자의 구분은 기본적인 것.

## - 다중 파일 편집 명령

r <file> : 지정한 파일을 읽어서 현재 커서 위치에 삽입

!e <file> : 지정한 파일을 편집 시작 (현재 파일을 저장하지 않고 이 명령을 사용하면 오류 메시지가 출력됨.)

n : vi 시작 시 여러 파일을 지정했을 경우, 다음으로 해당되는 파일을 편집 시작

**메모 포함[이27]:** 1. 복사 명령어에 따라 붙여 넣어지는 위치가 다르다.

다음 행에 삽입: yy, yG, <block> y

커서 오른쪽 칸에 삽입: y0, y\$, yw

2. 삭제 명령을 수행하고 p를 사용하면 삭제된 내용이 붙여 넣어진다. p는 붙여넣기 보다는 잘라넣기가 더 정확한 설명이다.

**메모 포함[이28]:** 아래 방향이 기본이기 때문에 위 방향은 shift를 눌러야 함.

**메모 포함[이29]:** 즉, 한 행에서 맨 앞에 있는 하나만 바뀜.

**메모 포함[이30]:** hello를 검색하면 "hello"만이 아니라 "qerqwrhello"같은 곳에서도 찾을 수 있다는 의미.

## - 범위 지정 명령

:인수1, 인수2 <명령> : 인수1, 인수2에는 행 숫자를 넣음. 인수 1번째 행~인수 2번째 행 블록이 만들어짐.

**특수한 의미를 가지는 용어들도 사용할 수 있음**

\$ : 파일에서 마지막 행

. : 현재 행 (커서가 있는 행)

% : 전체 행

+인수 : 현재 행으로부터 **아래로** 인수만큼의 행을 추가로 포함 (현재 행 또한 포함됨)

-인수 : 현재 행으로부터 위로 인수만큼의 행을 추가로 포함 (현재 행 또한 포함됨)

(ex.

**표 3-13** 범위 지정 명령 키

명령 키	기능
1, \$ 또는 %	1행부터 마지막 행까지 지정한다.
1..	1행부터 커서가 있는 행까지 지정한다.
..\$	커서가 있는 행부터 마지막 행까지 지정한다.
,-3	현재 행과 이전 세 행까지(총 네 행) 지정한다.
10,20	10행부터 20행까지 지정한다.

**메모 포함[이31]:** 항상 오른쪽, 아래가 기준이다.  
글 읽는 방향.

- + 인수1이 인수2보다 클 경우, 시스템에서 사용자 의도대로 입력한 것인지 확인 절차를 거친 후 수행하게 됨
- + 결과적으로 선택된 행이 하나일 경우, 그 행에서만 명령을 수행함
- + 블록 지정 시에는 인수에 연산을 쓸 수도 있음.

## - 셸 명령어 사용 명령

!  
:! <셸 명령> : vi 작업을 잠시 중단하고 셸 명령을 실행함. (<enter>키를 누르면 다시 vi로 돌아옴.)  
:sh : vi를 잠시 중단하고 셸로 감. ('exit'을 작성하면 vi로 돌아옴.) (shell)

## - 그 외 명령

xp : 커서가 위치한 문자를 다음 문자와 교환

~ : 대문자를 소문자로, 소문자를 대문자로 변환

u : 이전 명령 취소 (undo)

U : 커서가 위치한 행에서 한 모든 명령 취소. (Undo)

. : 마지막 수행 명령 반복

<ctrl> G, <ctrl> g : 문서 이름과 현재 커서의 위치 정보 표시

<ctrl> l (소문자 엘) : 현재 화면을 다시 출력 (refresh해줌.)

J : 현재 행과 아래 행을 연결해서 하나의 행으로 만들 (join)

표 3-14 마지막 행 모드에서의 복사하기, 잘라내기, 붙이기 명령 키

명령 키	기능
:#y	#로 지정한 행을 복사한다. 예를 들면 3y는 세 행을 복사한다.
:<범위>y	범위로 지정한 행을 복사한다. 예를 들면 2,4y는 2~4행을 복사한다.
:#d	#로 지정한 행을 잘라낸다(삭제). 예를 들면 3d는 세 행을 잘라낸다.
:<범위>d	범위로 지정한 행을 잘라낸다(삭제). 예를 들면 1,4d는 1~4행을 잘라낸다.
:pu	현재 행 다음에 버퍼의 내용을 붙인다.
:#pu	#로 지정한 행 다음에 버퍼의 내용을 붙인다. 예를 들면 4pu와 같이 지정한다.

메모 포함[이32]: 이거 안되는데..?

## 4. 환경설정

환경 변수 설정으로 사용자가 원하는 환경을 조성할 수 있게 함

vi 환경설정에는 세 가지 방법이 있음.

- 1) 홈 디렉터리에 `exrc` 파일을 만들어서 명령을 작성 (항상 해당 설정으로 vi를 사용할 수 있음)
- 2) `EXINIT` 환경 설정 변수를 이용 (항상 해당 설정으로 vi를 사용할 수 있음)
- 3) vi의 마지막 행 모드에서 명령하기. :를 작성하고 명령을 작성. (vi를 종료하면 default값으로 돌아감)

명령들.

set cindent : c언어 프로그래밍에 필요한 자동 들여쓰기 기능

set tabstop=<숫자> : 탭 키의 간격 조절 기능

set shiftwidth=<숫자> : 자동 들여쓰기의 간격 조절 기능

set nu : 행에 번호를 표시함

set nonu : 행 번호 감춤

set ruler : 현재 커서의 위치 출력 기능

set title : 타이틀 바에 현재 작업 중인 문서 이름 출력 기능

syntax on : 문법 구분 기능 (색깔)

set hlsearch : 패턴 검색 시 해당 부분의 하이라이트 표시 기능

set list : 가시화되지 않은 특수문자들을 가시화함.

set nolist : 가시화된 특수문자들을 비가시화함.

set showmode : 현재 모드를 표시함

set noshowmode : 현재 모드를 숨김

set : set으로 설정한 vi 환경 설정 값들을 출력함

set all : 모든 vi 환경 변수와 현재 값을 출력함.

**메모 포함[이33]:** 환경 변수란 컴퓨터에서 프로그램이 동작하는 방식에 영향을 미치는 동적인 값들의 모임.

**메모 포함[이34]:** vi를 실행할 때마다 이 파일을 확인하게 됨.

**메모 포함[이35]:** `EXINIT='<명령>'`  
`export EXINIT`

을 적으면 됨.

## 5. 버퍼

### 1) 네임드 버퍼

이름을 붙여서 사용할 수 있는 버퍼.

이름을 붙이면 각각 독립적으로 내용을 저장하고 사용할 수 있음.

네임드 버퍼에 내용을 저장하거나 붙여 넣으려면 명령어 앞에 ' '버퍼 이름'을 적어 주기만 하면 됨.

(띄어쓰기 없이)

"<버퍼 이름><명령어>"

**메모 포함[이36]:** 하나의 파일에서 복사한 것을 저장해 두면 다른 파일에서도 그것을 사용할 수 있음.

### 2) 언네임드 버퍼

이름을 붙이지 않은 버퍼.

yy 키로 복사하거나 dd 키로 잘라낸 경우 그 내용이 저장되는 곳.

하나의 내용만 저장함.

## 2. 리눅스 파일시스템

### 1. 파일시스템이란?

파일을 저장하고 구조화하는 체계.

### 2. 디렉토리 (=폴더)

컴퓨팅에서 파일을 분류하기 위해 사용하는 공간

리눅스는 효율적인 관리를 위해 디렉토리들로 이루어진 계층 구조를 가짐. (역 Tree 구조.)

#### - 루트 디렉토리

파일시스템의 최상위 디렉토리. (모든 파일 및 디렉토리 포함.)  
루트 디렉토리는 부모 디렉토리가 없음.

/

#### - 작업 디렉토리 (현재 디렉토리)

파일시스템 상에서 현재 위치를 의미.

.(점)

#### - 부모 디렉토리 (상위 디렉토리)

특정 디렉토리의 위에 있는 디렉토리.

.. (점점)

#### - 서브 디렉토리 (하위 디렉토리)

특정 디렉토리의 아래에 있는 디렉토리

#### - 홈 디렉토리

사용자에게 할당된 개인 디렉토리  
리눅스 최초 실행 시 작업 디렉토리는 홈 디렉토리임.

~

메모 포함[이37]: 윈도우의 폴더와 유사하다.

### 3. 경로 (path)

파일시스템의 특정 위치에서부터 지정된 위치까지 통과하게 되는 디렉토리 및 파일을 나열한 것.

특정 파일의 위치 표시.

각 파일을 구분하는 구분자로 /를 사용.

맨 앞에 /가 온다면 그건 구분자가 아니라 루트 디렉토리임.

#### - 절대경로

루트로부터 시작되는 경로.

/가 루트를 의미하므로, 반드시 /로부터 시작함.

특정 위치를 가리키는 절대경로는 항상 동일.

특정 위치까지 이동하면서 거치게 되는 모든 디렉토리명을 명시해야 함.

#### - 상대경로.

현재 디렉토리를 기준으로 시작되는 경로.

현재 위치가 어디인지에 따라 상대경로는 바뀔 수 있음.

현재 디렉토리를 기준으로 하위 디렉토리로 내려가려면 해당 디렉토리명을 명시.

현재 디렉토리를 기준으로 상위 디렉토리로 올라가려면 .. (점점)을 디렉토리 대신 추가.

(디렉토리명을 적어도 됨.)

## 4. 명령어들

### <파일시스템 관련 명령어들>

#### - pwd (present working directoy)

작업 디렉토리의 절대 경로 출력 명령어

```
pwd
```

#### - cd (change directory)

특정 위치로 작업 디렉토리를 이동하는 명령어

<path>에는 절대경로나 상대경로를 쓸 수 있음.

경로에 아무것도 안 적으면 사용자의 홈 디렉토리로 감

작업 디렉토리 내에 위치해 있는 디렉토리라면 경로 칸에 디렉토리명만 적어도 해당 디렉토리로 이동함.

```
cd <option> <path>
```

**메모 포함[이38]:** cd 명령어로 작업 디렉토리를 이동했으면,  
pwd 명령어로 확인하는 습관을 가지자.



## - ls (소문자 엘) (list)

작업 디렉토리의 파일목록 출력 명령어

파일을 적지 않으면 현재 디렉토리의 파일을 보여줌.

옵션들.

-a : 모든 파일목록 출력. (. 으로 시작하는 파일과 디렉토리도 보여줌.)

-A : . 와 .. 를 제외한 모든 파일목록 출력. (.과 ..로 시작하는 파일이 아니라 .과 .. 자체를 제외함)

-d : 디렉토리 자체를 지정. 디렉토리 자체의 정보를 출력할 때 사용.

-i (inode) : inode 번호 출력.

-R : 하위 디렉터리에 있는 파일까지 보기

-F : 파일과 함께 파일의 종류 출력. (/ : 디렉토리, @ : 심볼릭 링크, \* : 실행파일)

-L : 심볼릭 링크 파일이 있는 경우 원본 파일의 정보 출력.

-l (소문자 엘) : 파일의 상세정보 출력. 파일 종류, 권한, 파일 소유자 등 확인 가능

ls <option> <file>

**메모 포함[이39]:** dir, vdir등을 사용할 수도 있다. 비슷한 기능을 한다.

**메모 포함[이40]:** 주로 설정 파일로 사용되는 파일임.

**메모 포함[이41]:** 어떤 디렉토리를 지정해서 ls명령어를 사용할 때, 해당 디렉토리 안에 있는 파일에 대한 상세 정보를 출력함. 그 디렉토리 자체의 정보를 보려면 d옵션을 넣어줘야 함.

**메모 포함[이42]:** 모든 파일은 inode 번호를 가지고 있음. 즉, 리눅스의 모든 것은 inode 번호를 가지고 있음.

## + ls 명령어 -l 옵션을 사용하여 출력한 상세정보의 의미

표 2-2 파일의 상세 정보

필드 번호	필드 값	의미
1	d	파일 종류
2	rwxr-xr-x	파일 접근 권한 파일 소유자 그룹 기타 사용자가 파일을 읽고 수정하고 실행할 수 있는 권한이 어떻게 부여되어 있는지를 보여준다.
3	2	하드 링크의 개수
4	user1	파일 소유자
5	user1	파일이 속한 그룹
6	4096	파일 크기(바이트 단위)
7	11월 8 23:24	파일이 마지막으로 수정된 시간
8	공개	파일명

(ex. -rwer--r-- 2 jhun jhun 4003 Mar 6 18:28 test1.c)

표 2-3 파일의 종류

문자	파일 유형
-	일반(정규) 파일
d	디렉터리 파일
l	심볼릭 링크 파일
b	블록 단위로 읽고 쓰는 블록 장치 파일
c	섹터 단위로 읽고 쓰는 문자 장치 파일
p	파이프 파일(프로세스 간 통신에 사용되는 특수 파일)
s	소켓(네트워크 통신에 사용되는 특수 파일)

## + ls 명령어 -l 옵션 사용 시의 파일 접근 권한

사용자 유형별로 읽기(read), 쓰기(write), 실행(execute) 가능 여부를 나타냄.

사용자는 파일 소유자, 그룹 사용자, 기타 이용자로 구분.

다중 사용자 OS에서 파일 보호를 위해 사용.

ls -l 명령어의 두 번째 필드에서 확인할 수 있음  
(2~10번째 글자들.)

문자 위치	의미
첫 번째	파일의 종류를 나타냄; d : 디렉터리, - : 일반 파일, l : 심볼릭 링크, p : 이통있는 파이프, 등
2~4번째	파일 소유자의 권한; 3개의 문자는 각각 읽기, 쓰기, 실행 권한을 나타내고, 각 자리에 r, w, x가 있으면 그 권한이 있음을, -가 있으면 권한이 없음을 뜻함
5~7번째	그룹 사용자의 권한; 3개의 문자는 각각 읽기, 쓰기, 실행 권한을 나타내고, 각 자리에 r, w, x가 있으면 그 권한이 있음을, -가 있으면 권한이 없음을 뜻함
8~10번째	기타 사용자의 권한; 3개의 문자는 각각 읽기, 쓰기, 실행 권한을 나타내고, 각 자리에 r, w, x가 있으면 그 권한이 있음을, -가 있으면 권한이 없음을 뜻함

## <파일, 디렉토리 관련 명령어들>

### - mkdir (make directory)

디렉토리 생성 명령어

```
mkdir <option> <name>
```

디렉토리명은 절대경로 또는 상대경로로 작성함.

경로가 아닌 디렉토리명만을 적으면 작업 디렉토리에 디렉토리를 생성함.

디렉토리를 한 번에 여러 개 만들려면 디렉토리 이름을 여러 개 적으면 됨.

공백 문자로 구분하면서 작성.

(ex. mkdir -p name1 name2 name3)

옵션들.

-m <mode> : 접근 권한을 설정하며 디렉토리 생성.

-p : 하위 디렉토리를 생성할 때, 중간 단계의 디렉토리가 없으면 자동으로 생성하면서 작업 수행.

(중간 디렉토리가 없는데 이 옵션을 사용하지 않는다면 디렉토리가 생성되지 않음.)

메모 포함[이43]: 숫자 모드로만 되는지 확인

### - rmdir (remove directory)

비어있는 디렉토리 삭제 명령어

```
rmdir <option> <directory>
```

디렉토리를 한 번에 여러 개 삭제하려면 디렉토리 이름을 여러 개 적으면 됨.

옵션들.

-p : 해당 디렉토리를 삭제하며 그 디렉토리의 상위 디렉토리가 빈 디렉토리인 경우 상위 디렉토리도 삭제함.

메모 포함[이44]: 디렉토리가 비어있지 않은 경우에는 rm -r 을 사용해야 한다.

### - touch

빈 파일 생성, 파일 마지막 접근시간 또는 수정시간 수정 명령어

```
touch <option> <file>
```

작업 디렉토리 내 파일의 수정 시간 기록 변경.

옵션을 작성하지 않으면 수정 시간을 현재 시간으로 변경됨.

해당 파일이 작업 디렉토리에 없을 경우 같은 이름의 크기가 0바이트인 파일 생성.

옵션들

-a : 접근 시간만 변경

-m : 수정 시간만 변경

-t <시간> : 명시한 시간으로 시간 수정.

시간 표시

• 형식 [[CC]YY]MMDDhhmm[.ss]

CC : 연도의 첫 두 자리

MM : 달(01~12 범위에서 지정)

hh : 시간(00~23 범위에서 지정)

ss : 초(00~59 범위에서 지정)

YY : 연도의 마지막 두 자리

DD : 날짜(01~31 범위에서 지정)

mm : 분(00~59 범위에서 지정)

## - cp (copy)

```
cp <option> <source_file> <dest_file>
```

파일 복사 명령어 (이동 아님!)

첫 번째 인자에는 원본 파일을, 두 번째 인자에는 목적지 위치나 파일명 작성.

**메모 포함[이45]:** source. 출처.  
destination. 목적지.

### 1) 첫번째 인자가 파일인 경우

두번째 인자가 파일인 경우 첫번째 인자의 파일의 이름을 두번째 인자로 바꾸어서 작업 디렉토리에 복사함.

두번째 인자가 디렉토리인 경우 첫번째 인자의 파일이 두번째 디렉토리 안에 복사함.

두번째 인자가 "디렉토리/파일명"인 경우 첫번째 인자의 파일의 이름이 두번째 인자의 파일명으로 바뀌어서 복사함.

### 2) 첫번째 인자가 디렉토리인 경우

두번째 인자의 디렉토리 안에 첫번째 인자의 디렉토리가 복사됨.

두 번째 인자로 작성한 디렉토리가 없을 경우 해당 이름의 디렉토리를 생성.

디렉토리가 복사되면 원본 디렉토리(첫 번째 인자에 있던 거) 아래에 있던 모든 내용도 함께 복사.

인자가 3개 이상인 경우, 마지막 인자가 destination 임. 고로 마지막 인자는 반드시 디렉토리여야 함.

인자가 3개 이상인 경우에도 디렉토리와 파일을 혼용할 수 있음. -r 만 잘 써주면 됨.

### option 종류

-r : 디렉토리를 복사할 때 사용.

-i : dest\_file 이 이미 존재하는 파일이면 덮어쓸 것인지 물어봄.

**메모 포함[이46]:** i 옵션은 대체로 정말 실행할 것인지 확인 차 물어보는 기능으로 쓰인다.

## - rm (remove)

```
rm <option> <file>
```

일반 파일, 디렉토리 삭제 명령어

여러 개의 인자를 작성할 경우 그 인자 모두를 삭제함.

**메모 포함[이47]:** 리눅스에서는 휴지통의 개념이 없다.  
삭제 시 그냥 끝.

### option 종류

-r : 디렉토리 삭제. 그 아래 파일들도 삭제됨.

-i : 파일을 정말 삭제할 것인지 물어봄.

**메모 포함[이48]:** -r을 작성해도 일반 파일을 삭제할 수 있음.  
즉, 일반 파일과 디렉토리를 한꺼번에 삭제할 때에도 그냥 -r만 붙여주면 됨.

## - mv (move)

파일, 디렉토리 이동. 이름 변경 명령어

```
mv <option> <source_file> <dest_file>
```

첫 번째 인자에는 원본 파일을, 두 번째 인자에는 **목적지 위치**나 파일명 작성.

**메모 포함[이49]:** 작업 디렉토리에 목적지 디렉토리가 있다면  
그냥 이름만 적어도 된다.  
없다면 경로 명시해 줘야 함.

### 1) 파일을 파일로 이동하기 (파일명 바꾸기)

두 번째 인자에 작성한 파일명으로 파일명이 바뀜.

작업 디렉토리 내에서 작업이 수행됨.

두 번째 인자에 작성한 파일명을 가진 파일이 이미 존재할 경우 덮어씀.

### 2) 파일 하나를 다른 디렉토리로 이동하기

두 번째 인자로 디렉토리를 쓸 경우 첫 번째 인자의 파일이 그 디렉토리로 이동함.

두 번째 인자를 디렉토리/파일명 꼴로 작성할 경우 파일이 해당 디렉토리로 이동하며 해당 파일명으로 바뀜.

해당 디렉토리의 쓰기 권한이 없을 경우 이동 불가.

### 3) 파일 여러 개를 다른 디렉토리로 이동하기

3개 이상의 인자를 사용할 수 있음.

앞 인자들에 작성된 파일들이 맨 마지막 인자에 작성된 디렉토리로 이동함.

마지막 인자에는 반드시 디렉토리가 들어가야 함.

### 4) 디렉토리를 디렉토리로 이동하기

인자에 디렉토리를 넣으면 디렉토리를 이동시킴.

두 번째 인자가 원래 있던 디렉토리인 경우에는 그 디렉토리 밑으로 첫 번째 인자에 있는 디렉토리를 이동.

두 번째 인자가 원래 있던 디렉토리가 아닌 경우에는 그 디렉토리명으로 첫 번째 인자에 있는 디렉토리의 이름을 바꿈.

옵션들

-i : dest\_file 이 이미 존재하면 덮어쓸 것인지 물어봄.

## - file

지정한 파일의 종류를 알려주는 명령어.

```
file <file>
```

## - sort

텍스트 파일을 행 단위로 오름차순 정렬해서 출력해주는 명령어

```
sort <option> <file>
```

-r : 내림차순으로 정렬

## <파일 출력 명령어들>

### - cat (catenate)

파일 내용 출력 명령어.

```
cat <option> <file>
```

옵션들.

-n : 행 번호를 붙여서 출력. 빈 행도 번호를 매김.

-b : 행 번호를 붙여서 출력. 빈 행은 번호를 매기지 않음

```
cat <option> <file1> > <file2>
```

> 기호 : 출력 다시 지정.

file1의 내용을 file2에서 출력함. 즉, file2에 file1의 내용을 작성함.

file1 자리에는 여러 개의 인자(파일)가 들어가면 여러 개의 파일을 file2에서 출력함.

**메모 포함[이50]:** file2에 들어간 파일이 없을 경우 file1의 내용을 복사해서 출력한다.

file2에 들어간 파일이 있을 경우에는..?

file1을 작성하지 않고 > <file2>만 작성한다면 해당 이름을 가진 파일이 생성됨.

이때 해당 파일에 들어갈 내용을 작성하게 됨. <control>+d 키로 빠져나올 수 있음.

**메모 포함[이51]:** 원래 있는 파일이면?

### - more

```
more <option> <file>
```

파일 내용을 한 화면 단위로 출력하는 명령어.

출력이 시작된 부분부터 위아래로 넘기며 읽을 수 있음. 특정 행에서 출력이 시작되었으면, 그 위로는 이동할 수 없음.

space bar -> 다음으로

b -> 전으로.

출력되지 않은 내용이 있으면 화면 하단에 "--More--(0%)" 형식으로 표시됨.

옵션들.

pg : 명령어 뒤에 작성하여 페이지를 넘기는 형태로 파일 내용을 확인할 수도 있음. n 으로 넘김.

+<행 번호> : 출력을 시작할 행 번호를 지정함.

## - less (리눅스+)

파일 내용을 한 화면 단위로 출력하는 명령어.  
처음부터 출력되고, 위아래로 넘기며 읽을 수 있음.

```
less <file>
```

**메모 포함[이52]:** cat -> 단순 출력. 띄우기만 함.  
more -> 아래로만 움직이며 확인 가능.  
less -> 위아래 모두 움직이며 확인 가능.

표 2-4 less 명령에서 사용하는 키와 동작

키	동작
j	한 줄씩 다음 행으로 스크롤한다.
k	한 줄씩 이전 행으로 스크롤한다.
[Space Bar] [Ctrl]+f	다음 화면으로 이동한다.
[Ctrl]+b	이전 화면으로 이동한다.

## - head

파일 내용의 앞부분부터 행을 출력하는 명령어  
default 값은 10개의 행 출력임.

```
head <option> <file>
```

### 옵션들.

- c <number> : 파일 앞부분부터 해당 숫자만큼의 byte 까지 출력.
- n <number> : 파일 앞부분부터 해당 숫자만큼의 행까지 출력. (number)

## - tail

파일 내용의 뒷부분부터 행을 출력하는 명령어.  
default 값은 10개의 행 출력임.

```
tail <option> <file>
```

### 옵션들.

- +<행 번호> : 지정한 행부터 파일 내용 끝까지 출력.
- <숫자> : 화면에 출력할 행의 수를 지정함.
- f : 파일 출력을 종료하지 않고 반복적으로 출력. ctrl c 키로 출력을 종료함.
- c <number> : 파일 뒷부분부터 해당 숫자만큼의 byte 까지 출력.
- n <number> : 파일 뒷부분부터 해당 숫자만큼의 행까지 출력. (number)

**메모 포함[이53]:** 그리고 이거 옵션 섞어쓰는 건 어떻게 하는지 확인하기.

## <파일 소유와 권한 관련 명령어들>

### - chown (change owner)

```
chown <option> <owner> <file>>
```

파일 소유자를 변경하는 명령어

이 명령어에서 명시한 사용자로 파일 소유자가 변경됨.

소유자 변경 시에 관리자 권한을 사용해야 할 수 있음. (sudo)

(ex. chown jhun test 이면 test 라는 파일의 소유자가 jhun 이 됨.)

### - chgrp (change group)

```
chgrp <option> <group> <file>
```

파일 사용(소유)자 그룹 변경 명령어

### - chmod (change mode)

```
chmod <option> <mode> <file>
```

파일 접근권한 변경 명령어. (작성한 모드로 변경.)

mode 에는 기호 모드와 숫자 모드가 있음

-R : 하위 디렉토리까지 변경

#### 1) 기호 모드

오른쪽 표의 기호를 사용해서 작성.

“사용자+설정+권한”의 형식으로, 필요한 만큼 작성.

사용자끼리, 권한끼리 적는 순서는 상관이 없음.

쉼표로 이어서 적을 때 띄어쓰기 하면 오류남.

(ex. chmod g-r+xw,o+w test)

기호 분류	기호	의미
사용자	u	파일 소유자
	g	그룹 사용자
	o	기타 사용자
	a	모든 사용자
권한	r	읽기
	w	쓰기
	x	실행
설정	+	권한 추가
	-	권한 삭제
	=	권한 배정

**메모 포함[이54]:** 권한 배정은 무엇을 뜻하는지. 다른 명령어에서 기호 모드를 사용할 때에 쓰는 것? 혹은, 그 권한을 가지고 있다는 것을 명시?

#### 2) 숫자 모드

각 사용자 별로 3가지 권한의 유무를 이진수로 나타내고 8진수로 바꾸어 작성. (결국 8진수 3글자로 표현됨.)

권한 순서는 r w x 이고, 권한이 있다면 1, 없다면 0으로 나타냄.

이진수 3자리는 3비트 정보이므로, 8진수 한 자리의 크기와 정확히 맞아떨어짐.

(ex. chmod 754 test)

## - umask

파일, 디렉토리 생성 시에 적용되는 기본 접근권한 **마스크** 설정 명령어

파일, 디렉토리 생성 시에 부여하지 않을 권한을 지정하는 것. 권한 생성 X, 최대 권한에서 막을 권한을 지정.

일반파일 최대 권한: 666, 디렉토리 최대 권한: 777

최대권한에서 마스크만큼 뺀 값이 기본 접근권한이 됨.

```
umask <option> <마스크 값>
```

마스크 값은 가릴 접근권한을 숫자로 표기한 것.

umask 만 치면 현재 설정되어있는 mask 가 뜸.

리눅스에 다시 로그인하면 default 값으로 초기화됨.

옵션들

-S : 마스크 값을 문자로 출력함. 여기서 출력되는 것들은 허용되는 기본 접근권한임.

**메모 포함[이55]:** 마스크로 가린다는 의미.  
마스킹한다는 것.

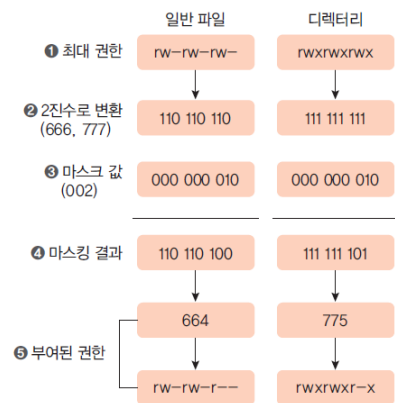


그림 5-8 마스크 값을 적용하는 과정

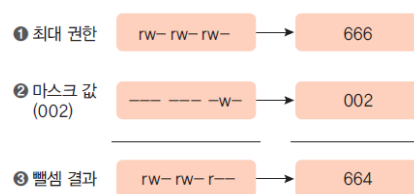


그림 5-9 마스크 값에 뺄셈을 적용하는 과정

+ mask 값은 일반파일과 디렉토리에 한 번에 적용되는데, 둘의 최대 권한이 다르므로 결과에 유의해야 함.



## <파일 비교 명령어들>

### - cmp (compare)

두 파일을 바이트 단위로 비교하는 명령어.  
처음으로 다른 부분을 출력함.

```
cmp <option> <file1> <file2>
```

### - diff (difference)

두 파일을 행 단위로 비교하는 명령어.  
다른 행을 모두 찾아서 출력함.

```
diff <option> <file1> <file2>
```

## <디스크 사용량 관련 명령어들>

### - df (disk free)

전체 파일시스템 디스크 사용량 확인 명령어

그냥 df 만 치면 여러 파일 시스템을 보여줌.

파일 시스템 란에 . 을 치면 현재 파일 시스템 사용 정도를 보여줌

```
df <option> <filesystem>
```

메모 포함[이56]: f는 filesystem으로 암기.

### - du (disk usage)

디렉토리가 사용 중인 디스크 사용량 확인 명령어.

```
du <option> <file>
```

## <검색 관련 명령어들>

### - grep

```
grep <option> <pattern> <file>
```

특정 일반 파일 내에서, 명시한 문자열 패턴을 찾는 명령어.

\* (별표기호) 특수문자를 이용해서 여러 파일에서 찾을 수도 있음.

옵션들.

-i : 대문자, 소문자를 구분하지 않고 검색.

-l : 해당 패턴이 포함되어 있는 파일들의 파일명 출력.

-n : 행 번호와 함께 출력.

-c : pattern 과 일치하는 행의 개수 출력.

-v : 검색 조건을 만족하지 않는 행을 출력

-w : 패턴과 완전히 일치하는 단어만을 출력. (with 이면 with 만 잡고 without 은 안 잡음.)

### - pgrep (process grep)

지정한 패턴과 일치하는 프로세스 정보를 출력해주는 명령어

```
grep <option> <pattern>
```

-x : 패턴과 정확히 일치하는 프로세스의 정보를 출력

-n : 패턴을 포함하고 있는 가장 최근 프로세스의 정보를 출력

-u <사용자명> : 특정 사용자에게 대한 모든 프로세스 정보를 출력

-l (엘) : PID 와 프로세스 이름 출력

-t <term> : 특정 단말기와 관련된 프로세스의 정보를 출력

## - find

파일 및 디렉토리 검색 명령어

```
find <option> <path> <expression>
```

path 에 디렉토리명을 적으면 해당 디렉토리나 그 하위 디렉토리에서 검색함.

한 번에 여러가지 expression 을 쓸 수 있음.

expression 을 작성하지 않으면 -print 가 자동으로 수행됨.

### 옵션들.

-name <file> : 해당 파일명으로 검색. -> 문자열이 아니라 완전한 파일명을 작성해야 찾을 수 있음.

-type <file\_type> : 파일 유형으로 검색.

-user <user\_ID> : 해당 사용자가 소유한 파일 검색.

-perm <mode> : 해당 사용권한 유형을 가진 파일 검색. (permission)

### expression(동작)들

-exec <command> {} \; : 검색된 파일에 해당 명령어 실행.

-ok <command> {} \; : 사용자의 확인을 받아서 검색된 파일에 해당 명령어 실행. (ok 받으면 실행)

-print : 검색된 파일의 절대 경로명 출력. (기본 동작)

-ls : 검색된 결과를 긴 목록 형식으로 출력.

**메모 포함[이57]:** find 명령으로 찾은 파일의 절대 경로가

{ }가 있는 위치에 대입되어 명령이 수행된다.

**메모 포함[이58]:** \; 이걸 역슬래시.

**메모 포함[이59]:** 이런 공백 하나하나 잘 지켜줘야 한다..!

<프로세스 관련 명령어들>

메모 포함[이60]: 뭘 소린지 모르겠음. 나중에 알려줄 듯

- ps (process)

ps <option> <-u UID>

현재 실행 중인 프로세스를 출력하는 명령어.

옵션들

-l (엘) : 프로세스 소유자, 부모 프로세스, 프로세스 그룹 등의 ID 출력.

-a : 모든 사용자의 프로세스 출력

<유닉스>

-e : 시스템에서 실행 중인 모든 프로세스의 정보 출력

-f : 프로세스의 자세한 정보 출력. 프로세스 소유자의 ID 와 부모 프로세스의 ID 출력

-u <UID> : 특정 사용자에게 대한 모든 프로세스 정보 출력

-p <PID> : 특정 프로세스의 정보를 출력

표 6-1 ps -f의 출력 정보

항목	의미	항목	의미
UID	프로세스를 실행한 사용자 ID	STIME	프로세스의 시작 날짜나 시간
PID	프로세스 번호	TTY	프로세스가 실행된 터미널의 종류와 번호
PPID	부모 프로세스 번호	TIME	프로세스 실행 시간
C	CPU 사용량(% 값)	CMD	실행되고 있는 프로그램 이름(명령)

<BSD>

a : 터미널에서 실행한 프로세스 정보 출력

u : 상세 프로세스 정보 출력

x : 시스템에서 실행 중인 모든 프로세스의 정보를 출력

표 6-2 STAT에 사용되는 문자의 의미

문자	의미	비고
R	실행 중(running)	
S	인터럽트가 가능한 대기(sleep) 상태	
T	작업 제어에 의해 정지된(stopped) 상태	
Z	좀비 프로세스(defunct)	
STIME	프로세스의 시작 날짜나 시간	
s	세션 리더 프로세스	BSD 형식
+	포그라운드 프로세스 그룹	
l(소문자 L)	멀티스레드	

<GNU>

--pid <PID> : 특정 PID 의 정보 출력

표 6-3 ps au의 출력 정보

항목	의미	항목	의미
USER	사용자 계정 이름	VSZ	사용 중인 가상 메모리의 크기(KB)
%CPU	퍼센트로 표시한 CPU 사용량	RSS	사용 중인 물리적 메모리의 크기(KB)
%MEM	퍼센트로 표시한 물리적 메모리 사용량	START	프로세스 시작 시간

## - kill

```
kill <option> -<number> <process>
```

프로세스에 signal 을 전송하는 명령어

프로세스는 PID 또는 작업 번호를 작성해서 명시할 수 있음.

작업 번호는 %<작업 번호>를 작성하면 됨.

옵션들

-l (엘) : 사용 가능한 signal 출력. 'kill -l'을 쓰면 됨.

## + signal

프로세스에 무언가 발생했음을 알리는 메시지.

표 6-4 주요 시그널

시그널	번호	기본 처리	의미
SIGHUP	1	종료	터미널과의 연결이 끊어졌을 때 발생한다.
SIGINT	2	종료	인터럽트로 사용자가 <b>Ctrl</b> + <b>c</b> 를 입력하면 발생한다.
SIGQUIT	3	종료, 코어덤프	종료 신호로 사용자가 <b>Ctrl</b> + <b>\</b> 을 입력하면 발생한다.
SIGKILL	9	종료	이 시그널을 받은 프로세스는 무시할 수 없으며 강제로 종료된다.
SIGALRM	14	종료	알람에 의해 발생한다.
SIGTERM	15	종료	kill 명령이 보내는 기본 시그널이다.

다른 시그널들과는 달리, 9번 시그널은 반드시 대상을 죽임.

## - pkill (process kill)

```
pkill <option> <name>
```

PID 가 아니라 프로세스의 이름(CMD)으로 프로세스를 찾아 종료하는 명령어

## - pgrep (process grep)

```
grep <option> <pattern>
```

지정한 패턴과 일치하는 프로세스 정보를 출력해주는 명령어

-x : 패턴과 정확히 일치하는 프로세스의 정보를 출력

-n : 패턴을 포함하고 있는 가장 최근 프로세스의 정보를 출력

-u <사용자명> : 특정 사용자에게 대한 모든 프로세스 정보를 출력

-l (엘) : PID 와 프로세스 이름 출력

-t <term> : 특정 단말기와 관련된 프로세스의 정보를 출력

## - top

top

현재 실행 중인 프로세스에 대한 실시간 정보를 출력하는 명령어

표 6-5 top 명령의 출력 정보

항목	의미	항목	의미
PID	프로세스 ID	SHR	프로세스가 사용하는 공유 메모리의 크기
USER	사용자 계정	%CPU	퍼센트로 표시한 CPU 사용량
PR	우선순위	%MEM	퍼센트로 표시한 메모리 사용량
NI	Nice 값	TIME+	CPU 누적 이용 시간
VIRT	프로세스가 사용하는 가상 메모리의 크기	COMMAND	명령 이름
RES	프로세스가 사용하는 메모리의 크기		

**메모 포함[이61]:** GNOME에 있는 '시스템 감지' 기능을 이용해서 확인할 수도 있다.

표 6-6 top 명령의 내부 명령

내부 명령	기능
<span>Enter</span> <span>Space Bar</span>	화면을 즉시 다시 출력한다.
h, ?	도움말 화면을 출력한다.
k	프로세스를 종료한다. 종료할 프로세스의 PID를 물어본다.
n	출력하는 프로세스의 개수를 바꾼다.
p	CPU 사용량에 따라 정렬하여 출력한다.
q	top 명령을 종료한다.
M	사용하는 메모리의 크기에 따라 정렬하여 출력한다.
u	사용자에 따라 정렬하여 출력한다.

## - jobs

jobs %<작업 번호>

백그라운드 작업 목록을 출력하는 명령어

작업 번호

%<번호> : 해당 번호의 작업 정보 출력

%+, %% : 작업 순서가 +인 작업 정보 출력

%- : 작업 순서가 -인 작업 정보 출력

표 6-7 jobs 명령의 출력 정보

항목	출력 예	의미
작업 번호	[1]	작업 번호로서 백그라운드로 실행할 때마다 순차적으로 증가한다([1] [2] [3]...).
작업 순서	+	작업 순서를 표시한다. • +: 가장 최근에 접근한 작업 • -: 작업보다 바로 전에 접근한 작업 • 공백: 그 외의 작업
상태	실행중	작업 상태를 표시한다. • 실행중: 현재 실행 중이다. • 완료: 작업이 정상적으로 종료되었다. • 종료됨: 작업이 비정상적으로 종료되었다. • 정지됨: 작업이 잠시 중단되었다.
명령	sleep 100 &	백그라운드로 실행 중인 명령이다.

## - nohup

로그아웃한 후에도 백그라운드 작업을 계속 진행하게 하는 명령어

```
nohup <명령, 프로그램> &
```

nohup 을 이용해서 명령을 내리면 그 명령은 로그아웃 후에도 수행이 됨.

프로그램명을 작성한 경우 그 프로그램은 로그아웃 후에도 돌아감.

## - at

예약한 명령을 정해진 시간에 한 번 수행하게 하는 명령어

```
at <option> <시간>
```

at 을 입력하면 시간을 입력하게 되고

시간을 입력하면 설정할 명령을 작성하게 됨. 명령 작성이 끝났으면 ctrl + d 로 빠져나올 수 있음.

at 작업 파일은 /var/spool/at 에 저장됨.

### 시간을 지정하는 형식

- \* at 4pm + 3 days : 지금부터 3일 후 오후 4시에 작업을 수행한다.
- \* at 10am Jul 31 : 7월 31일 오전 10시에 작업을 수행한다.
- \* at 1am tomorrow : 내일 오전 1시에 작업을 수행한다.
- \* at 10:00am today : 오늘 오전 10시에 작업을 수행한다.

**at 명령어 제한** -> /etc/at.allow(허가)와 /etc/at.deny(금지) 파일 이용

두 파일에는 모두 한 행에 하나의 사용자명을 적게 되어있음.

규칙 1. /etc/at.allow 파일이 존재하면 이 파일에 있는 사용자만 at 명령어를 사용할 수 있음. /etc/at.deny 파일은 무시됨.

규칙 2. /etc/at.allow 파일이 존재하지 않으면 /etc/at.deny 파일에 있는 사용자를 제외한 모든 사용자는 at 명령어를 사용할 수 있음. (/etc/at.deny 파일에 아무것도 적혀 있지 않다면 모든 사용자가 사용 가능.)

규칙 3. 두 파일 모두 존재하지 않는다면 root 만 at 명령어를 사용할 수 있음.

**메모 포함[이62]:** 초기 설정은 이 경우로 되어 있다.  
allow파일이 없고 deny파일이 비어있는 상태.

### 옵션들.

-l (엘) : 현재 실행 대기 중인 명령의 전체 목록을 출력 (atq 명령과 동일)

-r <작업 번호> : 현재 대기 중인 명령들 중에 해당 작업 번호를 가진 명령을 삭제

-m : 작업이 완료되면 사용자에게 메일로 알려줌

-f <file> : 표준 입력 대신 실행할 명령을 파일로 저장함.

-d : at 작업을 삭제함. (atrm 명령과 동일)

## - crontab

메모 포함[이63]: 다시 해보기. 잘 안됨.

작업을 정해진 시간마다 반복해서 실행시켜주는 crontab 파일을 관리하는 명령어.

```
crontab <option> <file>
```

### crontab 파일 형식

분(0~59)	시(0~23)	일(1~31)	월(1~12)	요일(0~6)	작업 내용
---------	---------	---------	---------	---------	-------

(\* 특수문자를 이용해서 여러 시각을 지정할 수 있음)

(요일은 일요일이 0.)

(- 와 /를 이용할 수 있음. 1시부터 23시까지 2시간 간격 -> 1-23/2 이렇게 작성하면 됨. \*/2 -> 2분마다 실행.)

**at 명령어 제한** -> /etc/cron.allow(허가)와 /etc/cron.deny(금지) 파일 이용

cron.deny 파일은 기본적으로 있지만 cron.allow 파일은 기본적으로 만들어져 있지 않아서 관리자가 만들어야 사용이 가능함.

규칙 1. /etc/cron.allow 파일이 존재하면 이 파일에 있는 사용자만 crontab 명령어를 사용할 수 있음.

/etc/cron.deny 파일은 무시됨.

규칙 2. /etc/cron.allow 파일이 존재하지 않으면 /etc/cron.deny 파일에 있는 사용자를 제외한 모든 사용자는 crontab 명령어를 사용할 수 있음. (/etc/at.deny 파일에 아무것도 적혀 있지 않다면 모든 사용자가 사용 가능.)

규칙 3. 두 파일 모두 존재하지 않는다면 시스템 관리자만 cron 명령어를 사용할 수 있음.

### 옵션들.

-e : 사용자의 crontab 파일을 편집 또는 생성. (edit) (/var/spool/cron/crontabs 디렉토리에 저장됨.)

-l (엘) : crontab 파일 목록을 출력.

-r : crontab 파일을 삭제. (remove)



## <백업 관련 명령어들>

### - gzip

파일 압축 및 압축 해제 명령어

```
gzip <option> <file>
```

디렉토리는 압축할 수 없음.

압축된 파일은 파일명 뒤에 .gz 가 붙음

옵션들

-d : 압축 해제

-l (엘) : 압축된 파일 목록 출력

-r : 명시한 하위 디렉토리 내의 모든 파일 압축. (<file>에 디렉토리를 적으면 됨.)

### - tar (tape archive)

아카이브 생성, 추출 명령어

```
tar <option> <file1> <file2>
```

file2에 있는 내용들을 묶어서 file1아카이브를 만듦.

옵션들

-c : 새로운 아카이브 생성

-x : 아카이브로부터 파일 추출

-t : 아카이브 내용 출력 (아카이브를 푸는 것.)

-v : 아카이브 처리 과정 출력

-f : 아카이브 이름 명시

-z : 아카이브에 gzip 사용

메모 포함[이64]: 원지 모르겠음. 나중에 배우면 강의 다시 들어보고 정리.

아카이브는 약간 묶거나 풀 수 있는 그룹을 말하는 것 같음.

## <네트워크 관련 명령어들>

메모 포함[이65]: **뭔 소린지 모르겠음. 나중에 알려줄 듯**

- netstat

네트워크 연결, 라우팅 테이블, 인터페이스 통계 등을 보여주는 명령어.

netstat <option>

옵션들

- a : 모든 연결 및 수신 대기 상태에 있는 포트들을 출력
- n : 주소 및 포트번호를 10진수로 출력
- r : 현재 설정된 라우팅 정보를 출력
- s : 각 프로토콜에 대한 통계 정보를 출력

- ping

네트워크상의 컴퓨터에 ICMP\_REQUEST 를 보내는 명령어

ping <option> <hostname>

주로 특정 컴퓨터가 네트워크에 활성화되어 있는지 확인하기 위해 사용함.

라우터나 대상 호스트의 보안 설정 시 응답 수신이 불가능함

## <계정 및 시스템 관련 명령어들>

- date

현재 날짜와 시각을 출력하거나 설정하는 명령어

date <option> <format>

옵션들

- s <string> : 해당 string 으로 시간 설정.

- uname, hostname

uname : 시스템 정보를 출력하는 명령어

hostname : 시스템의 호스트 이름을 출력하는 명령어

uname

hostname

## < 셸 관련 명령어 >

### - echo

배시 셸의 문장 출력 명령어

```
echo <option> <text>
```

옵션들

-n : 인자를 출력한 후 개행 문자를 삽입하지 않음

\$로 문장을 시작하면 셸에 문장을 친 것으로 됨.

### - printf

배시 셸에서 c 언어이 표준 출력 함수인 printf 를 구현한 명령어.

단순히 인수만 작성한다면 그 인수를 출력함.

printf 함수의 형식에 따라 ' "문자열", 인수 2, 인수 3 '과 이스케이프 시퀀스, 변환명세 등을 사용할 수 있음.

(ex. printf "%d + %d = %d", 4, 5, 9 와 같이 입력할 수 있음.)

(연산까지 다 해주는 것 같지는 않음.)

```
printf <option> <인수>
printf <option> <printf 함수의 형식>
```

### - export

셸 변수를 환경변수로 바꾸는 명령어

```
export <option> <셸 변수>
```

옵션들.

-n : 환경 변수를 셸 변수로 바꿈. (셸 변수 자리에 환경변수를 넣으면 됨.)

### - unset

지정한 환경설정 관련 변수를 해제하는 명령어.

```
unset <변수>
```

## <기타 명령어들>

### - whereis (리눅스+)

명령어의 절대 경로 출력 명령어.

```
whereis <option> <command>
```

옵션들.

-b : 바이너리 파일만 검색함.

-m : 메뉴얼 파일만 검색함.

-s : 소스 파일만 검색함.

### - which

```
which <option> <command>
```

명령어의 절대 경로 출력 명령어

하나의 절대 경로를 찾으면 즉시 명령을 종료함. -> 하나의 경로만 출력됨.

에일리어스가 지정된 명령일 경우에는 해당 에일리어스를 먼저 검색해 출력함.

option 종류

-a : 해당 명령어로 접근할 수 있는 모든 경로 출력

### - man

명령어의 메뉴얼 출력 명령어

```
man <명령어>
```

### - clear

터미널 화면을 초기화하는 명령어

```
clear
```

## - ln (link)

파일의 링크를 생성하는 명령어

```
link <option> <file> <link_file>
```

디렉토리는 하드링크를 생성할 수 없음. 심볼릭 링크는 생성할 수 있음.

심볼릭 링크는 파일 시스템이 달라도 생성할 수 있음

옵션들

-s : 심볼릭 링크 파일 생성. (symbolic)

## + 파일 링크

### 1) (하드) 링크

파일에 붙은 이름을 하드 링크라고 함.

하드 링크 파일은 원본 파일과 inode 번호가 같음. (단순히 기존 파일에 새로운 파일명을 추가하는 것임)

링크파일을 vi 로 수정하면 원본파일과 심볼릭 링크 파일에 반영됨.

원본파일을 제거해도 하드 링크 파일은 사용할 수 있음.

### 2) 심볼릭 링크

기존 파일을 가리키는, inode 가 다른 새로운 심볼릭 링크 파일 생성. (바로그가기)

심볼릭 링크 파일의 내용은 원본 파일의 경로임. -> 원본 파일이 삭제되면 사용할 수 없는 파일이 됨.

심볼릭 링크 파일의 파일 종류는 l(소문자 엘)로 시작함.

ls -l 명령어로 확인하면 원본 파일이 "링크파일 -> 원본파일" 형태로 명시됨.

심볼릭 링크 파일을 실행하면 원본파일의 경로가 아니라 원본파일의 내용이 출력됨.

심볼릭 링크 파일을 vi 로 수정하면 원본파일과 하드 링크 파일에 반영됨.

## + ls -l 으로 확인할 수 있는 하드링크 개수의 기본값은 1임.

기본적으로 1개의 이름을 가지고 있다는 의미.

하나의 하드링크를 생성할 경우 ls -l 로 보면 하드링크 개수가 2임.

## + 옵션 작성법

옵션은 -ab 형태로도 작성할 수 있고, -a -b 형태로도 작성할 수 있음.

## + sudo

관리자 권한으로 명령을 수행하라는 의미.

파일 관련 명령어 또는 루트 관련 명령어에는 관리자 권한이 필요할 수도 있음.

(ex. sudo chown jhun test)

메모 포함[이66]: 닉네임 부여.

메모 포함[이67]: 파일 복사와의 차이점.  
cp로 복사할 경우 둘은 별개의 파일임.  
링크의 경우에는 둘의 파일명은 다르지만  
inode 번호가 같기 때문에 동일한 파일을 가리킴.

메모 포함[이68]: 원본파일을 제거하면 하드 링크  
파일이 남아 있어도 심볼릭 링크 파일은 사용할 수  
없다.

## 3. c 언어 기초

# 1. c 언어 기초

### 1. printf() 기초

문자열 출력 함수.

```
printf("제어문자열", 인수2, 인수3);
```

#### 1) 인자

첫번째 인자에는 출력할 문자열인 **제어문자열**이 들어감.

두번째 인자부터는 제어문자열 내의 변환명세에 대입될 값이나 변수, 수식이 들어감.

인자의 개수에는 제한이 없음.

#### 2) 변환명세

값을 어떤 형식으로 출력할 것인지 명시하는 키워드. -> 적절하지 않은 형식을 사용하면 이상한 값이 출력됨.

%로 시작하여 문자(**변환문자**)로 끝남.

printf 함수에서는 변환명세를 반드시 맞춰 줘야한다. 자동변환 이런 거 없음.

**메모 포함[이69]:** 경고 메시지가 나오고 이상한 값이 출력됨.

자료형이 달라도 큰 문제가 없는 경우에는 정상적인 값이 나오기도 함.

#### + 정수와 문자의 변환명세 사용

문자는 아스키 값에 따라 어차피 정수로 변환되어 저장되므로 %d를 사용할 수 있음.

정수 또한 마찬가지로 %c를 사용할 수 있음.

## <정수>

1) %d (decimal), %i : 부호를 고려하는 10 진 정수로 출력.

%<#><부호><0><전체 칸 수>d

칸 수가 남는 경우, %와 변환문자 사이에 있는 숫자의 부호가 (-)인 경우 왼쪽 정렬됨.

(부호가 명시되지 않은 경우에는 오른쪽 정렬 로 인식함.)

칸 수가 남을 때 빈칸에 0 을 넣고 싶다면 부호와 숫자 사이에 0 을 넣으면 됨. (오른쪽 정렬일때만 가능)

전체 출력 칸 수를 조절할 수 있음.

필요한 기능만 명시하여 사용함.

메모 포함[i70]: - 와 0을 같이 쓰면 0이 무시된다.

2) %x (hexadecimal) : 16 진수로 출력

% 바로 뒤에 #을 넣으면 0x 까지 붙여서 출력함. (%#x)

#를 사용해서 출력한 0x 는 전체 칸수에 포함됨. (2 칸 추가로)

소문자 x 대신 대문자 X 를 작성하면 16 진수의 알파벳이 대문자로 출력됨.

메모 포함[i71]: #은 %와 숫자 사이에 작성해야 함.  
순서 틀리면 컴파일 에러남.

3) %o (octal) : 8 진수로 출력

% 바로 뒤에 #을 넣으면 0 까지 붙여서 출력함. (%#o)

#를 사용해서 출력한 0 은 전체 칸수에 포함됨.

%<부호><0><전체 칸 수>.<소수점 자릿수>f

## <실수>

1) %f (float) : 실수를 소수점 표기로 출력 / 복소수 출력

칸 수가 남는 경우, %와 변환문자 사이에 있는 숫자의 부호가 (+)인 경우 오른쪽 정렬, (-)인 경우 왼쪽 정렬됨.

(부호가 명시되지 않은 경우에는 (+)로 인식함.)

칸 수가 남는 경우, 빈칸에 0 을 넣고 싶다면 부호와 숫자 사이에 0 을 넣으면 됨.

전체 출력 칸 수를 조절할 수 있음. -> . 도 출력 칸 수에 포함되므로 유의.

마침표 뒤에 숫자를 작성하여 소수점 이하 몇 자리까지 출력할 것인지 정할 수 있음.

마침표를 적지 않으면 default 값이 반영됨.

마침표만 적고 뒤에 아무것도 적지 않으면 뒤에 0 을 적은 것으로 인식되어 소수점 밑은 작성하지 않음.

복소수를 출력하는 경우, %+f 와 같이 +기호를 넣으면 복소수의 실수부 또는 허수부 부호가 출력됨.

메모 포함[i72]: + 과 소수점 자릿수 등을 합했을 때  
전체 출력 칸 수보다 크면  
전체 출력 칸 수를 지키지 않고 그냥 넘겨서 씀

전체 출력 칸 수는 비는 칸에 0을 넣기 위한 정도로  
만 가능하다. 다른 곳에는 영향을 끼치지 않는 것으로  
보인다.

2) %e : 실수를 지수표현으로 출력.

%f 와 사용법 동일. 소수를 나타내는 부분도 동일하게 6 자리가 default 임.

3) %a : 실수를 16 진 부동형으로 출력.

%f 와 사용법 동일. f, e 와는 다르게 소수 부분이 6 자리로 고정되어 있지 않음.

메모 포함[i73]: + 소수점 6자리 출력이 default값(기본값)임.

+ 소수점 자릿수를 변경할 때에는 반올림이  
적용됨.

4) %g : 소수점 표기와 지수표현 중에서 적절한 것으로 출력.

<문자, 문자열>

1) %c : 문자 상수로 출력

2) %s : 문자열로 출력

printf()는 NULL 문자 전 까지만 출력함.

### 3) 이스케이프 시퀀스

첫번째 인자에 직접 입력하거나 두번째 이상의 인자에 넣고 %c 를 입력해서 사용할 수 있음.

**메모 포함[이74]:** \는 하나의 문자로 취급하지 않음. 즉, \n은 %c로 받을 수 있음.

## 2. scanf() 기초

문자열 입력 함수.

scanf("변환명세", &변수);

printf()와 사용법이 유사함.

변환 명세의 형태로 입력 받아서 명시한 변수에 저장함. -> 일반 변수의 경우, &를 반드시 써 줘야함

%c 는 공백 문자도 입력받음.

배열을 인자로 작성할 때는 배열 이름만을 명시하고, &를 작성하지 않음.

scanf()를 여러 번 사용할 때도 하나의 scanf 에서 여러 값을 받을 때와 동일하게 입력하면 됨.

변환 명세와 다른 종류의 값을 입력하면 이상한 결과가 나옴.

**메모 포함[이75]:** 따옴표 안에 작성한 형식대로 맞추어 입력 받은 후 enter를 누르면 해당 변수에 저장되는 듯. (해당 띄어쓰기, 문자 등을 해당 형식에 맞게 작성해야 함.)

**메모 포함[이76]:** 띄어쓰기 입력 시에 %c가 있으면 공백 문자가 입력될 수 있다.

**메모 포함[이77]:** scanf("%d", &a);  
scanf("%d", &b);  
scanf("%d", &c); 와  
scanf("%d%d%d", &a, &b, &c); 는 동일한 입력으로 값을 받게 됨.

**메모 포함[이78]:** 단, %d와 %f 등은 공백 문자를 자동으로 무시한다.

**메모 포함[이79]:** 컴파일러는 문법에 맞는 프로그램만 컴파일하기 때문에 발생하는 오류이다.

## 3. 디버깅

오류를 없애는 과정.

오류에는 문법적 오류와 논리적 오류가 있음. (디버깅은 크게 두 가지임.)

문법적 오류가 있을 경우 오류 메시지가 뜨는데, 그걸로 문제를 해결할 수 있음.

### + scanf 와 printf 의 변환명세 차이

기능, 표현 등에서 미묘한 차이가 있음.

	printf	scanf
float	%f	%f
double	%f	%lf (대문자 L 은 안됨.)



## 4. c 컴파일 과정

- 1) 작성한 코드가 구문에 맞는지 검사 시작.
- 2) 프로그램을 토큰으로 분리, 검사.
- 3) 오류가 있으면 오류 메시지 출력 / 오류가 없으면 목적 파일로 변환
- 4) 목적 파일을 실행 파일로 변환

## 5. 주석문

문서화 도구.

프로그램이 일부분이지만 실행되지는 않음.

주석은 토큰이 아님.

컴파일러는 주석문을 하나의 공백문자로 처리함.

### 1) 다중 줄 주석

여러 개의 행에서 작성할 수 있는 주석.

/\*와 \*/사이에 작성.

```
/* */
```

### 2) 줄 단위 주석

하나의 행에서만 작성할 수 있는 주석.

//부터 해당 행 끝까지 작성.

```
//
```

### + 중첩 주석문

어떤 컴파일러는 주석문 안에 주석문이 들어가는 다중 주석문을 허용하기도 한다.

이 수업에서 사용하는 c 컴파일러는 허용하지 않는다.

**메모 포함[이80]:** 유용하게 사용할 수 있다.

다중 줄 주석을 이용해 프로그램에 대한 설명, 만든 날짜, 만든 사람 등을 적자.

다중 줄 주석을 이용해 중요한 내용을 강하게 시각화 할 수도 있다.

**메모 포함[이81]:** 프로그램 설명 등을 작성.

**메모 포함[이82]:** 어떤 컴파일러는 주석문 안에 주석문 이 들어가는 다중 주석문을 허용하기도 한다.

이 수업에서 설치하는 c컴파일러는 허용하지 않는다.

## 6. 헤더 파일

### 1) stdio.h (standard input output)

표준 입출력 함수들에 대한 헤더파일 (standard input output)

### 2) limits.h

정수를 저장하는 자료형의 크기가 들어 있는 헤더 파일.

'자료형\_MIN' '자료형\_MAX' 등을 정수로 출력하면 확인 가능. (자료형은 대문자로 작성함)  
(printf 두번째 이상의 인수에 넣고 %d로 받으면 됨.)

### 3) ctype.h

문자를 다룰 때 유용하게 사용할 수 있는 함수들을 포함하는 헤더파일. (character)

toupper() : 인수에 들어간 문자가 소문자라면 대문자로 바꾸어서 반환하는 함수.

### 4) stdbool.h (standard bool)

논리값을 더 편하게 다룰 수 있게 해주는 매크로들을 포함하는 헤더파일.

bool: \_Bool 대신에 bool 이라고 쓸 수 있게 됨.

true: 1 을 나타냄.

false: 0 을 나타냄.

### 5) complex.h

복소수 관련 함수, 매크로를 포함하고 있음.

complex: \_Complex

creal(): 복소수의 실수부를 알려줌

cimag(): 복소수의 허수부를 알려줌

### 6) iso646.h

등가연산자, 논리연산자를 더 알아보기 쉽게 표기할 수 있는 매크로를 포함하는 헤더파일.

not\_eq : != (equal)

not : !

and : &&

or : ||

## 7) math.h

수학 함수들이 들어 있는 헤더파일.

복소수를 위한 수학 함수들은 complex.h 에 들어 있음.

## 8) tgmath.h

통합 자료형 수학 매크로가 들어 있는 헤더파일.

## 9) stdlib.h

abort() 함수: 프로그램의 실행을 종료하는 함수.

## 10) time.h

시간과 관련된 함수들이 들어있는 헤더파일.

## 11) string.h

문자열과 관련된 함수들이 들어 있는 헤더파일.

## 2. c 프로그램의 구성요소

### 1. 구문 (syntax)

프로그램 작성 규칙.

### 2. 토큰

컴파일 시에 프로그램을 분석하는 기본 단위. (5 가지)

#### - 키워드 (예약어)

특별한 의미를 가지고 있는 단어.

예약된 단어이기 때문에 다른 의미로 재정의될 수 없음. (식별자 등으로 사용될 수 없음.)

(ex. int, void, float, char, return)

#### - 식별자

식별을 위해 사용하는 이름.

함수, 변수, 태그, 레이블 등에 사용됨.

알파벳, 숫자, 밑줄(\_)로 구성됨

대소문자 구분됨.

함수와 변수의 식별자는 달라야 됨. 레이블 이름은 함수나 변수의 식별자와 같아도 됨.

띄어쓰기 사용할 수 없음.

숫자로 시작될 수 없음

사용자 식별자로는 키워드를 사용할 수 없음

+ c 시스템에서 \_로 시작하는 식별자를 많이 사용하므로 식별자는 \_로 시작하지 않는 게 좋음.

+ 표준 라이브러리 함수명, main 등으로 식별자를 작성하지 않는 것이 좋음.

+ 식별자를 작성할 때는 의미가 잘 드러나게 해야 함.

+ main 도 식별자임. main 함수의 이름이므로.

메모 포함[이83]: 컴파일러의 작업이기 때문에 컴파일러에 따라 세부사항은 다를 수 있다.

메모 포함[이84]: 키워드와 예약어는 미묘한 차이가 있다고 한다.  
이 수업에서는 둘을 같은 것으로 보는 듯.

메모 포함[이85]: printf, scanf 등등

메모 포함[이86]: 이런 것들도 사용은 할 수 있음.

## - 상수

### 1) 정수 상수 (integer constant)

10 진수: 0~9

8 진수: 0~7 -> 0 으로 시작 (숫자 0 임.)

16 진수: 0~9, a~f -> 0x 로 시작 (영어 o 가 아니라 숫자 0 임.)

### 2) 부동형 상수 (실수 상수) (floating constant)

10 진 부동형 상수: 소수점 표기법(123.1)이나 지수표현(1.231e2)으로 표기.

16 진 부동형 상수: 16 진 부동형 상수 표기법으로 표기(0x23p4).

### 3) 문자 상수 (character constant)

작은 따옴표를 이용해 표시.

'character'

### 4) 와이드 문자 상수

일반 문자 상수에 접두사 L 붙여서 표시. (Large)

L'character'

유니코드를 다룰 때 사용.

### 5) 이스케이프 시퀀스

인쇄할 수 없는 문자를 표현함.

Wcharacter 은 한 문자로 취급됨

'Wcharacter'

특별한 의미를 가지는 문자들의 특별한 의미를 제거하고 원래 의미를 가지게 함.

이스케이프 시퀀스 또한 printf 에서 두번째 이상의 인자에 문자열 형태로 넣고 %c 로 받아서 사용할 수 있음. (제어문자열에서 원래 의미로 사용됨.)

\a	경고
\b	백스페이스
\f	폼피드
\n	개행
\r	캐리지 리턴
\t	수평 탭
\v	수직 탭
\0	널 문자
\	큰따옴표
'	작은따옴표
\\	역슬래시

### 6) 열거 상수 (enumeration constant)

enum 으로 정의된 식별자. 정수 상수로 취급됨. (나중에 배움.)

메모 포함[이87]: 16진수의 알파벳은 대문자로 써도 되고

소문자로 써도 됨.

메모 포함[이88]: 부동소수점이란 소수점의 위치를 고정하지 않고 그 위치를 나타내는 수를 따로 적는 표현 방식을 말함.

부동형 상수는 소수점 표기법, 지수 표현, 16 진 부동형 등으로 표현 가능함.

메모 포함[이89]: 전 세계 모든 문자를 컴퓨터에서 일관되게 표현하고 다룰 수 있도록 제정된 표준.

메모 포함[이90]: " ' w 등. %를 쓰려면 %%를 작성해야 한다. w로 안됨.

## - 문자열 상수 (string constant)

큰따옴표로 표시.

"string"

여러 개의 문자열 상수가 나란히 있다면 **접합되어 하나의 문자열 상수로 취급됨.**

문자열 중간에 **w**를 작성하면 그 뒤부터는 다음 줄에서 작성할 수 있음.

**(코드에서만 적용되는 것. 출력 시에는 이어짐.)**

문자열 상수에는 (가시적으로 드러나진 않지만) 마지막 문자 뒤에 끝을 알리는 NULL 문자가 들어감.

**메모 포함[이91]:** "A" "B" "C" 이런 경우.

이때 문자열 상수 사이의 공백은 반영되지 **않음**  
즉, ABC이런 식으로 취급된다는 것.

## - 구두자

의미를 가지는 기호들.

사용되는 위치, 같이 사용되는 문자들에 따라 의미가 달라지기도 함.

(ex. ( ), ++ 등)

**구두자의 대부분은 연산자임.**

## + 지수표현

10의 거듭제곱 꼴로 표현.

e 뒤에 10의 지수를 명시.

**printf()로 출력하면 e 뒤에 오는 지수는 <부호> <수> 형태로 표시하는데,**

**수가 한 자리이면 03 등의 형태로 표시하고, 부호는 항상 표시함.**

(ex. 314.0e-02)

## + 16진 부동형 상수 표기법

16진수와 2의 거듭제곱 꼴로 표현. **p 뒤에 2의 지수를 명시함.**

**printf()로 출력하면 항상 0x를 표시하고, 항상 p 뒤에 부호를 표시함.**

(ex. 0x12fp4)

## + c 프로그램의 논리적 무한대

inf (infinity) : 무한대를 의미함.

-nan -> 무한대끼리 빼니까 이게 나옴.

## + 정수와 실수

컴퓨터에서 정수와 실수는 독립된 집단임. (ex. 10과 10.0은 다른 수.)

정수와 실수는 저장되는 방식, 읽히는 방식이 완전히 다르기 때문에 명백히 구분됨.

## 4. 저장

# 1. 자료형과 변수 선언

## 1. 자료형 (Data type)

변수의 정보 저장 방식을 지정하는 문법.

c 프로그램의 모든 변수는 자료형이 명시되어야 사용 가능함.

자료형마다 정해진 변환 규칙이 있어 값을 이진수로 변환하거나 이진수를 값으로 변환함.  
자료형을 명시하면 이것이 자동적으로 수행됨.

대분류	소분류	기본 자료형
표준 정수형	부울형	<code>_Bool</code>
	문자형	<code>char</code> , <code>signed char</code> , <code>unsigned char</code>
	정수형	<code>signed short int</code> , <code>signed int</code>
		<code>signed long int</code> , <code>signed long long int</code>
		<code>unsigned short int</code> , <code>unsigned int</code> <code>unsigned long int</code> , <code>unsigned long long int</code>
표준 부동소수점형	부동소수점형	<code>float</code> , <code>double</code> , <code>long double</code>
	복소수형	<code>float _Complex</code> , <code>double _Complex</code> , <code>long double _Complex</code>

**메모 포함[이92]:** 컴퓨터는 모든 값을 이진수로 저장하기 때문에 변환이 필요함.

**메모 포함[이93]:** + 부울형과 문자형은 정수를 다룰 때에도 사용할 수 있다.

+ `signed` : 양수와 음수, 0 모두 다룸  
+ `unsigned` : 음수를 다루지 않음.

+ 복소수형은 지원하지 않을 수도 있다.

표준 정수형은 정수를 다루는 자료형, 표준 부동소수점형 (실수형) 은 실수를 다루는 자료형임.

### + 정수형 자료형의 생략된 형태

자료형은 대체로 생략된 형태로 표현함.

`signed` 자료형인 경우, `signed` 와 `int` 를 생략함. (`signed int` 제외.)

`unsigned` 자료형인 경우, `int` 를 생략함.

## + void 형

void 형은 값을 가지지 않음.

함수의 리턴 값이나 매개변수가 없음을 나타냄.

함수 정의, 원형 선언, 포인터 등에서 사용됨.

**메모 포함[이94]:** 모든 형을 다 포인팅하는 일반적인 포인터를 나타냄. -> 나중에 배움.

## 2. 선언문

변수를 선언하는 문장. 컴파일러가 해당 변수에 메모리 공간을 할당함.

<자료형> <식별자>;

콤마로 연결해서 한 번에 여러 변수를 선언할 수 있음.

선언문에서 값을 저장할 수도 있음.

선언한 변수에 어떤 다른 형의 값을 대입하면 해당 값의 형이 변수의 형으로 바뀌어 저장됨.

**메모 포함[이95]:** 이 메모리 공간은 하나의 값 만을 저장할 수 있음.  
특정 값을 배정하면 이전 값을 대체함.

## 3. sizeof 연산자 (~의 사이즈)

형의 크기를 반환하는 연산자.

sizeof (<피연산자>)

크기는 바이트 단위로 반환함.

인수에는 자료형이나 수식이 올 수 있음.

자료형이 아닌 수식이 피연산자에 들어갈 경우 괄호를 생략할 수 있음.

자료형이 피연산자에 들어갈 경우 괄호를 생략할 수 없음.

sizeof 연산자는 %ld 변환명세로 받아 줘야 경고 메시지가 안 남. 다른 정수 변환명세로 해도 컴파일은 됨.

**메모 포함[이96]:** 함수가 아니라 연산자이다.

**메모 포함[이97]:** 이때 제일 앞에 온 변수 하나만 처리한다.

예를 들어, sizeof a + b는 (sizeof a) + b와 동일함.

+ 8 비트 == 1 바이트



## 2. 정수의 저장

### 1. 정수를 저장하는 방식

메모리에는 0 또는 1 만 저장할 수 있기 때문에 10 진 정수를 저장하려면 이진수로 바꾸어 줘야 함.  
c에서는 부호 있는 10 진 정수를 부호 있는 2의 보수로 변환하여 저장함.

<정수가 비트에 저장되는 형식>

첫 번째 비트: 부호 비트 저장. (1 -> 음수, 0 -> 양수, 0)

나머지 비트: 2의 보수 저장.

<10진 정수를 2의 보수로 변환하는 방법>

양수 -> 2진수로 변환 (끝)

음수 -> 절댓값을 씌운 뒤 2진수로 변환, 그것의 1의 보수를 구한 후 1을 더함.

<2진수의 1의 보수 구하는 법>

각 값을 토글함. 즉, 0을 1로, 1을 0으로 바꿈.

+ 0과 -1은 부호 있는 2의 보수로 바꾸었을 때 특수한 값을 가짐.

0: 모든 비트가 0

-1: 모든 비트가 1

메모 포함[이98]: '나머지 비트'에 2의 보수가 저장된다는 것은,

오버플로가 될 때에도 첫 번째 비트에는 부호 비트가 들어가고 나머지 것들에서만 오버플로가 난다는 것.

첫 번째 비트에는 2의 보수를 저장하는 것이 아니다!!!!

## 2. 정수를 저장하는 자료형

기본적으로 정수는 int 자료형에 저장함.

자료형	크기 (바이트)	범위	변환명세
<b>short</b>	2	$-2^{15} \sim 2^{15}-1$ (약 $\pm 3$ 만)	%hd, %hi, %hx, %ho
<b>int</b>	4	$-2^{31} \sim 2^{31}-1$ (약 $\pm 21$ 억)	%d, %i, %x, %o
<b>long</b>	4	$-2^{31} \sim 2^{31}-1$ (약 $\pm 21$ 억)	%ld, %li, %lx, %lo
<b>long long</b>	8	$-2^{63} \sim 2^{63}-1$ (약 $\pm 922$ 경)	%lld, %lli, %llx, %llo
<b>unsigned short</b>	2	$0 \sim 2^{16}-1$ (약 6만)	%hu, %hx, %ho
<b>unsigned int</b>	4	$0 \sim 2^{32}-1$ (약 42억)	%u, %x, %o
<b>unsigned long</b>	4	$0 \sim 2^{32}-1$ (약 42억)	%lu, %lx, %lo
<b>unsigned long long</b>	8	$0 \sim 2^{64}-1$ (약 1844경)	%llu, %llx, %llo

(변환 명세의 h는 short의 h이고, l은 long의 l임.)

### + <limits.h>

정수를 저장하는 자료형의 크기가 들어 있는 헤더 파일.

'자료형\_MIN' '자료형\_MAX' 등을 정수로 출력하면 확인 가능. (자료형은 대문자로 명시해야 됨)

(printf 두번째 이상의 인수에 넣고 %d로 받으면 됨.)

## 3. 정수 상수가 가지는 자료형

상수 또한 수식이므로 형을 가지는데, 다음과 같은 규칙으로 형이 결정됨.

### 1) 접미사를 붙여서 형을 명시하는 경우.

접미사를 붙이면 해당하는 자료형들 중 제일 작은 자료형부터 시작해서(표에서는 위에서부터 시작해서), 그 정수 상수를 표현할 수 있는 첫 번째 것을 선택함.

접미사	자료형	예제
<b>u</b> 또는 <b>U</b>	unsigned	1024U
	unsigned long	0xfffffu
	unsigned long long	
<b>l</b> 또는 <b>L</b>	long	1024L
	long long	0xfffffl
<b>ul</b> 또는 <b>UL</b>	unsigned long	1024UL
	unsigned long long	0xfffffUL
<b>ll</b> 또는 <b>LL</b>	long long	1024LL
		0xfffffLL
<b>ull</b> 또는 <b>ULL</b>	unsigned long long	1024ULL
		0xfffffULL

### 2) 접미사가 붙어있지 않은 경우

int, long, long long 중에서 동일한 규칙으로 선택하여 해당 정수 상수의 형으로 함.

**메모 포함[이99]:** + long은 long double처럼 시스템마다 값이 다를 수 있음.

-> long, long double 이상한 놈.

+ short은 2바이트 인 것 유의하기.

+ 자료형 별 변환명세 명시 방법

출력 형식을 맨 뒤에 알파벳으로, 크기를 %바로 뒤에 명시한다.

unsigned에서는 d 대신 u를 쓴다.

x, o는 동일하게 사용된다.

i는 signed에만 존재한다.

short은 h를 추가.

long은 l을 추가.

long long은 ll을 추가.

**메모 포함[이100]:** 접미사와 변환명세가 형태가 조금 다르다.

혼동 주의!

**메모 포함[이101]:** + 읽는 순서로 알파벳을 명시한다.

+ 대문자 소문자 둘 다 됨. 대문자 소문자 혼용해서 작성해도 됨. ex. Ul, ul

변환명세에서는 대문자 쓰면 안됨. 완전히 동일하게 써야 함.

+ unsigned와 signed가 아예 나뉘어 있음.

+ 공통된 범위를 포함하고 있음.

## 4. 정수 오버플로

값이 저장공간을 초과할 때에 발생.

프로그램은 계속 진행되지만, 부정확한 값을 사용하게 됨.

2 진수로 변환하면 오버플로되는 값을 확인할 수 있음.

2 진수로 변환했을 때 칸을 넘어가는 왼쪽 비트를 버림.

---

+ 컴퓨터는 계산 가능한 범위 내에서만 계산이 정확함.

+ 실수를 정수 자료형의 변수에 저장하면 소수점 아래 수들은 삭제되어 저장됨. 올림 이런 거 아니고 삭제됨.

# 3. 문자의 저장

## 1. 문자의 변환

메모리에는 0 또는 1 만 저장할 수 있기 때문에 문자를 이진수로 바꾼 후 저장함.

문자를 숫자로 변환하는 방식은 정해져 있음.

-> ASCII 또는 EBCDIC 를 이용함.

## 2. 아스키 코드 (ASCII)

아스키 코드를 기준으로 문자를 숫자로 변환함.

아스키 코드에 있는 문자들은 인쇄 문자와 공백 문자로 나뉨.

**인쇄 문자**: 실질적으로 볼 수 있는 문자들.

**공백 문자**: 실질적으로 볼 수 없는 문자들, 공백 문자에는 스페이스, 탭, 개행 문자 등이 있음.

ASCII 표

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

이 표는 '행 열'로 읽으면 됨. (ex. a 는 9 번째 행의 7 번째 열에 있으므로 아스키 값은 97 임)

-> A, a, 0 정도는 아스키 값을 암기해두자. (순서대로이므로 하나씩만 외우면 됨.) 65 97 48

### 3. 문자를 저장하는 방식과 자료형

기본적으로 char 형을 사용함.

아스키 코드에 있는 문자는 총 256 개이기 때문에 char 형을 이용하면 아스키에 있는 문자들을 저장할 수 있음.  
모든 문자는 정수로 취급되기 때문에, 정수를 저장할 때 사용하는 int 자료형 등에는 문자를 저장할 수도 있음.

#### <char 형 (narrow 문자)>

char: 컴파일러에 따라 signed char 를 의미하기도 하고 unsigned char 를 의미하기도 함.

signed char: 1 바이트에 부호 있는 정수를 저장. (-128~127)

unsigned char: 1 바이트에 부호 없는 정수를 저장. (0~255)

자료형 char 의 변환 명세는 %c 임.

#### <wchar\_t 형 (wide 문자)>

문자의 개수가 256 개보다 많은 언어에서는 char 형을 사용하지 못하고 wchar\_t 형을 사용함.

일반적인 1 바이트 크기의 문자보다 크기가 더 큰 문자의 자료형.

#### <다중 바이트 문자>

char 형을 여러 개 사용해서 하나의 문자를 저장하는 방법.

wide 문자는 문자 각각의 크기가 같지만 다중 바이트 문자는 문자마다 크기가 다름.

### 4. 문자 상수가 가지는 자료형

문자 상수는 int 형을 가짐 (char 형 아님.)

#### + <ctype.h> 헤더파일

문자를 다룰 때 유용하게 사용할 수 있는 함수들을 포함하는 헤더파일.

toupper() : 인수에 들어간 문자가 소문자라면 대문자로 바꾸어서 반환하는 함수.

**메모 포함[이102]:** 문자를 주로 저장하는 자료형이기 때문에  
문자를 뜻하는 character의 앞 글자를 딴 이름이다.

**메모 포함[이103]:** char 형으로 선언한 변수에는  
문자와 정수 값을 저장할 수 있음.

애초에 문자도 정수로 바꾸어서 저장함.

**메모 포함[이104]:** 0을 포함하므로 양수에서 하나 가  
져오기 때문에 128까지가 아니라 127까지이다.

**메모 포함[이105]:** stddef.h 헤더파일을 사용해야 쓸  
수 있음.  
4바이트 크기.

# 4. 논리값의 저장

## 1. 참과 거짓

참: 0 이외의 값.

거짓: 0, (0, 0.0, 0.0, NULL 포인터)

## 2. 논리값의 저장

논리값만을 다루기 위한 자료형으로 `_Bool` 형이 있음.

`_Bool` 형은 1 바이트 크기임.

`_Bool` 형으로 선언한 변수에 0 이 아닌 값을 대입하면 1 로 변환되어 저장되고, 0 을 대입하면 0 으로 저장됨.

### + `<stdbool.h>` (standard bool)

논리값을 더 편하게 다룰 수 있게 해주는 매크로들을 포함하는 헤더파일.

`bool`: `_Bool` 대신에 `bool` 이라고 쓸 수 있게 됨.

`true`: 1 을 나타냄.

`false`: 0 을 나타냄.

메모 포함[이106]: 밑줄 왼쪽에 있다. 헛갈리지 말기.

# 5. 실수의 저장

## 1. 부동소수점

부동소수점이란 소수점의 위치를 고정하지 않고 그 위치를 나타내는 수를 따로 적는 표현 방식을 말함.

**메모 포함[이107]:** 여기서 '부동'은 움직이지 않는다는 뜻이 아니라 떠다니며 움직인다는 뜻임.

## 2. 실수를 저장하는 방식

c에서는 실수를 **이진 부동소수점**으로 변환하여 저장함.

<이진 부동소수점>

부호, 지수, 가수부로 나뉘어 있음.

부호 -> 해당 수의 부호. 1: 음수, 0: 양수

지수 -> 소수점의 위치를 나타내는 지수

가수 -> 수를 나타내는 가수

$$\begin{matrix} & & & & & & \text{지수} \\ & & & & & & -4 \\ 1.2345 & \times & 10 \\ \text{가수} & & & & \text{밑} \end{matrix}$$

**메모 포함[이108]:** 1을 옆으로 놓으면 -이므로 음수를 나타낸다. 라고 암기.

IEEE(전자 전기 기술자 협회)에서 개발한 IEC 60559 표준을 사용하여 저장함. (IEEE 부동소수 표준)  
이 표준에서 실수를 저장하는 방식에는 아래와 같은 것들이 있음.

### 1) 단정밀도

32 비트(4 바이트)에 저장.

0	1 ~ 8	9 ~ 31
부호	지수부	가수부

### 2) 배정밀도

64 비트(8 바이트)에 저장.

0	1 ~ 11	12 ~ 63
부호	지수부	가수부

3) 이외에도 단확장 정밀도(43 비트 이상), 배확장 정밀도(70 비트 이상) 등이 있음.

### 3. 실수를 저장하는 자료형 (부동형)

c에서는 실수를 저장하기 위해 부동형을 사용함.

float: 단정밀도로 저장

double: 배정밀도로 저장

long double: 시스템에 따라 다름

#### 부동형의 속성

가수부와 지수부의 크기(비트 수)에 의해 결정됨.

정밀도: 부동형이 표현할 수 있는 유효숫자의 수

범위: 부동형으로 저장할 수 있는 수의 범위

#### float

정밀도: 대략 유효숫자 6 자리

범위: 대략  $10^{-38} \sim 10^{38}$

#### double

정밀도: 대략 유효숫자 15 자리

범위: 대략  $10^{-308} \sim 10^{308}$

자료형	크기 (바이트)	범위	변환명세
float	4	$10^{-38} \sim 10^{38}$	%f, %e, %g, %a
double	8	$10^{-308} \sim 10^{308}$	%f, %e, %g, %a
long double	?	?	%Lf, %Le, %Lg

### 4. 부동형의 한계

부동형으로 모든 실수를 표현할 수는 없음

정수 산술 연산과는 달리 부동형 산술 연산은 정확하지 않을 수 있음

메모 포함[이109]: = 부동소수점형

메모 포함[이110]: printf(), scanf()에서 실수 변환명세의 소문자 대문자 유의하기.

long double은 우분투 리눅스에서 16바이트임.

long double 변환명세가 %Lf인데 여기서 L이 소문자가 아닌 것에 유의하기. 이런 거 하나 안지키면 컴파일 에러 남.

long double인데 Ld가 아니라 Lf인 것 유의하기. L만 추가하는 것임. 정수의 ld랑 헷갈리지 않도록 유의하기.

실수는 signed unsigned를 따로 나누어 저장하지 않는 것 유의하기.



## 5. 실수 상수가 가지는 자료형

접미사	자료형	예제
f, F	float	10.24F
		1.024e3f
		0x1p10f
없음	double	10.24
		1.024e3
		0x1p10
l, L	long double	10.24L
		1.024e3L
		0x1p10L

메모 포함[이111]: 대, 소문자 상관없음.

기본값은 double인 것 유의.

double은 접미사가 존재하지 않는 것 유의하기.

long double은 ld가 아니고 L인 것 유의하기.

변환명세에서도 그렇고, long double에는 L 또는 l 하  
나만 붙인다.

# 6. 복소수의 저장

## 1. 복소수

실수부와 허수부로 구성됨.

허수부는  $i()$ 나  $j()$ 로 나타냄. (대문자로 써도 됨)

(ex.  $2.5 + 4.2i$ )

## 2. 복소수를 저장하는 자료형

`_Complex` 형을 사용함.

크기에 따라 종류가 나뉨.

실수 자료형 뒤에 붙임.

`float _Complex` (8 바이트)

`double _Complex` (16 바이트)

`long double _Complex` (시스템에 따라 크기가 다름.)

`_Complex` 형 변수는 두 개의 부동형 실수 값을 가짐.

-> `float`, `double`, `long double` 에 비해 크기가 각각 두 배임.

실수부와 허수부를 한꺼번에 작성하여 복소수의 연산을 진행할 수 있음.

(출력할 때는 `creal()` 함수와 `cimag()` 함수를 사용해야 함)

메모 포함[이112]: 이거 C 대문자인 거 조심.  
밑줄 앞에 있는 거 조심.

메모 포함[이113]: ex.  $(1.3 + 14.2i) + (2.4 + 42.1i)$   
이런 식으로 연산할 수 있다.

### + <complex.h> 헤더파일

복소수 관련 함수, 매크로를 포함하고 있음.

`complex`: `_Complex` 를 대체.

`creal()`: 복소수의 실수부 값을 반환함. 이 함수의 반환값의 형은 `double` 임. (complex real)

`cimag()`: 복소수의 허수부 값을 반환함. 이 함수의 반환값의 형은 `double` 임. (complex imaginary)

# 7. 형변환

일시적으로 수식의 형을 바꾸어 해당 수식의 '값'을 사용할 수 있게 해주는 문법.

## 1. 일반적 산술 변환 (자동 형 변환) (묵시적 형 변환)

연산하는 두 피연산자의 형이 다른 경우, 컴파일러가 일시적으로 형을 변환시켜 특정 형을 가지게 하는 것.

형을 바꾸어 정확한 계산을 하도록 하기 위함.

일반적 산술 변환으로 연산 수식의 형 또한 결정할 수 있음.

프로그래머가 관여하지 않아도 컴파일러가 알아서 선언문을 참조하여 처리함.

### <일반적 산술 변환의 과정>

#### 1) 정수 승격 (해당되지 않는 경우 2 번으로.)

\_Bool, char, signed char, unsigned char, signed short, unsigned short, 비트 필드 형은

산술 수식에서 int 나 unsigned int 로 변환됨.

int 형으로 표현할 수 있으면 int 형으로 변환하고, 그렇지 않으면 unsigned int 형으로 변환함.

1, 2 바이트 크기의 자료형들을 4 바이트인 int 나 unsigned int 로 취급해버리는 것.

#### 2) 기본 규칙에 따라 변화를 가함.

두 피연산자의 자료형 중 순위가 높은 것으로 자료형을 통일함.

아래 표에서는 위로 갈수록 순위가 높고, 같은 행에서는 왼쪽으로 갈수록 순위가 높음.

값의 범위가 큰 것이 순위가 높음. (값의 범위가 큰 것으로 통일해야 자료의 손실이 최소화되기 때문)

자료형 순위
long double _Complex, long double
double _Complex, double
float _Complex, float
unsigned long long, long long
unsigned long, long
unsigned, int (정수 승격 후)

순위에서도 알 수 있듯이, signed 형과 unsigned 형을 연산하면

이 둘을 통일하는 과정에서 이상한 값이 나올 수 있음. 되도록이면 이 둘을 연산하지 말아야 함.

**메모 포함[이114]:** 해당 연산이 끝나면 두 피연산자는 원래의 형으로 고려해줘야 함. (변수인 경우)

해당 수식의 형은 당연히 그대로 사용하게 됨

**메모 포함[이115]:** 비트 필드 형을 제외하면 전부 int 형으로 바뀐다.

지금까지 배운 자료형들 중에서만 생각하면, 4 바이트 보다 작은 것들은 모두 int 형으로 바뀐다.

#### **메모 포함[이116]: <정수>**

- 크기가 큰 것이 순위가 높음
- 같은 크기(높이)라면 unsigned가 순위가 높음

#### **<실수>**

- 실수는 항상 정수보다 순위가 높음
- float, double, long double 순으로 순위가 높음
- \_Complex는 붙어 있는 자료형과 같은 높이의 순위임
- 같은 높이라면 \_Complex의 순위가 높음

**메모 포함[이117]:** 3개 이상의 피연산자들을 한꺼번에 연산해도

연산 순서에 따라 연산이 진행되기 때문에 한 번에 두 개씩 계산하게 됨.

## 2. 캐스트 연산자

프로그래머가 직접 명시하여 처리하는 형변환

(<형>) <수식>

특정 형을 괄호를 씌워 수식 앞에 작성하면 일시적으로 해당 수식의 값을 명시한 형으로 바꾸어 사용할 수 있음.

변수 앞에 캐스트 연산자를 사용한다고 해서 변수의 자료형이 바뀌는 것은 아님.  
변수 안에 저장된 값의 자료형이 바뀌는 것.

캐스트 연산자를 앞에 붙인 변수는 변수가 아니고 '값' 으로만 사용이 가능함.

(ex. (char)min 은 min 변수의 값을 char 형으로 바꾼 값임. 변수가 사용되는 게 아님.)

대입 연산자를 사용할 때 =의 왼쪽에 캐스트 연산자가 올 수 없음.

변수 자체의 형을 바꾸려는 의도로 인식되기 때문. (변수의 형은 바꿀 수 없음.)

예초에 대입 연산자의 왼쪽에 올 수 있는 것은 변수인데 캐스트 연산자가 붙으면 변수가 아님.

(double) (quarter = 2.5) 이런 건 됨. quarter = 2.5 라는 수식의 값에 적용되는 것이기 때문.

## 5. 수식과 연산자

### 1. 수식

#### 1. 수식

데이터에 대한 계산식

상수, 변수, 함수, 통합형 선택 등 모두 수식임.  
연산자와 피연산자로 구성된 것도 수식이라고 함.

대부분의 수식은 '값'과 특정 규칙에 의한 '형'을 가짐.

수식 뒤에 세미콜론이 오면 수식은 하나의 문장이 됨.

메모 포함[이118]: void는 값을 가지지 않음.

+ '상수;' 나 ' ; ' 같은 것도 아무런 기능을 하지는 않지만 문장임.

## 2. 연산자

### 1. 연산자

#### 1) 배정 연산자 (대입 연산자)

변수에 값을 대입하는 연산자.

오른쪽에 변수가 들어간다면 해당 변수에 저장된 값이 대입됨.

왼쪽에는 단 하나의 순수한 변수만 올 수 있음.

대입 연산자가 쓰인 수식의 형은 해당 변수의 형이고, 값은 대입하는 값임.

변수의 형과 대입하는 수식의 형이 다르다면 변수의 형으로 바뀌어 대입됨. (그 변수에 저장하는 것이므로)

<변수> = <수식>

**메모 포함[이119]:** ex. 3a, -a같은 것들은 변수가 아니기에 올 수 없다. 대입을 할 수 없는 것을 직관적으로도 알 수 있다.

#### 2) 산술 연산자

+: 더하기, 양수 부호

-: 빼기, 음수 부호

\*: 곱하기

/: 나누기

?: 나머지 값 구하기, **두 피연산자는 정수여야 함. 정수가 아니면 컴파일 에러 발생함.**

#### + 정수 연산자와 실수 연산자

산술 연산자는 정수 연산자와 실수 연산자로 나뉜.

연산의 대상인 수 중 하나라도 실수이면 실수 연산이 수행됨. 두 수 모두 정수라면 정수 연산이 수행됨.

실수 연산인 경우 답은 실수이고, 정수 연산인 경우 **소수점 아래 값은 버리고** 답은 정수임.

특히 / 연산의 경우, 정수 연산자는 몫을 값으로 가지지만, 실수 연산자는 소수점까지 계산한 값을 가짐.

**메모 포함[이120]:** 연산자의 종류는 컴파일러가 판단해 번역한다.

이렇게 나뉘는 이유는 정수와 실수가 완전히 다른 별개의 집단이기 때문이다.

**메모 포함[이121]:** 일반적 산술 변환을 생각하면 당연함.

실수 자료형들이 정수 자료형들보다 순위가 높음.

**메모 포함[이122]:** **소수점 아래 값은 반올림하는 것이 아니라 그냥 버린다.**

**메모 포함[이123]:** 방향과 관련된 규칙이 아니라 증감 연산자의 위치에 따라 연산자의 의미가 달라진다.

#### 3) 증감 연산자

++ : 1 증가

-- : 1 감소

<연산자><변수> 또는 <변수><연산자>

증감 연산자에는 전위와 후위가 있음.

전위 증감 연산자는 연산자를 앞에 씌. 변수 자체의 값을 1 증감시키고 연산의 결과값은 변수  $\pm 1$  임.

후위 증감 연산자는 연산자를 뒤에 씌. 변수 자체의 값을 1 증감시키고 연산의 결과값은 변수 임.

여기서 결과값은 '값'임. 여러 연산과 섞어서 사용할 때에도 '값'으로 받아들이면 이해가 쉬움.

**증감 연산자는 변수에만 사용할 수 있음. 변수가 아닌 것에 사용할 경우 컴파일 에러 발생함.**

#### 4) 복합 배정 연산자

<변수> <연산자>= <수식>

배정 연산자를 줄여 쓴 것.

다른 연산자와 배정 연산자가 연결된 형태.

배정 연산자는 자신의 연산 순위가 오면 왼쪽에 있는 대입할 변수와 오른쪽의 수식을 해당 연산자로 연산한 후 왼쪽에 있는 변수에 대입함.

여기서 연산 순서에 따라 수식의 연산이 끝난 뒤 변수와의 연산이 진행됨.

(ex. `value += 3 * 4` 는 `value = value + 12` 와 같음.)

---

#### + 여러 개의 배정 연산자를 동시에 사용할 수 있음

연산방향을 적용해서 연산하면 됨.

(ex. `a = b = c = 0`; 은 `a = (b = (c = 0))` 과 같음. `c` 에 0 을 대입하고 `c = 0` 수식을 `b` 에 대입하는 식.)

+ `a +++ b` 는 `(a ++)` + `b` 와 같음.

## 5) 관계 연산자, 등가 연산자, 논리 연산자

참 또는 거짓이 연산의 결과값으로 나오는 연산자들.

결과값이 참이면 정수 1 을 도출하고 거짓이면 정수 0 을 도출함.

### 관계 연산자

<, >, <=, >= : 두 값을 비교해서 결과값을 참 또는 거짓으로 도출하는 연산자들.

### 등가 연산자

== : 양쪽의 값이 같으면 참을 도출하는 연산자. -> 배정 연산자 =와 유의해야 함.

!= : 양쪽의 값이 다르면 참을 도출하는 연산자.

### 논리 연산자

! (느낌표) : 값이 거짓인 경우 참을 도출하는 단항 연산자. (부정)

&& : 두 값 모두 참인 경우 참을 도출하는 연산자. (교집합) (논리곱)

|| (shift+백슬래시) : 두 값 중 하나만 참이면 참을 도출하는 연산자. (합집합) (논리합)

## + 단축 평가

c에서는 논리곱, 논리합 연산자를 계산할 때 필요한 만큼만 계산함.

왼쪽부터 계산하는데, 연산 수식의 참 또는 거짓이 결정되는 순간 계산을 멈춤.

&&를 계산할 때, 왼쪽 수식이 거짓이면 오른쪽 수식은 수행 또는 계산되지 않음.

||를 계산할 때, 왼쪽 수식이 참이면 오른쪽 수식은 수행 또는 계산되지 않음.

(ex. apple || orange++ 에서 apple 이 참이면 orange 에는 +1 이 반영되지 않음.)

-> 이 성질을 이용해서 제어의 흐름에 영향을 끼칠 수도 있음. 조건문처럼 사용 가능.

## + <iso646.h> 헤더파일

등가연산자, 논리연산자를 더 알아보기 쉽게 표기할 수 있는 매크로를 포함하는 헤더파일.

not\_eq : !=

not : !

and : &&

or : ||

+ a != b 와 !(a == b)는 동일한 의미의 수식임.

메모 포함[이124]: !=은 항상 오른쪽에 작성한다.

메모 포함[이125]: !는 항상 왼쪽에 붙인다.

!는 단항 연산자이다.

메모 포함[이126]: 양 옆을 묶는 느낌. 둘 다 필요하니까.

메모 포함[이127]: 양 옆을 나누는 느낌. 둘 중에 하나만 있으면 되니까.



## 6) 조건부 연산자

<수식1> ? <수식2> : <수식3>

expression1 이 참이면 expression2 값을 반환하거나 expression2 를 수행하고,  
expression1 이 거짓이면 expression3 값을 반환하거나 expression3 를 수행하는 연산자

형: expression2 와 expression3 의 형 중 크기가 더 큰 것이 됨.

값: expression2 와 expression3 중 반환되는 것.

조건부 연산자는 중첩해서 사용할 수 있음

우에서 좌로 연산이 진행됨 (물음표와 :를 기준으로 묶으면 편함).

(ex. (n >= 90) ? 'A' : (n >= 80) ? 'B' : 'C'는 ((n >= 90) ? 'A' : ((n >= 80) ? 'B' : 'C'))와 같음.)

if~else 조건문을 어느 정도 대체할 수 있음. (복잡한 경우 가독성이 떨어질 수 있음.)

expression2, expression3 에 복합문은 적을 수 없음.

(expression = 조건수식)

메모 포함[이128]: 아마..? 실험해보니 그런 것 같기는 함.

## + 단항 연산자와 이항 연산자

단항 연산자: 하나의 수식에 사용하는 연산자. (ex. !)

이항 연산자: 두 개의 수식에 사용하는 연산자. (ex. <, >, <=, >=, ==, !=, &&, ||)

이항 연산자들은 두 개의 수식을 대상으로 사용한다는 것을 괄호로 명시해야 실수가 없음.

세 개 이상의 수식들을 이항 연산자로 한 번에 나타낼 경우 컴파일은 되지만 의미적 오류가 발생할 수 있음.

(ex. 0 <= x <= 100에서 <=은 이항 연산자이므로 왼쪽부터 연산을 함.

0 <= x의 참 거짓을 판별해 0 또는 1이 나오고 그 후 0 또는 1과 100을 비교하게 되므로

이 연산 수식의 결과는 항상 참임. 이런 경우 && 등을 이용해 각각 두 개의 수식을 대상으로 하는 두 개의 연산 수식으로 나누어서 계산해야 함.)

## 7) 콤마 연산자

수식과 수식을 연달아 쓸 때 사용하는 연산자.

언제나 사용할 수 있음.

(ex. 명령 1, 명령 2; )



왼쪽->오른쪽 순서로 수식들을 실행함.

값과 형은 오른쪽 수식의 값과 형이 됨.

모든 연산자들 중에 가장 낮은 우선순위를 가짐.

## 8) 부호 연산자 (+, -) (단항)

값에 + 또는 -부호를 붙여서 사용할 수 있게 해주는 연산자.

---

+ 함수의 매개변수나 선언문에서 변수를 구분하기 위해 사용하는 콤마는 콤마 연산자가 아니라 구두점임.

## 2. 연산자 우선순위

여기서는 위쪽일수록 연산순위가 높음

같은 순위일 경우 방향을 적용하여 연산함.

우선순위	결합방향
++(후위) --(후위)	좌에서 우
+(단항) -(단항) ++(전위) --(전위) (type-name) sizeof !	우에서 좌
* / %	좌에서 우
+ -	좌에서 우
< > <= >=	좌에서 우
== !=	좌에서 우
&&	좌에서 우
	좌에서 우
= += -= *= /= %=	우에서 좌

(방향은 우에서 좌인 것들만 암기하면 편함.)

### + 수식 평가

c에서는 각 수식의 평가 순서를 정의하지 않음.

수식의 값을 계산할 때의 우선순위는 존재하지만 수식 자체의 우선순위는 존재하지 않음.

(ex. printf("A") + printf("B") \* printf("C"); 일 경우 A, B, C 중 뭐가 먼저 출력될지는 알 수 없음)

연산자 우선순위는 값을 계산할 때 적용되는 것이고, 수식 자체의 우선순위는 존재하지 않는다는 거지?

메모 포함[이129]: 부호 연산자

전위 증감 연산자

캐스트 연산자

sizeof 연산자

부정 연산자

? 조건부 연산자

배정 연산자, 복합 배정 연산자

메모 포함[이130]: 컴파일러마다 다름.

## 6. 제어의 흐름

제어의 흐름이란 문장이 실행되는 순서를 의미함.

제어의 흐름은 위에서 아래로 진행되지만, 특수한 작업을 위해 제어의 흐름을 변경해 줌.

조건문(선택문), 반복문, 점프문을 이용함.

### 1. 조건문

#### 1. if 조건문 (만약 ~라면 ~를 수행.)

조건 수식의 결과가 참이면 (0 이 아닌 수이면) 하나의 명령을 수행하는 조건문.

여러 개의 명령을 사용하려면 복합문을 작성해야 함.

명령문이 간단하더라도 복합문으로 작성해야 실수가 없음.

```
if (expression)
{
    명령문;
}
```

(expression = 조건수식)

#### + 복합문 (블록)

여러 개의 명령을 중괄호 {}로 묶은 문장.

하나의 단위로 그룹화한 것.

복합문 또한 하나의 문장이므로 c에서 하나의 문장이 들어가는 자리에는 복합문을 사용할 수 있음.

#### + if 조건문을 중첩해서 사용하는 경우, 복합문으로 명령을 적어줘야 실수가 없음.

else 가 영동한 if 에 적용되는 등 논리적 실수가 발생할 수 있음.

## 2. if~ else 조건문 (만약 ~라면 ~를 수행. 그렇지 않으면~)

조건수식의 결과가 참이면 if에 있는 명령을 수행하고,  
조건수식의 결과가 거짓이면 else에 있는 명령을 수행하는 조건문.

else는 마지막으로 작성된 if와 호응함.

if 조건문에 else를 추가로 사용하는 것.

else가 나오면 어느 if와 호응하는지 확인해야 함.

else는 가장 먼저 만나는 if와 호응하기 때문에 괄호를 잘 봐야 함.

(else if도 마찬가지.)

```
if (조건 수식)
{
    명령문;
}
else
{
    명령문;
}
```

### + else if

중첩된 if~else 문을 정리하면 else if 형태로 작성할 수 있음.

```
if (조건 수식)
{
    명령문;
}
else if (조건 수식)
{
    명령문;
}
else
{
    명령문;
}
```

### 3. switch 조건문

조건 수식과 case 문의 수식을 비교해서 명령을 수행하게 하는 조건문.  
정수 수식끼리 비교하는 조건문.

조건 수식은 반드시 정수 수식이어야 함.

case 문에 들어가는 수식에는 반드시 정수 상수 수식이어야 함.

이 수식의 값은 switch 문에서 유일해야 함. (겹치면 컴파일 에러 발생)

```
switch (조건 수식)
{
    case <수식>:
        명령문;
    ...
    default:
        명령문;
}
```

**메모 포함[이131]:** 정수 상수 수식이어야 한다.  
a = 1; 이렇게 해놓고 a를 쓰면 컴파일 에러남.

#### 진행 과정

- 1) 조건 수식과 수식의 값이 같은 case 문을 찾음.
- 2) 찾았으면 해당 명령을 실행하고, 그 아래에 있는 모든 case 문의 명령들을 실행함.
- 3) 찾지 못했으면 default의 명령을 실행함. default가 없다면 switch 문에서 빠져나감.

일반적으로는 break 문을 각 case 별로 작성해서 원하는 case 문의 명령까지만 실행하게 함.

break의 사용에서 실수하면 의미적 오류가 발생할 수 있음.

일단 break 문을 적고 특정 상황에서만 빼 주는 게 좋을 듯.

다중 조건문이므로 if 문을 여러 개 쓰는 것보다 실행 속도가 빠름.

## 2. 반복문 (Loop)

### 1. while 반복문 (~하는 동안 ~)

조건 수식이 참인 경우 그것이 거짓이 될 때까지 명령을 반복해서 수행하는 반복문.

**제어 변수:** 조건 수식에 들어가서 반복의 조건을 결정하기 위해 사용되는 변수.

조건 수식에 1 을 작성해 무한 루프를 만들 수 있음.

#### 주의점 1: 무한루프

모든 가능한 경우를 고려하지 않으면 무한루프에 빠지기 쉬움.

#### 주의점 2: 공백 문장

공백 문장을 사용할 때에도 복합문으로 작성해야 실수를 줄일 수 있음.

#### 주의점 3: 등가(==, !=) 연산자를 사용한 실수 수식

실수 연산은 정확하지 않기 때문에 등가 연산자를 사용한 경우 실수가 발생할 수 있음.

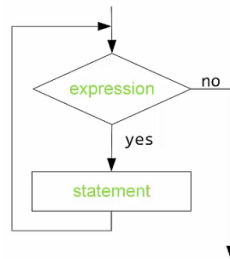
실수를 사용해야 할 경우 등가 연산자보다는 관계 연산자를 사용하는 것이 안전함.

(등가 연산자는 정확히 그 값이어야 하지만 관계 연산자는 약간 벗어나도 어느 정도는 커버가 됨.)

(ex. 0.0 에 0.1 을 99 개 더해도 정확히 9.9 가 나오지 않을 수 있음)

while (조건 수식)

```
{  
    명령문;  
}
```



**메모 포함[이132]:** 반복의 3요소

반복문을 사용할 때에는 3개의 요소를 기억해야 한다.

1. 시작조건
2. 조건수식 (종결조건)
3. 조건변화수식

**메모 포함[이133]:** 컴퓨터를 사용한 실수 계산의 한계.

컴퓨터는 모든 실수를 표현할 수는 없음.

+ 프로그램 작성 시에는 모든 경우의 수를 고려하였는지 검토해야 함.

#### + 무한 루프

종료되지 않는 반복문.

필요에 따라서 고의적으로 만들 수 있음.

#### + 공백 문장

수식이 작성되지 않고 세미콜론만 작성된 문장.

## 2. do-while 반복문 (~해라 ~하는 동안)

while 문과는 다르게, 반복문의 하단에서 검사를 함.  
고로, 명령이 한 번은 반드시 실행됨.

```
do
{
    명령문;
}
while (조건 수식)
```

## 3. for 반복문

조건식 3 요소를 정규화된 형식으로 표현하는 반복문.

순서를 고려하여 흐름을 따라가야 함.

각 요소는 생략 가능. (세미콜론은 적어야 됨.)

조건수식 칸에 아무것도 작성하지 않으면 무한루프가 됨.

(참으로 인식함)

시작조건에서 변수를 선언할 수 있음.

이때 이 변수는 for 안에서만 사용 가능. (지역변수)  
밖에서도 이 변수를 사용하려면 밖에서 선언해야 함.

식 사이에 :가 아니라 ;인 것 주의하기.

for() 뒤에 ; 안 붙이는 거 주의하기.

명령까지 실행해야 for 문이 끝나는 것이므로.

```
for(시작조건; 조건수식; 조건변화수식)
{
    명령문;
}
```



n 번 반복하기 위해 for 문을 사용하는 경우, 시작조건은 0 으로 시작하는 것이 편함.

배열도 0 부터 시작하므로.

for 문 안에서 선언한 변수는 지역변수로 취급됨.

### + 어떤 반복문을 사용해야 하는가?

정해진 횟수만큼 반복해야 할 경우에는 for.

반드시 한 번은 실행해야 하는 경우에는 do-while.

어떤 조건을 기준으로 반복을 진행할 때에는 while.

정해진 횟수가 없는데 for 등을 사용하면 고달파질 수 있음.



## 3. 점프문 (제어문)

### 1. break

제어를 현재의 구조물에서 벗어나게 하는 명령.

반복문이나 switch 문에서만 사용할 수 있음.

종결조건 등으로 사용될 수 있음.

```
break;
```

### 2. continue

1 회성 취소 명령.

반복문에서만 사용함.

사용 시 해당 반복문에 있는 명령 중 제일 마지막에 있는 것 뒤로 감.

즉, continue 는 해당 루프의 이후 명령들을 수행하지 않고 다음 루프로 넘어 감.

```
continue;
```

### 3. return

함수를 종료하고 함수의 값과 제어를 호출한 곳으로 보내는 명령.

함수의 형이 void 이면 수식을 작성하지 않고, void 가 아니면 수식을 포함함.

return 에 수식이 명시되면 해당 값을 리턴함.

리턴 시 값은 함수 정의에 명시된 함수의 형으로 변환됨.

함수에는 두 개 이상의 return 문이 작성될 수 있음.

return 을 작성하지 않은 경우, return; 이 생략된 것으로 취급됨.

```
return <수식>;
```

### 4. goto

지정된 레이블이 있는 곳으로 무조건 점프하게 하는 명령.

해당 레이블 문장은 goto 와 같은 함수 내에 있어야 함.

c 의 구조를 파괴하고 가독성이 떨어지기 때문에 사용하지 않는 것이 좋음.

```
goto <레이블명>;
```

## 5. 레이블 문장

### 1) <레이블명>: <문장>

레이블 이름은 함수 내에서 유일해야 함.

레이블은 자신의 이름 영역을 가짐.

-> 변수나 함수와 이름이 같아도 됨. 다만 가독성이 떨어짐

레이블 이름으로는 case 와 default 를 사용할 수 없음.

여러 개의 레이블을 붙이는 다중 레이블 문장도 사용할 수 있음.

다중 레이블 문장은 레이블 안에 레이블을 적은 형태임.

여러 개의 레이블 중에 goto 의 부름을 받은 것 다음부터 실행이 됨.

레이블을 지나갈 때에는 해당 레이블을 그냥 넘기는 것이 아니라 제어의 흐름에 따라 한 번 실행함.

여러 개의 단일 명령들을 복합문으로 만들지 않고 그냥 씀.

(ex. first: //첫번째

```
int n;
```

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

first 라는 레이블 안에 여러 명령이 들어갈 수 있음.)

<레이블명>: <문장>

### 2) case <수식>: <문장>

switch 조건문에서만 사용할 수 있음.

여러 개의 단일 명령들을 복합문으로 만들지 않고 그냥 씀.

case 문에 들어가는 수식에는 반드시 정수 상수 수식이어야 함.

case <수식>: <문장>

### 3) default: <문장>

switch 조건문에서만 사용할 수 있음.

여러 개의 단일 명령들을 복합문으로 만들지 않고 그냥 씀.

default: <문장>

### + getchar 응용.

```
while((op = getchar()) != ' ')\n{\n    ;\n}
```

공백 문자를 제외한 문자를 입력했을 때만 op 변수에 저장하는 반복문.

**메모 포함[이134]:** exit1: (명령1); exit2: (명령2); exit3: (명령3);  
인 경우, goto exit2; 명령을 내리면 명령 2와 명령 3만 수행됨.

# 7. 배열과 정렬

대량의 데이터를 다룰 때의 변수 사용법.

## 1. 배열

### 1. 배열

같은 형의 많은 데이터를 다룰 때 사용하는 문법.  
많은 변수를 같은 형으로 한 번에 선언하는 것.

<자료형> <식별자>[크기];

#### 1) 배열의 원소

배열의 각 객체를 원소라고 함.  
배열의 크기는 원소의 개수를 말함.

int math[5];  
- 메모리에 int 형 변수 5개 할당

math[0]  
math[1]  
math[2]  
math[3]  
math[4]

#### 2) 원소의 구분 (인덱스)

배열의 원소는 인덱스로 지정함.  
0 부터 (크기 - 1) 까지의 인덱스가 부여됨.  
인덱스를 명시할 때에는 정수 수식을 사용해야 함.

선언한 인덱스에 포함되지 않는 인덱스를 사용하여 원소를 명시하면 오류가 발생하거나 의미적 오류가 생김.

#### 3) 원소의 사용

배열의 각 원소는 일반 변수처럼 사용될 수 있음.  
(일반 원소처럼 값을 대입하고 사용될 수 있음)

#### 4) 배열 다루기

배열의 원소는 한꺼번에 다루어야 하는 경우가 많은데, 이때 반복문을 사용해서 접근하면 편리함.

메모 포함[이135]: 3.0 같은 수라도 실수 수식을 작성한 경우 컴파일 에러가 발생한다.

메모 포함[이136]: c는 배열의 범위를 검사해 주지 않음.

정확한 인덱스 범위만을 사용하는 것이 중요하다. 잘못된 정수를 사용하면 이상한 버그가 발생한다.

## 2. 배열의 초기화

<자료형> <식별자>[개수] = {초기화자1, 초기화자2, ...};

배열은 중괄호 안에 값들을 나열하여 초기화 할 수 있음.

(ex. int test[3] = {23, 15, 8};)

중괄호로 나타낸 숫자들의 목록을 '초기화 목록'이라고 함.

중괄호로 나타낸 초기화 목록의 각 값을 '초기화자'라고 함.

중괄호로 묶었더라도 ; (세미콜론)을 반드시 써 줘야 함. (복합문의 중괄호와와는 다름)

중괄호 안에 값들의 개수가 배열의 원소 개수보다 작다면, 인덱스 값이 작은 원소부터 초기화됨.

값이 지정되지 않은 원소들은 0 으로 초기화됨.

초기화자가 명시된 배열의 초기화에서는 배열의 개수를 명시하지 않아도 됨.

초기화자의 개수가 배열의 크기로 인식됨.

특정 원소를 초기화하고 싶다면 초기화 목록에서 대괄호(각괄호)로 나타낼 수 있음.

초기화 목록은 인덱스 값이 작은 원소부터 초기화하다가 특정 원소를 초기화하는 부분을 만나면 해당 부분 이후부터 인덱스 값이 작은 원소를 초기화함.

한 문장에서 두 번 초기화되는 원소는 마지막으로 초기화된 값을 저장함.

(ex. int test[10] = {12, 13, [5] = 23, 4, [1] = 10}; 이면 [1]이 10, [2]이 13, [5]이 23, [6]이 4 로 초기화됨.)

<자료형> <식별자>[개수] = {초기화자1, [2] = 초기화자2, ...};

### + 기호 상수 : define 전처리기

define 전처리기를 이용해서 기호를 정의하여 상수처럼 사용할 수 있음.

#define <기호> <수식>

자주 쓰이는 값 대신에 사용하여 수정을 용이하게 함.

기호와 수식은 반드시 괄호로 묶여줘야 의미적 오류가 생기지 않음.

### + 일반 변수의 크기와 배열의 크기

같은 int로 정의된 변수와 배열이라도 배열은 여러 개의 원소를 가지므로 그 크기가 다름.

**메모 포함[이137]:** 일반적인 대입은 이런 방식으로 할 수 없다.

선언문에서만 이런 방식의 초기화가 가능하다.

선언문이 아닌 문장에서 값을 대입하려면 각 원소별로 따로 대입해 줘야 한다.

**메모 포함[이138]:** 실수 자료형 변수라면 0.0으로 초기화됨.

**메모 포함[이139]:** 모든 원소들을 0으로 초기화하려면 int test[10] = {0};을 작성하면 됨.

(실수 자료형 변수라도 일반 변수처럼 0을 저장하면 0.0으로 저장됨.)

## 2. 다차원 배열

배열을 원소로 갖는 배열을 다차원 배열이라고 함.  
선언문에서 대괄호(각괄호)의 개수에 의해 차원이 결정됨.

### 1. 이차원 배열

<자료형> <식별자>[개수][개수];

1 차원 배열을 원소로 갖는 1 차원 배열.

선언문에서 대괄호(각괄호)의 개수가 2 개임.

앞에 오는 대괄호(각괄호)가 중심임. (높은 차원의 원소임)

(ex. int test[7][4]; 이면 [4]라는 일차원 배열을 일곱 개의 원소로 가지는 배열임.)

이차원 상의 직사각형으로 이해하면 쉬움.

#### 1) 이차원 배열의 사용

두 가지 속성을 갖는 정보를 저장할 때 효과적임.

이름	국어	영어	수학	과학
Noah	85	90	88	99
Emma	82	87	75	90
Liam	79	88	92	88
Olivia	99	75	81	75
Logan	85	92	77	92
Ava	100	85	81	67
William	76	95	87	90

#### 2) 이차원 배열 다루기

이차원 배열을 한꺼번에 다룰 때에는 두 개의 반복문을 중첩 사용하는 것이 편리함.

가장 바깥쪽 반복문에 배열의 가장 높은 차원의 원소를 대응시키는 것이 좋음.

메모 포함[이140]: 반대로 할 경우 프로그램 속도가 느려질 수 있음.

## 2. 삼차원 배열

```
<자료형> <식별자>[개수][개수][개수];
```

2차원 배열을 원소로 갖는 1차원 배열.

선언문에서 대괄호(각괄호)의 개수가 3개임.

제일 앞에 오는 대괄호(각괄호)가 중심임. (높은 차원의 원소임)

(ex. int test[3][7][4]; 이면 [7][4]라는 이차원 배열을 세 개의 원소로 가지는 배열임.)

삼차원 상의 직육면체로 생각하면 쉬움.

### 1) 삼차원 배열의 사용

세 가지 속성을 갖는 정보를 저장할 때 효과적임.

### 2) 삼차원 배열 다루기

삼차원 배열을 한꺼번에 다룰 때에는 세 개의 반복문을 중첩 사용하는 것이 편리함.

가장 바깥쪽 반복문에 배열의 가장 높은 차원의 원소를 대응시키는 것이 좋음.

**메모 포함[이141]:** 반대로 할 경우 프로그램 속도가 느려질 수 있음.

### 3. 다차원 배열의 초기화

일차원 배열처럼 초기화자의 개수가 부족하면 0으로 초기화됨.  
일차원 배열처럼 특정 원소를 초기화 할 수 있음. 동일한 원리임.

#### 1) 일차원 초기화 목록 사용

일차원 배열의 초기화처럼 초기화자를 일차원적으로 나열하여 초기화할 수 있음.

다차원 배열의 순서대로 초기화자가 대응됨.

(ex. [0][0] [0][1] [0][2] 순으로 초기화됨.)

배열의 차원과 초기화 목록의 차원이 달라 헷갈릴 수 있음.

<자료형> <식별자>[개수] = {{초기화자1, 초기화자2}, ...};

#### 2) 다차원 초기화 목록 사용

가장 높은 차원의 원소를 기준으로 중괄호를 씌워 초기화할 수 있음.

중괄호로 씌워진 각 초기화자 묶음을 높은 차원의 앞쪽 원소부터 배분하는 느낌임.

이차원 배열이면 중괄호가 한 번 더 쓰이고, 삼차원 배열이면 중괄호가 한 번 또는 두 번 더 쓰임.

배열의 첫 번째 대괄호(각괄호)가 비어 있으면 초기화 목록의 중괄호 수를 개수로 함.

첫 번째 대괄호(각괄호)를 제외하고는 크기를 모두 명시해 줘야 함.

삼차원 배열의 초기화 시에 중괄호가 한 번만 더 쓰인 경우, 해당 중괄호는 제일 높은 차원에 사용됨.

**메모 포함[이142]:** ex. int test[2][3] = {{1, 2, 3},{4, 5, 6}}  
이때 내부의 각 중괄호는 test[0], test[1]을 위한 초기화 목록이다.

**메모 포함[이143]:** 중괄호와 원소의 개수가 다른 경우에는, 초기화자 묶음을 받지 못한 원소는 0으로 초기화된다.

**메모 포함[이144]:** ex. int test[][3] = {{1, 2, 3},{4, 5, 6}}  
으로 쓸 수 있음. 빈 대괄호(각괄호)는 2로 인식된다.

**메모 포함[이145]:** ex. int a[2][2][2] = {{1, 2}, {3, 4}}; 인 경우,  
a[0] 에 1과 2가, a[1]에 3과 4가 순서대로 들어가게 된다.

# 3. 함수와 배열

배열은 주로 반복문과 사용하기 때문에 코드 길이를 줄이고 가독성을 높이기 위해  
따로 함수의 인자로 넣어 다루는 것이 일반적임.

## 1. 배열을 다루는 함수

### 1) 함수 정의

함수 정의 시 배열을 매개변수 목록에 작성하는 방법은 일반적인 변수와 같음.

매개변수 목록의 최상위 차원 배열의 [] 안에 아무것도 적지 않으면 임의의 크기의 배열을 인자로 받음.  
함수에 들어가는 명령들 중 배열의 크기를 알아야 하는 것들이 있다면, 배열의 크기를 인자로 받아주면 됨.

### 2) 함수 호출

호출 시에는 배열 이름만 명시함.

인자에 작성하는 배열의 크기는 매개변수 목록에 정의한 배열의 크기와 같아야 함.  
배열의 크기가 서로 다른 경우, 경고 메시지가 뜨며 예상치 못한 오류가 발생할 수 있음.

### 3) 함수 원형 선언

함수 원형 선언에서 배열을 명시할 때는 괄호 안 자료형 뒤에 차원의 개수에 맞게 []를 명시함.  
이때 첫 번째 [] 이외의 각괄호에는 원가를 명시해 줘야 함. 이때 \*를 쓸 수 있음.

<자료형> <식별자>(<자료형>[], ...);

### 4) 함수로 배열 전달하기

인자의 배열과 그걸 받은 매개변수의 배열은 동일한 배열임.

배열을 함수로 전달하면 매개변수의 배열은 인자 배열의 이름으로 초기화됨.

일반 변수를 함수로 전달할 때에는 값의 의한 호출이 적용되지만

배열을 함수로 전달할 때에는 배열의 위치가 복사되어 전달됨. (그렇다고 참조의 의한 호출인 것은 아님)

**메모 포함[이146]:** ex.  
void test(int x, int array[x])  
{  
 return;  
}

이런 식으로 정의하면 됨.

이때 [] 안을 비워도 된다.  
동일한 기능을 한다.

**메모 포함[이147]:** 이것을 이용하면 배열에 값을 입력  
받을 때에도  
함수를 쉽게 이용할 수 있다.



+ **각괄호의 생략** -> 원리가 있는 것이 아닌, 일종의 약속임.

각괄호 안에 숫자를 명시해서 배열을 해당 색인 하나로 한정한다고 이해하면 편함.

이차원 배열에서 맨 뒤 괄호를 모두 작성하면 이차원 상의 한 원소를 의미함.

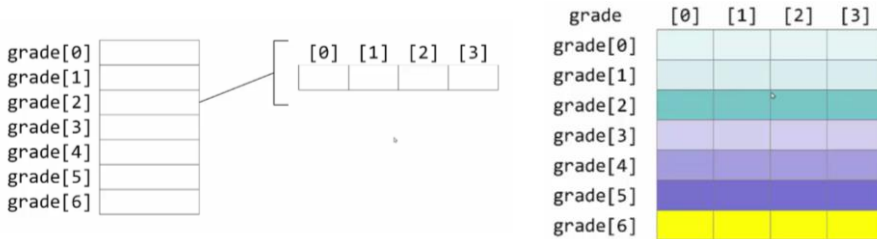
이차원 배열에서 맨 뒤 괄호 하나를 생략하면 높은 차원에 있는 일차원 배열의 원소를 의미함.

이차원 배열에서 맨 뒤 괄호를 모두 생략하면 이차원 배열 전체를 의미함.

grade : 이차원 배열

grade[i] : grade 의 i 행

grade[i][j] : grade 의 i 행의 j 원소



**메모 포함[이148]:** ex.

grade[0]은 grade[0][0], grade[0][1], grade[0][2], grade[0][3]을 원소로 갖는 일차원 배열을 의미한다.

**메모 포함[이149]:** 함수 호출 시에 인자로 grade[i]를 작성하면

i번째 행에 있는 원소들이 함수로 넘어간다.

즉, grade[i]라는 일차원 배열처럼 다룰 수 있다.

### 3. 가변길이 배열

배열의 크기를 컴파일 과정이 아닌 실행 과정에서 정할 수 있게 하는 기능.

가변길이 배열에는 두 가지 종류가 있음.

배열의 선언, 함수 정의, 함수 원형 선언 등, 배열의 크기가 명시되어 있는 상황에서는 최상위 차원 배열의 크기는 생략이 가능함.

최상위 차원이 아닌 배열의 크기는 생략할 수 없음.

#### 1) [\*]

함수 원형 선언 시에 최상위 차원이 아닌 [] 안에는 정수 수식만이 아니라 \*을 명시할 수도 있음.

(ex. void test(int array[\*]);)

#### 2) [<정수 수식>]

배열을 선언하거나 사용할 때 [] 안에는 정수 상수만이 아니라, 정수 수식도 들어갈 수 있음.

## 4. 정렬

## 1. 정렬

데이터를 특정한 크기 순으로 나열하는 것.

배열 데이터의 가장 기본적인 연산 중 하나.

배열은 큰 데이터를 다루는 데 사용되는데, 정렬을 통해 데이터를 잘 정리할 수 있음.

많은 정렬 방법들이 있음.

## 2. 버블 정렬

가장 기본적이고 단순한 정렬 방법.

적은 수의 데이터를 정렬할 때 적합함. 대량의 데이터에는 사용하지 않음.

## 1) 알고리즘

오름차순으로 정렬하는 방법

주어진 값들을 정렬된 부분과 정렬되지 않은 부분으로 나누어서 진행함.

1. 값들을 배열에 저장한다. (오른쪽이 높은 인덱스로 하여 각 배열의 원소를 하나의 칸으로 생각한다.)
2. 계산은 두 개의 원소끼리 이루어진다. 가장 큰 인덱스 값을 가진 원소와 그것 바로 이전 원소부터 계산을 시작한다.
3. 두 원소의 값을 비교하여 인덱스 값이 작은 쪽이 더 크다면 두 원소끼리 서로 값을 교환한다.
4. 한 칸 씩 왼쪽으로 이동하며 계산을 수행한다.
5. 결과적으로 제일 작은 값이 가장 작은 인덱스 값을 가진 원소(가장 왼쪽)에 저장된다.
6. 가장 작은 값이 저장되었다면 해당 부분은 정렬된 부분이다. 정렬되지 않은 부분끼리 동일한 과정을 계속해서 거친다.
7. 모든 칸이 정렬된 부분이 되면 끝난다.

+ 오름차순, 내림차순

오름차순: 제일 작은 것부터 제일 큰 것까지 나열하는 것.

내림차순: 제일 큰 것부터 제일 작은 것까지 나열하는 것.

[illegible]

**메모 포함[이150]:** 이 과정을 코드로 구현할 때에는 또 다른 변수 하나를 이용한다.  
한 번에 바뀌버릴 수 없으므로 다른 하나에 담아 두고 옮기는 것.

# 8. 함수

## 1. 함수

큰 프로그램을 여러 개의 작은 단위로 나누어 쉽게 코드를 작성할 수 있게 하는 문법.

### 1. 함수

프로그램을 기능별로 나누어 작성할 수 있게 하는 도구.

프로그램은 하나 이상의 함수로 구성됨.

식별자 뒤에 괄호가 붙는 것들은 함수와 관련이 있음. (함수 선언, 함수 정의, 함수 호출 중 하나)

### 2. 함수의 제어 흐름

1. 프로그램을 실행하면 main 함수가 실행된다.
2. 함수가 호출되면 함수의 첫 문장으로 제어가 넘어가 흐르고, return 문을 만나면 함수가 종료된다.
3. 함수가 종료되면 제어는 해당 함수가 호출되었던 곳으로 돌아가 흐르게 됨.
4. main 함수의 `return` 문을 만나면 프로그램이 종료됨.

**메모 포함[이151]:** main 함수 또한 return문을 만나면 값을 리턴함

### 3. 함수 지정자

#### 1) inline

함수의 구조적 장점을 살리면서 **실행 속도를 빠르게 하기 위해** 사용.

함수 헤더 자료형 앞에 inline 을 명시하여 사용함.

inline 함수를 호출하면 해당 호출 문장을 호출된 inline 함수의 몸체로 대체함.

반드시 대체되지는 않고, **컴파일러가 판단**하여 대체함.

대체되는 형태도 컴파일러마다 다름.

**주로 길이가 짧은 함수에 사용함.**

#### 2) \_Noreturn

함수를 호출한 곳으로 제어를 보내지 않기 위해 사용.

\_Noreturn 함수가 return 문을 가지고 있으면 경고 메시지가 출력됨.

\_Noreturn 함수가 return 문을 적용한 경우에는 컴파일러마다 행동 양식을 적용함.

\_Noreturn 함수에서 함수를 끝내려면 stdlib.h 헤더파일의 **abort()** 함수 등을 사용해야 함.

#### + 함수 사용의 이점

하향식 프로그래밍: 전체적으로 작업을 설계 후 작은 단위(함수)로 분해하여 코드를 작성하는 방식.

코드 재사용 가능

디버깅이 쉬움

유지보수가 쉬움

#### + **프로시저**

함수를 프로시저라고 하기도 함.

프로시저: 리턴값이 없는 함수

함수: 리턴값이 있는 함수

#### + 라이브러리로 제공되는 함수의 정의 및 선언

라이브러리에 있는 함수는 헤더파일로 읽기 때문에 따로 정의하거나 선언할 필요가 없음.

#### + **abort()** 함수

프로그램의 실행을 종료하고 프로그램 종료 시의 상태를 출력하는 함수.

<stdlib.h> 헤더파일에 들어 있음.

**메모 포함[이152]:** 함수는 호출과 리턴 시에 추가적인 연산이 있어서, 프로그램 실행시간을 길어지게 한다는 단점이 있다.

**메모 포함[이153]:** 컴파일러가 inline을 적용하지 않기로 한 경우에는 inline이 적힌 함수는 일반적인 방식으로 호출되지 않아서 의도하지 않은 결과가 발생한다.

이때 inline 앞에 static을 명시해 주면 해당 함수가 현재 파일에서만 참조된다는 의미가 부여되고, inline이 작동하지 않아도 해당 함수가 일반적인 함수처럼 작동함.

**메모 포함[이154]:** 길이가 너무 긴 함수에 inline을 사용하면, 여러 번 호출하거나 하는 경우에 실행 파일이 너무 길어진다는 문제가 발생한다.

**메모 포함[이155]:** **밀줄** 있는 거 조심.

**메모 포함[이156]:** 아무것도 작성하지 않을 경우 return이 생략된 것으로 취급되기 때문에 직접 종료해야 함

**메모 포함[이157]:** 이 수업에서는 굳이 구분하지 않음.

## 2. 함수의 정의, 호출, 선언

### 1. 함수 정의

함수가 호출되면 실행할 코드를 정의하는 것.

함수 헤더: 함수의 맨 위 행.

함수 몸체: 중괄호 부분.

```
<자료형> <식별자>(<매개변수 목록>)\n{\n    명령;\n}
```

#### 1) 함수 식별자 (이름)

#### 2) 매개변수 목록

함수가 입력 받아 처리하는 데이터 목록.

매개변수 목록을 작성할 때에는 각 변수를 선언해 줘야 함. (자료형을 명시해 줘야 함)

매개변수 목록이 필요하지 않다면 void 를 작성함.

#### 3) 함수의 형

함수 헤더의 맨 앞에 오는 자료형은 리턴값의 자료형임. (함수의 형이라고도 함)

리턴값이 없다면 void 를 작성함.

#### 4) 함수 정의의 절차

첫째, 함수 기능 정의

둘째, 함수 식별자 설정

셋째, 함수 매개변수 목록 결정 (자체적으로 얻을 수 없는 값들을 가져와야 함)

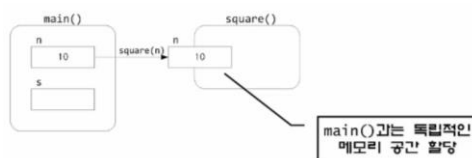
넷째, 함수 형 결정

다섯째, 함수 몸체 작성

#### 5) 메모리 할당

함수는 호출 시에 메모리가 할당되고, 실행을 종료하게 되면 메모리 공간은 사라짐.

함수마다 새로운 메모리 공간이 할당되므로 함수 내에서 선언한 변수는 지역변수인 것.



## 2. 함수 호출

정의되어 있는 함수를 사용하는 것.

함수는 호출해야 실행됨. (main 함수는 자동으로 호출됨)

<식별자>(<값>);

### 1) 인자의 작성

호출 시에는 함수 매개변수에 대응되는 인자를 작성해 줘야 함.

인자의 개수와 매개변수의 개수가 같아야 함. 다르면 컴파일 오류가 발생함.

인자와 매개변수는 서로 호환 가능한 형이어야 함.

함수가 호출되면 인자에 작성한 값으로 함수의 매개변수가 초기화됨.

함수 정의 시에 매개변수 목록이 없는 (void) 함수라면 인자에 아무것도 작성하지 않음.

이 경우 인자를 작성한다면 컴파일 오류가 발생함.

(ex. value = test(); )

**메모 포함[이158]:** main함수는 따로 호출하지 않는다. 프로그램을 실행하면 자동으로 호출됨.

**메모 포함[이159]:** 컴파일러가 인자의 형과 매개변수의 형을 일치시킨다.

다른 형을 가지고 있더라도 해당 값으로 초기화하는 것이기 때문에 인자의 형이 매개변수의 형으로 변환되어 사용됨.

### 2) 리턴값

함수가 리턴하는 값을 값으로 가짐.

함수 식별자 앞에 작성한 형으로 값을 반환하여 리턴함.

함수의 형이 void 인 경우 리턴값이 없음. 이 경우 함수를 호출하여 배정하면 컴파일 오류가 발생함.

리턴값이 있더라도 사용하지 않을 수 있음.

(ex. 리턴값이 없거나 사용하지 않으려면 test(a, b); 만 작성함)

### 3) 호출 시 인자들의 평가 순서

표준으로 정해져 있지 않음. (컴파일러마다 다름)

평가 순서에 따라 다른 결과를 가져오도록 함수를 작성해선 안 됨.

(ex. test(x, ++x); 인 경우 x가 먼저인지 ++x가 먼저인지 알 수 없음)

### + printf의 함수 호출 시 리턴값

printf의 리턴값은 화면에 출력되는 문자의 개수임.

공백 문자도 개수에 포함됨.

이스케이프 시퀀스는 하나의 문자로 취급됨. (ex. \n 은 하나로 취급.)

(ex. printf("Hello! \n"); 의 리턴값은 8 임)

### + scanf의 함수 호출 시 리턴값

scanf는 올바르게 입력된 변환명세의 개수를 리턴함.

(ex. int a, b;

scanf("%d%d", &a, &b); 인 경우 8.8 9.9를 입력하면 a에는 8이, b에는 .8이 저장되어야 하는데

.8은 정수로 변환할 수 없으므로 아무 값도 저장되지 않음. 즉, 올바르게 입력되지 않음. 리턴값은 1임)

### 3. 함수 원형 선언

<자료형> <식별자>(<매개변수 형 목록>;

함수 정보를 알려주는 문장.

함수 정의가 함수 호출보다 뒤에 있을 때 사용함.

괄호 뒤에 바로 ; (세미콜론)을 적어야 하는 것 유의하기.

#### 1) 함수 원형의 형태

함수 헤더와 유사한 형태.

함수 식별자, 인자의 개수, 리턴 값의 형을 알려줌.

매개변수 형 목록에는 매개변수들의 자료형만 작성함.

(ex. int test(int, float); 이렇게 자료형만 작성해도 됨)

매개 변수 형 목록에 변수명까지 작성해도 됨.

함수 원형 선언에서는 어떤 변수명을 작성하더라도 의미 없음. (아무거나 적어도 됨)

(매개변수 형 목록에 작성하는 매개변수 식별자는 컴파일러가 취급하지 않음)

(ex. int test(int); 와 int test(int A); 와 int test(int kk); 가 모두 같은 함수 원형 선언임)

#### 2) 함수 원형 선언의 필요성

함수를 호출하기 전에 해당 함수의 존재와 정보를 확인할 수 있어야 함.

함수 호출 전에 함수가 정의되어 있지 않다면 함수 원형을 선언하여 정보를 제공해야 함.

#### + 묵시적 int

함수의 정의나 함수의 선언에서 함수의 형이 생략되면 함수의 형을 int 로 취급하는 것.

C90 까지는 해당 기능을 사용할 수 있었지만 C99 부터는 삭제됨.

대부분의 최신 컴파일러는 이전에 작성된 프로그램을 위해 묵시적 int 를 지원하기는 하지만

함수의 형을 명시하는 것이 현재의 표준을 따르는 것임.

#### + main 함수의 위치

프로그램은 보통 main 함수에서 시작하고 끝나기 때문에

main 함수는 찾기 쉽도록 주로 맨 앞이나 맨 뒤에 작성함.

맨 앞에 작성하는 것이 일반적.



# 3. 함수의 매개변수 전달법

함수 호출 시에 인자가 매개변수로 전달되는데, 전달 방식에 따라 결과가 달라짐.  
c에서는 값에 의한 호출 방법을 사용함.

## 1. 값에 의한 호출

인자의 값이 매개변수를 초기화하는 데에 사용되는 방식.  
매개변수에 저장된 값이 달라져도 인자는 영향을 받지 않음.  
c에서는 값에 의한 호출 방법을 사용함.

## 2. 참조에 의한 호출

인자가 매개변수로 직접 전달되는 방식.  
인자 자체가 매개변수가 됨.  
매개변수에 저장된 값이 달라지면 인자의 값 또한 바뀜.  
c에서는 해당 방법을 쓰지 않지만 포인터로 주소를 다루면 비슷한 기능을 사용할 수 있음.

# 4. c 의 여러가지 함수들

## 1. 수학 함수

c 는 매우 다양한 수학 함수들을 제공함.

해당 라이브러리는 `<math.h>` 헤더파일로 사용할 수 있음.

### 1) 수학 함수의 형

형에 따라 버전이 구분되어 있음. 버전마다 함수와 매개변수의 형이 다름.

float 형, double 형, long double 형 버전이 각각 있음.

double 형 버전이 기본형임.

float 형에는 함수 이름에 f 접미사가 붙음.

long double 형에는 함수 이름에 l (소문자 엘) 접미사가 붙음.

**메모 포함[이160]:** 함수 호출 시에 어떤 형을 가진 수식을 인자로 넣을 것인지, 어떤 형으로 리턴받을 것인지를 고려하여 버전을 선택해야 함.

### 2) 복소수를 위한 수학 함수

복소수를 위한 수학 함수는 `<complex.h>` 헤더파일로 사용할 수 있음.

복소수 수학 함수 이름은 수학 함수 이름에 c 접두사를 추가로 붙임.

### + 통합 자료형 수학 매크로

수학 함수의 기능을 형에 상관없이 하나의 통일된 기능으로 사용할 수 있게 해 주는 매크로.

매번 자료형을 고려하여 수학 함수를 사용하는 것은 번거롭기 때문에 만들어짐.

컴파일러가 인자의 자료형을 파악하여 적절한 수학 함수를 호출해줌.

매크로 이름은 수학 함수에서 기본형인 double 형 버전의 것과 동일함.

`<tgmath.h>` (Type inteGrated math.h) 헤더파일을 명시해 사용할 수 있음.

**메모 포함[이161]:** complex 버전도 통합한 것?

## 2. 재귀 함수

직간접적으로 자기 자신을 호출하는 함수.

재귀적으로 정의되는 문제를 간단하게 다룰 수 있음.

호출 시, 원래 속해 있던 함수의 맨 처음으로 제어가 넘어가는 것이 아니라 새로운 함수의 계산이 시작됨.

이 새로운 함수가 끝나면 호출된 곳으로 값을 리턴함.

모든 재귀 함수는 반복문으로 대체할 수 있음

**메모 포함[이162]:** 재귀 함수는 편리하고 반복문은 효율적인 경우가 많다. 즉, 상황에 맞게 적절히 둘 중 하나를 사용해야 한다.

### 1) 주의점

무한 루프에 빠지지 않도록 해야 함.

많은 함수 호출이 발생하기 때문에 많은 메모리를 사용하고 실행시간이 길어질 수 있음. (비효율적임)

---

#### + 논리적으로 무한히 진행되는 프로그램

프로그램이 논리적으로는 끝나지 않아도 해당 프로그램에 할당된 자원이 고갈되면 종료됨.

#### + 효율 vs 편리

효율만을 고려한 코드는 유지보수나 공동작업 등에서 불편할 수 있음.

편리함을 고려한 코드는 비효율적일 수 있음.

둘 사이의 관계를 적절히 조절하여 코드를 작성해야 함.

### 3. 시간 함수

c 는 시간과 관련된 함수와 **형**을 제공함.

<time.h> 헤더파일로 사용 가능.

다음과 같이 선언되어 있음.

```
clock_t clock(void);
```

```
time_t time(time_t *p);
```

```
double difftime(time_t time1, time_t time2);
```

#### 1) clock()

프로그램이 현재까지 사용한 cpu 클럭 시간을 리턴하는 함수.

**CLOCKS\_PER\_SEC : 1 초가 몇 클럭인지 정의된 매크로.**

이 매크로의 값으로 cpu 클럭 시간을 나누면 초 단위로 확인 가능.

clock() 함수의 리턴값은 clock\_t 형임.

#### 2) time()

1970 년 1 월 1 일부터 경과된 시간을 초 단위로 리턴하는 함수.

리턴값을 변수에 저장할 때에는 인자에 NULL 을 작성해야 함.

인자에 변수를 작성하면 해당 변수에 리턴값이 저장됨. 이때 인자로 들어가는 변수 앞에는 &를 써야함.

time() 함수의 리턴값은 time\_t 형임.

#### 3) difftime()

time 함수가 리턴한 두 개의 값을 입력 받아서 둘의 시간차를 초 단위로 리턴함.

나중 값을 앞 인자에 작성해야 함.

#### 4) clock() vs. time()

clock()과 time()은 유사하지만 결과가 약간 다르므로 상황에 맞춰 사용해야 함.

해당 함수를 특정 코드 앞뒤로 작성하여 값을 얻은 후 둘의 시간차를 계산하면 해당 코드의 실행시간을 측정할 수 있음.

clock()은 스레드 프로그램 시간 측정 시, 프로그램을 구성하는 모든 스레드가 사용한 cpu 클럭 시간의 합을 알려주기 때문에, 스레드 프로그램 측정 시에는 원한 값보다 더 큰 값이 나온다는 단점이 있음.

time()은 운영체제의 다른 프로그램들이 실행된 시간까지 포함하여 측정한다는 단점이 있음.

**메모 포함[이163]:** clock\_t, time\_t 등이 있음.

**메모 포함[이164]:** clock\_t 형과 time\_t 형은 long 형과 호환되는 것 같다.

visual studio에서는 time\_t가 long long과 크기가 같다.

**메모 포함[이165]:** CLOCKS\_PER\_SEC 값이 정수이기 때문에 값을 실수로 구하려면 캐스트 연산자를 사용해야 한다.

**메모 포함[이166]:** 대부분의 운영체제에서는 여러 개의 프로그램들이 동시에 실행된다.

## 9. 변수

# 1. 지역변수와 전역변수

## 1. 유효범위 규칙

식별자(변수)가 사용되는 범위에 관한 규칙.

식별자는 선언된 블록(복합문) 안에서만 유효함.

외부 블록에 선언된 변수는 내부 블록에서 다시 선언되지 않았을 경우 내부 블록에서도 유효함.

외부 블록에 선언된 변수가 내부 블록에서 다시 선언된 경우 해당 내부 블록에서는 외부 블록에서 선언된 변수가 유효하지 않음. (내부에서 다시 선언된 변수만이 유효함)

메모 포함[이167]: 이 블록은 함수도 포함하는 것?  
for문에서는 ()안에 있는 것까지 한 블록으로 취급함?

## 2. 지역변수

함수 내에서 선언된 변수

해당 함수가 호출될 때에만 사용 가능함.

다른 함수의 지역변수는 사용할 수 없음.

### 3. 전역변수 (외부 변수)

함수 외부에서 선언된 변수

전역변수는 프로그램이 종료될 때까지 메모리에 남기 때문에, 모든 함수에서 사용이 가능함.

자동으로 0으로 초기화됨.

전역변수는 스토리지 클래스를 가질 수 없음. (전부 extern 임)

#### 1) 유용성

하나의 함수는 하나의 값만 리턴할 수 있는데, 전역변수를 이용하면 한 번에 여러 개의 값을 보낼 수 있음.

#### 2) 사용 시 주의점

전역변수 사용 시 모든 함수에서 적용되므로 의미적 오류가 발생, 가독성 저하, 함수의 모듈성 파괴 등 부작용들이 존재하기 때문에 전역변수는 많이 사용하지 않는 것이 좋음.

(대신 포인터를 사용하는 것이 좋음)

## 2. 스토리지 클래스

### 1. 스토리지 클래스

<스토리지 클래스> <자료형> <식별자>;

**메모 포함[이168]:** storage class.  
즉 저장의 방법의 종류를 의미한다.

변수가 메모리를 할당 받는 위치와 사용 가능한 범위를 결정함

해당 변수의 선언문에 명시함.

한 번에 여러 개의 변수를 다루는 선언문에서도 한 번에 적용 가능함.

변수 선언 시에 아무것도 명시하지 않으면 해당 변수가 선언된 위치에 의해 스토리지 클래스가 결정됨.

auto, register, extern, static, \_Thread\_local 등의 스토리지 클래스가 있음.

### 2. auto

블록이나 함수에 들어갈 때 메모리를 할당받고, 빠져나갈 때 메모리를 반납하는 스토리지 클래스.

지역변수에만 사용 가능

auto 변수는 자동으로 초기화되지 않음. (초기화하지 않은 경우 임의의 값을 가짐)

함수 몸체 내 또는 블록에서 선언된 변수는 default 로 auto 스토리지 클래스를 가짐.

default 값으로 적용되므로 선언문에서 보통 생략함.

**메모 포함[이169]:** 함수를 두 번 호출하여 사용할 때에도  
두 번째 사용하는 함수 지역변수는  
첫 번째의 값을 가지고 있지 않다.

### 3. register

메모리보다 접근속도가 빠른 register 에 변수를 저장하는 스토리지 클래스

지역변수에만 사용 가능.

register 변수는 자동으로 초기화되지 않음.

register 변수는 주소 연산자를 사용할 수 없음. (& 못씀)  
(register 스토리지 클래스를 지원하지 않는 환경에서도 register 로 선언 시 사용 불가)

#### 1) 사용

빠른 접근이 필요한 변수에 사용.

가장 자주 사용되는 변수에 사용.

for 문에서 선언한 변수에 사용

해당 변수의 사용 직전에 register 로 선언하는 것이 좋음.

메모 포함[이170]: 포인터 배열 때 알려주신다고 하심.

메모 포함[이171]: 주로 횟수를 나타내는 변수인데, 이 때 이 변수는 for문 내에서 가장 자주 사용되는 것이 일반적이므로 register로 정의하면 효과적이다.

#### 2) register 가 할당되지 않는 경우

register 변수여도 실제로는 register 에 배정되지 않을 수 있음.

컴파일러나 하드웨어에 따라 register 스토리지 클래스 기능 지원 여부에 차이가 있음.  
지원하더라도 register 는 매우 제한적이기 때문에 모든 register 변수가 배정될 수 없음.

register 에 할당되지 않은 register 변수는 default 로 auto 가 됨.

### 4. static

지역변수와 전역변수에 지정하는 스토리지 클래스.

#### 1) 전역변수에 지정

정적 외부변수. 이것에 대한 설명은 프로그래밍응용및실습의 필기에서 다름,

#### 2) 지역변수에 지정

지역변수에 static 을 지정하면 정적 변수가 되어 프로그램이 끝날 때까지 해당 변수의 메모리 공간이 반납되지 않음. (값을 그대로 가지고 있음)

해당 지역변수가 메모리를 반납하지 않고 값을 가지고 있어도 여전히 자신이 속한 곳에서만 사용 가능함.

static 을 지정한 지역변수(정적변수)는 0 으로 자동 초기화됨.

메모 포함[이172]: 일반적인 지역변수는 auto 변수이기 때문에 포함된 함수나 블록이 종료되면 메모리 공간이 반납된다.



## + 변수의 선언과 정의

변수 선언: 컴파일러에게 변수의 이름과 형을 알려주는 것.

변수 정의: 컴파일러에게 변수에 메모리를 할당하게 하는 것.

한 프로그램에서 변수 선언은 여러 번 해도 되지만 변수 정의는 한 번만 해야 함.

일반적인 선언문에서는 변수 선언과 변수 정의가 한꺼번에 일어남.

## 5. extern

전역변수를 정의하거나 다른 파일의 전역변수를 참조하기 위한 스토리지 클래스.

함수 밖에서 선언된 변수(전역변수)의 스토리지 클래스.

extern 변수는(전역변수는) 0 으로 자동 초기화됨.

### 1) 전역변수 정의

전역변수의 선언문에서 extern 을 생략하면 전역변수를 정의하는 것.

전역변수의 선언문에서 extern 을 명시하고 해당 전역변수를 초기화하면 전역변수를 정의하는 것.

(ex. int test; 또는 extern int test1 = 0, test2 = 0; )

### 2) 전역변수 선언

전역변수의 선언문에서 extern 을 명시하고 해당 전역변수를 초기화하지 않으면 전역변수를 선언하는 것.

(ex. extern test; )

여기서 선언한다는 것은 한 프로그램에 여러 파일이 존재하는 경우 다른 파일에서 정의된 전역변수를 참조하여 사용한다는 것임.

하나의 파일 내에서도 전역변수를 참조할 수 있음.

전역변수의 선언은 해당 변수를 사용할 블록 안에서 함.

함수 안이든 밖이든 상관없음.

## 6. \_Thread\_local

thread 프로그래밍 시에 사용함.

**메모 포함[이173]:** 변수 선언은 여러 번 해도 되기 때문에

하나의 파일에서도 여러 번 전역변수를 선언할 수 있다.

(ex. extern int test;  
extern int test;  
이렇게 두 번 적어도 문제없다.)

**메모 포함[이174]:** 예를 들어,

```
int main (void)
{
    extern int test;
    return 0;
}
int test;
```

이런 식으로 작성할 수 있다.

# 10. 문자열

## 1. 문자열

### 1. 문자열

char 형의 1 차원 배열.

문자열은 배열이기 때문에 문자열 상수 뒤에 색인을 명시할 수 있음.

마지막 원소에는 NULL 문자(w0)가 들어감.

배열의 크기는 NULL 문자까지 포함한 문자의 개수임.

**메모 포함[이175]:** c 에서 char 형의 1 차원 배열은 다른 1 차원 배열들과는 다르게 취급됨.

**메모 포함[이176]:** ex.  
"test"[3]는 t를 의미한다.

### 2. 문자열 상수

큰따옴표 안에 작성함.

문자열 상수를 다룰 때에는 배열을 사용함.

---

#### + NULL 문자

w0 으로 나타냄.

모든 비트가 0 인 바이트.

NULL 문자의 십진수 값은 0 임.

### 3. 문자열 사용법

#### 1) 문자열 상수로 배열 초기화하기

가. 일반 배열처럼 선언문에서 초기화 목록을 작성하여 초기화할 수 있음.  
초기화자에 문자열의 문자 각각을 문자 상수로 명시해주면 됨.

나. 문자열 상수를 통째로 명시해서 초기화할 수도 있음.

일반 배열처럼, 초기화 시에 초기화자를 명시하면 배열 크기를 생략할 수 있음.  
이때 생략된 배열 크기는 NULL 문자까지 포함한 문자의 개수인 것 유의.

```
<자료형> <식별자>[개수] ="<string>";
```

**메모 포함[이177]:** 일반적인 배열과 같이, 이런 방식의 초기화는 선언문에서만 사용이 가능하다.

선언문이 아닌 문장에서 값을 대입하려면 각 원소별로 따로 대입하거나, strcpy() 또는 sprintf() 등을 사용해야 한다.

#### 2) 문자열 상수 출력하기

배열에 저장된 문자열 상수를 출력할 때는 반복문 또는 printf를 사용함.

반복문 안에 조건문을 넣어서 NULL 문자가 나올 때까지 반복해서 출력하는 방법을 사용할 수 있음.

printf에서 변환명세 %s로 받아서 출력할 수 있음.

printf는 배열을 명시하면 NULL 문자 전 문자까지 출력함.

이때 인자에는 배열의 이름만을 명시함.

(ex. printf("%s", test);)

#### 3) 문자열 상수 입력받기

scanf로 문자열 상수를 입력 받을 수 있음.

변환명세는 %s 또는 %[ ]를 사용함.

%s인 경우 공백문자 또는 개행문자 직전까지만 입력받음.

[ ]인 경우 안에 명시한 문자가 오기 전 까지만 입력받음. 공백문자까지 입력받을 수 있음. ^ 뒤에 문자를 작성하는 형태임. (ex. %[^\n])

배열을 인자에 작성할 때는 이름만을 명시하고, 이름 앞에 &를 쓰지 않음.

배열 크기보다 더 큰 크기의 문자열 상수가 입력되면 오류 발생함. 결과가 도출되지 않음.

**메모 포함[이178]:** 입력 받은 문자열 맨 마지막에는 자동으로 NULL 문자가 들어간다.

### 4. 문자열의 비교

두 문자열은 등가 연산자(==, !=)나 관계 연산자(>, <, <=, >=)로 비교할 수 없음.

등가, 관계 연산자로 비교하는 경우 포인트하는 문자열이 아니라 두 문자열의 주소를 비교하는 것임.

두 문자열을 비교할 때는 원소 각각을 비교하거나 strcmp()를 이용해야 함.

**메모 포함[이179]:** 반복문으로 비교를 진행하고, w0이 나올 때까지 원소끼리 같은지를 확인한다.

## 2. 문자열 처리 함수

c에서는 문자열을 다루는 함수들을 제공함. <string.h> 헤더파일을 명시하여 사용할 수 있음.

### 1. strcpy() (string copy) (문자열 복사)

선언문이 아닌 문장에서 배열에 문자열을 저장하는 함수.

#### 1) 사용법

식별자에는 해당 문자열을 저장할 배열 이름을 명시함.

문자열 대신 배열 이름을 작성해도 됨.

(ex. strcpy(test1, test2); , strcpy(test, "Hello"); )

```
strcpy(<식별자>, <문자열>);
```

**메모 포함[이180]:** strcpy() 나 strcat() 함수 모두 두 번째 인자의 값을 첫 번째 인자의 값에 사용한다.

#### 2) 주의점

배열의 크기가 문자열의 크기보다 크거나 같아야 함.

#### + sprintf()

printf()와 유사하지만, 화면으로 출력하지 않고 명시한 배열에 출력(저장)하는 함수.

<stdio.h> 헤더파일을 명시하여 사용 가능함.

첫 번째 인자에는 문자열을 저장할 배열 이름을 명시함.

두 번째 인자부터는 printf()와 동일한 형태로 작성함.

(ex. sprintf(test, "Hello"); , sprintf(test, "%s", "Hello"); , sprintf(test1, "%s", test2); )

```
sprintf(<식별자> , <제어문자열>, ...);
```

배열이 가진 메모리의 크기가 저장되는 정보들의 크기보다 크거나 같아야 함

문자열 저장, 복사가 가능한 함수. strcpy()와 동일한 기능을 할 수 있음.

문자열 연결이 가능한 함수. strcat()과 동일한 기능을 할 수 있음 (두 문자열을 출력하면 됨)

## 2. strlen() (string length)

```
strlen(<string>);
```

₩0 을 뺀 문자의 개수를 리턴하는 함수.

문자열을 다루기 전에 문자열의 길이를 확인하여 오류가 발생하지 않음을 검증함.

strlen()과 sizeof 를 사용해서 검증 가능.

strlen 은 ₩0을 제외한 개수를 리턴하기 때문에 검증 시에는 1 을 더해줘야 함.

문자열 대신 배열 이름을 작성해도 됨.

**메모 포함[이181]:** ex. test배열에 "Hello"를 저장할 수 있는지 확인하는 코드.

```
if(sizeof(test) >= (strlen("Hello") + 1))  
{  
    strcpy(test, "Hello");  
}
```

## 3. strcmp() (string compare)

```
strcmp(<string1>, <string2>);
```

두 문자열을 사전순으로 비교해서 값을 리턴하는 함수.

앞의 문자열이 사전순으로 나중이면 양수를 리턴함.

앞의 문자열이 사전순으로 먼저이면 음수를 리턴함.

두 문자열이 같으면 0 을 리턴함.

특히 두 문자열이 같은지 확인할 때 사용함.

문자열 대신 배열 이름을 작성해도 됨.

**메모 포함[이182]:** 사전순으로 나중일수록 번호가 크고, strcmp()는 앞 - 뒤인 것으로 암기

## 4. strcat()

```
strcat(<string1>, <string2>);
```

첫 번째 인자의 문자열에 두 번째 인자의 문자열을 복사하여 붙이는 함수.

첫 번째 문자열은 두 문자열을 이은 것으로 바뀌지만, 두 번째 문자열은 그대로임.

문자열 대신 배열 이름을 작성해도 됨.

**메모 포함[이183]:** 이때 첫 번째 문자열에 있던 NULL 문자는 지우고 NULL문자까지 포함한 두 번째 문자열 전체를 복사한다.

### 1) 주의점

첫 번째 문자열에는 두 번째 문자열까지 저장해야 하므로 충분한 공간을 가지고 있어야 함.

공간이 충분하지 않다면 다른 배열의 공간을 침범해서 저장하는 등 문제가 발생함.

(ex. 크기가 각각 16 인 배열 s1 과 s2 를 가지고 strcat(s2, s1); 을 수행한 결과.

s1 과 s2 는 바로 붙어 있는 메모리에 저장되는데, s2 가 s1 의 메모리를 침범해 저장하게 됨.

```
s1 : 123456789012345  
s2 : abcdefghijkl  
s1 after strcat : 56789012345  
s2 after strcat : abcdefghijkl123456789012345 )
```

## 5. 문자열 -> 숫자 변환 함수

c 에서 제공하는 <stdlib.h>를 명시하여 사용할 수 있음.

```
ato...(<string>);  
strto...(<string>, NULL, <진법>);
```

문자열을 숫자로 바꾼다는 것은 문자를 문자에 해당하는 아스키 값으로 바꾼다는 의미가 아니라, 문자열과 동일한 형태의 숫자로 바꾼다는 의미임.

ato...() 부류 함수: 문자열을 10 진수로 변환.

strto...() 부류 함수: 변환 진수를 지정(정수인 경우)하여 변환. 실수인 경우 진법이 들어가는 인자는 생략.

변환은 문자열 맨 처음부터 변환이 안 되는 문자가 나올 때까지 진행함.

**메모 포함[이184]:** ex. value = atoi("123.123");  
여기서 value에는 123이 저장됨.

**메모 포함[이185]:** 소수점으로 사용되는 온점(.)이나 해당 진법의 숫자가 아닌 문자.

ex. 102ab.31 을 11진수로 변환할 때는, 102a까지만 변환된다.

함수	변환 형
atof()	double
atoi()	int
atol()	long
atoll()	long long

함수	변환 형
strtod()	double
strtof()	float
strtold()	long double
strtol()	long
strtoll()	long long
strtoul()	unsigned long
strtoull()	unsigned long long

### + Stack overflow 공격

strcpy()와 strcat()의 취약점(공간이 부족할 때 메모리를 침범하는 것 등)을 이용하는 해킹.

이런 해킹이 존재하기 때문에 최신 컴파일러들은 공간이 부족할 때 프로그램이 종료되게 하기도 함.

strncpy(), strncat(), strlcpy(), strlcat() 함수들은 이런 취약점의 대안임.

문자열의 크기를 인자로 지정함.

다루는 배열의 크기가 부족한 경우,

n 은 지정된 수만큼의 문자열만을 다룸. NULL 문자가 들어갈 자리를 확보해 줘야 함.

l 은 지정된 수보다 1 작은 문자열의 개수만을 다루고 끝에 NULL 문자를 붙임.

n 은 맨 뒤 인자에 정수 수식을 명시해 주면 됨. strncat에서는 복사될 문자의 크기를 명시함.

l 은 맨 뒤 인자에 정수 수식을 명시해 주면 됨. strlcat에서는 첫 번째 인자의 문자열 크기를 명시함.

l 은 n 과는 다르게 표준 라이브러리에서 지원하지 않으므로 컴파일러가 지원하는지 확인해야 함.

**메모 포함[이186]:** ex. strncat(A, B, 3); 이면 B에서 3글자만 복사해서 A에 붙여넣는다는 의미이다.

**메모 포함[이187]:** ex. strlcat(A, B, 3); 이면 A에 총 3글자만 넣는다는 의미이다.

# 부록 1. 파일 입출력

c 언어에서 파일을 다루는 방법은 두 가지임.  
하나는 리눅스에서 리다이렉션하는 것.  
다른 하나는 프로그램에서 특정 함수를 사용하는 것.

## 1. 리다이렉션

### 1. 스트림

다양한 입출력 장치를 일관성 있게 접근하게 해 주는 기능.  
c 언어에서 자료를 입출력하기 위해 사용하는 것.  
입출력 장치와 프로그램 사이에서 입출력 자료들을 중계하는 역할을 함.

### 2. 표준 입출력 장치 (기본값)

표준 입력 장치 (0): 셸이 작업할 때 필요한 정보를 받아들이는 장치 -> 키보드  
표준 출력 장치 (1): 실행 결과를 내보내는 장치 -> 모니터  
표준 오류 장치 (2): 오류 메시지를 내보내는 장치 -> 모니터

표준 입출력 장치를 일시적으로 파일로 바꾸는 것을 '리다이렉션'이라고 함. (redirection)  
c에서는 파일 포인터 변수 자리에 표준 입출력 장치를 stdin, stdout, stderr 등으로 명시하여 사용할 수 있음.

메모 포함[이188]: 셸이 출력하는 것은 두 가지가 있다.  
하나의 실행 결과이고  
다른 하나는 오류 메시지이다.

#### + 파일 디스크립터 (descriptor)

파일 관리를 위해 붙이는 번호.  
리눅스에서는 장치도 파일로 관리하기 때문에 표준 입출력 장치에 번호가 붙어있음.

표 4-9 표준 입출력 장치의 파일 디스크립터

파일 디스크립터	파일 디스크립터 대신 사용하는 이름	정의
0	stdin	명령의 표준 입력
1	stdout	명령의 표준 출력
2	stderr	명령의 표준 오류

(standard input, output, error)

### 3. 입출력 재지정 (리다이렉션) (리눅스 명령어)

표준 입출력을 재지정하는 것을 리다이렉션이라고 함.

printf 로 출력 시에 출력이 재지정되어 있으면 지정된 파일로 내용을 출력함.

scanf 로 입력 받을 때에 입력이 재지정되어 있으면 지정된 파일에서 내용을 입력 받음.

#### 1) > (출력 리다이렉션 방법 1)

출력되는 정보의 방향을 해당 파일로 리다이렉션함.

```
<command> <number>> <file>
```

정보를 저장할 파일이 이미 존재하면 덮어쓰기함. (원래의 정보를 지우고 해당 정보를 저장함)

지정한 이름의 파일이 없으면 해당 파일을 생성해서 수행함.

<number>에는 표준 입출력 장치의 파일 디스크립터를 명시함.

특정 번호를 명시하면 일시적으로 오른쪽에 오는 파일을 해당 번호로 취급하겠다는 의미.

아무것도 명시하지 않은 경우, 1 로 취급됨.

출력된 실행 결과를 리다이렉션하려면 1 번을, 출력된 오류 메시지를 리다이렉션하려면 2 를 작성함.

연속해서 리다이렉션할 수 있음.

메모 포함[이189]: 표준 입출력 장치를 일시적으로 파일로 바꾸는 것.

#### 2) >> (출력 리다이렉션 방법 2)

출력되는 정보의 방향을 해당 파일로 리다이렉션함.

출력되는 문자열을 해당 파일에 추가함.

>와는 달리 삭제하고 넣는 게 아니라 그냥 넣음.

지정한 이름의 파일이 없으면 해당 파일을 생성해서 수행함.

```
<command> <number>>> <file>
```

#### 3) < (입력 리다이렉션 특수기호)

입력되는 정보를 해당 파일에서 가져오도록 리다이렉션함.

해당 file 의 데이터를 명령에 입력함.

<는 0<를 생략하여 표현한 형태임.

```
<command> <number>< <file>
```

#### + 입출력을 한 번에 리다이렉션하기.

test < input\_file > output\_file 등으로 작성.



## 2. c 의 파일 입출력 함수

### 1. 파일 열기, 닫기 함수

파일을 다루려면 우선 파일을 열어야 함.

사용이 끝난 파일은 닫아야 함.

#### 1) fopen() : 파일 열기

1. 파일을 읽기 위해서는 우선 파일 포인터 변수를 선언해야 함.

```
FILE *<식별자>;
```

반드시 \*(별표)를 작성해줘야 함.

(ex. FILE \*A, \*B;)

2. 해당 변수에 fopen 함수를 배정함.

```
<변수> = fopen("<파일명>", "<모드>");
```

모드들. 적절한 모드를 선택해 여러 개 명시할 수 있음.

r : 읽기를 위해 열기

w : 쓰기를 위해 열기

x : 베타적 파일 접근

a : 첨부하기 위해 열기

b : 이진 파일 열기

+ : 읽기와 쓰기를 위해 열기

#### 2) fclose() : 파일 닫기

해당 함수의 인자에 사용한 파일 포인터 변수를 넣어주면 됨.

```
fclose(<변수>);
```

## 2. 파일 쓰기 함수

### 1) fprintf() : 파일에 쓰기

```
fprintf(<변수>,"<제어문자열>", <수식>);
```

맨 앞 인자에 파일 포인터 변수를 작성하고, 나머지는 printf 와 동일한 형식임.

## 3. 파일 읽기 함수

### 1) fscanf() : 파일 읽기

```
fscanf(<변수>,"<변환명세>", &<변수>);
```

맨 앞 인자에 파일 포인터 변수를 작성하고, 나머지는 scanf 와 동일한 형식임.

# 3. c 의 입출력 함수들

## 1. printf(), scanf()

c 언어 기초에 설명이 있으므로 넘어감.

## 2. getchar(), putchar()

### 1) getchar()

표준 입력 장치로 문자를 하나 입력 받아서 그것을 리턴하는 함수.

getchar 함수를 호출할 때에는 인자를 작성하지 않음.

읽을 문자가 없으면 EOF(셋 다 영어)를 리턴 (End Of File)

### 2) putchar()

표준 출력 장치로 문자를 하나 출력하는 함수.

## 3. getc(), putc()

파일로부터 한 문자 씩 입력 받을 때에는 getc 나 putc 를 사용함.

파일 포인터를 갖는다는 것을 제외하면 getchar(), putchar()와 동일한 기능을 함.

(getchar() <-> getc(stdin), putchar(c) <-> putc(c, stdout))

### 1) getc()

파일 포인터 변수까지 명시함.

```
getc(<변수>);
```

### 2) putc()

첫 번째 인자에는 출력할 문자를 명시함.

두 번째 인자에는 파일 포인터 변수를 명시함.

```
putc(<변수>, <변수>);
```

### + 파일 디스크립터로 명시

표준 입출력 장치를 대상으로 하려면 파일 포인터 변수 대신에 stdin, stdout, stderr 등으로 명시할 수 있음.

메모 포함[이190]: scanf() 등을 사용해서 파일을 읽을 때도 EOF가 뜰 수 있다. (리다이렉션 하는 경우)