

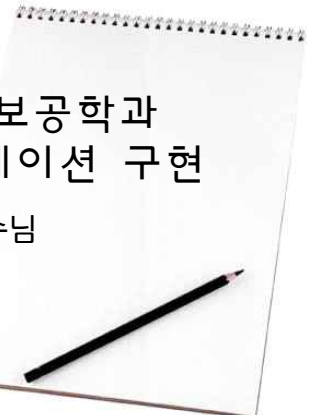


# C Language Portrait

---



학	과	컴퓨터정보공학과
과	목	C애플리케이션 구현
담당	교수	강환수 교수님
이	름	엄대용
학	번	20193432





## 9. 배열

- 배열 선언과 초기화
- 이차원과 삼차원 배열
- 배열과 포인터 관계
- 포인터 배열과 배열 포인터

## 10. 함수 기초

- 함수정의와 호출
- 함수의 매개변수 활용
- 재귀와 라이브러리 함수

## 11. 문자와 문자열

- 문자와 문자열
- 문자열 관련 함수
- 여러 문자열 처리

## 12. 변수 유효범위

- 전역변수와 지역변수
- 정적 변수와 레지스터 변수
- 메모리 영역과 변수 이용

## 13. 구조체와 공용체

- 구조체와 공용체
- 자료형 재정의
- 구조체와 공용체의 포인터와 배열

# 1. 프로그래밍 언어 개요

## 1.1 - '프로그램'이 무엇일까?

### ◎ 스마트폰과 컴퓨터에서의 프로그램

- 프로그램이란, 컴퓨터와 스마트폰에서 특정 목적의 작업을 수행하기 위한 관련 파일의 모임을 프로그램(program)이라 한다.



- 프로그램이라는 용어의 공통적 의미는 특정한 목적을 수행하기 위하여 이미 정해놓은 순서적인 계획이나 절차를 의미한다.
- 정보기술 분야에서의 프로그램은 특정 작업을 수행하기 위하여 그 처리 방법과 순서를 기술한 명령어와 자료로 구성되어있으며, 컴퓨터에게 지시할 일련의 처리 작업 내용을 담고 있고, 사용자의 프로그램 조작에 따라 컴퓨터에게 적절한 명령을 지시하여 프로그램이 실행된다.

### ◎ 프로그래머와 프로그래밍 언어

- 컴퓨터와 스마트폰 등의 정보기기에서 사용되는 프로그램을 만드는 사람을 프로그래머(Programmer)라고 한다.
- 프로그래머가 프로그램을 개발하기 위해 사용하는 언어이자, 사람과 컴퓨터가 서로 의사 교환을 하기 위한 언어가 프로그래밍 언어(Programming Language)이다.
- 프로그래밍 언어는 사람이 컴퓨터에게 지시할 명령어를 기술하기 위하여 만들어진 언어이다.
- 대표적인 프로그래밍 언어로는 FORTRAN, BASIC, C, C++, Visual Basic, Java, Python, C# 등 매우 다양하다.

### 1.2 - 언어의 계층과 번역

#### ◎ 하드웨어와 소프트웨어

- 컴퓨터는 영어단어인 compute와 er의 조합으로, 전자적으로 계산을 수행하는 장치이다.
- 하드웨어(Hardware)와 소프트웨어(Software)로 구성되어있으며, 하드웨어의 주요 구성요소로는 중앙처리장치(CPU), 주기억장치(Main Memory), 보조기억장치(Secondary Memory), 입력장치(Input Device), 출력장치(Output Device)로 구성되어있다.  
소프트웨어는 컴퓨터가 수행할 작업을 지시하는 전자적 명령어들의 집합으로 구성된 프로그램을 말한다.

#### ◎ 소프트웨어의 분류

- 소프트웨어란, 컴퓨터가 특정 작업을 수행할 수 있도록 해주는 전자적인 명령어의 집합으로 구성되며, 컴퓨터의 하드웨어가 해야 할 작업 내용을 지시한다.
- 소프트웨어는 응용 소프트웨어와 시스템 소프트웨어로 나뉜다.  
응용 소프트웨어란 문서 작성, 인터넷 검색, 게임, 동영상 재생 등과 같은 특정 업무에 활용되는 소프트웨어를 말하고, 시스템 소프트웨어란 컴퓨터가 잘 작동하도록 도와주는 기본 소프트웨어를 말한다.
- 시스템 소프트웨어는 운영체제(Operating System)와 각종 유틸리티(Utility) 프로그램으로 구분할 수 있다.
- 운영체제는 특정 CPU에 맞게 관련된 하드웨어를 작동하게 하고, 응용 소프트웨어를 실행해주는 소프트웨어이자, 컴퓨터 하드웨어 장치의 전반적인 작동을 제어하고 조정하며, 사용자가 최대한 컴퓨터를 효율적으로 사용할 수 있도록 돕는 시스템 프로그램을 말한다.
- 컴퓨터의 운영체제로는 유닉스(Unix), 리눅스(Linux), 윈도우즈(Windows), 맥(Mac)등이 있으며, 스마트폰의 운영체제로는 IOS, 안드로이드(Android), 파이어폭스(FireFox)등이 있다.

#### ◎ 기계어와 어셈블리어, 고급언어

- 기계어란 컴퓨터가 유일하게 바로 인식하는 언어로서, 전기의 흐름을 표현하는 1과 흐르지 않음을 표현하는 0으로 구성되어있는 것을 뜻한다.
- 프로그래밍 언어로 컴퓨터에게 명령을 내리기 위해서는 프로그래밍 언어를 기계어로 변환하는 컴파일러(Compiler)가 필요하다.
- 어셈블리어(Assembly Language)란 기계어를 프로그래머가 이해하기 쉬운 기호형태로 일대일 대응시킨 프로그래밍 언어이다. 어셈블리어는 CPU마다 제각각 다르므로, 다른 CPU를 사용한다면 그 CPU의 어셈블리어로 다시 작성해야 하는 번거로움이 있다.
- 컴퓨터의 중앙처리장치(CPU)에 적합하게 만든 기계어와 어셈블리어를 저급언어(Low Level Language)라 하며, 컴퓨터의 CPU에 의존하지 않고 유저가 쉽게 이해할 수 있도록 만들어진 언어를 고급언어(High Level Language)라 한다.
- 컴파일러(Compiler)란 고급언어로 작성된 프로그램을 기계어 or 목적코드(Object Code)로 바꾸어주는 프로그램을 말하며, 어셈블러(Assembler)는 어셈블리어로 작성된 프로그램을 기계어로 바꾸어주는 프로그램이다.

## 1. 프로그래밍 언어 개요

### 1.3 - 왜 C언어를 배워야 할까?

#### ◎ C언어의 역사

- C 언어는 1972년 데니스 리치(Dennis Ritchie)가 개발한 프로그래밍 언어이다.
- 새로운 운영체제인 유닉스(Unix)를 개발하기 위해 어셈블리 언어의 속도를 내며, 좀 더 쉽고, 서로 다른 CPU에서도 작동하는 프로그래밍 언어가 필요했고 이를 위해 만든 언어가 C 언어이다.
- C 언어는 켄 톰슨이 1970년에 개발한 B 언어에서 유래된 프로그래밍 언어이다.

#### ◎ C언어의 발전과 그 이후의 언어

- ANSI C란 1989년 미국표준화위원회(ANSI : America National Standards Institute)에서 공인한 표준 C(Standard C)를 지칭한다.

※ ALGOL(1960) -> BCPL(1967) -> B(1970) -> Traditional C(1972) -> ANSI C(1989) -> ANSI / ISO C(1999)

- 1983년에 얀 스트로스트룹이라는 인물이 C언어에 객체지향 프로그래밍 개념을 확장시킨 프로그래밍 언어 C++를 개발하였고, 1995년에는 중대형 컴퓨터 시스템 회사인 선 마이크로시스템즈사에서 인터넷에 적합한 언어인 자바(Java)를 발표하였다.  
이를 기반으로 2000년에는 마이크로소프트사가 개발한 C#에 많은 영향을 끼쳤으며, 2009년에는 구글이 C와 자바를 기반으로 한 새로운 프로그래밍 언어인 고(GO)를 발표하였다.
- 최근에는 Objective-C가 아이폰의 앱 개발언어로서 주목받고 있는데, 이는 1980년대에 브래드 콕스가 개발한 언어로, C언어의 기본 문법을 바탕으로 스몰토크(Smalltalk)언어의 특징을 접목한 객체지향 프로그래밍 언어이다.

#### ◎ C언어의 특징 및 필수적으로 배워야 하는 이유

- C언어는 함수 중심으로 구현되는 절차지향 언어(Procedural language)이다.

※ 절차지향 언어 - 시간의 흐름에 따라 정해진 절차를 실행한다는 의미로서, C언어는 문제의 해결 순서와 절차의 표현과 해결이 쉽도록 설계된 프로그램 언어이다.

- C언어는 간결하고 효율적인 언어이다.

※ 다양한 연산(비트연산, 중감연산 등)과 이미 개발된 시스템 라이브러리를 제공하며, 함수의 재귀호출이 가능함은 물론, 포인터와 메모리 관리기능을 갖고 있어 메모리를 적게 효율적으로 사용함으로써 프로그램의 실행 속도가 빠르다.

- C언어는 이식성(Portability)이 좋다.

※ 다양한 CPU와 플랫폼의 컴파일러를 지원하여 이식성이 좋다.

- C언어는 다소 배우기 어렵다는 단점이 있다.

※ 문법이 상대적으로 간결한 대신, 많은 내용을 함축하고 있어 비트와 포인터의 개념, 메모리 할당과 해제등의 관리 등 알아야 할 것이 많아 배우는 입장에선 다소 어려울 수 있다.

- C언어는 범용적인 프로그래밍 언어이자 자바, C#, Python 등의 여러 프로그래밍 언어의 뿌리가 되는 언어이기 때문에, C언어를 잘 알수록 자바, Objective-C 등의 여러 프로그래밍 언어의 습득 난이도가 쉬워짐은 물론, 임베디드 시스템, 응용 프로그램, 시스템 소프트웨어 개발 등 여러 분야에도 널리 사용되는 언어이므로 많이 알아갈수록 좋은 언어이다.

## 1. 프로그래밍 언어 개요

### 1.4 - 프로그래밍의 자료 표현

#### ◎ 프로그래밍의 내부 표현

- 십진수(decimal number)란, 하나의 자릿수(digits)에 사용하는 숫자가 0,1,2,3,4,5,6,7,8,9까지 열 개이므로 십진수라고 칭하며, 여기서 십이라는 것은 기수(base)라고 한다.

ex) 십진수 5319 →  $5 \times 10(3) / 3 \times 10(2) / 1 \times 10(1) / 9 \times 10(0)$

- 이진수(binary number)란, 수의 자릿수에 사용할 수 있는 숫자와 0과 1까지 두 개이므로 이진수라 한다.

ex) 이진수 11010 →  $1 \times 2(4) / 1 \times 2(3) / 0 \times 2(2) / 1 \times 2(1) / 0 \times 2(0) = 26$

- 시스템 내부에서는 이진수를 사용하여 저장하는데, 디지털 신호에서는 전기가 흐를 경우 1, 그렇지 않을 경우 0으로 표현되므로, 컴퓨터 내부에서 처리하는 숫자는 0과 1을 표현하는 이진수 체계를 사용한다.

#### ◎ 비트와 바이트

- 컴퓨터 메모리의 저장단위 or 정보 처리 단위중에서 가장 작은 기본 정보 단위를 비트(bit)라 한다.

- 비트가 연속적으로 8개 모인 정보 단위를 바이트(byte)라 한다.

※ 1바이트 = 8비트

- 바이트의 1/2 크기인 4비트를 니블(Nibble)이라 하며, 바이트가 4개, 8개가 모인 경우를 워드(Word)라 한다.

#### ◎ 논리와 문자 표현

- 참(True)과 거짓(False)을 의미하는 두 가지 정보를 논리값이라 하며, 하나의 비트 정보도 0과 1이므로 이를 각각 참과 거짓으로 표현할 수 있다.

- 앞에서의 이진 논리 변수와 AND, OR, NOT의 논리 연산을 이용한 **부울 대수(Boolean Algebra)**는 논리 회로를 수학적으로 해석하기 위해 영국의 수학자 조지 부울(George Boole)이 제창한 기호 논리학의 한 분야이다.

- 부울 대수는 컴퓨터가 정보를 처리하는 방식에 대하여 이론적인 배경을 제공하며, 0과 1 두 값 중 하나로 한정된 변수들의 상관관계를 AND, OR, NOT 등의 여러 연산자를 이용하여 논리적으로 나타낸다.

- 논리 연산에서 AND 연산은 두 개의 항이 모두 1이어야 1이며, OR 연산은 둘 중 하나만 1이면 결과가 1이다. 항이 하나인 NOT 연산은 항이 0이면 1로, 1이면 0인 결과를 반환하는 연산이다.  
또 다른 연산인 AND와 NOT을 결합한 NAND 연산, OR과 NOT을 결합한 NOR 연산, 그리고 XOR 연산은 두 개의 입력값이 같으면 0을 출력하고, 다르면 1을 출력하는 연산이다.

0	0	0
0	1	0
1	0	0
1	1	1

AND

0	0	0
0	1	1
1	0	1
1	1	1

OR

0	1
1	0

NOT

0	0	1
0	1	1
1	0	1
1	1	0

NAND

0	0	1
0	1	0
1	0	0
1	1	0

NOR

0	0	0
0	1	1
1	0	1
1	1	0

XOR

#### ◎ 아스키코드와 유니코드

- 아스키(ASCII)코드는 개인용 컴퓨터에서 가장 많이 사용하는 표준 코드이자, 영문 알파벳을 사용하는 대표적인 문자 인코딩 방법으로서, 초기에는 128개의 코드로 구성되었다가 컴퓨터가 발전함에 따라 256개로 확장되었다.
- 유니코드(Unicode)는 전 세계의 모든 언어를 모두 표현하고, 하나의 코드 체계로 통합하기 위하여 만들어진 코드로서, 모든 문자를 컴퓨터에서 일관되게 표현하고 다룰 수 있도록 설계된 산업표준이다.

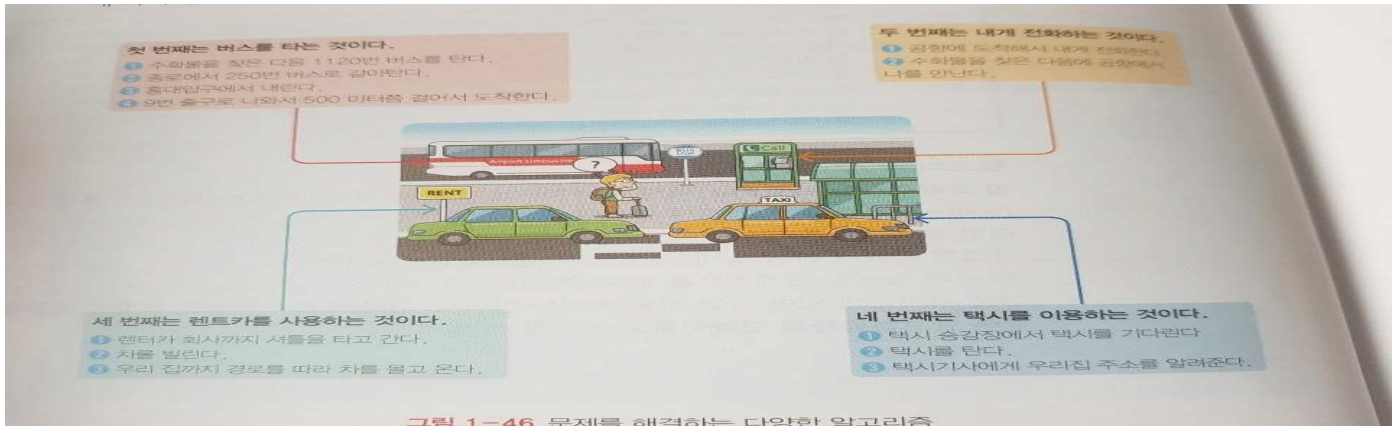


# 1. 프로그래밍 언어 개요

## 1.5 - 소프트웨어 개발

### ◎ 소프트웨어와 알고리즘

- 소프트웨어란 보통 '프로그램'이라고 부르는 것 이외에도 데이터와 문서까지를 포함하는 포괄적인 개념이다.
- 알고리즘(Algorithm)이란 어떠한 문제를 해결하기 위한 절차나 방법으로 명확히 정의된 유한 개의 규칙과 절차의 모임으로, 컴퓨터에게 작업을 시키는 데 있어 가장 기초가 되는 것이며 컴퓨터 프로그램은 특정한 업무를 수행하기 위한 정교한 알고리즘의 집합이라고 할 수 있다.



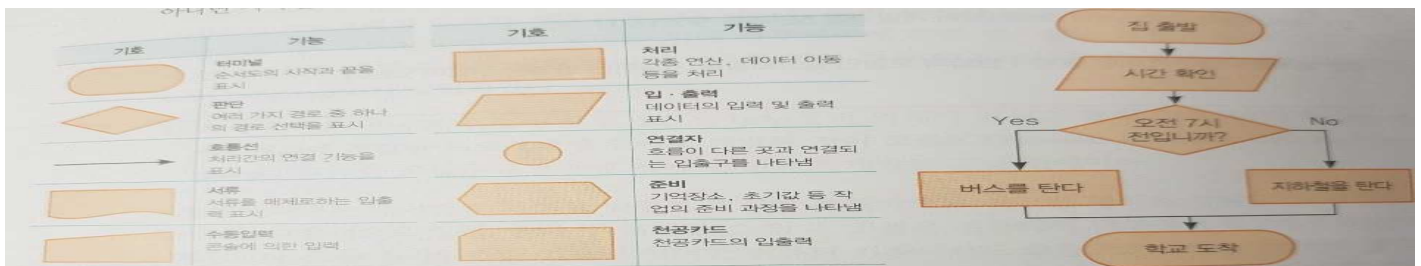
### ◎ 소프트웨어 개발 방법

- 건물을 건축할 때 건설을 기획 -> 설계 도면을 제작 -> 건물을 짓는 시공의 3단계로 이루어져 있듯이, 소프트웨어를 개발할 때도 비슷한 과정이 필요하며 이러한 과정을 연구하는 분야를 소프트웨어 공학이라 한다.
- 소프트웨어 공학이란 실제 컴퓨터에서 신뢰성이 있고 효과적으로 작동하는 소프트웨어를 경제적으로 얻기 위해 올바른 공학적 원리들을 체계화시킨 공학 분야로서, 공학적 원리에 의하여 소프트웨어를 개발하는 학문이자 개발과정인 분석, 설계, 개발, 검증, 유지보수 등 개발수명주기 전반에 걸친 계획, 개발, 검사 등을 연구하는 분야이다.

Requirements Analysis(요구사항 분석)	Requirements Specification	요구사항 또는 해결할 문제를 먼저 파악한다.
Design(설계)	System Architecture	프로그램을 설계한다.
Implementatin(구현)	Software	설계된 알고리즘에 따라 코딩한다.
Verification(검증)	System Testing	작성된 프로그램을 테스트한다.
Maintenance(유지보수)	System Support	프로그램을 문서화하고 유지 보수한다.

### ◎ 설계 단계에서의 알고리즘 기술 방법

- 소프트웨어 개발 과정에서 분석단계가 끝나면, 프로그래머는 알고리즘을 생각해 소프트웨어 설계에 들어간다.
- 문제를 해결하기 위한 절차나 방법의 모임인 알고리즘은 우리가 사용하는 자연어 또는 흐름도(Flow Chart)나 의사코드(Pseudo Code)등을 사용하여 표현될 수 있다.
- ※ 의사코드는 슈도코드라고도 하는데, 특정 프로그래밍 언어의 문법을 따르지 않고, 간결한 특정 언어로 코드를 흉내내어 알고리즘을 써놓은 코드를 말한다.
- ※ 흐름도는 알고리즘을 표준화된 기호 및 도형으로 도식화하여 데이터의 흐름과 수행되는 연산들의 순서를 표현하는 방법이다.





### 1.6 - 다양한 프로그래밍 언어

#### ◎ 50 ~ 60년대에 개발된 프로그래밍 언어

- **포트란(FORTRAN)**은 IBM 704 시스템에서 과학과 공학 및 수학적 문제들을 해결하기 위해 고안된 프로그래밍 언어로, 널리 보급된 최초의 고급 언어이다.  
FORmula TRANslating system의 약자로서, 수학에서 사용하는 수식을 컴퓨터가 읽을 수 있도록 기계어로 변환해준다고 해서 붙인 이름이다.  
가장 오래된 언어지만 언어 구조가 단순해 지금도 과학 계산 분야에서는 많이 사용되고 있는 언어이다.
  - **코볼(COBOL)**은 사무처리 언어의 통일을 위해 데이터 시스템 언어 협회(CODASYL)가 설립되었고, 사무처리에 적합한 프로그래밍 언어로 개발한 것이 시초이다. 포트란에 이어 두 번째로 개발된 고급언어이자, 컴퓨터의 내부적인 특성에 독립적으로 설계되었고 사무처리에 주 목적을 두고 있어 다른 언어에 비해 데이터를 쉽게 읽고 쓰며, 양식을 가진 보고서를 쉽게 만들 수 있고 구어체 문장 형태를 갖고 있어 포트란에 비해 쉽게 프로그램 작성이 가능한 언어이다.
  - **알골(ALGOL)**은 미국의 포트란에 대항하기 위해 유럽을 중심으로 만든 과학기술 계산용 프로그래밍 언어로서, 알고리즘을 표현하기 위한 언어인 ALGOrithmic Language를 줄여서 만든 이름이다.  
알고리즘 연구 개발에 적합한 언어로 개발되었고, 최초로 재귀호출이 가능한 프로그래밍 언어이다.  
또한, 파스칼, C언어 등 이후 언어의 발전에 큰 영향을 주었으나 대중화엔 성공하지 못했고, 이후에 개발된 후속작도 너무 복잡하고 방대한데다 입출력 문장을 지원하지 않아 사장되다시피 한 언어이다.
  - **베이직(BASIC)**은 1964년경 미국 다트머스 대학의 캄니 교수와 커프 교수가 개발한 프로그래밍 언어로서, 'Beginner's All-purpose Symbolic Instruction Code'의 약어로 초보자도 쉽게 배울 수 있도록 만들어진 대화형 프로그래밍 언어이다.  
대화형의 영어 단어를 바탕으로 약 200개의 명령어들로 구성되어 있으며, 문법이 간단하고 문장의 종류가 많지 않아 배우고 쓰기가 간단하고 쉬운 언어이다.  
또한, 베이직은 대화형 프로그래밍 언어로서 인터프리터를 사용하기 때문에, 프로그램상의 코드 문제점을 쉽게 파악할 수 있으며 컴파일 없이 바로 작동되는 것이 장점이다.  
하지만 인터프리터를 거쳐야 하기 때문에 실행속도가 느리다는 단점 또한 존재한다.
- ※ 개인용 컴퓨터의 출현과 함께 베이직은 기본 개발 언어로 탑재되어 범용적인 언어로 널리 사용되었고, 후에 마이크로소프트가 이 베이직을 기본으로 비주얼 베이직(Visual Basic)이라는 프로그램 언어를 개발하였다.

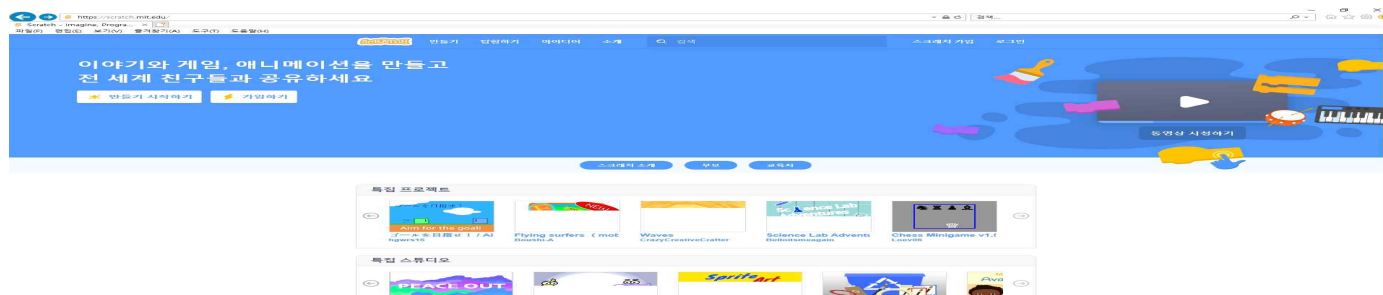
#### ◎ 70년대 이후 개발된 프로그래밍 언어

- **파스칼(PASCAL)**은 프랑스의 수학자인 파스칼의 이름에서 따온 언어로서, 알고리즘 학습에 적합하도록 1971년 스위스 취리히 공과대학교의 니콜라우스 비르트 교수에 의해 개발된 프로그래밍 언어이다.  
파스칼은 알골(ALGOL)을 모체로 한 언어로, 초보자들이 프로그램의 문법적 에러를 줄일 수 있도록 매우 엄격한 문법을 가진 프로그래밍 언어이자 구조적인 프로그래밍이 가능하도록 begin~end의 블록 구조가 설계되어 있다.
- **C++**는 1972년에 개발된 C언어로부터 1983년에 C++로 발전한 프로그램 언어이다.  
C++는 객체지향 프로그래밍을 지원하기 위해 C언어가 가지는 장점을 그대로 계승하면서, 객체의 상속성 등의 개념을 추가한 효과적인 언어로, 안 스트로스트롭이 개발하였다.  
C++는 C언어의 확장으로서 기존의 C언어로 개발된 모든 프로그램들을 수정없이 그대로 사용할 수 있으며, C++언어가 C언어와 유사한 문법을 사용함으로 인해서 C언어에 익숙한 프로그래머들이 C++언어를 쉽게 배울 수 있다는 장점을 가지게 된 언어이다.

- **파이썬(Python)**은 1991년 네덜란드의 귀도 반 로섬(Guido Van Rossum)이 개발한 객체지향 프로그래밍 언어로서, 현재까지 미국의 대학에서 컴퓨터 기초 과목으로 가장 많이 가르치는 프로그래밍 언어 중 하나이다. 또한, 파이썬은 비영리의 파이썬 소프트웨어 재단이 관리하는 개방형, 공동체 기반 모델을 가지고 있으며, C#으로 구현된 닷넷프레임워크 위에서 동작하는 **IronPython**, Java로 구현되어 JVM 위에서 동작하는 **Jython**, 파이썬 자체로 구현된 **PyPy**등의 다양한 언어로 만들어진 버전이 있다. 파이썬이 대학의 기초 교육에 많이 활용되는 이유로는 무료이며, 간단하면서도 효과적으로 객체지향을 적용할 수 있는 강력한 프로그래밍 언어이기 때문이다. 파이썬의 특징으로는, 베이직과 같은 인터프리터 언어로서 간단한 문법구조를 가진 대화형 언어이며, 동적 자료형(Dynamic Typing)을 제공하여 변수를 선언하지 않고 사용할 수 있고, 위에서 나열했듯 다양한 모듈을 제공하기 때문에 여러 플랫폼에서 활용될 수 있는 프로그램을 쉽고 빠르게 개발할 수 있어 개발기간이 매우 단축되는 것이 또 하나의 특징이다.
- **자바(JAVA)**는 1992년 미국의 선 마이크로시스템즈에서 가전제품들을 제어하기 위해 고안한 언어에서 비롯되었다. 초기에는 객체지향 언어로 이용되고 있는 C++를 이용하였으나, 다양한 하드웨어를 지원하는 분산 네트워크 시스템 개발에 부족함을 느낀 개발팀은 C++ 언어를 기반으로 한 오크(Oak)라는 언어를 직접 개발하게 된다. 이 개발의 책임자였던 제임스 고슬링(James Gosling)은 이 오크라는 언어를 발전시켜 자바라는 프로그래밍 언어를 개발하게 된다. 결국 자바는 1995년에 공식 발표되었고, 더 나아가 월드와이드웹 이용에도 적합하도록 자바를 발전시키게 된다. 이와 더불어 자바의 개발도구인 JDK(Java Development Kit)를 발표하였고, 이 개발환경은 현재까지도 계속 발표되고 있다.

## ◎ 프로그래밍 초보를 위한 비주얼 프로그래밍 언어

- **스크래치(Scratch)**는 2007년 MIT 대학의 미디어랩에서 개발한 비주얼 프로그래밍 개발 도구이다. 스크래치는 브라우저에서 직접 개발하는 환경으로, 커뮤니티 기반 웹 인터페이스로 구성되어 있다. 이는 컴퓨터에 대한 지식이 전혀 없는 일반인, 청소년, 또는 프로그래밍에 갓 입문한 학생들을 대상으로 컴퓨터 프로그래밍의 개념을 이해할 수 있도록 도와주는 교육용 프로그래밍 언어가 되었다.



- **앨리스**는 카네기 멜론 대학교의 랜디 포시 교수가 주도하여 개발한 프로그래밍 교육 도구로, 삼차원 인터랙티브 그래픽 콘텐츠를 제작하면서 객체지향 프로그래밍 기법을 학습할 수 있는 교육용 소프트웨어이다. 특히 앨리스는 컴퓨터 전공학과 의 저학년 학생들에게 프로그래밍 개념과 실습을 학습할 수 있도록 지원하는 프로그래밍 소프트웨어로 특히 자바, C++, C#과 같은 객체지향 프로그래밍 기법을 학습하는 데 유용한 개발 도구로 알려져 있다. 또한, 다른 비주얼 프로그래밍 개발 도구처럼 문서를 편집하는 형식이 아니라 다양한 캐릭터나 객체 등을 끌어다 놓는 드래그 앤 드롭(Drag & Drop)방식으로 프로그래밍을 할 수 있도록 해줬기 때문에 프로그래밍 문법 학습에 대한 학습자들의 부담을 덜어주고 쉽게 프로그래밍이 가능하도록 하는 장점도 있다.

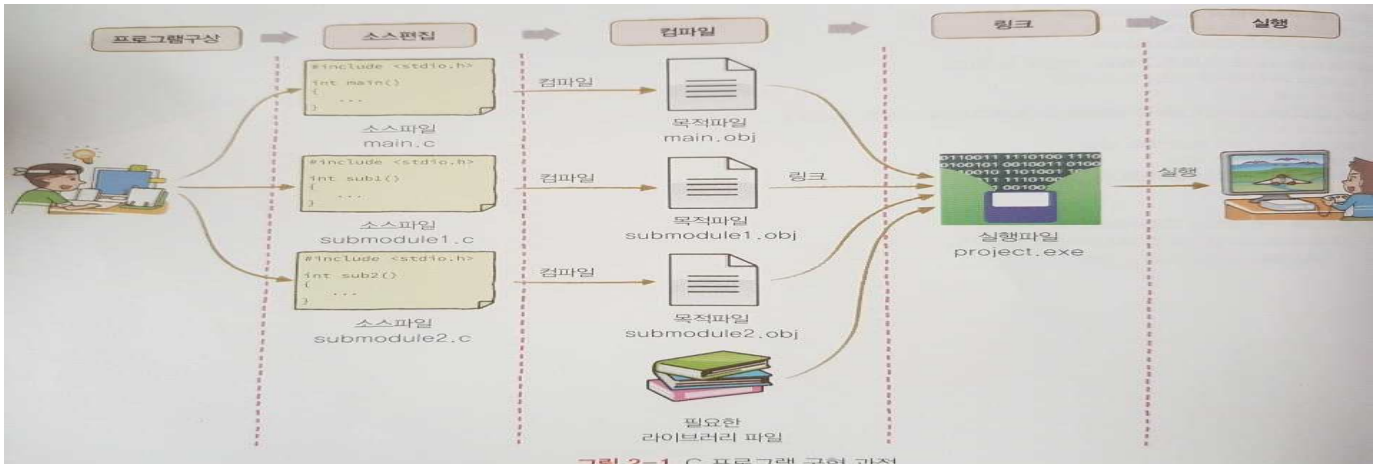


## 2. C 프로그래밍 첫걸음

### 2.1 - 프로그램 구현 과정과 통합개발환경

#### ◎ 프로그램 구현 과정

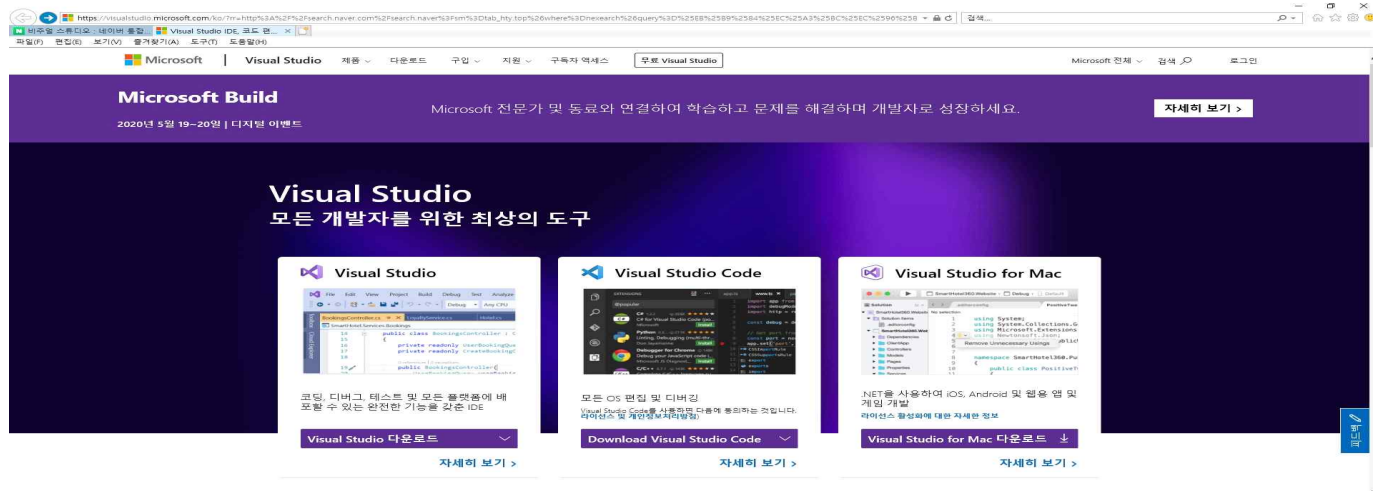
- 하나의 소프트웨어를 개발하기 위해서는 요구분석, 설계, 구현, 검증, 유지보수라는 5단계의 과정을 거친다. 이 과정 중에서 프로그램을 실제로 구현하기 위해서는 프로그램구상, 소스편집, 컴파일, 링크, 실행의 5단계를 거치며, 구현을 생각한다는 것은 C나 자바처럼 개발할 프로그래밍 언어는 이미 선정되어있다고 볼 수 있다.



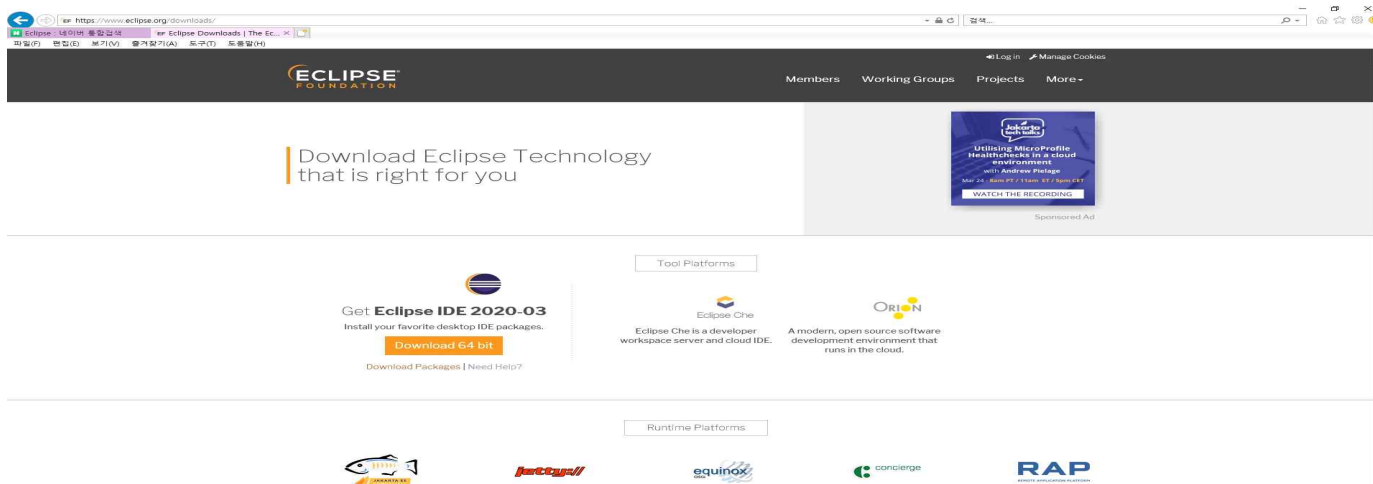
- 프로그램을 개발하기 위해 가장 먼저 해야할 것은 소스파일을 어떻게 작성해야 할지 생각하는 것으로, 소스 또는 소스코드는 선정된 프로그래밍 언어인 C 프로그램 자체로 만든 일련의 명령문을 의미한다. C와 같은 프로그래밍 언어로 원하는 일련의 명령어가 저장된 파일을 **소스파일(Source File)**이라 하며 일반 텍스트파일로 저장되어야 한다. 또한, 소스파일은 프로그래밍 언어에 따라 고유한 확장자를 갖는데, C언어의 경우 **.c** / 자바의 경우 **.java** / C++는 **.cpp**이다.
- **컴파일러(Compiler)**는 고급언어인 프로그래밍 언어로 작성된 소스파일에서 기계어로 작성된 목적파일을 만들어내는 프로그램이다. 이 과정을 컴파일 과정이라 하며, 컴파일러에 의해 처리되기 전의 프로그램을 소스코드라 하면 컴파일러에 의해 기계어로 번역된 프로그램은 목적코드라 한다.
- **링커(Linker)**는 하나 이상의 목적파일을 하나의 실행파일(Execute File)로 만들어 주는 프로그램으로서, 여러 개의 목적파일을 연결하고 참조하는 라이브러리를 포함시켜 하나의 실행 파일을 생성하는데, 이 과정을 링크(Link) 또는 링킹(Linking)이라 한다. 또한, 자주 사용하는 프로그램을 작성할 때, 프로그래머마다 새로 작성할 필요 없이 개발환경에서 미리 만들어 컴파일해 저장해 놓는데 이 모듈을 **라이브러리(Library)**라 한다. 라이브러리란 공용으로 사용하기 위해 이미 만든 목적코드로서 파일 \*.lib 또는 \*.dll 등으로 제공되며, 링크의 결과인 실행파일의 확장자는 .exe, .dll, .com 등이다. 비주얼 스튜디오에서는 컴파일과 링크 과정을 하나로 합쳐 **빌드(Bulid)**라고 칭한다.
- **오류(Error)**란 프로그램 개발과정에서 나타나는 모든 문제를 뜻하며, 발생 시점에 따라 컴파일(시간) 오류, 링크(시간) 오류, 실행(시간) 오류로 구분할 수 있다. 또한, 오류의 원인과 성격에 따라 프로그래밍 언어 문법을 잘못 기술한 문법 오류(Syntex Error)와 내부 알고리즘이 잘못되거나 원하는 결과가 나오지 않은 등의 논리 오류(Logic Error)로 분류할 수도 있다.
- **디버깅(Debugging)**이란 프로그램 개발 과정에서 발생하는 다양한 오류를 찾아 소스를 수정하여 다시 컴파일, 링크, 실행하는 과정을 뜻하며, 이를 도와주는 프로그램을 디버거(Debugger)라 한다.

## ◎ 다양한 통합개발환경

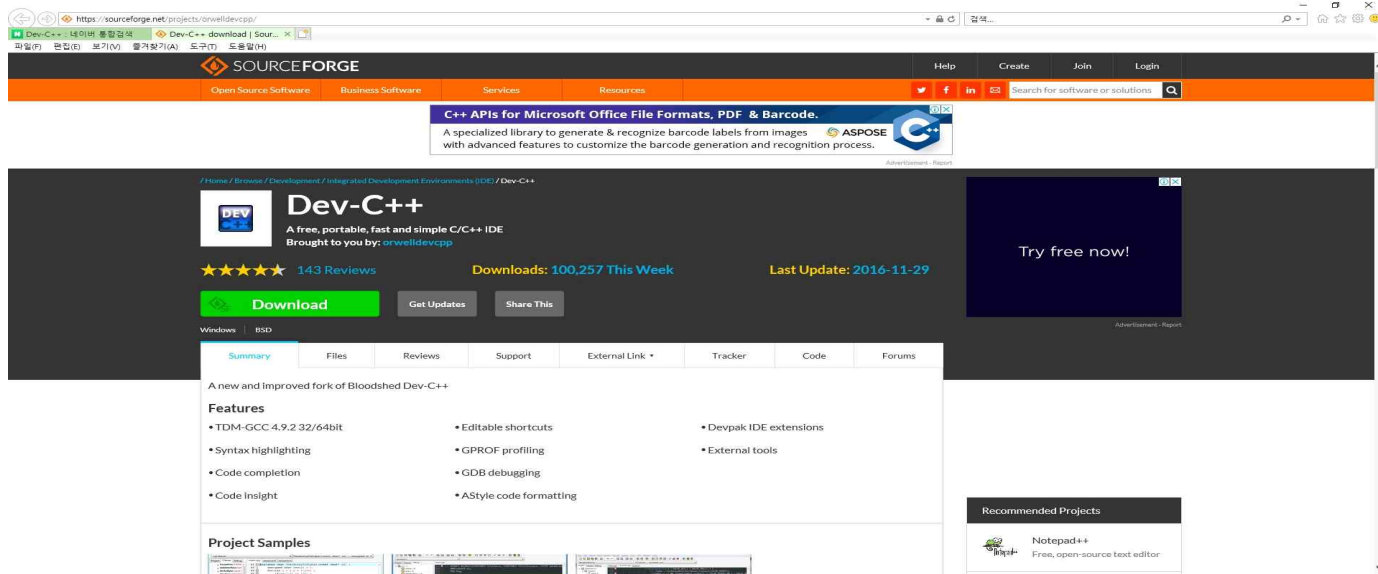
- 프로그램 개발에 필요한 편집기(Editor), 컴파일러, 링커, 디버거 등을 통합하여 편리하고 효율적으로 제공하는 개발환경을 통합개발환경, 약자로 IDE(Integrated Development Environment)라 한다.
- **마이크로소프트 비주얼 스튜디오**는 여러 프로그램 언어를 이용하여 응용 프로그램 및 앱을 개발할 수 있는 다중 플랫폼 개발 도구이자 윈도우 운영체제에서 이용되고 있고, 상용버전인 엔터프라이즈, 무료 버전인 커뮤니티로 나뉘어져 있다.



- **이클립스**는 이클립스 컨소시엄이 개발한 유니버설 도구 플랫폼으로서, 모든 부분에 대해 개방형이며 PDE(Plug-in Development Environment) 환경을 지원하여 확장 가능한 통합개발환경이다.



- **DEV-C++**는 완전 무료인 통합개발환경으로서, 유닉스 운영체제용 컴파일러인 GCC와 MinGW를 포함하여 배포한다.



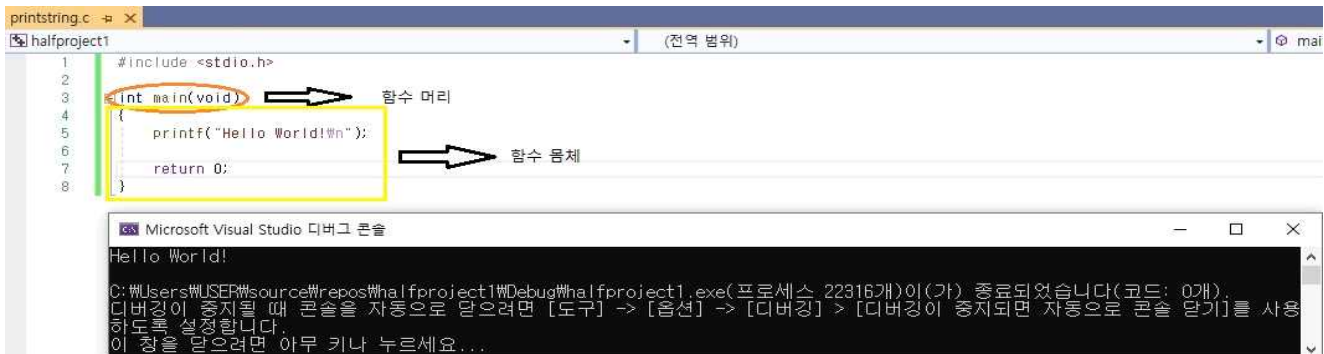


## 2. C 프로그래밍 첫걸음

### 2.2 - C 프로그램의 이해와 디버깅 과정

#### ◎ 함수의 이해

- C프로그램의 시작과 끝은 함수이며, C프로그램과 같은 절차지향 프로그램 또한 함수로 구성된다.  
함수는 크게 프로그래머가 직접 만드는 함수인 **“사용자 정의 함수”**와 시스템이 미리 만들어 놓은 함수인 **“라이브러리 함수”**로 분류한다.
- 이러한 함수에서의 용어는 크게 4가지로 정의된다.
  - ※ **함수 정의** : 사용자 정의 함수를 만드는 과정 / **함수 호출** : 라이브러리 함수를 포함해서 만든 함수를 사용하는 것
  - 매개변수** : 함수를 정의할 때 나열된 여러 입력 변수 / **인자** : 함수 호출 과정에서 전달되는 여러 입력값
- 프로그램이 실행되면 운영체제는 프로그램에서 가장 먼저 main() 함수를 찾고, 입력 형태의 인자로 main() 함수를 호출하며 호출 된 main()함수의 첫 줄을 시작으로 마지막 줄까지 실행하면 프로그램은 종료된다.
- ※ **함수 main()**은 프로그램 실행 시 가장 먼저 호출되는 특별한 함수로, CRT 시작함수(C Runtime Startup Function)라 한다.
- C프로그램에서 main() 함수는 자동차에 시동을 켜는 열쇠와 같은 역할을 하므로 반드시 정의되어야 하며, 이러한 함수의 형태는 **함수 머리(function header)**, **함수 몸체(function body)**로 나뉘어진다.
- ※ **함수 머리** : int main(void)와 같이 함수에서 제일 중요한 결과값의 유형, 함수이름, 매개변수인 입력 변수 나열을 각각 표시한다.  
**함수 몸체** : 함수 머리 이후 {...}의 구현 부분.
- 예제) printstring.c



- 함수 **puts(“”)**는 원하는 문자열을 괄호(“”)안에 기술하면 그 인자를 현재 위치에 출력한 후 다음 줄 첫 열로 이동하여 출력을 기다리는 함수이고, 함수 **printf(“”)**는 원하는 문자열을 괄호(“”)안에 기술하면 그 인자를 현재 줄의 출력 위치에 출력하는 함수이지만 다음 줄로 이동하지 않기 때문에 끝에 \n을 꼭 붙여줘야 한다.

#### ◎ 디버깅 및 다양한 오류 점검

- 소스코딩을 하다보면 ;, <>, (), {} 등의 문자를 빠뜨려서 발생하는 컴파일 에러들은 흔하게 발생하게 된다. 그럼에도 불구하고 생기는 오류들은 통합개발환경 IDE가 잡아주는데, 오류가 발생한 줄과 오류 원인등을 알려 주기 때문에 수정이 간편해진다.



### 3. 자료형과 변수

#### 3.1 - 프로그래밍 기초

##### ◎ C 프로그램 구조와 프로그램 실행

- C 프로그램은 하나 이상의 여러 함수가 모여 한 프로그램으로 구성되며, 적어도 `main()` 함수 하나는 구현되어야 응용프로그램으로 실행될 수 있다.
- 비주얼 스튜디오를 예로 들면 이 프로그램에서 솔루션은 여러 개의 프로젝트를 가지며, 프로젝트는 소스파일을 포함한 여러 자원으로 구성된다.  
다시 말해서, 한 프로젝트는 단 하나의 함수 `main()`과 다른 여러 함수로 구현되며, 최종적으로는 프로젝트의 이름으로 하나의 실행파일이 만들어지게 된다.  
함수 `main()`이 구현된 소스의 구조를 요약하면 다음과 같다.
- ※ **전처리(전처리 지시자)** - `#include <stdio.h>`, `#define PI 3.14` -> **외부선언(함수 및 자료형, 변수 선언)** - `typedef int my_int` 등  
-> **함수 `main()`(기본 함수인 `main()` 구현)** - `/* 블록 주석 */`, `//한 줄 주석` 등 -> **기타 함수구현(필요한 다른 함수 구현)** - **지역변수 선언** 등
- `main()` 함수는 프로그램이 실행되면 가장 먼저 시작되는 부분이며, 위에서 아래로, 좌에서 우로, 문장이 위치한 순서대로 실행된다.

##### ◎ 키워드와 식별자

- **키워드(Keyword)**란 문법적으로 고유한 의미를 갖는 예약된 단어인 예약어를 일컫는 말이며, 여기서 ‘예약’이라는 의미는 프로그램 코드를 작성하는 사람이 이 단어들을 다른 용도로 사용해서는 안된다는 뜻이 담겨있다.  
기본적인 키워드는 `break`, `char`, `default`, `float`, `if`, `static`, `void` 등이 정의되어있으며, 통합개발환경의 컴파일러에 따라 기본적인 키워드 이외에 다른 키워드도 추가될 수 있다.
- **식별자**란 키워드와는 달리 프로그래머가 자기 마음대로 정의해서 사용하는 단어들을 일컫는다.  
마음대로 정의하고 사용할 수 있지만, 몇 가지의 사용 규칙을 준수해야만 한다.
- ※ **C 프로그램에서의 예약자인 키워드와 비교하여 철자, 대문자, 소문자 등 무엇이랄도 달라야만 한다.**  
식별자는 영문자(**대소문자 알파벳**), 숫자(`0 ~ 9`), 밑줄(`_`)로 구성되며, 식별자의 첫 문자로 숫자가 나올 수 없다.  
프로그램 내부의 일정한 영역에서는 서로 구별되어야 한다.  
키워드는 식별자로 이용할 수 없다.  
식별자는 대소문자를 모두 구분한다. (ex - `Count`, `count`, `COUNT`는 모두 다른 변수)  
식별자의 중간에 공백(`space`)문자가 들어갈 수 없다.

##### ◎ 문장과 주석

- **문장**이란 프로그래밍 언어에서 컴퓨터에게 명령을 내리는 최소 단위를 말하며, 마지막에 세미콜론 `;`으로 종료된다.  
여러 개의 문장을 묶으면 블록(block)이라고 하며, { 문장1, 문장2, ... }처럼 중괄호로 열고 닫는다.  
또한, 블록 내부에서 문장들을 탭(Tab) 정도만큼 오른쪽으로 들여쓰는 소스 작성 방식을 들여쓰기라 한다.
- **주석**이란 프로그램 내용에는 전혀 영향을 미치지 않는 설명문을 말한다.  
그러나 프로그램 개발을 하거나 프로그램의 유지보수를 하는데 있어 매우 중요한 프로그램의 과정 중 하나이기 때문에, 주석에는 자신을 비롯한 이 소스를 보는 모든 사람이 이해할 수 있도록 도움이 되는 설명을 담고 있어야 하며 시각적으로 정돈된 느낌을 주면서도 프로그램의 내용을 적절히 설명해주어야 한다.
- 주석에는 `//(한 줄 주석)`과 `/* ... */(블록 주석)`이 있으며, 한 줄의 코드 설명이 필요할 때는 `//`를 사용하고 여러 줄에 걸쳐 코드 설명이 필요할 때는 `/* ... */`를 사용한다.
- 주석은 컴파일 과정의 대상이 아니므로 프로그램의 실행에 영향을 미치지 않으며, 실행 파일의 속도와 크기에도 전혀 영향을 미치지 않는다.

### 3. 자료형과 변수

#### 3.2 - 자료형과 변수 선언

##### ◎ 자료형과 변수 개요

- **자료형**이란 프로그래밍 언어에서 자료를 식별하는 종류를 말한다.  
C 프로그래밍에서의 자료형은 기본형(Basic Type), 유도형(Derived Type), 사용자정의형(User Defined Type) 등으로 나눌 수 있고, 기본형은 다시 정수형, 실수형, 문자형, void로 나뉘는데 프로그래머는 이러한 자료형에 적당한 알고리즘을 적용해 프로그램을 작성한다.
- **변수**란 정수, 실수, 문자 등의 자료값을 저장하는 공간이다.  
변수에는 고유한 이름이 붙여지며, 물리적으로 기억장치인 메모리에 위치한다.  
또한, 변수는 선언된 자료형에 따라 변수의 저장공간 크기와 저장되는 자료값의 종류가 결정되며, 자료형 키워드는 정수형(short, int, long) / 실수형(float, double) / 문자형(char) 등이 있다.  
변수에 여러 값을 저장할 수 있고 저장되는 값에 따라 변수값이 바뀔 수 있지만 **마지막에 저장된 하나의 값만 저장 유지**되는 특징이 있다.

##### ◎ 변수선언과 초기화

- **변수선언**이란 컴파일러에게 프로그램에서 사용할 저장 공간인 변수를 알리는 역할이자, 자신이 선언한 변수를 사용하겠다는 약속의 의미를 일컫는다.  
변수를 사용하기 위해서는 변수선언(Variables Declaration)과정이 반드시 필요하며, 특징은 다음과 같다.
  - ※ 변수선언은 자료형을 지정한 후 고유한 이름인 변수이름을 나열하여 표시한다.  
자료형은 int, double, float와 같이 원하는 자료형 키워드를 사용하며, 변수이름은 되도록 소문자를 이용하고, 사용 목적에 알맞은 이름으로 중복되지 않게 붙여야 한다.  
변수선언도 하나의 문장이므로 세미콜론(;)으로 종료된다.  
변수선언 이후에는 정해진 변수이름으로 값을 저장하거나 참조할 수 있다.  
하나의 자료형으로 여러 개의 변수를 한번에 선언하려면 자료형 이후에 변수이름을 콤마로 나열한다.(ex - int height, weight;)
- 변수에 원하는 자료값을 저장하기 위해서는 대입연산자 '='를 사용하며, 대입연산자 '='의 의미는 오른쪽에 위치한 값을 이미 선언된 왼쪽 변수에 저장한다는 의미이며, 대입연산이 있는 문장을 대입문이라고 한다.  
(ex - int age; -> age = 20; / age = 21; => age = 21;)
- **변수의 초기화**란 변수를 선언만 하고 자료값을 아무것도 저장하지 않으면 이상한 값이 저장되거나 오류가 발생하기 때문에, 변수를 선언한 이후에 반드시 값을 저장하는 것을 말한다.

##### ◎ 변수의 3요소와 이용

- 변수에서 주요 정보인 **변수 이름**, **변수의 자료형**, **변수의 저장값**을 변수의 3요소라 한다.  
변수선언 이후 저장값이 대입되면 변수의 3요소가 결정되며, 예를 들면 (int num1 = 30;)일 경우  
**int - 자료형 / num1 - 변수 이름 / 30 - 저장값**의 3요소가 결정된 것이다.
- 문장에서 변수의 의미는 저장공간 자체와 저장공간에 저장된 값으로 나눌 수 있다.  
보통 대입연산자 '=' 왼쪽에 위치한 변수는 저장공간 자체의 사용을 의미하며, '=' 오른쪽에 위치한 변수는 저장값의 사용을 의미한다.  
(ex - int num1 = 30, num2 = 14;  
int difference;  
difference = num1 - num2; => difference = 16;)  
저장공간                  저장값



### 3. 자료형과 변수

#### 3.3 - 기본 자료형

##### ◎ 정수 자료형

- 정수형(Integer Types)의 기본 키워드는 int이다.  
즉, int형으로 선언된 변수에는 365, 030, 0xF3과 같이 십진수, 팔진수, 십육진수의 정수가 다양하게 저장될 수 있으며, 여기서 파생된 자료형이 short와 long이다.  
short와 long의 기준이 되는 것이 int이며, short는 int보다 작거나 같고 long은 int보다 크거나 같은 것을 의미한다.
- **signed**는 [부호가 있는]을 의미하는 키워드인데, 정수형 short, int, long은 모두 양수, 0, 음수를 표현할 수 있으므로 signed 키워드는 정수형 자료형 앞에 표시될 수도 있고, 생략될 수도 있다.  
즉, signed int와 int는 같은 자료형임을 의미한다.
- **unsigned**는 [부호가 없는]을 의미하는 키워드로, 양수와 0을 처리하는 정수형 short, int, long은 앞에 unsigned를 표시해야 하며, 자료형 unsigned int에서 int는 생략 가능하다.  
마찬가지로, unsigned int와 unsigned는 같은 자료형임을 의미한다.
- 또한, **정수형 자료공간**도 키워드와 자료형에 따라 크기와 표현 범위가 각각 다르다.

음수 지원여부	자료형	크기	표현 범위
부호가 있는 정수형 signed	signed short	2 바이트	$-32,768(-2^{15}) \sim 32,767(2^{15}-1)$
	signed int	4 바이트	$-2,147,483,648(-2^{31}) \sim 2,147,483,648(2^{31}-1)$
	signed long	4 바이트	$-2,147,483,648(-2^{31}) \sim 2,147,483,648(2^{31}-1)$
부호가 없는 정수형 unsigned	unsigned short	2 바이트	$0 \sim 65,535(2^{16}-1)$
	unsigned int	4 바이트	$0 \sim 4,294,967,295(2^{32}-1)$
	unsigned long	4 바이트	$0 \sim 4,294,967,295(2^{32}-1)$

##### ◎ 부동소수 자료형

- 부동소수형(Floating Point Data Type)은 3.14, 3.26567과 같이 실수를 표현하는 자료형이며, 키워드는 float, double, long double 세 가지이다.
- 저장공간의 크기로는 float는 4바이트이며, double과 long double은 8바이트지만 long double형이 double형보다 표현 범위가 같거나 보다 정확하다. 또한, 소수 3.14와 같은 표현은 일반적으로 double형으로 인식하기 때문에 float형 변수에 저장할 경우 뒤에 F를 붙여 3.14F와 같이 상수로 저장해야만 한다.

자료형	크기	정수의 유효자릿수	표현범위
float	4 바이트	6~7	1.175494351E-38F에서 3.402823466E+38F까지
double	8 바이트	15~16	2.2250738585072014E-308에서 1.7976931348623185E+308까지
long double	8 바이트	15~16	2.2250738585072014E-308에서 1.7976931348623185E+308까지

##### ◎ 문자형 자료형

- 문자형 자료형은 char, signed char, unsigned char 세 가지 종류가 있으며, 저장공간 크기는 모두 1바이트이다.

자료형	저장공간 크기	표현범위
char	1 바이트	-128에서 127까지(문자는 실제로 0에서 127까지 이용)
signed char	1 바이트	-128에서 127까지
unsigned char	1 바이트	0에서 255까지

##### ◎ 오버플로와 언더플로

- 자료형의 범주에서 벗어난 값을 저장하면 오버플로(Overflow) 혹은 언더플로(Underflow)가 발생한다.  
오버플로란 자료형이 표현할수 있는 범위를 넘어서서 최대값 + 1이 저장된다면 범위의 최소값이 출력되는 현상을 말한다.  
언더플로란 자료형이 표현할수 있는 범위를 넘어서서 최소값 - 1이 저장된다면 범위의 최대값이 출력되는 현상을 말한다.  
이러한 특징을 정수의 순환이라고도 한다.

### 3. 자료형과 변수

#### 3.4 - 상수 표현방법

##### ◎ 상수의 개념과 표현방법

- 상수(Constant)란 이름이 없이 있는 그대로 표현한 자료값이거나 이름이 있으나 정해진 하나의 값만으로 사용되는 자료값을 말하며, 크게 **리터럴 상수(Literal Constant)**와 **심볼릭 상수(Symbolic Constant)**로 구분될 수 있다.
- 리터럴 상수란 숫자 32, 문자 \*, #, "Hello World"와 같이 소스에 그대로 표현해서 의미가 전달되는 자료값이고, 심볼릭 상수란  $PI = 3.141592$ 로 표현할 때의  $PI$ 처럼 이름을 갖는 상수를 말하며, 이러한 심볼릭 상수를 표현하는 방법은 `const` 상수, 매크로 상수, 열거형 상수를 이용하는 세 가지 방법이 있다.

구분	표현방법	설명	예
리터럴 상수	정수형, 실수형, 문자, 문자열 상수	다양한 상수를 있는 그대로 기술	32, 025, 0xf3, 3.2F, '\n' 등
심볼릭 상수 (이름이 있는 상수)	<code>const</code> 상수	키워드 <code>const</code> 를 이용한 변수 선언과 같으며, 수정할 수 없는 변수 이름으로 상수 정의	<code>const double PI = 3.141592;</code>
	매크로 상수	전처리기 명령어 <code>#define</code> 으로 다양한 형태를 정의	<code>#define PI 3.141592</code>
	열거형 상수	정수 상수 목록 정의	<code>enum bool {FALSE, TRUE};</code>

- 문자 상수는 문자 하나의 앞 뒤에 작은따옴표(')를 넣어 표현하며, `\ddd`, `\xab`와 같이 팔진수 코드값이나 십육진수 코드값으로 표현할 수도 있다.  
함수 `printf()`에서 문자 상수를 출력하려면 `%c`나 `%C`의 형식 제어문자가 포함되는 형식 제어문자열을 사용한다.  
(ex - `printf("%c", 'A');` / `printf("%C", '\n');`)

##### ◎ 심볼릭 상수 `Const`

- 심볼릭 상수 **`Const`**는 일반 변수처럼 고유한 이름이 있는 상수로서, 변수 선언에서 키워드 `const`를 자료형이나 변수 앞에 삽입하면 변수로는 선언되지만 초기값을 아무데서나 수정할 수 없는 것을 말한다.  
(ex - `const double RATE = 0.03;` / `RATE = 0.032;`(X, 컴파일 오류 발생))

##### ◎ 매크로 상수

- 전처리 지시자 `#define`은 매크로 상수를 정의하는 지시자로서, 지시자에 의한 심볼릭 상수를 주로 대문자 이름으로 정의하는데 이를 **매크로 상수**라고 한다.  
(ex - `#define KPOP 5000000` - 정수 매크로 상수 / `#define PI 3.14` - 실수 매크로 상수)

##### ◎ 열거형 상수

- 열거형이란 키워드 `enum`을 사용하여 정수형 상수 목록 집합을 정의하는 자료형이며, 열거형 상수의 정의는 `enum` 다음에 열거형태그명을 기술하고 중괄호를 사용하여 열거형 정수 상수 목록을 쉼표로 분리하여 기술한다.  
(ex - `enum DAY {SUN, MON, TUE, WED ...};`  
`enum 열거형태그명 {열거형상수1, 열거형상수2, ...};`)
- 열거형 상수에서 목록 첫 상수의 기본값은 0이며, 다음 상수부터 1씩 증가하는 방식으로 상수값이 자동으로 부여된다.  
(ex - `enum DAY {SUN, MON, TUE, WED ...};`  
                  1      2      3      4
- 상수 목록에 특정한 정수값을 부분적으로 직접 지정할 수도 있다.  
(ex - `enum SHAPE {POINT, LINE, TRI = 3, RECT, OCTA = 8, ...}`  
      = `POINT - 0, LINE - 1, TRI - 3, RECT - 4, OCTA - 8`)

## 4. 전처리와 입출력

### 4.1 - 전처리

#### ◎ 전처리 개요

- **전처리**란 C언어에서 컴파일러가 컴파일하기 전에 전처리의 전처리 과정을 하는 것을 의미하며, 이는 전처리 지시자인 `#include`로 헤더파일을 삽입하거나 `#define`에 의해 정의된 상수를 대체시키는 등의 전처리 출력파일을 만들어 컴파일러에게 보내는 작업을 칭한다.

※ 전처리 과정에서 처리되는 문장을 전처리 지시자(preprocess directives)라 한다.  
`#include`, `#define`과 같은 전처리 지시자는 항상 `#`으로 시작하고, 마지막에 `;`(세미콜론)이 없다.

#### ◎ 전처리 지시자 #include

- **헤더파일**이란 `printf()`, `scanf()`, `getchar()`와 같은 입출력 함수를 위한 자료형이나 함수원형이 정의된 텍스트 파일을 말한다. (ex - `#include <stdio.h>` <- 헤더파일)
- 헤더파일의 확장자는 `h`이며, 주요 헤더파일의 이름과 역할은 다음 표와 같다.

헤더 파일	파일 이름	파일 내용
<code>stdio.h</code>	STanDard Input Output(표준 입출력)	표준 입출력 함수와 상수
<code>stdlib.h</code>	STanDard LIBrary(표준 함수)	주요 메모리 할당 함수와 상수
<code>math.h</code>	math	수학 관련 함수와 상수
<code>string.h</code>	string	문자열 관련 함수와 상수
<code>time.h</code>	Time	시간 관련 함수와 상수
<code>ctype.h</code>	Character TYPE	문자 관련 함수와 상수
<code>limits.h</code>	limits	정수 상수 등 여러 상수
<code>float.h</code>	float	부동소수에 관련된 각종 상수

- **전처리 지시자 #include**는 헤더파일을 삽입하는 지시자이다.

#### ◎ 전처리 지시자 #define

- 전처리 지시자 `#define`은 매크로 상수를 정의하는 지시자로서, `#define`에 의한 심볼릭 상수는 주로 대문자 이름으로 정의하는데 이를 매크로 상수라고 한다.

※ 다음 `#define`에 정의된 `identifier_name`은 전처리기에 의해 모두 `value`로 대체되어 컴파일된다.

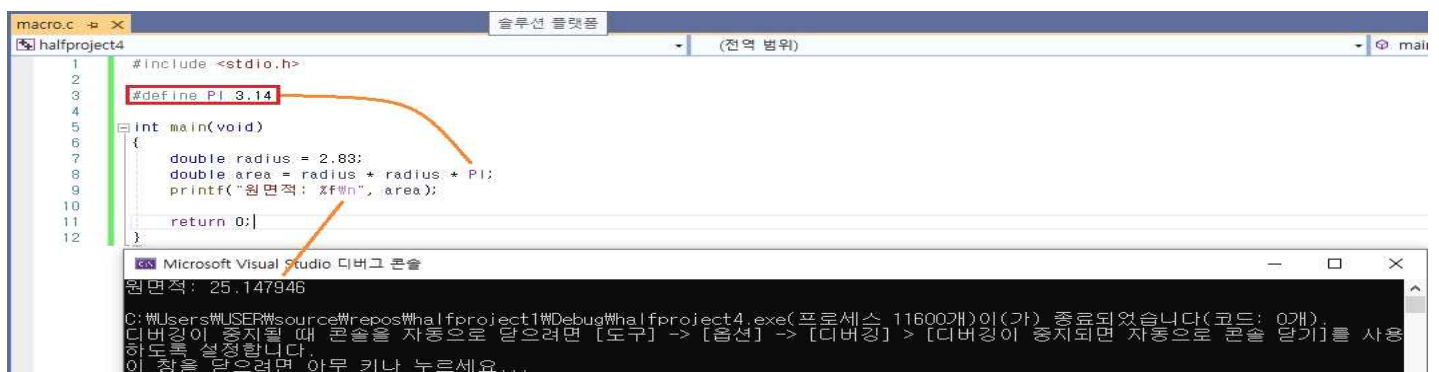
(ex - `#define identifier_name [value]`)

※ 다음 지시자 `#define`은 정수, 실수, 문자열 등의 상수를 `KPOP`, `PI`, `PRT`등의 이름으로 정의한다.

다음 소스에서 `PI`라는 매크로 상수는 전처리 과정에서 모두 `3.14`라는 실수로 값이 바뀐 소스로 컴파일된다.

단, 매크로 상수는 문자열 내부 또는 주석 부분에서는 대체되지 않는다.

(ex - `#define KPOP 50000000` -> 정수 매크로 상수 / `#define PI 3.14` -> 실수 매크로 상수 /  
`#define PRT printf("종료\n")` -> 문자열 매크로 상수)



- `#define`에서 기호 상수 뒤에 오는 치환문자열에는 일반 상수, 상수의 연산, 이미 정의된 기호상수 등이 올 수 있으나, 활용도를 높이기 위한 방안이 함수와 같이 인자(parameter)를 이용하는 방법이다.

(ex - `#define SQUARE(x) ((x) * (x))` -> `printf("%d, %d\n", SQUARE(2), SQUARE(3));`  
-> `printf("%d, %d\n", ((2) * (2)), ((3) * (3)));`)

## 4. 전처리와 입출력

### 4.2 - 출력 함수 printf()

#### ◎ 정수와 실수의 출력

- printf()는 일련의 문자 및 값으로 서식을 지정하여 표준 출력인 stdout에 출력하는 함수로서, printf()의 인자는 크게 형식문자열과 출력할 목록으로 구분되어 출력 목록의 각 항목을 형식문자열에서 %d와 같이 %로 시작하는 형식지정자 순서대로 서식화하여 그 위치에 출력한다.

(ex - int term = 15 / printf("%d의 두 배는 %d입니다.\n", term, 2\*term);)

형식 문자열

출력목록

- 함수 printf()의 첫 번째 인자인 형식문자열은 일반문자와 이스케이프 문자, 형식 지정자로 구성된다.

※ “\” 로 시작하는 문자는 이스케이프 문자이며, %d나 %s처럼 “%” 로 시작하는 문자는 형식지정자이다.

- 출력하는 문자열 내부에서 적당한 정수와 실수 문자열 등의 자료형을 적당한 위치에 출력하기 위해 형식 지정자와 이스케이프 문자가 필요하며, 형식 지정자는 정수, 실수, 문자등에 따라 다르다.

※ 형식 지정자는 출력 내용의 자료형에 따라 %d, %i, %c, %s와 같이 %로 시작한다.

두 번째 인자부터는 형식 지정자에 맞게 콘솔로 출력할 값이 표시되는 연산식 목록이 나열된다.

형식 지정자는 두 번째부터 표시된 인자인 연산식의 수와 값의 종류에 따라 순서대로 서로 일치해야 한다.

- 함수 printf()에서 정수의 10진수, 8진수, 16진수 출력을 위한 형식 지정자는 다음과 같다.

※ 정수의 십진수 출력을 위한 형식 지정자는 %d, %i이다. (ex - printf("%d %i\n", 16, 16);) => 16 16

정수의 8진수 출력을 위한 형식 지정자는 %o이며, 앞부분에 숫자 0이 붙는 출력을 하려면 #을 삽입한다.

(ex - printf("%o %#o\n", 16, 16);) => 20 020

정수의 16진수 출력을 위한 형식 지정자는 %x이며, 소문자로 출력할 경우 %x이고 대문자로 출력할 경우 %X이다.

마찬가지로 앞부분에 숫자 0이 붙는 출력을 하려면 #을 삽입한다. (ex - printf("%x %#x %#X\n", 10, 10, 10);) => a 0xa 0XA

- 함수 printf()에서 float와 double의 실수 출력을 위한 형식 지정자는 다음과 같다.

※ 실수의 간단한 출력을 위한 형식 지정자는 %f이며, %f는 실수를 기본적으로 소수점 6자리까지 출력한다. (ex - 3.400000)

함수 printf()에서 실수 출력 형식 지정자로 %f와 함께 %lf도 사용된다.

- 정수와 실수의 출력 폭은 기본적으로 시스템이 알아서 출력하지만, 개인의 기호에 맞게 출력을 정교하게 하기 위해 출력 폭(width)과 정렬(alignment)이 필요하다.

일반적으로 출력 필드 폭이 출력 내용의 폭보다 넓으면 정렬은 기본이 오른쪽이며, 필요하다면 왼쪽으로 지정할 수 있다.

				7	6	2	9
--	--	--	--	---	---	---	---

- 형식지정자 %8d를 십진수로 8자리 폭에 출력 -> printf("%8d\n", 7629);

7	6	2	9				
---	---	---	---	--	--	--	--

- 형식지정자 %8d를 십진수로 출력 폭을 지정하며 정렬을 오른쪽으로 출력 -> printf("%-8d\n", 7629);

				3	2	.	3	6	9
--	--	--	--	---	---	---	---	---	---

- 정수 32,369를 %10.3f로 정수로 출력(전체 폭 10, 소수점 이하 자릿수 3) -> printf("%10.3f\n", 32.369);

		3	2	.	3	6	9	0	0	0
--	--	---	---	---	---	---	---	---	---	---

- 정수 32,369를 %10f로 정수로 출력(소수점 이하 자릿수는 기본 6자리) -> printf("%10f\n", 32.369);

3	2	.	3	6	9	0	0	0
---	---	---	---	---	---	---	---	---

- 지정된 전체 폭이 출력값의 전체 폭보다 작으면, 지정된 작은 폭은 무시하고 원래 출력값의 폭으로 출력된다.  
-> printf("%5f\n", 32.369);

3	2	.	3	6	9				
---	---	---	---	---	---	--	--	--	--

- 지정된 폭이 출력값의 폭보다 넓으면 기본으로 오른쪽 정렬을 하며, 왼쪽 정렬을 하려면 %-10.3f와 같이 폭 앞에 -를 넣는다. -> printf("%-10.3f\n", 32.369);

			+	3	2	.	3	6	9
--	--	--	---	---	---	---	---	---	---

- %+10.3f와 같이 폭 앞에 +를 넣으면 양수라도 +부호가 표시된다. -> printf("%+10.3f\n", 32.369);

+	3	2	.	3	6	9			
---	---	---	---	---	---	---	--	--	--

- %+10.3f와 같이 폭 앞에 +와 -를 함께 붙이면(+-, -+ 모두 가능) 정렬은 오른쪽이며, 부호 +를 삽입한다.  
-> printf("%+-10.3f\n", 32.369);

### 4.3 - 입력 함수 scanf()

#### ◎ 함수 scanf()와 정수 입력

- **scanf()**는 대표적인 입력함수로서, 표준 입력으로부터 여러 종류의 자료값을 훑어 주소연산자 &가 붙은 변수 목록에 저장한다.
- \* 형식지정자(format specification)는 %d, %c, %lf와 같이 %로 시작한다.  
 함수 scanf()에서 첫 번째 인자는 형식문자열(format string)이라 하며, 형식지정자와 일반문자로 구성된다.  
 함수 scanf()에서 두 번째 인자부터는 키보드 입력값이 복사, 저장되는 입력변수 목록으로 변수 이름 앞에 반드시 주소연산자 &를 붙인다.  
 함수 scanf()의 반환 유형은 int이며, 표준입력으로 변수에 저장된 입력 개수를 반환한다. (ex - scanf("%d, %d, %d", &a, &b, &c);)
- scanf()로 int형 변수 year에 키보드 입력값을 저장하려면, scanf("%d, &year)와 같이 입력값이 저장되는 변수는 주소연산식인 &year로 사용해야 한다.
- \* '&' 는 주소연산자이고 뒤에 표시된 피연산자인 변수 주소값이 연산값으로, scanf()의 입력변수목록에는 키보드에 입력값이 저장되는 변수를 찾는다는 의미에서 반드시 변수의 주소연산식 '&변수이름'이 인자로 사용되어야 한다.  
 만일 주소연산이 아니라 변수 year로 기술하면 입력값이 저장될 주소를 찾지 못해 오류가 발생한다.

- scanf()는 실행 시에 지정된 형식지정자에 맞게 키보드로 적당한 값을 입력한 후 Enter 키를 누르기 전까지는 실행을 멈추고 사용자의 입력을 기다리며, 여러 입력값을 구분해주는 구분자(separator)로 -, /, 콤마(,) 등을 사용할 수 있다.

```

scanf.c
1 #define _CRT_SECURE_NO_WARNINGS // scanf()의 오류를 방지하기 위한 상수 정의
2
3 #include <stdio.h>
4
5 int main(void)
6 {
7     int year = 0;
8     printf("당신의 입학년도는? ");
9     scanf("%d", &year);
10    printf("입학년도: %d\n", year);
11
12    int month, day;
13    printf("당신의 생년월일은? ");
14    scanf("%d - %d - %d", &year, &month, &day);
15    printf("생년월일: %d-%d-%d\n", year, month, day);
16
17    return 0;
18 }
    
```

Microsoft Visual Studio 디버그 콘솔

```

당신의 입학년도는? 2019
입학년도: 2019
당신의 생년월일은? 1990-06-12
생년월일: 1990-6-12
    
```

- 입력 scanf()에서는 저장될 자료형이 float이면 %f, double이면 %lf로 구분하여 사용해야 하며, 문자 char 형이면 %c를 사용한다.
- 함수 scanf()에서 정수의 콘솔입력값을 8진수로 인지하려면 %0을 사용하고 16진수로 인지하려면 %x를 사용하며, 이외에도 scanf()에서 다양하게 이용되는 형식 지정자는 다음 표와 같다.

형식 지정자	콘솔 입력값의 형태	입력 변수 인자 유형
%d	십진수로 인식	정수형 int 변수에 입력값 저장
%i	십진수로 인식하지만, 입력값 앞에 0이 붙으면 8진수 / 0x가 붙으면 16진수로 인식하여 저장된다.	정수형 int 변수에 입력값 저장
%u	unsigned int로 인식	정수형 unsigned int 변수에 입력값 저장
%o	8진수로 인식	정수형 int 변수에 입력값 저장
%x, %X	16진수로 인식	정수형 int 변수에 입력값 저장
%f	부동소수로 인식	부동소수형 float 변수에 입력값 저장
%lf	부동소수로 인식	부동소수형 double 변수에 입력값 저장
%e, %E	지수 형태의 부동 소수로 인식	부동소수형 float 변수에 입력값 저장
%c	문자로 인식	문자형 char 변수에 입력값 저장
%s	문자열(string)로 인식	문자열을 저장할 배열에 입력값 저장
%p	주소(address) 값으로 인식	정수형 unsigned int 변수에 입력값 저장

- 함수 getchar()는 문자 하나를 입력하는 매크로 함수이고, putchar()는 출력하기 위한 함수이다.  
 이 함수를 이용하려면 헤더파일 <stdio.h>가 필요하다.



## 5. 연산자

### 5.1 - 연산식과 다양한 연산자

#### ◎ 연산식과 연산자 분류

- 변수와 다양한 리터럴 상수, 함수의 호출 등으로 구성되는 식을 연산식(expression)이라 하며, 연산식은 반드시 하나의 결과값인 연산값을 갖는다.

연산자(operator)는 산술연산자 +, -, \* 기호와 같이 이미 정의된 연산을 수행하는 문자 또는 문자조합기호를 말하며, 연산에 참여하는 변수나 상수를 피연산자(operand)라 한다.

※  $3 + 4 = 7 \rightarrow$  (빨간색 - 피연산자 / 검정색 - 연산자 / 보라색 - 연산값)

- 연산자는 연산에 참여하는 피연산자의 개수에 따라 단항(unary), 이항(binary), 삼항(ternary) 연산자로 나눌 수 있고 삼항연산자는 조건연산자 '? :'가 유일하다.  
또한, ++a처럼 연산자가 앞에 있으면 전위(prefix) 연산자이며, a++같이 연산자가 뒤에 있으면 후위(postfix) 연산자라고 한다.

- 산술연산자는 +, -, \*, /, %로써 각각 더하기, 빼기, 곱하기, 나누기, 나머지 연산자이다.

+, -, \*, /의 피연산자는 정수형 또는 실수형이 가능하지만, % 나머지 연산자는 피연산자로 정수형만 가능하다.

※ 정수끼리의 나누기 연산(/) 결과는 소수 부분을 버린 정수이지만, 실수끼리의 나누기 연산  $10.0 / 4.0$ 의 결과는 2.5이다.

- 나머지 연산식  $a \% b$ 의 결과는 a를 b로 나눈 나머지 값이며, 피연산자는 반드시 정수여야만 한다.

※ (ex -  $10 \% 3 = 1, 10 / 3 = 3$ )

- 연산자 \*, /, %는 +, - 보다 먼저 계산된다. (ex -  $(3 / 2) * 2 \rightarrow 2$  (O),  $3 / (2 * 2)$  (X))

- 대입연산자는 '='로 연산자 오른쪽의 연산값을 변수에 저장하는 연산자이며, 대입연산자의 왼쪽 부분에는 반드시 하나의 변수만이 올 수 있다. 이 하나의 변수를 왼쪽을 의미하는 left 단어에서 파생된 l-value라 하며, 오른쪽에 위치하는 연산식의 값을 오른쪽을 의미하는 right 단어에서 파생된 r-value라 한다.

※ var(변수) = exp(결과값)  $\rightarrow n = 2; / n = n + 1; / n = n + 3*4; \rightarrow n = 15$

- 대입연산식  $a = a+b$ 는 중복된 a를 생략하고 간결하게  $a+=b$ 로 쓸 수 있다. 이와 같이 산술연산자와 대입연산자를 이어 붙인 연산자 +=, -=, \*= 등을 축약 대입연산자라 한다. ※ (ex -  $a += 2 \rightarrow a = a + 2$ )

- 증가연산자 ++와 감소연산자 --는 변수값을 각각 1 증가시키거나, 1 감소시키는 기능을 수행한다.

증가연산자에서 n++와 같이 연산자 ++가 피연산자 n보다 뒤에 위치하는 후위이면 1 증가되기 전 값이 연산 결과값이며, ++n같이 전위이면 1 증가된 값이 연산 결과값이다. 감소 연산자도 마찬가지로 결과값이 산출된다.

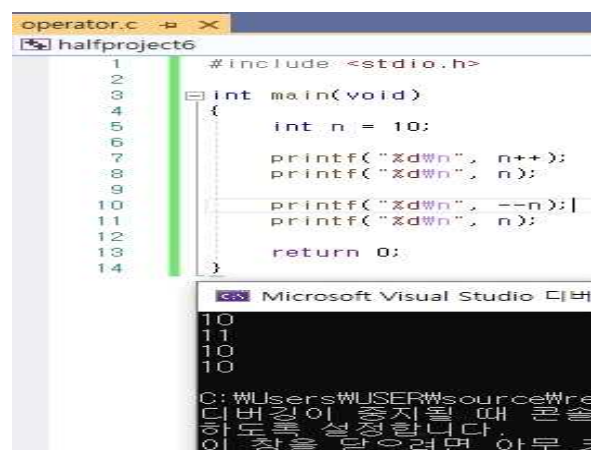
- 증감연산자는 변수만을 피연산자로 사용할 수 있으며, 상수나 일반 수식을 피연산자로 사용할 수 없다.

※ `int a = 10;`

`++300; // 상수에는 증감연산자 사용 불가`

`(a+1)--; // 일반 수식에는 증감연산자 사용 불가`

`a = ++a * a--;` // 하나의 연산식에 동일한 변수의 증감연산자는 되도록 사용하지 않는 것이 좋다.



```
operator.c
halfproject6
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int n = 10;
6
7     printf("%d\n", n++);
8     printf("%d\n", --n);
9     printf("%d\n", n);
10    return 0;
11
12
13
14
```

Microsoft Visual Studio 디버거 콘솔 출력:

```
10
11
10
10
```

## 5.2 - 관계와 논리, 조건과 비트연산자

## ◎ 관계와 논리연산자

- **관계연산자**는 두 피연산자의 크기를 비교하기 위한 연산자로, 관계연산자의 연산값은 비교 결과가 참이면 1, 거짓이면 0이다.

※ 관계연산자 !=, >=, <=는 연산 기호의 순서가 명확해야 한다.

관계연산자 ==는 피 연산자 두 값이 같은지를 알아보는 연산자로, 대입연산자 =와 혼동하지 않도록 주의해야 한다.

관계연산자에서 피연산자는 정수형, 실수형, 문자형 등이 피연산자가 될 수 있다.

피연산자가 문자인 경우, 문자 코드값(아스키 코드)에 대한 비교의 결과이다. (ex - 'a' = 97 / 'Z' = 90 -> ('Z' < 'a') = 참)

연산자	연산식	의미	예제	연산값
>	$x > y$	x가 y보다 큰가?	$3 > 5$	0(거짓)
>=	$x \geq y$	x가 y보다 크거나 같은가?	$5-4 \geq 0$	1(참)
<	$x < y$	x가 y보다 작은가?	'a' < 'b'	1(참)
<=	$x \leq y$	x가 y보다 작거나 같은가?	$3.43 \leq 5.862$	1(참)
!=	$x \neq y$	x와 y가 다른가?	$5-4 \neq 3/2$	0(거짓)
==	$x == y$	x와 y의 값이 같은가?	'%' == 'A'	0(거짓)

- 논리연산자 &&, ||, !은 각각 and, or, not의 논리연산을 의미하며, 결과가 참이면 1을, 거짓이면 0을 반환한다.  
C언어에서 참과 거짓의 논리형은 따로 없으므로, 0, 0.0, \0은 거짓을 의미하며, 0이 아닌 모든 정수와 실수, NULL문자 '\0'가 아닌 모든 문자와 문자열은 모두 참을 의미한다.

※ 논리연산자 && - 두 피연산자가 모두 참이면 결과도 참이며, 나머지 경우는 모두 거짓이다.

논리연산자 || - 두 피연산자 중에서 하나만 참이면 1이고, 모두 거짓이면 0이다.

논리연산자 ! - 단항연산자로서, 피연산자가 0이면 결과는 1이고, 피연산자가 1이면 결과는 0이다.

x	y	$x \&\& y$	$x \parallel y$	!x
0(거짓)	0(거짓)	0	0	1
0(거짓)	1(참)	0	1	1
1(참)	0(거짓)	0	1	0
1(참)	1(참)	1	1	0

- 논리연산자 &&와 ||는 피연산자 두 개 중에서 왼쪽 피연산자만으로 논리연산 결과가 결정된다면 오른쪽 피연산자는 평가하지 않는데, 이런 방식을 단축평가라고 한다.

※  $(x \&\& y) \rightarrow x = 0 \rightarrow y$ 의 값을 몰라도  $x\&\&y = 0$  /  $(x \parallel y) \rightarrow x$ 가 0이 아니면  $y$ 의 값을 몰라도  $(x \parallel y) = 1$

- **조건연산자**란 조건에 따라 주어진 피연산자가 결과값이 되는 삼항연산자이다.  
즉, 연산식  $(x ? a : b)$ 에서 피연산자는 x, a, b 3개이며 x가 1이면 결과는 a이고 x가 0이면 결과는 b이다.

- **비트연산자**란 정수의 비트 중심 연산자로 비트 논리 연산자와 비트 이동 연산자를 제공한다.

- **비트 논리 연산자**란 피연산자 정수값을 비트 단위로 논리 연산을 수행하는 연산자로, &, |, ^, ~ 4가지로 구분된다.

연산자	연산자 이름	사용	의미
&	비트 AND	$op1 \& op2$	비트가 모두 1이면 결과는 1, 아니면 0
	비트 OR	$op1   op2$	비트가 적어도 하나가 1이면 결과는 1, 아니면 0
^	비트 배타적 OR(XOR)	$op1 \wedge op2$	비트가 서로 다르면 결과는 1, 아니면 0
~	비트 NOT 또는 보수	$\sim op1$	비트가 0이면 결과는 1, 1이면 0

- **비트 이동 연산자**란 연산자의 방향인 왼쪽이나 오른쪽으로, 비트 단위로 줄줄이 이동시키는 연산자를 말하며,  
>>(왼쪽->오른쪽)일 때는 원래 비트와 동일한 값이 채워지고 <<(오른쪽->왼쪽)일 때는 무조건 0의 값이 채워진다.

(ex -  $120 \gg 3 \rightarrow$ 

0	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---

 $\rightarrow$ 

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 = 15

$25 \ll 2 \rightarrow$ 

0	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---

 $\rightarrow$ 

0	1	1	0	0	1	0	0
---	---	---	---	---	---	---	---

 = 100



## 5. 연산자

### 5.3 - 형변환 연산자와 연산자 우선순위

#### ◎ 내림변환과 올림변환

- 자료형 char와 int는 각각 문자와 정수를 표현하고, 크기도 1바이트와 4바이트로 다르다.  
이러한 자료형들을 필요에 따라 자료의 표현방식을 바꾸는 것을 자료형 변환이라 하며, 크기 자료형의 범주 변화에 따른 구분으로 **내림변환**과 **올림변환**으로 나눌 수 있으며, 다른 자료형 구분 방식으로 명시적 형변환과 묵시적 형변환으로 나뉜다.
- \* **내림변환** : 큰 범주의 자료형(double)에서 작은 범주의 자료형(int)로의 형변환 방식  
**올림변환** : 작은 범주의 자료형(int)에서 큰 범주의 자료형(double)로의 형변환 방식  
**명시적(강제) 형변환** : 소스에서 직접 형변환 연산자를 사용하는 방식  
**묵시적 형변환** : 컴파일러가 알아서 자동으로 수행하는 방식
- **내림변환**이란 큰 범주의 자료형에서 작은 범주의 자료형으로 변환되는 방식으로, int a = 3.4일 경우 저장값과 자료 유형이 다르기 때문에 변수의 자료형으로 변환되어 int a = 3으로 저장됨과 동시에 정보의 손실이 발생할 수 있기 때문에 경고를 발생한다.
- **올림변환**이란 작은 범주의 자료형에서 큰 범주의 자료형으로 변환되는 방식으로, 7 + 5.2일 경우 int형 7이 자동으로 7.0으로 변환되어 7.0 + 5.2가 되며, 정보의 손실이 없으므로 컴파일러에 의해 자동으로 수행될 수 있으며 이를 **묵시적 형변환**(Implicit Type Conversion)이라고도 한다.
- **명시적 형변환**(Explicit Type Conversion)이란 형변환 연산자(type)를 사용하여 뒤에 나오는 피연산자의 값을 괄호에서 지정한 자료형으로 변환하는 연산자를 사용하는 방식을 말한다.
- \* (int) 30.525 -> 30 / (int) 3.14 -> 3 / (double) 9 -> 9.0 / (double) 3.4F -> 3.4 등

#### ◎ 연산자 sizeof와 콤마연산자

- 연산자 **sizeof**는 연산값 또는 자료형의 저장장소의 크기를 구하는 연산자이며, 결과값은 바이트 단위의 정수이다.  
피연산자가 int와 같은 자료형인 경우 반드시 괄호를 사용해야 하지만, 피연산자가 상수나 변수, 연산식인 경우에는 괄호를 생략할 수 있다.
- \* **sizeof (int)** - int가 4바이트이므로 결과값 4 / **sizeof (double)** - double이 8바이트이므로 결과값 8 / **sizeof a** = 결과값 2(short a);
- 콤마연산자 ,는 왼쪽과 오른쪽 연산식을 각각 순차적으로 계산하며, 결과값은 가장 오른쪽에서 수행한 연산의 결과이다. 또한, 콤마연산자는 연산자 우선순위가 가장 늦기 때문에 대입연산자 '='보다 나중에 계산된다.
- \* **x = 3+4, 2\*3; -> (x = 3+4), 2\*3; = x**에는 7 저장, 콤마연산 결과값 6 / **x = (3+4, 2\*3); -> 콤마연산의 결과값인 6이 x에 저장**

#### ◎ 복잡한 표현식의 계산

- 수식의 계산을 하는데 있어서, 다음 3개의 규칙에 따라 연산의 우선순위와 결합성을 적용해야 계산이 가능하다.
- \* **첫 번째 규칙** - 괄호가 있으면 먼저 계산한다. / **두 번째 규칙** - 연산의 우선순위 규칙을 적용한다. / **세 번째 규칙** - 동일한 우선순위인 경우 결합성을 따른다.

- 연산자 우선순위는 다음과 같다.

괄호와 대괄호>단항>산술>관계>논리>조건(삼항)>대입>콤마
<b>이항연산자</b>

변수값	표현식	설명	해석	결과
x=3	x>>1+1>1&y	산술>이동>관계>비트	((x>>(1+1))>1)&y	0
	x-3    y&2	산술>비트>논리	(x-3)    (y&2)	1
y=3	x&y && y>=4	관계>비트>논리	(x&y) && (y>=4)	0
	x && x   y++	증가>비트>논리	x && (x   (y++))	1

#### ◎ 결합성

- 연산자의 결합성은 대부분 좌에서 우(->)로 수행하나 우선순위가 2위인 **전위의 단항 연산자**, 우선순위 14위인 **조건 연산자**, 우선순위 15위인 **대입연산자**에서는 우에서 좌(<-)로 수행한다.

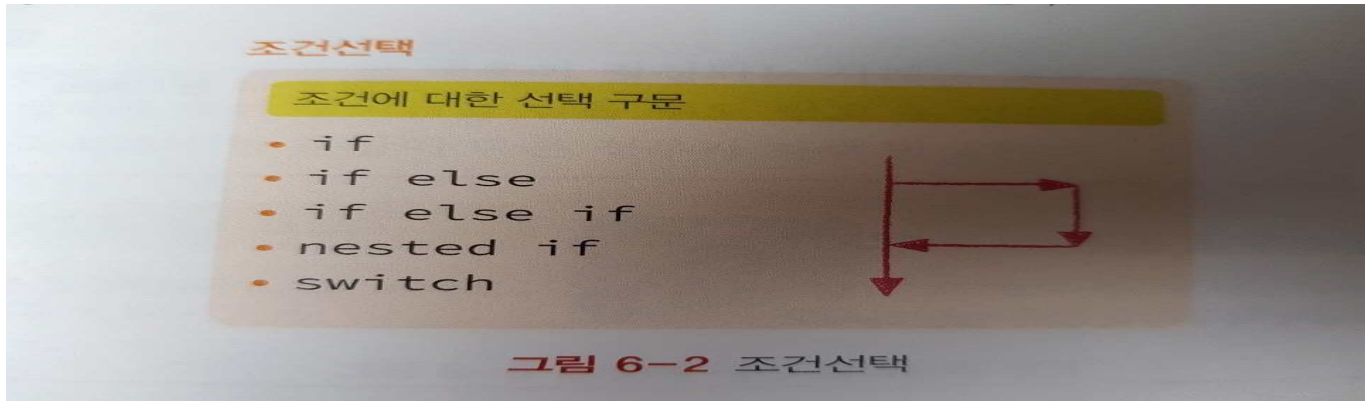
int n = 10, m = 5; n += m /= 3;	n +=	m /= 3
		① m = m/3의 결과 1
		② n = n+1의 결과 11

## 6. 조건

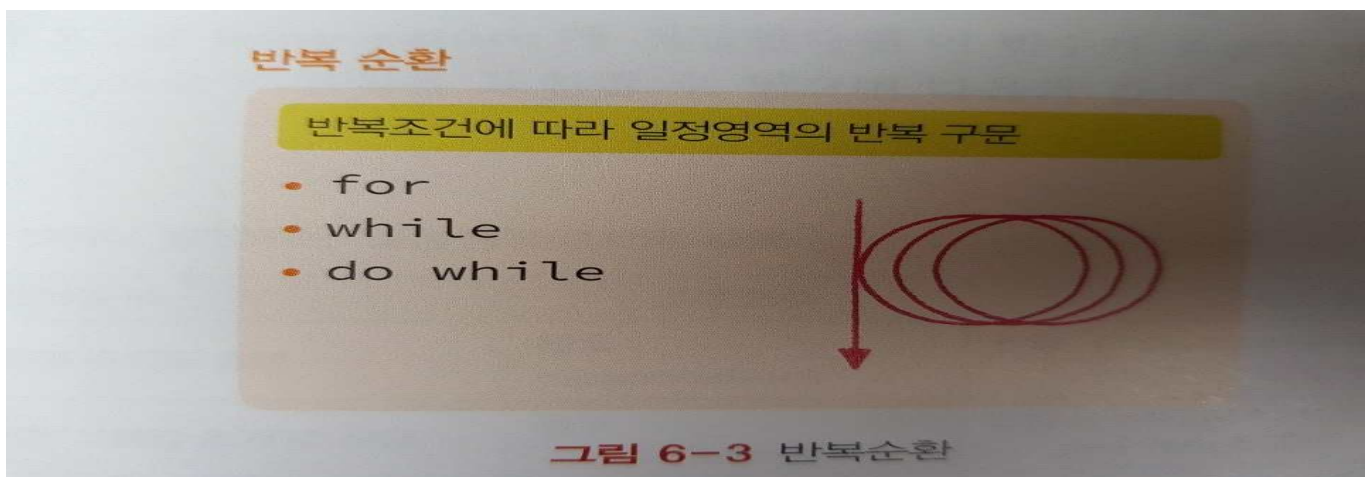
### 6.1 - 제어문 개요

#### ◎ 제어문의 종류

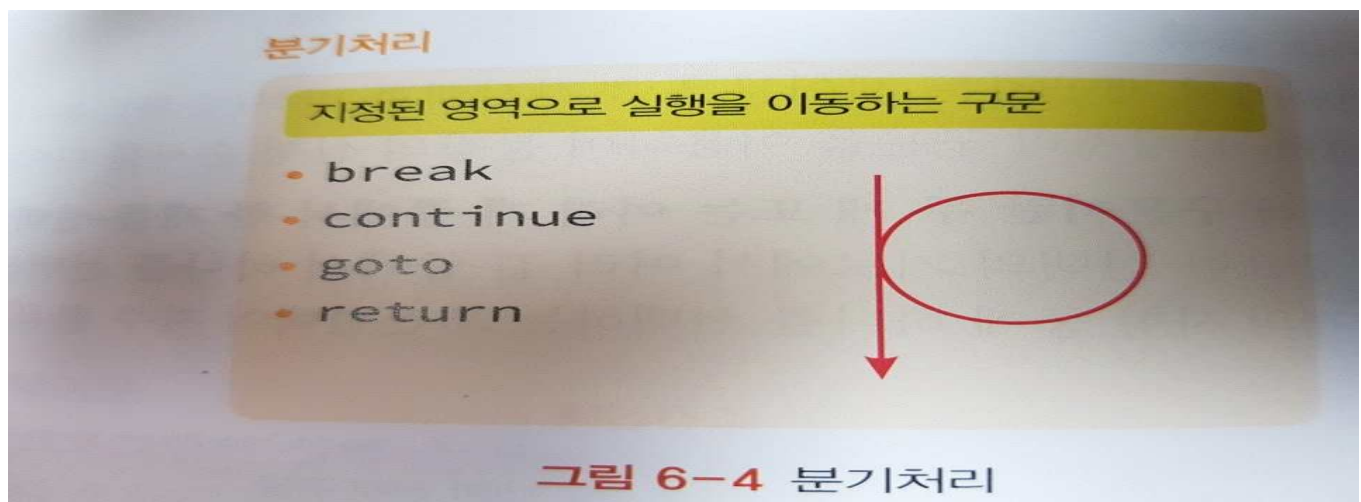
- 제어문이란 프로그램의 실행 흐름에서 순차적인 실행뿐만 아니라 선택과 반복 등 순차적인 실행을 변형하여 프로그램의 실행 순서를 제어하는 것을 뜻하며, 제어문의 종류로는 조건선택, 반복, 분기처리 등으로 나눌 수 있다.
- 조건선택 구문이란 두 개 또는 여러 개 중에서 한 개를 선택하도록 지원하는 구문이다.



- 반복(순환) 구문이란 정해진 횟수 또는 조건을 만족하면 정해진 몇 개의 문장을 여러 번 실행하는 구문이다.



- 분기 구문은 작업을 수행 도중 조건에 따라 반복이나 선택을 빠져나가거나(break;), 일정 구문을 실행하지 않고 다음 반복을 실행하거나(continue), 지정된 위치로 이동하거나(goto), 작업 수행을 마치고 이전 위치로 돌아가는(return) 구문이다.



## 6. 조건

### 6.2 - 조건에 따른 선택 if문

#### ◎ 조건에 따른 선택 개요

- 프로그램에서도 조건에 따라 처리해야 할 문장이 다른 경우가 자주 발생하며, 이를 일상생활에서의 사례로 대입해 보면 프로그램에서의 조건에 따른 선택이 발생하는 것을 이해할 수 있다.

조건 선택의 예	기준변수	조건 표현의 의사코드
온도가 32도 이상이면 폭염 주의를 출력	온도(temperature)	만일 (temperature >= 32) -> printf("폭염 주의");
낮은 혈압이 100이상이면 '고혈압 초기'로 진단	혈압(low_pressure)	만일 (low_pressure >= 100) -> printf("고혈압 초기");
속도가 40km와 60km 사이이면 "적정 속도"라고 출력	속도(speed)	만일 (40 <= speed && speed <= 60) printf("적정 속도");
운전면허 필기시험에서 60점 이상이면 합격, 아니면 불합격 출력	시험 성적(point)	만일 (point >= 60) -> printf("면허시험 합격"); 아니면 printf("면허시험 불합격");

#### ◎ If 문장

- 문장 if는 위에서 살펴본 조건에 따른 선택을 지원하는 구문이며, 가장 간단한 if문의 형태는 if (cond) stmt;이다. if문에서 조건식 cond가 0이 아니면(참) stmt를 실행하고, 0이면(거짓) stmt를 실행하지 않는다.
- 문장 if의 조건식 (cond)는 반드시 괄호가 필요하며, 결과값이 참이면 실행되는 문장은 반드시 들여쓰기를 해야한다.

```

1 #define _CRT_SECURE_NO_WARNINGS
2 #include <stdio.h>
3
4 int main(void)
5 {
6     double temperature;
7
8     printf("현재 온도 입력: ");
9     scanf("%lf", &temperature);
10
11     if (temperature >= 32.0)
12     {
13         printf("폭염 주의를 발령합니다.\n");
14     }
15     printf("현재 온도는 섭씨 %.2f 입니다.\n", temperature);
16
17     return 0;
18 }

```

Microsoft Visual Studio 디버그 콘솔

```

현재 온도 입력: 34
폭염 주의를 발령합니다.
현재 온도는 섭씨 34.00 입니다.
C:\Users\WUSER\source\repos\halfproject7\Debug\halfproject7.exe
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] ->
이 창을 닫으려면 아무 키나 누르세요...

```

#### ◎ If else 문장

- if문은 조건이 만족되면 특정한 문장을 실행하는 구문인듯이, 조건이 만족되지 않은 경우에 실행할 문장이 있다면 **else**를 사용한다. 조건문 if (cond) stmt1; else stmt2; 는 조건 cond를 만족하면 stmt1을 실행하고, 조건 cond를 만족하지 않으면 stmt2를 실행하는 문장이며, 결론적으로 조건문 if else는 stmt1과 stmt2 둘 중 하나를 선택하는 구문이라고 할 수 있다.

if (cond)	if (n % 2 == 0)
stmt1;	printf("짝수");
else	else
stmt2	printf("홀수");
next;	printf("입니다.\n");

※ (조건식)은 괄호가 필요하다.

조건식에서 등호를 대입으로 잘못 쓰는 것에 주의가 필요하다. (ex - (n == 100)을 (n = 100)으로 잘못 쓰면 항상 참으로 인식)

if (조건식); 이나 else; 와 같이 필요없는 곳에 세미콜론을 넣지 않는다.

조건식이 참이면 실행되는 stmt1이나 거짓이면 실행되는 stmt2 부분이 여러 문장이면 [여러문장들]의 블록으로 구성한다.

#### ◎ 반복된 조건에 따른 선택 if else if

- 조건문 if else문에서 else 이후에 **if else**를 필요한 횟수만큼 반복할 수 있다. 다음과 같이, cond1 조건식이 참이면 바로 아래의 문장 stmt1을 실행하고 종료되며, 거짓이면 다음 조건식 con2로 계속 이어간다. 조건식이 모두 만족되지 않을 경우 마지막 else 다음 문장인 stmt4를 실행하며, stmt1 ~ stmt4에 이르는 여러 문장 중 실행되는 문장은 단 하나이다.

if (cond 1)	stmt1;
else if (cond 2)	stmt2;
else if (cond 3)	stmt3;
else	stmt4;
next;	

## 6. 조건

### 6.2 - 조건에 따른 선택 if문 - 2

#### ◎ 중첩된 if

- if 문 내부에 또 다른 if문이 존재하면 중첩된 if문이라 한다.

```

1 #define _CRT_SECURE_NO_WARNINGS
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int type, point;
7
8     printf("번호를 선택: 1(1종면허), 2(2종면허): ");
9     scanf("%d", &type);
10    printf("필기시험 점수 입력: ");
11    scanf("%d", &point);
12
13    if (type == 1)
14    {
15        if (point >= 70)
16            printf("1종면허 합격\n");
17        else
18            printf("1종면허 불합격\n");
19    }
20    else if (type == 2)
21    {
22        if (point >= 60)
23            printf("2종면허 합격\n");
24        else
25            printf("2종면허 불합격\n");
26    }
27
28    return 0;
29 }
30

```

Microsoft Visual Studio 디버그 콘솔

```

번호를 선택: 1(1종면허), 2(2종면허): 1
필기시험 점수 입력: 70
1종면허 합격
C:\Users\WUSER\source\repos\half project\half project>
디버깅이 중지될 때 콘솔을 자동으로 닫으
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...

```

#### ◎ 다양한 if문의 이용과 조건연산자

- 조건에 따라 해야 할 내용이 결정되는 사례는 다음과 같으며, 이러한 사례들에서 가장 적절한 if문을 선택하여 구현해야 한다.

조건 선택의 예	if 형태	기준변수	조건 표현의 의사코드
온도가 32도 이상이면 폭염 주의를 출력	if	온도(tempe- rature)	만일 (temperature >= 32) -> printf("폭염 주의");
속도가 40km와 60km 사이이면 "적정 속도"라고 출력	if	속도(speed)	만일 (40 <= speed && speed <= 60) printf("적정 속도");
운전면허 필기시험에서 60점 이상이면 합격, 아니면 불합격 출력	if else	시험 성적(point)	만일 (point >= 60) -> printf("면허시험 합격"); 아니면 printf("면허시험 불합격");

- 또한, 조건연산자의 기능은 if문으로도 가능하지만, 간단한 조건연산자로 구현할 수 있는 기능은 조건연산자를 사용하는 것이 더 효율적이다.

구현 내용	조건연산자	if
두 수의 최대값 구하기	max = x > y ? x : y;	if ( x > y) max = x; else max = y;
두 수의 최소값 구하기	min = x > y ? y : x;	if ( x > y) min = y; else min = x;
절대값 구하기	abs = x >= 0 ? x : -x;	if ( x >= 0) abs = x; else abs = -x;
홀수와 짝수 구하기	a % 2 ? printf("홀수") : printf("짝수");	if ( a % 2) printf("홀수"); else printf("짝수");



## 6. 조건

### 6.3 - 다양한 선택 switch 문

#### ◎ switch 문장 개요

- **switch문**이란 주어진 연산식이 문자형 또는 정수형이라면 그 값에 따라 case의 상수값과 일치하는 부분의 문장들을 수행하는 선택구문이다.

예를 들어, `switch(exp) {...}` 문은 표현식 `exp` 결과값 중에서 case의 값과 일치하는 항목의 문장 `stmt1`을 실행한 후 `break`를 만나 종료되며, 여기서 `switch`, `case`, `break`, `default`는 키워드이고 연산식 `exp`의 결과값은 반드시 문자 또는 정수여야 한다.

또한, case 다음의 value 값은 변수가 올 수 없으며 상수식으로 그 결과가 정수 또는 문자 상수여야 하고 중복될 수 없으며, `default`는 선택적이므로 사용하지 않을 수 있다.

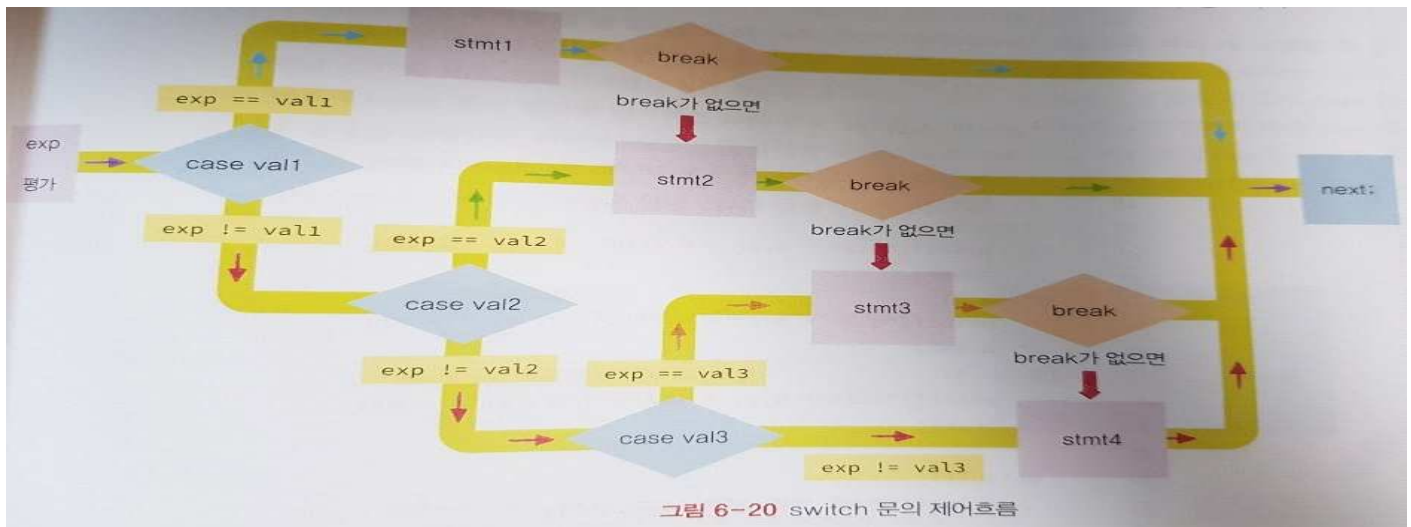
※ **switch문의 실행 순서** : 표현식 `exp`를 평가하여 그 값과 일치하는 상수값을 갖는 case 값을 찾아 case 내부의 문장을 실행한다.

`break`를 만나면 switch 문을 종료하며, case 문의 내부 문장을 실행하고 `break` 문이 없을 경우 `break`문을 만나기 전까지 다음 case의 내부로 무조건 이동하여 내부 문장을 실행한다.

일치된 case 값을 만나지 못하여 `default`를 만나면 `default` 내부의 문장을 실행한다.

`switch` 몸체의 마지막 문장을 실행하면 `switch` 문은 종료된다.

`default`의 위치는 모든 case 뒤에 오는 것이 일반적이지만, 중간에 위치하면서 `break` 문이 없다면 하부 case 문장을 모두 실행하므로 주의가 필요한 부분이다.



- switch문에서 주의할 것 중 하나는 case 이후 정수 상수를 콤마로 구분하여 여러 개 나열할 수 없다는 것이다. case 문 내부에 `break` 문이 없다면 일치하는 case 문을 실행하고, `break` 문을 만나기 전까지 다음 case 내부 문장을 실행하지만, `case 4 : case 5 :` 는 가능해도 `case4, 5` 와 같은 나열은 문법오류가 발생한다.

```
scoreswitch.c  half project9  (전역 범위)  Microsoft Visual Studio 디버그 콘솔

1  #define _CRT_SECURE_NO_WARNINGS
2  #include <stdio.h>
3
4  int main(void)
5  {
6      int score;
7
8      printf("점수 입력: ");
9      scanf("%d", &score);
10
11      switch (score / 10) {
12          case 10: case 9:
13              printf("점수가 %d 점으로 성적이 %c 입니다.\n", score, 'A');
14              break;
15          case 8:
16              printf("점수가 %d 점으로 성적이 %c 입니다.\n", score, 'B');
17              break;
18          case 7:
19              printf("점수가 %d 점으로 성적이 %c 입니다.\n", score, 'C');
20              break;
21          case 6:
22              printf("점수가 %d 점으로 성적이 %c 입니다.\n", score, 'D');
23              break;
24          default:
25              printf("점수가 %d 점으로 성적이 %c 입니다.\n", score, 'F');
26          }
27
28      return 0;
29  }
```

점수 입력: 80  
점수가 80 점으로 성적이 B 입니다.  
C:\Users\USER\source\repos\half pr...  
디버깅이 중지될 때 콘솔을 자동으로  
하도를 설정합니다.  
이 창을 닫으려면 아무 키나 누르세요

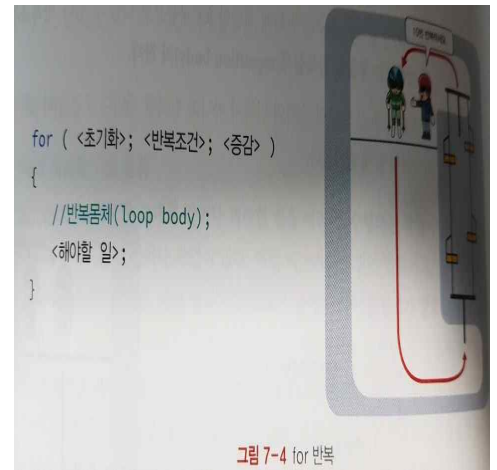
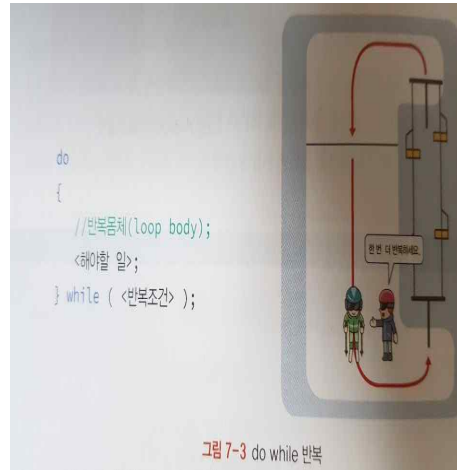
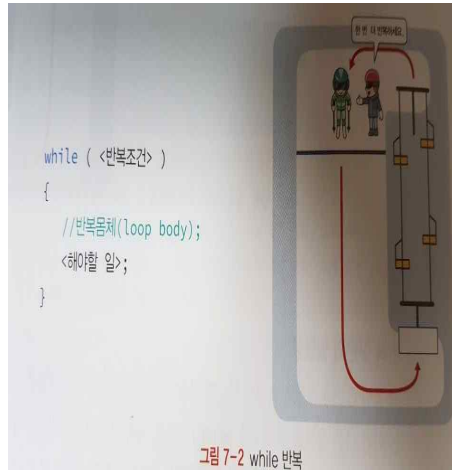
## 7. 반복

### 7.1 - 반복 개요와 while 문

#### ◎ 반복 개요

- 반복(repetition)은 말 그대로 같거나 비슷한 일을 여러 번 수행하는 작업이며, 반복과 같은 의미로 순환(loop)이라는 표현도 함께 사용한다.

C 언어에서는 while, do while, for 세 가지 종류의 반복 구문을 제공하며, 반복조건을 만족하면 일정하게 반복되는 부분을 반복몸체(repetition body)라 한다.



반복문 while	반복문 do while	반복문 for
반복할 때마다 조건을 따지는 반복문으로, 조건식이 반복몸체 앞에 위치.	무조건 한 번 실행한 후 조건을 검사하고, 조건식이 참이 아닐 경우 반복을 더 실행하며 조건식이 반복몸체 뒤에 위치.	숫자로 반복하는 횟수를 제어하는 반복문이며, 명시적으로 반복 횟수를 결정할 때 사용.

#### ◎ while 문장

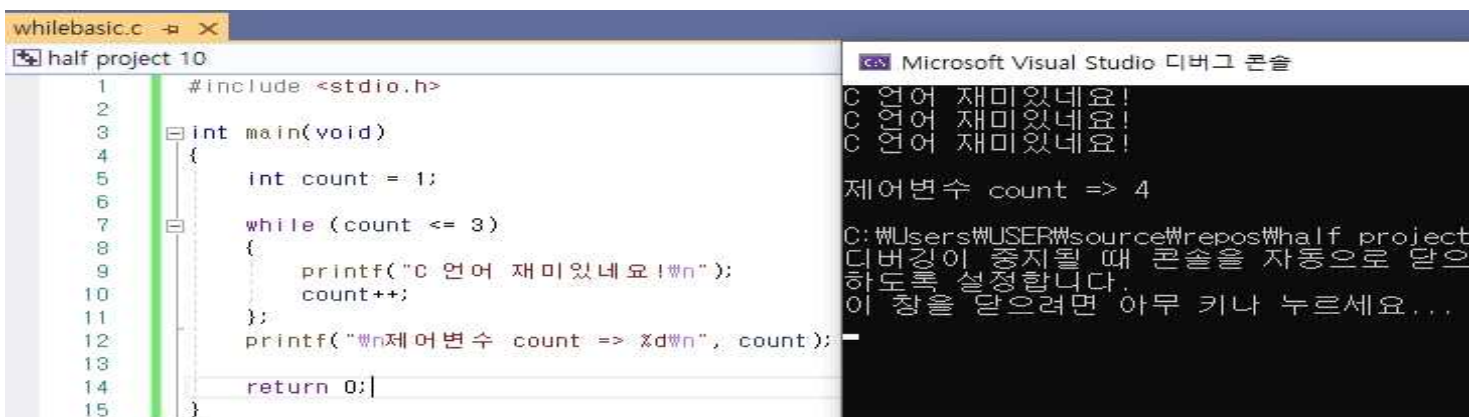
- while문은 숫자의 반복이 아니라, 반복할 때마다 조건을 따지는 반복문이며, 조건식을 만족할 때 반복이 실행된다.

※ while문의 반복은 cond가 0(거짓)이 될 때 계속된다.

반복이 실행되는 stmt를 반복 몸체라 부르며, 필요하면 블록으로 구성될 수 있다.

while 문은 for나 do while 반복문보다 간단하며 모든 반복 기능을 수행할 수 있다.

반복횟수	변수 count 값	조건식	조건식 평가	반복몸체
1	1	count <= 3 1 <= 3	while (1)	printf("C 언어 재미있네요!\n"); count++;
2	2	count <= 3 2 <= 3	while (1)	printf("C 언어 재미있네요!\n"); count++;
3	3	count <= 3 3 <= 3	while (1)	printf("C 언어 재미있네요!\n"); count++;
4	4	count <= 3 4 <= 3	while (0) while 종료	실행되지 못함



## 7. 반복

## 7.2 - do while 문과 for 문

◎ do while 문

- do while문은 반복몸체 수행 후에 반복조건을 검사하므로, 반복조건을 나중에 검사해야 하는 반복에 적합하다. 반복 횟수가 정해지지 않고 입력받은 자료값에 따라 반복 수행의 여부를 결정하는 구문에 유용하며, 반복몸체에 특별히 분기 구문이 없는 경우, do while의 몸체는 적어도 한 번은 실행되는 특징이 있다.

또한, 반복의 종료를 알리는 특정한 자료값을 센티넬 값(Sentinel Value)라 하며, 센티넬 값 검사에도 유용하게 사용된다.

The screenshot displays the Visual Studio IDE with the file 'dowhile.c' open. The code is as follows:

```

1  #define _CRT_SECURE_NO_WARNINGS
2  #include <stdio.h>
3
4  int main(void)
5  {
6      int input;
7
8      do
9      {
10         printf("정수 또는 0(종료)을 입력: ");
11         scanf("%d", &input);
12     } while (input != 0);
13
14     puts("종료합니다.");
15
16     return 0;
17 }

```

The output window on the right shows the program's execution:

```

C:\Users\USER\source\repos\half project\half project> .\dowhile.exe
정수 또는 0(종료)을 입력: 7
정수 또는 0(종료)을 입력: -3
정수 또는 0(종료)을 입력: 5
정수 또는 0(종료)을 입력: 0
종료합니다.

```

## © for 문

- for 문은 주로 반복횟수를 제어하는 제어변수를 사용하며 초기화와 증감부분이 있는 반복문에 적합하다. 반복문 `for(init; cond; inc) stmt;`에서 `init`에서는 주로 초기화가 이루어지며, `cond`에서는 반복조건을 검사하고, `inc`에서는 주로 반복을 결정하는 제어변수의 증감을 수행한다.

또한, `for( ; )`의 괄호 내부에서 세미콜론으로 구분되는 항목은 모두 생략될 수 있지만 2개의 세미콜론은 반드시 필요하며, 반복조건 `cond`를 아예 제거하면 반복은 무한히 계속된다.

※ for문의 실행 순서 : 초기화를 위한 init를 실행하며, 이 init는 단 한번만 수행된다.

반복조건 검사 cond를 평가해 0이 아닌 결과값(참)이면 몸체에 해당하는 stmt를 실행하며, cond가 0(거짓)이면 for 문을 종료하고 다음 문장 next를 실행한다.

반복문체 stmt를 실행한 후 증감연산 inc를 실행한다.

다시 반복조건인 cond를 검사하여 반복한다.

The screenshot shows the Microsoft Visual Studio IDE with two windows. The left window, titled 'forbasic.c', displays the source code of a C program. The code includes `<stdio.h>`, defines `MAX` as 5, and contains a `main` function with a loop that prints 'C 언어 재미있네요!' 5 times. The right window, titled 'Microsoft Visual Studio 디버거 콘솔', shows the program's execution output. It displays the message 'C 언어 재미있네요!' 5 times, followed by '제어변수 i => 6', indicating the loop has completed.

```

1  #include <stdio.h>
2  #define MAX 5
3
4  int main(void)
5  {
6      int i;
7
8      for (i = 1; i <= MAX; i++)
9      {
10         printf("C 언어 재미있네요! %d\n", i);
11     }
12
13     printf("\n제어변수 i => %d\n", i);
14
15     return 0;
16 }

```

Microsoft Visual Studio 디버거 콘솔

```

C 언어 재미있네요! 1
C 언어 재미있네요! 2
C 언어 재미있네요! 3
C 언어 재미있네요! 4
C 언어 재미있네요! 5

제어변수 i => 6

```

## © for 문 활용

- for문을 이용하여 1에서 10까지 합을 구하는 모듈을 작성할 경우, 순환하는 제어변수 I의 값을 계속 합하여 변수 sum에 누적시키며, for(i=1, sum=0; i<10; I++)와 같이 초기화부분에 콤마연산자를 이용하여 나열할 수 있다.

	sum = 0 + 1	+ 2	+ 3 + 4 + 5 + 6 + 7 + 8 + 9 + 0		
	sum = sum + 1				
	sum = sum + 2				



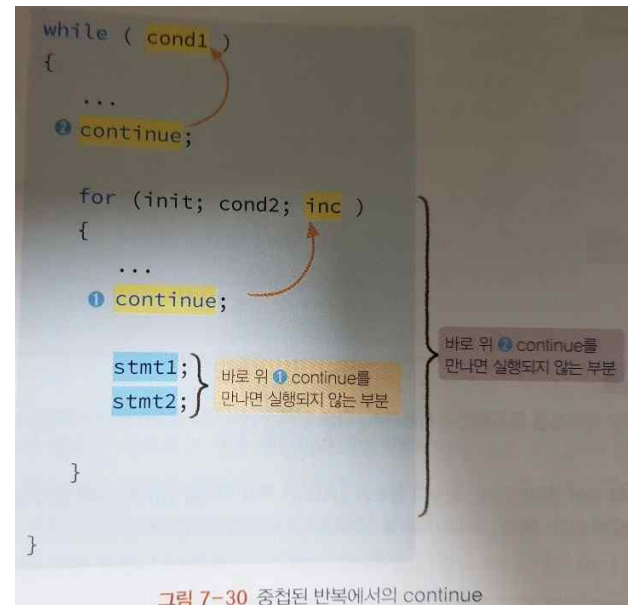
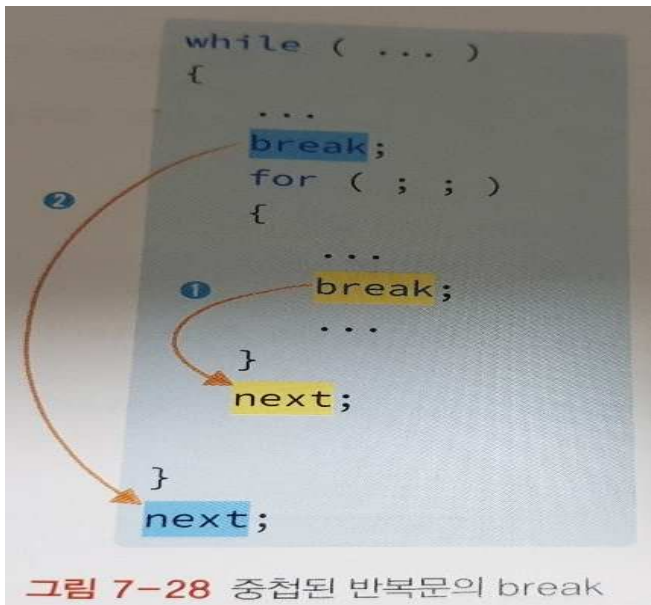
## 7.3 - 분기문

## ◎ break와 continue 문

- 분기문이란 정해진 부분으로 바로 실행을 이동하는 기능을 수행하는 문장으로, C 언어에서 지원하는 분기문으로는 break, continue, goto, return 문이 있다.

반복 내부에서 반복을 종료하려면 **break** 문장을 사용하며, 중첩된 반복에서의 break는 자신이 속한 가장 근접한 반복에서 반복을 종료한다.

**continue** 문은 반복의 시작으로 이동하여 다음 반복을 실행하는 문장으로, continue 문이 위치한 이후의 반복 몸체의 나머지 부분을 실행하지 않고 다음 반복을 계속 유지하는 문장이며, 중첩된 반복에서의 continue는 자신이 속한 가장 근접한 반복에서 다음 반복을 실행한다.



## ◎ goto 문과 무한반복

- goto 문은 레이블(label)이 위치한 다음 문장으로 실행순서를 이동하는 문장으로, 레이블은 식별자와 콜론(:)을 이용하여 지정하지만, goto 문은 프로그램의 흐름을 어렵고 복잡하게 만들 수 있기 때문에 잘 사용하지 않는다.

goto.c
half project 13

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int count = 1;
6
7      loop :
8          printf("%3d", count);
9          if (++count <= 10)
10             goto loop;
11
12     printf("\n프로그램을 종료합니다.\n");
13
14     return 0;
15 }

```

Microsoft Visual Studio 디버깅 콘솔

```

1 2 3 4 5 6 7 8 9 10
프로그램을 종료합니다.
C:\Users\USER\source\repos\half project
디버깅이 중지될 때 콘솔을 자동으로 닫으
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...

```

- 반복문에서 무한히 반복이 계속되는 것을 무한반복이라 하며 개발자가 의도하지 않은 무한반복은 프로그램이 종료되지 않는 결과가 발생하기도 한다.

while 문과 do while 문은 반복조건이 아예 없으면 오류가 발생하지만, for 문에서 (init; inc;)와 같이 반복조건에 아무것도 없을 경우 오류가 발생하지 않고 무한반복이 실행된다.

## 7. 반복

### 7.4 - 중첩된 반복문

#### ◎ 중첩된 for 문

- 반복문 내부에 반복문이 또 있는 구문을 중첩된 반복문(nested loop)이라 하며, 외부반복과 내부반복으로 구성되어 있다.

```
nestedloop.c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int m, n;
6
7      for (m = 1; m <= 5; m++)
8      {
9          printf("m = %-2d\n", m);
10         for (n = 1; n <= 7; n++)
11             printf("n = %-3d", n);
12         puts("");
13     }
14
15     return 0;
16 }
```

```
Microsoft Visual Studio 디버그 콘솔
m = 1
n = 1 n = 2 n = 3 n = 4 n = 5 n = 6 n = 7
m = 2
n = 1 n = 2 n = 3 n = 4 n = 5 n = 6 n = 7
m = 3
n = 1 n = 2 n = 3 n = 4 n = 5 n = 6 n = 7
m = 4
n = 1 n = 2 n = 3 n = 4 n = 5 n = 6 n = 7
m = 5
n = 1 n = 2 n = 3 n = 4 n = 5 n = 6 n = 7

C:\Users\USER\source\repos\half project8\Debug\half project 14.exe(프로세스 61184개)이(가) 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅] > [디버깅이 중지] 하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

## 8. 포인터 기초

### 8.1 - 포인터 변수와 선언

#### ◎ 메모리 주소와 주소연산자 &

- 메모리 공간은 8비트인 1 바이트마다 고유한 주소(address)가 있으며, 메모리 주소는 0부터 바이트마다 1씩 증가한다. 또한, 메모리 주소는 저장 장소인 변수 이름과 함께 기억장소를 참조하는 또 다른 방법이다.
- 주소는 변수 이름과 같이 저장장소를 참조하는 하나의 방법이며, 함수 scanf()를 사용할 때 입력 자료의 값을 저장하기 위해 인자를 &'변수이름'으로 사용하였는데 여기서 **&(ampersand)**가 피연산자인 변수의 메모리 주소를 반환하는 주소연산자이다.
- ※ 함수 scanf()에서 입력값을 저장하는 변수의 주소값이 인자의 자료형이다.  
함수 scanf()에서 일반 변수 앞에는 주소연산자 &를 사용해야 한다.
- 변수의 주소값은 형식제어문자 %u 또는 %d로 직접 출력할 수 있으며, 만일 16진수로 출력하려면 형식제어문자 %p를 사용한다. 다만, 주소연산자 &는 다음 사용에 주의가 필요하다.
- ※ & 연산자는 '&변수'와 같이 피연산자 앞에 위치하는 전위연산자로 변수에만 사용할 수 있다.  
'&32'와 '&(3+4)'와 같이 상수나 표현식에는 사용할 수 없다.

```
address.c
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int input;
7
8     printf("정수 입력: ");
9     scanf("%d", &input);
10    printf("입력값: %d\n", input);
11    printf("주소값: %u(10진수), %p(16진수)\n", (int) &input, &input);
12    printf("주소값: %d(10진수), %#X(16진수)\n", (unsigned) &input, (int)&input);
13    printf("주소값 크기: %d\n", sizeof(&input));
14
15    return 0;
16 }
```

Microsoft Visual Studio 디버그 콘솔

```
정수 입력: 100
입력값: 100
주소값: 11795260(10진수), 00B3FB3C(16진수)
주소값: 11795260(10진수), 0XB3FB3C(16진수)
주소값 크기: 4

C:\Users\USER\source\repos\half project8\Debug\half project 15.exe(프로세스 5815)
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅] > [디버깅
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

#### ◎ 포인터 변수 개념과 선언

- 포인터 변수란 주소값을 저장하는 변수로 일반 변수와 구별되며 선언방법이 다르며, 포인터 변수 선언은 자료형과 포인터 변수 이름 사이에 연산자 \*(asterisk)를 삽입한다. ※ ex) `int *pint = 'int 포인터 pint'`
- 변수의 자료형이 다르면 변수의 주소를 저장하는 포인터의 자료형도 달라야 하며, 어느 변수의 주소값을 저장하려면 반드시 그 변수의 자료유형과 동일한 포인터 변수에 저장해야 한다.

자료형	*변수이름
int	*pint;
short	*ptrshort;
char	*ptrchar;
double	*ptrdouble;

- 포인터 변수도 선언된 후 초기값이 없으면 쓰레기 값이 저장되기 때문에, 예를 들어 포인터 변수에 data 주소를 대입하려면 주소연산자 &를 사용한 &data를 이용하면 된다. 이러한 관계를 '참조'한다고도 표현하며, 포인터 변수는 가리키는 변수의 종류에 관계없이 크기가 모두 4바이트이다.
- ※ `int data = 100; -> int *pint; -> pint = &data;`

## 8. 포인터 기초

### 8.2 - 간접 연산자 \*와 포인터 연산

#### ◎ 다양한 포인터 변수 선언과 간접 연산자 \*

- 여러 개의 포인터 변수를 한 번에 선언하기 위해서는 다음과 같이 콤마 이후에 변수마다 \*를 앞에 기술해야 한다.

※ `int *ptr1, *ptr2, *ptr3;` // 모두 int형 포인터 , `int *ptr1, ptr2, ptr3` // \*ptr1은 int형 포인터지만 ptr2, ptr3은 int형 변수

- 포인터 변수를 다른 일반변수와 같이 지역변수로 선언하는 경우, 초기값을 대입하지 않으면 쓰레기값이 들어가므로 포인터 변수에 지정할 특별한 초기값이 없는 경우에 0번 주소값인 NULL로 초기값을 저장한다.

※ `int *ptr = NULL;`

- 자료유형 (void \*)는 아직 결정되지 않은 자료형의 주소이자, 유보된 포인터이므로 모든 유형의 포인터 값을 저장할 수 있는 포인터 형이다.

※ `#define NULL ((void *)0)`

- 포인터 변수가 가리키고 있는 변수를 참조하려면 **간접연산자 \***를 사용하며, 다음의 예에서 보듯이 변수 data 자체를 사용해 자신을 참조하는 방식을 **직접참조**라 한다면 \*pprint를 사용해서 변수 data를 참조하는 방식을 **간접참조**라 한다.

```
int data = 100;
char ch = 'A';
int *pprint = &data;
char *ptrchar = &ch;
printf("간접참조 출력: %d %c\n", *pprint, *ptrchar);

*pprint = 200;
printf("직접참조 출력: %d %c\n", data, ch);
```

#### ◎ 포인터 변수와 연산

- 포인터 변수는 간단한 더하기와 뺄셈 연산으로 이웃한 변수의 주소 연산을 수행할 수 있으며, 포인터에 저장된 주소값의 연산으로 이웃한 이전 또는 이후의 다른 변수를 참조할 수 있다.  
또한, 더하기와 빼기 연산에는 포인터 변수가 피연산자로 참여할 수 있다.

```
calcptr.c  X
half project 16 (전역 범위)
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char* pc = (char *)100;
6      int* pi = (int *)100;
7      double* pd = (double *)100;
8      pd = 100;
9
10     printf("%u %u %u\n", (int)(pc - 1), (int)pc, (int)(pc + 1));
11     printf("%u %u %u\n", (int)(pi - 1), (int)pi, (int)(pi + 1));
12     printf("%u %u %u\n", (int)(pd - 1), (int)pd, (int)(pd + 1));
13
14     return 0;
15 }
```

Microsoft Visual Studio 디버그 콘솔

```
99 100 101
96 100 104
92 100 108

C:\Users\USER\source\repos\half project 16\Debug\half project 16.exe(프
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```



## 8. 포인터 기초

### 8.3 - 포인터 형변환과 다중 포인터

#### ◎ 내부 저장 표현과 포인터 변수의 형변환

- 예를 들어, 변수 value에 16진수 0x61626364를 저장한다면 변수의 주소에 따라 값도 다르게 표현된다.

포인터: pi	주소	값	int value = 0x61626364; / int *pi = &value;	
&value	0012FF56	0110 0100	1633837924	4바이트
	0012FF57	0110 0011		
	0012FF58	0110 0010		

- 포인터 변수는 동일한 자료형끼리만 대입이 가능하며, 만약 대입문에서 포인터의 자료형이 다르면 경고가 발생한다. 포인터 변수는 자동으로 형변환이 불가능하며, 필요하다면 명시적으로 형변환을 수행할 수 있다.

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int value = 0x61626364;
6     int* pi = &value;
7     char* pc = (char*)&value;
8
9     printf("변수명      저장값      주소값\n");
10    printf("-----\n");
11    printf("value      %0#x      %u\n", value, pi);
12
13    for(int i = 0; i <= 3; i++)
14    {
15        char ch = *(pc + i);
16        printf("(pc+%d)  %0#x      %2c  %u\n", i, ch, ch, pc + i);
17    }
18
19    return 0;
20 }

```

변수명 저장값 주소값

value 0x61626364 10747280

\*(pc+0) 0x0064 d 10747280

\*(pc+1) 0x0063 c 10747281

\*(pc+2) 0x0062 b 10747282

\*(pc+3) 0x0061 a 10747283

C:\Users\USER\source\repos\half project 18\Debug 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [5] 하도록 설정합니다. 이 창을 닫으려면 아무 키나 누르세요...

#### ◎ 다중 포인터와 중감 연산자의 활용

- 포인터 변수의 주소값을 갖는 변수를 이중 포인터라 하며, 이러한 포인터의 포인터를 모두 다중 포인터라고 한다. 다음 소스에서 pi는 포인터이며, 포인터 변수 pi의 주소값을 저장하는 변수 dpi는 이중 포인터이다.

※ int i = 20; / int \*pi = &i; - 포인터 / int \*\*dpi = &pi; - 이중 포인터

- 간접 연산자 \*는 중감 연산자 ++, --와 함께 사용하는 경우가 많으며 연산자마다 우선순위가 정해져있는데, 포인터 변수 p를 예로 들면 다음과 같다.

※ \*p++는 \*(p++)로 (\*p)++와 다르다. / ++\*p와 ++(\*p)는 같다. / ++\*p와 \*(++p)는 같다.

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int i;
6     int* pi = &i;
7     int** dpi = &pi;
8
9     *pi = 5;
10    *pi += 1;
11    printf("%d\n", i);
12    printf("%d\n", (*pi)++);
13    printf("%d\n", *pi);
14
15    *pi = 10;
16    printf("%d\n", ++*pi);
17    printf("%d\n", ++**dpi);
18    printf("%d\n", i);
19
20    return 0;
21 }

```

5

6

7

11

12

12

C:\Users\USER\source\repos\half project 18\Debug 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [5] 하도록 설정합니다. 이 창을 닫으려면 아무 키나 누르세요...

#### ◎ 포인터 상수

- 키워드 const를 이용하는 변수 선언은 변수를 상수로 만들 듯이, 포인터 변수도 포인터 상수로 만들 수 있다.

※ 1. int <-> const \*pi = &i; / 2. int\* const pi = &i;

① 키워드 const가 먼저 나오거나 중간에 나오는 선언은, 간접 연산식 \*pi를 상수로 만드는 방법으로 변수인 i를 수정할 수 없게끔 한다.

② 키워드 const가 int\*와 변수 pi 사이에 나오는 선언은, 포인터 변수 pi 자체를 상수로 만드는 방법으로, pi 자체를 수정할 수 없다.

## 9.1 - 배열 선언과 초기화

## ◎ 배열의 필요성과 정의

- **배열(Array)**은 여러 변수들이 같은 배열 이름으로 일정한 크기의 연속된 메모리에 저장되는 구조를 말하며, 이를 이용하면 변수를 일일이 선언하는 번거로움을 해소할 수 있고, 배열을 구성하는 각각의 변수를 참조하는 방법도 간편하며 반복 구문으로도 쉽게 참조할 수 있다.
- 배열은 동일한 자료 유형이 여러 개 필요한 경우에 유용한 자료 구조로서, 한 자료유형의 저장공간인 원소를 동일한 크기로 지정된 배열크기만큼 확보한 연속된 저장공간이다.  
여기서 배열을 구성하는 각각의 항목을 **배열의 원소(elements)**라 하며, 배열에서 중요한 요소로는 **배열이름**, **원소 자료유형**, **배열크기**이다.

## ◎ 배열 선언과 원소참조

- 배열선언은 `int data[10];`과 같이 원소자료유형 배열이름[배열크기]로 명시되어야 하며, 배열선언 시 초기값 지정이 없다면 반드시 배열크기는 양의 정수로 명시되어야 한다.
- \* 배열크기는 대괄호 사이에 [배열크기]와 같이 기술한다.  
배열의 크기를 지정하는 부분에는 양의 정수로 리터럴 상수와 매크로 상수 또는 이들의 연산식이 올 수 있으며, 변수와 `const` 상수로는 배열의 크기를 지정할 수 없다.  
원소 자료형으로는 모든 자료형이 올 수 있으며, 배열 이름은 식별자 생성 규칙에 따른다.
- 배열 선언 후 배열원소에 접근하려면 배열이름 뒤에 대괄호 사이 첨자(index)를 이용한다.
- \* 배열에서 유효한 첨자의 범위는 0부터 (배열크기-1)까지이며, 첨자의 유효 범위를 벗어난 원소를 참조할 경우 문법오류 없이 실행오류가 발생한다.  
배열선언 시 대괄호 안의 수는 배열 크기이다. 그러나 선언 이후 대괄호 안의 수는 원소를 참조하는 번호인 첨자라는 것을 명심해야 한다.

The screenshot shows a C program in a file named `declarearray.c`. The code defines a constant `SIZE` as 5 and declares an array `score` of type `int` with size `SIZE`. It initializes the first three elements of the array with values 78, 97, and 85. A comment indicates that the fourth element is not stored because it's out of bounds, and the fifth element is set to 91. A loop prints each element of the array. The output in the console window shows the values: 78 97 85 -858993460 91. The error message in the console indicates that the debugger will stop when the console is closed.

```

1 #include <stdio.h>
2 #define SIZE 5
3
4 int main(void)
5 {
6     int score[SIZE];
7
8     score[0] = 78;
9     score[1] = 97;
10    score[2] = 85;
11    // 배열 4번째 원소에 값을 저장하지 않아서 쓰레기값 저장
12    score[4] = 91;
13
14    for (int i = 0; i < SIZE; i++)
15        printf("%d ", score[i]);
16    printf("\n");
17
18    return 0;
19 }

```

Microsoft Visual Studio 디버그 콘솔

```

78 97 85 -858993460 91
C:\Users\WUSER\source\repos\half project\half project\Debug\half project.exe
디버깅이 중지될 때 콘솔을 자동으로 닫으
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...

```

## ◎ 배열 초기화

- 배열을 선언할 때 크기가 클 경우 원소값을 일일이 저장하기 어려운데, C언어에서는 배열을 선언하면서 동시에 원소값을 손쉽게 저장하는 **배열선언 초기화(initialization)** 방법을 제공한다.  
배열선언 초기화 구문은 배열선언을 하면서 대입연산자를 이용하여 중괄호 사이에 여러 원소값을 쉼표로 구분하여 기술하는 방법이다.  
만약 배열크기가 초기값 원소수보다 클 경우, 저장하지 않은 원소의 초기값은 모두 기본값(0, 0.0, \0)으로 저장된다.
- \* 중괄호 사이에는 명시된 배열크기를 넘지 않게 원소값을 나열할 수 있다.  
일반 배열 선언과 다르게 배열크기는 생략할 수 있으며, 생략할 경우 자동으로 중괄호 사이에 기술된 원소 수가 배열크기가 된다.  
원소값을 나열하기 위해 콤마를 사용하고, 전체를 중괄호로 묶는다.
- \* `int grade[4] = {98, 88, 92, 95};`  
`int cpoint[] = {99, 76, 84, 76, 68}; -> int cpoint[5] = {99, 76, 84, 76, 68};`

## 9.2 - 이차원과 삼차원 배열

## ◎ 이차원 배열 선언과 사용

- 이차원 배열은 테이블 형태의 구조를 나타낼 수 있어서, **행(row)과 열(column)**의 구조로 표현할 수 있다. 이차원 배열선언은 2개의 대괄호가 필요하며, 첫 번째 대괄호에는 배열의 행 크기, 두 번째는 배열의 열 크기를 지정한다. 다만, 배열 선언 시 초기값을 저장하지 않을 경우 반드시 행과 열의 크기는 명시되어야 한다.

※ 원소자료형 배열이름[배열행크기][배열열크기]; -> int score [RSIZE][CSIZE]; / double point[2][3]; / char ch[5][80]; 등

- 이차원 배열에서 각 원소를 참조하기 위해서는 2개의 첨자가 필요하며, 예를 들어 배열 선언 int td[2][3];으로 선언된 배열 td에서 첫 번째 원소는 td[0][0]로 참조한다. 일차원 배열과 마찬가지로 이차원 배열원소를 참조하기 위한 행 첨자는 0에서 (행크기 - 1), 열 첨자는 0에서 (열크기 - 1)까지 유효하다. 또한, 이차원 배열은 첫 번째 행 모든 원소가 메모리에 할당된 이후에 두 번째 행의 원소가 순차적으로 할당되는데, 이러한 특징을 행 우선(row major)배열이라 한다.

The screenshot shows a C program in Visual Studio. The code defines a 2D array 'td' with 2 rows and 3 columns, initializes it with values 1 through 6, and prints each element. The debug console shows the output of the program, confirming the values stored in the array.

```

1 #include <stdio.h>
2 #define ROWSIZE 2
3 #define COLSIZE 3
4
5 int main(void)
6 {
7     int td[ROWSIZE][COLSIZE];
8
9     td[0][0] = 1; td[0][1] = 2; td[0][2] = 3;
10    td[1][0] = 4; td[1][1] = 5; td[1][2] = 6;
11
12    for (int i = 0; i < ROWSIZE; i++)
13    {
14        for (int j = 0; j < COLSIZE; j++)
15            printf("td[%d][%d] == %d ", i, j, td[i][j]);
16        printf("\n");
17    }
18
19    return 0;
20 }

```

Microsoft Visual Studio 디버그 콘솔

```

td[0][0] == 1 td[0][1] == 2 td[0][2] == 3
td[1][0] == 4 td[1][1] == 5 td[1][2] == 6

```

C:\Users\USER\source\repos\half project 16\Debug\half project 16.exe: 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] 하도록 설정합니다. 이 창을 닫으려면 아무 키나 누르세요...

## ◎ 이차원 배열 초기화

- 이차원 배열을 선언하면서 초기값을 지정하는 방법은 중괄호를 중첩되게 이용하는 방법과 일차원 배열같이 하나의 중괄호를 사용하는 방법이 있으며, 다른 방법으로는 하나의 중괄호로 모든 초기값을 쉼표로 분리하는 방법이 있다.

※ int score[2][3] = {{30, 44, 67}, {87, 43, 56}}; / int score[][3] = {30, 44, 67, 87, 43, 56};

- 이차원 배열선언 초기값 지정에서도 첫 번째 대괄호 내부의 행의 크기는 명시하지 않을 수 있으나, 두 번째 대괄호 내부의 열의 크기는 반드시 명시해야 한다.

또한, 이차원 배열의 총 배열원소 수보다 적게 초기값이 주어진다면 나머지는 모두 기본값인 0, 0.0 '\0'이 저장된다.

※ int a[2][4] = {10, 30, 40, 50, 1, 3, 0, 0}; / int a[2][4] = {10, 30, 40, 50, 1, 3}; / int a[][4] = {10, 30, 40, 50, 1, 3};  
 int a[2][4] = { {10, 30, 40, 50}, {1, 3} }; / int a[][4] = { {10, 30, 40, 50}, {1, 3} };

## ◎ 삼차원 배열

- C언어는 다차원 배열을 지원하지 않지만, 실제로 삼차원 이상의 배열을 사용하는 경우는 드물기 때문에 여기서는 삼차원 배열을 위한 선언문과 그 구조를 나타낸다.

※ int a[3] = {88, 78, 78}; - 일차원 배열 / int b[2][3] = { {95, 85, 86}, {90, 88, 90} };  
 int c[2][2][3] = { { {95, 85, 86}, {90, 88, 90} },  
 { {95, 85, 86}, {90, 88, 90} } }



## 9.3 - 배열과 포인터 관계

## ◎ 일차원 배열과 포인터

- 배열은 실제 포인터와 연관성이 매우 많으며, 다음 배열 score에서 배열이름 score 자체가 배열 첫 원소의 주소값인 상수인데, 배열 score의 특징은 다음과 같다. (int score[] = {89, 98, 76};)

\* 배열이름 score는 배열 첫 번째 원소의 주소를 나타내는 상수로 &score[0]과 같으며 배열을 대표하기 때문에 간접연산자를 이용한 \*score는 변수 score[0]와 같다.

배열이름 score가 포인터 상수로 연산식 (score + 1)이 가능하고, 이것은 배열 score의 다음 배열 원소의 주소값을 의미하기 때문에 (score + 1)은 &score[1]이다. 이것을 확장하면 (score + 1)은 &score[i]이다.

이와 마찬가지로 간접연산자를 이용한 \*score는 변수 score[0]인 것을 확장하면 \*(score + 1)은 score[i]와 같다.

배열 초기화 문장		int score[] = {89, 98, 76};		
배열 원소값		89	98	76
배열원소 접근방법	score[i]	score[0]	score[1]	score[2]
	*(score+i)	*score	*(score+1)	*(score+2)
주소값 접근방법	&score[i]	&score[0]	&score[1]	&score[2]
	score+i	score	score+1	score+2
실제 주소값	base + 원소크기*i	만일 4라면	8 = 4+1*4	12 = 4+2*4

## ◎ char 배열을 int 자료형으로 인식

- 포인터 변수는 동일한 자료형끼리만 대입이 가능하며, 만약 대입문에서 포인터의 자료형이 다르면 경고가 발생한다.
- \* char c[4] = {'A', '\0', '\0', '\0'}; / int \*pi = &c[0]; -> warning C4133 : '초기화중' : 'char'와 'int'사이의 형식이 호환되지 않습니다.
- 동일한 메모리의 내용과 주소로부터 참조하는 값이 포인터의 자료형에 따라 달라진다는 것을 알 수 있다.

```

ptypecast.c → ×
half project 21
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char c[4] = { 'A', '\0', '\0', '\0' };
6     int* pi = (int*)&c[0];
7
8     printf("%d %c\n", (int)c[0], c[0]);
9     printf("%d %c\n", *pi, (char)*pi);
10
11     return 0;
12 }

```

Microsoft Visual Studio 디버그 콘솔

```

65 A
65 A
C:\Users\USER\source\repos\half project 1\half project 1\Debug>
디버깅이 중지될 때 콘솔을 자동으로 닫으려
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...

```

## ◎ 이차원 배열과 포인터

- 다음과 같은 이차원 배열에서 배열이름인 td는 포인터 상수 td[0]을 가리키는 포인터 상수이며, 포인터 상수 td[0]은 배열의 첫 번째 원소 td[0][0]의 주소값 &td[0][0]을 갖는 포인터 상수이다.
- 그러므로 배열이름인 td는 포인터의 포인터인 이중포인터이다.

\* int td[][3] = { {8, 5, 4}, {2, 7, 6} };

배열이름 td는 이차원 배열을 대표하는 이중 포인터이며, sizeof(td)는 배열 전체의 바이트 크기를 반환한다.

배열이름 td를 이용하여 변수 td[0][0]의 값을 20으로 수정하려면 \*\*td = 20; 문장을 이용할 수 있으며, td가 이중 포인터이므로 간접연산자 \*이 2개 필요하다.

td[i]는 (i+1) 번째 행을 대표하며, (i+1) 번째 행의 처음을 가리키는 포인터 상수이다.

td[i]은 두 번째 행의 첫 원소의 주소이므로 \*td[i]로 td[i][0]을 참조할 수 있다.

## ◎ 포인터 배열

- **포인터 배열(Array of pointer)**이란 주소값을 저장하는 포인터를 배열 원소로 하는 배열이다.

```

* int a = 5, b = 7, c = 9; / int *pa[3]; / pa[0] = &a; -> 5 / pa[1] = &b; -> 7 / pa[2] = &c; -> 9

```

일차원 배열 포인터	이차원 배열 포인터
<p>원소자료형 *변수이름;  변수이름 = 배열이름;  원소자료형 *변수이름 = 배열이름;</p>	<p>원소자료형 (*변수이름)[ 배열열크기];  변수이름 = 배열이름;  원소자료형 (*변수이름)[ 배열열크기] = 배열이름;</p>
<pre>int a[] = {8, 2, 8, 1, 3}; int *p = a;</pre>	<pre>int ary[][4] = {5, 7, 6, 2, 7, 8, 1, 3}; int (*ptr)[4] = ary;</pre>

## ◎ 배열 포인터

- 자료형이 int인 일차원 배열 int a[]의 주소는 (int \*)인 포인터 변수에 저장할 수 있으며, 열이 4인 이차원 배열 ary[][4]의 주소를 저장하려면 **배열 포인터(pointer to array)** 변수 ptr을 문장 int (\*ptr)[4];로 선언해야 한다.

즉, 이차원 배열의 주소를 저장하는 포인터 변수는 열 크기에 따라 변수 선언이 달라지며, 주의할 점은 다음과 같다.

※ 괄호가 없는 `int *ptr[4];`는 `int`형 포인터 변수 4개를 선언하는 포인터 배열 선언문장이다.

**int (\*ptr)[4];**는 열이 4인 이차원 배열 포인터 선언문장이다.

### ◎ 배열 크기 연산

- 배열크기의 계산방법은 연산자 sizeof를 이용한 식 ( sizeof(배열이름) / sizeof(배열원소) )의 결과이다.

※ sizeof(배열이름)은 배열의 전체 공간의 바이트 수이다. / sizeof(배열원소)는 배열원소 하나의 바이트 수이다.

```
arraysize.c → x
half project 22 (전역 범위) Microsoft Visual Studio 디버깅 콘솔

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int data[] = { 3, 4, 5, 7, 9 };
6
7     printf("%d %d\n", sizeof(data), sizeof(data[0]));
8     printf("일차원 배열: 배열 크기 == %d\n", sizeof(data) / sizeof(data[0]));
9
10    double x[3][3] = { { 1, 2, 3 }, { 7, 8, 9 }, { 4, 5, 6, }, { 10, 11, 12 } };
11
12    printf("%d %d %d\n", sizeof(x), sizeof(x[0]), sizeof(x[1]), sizeof(x[0][0]));
13    int rowsize = sizeof(x) / sizeof(x[0]);
14    int colsize = sizeof(x[0]) / sizeof(x[0][0]);
15    printf("이차원 배열: 행수 == %d 열수 == %d\n", rowsize, colsize);
16    printf("이차원 배열: 전체 원소 수 == %d\n", sizeof(x) / sizeof(x[0][0]));
17
18    return 0;
19 }
```

20 4  
일차원 배열: 배열 크기 == 5  
96 24 24  
이차원 배열: 행수 == 4 열수 == 3  
이차원 배열: 전체 원소 수 == 12

C:\Users\USER\source\repos\half project 16\half project 22\half project 22.c: 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 하도록 설정합니다.  
이 창을 닫으려면 아무 키나 누르세요...

## 10. 함수 기초

### 10.1 - 함수정의와 호출

#### ◎ 함수의 이해

- 프로그램에서 특정한 작업을 처리하도록 작성한 프로그램 단위를 **함수(function)**라고 하며, 함수는 필요한 입력을 받아 원하는 어떤 기능을 수행한 후 결과를 반환(return)하는 프로그램 단위이다.  
그러므로 C 프로그램은 최소한 main() 함수와 다른 함수로 구성되는 프로그램이다.
- 함수는 **라이브러리 함수(Library function)**와 **사용자 정의 함수(User defined function)**으로 구분할 수 있으며, 라이브러리 함수란 이미 개발환경에 포함되어 있는 printf(), scanf()같은 함수를 말하고 사용자 정의 함수란 필요에 의해서 개발자가 직접 개발하는 함수를 일컫는다.
- 사용자가 직접 개발한 함수를 사용하기 위해서는 **함수선언(function declaration)**, **함수호출(function call)**, **함수정의(function definition)**가 필요하다.
- 적절한 함수로 잘 구성된 프로그램을 모듈화 프로그램(modular program) 또는 구조화된 프로그램이라 하며, 한번 정의된 함수는 여러 번 호출이 가능하므로 소스의 중복을 최소화하고 프로그램의 양을 줄이는 효과를 가져온다.  
이러한 함수 중심의 프로그래밍 방식을 **절차적 프로그래밍(procedural programming)**방식이라고 한다.

#### ◎ 함수정의

- 함수정의는 함수머리와 함수몸체로 구성된다.
- \* 함수머리는 반환형과 함수이름, 매개변수 목록으로 구성된다.  
함수머리에서 반환형은 함수 결과값의 자료형이며, 이 반환형은 int, float, double과 같은 다양한 자료형이 올 수 있다.  
함수이름은 식별자의 생성규칙을 따르며, 괄호 안에 기술되는 매개변수 목록은 자료형 변수이름의 쌍으로 필요한 수만큼 콤마로 구분하여 기술한다.  
함수몸체는 {...}와 같이 중괄호로 시작하여 중괄호로 종료된다.  
함수몸체에서는 함수가 수행해야 할 문장들로 구성되며, 몸체의 마지막은 대부분 결과값을 반환하는 return 문장으로 종료된다.

```
반환형 함수이름 (매개변수 목록)
{
    ...
    여러 문장들;
    return (반환연산식);
}
```

```
int add2(int a, int b)
{
    int sum = a + b;

    return (sum);
}
```

#### ◎ 함수선언과 함수호출

- 정의된 함수를 실행하려면 프로그램 실행중에 함수호출이 필요하다.
- \* 함수를 호출하려면 함수이름과 함께 괄호 안에 적절한 매개변수가 필요하다. (ex - findMax2(a, b), add2(a,b) 등)  
반환값이 있는 함수는 반환된 값을 저장하기 위해 왼쪽에 변수(l-value)가 위치하는 대입연산자가 필요하다.
- 함수원형이란 함수를 선언하는 문장이며, 함수를 호출하기 이전에 반드시 선언되어야 한다.
- \* 함수원형 구문에서 매개변수의 변수이름은 생략할 수 있다. 즉, int add2(int, int)도 가능하다.  
함수원형의 매개변수 정보는 자료형과 개수 그리고 그 순서가 중요하므로 변수이름은 생략할 수 있다.

The screenshot shows a C program named 'functionadd2.c' in a code editor. The program includes <stdio.h> and defines a main function and an add2 function. The main function calls add2 with arguments 3 and 5, and prints the result. The add2 function returns the sum of its arguments. The output window shows the result '합: 8'.

```
functionadd2.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 3, b = 5;
6     int add2(int a, int b);
7
8     int sum = add2(a, b);
9     printf("합: %d\n", sum);
10
11     return 0;
12 }
13
14 int add2(int a, int b)
15 {
16     int sum = a + b;
17
18     return (sum);
19 }
20
```

Microsoft Visual Studio 디버그 콘솔

합: 8

C:\Users\USER\source\repos\half project\half project\16\디버깅이 중지될 때 콘솔을 자동으로 닫으려면 하도록 설정합니다.  
이 창을 닫으려면 아무 키나 누르세요...

## 10. 함수 기초

### 10.2 - 함수의 매개변수 활용

#### ◎ 매개변수와 인자

- 함수 매개변수는 함수를 호출하는 부분에서 함수몸체로 값을 전달할 목적으로 이용되며, 함수정의에서 매개변수는 필요한 경우 자료형과 변수명의 목록으로 나타내며 필요없으면 키워드 void를 기술한다.  
다만, 함수 매개변수는 여러 개를 사용할 수 있으나 반환값은 하나만 이용할 수 있는 제약이 있다.

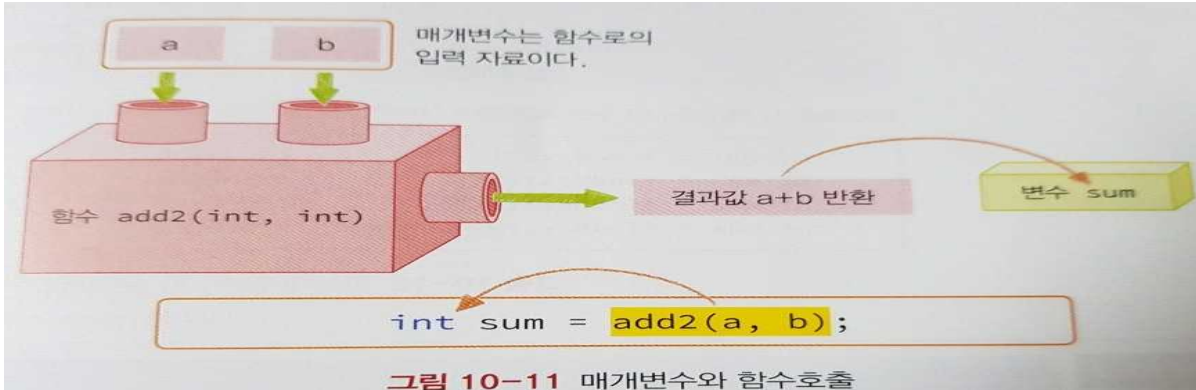


그림 10-11 매개변수와 함수호출

- 함수정의에서 기술되는 매개변수 목록의 변수를 형식매개변수(formal parameters)라 하며, 함수를 호출할 때 기술되는 변수 또는 값을 실 매개변수(real parameters), 실인자(real argument)라고 한다.
- \* 실인자를 기술할 때는 함수예터에 정의된 자료유형과 순서가 일치하도록 해야하며, 실인자의 수와 자료형이 다르면 문법 오류가 발생한다. 형식인자의 변수는 그 함수가 호출되는 경우에 메모리에 할당되며, 함수가 종료되면 메모리에서 자동으로 제거된다. 따라서, 형식매개변수는 함수 내부에서만 사용할 수 있는 변수이다.  
함수가 호출되는 경우 실인자의 값이 형식인자의 변수에 각각 복사된 후 함수가 실행되며, 이를 값에 의한 호출(call by value)이라고 한다.

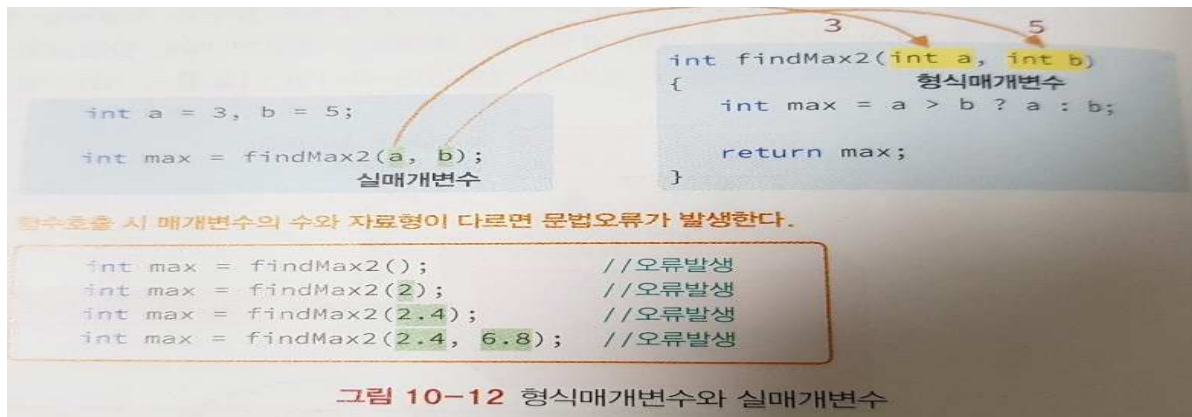


그림 10-12 형식매개변수와 실매개변수

- 함수의 매개변수로 배열을 전달할 경우 한 번에 여러 개의 변수를 전달하는 효과를 가져오며, 다음과 같이 배열을 매개변수로 하는 함수 sum()은 실수형 배열의 모든 원소의 합을 구하여 반환하는 함수이다.
- \* 함수 sum()의 형식매개변수는 실수형 배열 double ary[]와 배열크기 int n으로 하며, 첫 번째 형식매개변수에서 배열자체에 배열크기를 기술하는 것은 아무 의미가 없다.  
실제로 함수내부에서 실인자로 전달되는 배열크기를 알 수 없으므로 배열크기를 두 번째 인자로 사용한다.

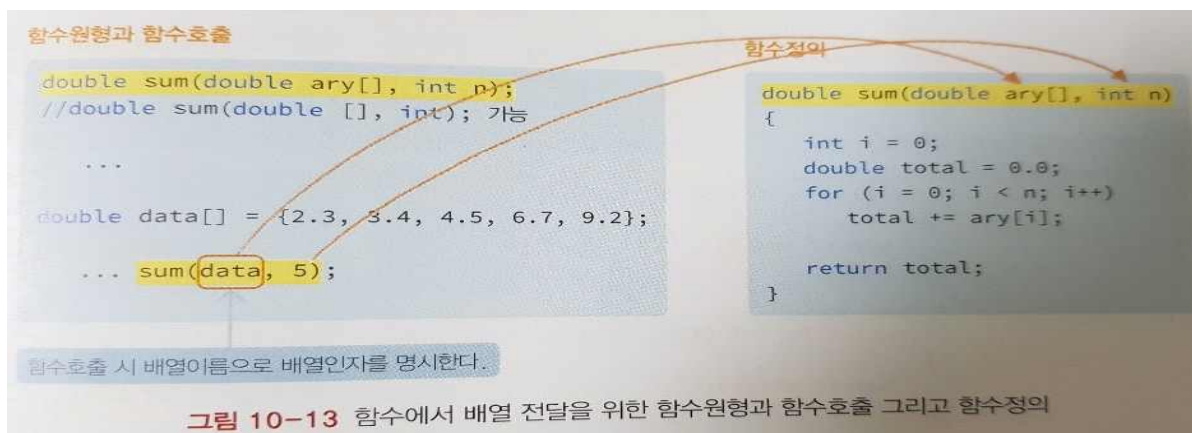


그림 10-13 함수에서 배열 전달을 위한 함수원형과 함수호출 그리고 함수정의



## 10. 함수 기초

### 10.3 - 재귀와 라이브러리 함수

#### ◎ 재귀와 함수 구현

- 함수구현에서 자기자신의 함수를 호출하는 함수를 **재귀 함수(recursive function)**라 한다.

```
factorial.c - half project 24
1 #include <stdio.h>
2
3 int factorial(int);
4
5 int main(void)
6 {
7     for (int i = 1; i <= 10; i++)
8         printf("%2d! = %d\n", i, factorial(i));
9
10    return 0;
11
12
13 int factorial(int number)
14 {
15     if (number <= 1)
16         return 1;
17     else
18         return (number * factorial(number - 1));
19 }
```

```
Microsoft Visual Studio 디버그 콘솔
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
C:\Users\USER\source\repos\half_project_1
디버깅이 중지될 때 콘솔을 자동으로 닫으려
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

- 재귀함수는 함수의 호출이 계속되면 시간이 오래 걸리고 메모리의 사용이 많다는 단점이 있다.

#### ◎ 난수 라이브러리 함수

- 특정한 나열 순서나 규칙을 가지지 않는 연속적인 임의의 수를 **난수(random number)**라 하며, 난수를 생성하는 함수 **rand()**의 함수원형은 헤더파일 **stdlib.h**에 정의되어 있다.

\* 함수 **rand()**를 사용하려면 헤더파일 **stdlib.h**를 추가해야 하며, **VISUAL C++**에서 함수 **rand()**는 0 ~ 32767의 정수 중 임의로 하나의 정수를 반환한다.

```
#include <stdlib.h>
int main(void)
{
    ...
    printf("%5d ", rand());
}
```

- 함수 **srand()**는 **rand()**와 다르게 매번 난수를 다르게 생성하는 시드(seed)값의 **s**를 붙여 **srand()**로 명명되었고, 항상 서로 다른 seed값을 지정하기 위해 함수 **time()**을 이용한다.

\* 함수 **time()**을 이용하기 위해 헤더파일 **time.h**를 삽입한다.  
난수에 시드를 지정하기 위해 함수 **srand((long) time(NULL))**을 호출한다.  
1에서 n까지의 난수를 발생시키려면 함수 **rand()**를 이용하여 수식 **rand() % n + 1**을 이용한다.  
이를 일반화시켜 a에서 b까지의 난수를 발생시키려면 함수 **rand()**를 이용한 수식 **rand() % (b - a + 1) + a**를 이용한다.

```
#include <stdlib.h>
#include <time.h>
#define MAX 100

int main(void)
{
    ...
    srand((long) time(NULL));
    ...
    number = rand() % MAX + 1;
}
```

#### ◎ 수학과 문자 라이브러리 함수

- 수학 관련 함수를 사용하려면 헤더파일 **math.h**를 삽입하고, 문자 관련 함수를 사용하려면 헤더파일 **ctype.h**를 삽입한다.

수학 관련 함수		문자 관련 함수	
double sin(double x)	삼각함수 sin	isalpha(char)	영문자 검사
double cos(double x)	삼각함수 cos	isupper(char)	영문 대문자 검사
double tan(double x)	삼각함수 tan	islower(char)	영문 소문자 검사
double sqrt(double x)	제곱근	isdigit(char)	숫자(0 ~ 9) 검사
int abs(int x)	정수 x의 절대 값	toupper(char)	영문 소문자를 대문자로 변환
double fabs(double x)	실수 x의 절대 값	tolower(char)	영문 대문자를 소문자로 변환



## 11. 문자와 문자열

## 11.1 - 문자와 문자열

## ◎ 문자와 문자열 선언

- **문자**는 영어의 알파벳이나 한글의 한 글자를 작은 따옴표로 둘러싸서 'A'와 같이 표기하며, 작은 따옴표에 의해 표기된 문자를 문자 상수라 한다.
  - 이런 문자의 모임인 일련의 문자를 **문자열(String)**이라 하고, 문자열은 일련의 문자 앞 뒤로 큰 따옴표로 둘러싸서 "Java"로 표기하며, 큰 따옴표에 의해 표기된 문자열을 문자열 상수라 한다.  
그러나 문자의 나열인 문자열은 'ABC'처럼 작은 따옴표로 둘러싸도 문자가 될 수 없으며 오류가 발생한다.
  - C언어에서는 char형 변수에 문자를 저장하며, 문자열을 저장하려면 문자의 모임인 '문자 배열'을 사용한다.
- \* 문자열의 마지막을 의미하는 NULL 문자 '\0'가 마지막에 저장되어야 하며, 문자열이 저장되는 배열 크기는 반드시 저장될 문자 수보다 1이 커야 한다.  
만약 지정한 배열 크기가 (문자수 +1)보다 크면 나머지 부분은 모두 '\0'으로 채워진다.
- ```
char ch = 'A';  
char csharp[3];
```

```
char ch = 'A';
char csharp[3];

csharp[0] = 'C'; csharp[1] = '#';
csharp[2] = '\0';
```

## ◎ 함수 printf()를 사용한 문자와 문자열 출력

- 함수 **printf()**에서 형식제어문자 %c로 문자를 출력하며, %s로 문자열을 출력한다.

```

chararray.c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char ch = 'A';
6     printf("%c %d\n", ch, ch);
7
8     char java[] = { 'J', 'A', 'V', 'A', '\0' };
9     printf("%s\n", java);
10
11     char c[] = "C language";
12     printf("%s\n", c);
13
14     char csharp[5] = "C#";
15     printf("%s\n", csharp);
16
17     printf("%c%c\n", csharp[0], csharp[1]);
18
19     return 0;
20 }

```

Microsoft Visual Studio 디버그 콘솔

```

A 65
JAVA
C language
C#
C#
C:\Users\WUSER\source\repos\half project\half project\Debug\half project.exe
디버깅이 중지될 때 콘솔을 자동으로 닫으
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...

```

## ◎ 다양한 문자 입출력

- 함수 **getchar()**는 문자의 입력에 사용되고, **putchar()**는 문자의 출력에 사용된다.
- \* 문자 입력을 위한 함수 **getchar()**는 라인 버퍼링(line buffering) 방식을 사용하므로 문자를 입력하고 [enter] 키를 눌러야 입력이 실행된다.
- 함수 **getche()**는 버퍼를 사용하지않고 문자를 바로 입력할 수 있는 함수이며, 헤더파일 **conio.h**를 삽입해야 한다.
- 함수 **getch()**는 입력한 문자가 화면에 보이지 않는 특성이 있는 함수이며, 헤더파일 **conio.h**를 삽입해야 한다.

| 함수        | scanf("c", &ch)   | getchar() | getche() / _getche() | getch() / _getch() |
|-----------|-------------------|-----------|----------------------|--------------------|
| 헤더파일      | stdio.h           |           | conio.h              |                    |
| 버퍼 이용     | 버퍼 이용함            |           | 버퍼 이용 안함             |                    |
| 반응        | [enter] 키를 눌러야 작동 |           | 문자 입력마다 반응           |                    |
| 입력 문자의 표시 | 누르면 바로 표시         |           | 누르면 바로 표시            | 표시 안됨              |
| 입력문자 수정   | 가능                |           | 불가능                  |                    |

- 함수 **gets()**는 한 행의 문자열 입력에 유용한 함수이며, **puts()**는 한 행에 문자열을 출력하는 함수이다.

| 문자열 입출력 함수 : 헤더파일 stdio.h 삽입                                  |                                                                                                                            |
|---------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>char * gets(char * buffer);</code>                      | 함수 gets()는 문자열을 입력받아 buffer에 저장하고 입력받은 첫 문자의 주소값을 반환한다.<br>함수 gets()는 표준입력으로 [enter] 키를 누를 때까지 공백을 포함한 한 행의 모든 문자열을 입력받는다. |
| <code>char * gets_s(char * buffer, size_t sizebuffer);</code> | 두 번째 인자인 sizebuffer는 정수형으로 buffer의 크기를 입력한다.<br>Visual C++에서는 앞으로 gets() 대신 gets_s()의 사용을 권장한다                             |
| <code>int puts(const char * str);</code>                      | 함수 puts()는 일반적으로 정수값 0을 반환하는데, 오류가 발생하면 EOF(-1)를 반환한다.                                                                     |

## 11. 문자와 문자열

### 11.2 - 문자열 관련 함수

#### ◎ 문자열 라이브러리와 문자열 비교

- 문자열 비교와 복사, 문자열 연결 등과 같은 다양한 문자열 처리는 헤더파일 `string.h`에 함수원형으로 선언된 라이브러리 함수로 제공되며, 문자열 배열에 관한 다양한 함수는 다음과 같다.

| 함수원형                                                                  | 설명                                                                                                |
|-----------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <code>void *memchr(const void *str, int c, size_t n)</code>           | 메모리 <code>str</code> 에서 <code>n</code> 바이트까지 문자 <code>c</code> 를 찾아 그 위치를 반환                      |
| <code>int memcmp(const void *str1, const void *str2, size_t n)</code> | 메모리 <code>str1</code> 과 <code>str2</code> 를 첫 <code>n</code> 바이트를 비교, 검색하여 같으면 0, 다르면 음수 또는 양수 반환 |
| <code>void *memcpy(void *dest, const void *src, size_t n)</code>      | 포인터 <code>src</code> 위치에서 <code>dest</code> 에 <code>n</code> 바이트를 복사한 후 <code>dest</code> 위치 반환   |
| <code>void *memmove(void *dest, const void *src, size_t n)</code>     | 포인터 <code>src</code> 위치에서 <code>dest</code> 에 <code>n</code> 바이트를 복사한 후 <code>dest</code> 위치 반환   |
| <code>void *memset(void *str, int c, size_t n)</code>                 | 포인터 <code>str</code> 위치에서부터 <code>n</code> 바이트까지 문자 <code>c</code> 를 지정한 후 <code>str</code> 위치 반환 |
| <code>size_t strlen(const char *str)</code>                           | 포인터 <code>str</code> 위치에서부터 널 문자를 제외한 문자열의 길이 반환                                                  |

- 대표적인 문자열 처리 함수인 `strcmp()`는 인자인 두 문자열을 사전상의 순서로 비교하는 함수이며, `strncmp()`는 두 문자를 비교한 문자의 최대 수를 지정하는 함수이다.

※ 비교 방법은 인자인 두 문자열을 구성하는 각 문자를 처음부터 비교해 나가며, 비교 기준은 아스키 코드값이다.  
두 문자가 다른 경우 앞 문자가 작으면 음수, 뒤 문자가 작으면 양수, 같으면 0을 반환한다.  
대문자가 소문자보다 아스키 코드값이 작으므로 `strcmp("java", "javA")`는 양수를 반환한다.

#### ◎ 문자열 복사와 연결

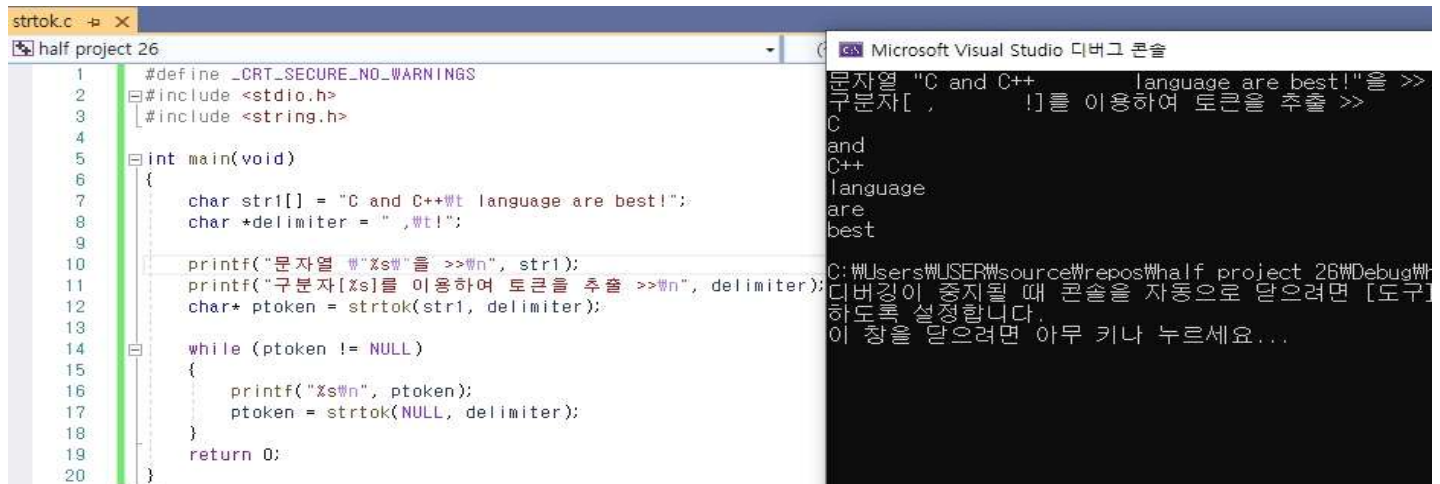
- 함수 `strcpy()`와 `strncpy()`는 문자열을 복사하는 함수이다.

| 문자열 복사 함수                                                                   |                                                                                                                                                                                                                            |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>char * strcpy(char * dest, const char * source);</code>               | 앞 문자열 <code>dest</code> 에 처음에 뒤 문자열 <code>source</code> 를 복사하여 그 복사된 문자열을 반환한다.<br>앞 문자열은 수정되지만, 뒤 문자열은 수정될 수 없다.                                                                                                          |
| <code>char * strncpy(char * dest, const char * source, size_t maxn);</code> | 앞 문자열 <code>dest</code> 에 처음에 뒤 문자열 <code>source</code> 에서 <code>n</code> 개 문자를 복사하여 그 복사된 문자열을 반환한다.<br>만일 지정한 <code>maxn</code> 이 <code>source</code> 의 길이보다 길면 나머지는 모두 NULL 문자가 복사된다.<br>앞 문자열은 수정되지만, 뒤 문자열은 수정될 수 없다. |

- 함수 `strcat()`는 앞 문자열에 뒤 문자열의 NULL 문자까지 연결하여, 앞의 문자열 주소를 반환하는 함수이다.

| 문자열 연결 함수                                                                   |                                                                                                                                                                                    |
|-----------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>char * strcat(char * dest, const char * source);</code>               | 앞 문자열 <code>dest</code> 에 뒤 문자열 <code>source</code> 를 연결해 저장하며, 이 연결된 문자열을 반환하고 뒤 문자열은 수정될 수 없다.                                                                                   |
| <code>char * strncat(char * dest, const char * source, size_t maxn);</code> | 앞 문자열 <code>dest</code> 에 뒤 문자열 <code>source</code> 중에서 <code>n</code> 개의 크기만큼을 연결하여 저장하며, 이 연결된 문자열을 반환하고 뒤 문자열은 수정될 수 없다.<br>지정한 <code>maxn</code> 이 문자열 길이보다 크면 null 문자까지 연결한다. |

- 함수 `strtok()`은 문자열에서 구분자(delimiter)인 문자를 여러 개 지정하여 토큰을 추출하는 함수이다.



```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     char str1[] = "C and C++\t language are best!";
8     char *delimiter = " ,\t!";
9
10    printf("문자열 \"%s\"을 >>\n", str1);
11    printf("구분자[%s]를 이용하여 토큰을 추출 >>\n", delimiter);
12    char* ptoken = strtok(str1, delimiter);
13
14    while (ptoken != NULL)
15    {
16        printf("%s\n", ptoken);
17        ptoken = strtok(NULL, delimiter);
18    }
19    return 0;
20 }
```

Microsoft Visual Studio 디버깅 콘솔

문자열 "C and C++ language are best!"을 >>  
구분자[ ,\t!]를 이용하여 토큰을 추출 >>  
C  
and  
C++  
language  
are  
best

C:\Users\USER\source\repos\half project 26\Debug 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] 하도록 설정합니다.  
이 창을 닫으려면 아무 키나 누르세요...

- 함수 `strlen()`은 NULL 문자를 제외한 문자열 길이를 반환하는 함수이며, 함수 `strlwr()`은 인자를 모두 소문자로 변환하여 반환하고 함수 `strupr()`은 인자를 모두 대소문자로 변환하여 반환하는 함수이다.

## 11. 문자와 문자열

### 11.3 - 여러 문자열 처리

#### ◎ 문자 포인터 배열과 이차원 문자 배열

- 여러 개의 문자열을 처리하는 하나의 방법은 문자 포인터 배열을 이용하는 방법이며, 다른 방법은 문자의 이차원 배열을 이용하는 방법이다.

The screenshot shows a C program in a file named `strarray.c`. The program defines two arrays: `char* pa[]` and `char ca[][5]`, both containing the strings "JAVA", "C#", and "C++". It then uses `printf` to print these arrays. The output in the Microsoft Visual Studio 디버그 콘솔 shows the strings being printed correctly.

```
#include <stdio.h>

int main(void)
{
    char* pa[] = { "JAVA", "C#", "C++" };
    char ca[][5] = { "JAVA", "C#", "C++" };

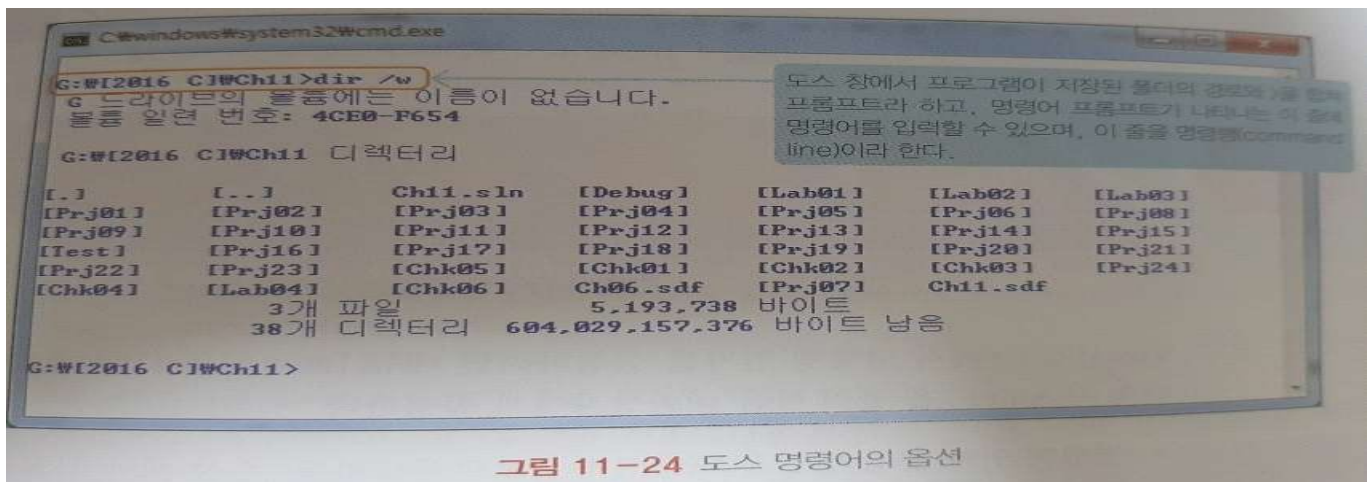
    printf("%s ", pa[0]); printf("%s ", pa[1]); printf("%s\n", pa[2]);
    printf("%s ", ca[0]); printf("%s ", ca[1]); printf("%s\n", ca[2]);

    printf("%c %c %c\n", pa[0][1], pa[1][1], pa[2][1]);
    printf("%c %c %c\n", ca[0][1], ca[1][1], ca[2][1]);

    return 0;
}
```

#### ◎ 명령행 인자

- 다음과 같이 명령행에서 입력하는 문자열을 프로그램으로 전달하는 방법이 **명령행 인자(Command line argument)**를 사용하는 방법이다.



- \* 프로그램에서 명령행 인자를 받으려면 `main()` 함수에서 두 개의 인자 `argc`와 `argv`를 (`int argc, char * argv[]`)로 기술해야 한다. 매개변수 `argc`는 명령행에서 입력한 문자열의 수이며 `argv[]`는 명령행에서 입력한 문자열을 전달받는 문자 포인터 배열이다. 주의할 점은 실행 프로그램 이름도 하나의 명령행 인자에 포함된다는 사실이다.

The screenshot shows a C program in a file named `commandarg.c`. The program takes command line arguments and prints them. The output in the Microsoft Visual Studio 디버그 콘솔 shows the program receiving the full path to the executable as its first argument.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i = 0;

    printf("실행 명령행 인자(command line arguments) >>\n");
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);

    return 0;
}
```

## 12. 변수 유효범위

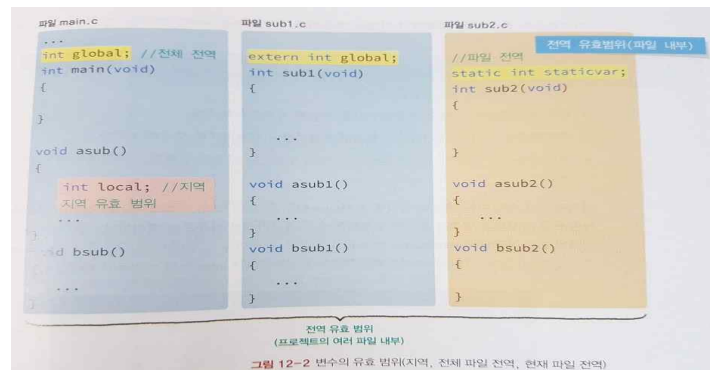
### 12.1 - 전역변수와 지역변수

#### ◎ 변수 범위와 지역변수

- 변수의 참조가 유효한 범위를 **변수의 유효 범위(scope)**라 하며, 유효 범위는 **지역 유효 범위(local scope)**와 **전역 유효 범위(global scope)**로 나눌 수 있다.

※ 지역 유효 범위는 함수 또는 블록 내부에서 선언되어 그 지역에서 변수의 참조가 가능한 범위이다.

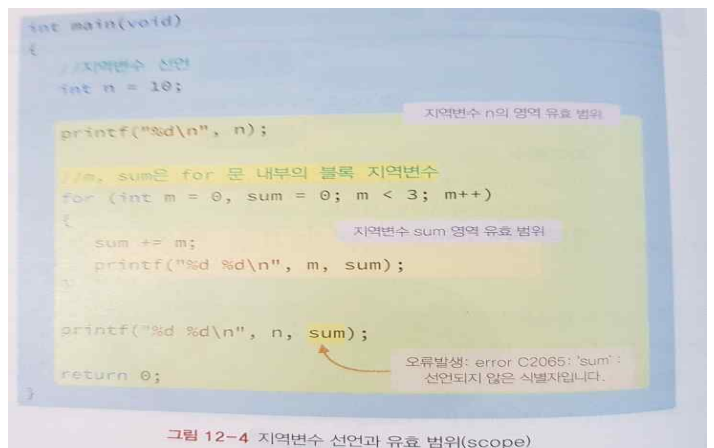
전역 유효 범위는 2가지로 나눌 수 있는데, 하나의 파일에서만 변수의 참조가 가능한 범위와 프로젝트를 구성하는 모든 파일에서 변수의 참조가 가능한 범위로 나눌 수 있다.



- **지역변수**란 함수 또는 블록에서 선언된 변수로서, 내부변수 혹은 자동변수라고도 부른다.

※ 함수나 블록에서 지역변수는 선언 문장 이후에 함수나 블록의 내부에서만 사용이 가능하며, 다른 함수나 블록에서는 사용될 수 없다. 함수의 매개변수도 함수 전체에서 사용 가능한 지역변수와 같으며, 지역변수는 선언 후 초기화하지 않으면 쓰레기값이 저장된다.

- 지역변수가 할당되는 메모리 영역을 **스택(stack)**이라고 부르며, 지역변수는 선언된 부분에서 자동으로 생성되고 함수나 블록이 종료되는 순간 메모리에서 자동으로 제거된다.

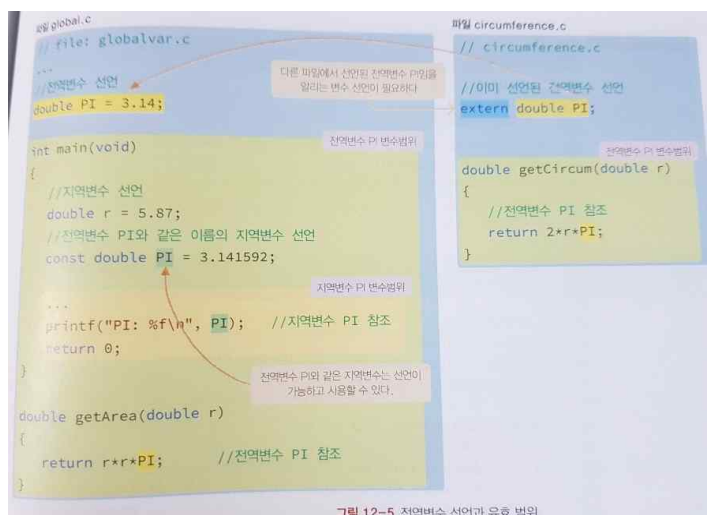


- 전역변수(global variable)는 함수 외부에서 선언되는 변수이며, 외부변수라고도 부른다.

전역변수는 일반적으로 프로젝트의 모든 함수나 블록에서 참조할 수 있다.

※ 전역변수는 선언되면 자동으로 초기값이 자료형에 맞는 0으로 지정된다. 함수나 블록에서 전역변수와 같은 이름으로 지역변수를 선언할 수 있으나 이런 경우 함수내부나 블록에서 그 이름을 참조하면 지역변수로 인식하므로 되도록 사용하지 않도록 한다.

전역변수는 프로젝트의 다른 파일에서도 참조가 가능하지만, 다른 파일에서 선언된 전역변수를 참조하려면 키워드 **extern**을 사용하여 이미 다른 파일에서 선언된 전역변수임을 선언해야 한다.





## 12. 변수 유효범위

### 12.2 - 정적 변수와 레지스터 변수

#### ◎ 기억부류와 레지스터 변수

- 변수 선언의 위치에 따라 변수가 전역과 지역으로 나뉘듯이, 변수는 4가지의 기억부류(storage class)인 auto, register, static, extern에 따라 할당되는 메모리 영역이 결정되고 메모리의 할당과 제거 시기가 결정된다.

※ 전역 변수와 지역 변수는 변수의 선언 위치에 따라 결정되나, 기억 부류는 키워드(auto, register 등)에 의해 구분되므로 구분하기가 쉽다.  
그러나 자동변수인 auto는 일반 지역변수로서 생략될 수 있으므로 주의가 필요하며, 모든 기억 부류는 전역 변수 또는 지역 변수이다.

| 기억부류 종류  | 전역 | 지역 |
|----------|----|----|
| auto     | X  | O  |
| register | X  | O  |
| static   | O  | O  |
| extern   | O  | X  |

- 레지스터(register) 변수는 변수의 저장공간이 일반 메모리가 아니라 CPU 내부의 레지스터(register)에 할당되는 변수이다.

※ 레지스터 변수는 키워드 register를 자료형 앞에 넣어 선언한다.

레지스터 변수는 지역변수에만 이용이 가능하며, 함수나 블록이 시작되면서 CPU의 내부 레지스터에 값이 저장되고, 함수나 블록을 빠져나오면서 값이 소멸되는 특성을 갖는다.

레지스터는 CPU 내부에 있는 기억장소이므로, 일반 메모리보다 빠르게 참조될 수 있어 처리 속도가 빠르다.

레지스터 변수는 일반 메모리에 할당되는 변수가 아니므로 주소연산자 &를 사용할 수 없다.

#### ◎ 정적 변수

- 변수 선언에서 자료형 앞에 키워드 static을 넣어 정적변수(static variable)를 선언할 수 있으며, 정적변수는 정적 지역변수(static local variable)와 정적 전역변수(static global variable)로 나눌 수 있다.

※ 정적변수는 초기에 생성된 이후 메모리에서 제거되지 않으므로 지속적으로 저장값을 유지하거나 수정할 수 있는 특성이 있다.

정적변수는 프로그램이 시작되면 메모리에 할당되고, 프로그램이 종료되면 메모리에서 제거된다.

정적변수는 초기값을 지정하지 않으면 자동으로 자료형에 따라 0이나 '\0', NULL 값이 저장된다.

정적변수의 초기화는 단 한번만 수행되며, 초기화는 상수로만 가능하다.

- 정적 지역변수란 함수나 블록에서 정적으로 선언되는 변수를 말하며, 함수나 블록을 종료해도 메모리에서 제거되지 않고 계속 메모리에 유지 관리되는 특성이 있다.

※ 변수 유효 범위(scope)는 지역변수와 같으나 할당된 저장공간은 프로그램이 종료되어야 메모리에서 제거되는 전역변수의 특징을 갖는다.

함수에서 이전에 호출되어 저장된 값을 유지하여 이번 호출에 사용될 수 있다.

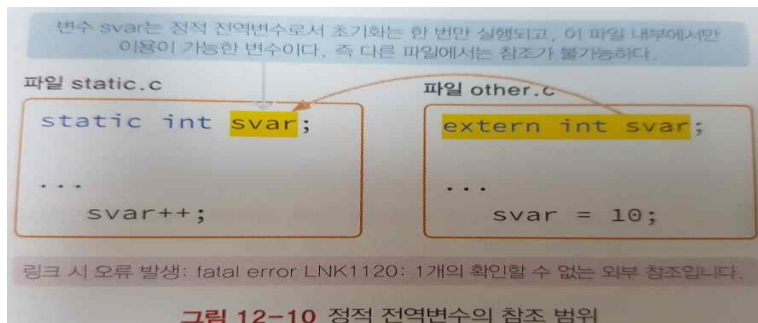
```
staticlocal.c - half project 29
1 #include <stdio.h>
2
3 void increment(void);
4
5 int main(void)
6 {
7     for (int count = 0; count < 3; count++)
8         increment();
9 }
10
11 void increment(void)
12 {
13     static int sindex = 1;
14     auto int aindex = 1;
15
16     printf("정적 지역변수 sindex: %2d\n", sindex++);
17     printf("자동 지역변수 aindex: %2d\n", aindex++);
18 }
```

Microsoft Visual Studio 디버그 콘솔

```
정적 지역변수 sindex: 1,   자동 지역변수 aindex: 1
정적 지역변수 sindex: 2,   자동 지역변수 aindex: 1
정적 지역변수 sindex: 3,   자동 지역변수 aindex: 1

C:\Users\USER\source\repos\half_project_26\Debug\half_pro
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [출
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

- 정적 전역변수란 함수 외부에서 정적으로 선언되는 변수를 말하며, 선언된 파일 내부에서만 참조가 가능한 변수이므로 extern에 의해 다른 파일에서 참조가 불가능하다.





## 12. 변수 유효범위

### 12.3 - 메모리 영역과 변수 이용

#### ◎ 메모리 영역

- 메인 메모리의 영역은 프로그램 실행 과정에서 데이터 영역, 힙(heap) 영역, 스택 영역의 세 부분으로 나뉜다. 이러한 메모리 영역은 변수의 유효범위와 생존기간(life time)에 결정적 역할을 하며, 변수는 기억부류에 따라 할당되는 메모리 공간이 달라진다.
  - ※ 기억부류는 변수의 유효범위(scope)와 생존기간(life time)을 결정한다.  
기억부류는 변수의 저장공간의 위치가 데이터 영역, 힙 영역, 스택 영역인지도 결정하며, 초기값도 결정한다.
  - 데이터 영역**이란 전역변수와 정적변수가 할당되는 저장공간이다.
  - 힙 영역**이란 동적 할당(dynamic allocation)되는 변수가 할당되는 저장공간이다.
  - 스택 영역**이란 함수 호출에 의한 형식 매개변수와 함수 내부의 지역변수가 할당되는 저장공간이다.
- 데이터 영역은 메모리 주소가 낮은 값에서 높은 값으로 저장 장소가 할당되며, 프로그램이 시작되는 시점에 정해진 크기대로 고정된 메모리 영역이 확보된다.  
그러나 힙 영역과 스택 영역은 프로그램이 실행되면서 영역 크기가 계속적으로 변한다.
  - ※ **힙 영역**은 데이터 영역과 스택 영역 사이에 위치하며, 메모리 주소가 낮은 값에서 높은 값으로 사용하지 않는 공간이 동적으로 할당된다.
  - 스택 영역**은 메모리 주소가 높은 값에서 낮은 값으로 저장 장소가 할당되며, 함수 호출과 종료에 따라 높은 주소에서 낮은 주소로 메모리가 할당되었다가 다시 제거되는 작업이 반복된다.

#### ◎ 변수의 이용

- 변수의 특성 및 이용기준과 변수의 종류, 변수의 유효 범위, 변수의 초기값 등은 다음과 같다.
- ※ 변수의 특성 및 이용기준 : 실행 속도를 개선하고자 하는 경우에 제한적으로 특수한 지역변수인 **레지스터 변수**를 이용한다.  
함수나 블록 내부에서 함수나 블록이 종료되더라도 계속적으로 값을 저장하고 싶을 때는 **정적 지역변수**를 이용한다.  
해당 파일 내부에서만 변수를 공유하고자 하는 경우는 **정적 전역변수**를 이용한다.  
프로그램의 모든 영역에서 값을 공유하고자 하는 경우는 **전역변수**를 이용한다.

<변수의 종류>

| 선언위치 | 상세 종류   | 키워드         | 유효범위      | 기억장소                | 생존기간              |
|------|---------|-------------|-----------|---------------------|-------------------|
| 전역   | 전역 변수   | 참조선언 extern | 프로그램 전역   | 메모리<br>(데이터 영역)     | 프로그램 실행 시간        |
|      | 정적 전역변수 | static      | 파일 내부     |                     |                   |
| 지역   | 정적 지역변수 | static      | 함수나 블록 내부 | 레지스터<br>메모리 (스택 영역) | 함수 또는<br>블록 실행 시간 |
|      | 레지스터 변수 | register    |           |                     |                   |
|      | 자동 지역변수 | auto(생략가능)  |           |                     |                   |

<변수의 유효 범위>

| 구분 | 종류      | 메모리 할당시기  | 동일 파일 외부<br>함수에서의 이용 | 다른 파일 외부<br>함수에서의 이용 | 메모리 제거 시기 |
|----|---------|-----------|----------------------|----------------------|-----------|
| 전역 | 전역 변수   | 프로그램 시작   | O                    | O                    | 프로그램 종료   |
|    | 정적 전역변수 | 프로그램 시작   | O                    | X                    | 프로그램 종료   |
| 지역 | 정적 지역변수 | 프로그램 시작   | X                    | X                    | 프로그램 종료   |
|    | 레지스터 변수 | 함수(블록) 시작 | X                    | X                    | 함수(블록) 종료 |
|    | 자동 지역변수 | 함수(블록) 시작 | X                    | X                    | 함수(블록) 종료 |

<변수의 초기값>

| 지역, 전역 | 종류      | 자동 저장되는 기본 초기값                      | 초기값 저장          |
|--------|---------|-------------------------------------|-----------------|
| 전역     | 전역 변수   | 자료형에 따라 0이나 '\0'<br>또는 NULL 값이 저장됨. | 프로그램 시작 시       |
|        | 정적 전역변수 |                                     |                 |
| 지역     | 정적 지역변수 | 쓰레기 값이 저장됨.                         | 함수나 블록이 실행될 때마다 |
|        | 레지스터 변수 |                                     |                 |
|        | 자동 지역변수 |                                     |                 |

## 13. 구조체와 공용체

### 13.1 - 구조체와 공용체

#### ◎ 구조체 개념과 정의

- **구조체**란 연관성이 있는 서로 다른 개별적인 자료형의 변수들을 하나의 단위로 묶은 새로운 자료형을 말하며, 연관된 멤버로 구성되는 통합 자료형으로써 대표적인 유도 자료형이다.  
즉, 기존 자료형으로 새로이 만들어진 자료형을 **유도 자료형(derived data types)**이라고 한다.
- 구조체를 자료형으로 사용하려면 먼저 구조체를 정의해야 하며, 구조체를 만들 구조체 틀을 정의하여야 한다.  
구조체를 정의하는 방법은 키워드 **struct** 다음에 구조체 태그이름을 기술하고 중괄호를 이용하여 원하는 멤버를 여러 개의 변수로 선언하는 구조이며, 구조체를 구성하는 하나 하나의 항목을 **구조체 멤버** 또는 **필드**라 한다.
- \* 구조체 정의는 변수의 선언과는 다른 것으로 변수선언에서 이용될 새로운 구조체 자료형을 정의하는 구문이다.  
구조체 내부의 멤버 선언 구문은 모두 하나의 문장이므로 반드시 세미콜론(;)으로 종료해야 하며, 각 구조체 멤버의 초기값을 대입할 수 없다.  
멤버가 `int credit; int hour;` 처럼 같은 자료형이 연속적으로 놓일 경우, 콤마연산자(,)를 사용해 `int credit, hour;`로도 가능하다.  
한 구조체 내부에서 선언되는 구조체 멤버의 이름은 모두 유일해야 하며, 구조체 멤버로는 일반 변수, 포인터 변수, 배열, 다른 구조체 변수 및 구조체 포인터도 허용된다.

```
struct 구조체태그이름
{
    자료형 변수명1;
    자료형 변수명2;
    ...
};
```

```
struct lecture
{
    char name[20];
    int credit;
    int hour;
};
```

#### ◎ 구조체 변수 선언과 초기화

- 구조체가 정의되면 구조체형 변수 선언 및 초기화가 가능하며, 선언된 구조체형 변수는 접근연산자 `.`를 사용하여 멤버를 참조할 수 있다.

The screenshot shows a C program named `structbasic.c` in Visual Studio. The code defines a `struct account` with members `char name[12];`, `int actnum;`, and `double balance;`. In the `main` function, it creates two `struct account` variables, `mine` and `yours`, and initializes them. It then prints the size of the struct and the values of the members for both variables.

```
1 #define _CRT_SECURE_NO_WARNINGS
2 #include <stdio.h>
3 #include <string.h>
4
5 struct account
6 {
7     char name[12];
8     int actnum;
9     double balance;
10 };
11
12 int main(void)
13 {
14     struct account mine = { "홍길동", 1001, 300000 };
15     struct account yours;
16
17     strcpy(yours.name, "이동원");
18     yours.actnum = 1002;
19     yours.balance = 500000;
20
21     printf("구조체 크기: %d\n", sizeof(mine));
22     printf("%s %d %.2f\n", mine.name, mine.actnum, mine.balance);
23     printf("%s %d %.2f\n", yours.name, yours.actnum, yours.balance);
24
25     return 0;
26 }
```

The output in the console shows the struct size as 24 and the values for `mine` and `yours`.

```
구조체 크기: 24
홍길동 1001 300000.00
이동원 1002 500000.00
C:\Users\WUSER\source\repos\half project\half project>
디버깅이 중지될 때 콘솔을 자동으로 닫으
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

#### ◎ 공용체 활용

- **공용체(union)**란 서로 다른 자료형의 값을 동일한 저장공간에 저장하는 자료형이며, 공용체 변수의 크기는 멤버 중 가장 큰 자료형의 크기로 정해진다.
- \* 공용체의 멤버는 모든 멤버가 동일한 저장 공간을 사용하므로 동시에 여러 멤버의 값을 저장하여 이용할 수 없으며, 마지막에 저장된 단 하나의 멤버 자료값만을 저장한다.  
공용체도 구조체와 같이 `typedef`를 이용하여 새로운 자료형으로 정의할 수 있다.  
공용체의 초기화 값은 공용체 정의 시 처음 선언한 멤버의 초기값으로만 저장이 가능하다.

```
union 공용체태그이름
{
    자료형 멤버변수명1;
    자료형 멤버변수명2;
    ...
} [변수명1] [, 변수명2];
```

```
union udata
{
    char name[4];
    int n;
    double val;
};
```

### 13.2 - 자료형 재정의

#### ◎ 자료형 재정의 typedef

- **typedef**는 이미 사용되는 자료 유형을 다른 새로운 자료형 이름으로 재정의할 수 있도록 하는 키워드이며, 일반적으로 자료형을 재정의하는 이유는 프로그램의 시스템 간 호환성과 편의성을 위해 필요하다.
- 그러나 문장 typedef도 일반 변수와 같이 사용 범위를 제한하기 때문에, 함수 내부에서 재정의된다면 선언된 이후의 그 함수에서만 이용이 가능하고, 함수 외부에서 재정의된다면 재정의된 이후 그 파일에서 이용이 가능하다.

typedef 기존자료유형이름 새로운자료형1, 새로운자료형2, ...;

```
typedef int profit;
typedef unsigned int budget;
typedef unsigned int size_t;
```

The screenshot shows a C program named 'typedef.c' in Visual Studio. The code defines two typedefs: 'profit' as 'int' and 'budget' as 'unsigned int'. In the 'main' function, it declares 'year' as 'budget' and 'month' as 'profit', assigns values 24500000 and 4600000 respectively, and prints them. A 'test' function also declares 'year' as 'budget' and assigns the value 24500000. The console output shows the printed values: '올 예산은 24500000, 이달의 이익은 4600000 입니다.'

#### ◎ 구조체 자료형 재정의

- 예를 들어 구조체 struct date가 정의된 상태에서 typedef를 사용하여 구조체 struct date를 date로 재정의할 수 있다.

The screenshot shows a C program named 'typedefstruct.c' in Visual Studio. It defines a 'struct date' with 'year', 'month', and 'day' fields. Then, it uses 'typedef struct date date;' to create an alias. In the 'main' function, it defines a 'software' struct with 'title', 'company', 'kinds', and 'release' (a 'date' type). It creates a 'vs' variable of type 'software' and prints its fields. The console output shows the details of '비주얼스튜디오 커뮤니티' (Visual Studio Community), including its company (MS), kind (통합개발환경), and release date (2018, 8, 29).

## 13. 구조체와 공용체

### 13.3 - 구조체와 공용체의 포인터와 배열

#### ◎ 구조체 포인터

- 구조체 포인터는 구조체의 주소값을 저장할 수 있는 변수이며, 예를 들어 구조체 자료형 lecture를 선언한 구문에서 구조체 포인터 변수 p는 lecture \*p로 선언된다.

```
struct lecture
{
    char name[20];
    int type;
    int credit;
    int hours;
};
typedef struct lecture lecture;
lecture *p;
```

- 구조체 포인터 멤버 접근연산자 ->는 p->name과 같이 사용하여, 연산식 p->name은 포인터 p가 가리키는 구조체 변수의 멤버 name을 접근하는 연산식이다.

\* 구조체 포인터의 접근연산자인 ->는 두 문자가 연결된 하나의 연산자이므로 -와 > 사이에 공백이 들어가서는 절대 안된다.

연산식 p->name은 접근연산자(.)와 간접연산자(\*)를 사용한 연산식 (\*p).name으로도 사용가능하지만, (\*p).name과 \*p.name은 다르다.

연산식 \*p.name은 접근연산자(.)가 간접연산자(\*)보다 우선순위가 빠르므로 \*(p.name)과 같은 연산식이다.

구조체 포인터 멤버 접근연산자 ->와 구조체 변수의 구조체 멤버 접근연산자 .의 연산자 우선순위는 다른 어떠한 연산자 우선순위보다 가장 높다.

연산자 ->와 .은 연산자 우선순위 1위이고 결합성은 좌에서 우이며, 연산자 \*은 연산자 우선순위 2위이고 결합성은 우에서 좌이다.

| 접근 연산식    | 구조체 변수 os와 구조체 포인터변수 p인 경우의 의미                                                                             |
|-----------|------------------------------------------------------------------------------------------------------------|
| p->name   | 포인터 p가 가리키는 구조체의 멤버 name                                                                                   |
| (*p).name | 포인터 p가 가리키는 구조체의 멤버 name                                                                                   |
| *p.name   | *(p.name)이고 p가 포인터이므로 p.name은 문법 오류가 발생                                                                    |
| *os.name  | *(os.name)를 의미하며, 구조체 변수 os의 멤버 포인터 name이 가리키는 변수로 이 경우는 구조체 변수 os 멤버 강좌명의 첫 문자이지만 한글인 경우에는 실행 오류 발생       |
| *p->name  | *(p->name)를 의미하며, 포인터 p가 가리키는 구조체의 멤버 포인터 name이 가리키는 변수로 이 경우는 구조체 변수 os 멤버 강좌명의 첫 문자이지만 한글인 경우에는 실행 오류 발생 |

#### ◎ 구조체 배열

- 다른 배열과 같이 동일한 구조체 변수가 여러 개 필요하면 구조체 배열을 선언하여 이용할 수 있다.

structarray.c -> X

half project 33

(전역 범위)

```
1 #include <stdio.h>
2
3 struct lecture
4 {
5     char name[20];
6     int type;
7     int credit;
8     int hours;
9 };
10 typedef struct lecture lecture;
11
12 char *lectype[] = { "교양", "일반선택", "전공필수", "전공선택" };
13 char *head[] = { "강좌명", "강좌구분", "학점", "시수" };
14
15 int main(void)
16 {
17     lecture course[] = { { "인간과 사회", 0, 2, 2 },
18                          { "경제학개론", 1, 3, 3 },
19                          { "자료구조", 2, 3, 3 },
20                          { "모바일 프로그래밍", 2, 3, 4 },
21                          { "고급 C프로그래밍", 3, 3, 4 } };
22
23     int arysize = sizeof(course) / sizeof(course[0]);
24
25     printf("배열 크기: %d\n", arysize);
26     printf("%12s %12s %6s %6s\n", head[0], head[1], head[2], head[3]);
27     printf("=====");
28     for (int i = 0; i < arysize; i++)
29         printf("%16s %10s %5d %5d\n", course[i].name,
30             lectype[course[i].type], course[i].credit, course[i].hours);
31
32     return 0;
33 }
```

Microsoft Visual Studio 디버그 콘솔

배열 크기: 5

| 강좌명       | 강좌구분 | 학점 | 시수 |
|-----------|------|----|----|
| 인간과 사회    | 교양   | 2  | 2  |
| 경제학개론     | 일반선택 | 3  | 3  |
| 자료구조      | 전공필수 | 3  | 3  |
| 모바일 프로그래밍 | 전공필수 | 3  | 4  |
| 고급 C프로그래밍 | 전공선택 | 3  | 4  |

C:\Users\WUSER\source\repos\half\_project\_26\Debug 디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도] 하도록 설정합니다.  
이 창을 닫으려면 아무 키나 누르세요...