컴퓨터프로그래밍I

프로그래밍 개념, C 언어, 컴파일러, 입출력과 제어문

김지수T

2026학년도

경기과학고등학교

Wi-Fi 정보

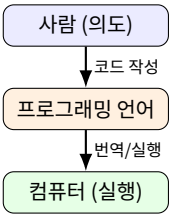
- SSID: U+Net2491_5G
- Password: 19917182M*

Table of Contents

프로그래밍의 개념

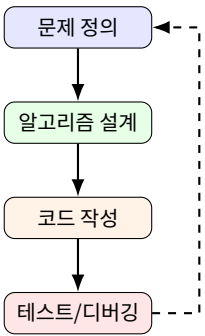
프로그래밍이란?

- 프로그래밍: 컴퓨터에게 할 일을 **명령어**로 적어주는 것
- 프로그램: 그 명령어를 모아 놓은 결과물
- 프로그래밍 언어: 사람과 컴퓨터 사이의 **약속된 문법**
- 핵심: 문제를 쪼개고, 풀이 방법을 설계하고, 코드로 옮기는 것



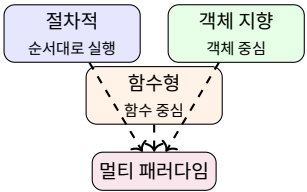
알고리즘과 프로그램

- 알고리즘: 문제를 풀기 위한 **단계별 레시피**
- 프로그램: 알고리즘을 특정 언어로 **구현**한 것
- 같은 알고리즘도 언어가 다르면 표현이 달라진다
- 좋은 알고리즘이란?
  - 입력과 출력이 명확하다
  - 각 단계를 실제로 수행할 수 있다
  - 유한 시간 안에 끝난다



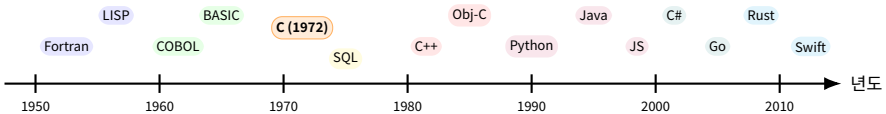
프로그래밍 패러다임

- **절차적** (Procedural): 명령을 순서대로 나열 (C, Pascal)
- **객체 지향** (OOP): 데이터와 동작을 객체로 묶음 (C++, Java, Python)
- **함수형** (Functional): 순수 함수와 불변 데이터 중심 (Haskell, OCaml)
- 요즘 언어들은 여러 패러다임을 섞어서 쓴다
- C는 대표적인 절차적 언어



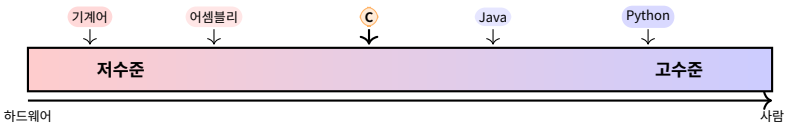
프로그래밍 언어의 역사

프로그래밍 언어의 흐름



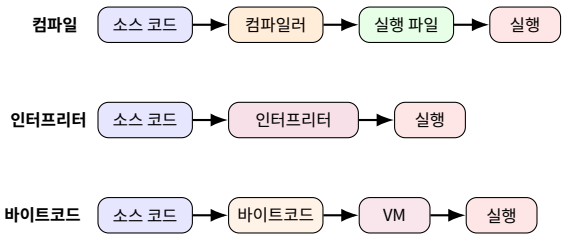
- 1950s: 기계어/어셈블리 → Fortran, LISP
- 1970s: **C** (1972, 시스템 프로그래밍), Pascal, SQL
- 1990s: Python, Java, JavaScript
- 2010s: Rust (메모리 안전), Swift (iOS), TypeScript

언어 수준: 저수준 vs 고수준



- **저수준**: 하드웨어 직접 제어, 최대 성능, 이식성 낮음
- **고수준**: 가독성, 이식성, 빠른 개발, 세밀 제어 어려움
- C는 고수준이지만 **저수준 조작**(포인터, 비트 연산)이 가능하여 “중간 수준 언어”로 불리기도 함

실행 방식에 따른 분류



- **컴파일**: C, C++, Rust, Go — 실행 속도 빠름
- **인터프리터**: Python, JavaScript — 개발 빠름, 실행 느림
- **바이트코드**: Java, Kotlin, C# — 플랫폼 독립성

여담: 프로그래밍 언어는 도구일 뿐

- 프로그래밍 언어는 도구일 뿐이다
- 문제에 맞는 언어를 고르는 게 중요
  - 시스템/임베디드 → C, Rust
  - 웹 서버 → Go, Java, Python
  - 데이터 분석/AI → Python
  - 모바일 앱 → Swift (iOS), Kotlin (Android)
- 하나를 제대로 익히면 다른 언어는 금방 배운다
- C를 배우면 컴퓨터가 실제로 어떻게 돌아가는지 감이 온다

C 프로그래밍

C 언어란?

- 1970년대에 등장한 범용 절차형(Procedural) 프로그래밍 언어
- 하드웨어에 가까운 low-level 조작이 가능해서 OS, 시스템 소프트웨어에 많이 쓰인다
- 컴파일 언어라서 실행이 빠르고 메모리를 아낄 수 있다 (Python보다 수십 배 빠름)
- C 문법이 C++, Java, Go 등 현대 언어의 뿌리

C의 역사

- 1972년 Bell Labs의 Dennis Ritchie가 개발
- B 언어에서 발전하여 UNIX 운영체제 구현에 사용
- 표준화: ANSI C(C89/C90) $\rightarrow$ C99 $\rightarrow$ C11 $\rightarrow$ C17/C23
- 표준 덕분에 플랫폼이 달라도 같은 코드를 돌릴 수 있다
- UNIX, Linux 커널, Git, Python 인터프리터(CPython), SQLite 등이 C로 작성됨

C의 특징

- **정적 타입**: 컴파일 타임에 타입 검사
- 포인터로 메모리를 직접 건드릴 수 있다
- 전처리기와 헤더 파일로 코드를 나눠서 관리
- 런타임이 작고 실행 모델이 단순하다

C의 장점과 단점

장점

- 빠르다: 하드웨어 가까이에서 최적화 가능
- 어디서든 돌린다: 대부분의 플랫폼에 컴파일러가 있다
- 마음대로 제어: 메모리, 하드웨어를 직접 다룰 수 있다
- 임베디드, OS, 컴파일러, DB 등에 사용

단점

- 메모리 사고가 잦다: 버퍼 오버플로우, 댕글링 포인터
- 표준 라이브러리가 빈약하다
- 추상화가 약해서 큰 프로젝트 유지보수가 고되다
- 포인터 초기화를 빼먹으면 디버깅 지옥이 열린다

Hello, World로 보는 C 기본 문법

- `#include <stdio.h>`: 표준 입출력 함수(`printf`)를 쓰기 위해 헤더를 포함
- `int main(void)`: 프로그램의 시작점, 실행 시 운영체제가 이 함수부터 호출
- 중괄호 `{ }`: 함수/조건/반복문의 코드 블록을 묶음
- `return 0;`: 프로그램이 정상 종료되었음을 운영체제에 알림

```c
#include <stdio.h>      // 헤더 포함: printf 선언

int main(void) {        // 프로그램 시작점
    printf("Hello, world!\n"); // 한 줄 출력, 문장 끝은 세미콜론
    return 0;           // 정상 종료 코드
}
```

C 기본 문법

- 하나의 문장(statement)은 **세미콜론** ; 으로 끝난다
- 공백/탭/줄바꿈은 대부분 무시되지만, **가독성**을 위해 일관되게 쓰는 것이 중요
- 중괄호 { }는 여러 문장을 하나의 **블록**으로 묶는다
- 주석: 한 줄 //, 여러 줄 /* ... */

```c
int x = 1; int y = 2;  // 같은 줄에 써도 되지만 읽기 어렵다

int sum = x + y;       // 한 문장은 세미콜론으로 끝난다

if (sum >= 3) {        // 블록 시작
    printf("ok\n");
}                      // 블록 끝

/* 공백/탭/줄바꿈은
   대부분 의미가 같지만, 보기 좋게 정렬하는 것이 중요 */
```

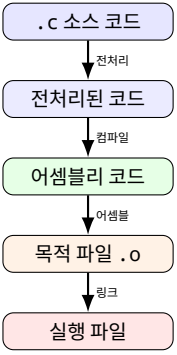
컴파일러와 개발 환경

소스 코드와 실행 파일

- 소스 코드(source code): 사람이 읽고 쓰는 텍스트 파일 (`.c`, `.h`)
- 헤더 파일(`.h`): 함수 선언, 타입 정의, 매크로 등 약속을 모아둔 파일
- 실행 파일(executable): CPU가 바로 돌릴 수 있는 기계어
- 소스 코드를 바로 실행할 수는 없고, **컴파일/링크**를 거쳐야 한다
- 주석이나 공백은 컴파일할 때 전부 사라진다

컴파일과 링킹: 단계별 흐름

- **전처리**(Preprocess): `#include`, `#define` 처리 → 하나의 코드로 합침
- **컴파일**(Compile): C 코드를 어셈블리로 변환
- **어셈블**(Assemble): 어셈블리를 **목적 파일**(`.o`/`.obj`)로 변환
- **링크**(Link): 여러 목적 파일 + 라이브러리를 묶어 **실행 파일** 생성



분리 컴파일과 링킹

```
# 1) 각 소스 파일을 목적 파일로 컴파일
gcc -c main.c   # main.o 생성
gcc -c util.c   # util.o 생성

# 2) 목적 파일들을 링크하여 실행 파일 생성
gcc main.o util.o -o app

# 3) 한 번에 컴파일+링크 (간단한 경우)
gcc main.c util.c -o app
```

- 링크 에러 `undefined reference`는 함수 정의가 빠졌거나 파일을 빼먹은 경우가 많다
- 라이브러리를 사용할 때는 `-l` 옵션으로 링커에 알려준다 (예: `-lm`)

컴파일러의 역할

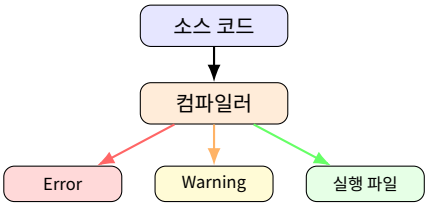
- 컴파일러: C 소스 코드를 기계어(실행 파일)로 변환하는 프로그램
  - **GCC** (GNU Compiler Collection): 리눅스/Windows에서 널리 사용
  - **Clang/LLVM**: macOS 기본, 빠른 컴파일과 좋은 오류 메시지
  - **MSVC**: Windows Visual Studio의 기본 컴파일러
- 컴파일러가 하는 일:
  - **문법 검사**: 문법 오류를 감지하고 오류 메시지 출력
  - **최적화**: 코드의 실행 속도/크기를 개선
  - **코드 생성**: 타겟 플랫폼에 맞는 기계어 생성

개발 도구: IDE와 디버거

- IDE(통합 개발 환경): 코드 작성 + 빌드 + 디버깅 + 프로젝트 관리를 한 번에 수행
- 주요 IDE/에디터:
  - **Visual Studio Code**: 무료, 오픈 소스, 널리 사용됨
  - **Code::Blocks**: 무료 C/C++ IDE (Windows/Linux)
  - **Xcode**: macOS 전용 IDE
  - **CLion**: JetBrains의 유료 C/C++ IDE
- 디버거: 중단점, 한 줄씩 실행, 변수/메모리 상태 확인
  - 디버깅을 통해 코드의 실행을 따라 가며 오류를 찾을 수 있다

컴파일러 경고와 오류

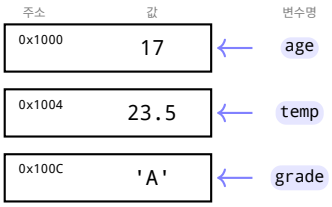
- **오류**(Error): 문법이 잘못되어 컴파일 자체가 실패
  - 세미콜론 누락, 괄호 불일치, 선언되지 않은 변수 등
- **경고**(Warning): 컴파일은 되지만 잠재적 문제
  - 초기화되지 않은 변수, 타입 불일치 등
- `-Wall -Wextra` 옵션으로 경고를 최대한 켜기
- **경고 0개**를 목표로 하는 습관이 중요



변수와 타입

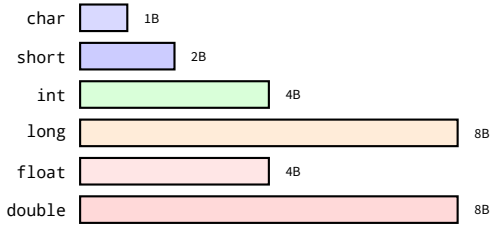
변수란?

- 변수: 값을 담아두는 **이름 붙은 메모리 칸**
- 컴퓨터는 주소로 메모리를 읽고 쓰는데, 변수는 그 주소에 붙인 별명
- 타입이 칸의 크기와 해석 방식을 정한다
- 지역 변수는 **스택**에 놓이고, 함수가 끝나면 사라진다



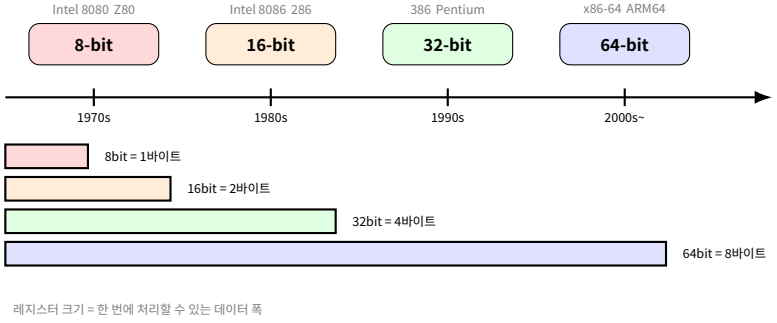
변수 선언과 타입

- 기본 형태: `type name = value;`
- 자주 쓰는 타입: `int`, `long`, `float`, `double`, `char`
- 크기/범위는 플랫폼에 따라 달라질 수 있다




```
1  int age = 17;
2  double temp = 23.5;
3  char grade = 'A';
4  unsigned int count = 10;
```

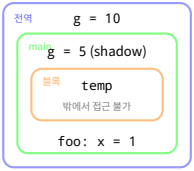
CPU 워드 크기의 변천: 8비트에서 64비트까지



- CPU의 **워드 크기**가 커질수록 한 번에 다루는 데이터가 넓어진다
- 8비트: 0~255, 16비트: 0~65535, 32비트: 약 42억, 64비트: 약 1.8×10^{19}
- `int`가 보통 32비트(4바이트)인 이유: 현대 CPU의 기본 처리 단위

변수 초기화와 스코프

- 초기화 안 한 지역 변수에는 **쓰레기값**이 들어 있다
- 전역 변수는 알아서 0으로 초기화된다
- 이름이 겹치면 **가장 안쪽** 스코프가 이긴다




```c
#include <stdio.h>
int g = 10; // 전역
void foo(void) {
    int x = 1; // 지역
    printf("g=%d, x=%d\n", g, x);
}
int main(void) {
    int g = 5; // 전역 g를 가림
    printf("local g=%d\n", g);
    foo();
}
```

연산자

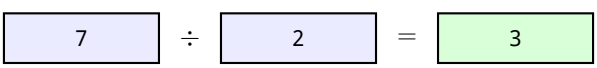
산술 연산자

- 덧셈/뺄셈/곱셈/나눗셈/나머지 연산
- 정수끼리 나누면 **소수점 아래를 버린다** (C99 이후 0 방향 절단)
- /, %에서 0으로 나누면 정의되지 않은 동작(UB)

```c
int a = 7, b = 3;
printf("%d %d\n", a / b, a % b);  // 2 1

int c = -7;
printf("%d %d\n", c / b, c % b);  // -2 -1 (C99+)

double avg = 3 / 2;      // 1.0 (정수 나눗셈)
double avg2 = 3.0 / 2;  // 1.5 (실수 나눗셈)
```

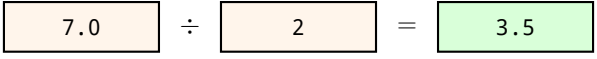
정수 나눗셈 vs 실수 나눗셈

정수 나눗셈: `7 / 2`



.5 버림!

실수 나눗셈: `7.0 / 2`



피연산자 중 하나라도 실수면 실수 나눗셈으로 승격된다

비교 연산자와 논리 연산자

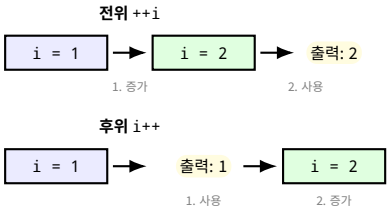
- 비교: ==, !=, <, <=, >, >=
- 논리: && (AND), || (OR), ! (NOT)
- C의 조건식: 0이면 거짓, 0이 아니면 참
- &&와 ||는 **단락 평가**를 수행

A	B	A && B	A || B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

```c
int x = 0;
if (x != 0 && (10 / x) > 1) { // 왼쪽이 거짓이면 오른쪽 계산 안 함
    printf("safe\n");
}
```

대입 연산자와 증감 연산자

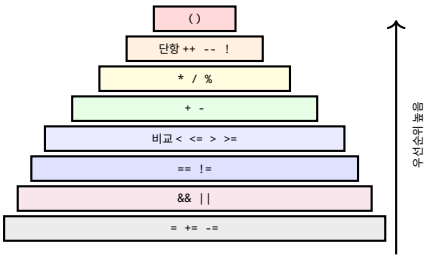
```c
int x = 10;
x += 3;   // 13
x *= 2;   // 26

int i = 1;
printf("%d\n", ++i); // 2
i = 1;
printf("%d\n", i++); // 1
```



- 복합 대입: +=, -=, *=, /=
- 전위: **먼저 변경**, 후위: **사용 후 변경**

연산자 우선순위

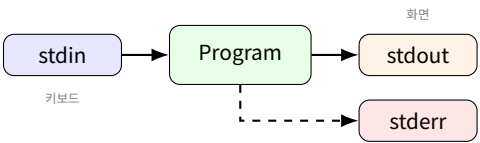
- **높음** → **낮음**:
- ( ) → 단항 → * / %
- → + - → 비교
- → == != → &&
- → || → =
- 대부분 **왼**→**오른쪽**
- 대입/삼항은 **오른**→**왼쪽**
- 헷갈리면 **괄호**로 명확히



표준 입출력 (I/O)

표준 입출력(stdio)

- 표준 입력: 키보드(`stdin`)
- 표준 출력: 화면(`stdout`)
- 표준 에러: 오류 메시지(`stderr`)
- 기본적으로 `stdio.h`의 함수로 처리



printf: 기본 출력

- 기본 형태: `printf("format", 값1, 값2, ...);`
- 포맷 지정자는 %와 함께 사용: 출력할 값의 타입과 형식을 지정
- 문자(char)는 내부적으로 정수로 취급됨 → ASCII 코드 값을 %d로 출력 가능

```c
#include <stdio.h>

int main(void) {
    int age = 17;
    double temp = 23.5;
    char ch = 'A';
    printf("Age = %d\n", age);
    printf("Temp = %.1f\n", temp);
    printf("Char = %c (%d)\n", ch, ch); // A (65)
    return 0;
}
```

printf 포맷 지정자

지정자	의미
%d, %i	int (부호 있는 정수)
%u	unsigned int
%x/%X	16진수 (소문자/대문자)
%f	실수 (double, 고정 소수)
%e/%E	실수 (지수 표기)
%c	문자 (char)
%s	문자열 (char *)
%p	포인터 주소
%%	% 문자 자체 출력

printf: 폭/정렬/채우기

```c
#include <stdio.h>

int main(void) {
    int x = 42;
    double pi = 3.14159;
    printf("[%5d]\n", x);      // 오른쪽 정렬
    printf("[%-5d]\n", x);     // 왼쪽 정렬
    printf("[%05d]\n", x);     // 0으로 채움
    printf("%.2f\n", pi);      // 소수점 2자리
    return 0;
}
```

- 특수 문자: %% (%), \t (탭), \n (줄바꿈), \" (큰따옴표)

scanf: 기본 입력

- 변수의 **주소**를 사용한다: &
- 공백/줄바꿈은 자동으로 건너뜀
- 형식 지정자와 변수 타입이 다르면 UB

```c
#include <stdio.h>

int main(void) {
    int x, y;
    scanf("%d %d", &x, &y);
    printf("%d + %d = %d\n", x, y, x + y);
    return 0;
}
```

scanf: 여러 타입 입력과 주의사항

```c
#include <stdio.h>
int main(void) {
    int age;
    double height;
    char initial;
    scanf("%d %lf", &age, &height);
    scanf(" %c", &initial); // 앞의 공백이 줄바꿈을 건너뜀
    printf("age=%d, height=%.1f, initial=%c\n", age, height, initial);
    return 0;
}
```

- %c는 공백/줄바꿈도 읽으므로 " %c"처럼 앞에 공백을 넣어 건너뜀
- scanf에서 double은 %lf, printf에서는 %f

조건문

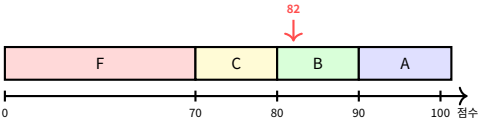
if / else 기본 구조

- 참(true)은 **0이 아닌 값**, 거짓(false)은 **0**이다
- 조건이 참이면 `if` 블록, 거짓이면 `else` 블록 실행
- `else`는 생략 가능

```c
if (score >= 60) {
    printf("Pass\n");
} else {
    printf("Fail\n");
}
```

else if: 구간 분기

```c
int score = 82;
if (score >= 90) {
    printf("A\n");
} else if (score >= 80) {
    printf("B\n");
} else if (score >= 70) {
    printf("C\n");
} else {
    printf("F\n");
}
```



- 위에서부터 순서대로 검사
- **첫 번째로 참인 블록만** 실행

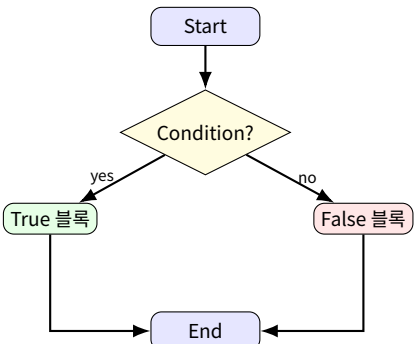
조건문 예제: 윤년 판정

```c
int year = 2024;
if (year % 400 == 0) {         // 400의 배수는 윤년
    printf("leap\n");
} else if (year % 100 == 0) { // 100의 배수는 평년
    printf("common\n");
} else if (year % 4 == 0) {    // 4의 배수는 윤년
    printf("leap\n");
} else {                       // 그 외는 평년
    printf("common\n");
}
```

- 조건 순서가 바뀌면 결과가 달라질 수 있으므로 **우선순위가 중요**하다

조건문의 흐름

- 조건이 참이면 True 경로, 거짓이면 False 경로로 분기
- 분기 이후에는 다시 하나의 흐름으로 합쳐진다
- 중첩 조건은 **가독성**이 떨어지므로 함수로 분리하는 것이 좋다



삼항 연산자와 switch

```c
// 삼항 연산자: 간단한 선택
int a = 3, b = 7;
int max = (a > b) ? a : b;

// switch: 정수/문자 값에 따른 분기
switch (op) {
    case '+': result = a + b; break;
    case '-': result = a - b; break;
    case '*': result = a * b; break;
    default:  printf("Unknown\n"); break;
}
```

- `break`가 없으면 다음 case로 계속 진행됨 (fall-through)
- `case` 라벨은 정수 상수(문자 포함)만 가능

조건문 주의사항

```c
int x = 0;
if (x = 1) {   // 비교(==)가 아니라 대입! -> 항상 참
    printf("always true\n");
}
if (x == 1) {  // 올바른 비교
    printf("x is 1\n");
}
```

- =와 ==을 혼동하지 않도록 주의
- 실수(float/double) 비교는 오차가 있으므로 `fabs(a - b) < 1e-9`처럼 비교
- &&(논리)와 &(비트)를 혼동하지 않도록 주의
- `-Wall` 경고 옵션을 켜면 실수를 잡아줄 수 있다

반복문

for 문: 기본

```c
for (int i = 0; i < 5; i++) {
    printf("%d ", i);
}
// 출력: 0 1 2 3 4
```

- `for(초기화; 조건; 증감)` 형태로 반복 횟수가 명확할 때 사용
- 인덱스는 0부터 시작하는 것이 C의 관례

반복문 예제: 1부터 N까지 합

```c
#include <stdio.h>
int main(void) {
    int n;
    scanf("%d", &n);
    long long sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    printf("%lld\n", sum);
    return 0;
}
```

- `i <= n`이므로 **n까지 포함**하여 더한다 (오프바이원 주의)
- 합이 커질 수 있으므로 `long long`을 쓰면 안전

while 문과 do-while 문

```c
// while: 조건이 참인 동안 반복
int x, sum = 0;
while (scanf("%d", &x) == 1
       && x != 0) {
    sum += x;
}
printf("sum=%d\n", sum);

// do-while: 최소 한 번은 실행
int choice;
do {
    printf("1) Add  2) Exit\n");
    scanf("%d", &choice);
} while (choice != 2);
```

- while: 조건을 **먼저** 검사 → 0회 실행 가능
- do-while: 본문을 **먼저** 실행 → 최소 1회 보장
- 입력 루프에는 scanf 반환값으로 종료 조건을 만들기

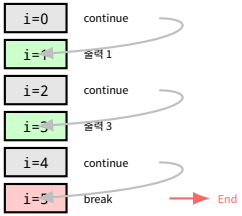
중첩 반복문: 구구단

```c
for (int i = 2; i <= 9; i++) {
    for (int j = 1; j <= 9; j++) {
        printf("%d*%d=%d\n", i, j, i * j);
    }
}
```

- 바깥 반복마다 안쪽 반복이 전체 실행 → **총 실행 횟수는 곱**으로 늘어남
- `break`/`continue`는 **가장 안쪽 루프**에만 적용

break / continue

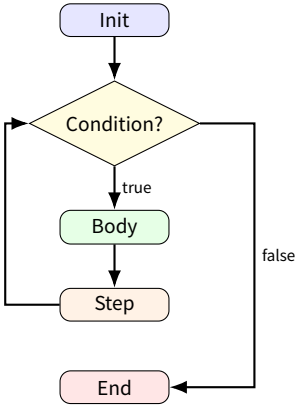
```c
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    if (i % 2 == 0) continue;
    printf("%d ", i);
}
// 출력: 1 3
```



- `break`: 루프를 즉시 빠져나감
- `continue`: 나머지를 건너뛰고 다음 반복으로

반복문의 흐름

- 초기화 → 조건 검사 → 본문 실행 → 증감 순서로 반복
- 조건이 거짓이 되면 반복이 종료
- `break`는 즉시 End로 이동
- `continue`는 Step으로 이동



반복문 주의사항

- **오프바이원(off-by-one)** 오류: <와 <= 선택에 주의
  - 0부터 $n-1$까지는 `i < n`, 1부터 n까지는 `i <= n`
- 반복 변수는 루프 내부에서 예측 가능하게 변경
- 중첩 반복은 **복잡도 증가** (데이터 크기 고려)
  - 반복문은 일반적인 컴퓨터에서 1초에 약 1억 번 실행
- 무한 루프는 `break`로 탈출 조건을 명확히
- 경계 조건에 주의 (0, 음수, 최대값, 최소값)

배열

배열이란?

- 같은 타입의 값을 **연속된 메모리**에 저장
- 인덱스는 0부터 시작: `a[0]`
- 경계 검사 없음: 범위를 초과해도 컴파일러가 경고하지 않는다


```c
int a[5] = {1, 2, 3, 4, 5};
printf("%d\n", a[0]); // 1
printf("%d\n", a[4]); // 5
// a[5]는 범위 밖 -> UB!
```

배열 활용: 최댓값 찾기

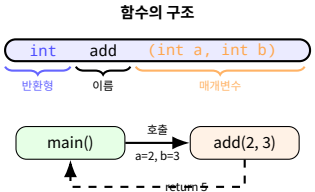
```c
#include <stdio.h>

int main(void) {
    int a[5] = {3, 7, 2, 9, 4};
    int max = a[0]; // 첫 원소로 초기화
    for (int i = 1; i < 5; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    printf("max=%d\n", max); // 9
    return 0;
}
```

- 지역 배열은 스택에 생성됨 → 큰 배열은 전역/동적 할당 사용

함수

함수의 기본 형태

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

int main(void) {
    printf("%d\n", add(2, 3));
    return 0;
}
```



- **반환형 + 이름 + 매개변수**로 정의
- `return`은 결과를 호출자에게 돌려준다
- 반환 값이 없으면 `void`

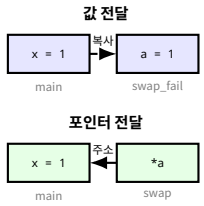
함수 선언 (Prototype)

```c
int max(int x, int y); // 선언 (prototype)

int main(void) {
    printf("%d\n", max(4, 7));
    return 0;
}

int max(int x, int y) { // 정의
    return (x > y) ? x : y;
}
```

- 함수를 쓰기 전에 **선언**(prototype)이 먼저 있어야 한다
- 선언과 정의의 시그니처가 다르면 컴파일 에러

값에 의한 전달 vs 포인터 전달

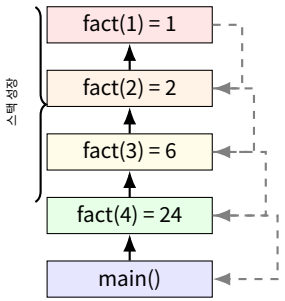
```c
// 값 전달: 원본 변경 안 됨
void swap_fail(int a, int b) {
    int t = a; a = b; b = t;
}
// 포인터 전달: 원본 변경됨
void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}
int main(void) {
    int x = 1, y = 2;
    swap_fail(x, y); // 1 2
    swap(&x, &y);    // 2 1
}
```



재귀 함수

```
1  int fact(int n) {
2      if (n <= 1) return 1;
3      return n * fact(n - 1);
4  }
```

- 재귀: 함수가 **자기 자신을 호출**
- 반드시 **기저 조건**(base case) 필요
- 깊은 재귀는 스택 오버플로우 위험



함수 주의사항 정리

- 선언과 정의의 시그니처가 다르면 에러
- `void` 함수는 값을 돌려주지 않는다
- C는 기본이 **값 복사** → 원본을 바꾸려면 포인터를 넘겨야 한다
- 재귀는 **기저 조건**을 꼭 확인 — 없으면 무한 재귀
- 지역 변수의 주소를 반환하면 안 된다 (함수 끝나면 사라지니까)
- `static` 지역 변수는 호출 사이에도 값이 살아 있다

정리

- **프로그래밍**: 문제를 쪼개고, 풀이를 설계하고, 코드로 옮기는 것
- **언어의 역사**: 기계어에서 고수준 언어로 발전, 8비트에서 64비트로 확장
- **C**: 1972년 탄생, 절차적/정적 타입/컴파일 언어, 시스템 프로그래밍의 기본
- **컴파일러**: 전처리 $\rightarrow$ 컴파일 $\rightarrow$ 어셈블 $\rightarrow$ 링크를 거쳐 실행 파일이 나온다
- **변수/연산자**: 타입이 메모리 크기를 정하고, 정수/실수 나눗셈 차이에 주의
- **입출력**: `printf`/`scanf`에서 포맷 지정자와 타입을 맞춰야 한다
- **제어문**: `if`/`switch`로 분기, `for`/`while`로 반복
- **배열**: 같은 타입을 연속 메모리에 저장, 인덱스는 0부터
- **함수**: 코드 재사용의 기본 단위, 값 복사가 기본이고 포인터로 원본 수정