

C언어를 활용한 임베디드 시스템

목 차

Chapter 01. 임베디드시스템과 마이크로컨트롤러의 이해

1.1 컴퓨터의 구조	1
1.2. 마이크로프로세서와 마이크로컨트롤러	6
1.3. 마이크로프로세서 분류방법	10
1.4. 마이크로컨트롤러 동향	17

Chapter 02. 개발 환경 설정

2.1. 아두이노에 대한 이해	20
2.2. 아두이노 우노, 메가 2560, 멀티 평션 실드	21
2.3. 아두이노 IDE 설치	24
2.4. CH340 드라이버 설치	31
2.5. 아두이노 시뮬레이터	38
2.6. 온라인 C 컴파일러	44

Chapter 03. C와의 만남

3.1 프로그래밍이란 무엇인가?	48
3.2. 프로그래밍 패러다임	50
3.3. 프로그래밍 스타일	51
3.4. C 와 C++	58
3.5. 아두이노 검색하기	60
3.6 Hello World 그리고 Hello Led	63
3.7. 아두이노 코드의 특징	67
3.8. 다양한 라이브러리	70
3.9. 시리얼 모니터	71

Chapter 04. 주석, 변수, 상수

4.1. 아두이노 기본구조.....	77
4.2. 주석.....	78
4.3. 자료형.....	83
4.4. 변수.....	84
4.6. 정수형.....	100
4.7. 실수형.....	110
4.8. 문자형.....	111
4.9. 문제.....	115

Chapter 05. 문자열과 입출력 함수

5.1. 입출력장치와 시리얼 모니터.....	119
5.2. 문자열 처리 방법과 String	123
5.3. PC 와 표준입출력	130
5.4. MCU 시리얼통신을 이용한 입출력	137
5.5. 문자열	147
5.6. 합성, 추출, 교체	151
5.7. 기타 함수.....	156

Chapter 06. 연산자

6.1. 연산자란	163
6.2. 산술 연산자	166
6.3. 할당 연산자, 대입 연산자	177
6.4. 증감연산자	179
6.5. 관계 연산자, 비교 연산자	183
6.6. 논리 연산자	184
6.7. 윤년 판단 예제	186
6.8. 조건 연산자	187

6.9. 콤마 연산자.....	190
6.10. 비트 연산자.....	195
6.11. 연산자의 우선 순위와 결합 규칙.....	204
6.12. Mini Project 화씨 온도를 섭씨로 바꾸기.....	205

Chapter 07. 제어구조

7.1. 제어흐름	209
7.2. if 문	210
7.3. if–else 문	211
7.4. 다중 if 문	213
7.5. switch 문	221
7.6. while 문	229
7.7. do–while 문	240
7.8. for 반복문	242
7.9. 중첩 루프, 무한 루프	246
7.10. break, continue	249
7.11. goto	251

Chapter 08. 아두이노 실습환경

8.1. Multi Function Shield.....	257
8.2. MFS 하드웨어 수정	264
8.3. MFS 부저 알람 프로그래밍	268
8.4. MFS 라이브러리 사용방법	269
8.5. MFS 에서 LED 와 7 Segments 사용.....	274
8.6. MFS 에서 센서 사용.....	277

Chapter 09. 함수

9.1. 함수란.....	282
---------------	-----

9.2. 함수선언	284
9.3. 함수정의	285
9.4. 함수호출	289
9.5. 매개변수 없는 함수	293
9.6. 매개변수 있는 함수	294
9.7. 반환값	298
9.8. void 함수	299
9.9. 변수와 범위	301
9.10. 재귀함수	316
9.11. 함수 이용	319

Chapter 10. 배열

10.1. 배열이란	329
10.2. 배열 선언과 초기화	332
10.3. 배열의 크기	335
10.4. 2차원 배열, 3차원이상 배열	335
10.5. 배열 연습	336

Chapter 11. 포인터

11.1. 포인터의 선언	341
11.2. 포인터 연산	349
11.3. 함수의 포인터	353

Chapter 12. 구조체

12.1. 구조체 기본 개념	357
12.2. 구조체 변수 초기화	362
12.3. 구조체 연산	363
12.4. typedef 구조체 선언	366
12.5. 공용체	367

12.6. 열거형	368
-----------------	-----

Chapter 13. 파일입출력

13.1. 기본 개념	371
13.2. 파일 열기 및 닫기	372
13.3. 파일 읽기와 쓰기	373
13.4. 파일 입출력 예제	373

Chapter 14. 마무리

14.1. 인공지능과 개발자	376
14.2. ChatGPT를 이용한 C코딩	376

남기는 말	377
-------------	-----

Chapter 01

주요 학습 목표

- 임베디드 시스템에 대한 이해
 - 마이크로콘트롤러 구분

Chapter 01. 임베디드시스템과 마이크로컨트롤러의 이해

1.1 컴퓨터의 구조

마이크로프로세서를 이해하기에 앞서 컴퓨터의 구조를 살펴본다. 기본적으로 우리가 눈으로 보는 컴퓨터는 본체와 모니터, 키보드, 마우스 등으로 구성되어 있다. 여기에 유선으로 된 랜 혹은 무선으로 된 WIFI 가 있고, 프린터가 연결된다. 종종 그래픽을 하는 이들은 타블렛이라는 펜모양으로 된 마우스를 대체하는 도구를 연결해서 사용하기도 한다.



수 있게 만들어진 출력장치다. 마우스와 키보드는 컴퓨터에 어떤 신호 혹은 값을 전달 할 수 있는 입력장치다.

본체의 뚜껑을 열어보면 그 안에 보통 메인보드라고 부르는 CPU 와 RAM, 그래픽 카드 등이 함께 있는 큰 PCB 를 볼 수 있다. PCB 와 함께 전력을 공급해 주는 파워서플라이, 하드디스크(HDD) 혹은 SSD(Solid State Drive), CD-ROM 등이 들어있다.

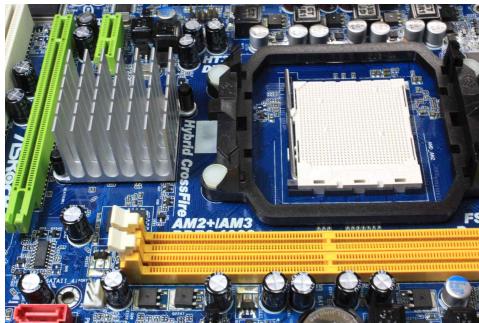


그림 3 PC 내부 1



그림 4 PC 내부 2

메인보드에는 큰 팬이 붙어있다. 이 팬은 CPU 가 과열되는 것을 방지하는 역할을 한다. CPU 는 컴퓨터의 두뇌에 해당된다. 중요한 연산(계산)을 담당하는 것이 CPU 이다. 대부분의 사람들의 PC 에는 인텔 혹은 AMD에서 만들어진 CPU가 장착되어 있다. 현재 개인용 PC에 적용할 수 있는 최고사양의 CPU 는 5GHz 로 작동하는 8개의 코어를 가지고 있다. 즉, 8개의 CPU 들이 하나로 모인 것이라고 생각할 수 있다. 1GHz 라는 속도는 단순한 연산을 1초에 10억번 할수 있다는 뜻이다. 엄청나게 빠른 연산을 해야 하기 때문에 CPU 는 많은 전력을 소모한다. 열이 많이 발생하고, 그 열을 식히기 위해 다양한 방법들이 사용된다. 일반적으로는 팬을 돌려 차가운 공기로 CPU의 열을 식힌다.



그림 5 PC 메인보드와 냉각팬



그림 6 CPU

즉, 컴퓨터는 다양한 많은 장치들이 연결되어서 작동되지만 CPU라는 두뇌역할을 하는 중앙처리장치가 핵심이고 나머지는 보조적인 역할을 한다는 것을 알수 있다.

그래서 컴퓨터를 설명할 때 조금 단순화 시켜서 CPU를 의미하는 중앙처리장치, 마우스, 키보드, 모니터, 프린터 등의 입출력장치, 그리고 주기억장치와 보조기억장치로 설명한다. 주 기억 장치는 메인보드에 꽂아서 사용하는 RAM을 의미하고 보조기억장치는 HDD, SSD 등의 장치를 의미한다.

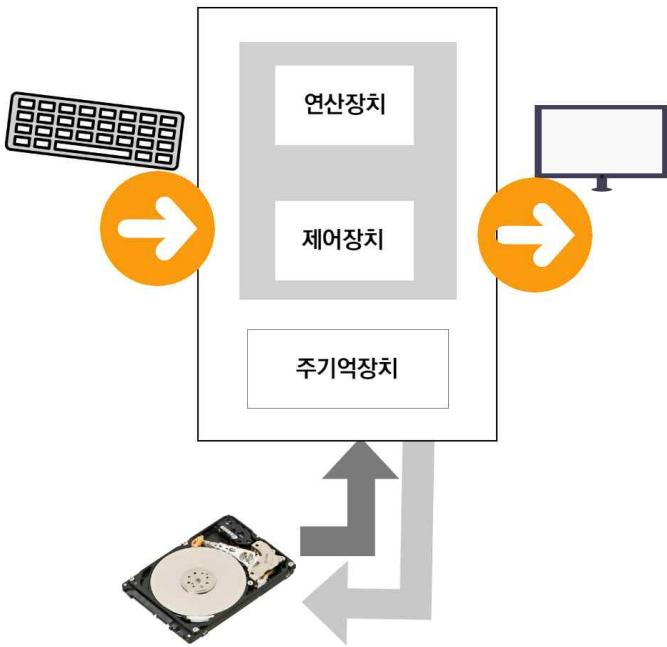


그림 7 컴퓨터의 내부

핵심적인 CPU, 중앙처리장치는 산술논리연산장치(Arithmetic Logic Unit, ALU), 레지스터 및 제어회로로 구성된다. 이러한 회로들이 나뉘어 여러개의 칩(IC)에 있는 것이 아니라 하나의 칩(IC)에 함께 들어있다.

CPU 의 레지스터는 데이터를 임시로 기억하는 장소로, 주기억장치인 RAM 과는 별도인 작은 메모리가 CPU 내부에 있고, 그것을 사용하여 빠른 연산 을 가능하게 한다.

입력장치는 키보드, 마우스, 태블릿, 스캐너, 마이크, 카메라 등이 있으며 다양한 종류의 외부 신호(음성, 영상, 문자, 숫자 등)를 0과 1로 이루어진 이진 데이터로 변환하여 CPU로 전달한다.

CPU 는 HDD 등의 보조기억장치에 저장된 프로그램을 주기억장치인 RAM(Random Access Memory) 으로 불러들인 다음 주기억장치에 있는 프로그램을 작동시킨다.

프로그램의 작동에 따라 입력장치의 신호들이 들어오고, 연산이 이루어진 후 사용자가 인식할 수 있는 형태로 변환되어 출력장치를 통해 나가게 된다. 출력장치는 모니터, 프린터, 플로터, 스피커 등이 있다.

보조기억장치는 데이터와 프로그램을 저장할 수 있는 HDD, SSD 등의 장치를 말합니다.

주기억장치는 CPU가 직접 접근해서 처리할 수 있는 상대적으로 고속의 기억장치이며 프로그램과 데이터를 저장한다. 일반적으로 RAM 을 의미하지만 ROM 도 주기억장치에 포함된다. ROM 은 Read Only Memory 의 약자로 공장출하시에 한번 기록된 내용을 변경할 수 없는 메모리이다. PC의 ROM은 기본 입출력 시스템(BIOS)과 자기진단프로그램 같은 메인보드에 포함되어 운용을 위한 작은 프로그램을 담고 있다.

RAM 은 전원이 꺼지면 내용이 모두 사라지는 휘발성 메모리로 일반적으로 주기억장치라고 부를때는 룬과 램을 의미한다.

마이크로프로세서(Microprocessor)는 중앙처리장치를 하나의 IC로 만들어 둔 것을 말한다. PC에서 사용하는 CPU 도 마이크로프로세서이다. 1971년 인텔에서 4비트 마이크로프로세서인 4004 를 발표하였고 이후 기술의 발전과 함께 마이크로프로세서의 사양은 비약적으로 발전해왔다. 4004가 2250 개의 트랜지스터를 사용하여 만들어졌다면 최신 CPU 에는 50억개 이상의 트랜지스터가 들어간다.

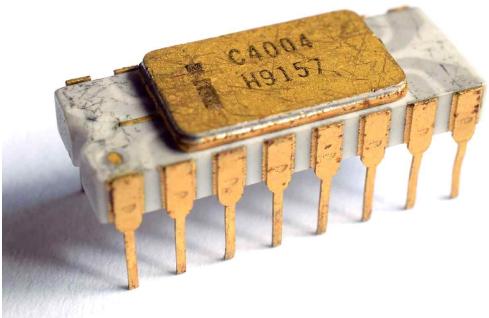


그림 8 인텔 4비트 CPU 4004



그림 9 최신 CPU

1.2. 마이크로프로세서와 마이크로컨트롤러

A. 마이크로프로세서(MPU)

마이크로프로세서의 구조는 CPU에 해당하는 ALU(Arithmetic Logic Unit), 레지스터, 제어장치(Control Unit, CU) 등으로 구성된다.

제어장치(CU)에는 명령어의 주소를 기억하는 레지스터인 프로그램 카운터(Program Counter, PC), 수행할 명령어를 해석해서 부호기로 전달하는 명령 해독기(Instruction Decoder, ID), 전송된 명령어를 제어 신호로 변환하는 부호기(Encoder), 현재 수행중인 명령어를 기억하는 명령 레지스터(Instruction Register, IR), 주소를 기억하는 번지레지스터(Memory Address Register, MAR)와 내용을 기억하는 기억레지스터(Memory Buffer Register, MBR)이 있다.

연산장치(ALU)는 명령을 실행하기 위한 최소연산을 수행하는 장치로 산술 연산, 논리연산, 비트연산 등의 다양한 실제 연산을 수행한다. 여기엔 2진 덧셈을 수행하는 가산기(Adder), 중간 연산 결과를 일시적으로 저장하는 누산기(Accumulator), 뺄셈을 위해 이진 보수 변환을 위한 보수기

(Complementor) 등이 있다.

레지스터(Register)는 CPU 안에 있는 작은 임시기억장소로 메모리 중에서 가장 속도가 빠르다. 레지스터 중 일부는 특별한 기능을 가지고 있으며, 스위치처럼 작동하게 된다. 예를들면 아두이노의 0번 핀을 입력으로 사용하려면 DDRD 레지스터의 맨 마지막 비트를 0으로 만들어주면 된다. 반대로 출력으로 사용하려면 DDRD 레지스터 마지막 비트를 1로 만들어준다. 마지막비트를 1로 만들때 다음과 같이 프로그래밍한다. (지금 아래 코드를 설명하지는 않는다. 이후 C언어를 다룰때 아래와 같은 형태의 구문을 설명한다.)

```
DDRD = DDRD | B00000001;
```

아래 그림은 아두이노 우노에 사용된 Atmel 사의 ATmega328P의 다이어그램이다. 여기서 AVR CPU라고 적힌 박스에 ALU, Register, Decoder 등이 포함되어 있으며, 이 부분이 마이크로프로세서에 해당된다.

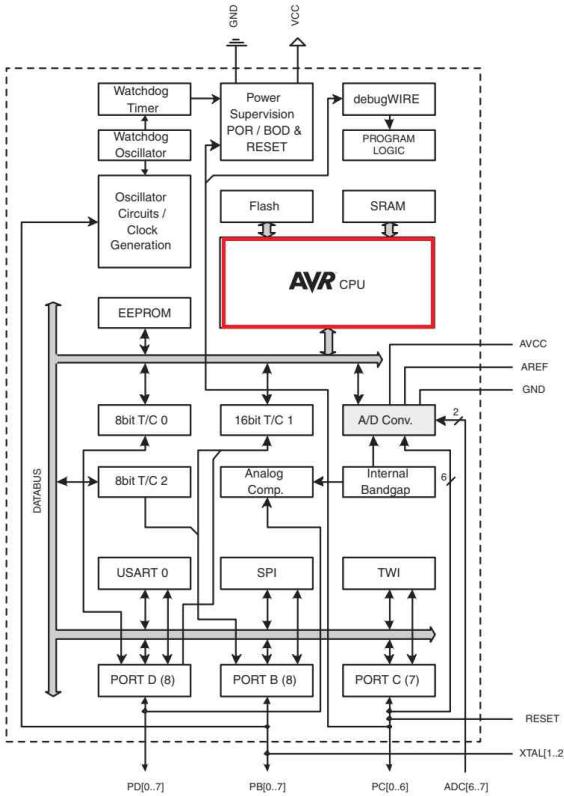


그림 10 AVR MCU 내부 데이터시트 내용

B. 마이크로컨트롤러(MCU)

점선으로 된 부분은 마이크로프로세서에 연결되는 RAM(SRAM), ROM(Flash)과 함께 Timer, SPI, I2C, ADC, UART 등이 포함되어 있으며, 이 모든것이 하나의 칩 안에 포함되어 있다. 이것을 마이크로컨트롤러 (MicroController Unit, MCU)라고 한다. PC에 사용되는 팬티엄 CPU는 마이크로프로세서이기 때문에 외부에 별도의 램과 룸이 있어야하고, HDD 등의 보조기억장치가 필요하다. 하지만 아두이노에 사용되는 ATmega328P는 마이크로프로세서 부분과 함께 플래시메모리와 램등이 하나의 IC 안에 포함되어 있기 때문에 별도의 램이나 룸을 외부에 장착할 필요가 없다. 마이크로컨트롤러는 별도의 주변 IC 없이 하나의 IC만으로 원하는 동작을 구

현할 수 있게 되어있다. 성능을 중요시하는 마이크로프로세서와 달리 마이크로컨트롤러는 성능보다는 소형화와 저전력화에 집중한다. 단적으로 4GHz라는 속도를 가진 8코어 짜리 인텔 CPU에 비해 ATmega328P는 16MHz의 속도에 하나의 코어만을 가지고 있다($1\text{GHz} = 1,000\text{ MHz}$). 단순비교로도 인텔 CPU와 아두이노 MCU는 2,000 배의 차이가 있다($8 \times 4\text{GHz} = 2000 \times 16\text{MHz}$).

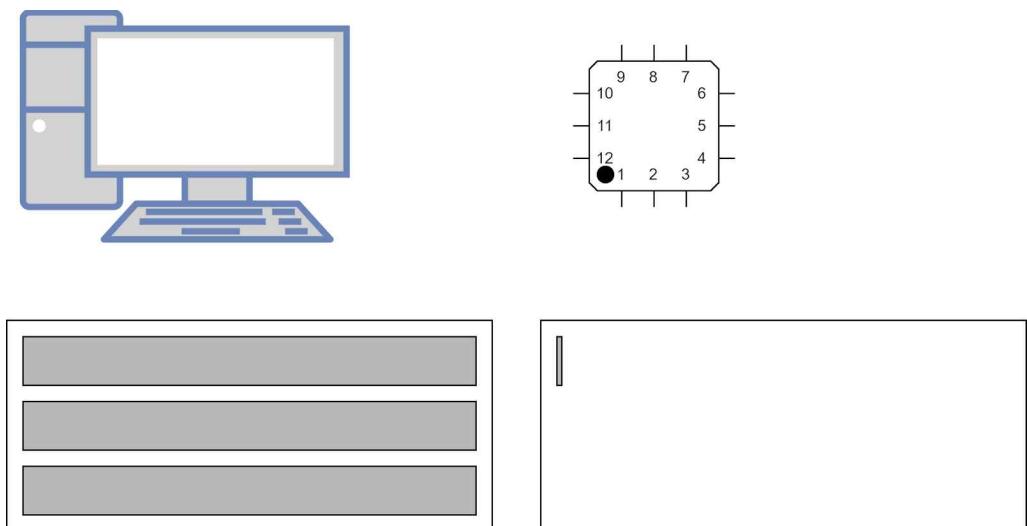


그림 11 8 core 4GHz는 16MHz MCU 의 2000 배

그리면 빠른 속도를 가진 인텔 CPU 가 아두이노 MCU 보다 좋은 것인가? 이에 대해서는 용도에 따른 접근법이 필요하다. 영상처리를 위한 머신러닝 프로그램을 가동시킨다면 빠른 속도를 가진 고사양의 인텔 CPU를 사용하는 것이 옳다. 밥솥은 만들어 정해진 시간에 전열장치를 끄는 정도의 용도라면 PC에 사용되는 고성능 CPU를 사용하기보다는 저렴하고 저전력에서 동작하며 주변 기능이 하나의 칩에 모두 포함되어 있는 MCU를 사용하는 것이 낫다. 즉, 단순한 연산과 센서 값의 처리와 주변 기기 제어등의 역할을 하고 배터리 등으로 작동한다면 MCU를 사용하는 것이 좋다.

마이크로프로세서와 마이크로컨트롤러를 최근 들어 구분없이 사용하기도

한다. 하지만 마이크로프로세서는 단독으로 동작하지 못하는 CPU 외부에 메모리와 주변장치들이 연결되어 동작하는 것이고, 마이크로컨트롤러는 CPU IC 내부에 메모리와 주변장치들이 포함되어 있어 단독 동작이 가능한 것이다. 마이크로컨트롤러는 상대적으로 처리속도가 느리고, 크기가 작고, 전력소비가 적으며, 단일칩으로 구성되고, 가격이 저렴하다.

1.3. 마이크로프로세서 분류방법

A. 구조에 따른 분류

CISC (Complex Instruction Set Computer)

CISC 는 복잡하고 기능이 많은 명령어로 구성되어 있다. 고급언어의 사용이 어려워서 어셈블러를 사용해야 했던 시기에는 프로그래머가 개발하기 쉽도록 하드웨어를 만들어야 했다. 즉, 프로그래머들이 원하는 명령이 생기면 그 명령을 수행하는 하드웨어를 만드는 방식이다. 이렇게 만들어지는 MPU 들은 처음에 있던 것과는 조금씩 달라진 명령들이 쌓이게 되고 그 명령들은 예전에 사용하던 것들과 유사하면서도 별도의 공간을 차지하는 방식으로 개발되어졌다. 결과적으로 성능 향상은 곧 이전보다 조금 더 많은 명령어들을 지원하는 방향으로 개발이 이루어졌다.

CISC 에서는 모든 명령어에 대해 1:1로 기계어 명령이 존재한다. 명령어 개수의 증가에 따라 프로세서의 내부구조가 복잡해지고, 고속으로 작동하는 프로세서를 만들기 어려워졌다. 인텔 계열의 8051 이 대표적인 CISC 방식을 사용하고 있다.

CISC 명령셋	A	B	C	D
AB : A → B		A----->B		
AC : A → C		A----->C		
AD : A → D		A----->D		
BC : B → C			B----->C	
BD : B → D			B----->D	
CD : C → D			C----->D	

그림 12 CISC 명령셋

A를 B로 만드는 명령이 AB이고, B를 C로 만드는 명령이 BC일 때, A를 C로 만들기 위해서 AB와 BC를 사용하면 되겠지만 두 명령을 수행할 때 걸리는 시간과 프로그래밍에 사용되는 어려움을 고려해서 그냥 AC 명령을 따로 만들어 사용한다. 마찬가지로 AD 명령도 AB, BC, CD를 조합해서 만들면 되지만 현실적인 어려움 때문에 AD 명령을 만들어 사용한다. 이때 AB 명령에 비해 AD 명령의 간격이 훨씬 큰 것이 보이는 것처럼 실제 AD 명령은 AB 명령에 비해 더 복잡해진다. 그래서 CISC는 Complex Instruction, 복잡한 명령구조를 가지게 된다.

RISC (Reduced Instruction Set Computer)

RISC는 CISC보다 늦게 만들어졌다. CISC의 단점을 극복하기 위해 만들어진 RISC는 CISC와 비교해 비교적 작은 단순화된 명령어들의 조합으로 복잡한 명령어를 구성할 수 있게 만들었다. RISC가 등장할 때의 컴퓨터 환경은 상당히 많이 달라졌다. C와 같은 고급언어들이 많아졌고, 반도체 칩을 만드는 공정도 많은 개선이 이루어져서 속도가 빨라졌다. 반도체의 처리 속도가 빨라지고, C 언어가 안정화되면서 AC 명령을 AB + BC의 조합으

로 사용하는 것이 가능해졌다. 즉, 다음과 같이 AC, AD, BD 과정을 생략한 명령구조만을 가진 보다 간편한 구조의 칩이 생산되기 시작했다.

RISC 명령셋	A	B	C	D
AB : A → B		A----->B		
BC : B → C			B----->C	
CD : C → D				C----->D

그림 13 RISC 명령셋

AC, AD, BD 는 칩 내부에는 그 명령이 없지만 기존의 명령을 조합해서 구성할 수 있을 뿐만 아니라 실행성능도 많이 떨어지지 않게 되었다. RISC 구조가 보편화되면서 컴파일러들의 최적화기법도 동반해서 발전하게 된다. RISC 구조를 가지게 되면서 상대적으로 여유를 가지게 된 MPU 는 그 여유분을 메모리추가와 속도향상을 위해 사용하게 된다.

RISC 명령어 조합	A	B	C	D
AC : AB+BC	A----->B, B----->C			
AD : AB+BC+CD	A----->B, B----->C, C----->D			
BD : BC+CD		B----->C, C----->D		

그림 14 RISC 명령어 조합

아두이노에 사용되는 Atmeg328p, Atmega2560 등은 대표적인 RISC 구조의 MCU 로 CPU 의 명령을 잘게 쪼개 단순하게 설계한 마이크로프로세서이다. 모든 명령은 논리회로로 구성되며 명령어가 복잡하거나 많다는 것은 명령을 구성하는 논리회로가 많아진다는 것이고, 논리회로가 많아진다는

것은 내부 트랜지스터 개수가 더 많이 필요하다는 것을 의미한다. CPU 의 명령어 개수를 줄여 하드웨어 구조를 간단하게 만드는 방식인 RISC 는 최근 마이크로프로세서를 설계할 때 많이 사용되는 방식이다. 특징으로는 고정소수점(Fixed Point)방식의 명령어를 사용하여 명령어를 해석, 처리하는 속도를 빠르게 했으며, 파이프라인을 사용하여 하나의 클럭에 하나의 명령을 처리하도록 구성하고 있다. CISC 방식에 비해 동일성능이라면 상대적으로 가격이 저렴하고 발열이 적다. 대표적인 RISC 방식은 ARM 계열이며 아두이노에 사용된 AVR MCU 도 RISC 방식을 채택하고 있다.

	CISC - ex) 8051	RISC - ex) Atmega328 (아두이노)
구조	복잡	단순
구성	(복합적으로 구성된) 많은 명령어	(단순하게 구성된) 기본 명령어
길이	명령어 별로 다름	고정
레지스터 수	작음	많음
실행속도	느림	빠름

그림 15 CISC와 RISC 비교

B. 성능에 따른 분류

마이크로프로세서를 처리능력에 따라 비트의 수로 구분할 수 있다. 인텔의 4004 는 4비트 마이크로프로세서로, 이후 8비트, 16비트, 32비트, 64비트의 순으로 새로운 마이크로프로세서가 등장했다. 비트는 CPU 와 RAM 사이에 연결된 데이터버스(Data Bus)의 크기에 따라 구분한다. 현재 4비트 마이크로프로세서는 사용되지 않고 있으며, 저사양, 저전력을 위한 8비트 마이크로프로세서와 고사양으로 32비트와 64비트가 주로 사용되는 추세이다.

a. 8비트 프로세서

한번에 주고 받을 수 있는 데이터버스의 크기가 8비트로 Atmel 의 AVR 시리즈, PIC 의 PIC 시리즈, Intel 의 8051 시리즈가 대표적인 8비트 프로세서이다. 아두이노는 8비트 프로세서인 ATmega328p 를 사용한 아두이노 우노와 ATmega2560 을 사용한 아두이노메가2560 이 있다.

b. 16비트 프로세서

데이터버스의 크기가 16비트로, TI사의 MSP430 시리즈, Intel의 80196 시리즈등이 있다. 32비트 프로세서가 나오기 전까지는 16비트 프로세서가 고사양의 프로세서로 많이 사용되었으나 32비트, 64비트가 등장하면서 16비트는 중간에 낀 애매한 위치가 되었다. 일부 제조사에서는 8비트 대신 16비트를 저사양 프로세서로 제조하기도 한다.

c. 32비트 프로세서

데이터버스의 크기가 32비트인 프로세서다. 상용화되고 있는 대부분의 32비트 프로세서는 ARM 계열이 많다. 대표적인 프로세서로는 ST 의 STM32F10xx 시리즈, 인텔의 x86 계열인 80386, 80486, 팬티엄등이 있다. 아두이노 DUE 는 ARM 계열의 32비트 AT91SAM3X8E 프로세서를 사용한다.

d. 64비트 프로세서

데이터버스의 크기가 64비트인 프로세서다. 상용화되고 있는 64비트 프로세서는 인텔의 팬티엄4 이후로 나오는 프로세서들이다.

C. 제조사에 따른 분류

a. INTEL 계열

8051 은 인텔에서 개발한 8비트 MCU 이다. 이후 많은 회사에서 8051 코어를 이용하여 MCU 를 제작해왔다. 인텔 8051 은 내부에 메모리가 없기 때문에 외부에 별도의 RAM 과 ROM 을 장착해야만 했다. 이런 단점을 보완하기 위해 ATMEL 에서는 플래시메모리를 내장한 8051 을 만들어었다. AT89C5x, AT89S5x 시리즈는 8051 코어를 가지고 내부에 플래시메모리 와 ISP 기능등을 포함한 제품이다. 또한 인텔은 전세계 PC 시장에서 4분의 3이상을 차지한다. PC에 사용되는 마이크로프로세서는 이 책에서 다루지 않는다. 단, 윈도우 환경을 사용하는 산업용 PC 등이 등장하면서 개인용 PC 와 서버만이 아닌 공장 제어용 PC 도 인텔의 고성능 CPU 를 사용하는 경우가 늘어나고 있는 추세이다.

b. AVR 계열

AVR 은 ATMEL 에서 개발된 MCU로 8비트 프로세서다. 아두이노에 사용된 이후로 더 유명해졌지만 이전에도 산업용 마이크로컨트롤러도 많이 사용되고 있었다. AVR 은 ISP(In System Programming) 와 ADC(Analog Digital Converter) 기능이 일반적으로 포함되어 있다. 별도의 ADC 칩을 외부에 장착해야 했던 다른 MCU 들에 비해 AVR 은 저렴한 가격으로 별도의 ADC 칩 없이 가벼운 ADC 를 처리할 수 있다. ISP 는 루م에 프로그램 을 넣는 장비인 루미레이저(EPROM Eraser) 와 루라이터(EPROM Programmer)를 사용하지 않고 PC 와 간단한 케이블 연결만으로 EPROM 대신 Flash Memory 에 프로그램을 지우거나 쓸수 있게 했다. EPROM Eraser 와 EPROM Programmer 없이 MCU 를 사용할 수 있게 된 것은 개인 개발자들에게 큰 기회였다. 기업 연구소나 대학 연구실이 아닌 개인도 얼마든지 MCU 를 사용할 수 있게 되었다.

c. PIC 계열

PIC는 마이크로 칩 테크놀로지(Micro Chip Technology)에서 개발한 RISC 구조의 8비트 마이크로컨트롤러다. PIC는 상대적으로 저렴한 8비트 시리즈와 16비트, 32비트 시리즈를 모두 갖추고 있다. 8비트 MCU 중에서는 가장 저렴한 편에 속하므로 저가의 기기나 아동용 장난감들에 많이 사용된다. 아두이노가 등장하기 전까지 메이커들이 가장 많이 사용하는 MCU이기도 하다. 다양한 사양과 가격대의 시리즈를 구성하고 있다.

d. TI 계열

텍사스 인스트루먼트(Texas Instrument, TI)사는 디지털 신호처리 분야에 강점을 가지고 있다. 신호처리, 영상처리, 음성처리, 무선통신 등의 시스템에 사용되는 고사양의 프로세서를 주로 생산하며 최근 들어 MSP430 등의 저사양 MCU도 출시하고 있다.

e. Freescale 계열

68HC11 시리즈는 모토롤라의 6800 시리즈로 8비트 프로세서이다. 역시 8비트 프로세서 외에 32비트 프로세서도 출시하고 있다.

f. ARM 계열

ARM 코어를 사용하여 개발되는 프로세서로 다른 제조사들이 설계와 생산을 함께 하는 것과 달리 ARM은 코어 설계만을 하고 제조회사에 코어 라이선스를 제공한다. 이런 이유는 ARM 프로세서는 여러 회사에서 생산되고 있다.

1.4. 마이크로컨트롤러 동향

마이크로컨트롤러는 임베디드 산업분야에서 빠르게 발전하고 있다. 4차 산업혁명시대를 맞이하면서 사물인터넷(Internet of Things, IoT), 빅데이터(Big Data), 클라우드(Cloud), 머신러닝(Machine Learning, ML) 기술이 발전하면서 스마트워치, 구글 글래스, 자율주행차등의 실용화와 함께 마이크로컨트롤러의 사용은 더욱 증가하는 추세다. IT 기술의 발전은 마이크로프로세서/마이크로컨트롤러의 성능을 크게 향상 시켰으며, 암호화와 보안 등의 필요한 기술과 함께 Soc(System on Chip) 설계기술의 발전으로 가격은 더 저렴해지고 소형화되면서 고성능으로 발전하고 있다. 한 조사기관에 따르면 앞으로 8비트 MCU는 점점 사라지고, 그 자리를 16비트 MCU가 차지할 것으로 시장을 예측하고 있다.

마이크로컨트롤러(MCU) 분야의 미래 동향은 기술 발전과 시장 요구에 따라 계속 변화하고 있습니다. 몇 가지 주요 동향은 다음과 같다.

IoT 및 연결성: 마이크로컨트롤러는 인터넷의 사물과 연결되는 IoT(사물인터넷) 디바이스에서 중요한 역할을 합니다. 미래에는 더 많은 MCU가 무선 통신 및 저전력 기술을 지원하고, 다양한 센서와 통신 프로토콜과 통합된 솔루션을 제공할 것으로 예상

AI와 머신 러닝: AI 및 머신 러닝 응용 프로그램이 증가함에 따라 MCU에서는 AI 가속 하드웨어 및 소프트웨어가 통합되는 경향이 있습니다. 이로써 MCU는 실시간 데이터 분석 및 의사 결정을 수행할 수 있게 될 것

보안 강화: IoT 장치의 증가와 함께 보안 문제도 중요해지고 있습니다. MCU 제조업체는 하드웨어 기반 보안 및 소프트웨어 보안 솔루션을 통합하여 디바이스의 안전성을 강화하고 데이터 유출을 방지하기 위해 노력하고 있습니다.

저전력 MCU: 저전력 MCU는 배터리 수명을 연장하고 에너지 소비를 최소화하는 데 중요합니다. 미래에는 더욱 효율적인 저전력 MCU가 출시되어

더 오래 작동하는 IoT 디바이스를 가능하게 할 것입니다.

멀티코어 및 다중 작업: 복잡한 응용 프로그램을 처리하기 위해 멀티코어 MCU가 보다 일반적으로 사용될 것으로 예상됩니다. 이는 실시간 운영체제 및 병렬 처리를 지원하며 성능을 향상시킬 것입니다.

환경 친화적 디자인: 환경 문제의 중요성이 높아지면서, MCU 제조업체는 더욱 친환경적이고 에너지 효율적인 제품을 개발하는 방향으로 나아갈 것입니다.

생산성 향상: MCU 개발과 프로그래밍 과정을 단순화하고 생산성을 향상하기 위한 툴체인 및 개발 환경의 개선이 예상됩니다.

마이크로컨트롤러 기술은 계속 발전하고 있으며, 다양한 응용 분야에서 사용되고 있습니다. 미래에는 더 많은 혁신과 기술 발전이 예상되며, MCU 제조업체 및 개발자들은 이러한 동향을 주시하고 적용하여 미래의 MCU 디바이스를 발전시킬 것입니다.

Chapter 02

주요 학습 목표

- 아두이노 이해
- 개발환경 설치

Chapter 02. 개발 환경 설정

2.1. 아두이노에 대한 이해

아두이노(Arduino)는 오픈소스를 기반으로 한 단일 보드 마이크로컨트롤러로 만들어진 개발용 보드다. 2005년 이탈리아에서 활동하던 마시모 반지(Massimo Banzi)는 이브레아(Interaction Design Institute Ivrea, IDII)의 교수였다. 학생들에게 현대 예술, 특히 Interaction Art 를 강의하면서 학생들이 기존의 MCU 보드 사용에 어려움이 있다는 것에 착안해서 저렴하고 쉽게 사용할 수 있는 보드를 고안하기 시작했고, 이렇게 만들어진 보드를 공개해서 누구나 사용할 수 있도록 했다. 아두이노는 이탈리아의 왕이었던 이브레아 아르두노(Arduino d'Ivrea)의 이름에서 유래한 '아두인왕의 술집(Bar di Re Arduino)'에서 유래했다.

아두이노 하드웨어 및 소프트웨어는 공개 라이센스로 누구나 이용이 가능하다. 아두이노 하드웨어는 디자인을 포함해서 하드웨어를 누구나 개발, 수정, 유통, 제작, 판매까지 공개적으로 가능하다. 소프트웨어 또한 기존의 IAR, KEIL 등의 고가의 상용 소프트웨어 없이는 MCU 프로그램이 어려웠던 것과 달리 gcc 컴파일러를 사용함으로 무료 통합 개발 환경(Integrated Development Environment, IDE)로 구성되어 있다. 아두이노가 발표된 이후 세계의 개발 커뮤니티에서는 정보와 다양한 기술 솔루션을 공유하고 있다.

아두이노는 대표적인 UNO 부터 최근의 IoT 를 적용한 다양한 제품군이 존재한다. ARM 계열의 Cortex-M0를 사용한 Arduino M0 Pro와 Cortex-M3를 사용한 32비트 Arduino Due 제품도 있다.

아두이노의 공식 웹 사이트는 <https://www.arduino.cc> 이며, 다양한 보드에 관련된 데이터를 볼수 있으며 IDE 프로그램을 다운받아 사용할 수 있다. 아두이노는 마이크로컨트롤러로 만들수 있는 다양한 프로젝트를 만들 수 있다. 스위치와 센서로부터 값을 받아들여, LED나 모터와 같은 외부 전자 장치들을 제어할 수 있으며, 상호작용이 가능한 물건을 만들어낼 수 있다.

2.2. 아두이노 우노, 메가 2560, 멀티 평션 쉴드

우선 이 책의 실습을 위해 준비할 것은 아두이노 우노와 멀티 평션 쉴드이다. 두가지 모두 시중에서 어렵지 않게 구입할 수 있다. 온라인 쇼핑몰(네이버 스마트스토어, 쿠팡, 지마켓, 11번가 등)에서 'multi function shield' 를 검색하면 된다. 혹은 해외 주문을 할 경우 aliexpress.com 에서 'multi function shield' 를 검색해서 구입하면 된다. 아두이노 우노를 사용하여 대부분의 프로젝트를 진행할 것이며 후반부에 아두이노 메가 2560 을 사용하게 된다. 아두이노 메가 2560 은 아두이노 우노에 비해 램과 플래시메모리의 크기가 커서 프로젝트를 진행할 때 메모리로 인한 오류 발생은 거의 없다. 또한 IO 핀의 갯수도 디지털 I/O 가 54개, 아날로그 입력이 16개로 아두이노 우노의 디지털 I/O 14개, 아날로그 입력 6개보다 충분히 많기 때문에 확장성이 뛰어나다. 우노가 있으면 우노로도 가능하지만 가능하면 아두이노 메가 2560 을 준비하여 실습하는 것을 추천한다.



그림 16 우노



그림 17 메가2560



그림18 멀티펑션실드

아두이노 우노와 메가 2560 의 간단한 스펙을 비교하면 다음과 같다.

	아두이노 우노	아두이노 메가 2560
Microcontroller	ATmega328P	ATmega2560
Operating Voltage	5V	5V
Input Voltage	7~12V	7~12V
Digital I/O Pins	14 (6 PWM output)	54 (15 PWM output)
Analog Input Pins	6	16
DC Current per I/O Pin	20 mA	20 mA
DC Current for 3.3V Pin	50 mA	50 mA
Flash Memory	32 KB (0.5 KB bootloader)	256 KB (8 KB bootloader)
SRAM	2 KB	8 KB
EEPROM	1 KB	4 KB
Clock Speed	16 MHz	16 MHz
LED_BUILTIN	13	13
Length	68.6 mm	101.52 mm
Width	53.4 mm	53.3 mm
Weight	25 g	37 g

플래시 메모리 용량이 우노는 32 KB, 메가 2560은 256 KB로 8배 차이가 난다. SRAM, EEPROM, IO 등에서 우노에 비해 메가 2560은 넉넉한 용량을 가지고 있다.

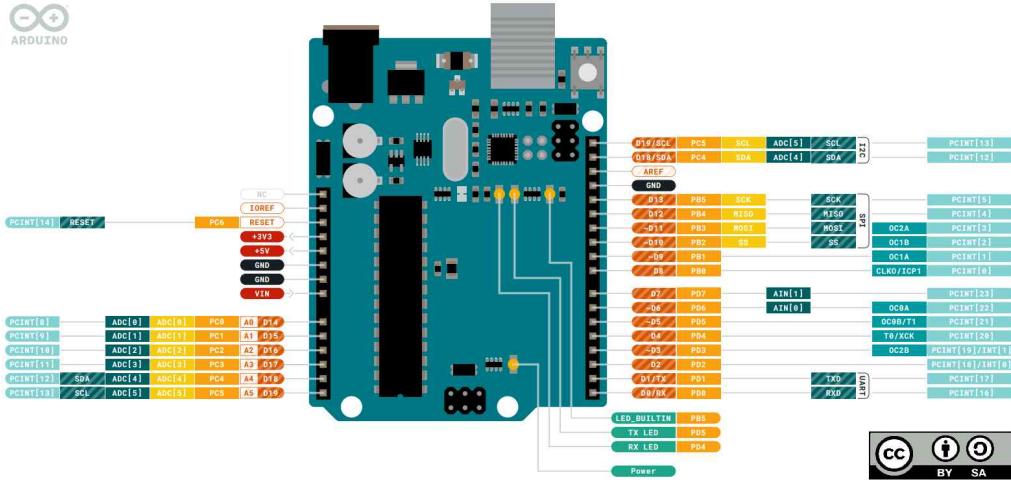


그림 19 아두이노 우노 펀맵

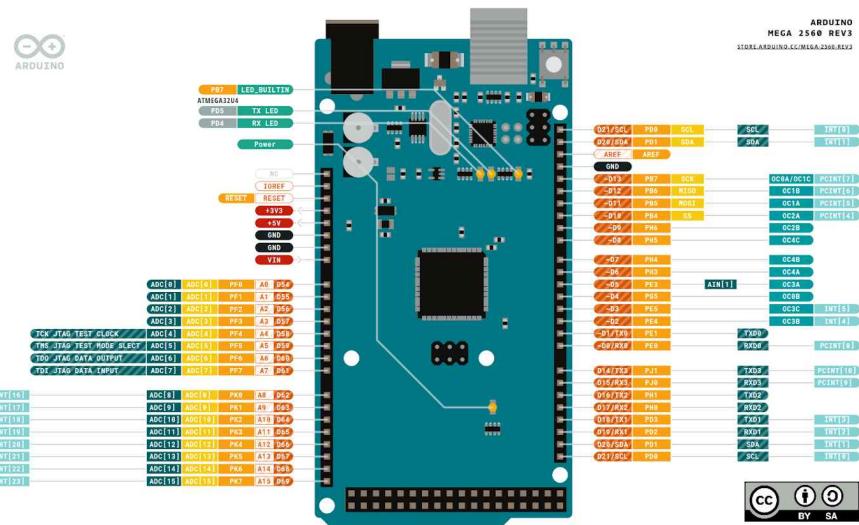


그림 20 아두이노 메가2560 핀맵

실제로 우노의 핀맵은 메가 2560 과 동일하다. 메가 2560 은 우노핀맵에 자체 핀맵을 별도로 추가되어 있는 형태다. 그래서 우노용으로 제작된 실드를 사용할 수 있다. 멀티펑션실드는 아두이노 레오나르도, 아두이노 우노,

아두이노 메가 2560 등에서 사용할 수 있다.

2.3. 아두이노 IDE 설치

아두이노에 프로그램하기 위해서는 프로그램툴을 PC에 설치해야 한다. 이런 프로그래밍 툴을 컴파일러라고 부른다. 컴파일러는 인간이 만든 소스코드를 기계(컴퓨터)가 이해할 수 있는 기계어로 바꾸는 프로그램이다. PC에서 사용되는 일반적인 C 컴파일러로는 Dev C++, MinGW, Codeblocks, Visual Studio 등이 있다. 기본적으로 컴파일러는 설치된 환경에 맞는 기계어로 컴파일한다. 즉, 윈도우10 환경에서 컴파일러를 설치한 후 컴파일하면 윈도우10에 맞는 기계어로 컴파일된 이진파일(실행 파일)이 만들어진다. 맥이나 리눅스도 마찬가지다.

아두이노는 8비트 AVR MCU를 사용한다. 8비트 AVR은 이런 컴파일러를 실행시키기에는 리소스가 턱없이 부족하기 때문에 직접 컴파일할 수 없다. 그래서 다른 환경에서 컴파일한 후 컴파일된 이진 파일을 AVR 칩에 넣는 식으로 진행된다. 이때 사용되는 컴파일러를 크로스컴파일러라고 부른다.

아두이노는 gcc 컴파일러를 사용하고, 여기에 코드 작성에 필요한 간단한 에디터와 디버깅에 필요한 기능을 포함한 통합개발환경을 만들어서 제공한다. 아두이노를 사용할 때 필요한 기본적인 것은 아두이노 공식 홈페이지에서 공식 IDE를 다운받아 실행하면 컴파일러와 에디터, 디버깅툴과 기본적인 라이브러리 등을 한번에 설치할 수 있다.

아두이노 이전의 AVR 프로그래밍은 AVR Studio(혹은 Atmel Studio)와 WinAVR, AVR-GCC 등을 사용하여 컴파일하거나 혹은 IAR, CodeVision 등의 고가의 상용컴파일러를 구입해서 써야만 했다. 현재는 아두이노 IDE를 사용하여 이전보다 편리하고 간편한 방법으로 코드를 작성하고, 컴파일한 후 바로 PC에서 아두이노로 USB 케이블을 이용하여 아두이노에서 실

행가능한 이진파일(정확하게는 hex 파일)을 전달할 수 있다.

아두이노 IDE 는 아두이노 공식 홈페이지로 가서 다운받은 후 설치를 진행 한다. 이 책에서는 윈도우 환경에서 아두이노를 사용하는 것을 기준으로 설명한다. 리눅스나 맥에서도 동일한 방식으로 아두이노 IDE 를 설치하고 사용할 수 있다.

a. 아두이노 홈페이지에서 IDE 다운로드

<https://arduino.cc> 에 접속

상단메뉴 [SOFTWARE] >> [DOWNLOADS] >> 를 차례로 클릭한 후 화면은 조금 아래로 내려서 Download the Arduino IDE 를 찾고 우측에 있는 [Windows installer, for Windows 7 and up]을 클릭한다.

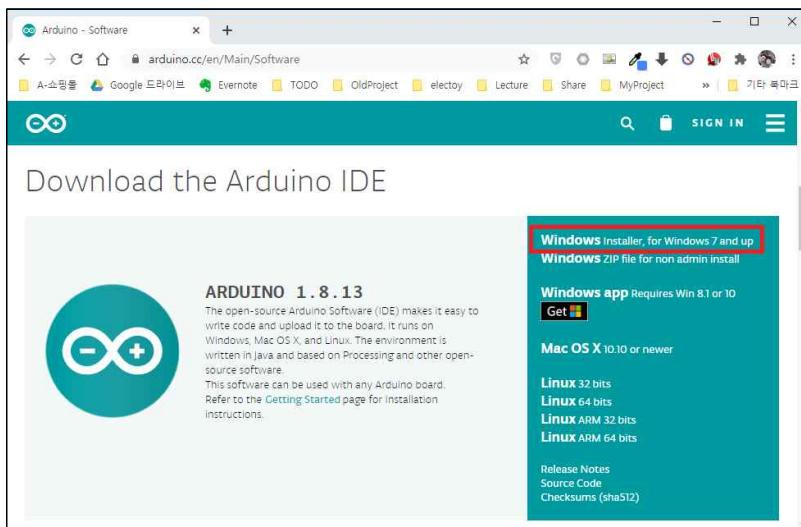


그림 21 IDE 다운로드1

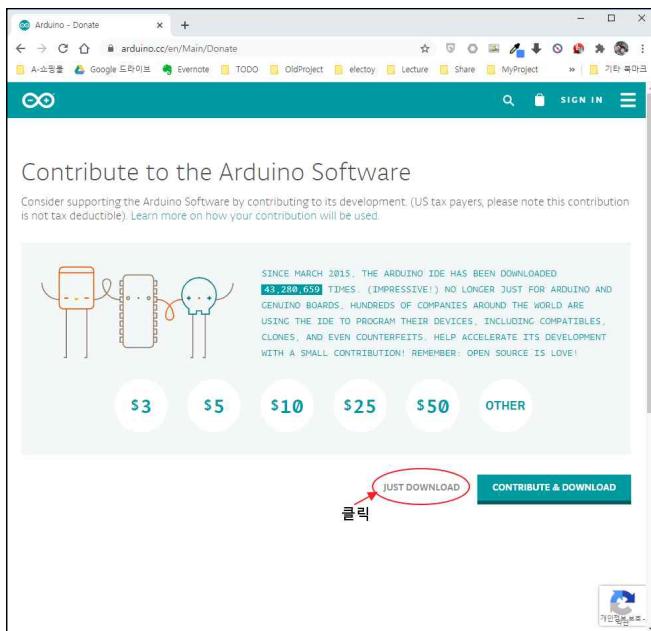


그림 22 IDE 다운로드2

다운이 완료된 다음 마우스 더블 클릭하여 설치를 시작한다.

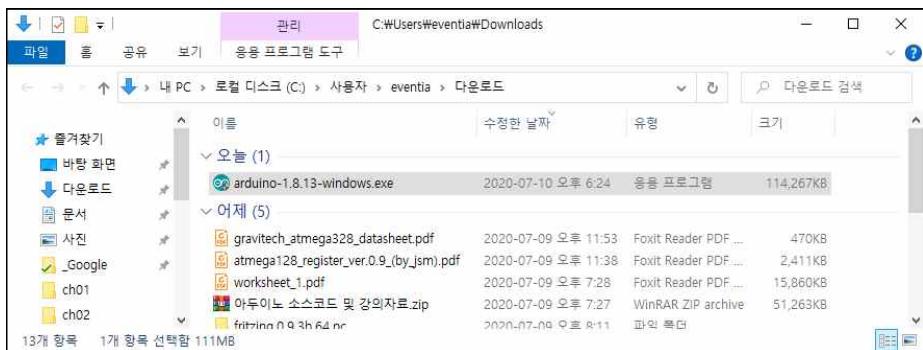


그림 23 설치1

I Agree 버튼을 눌러 설치를 시작한다.

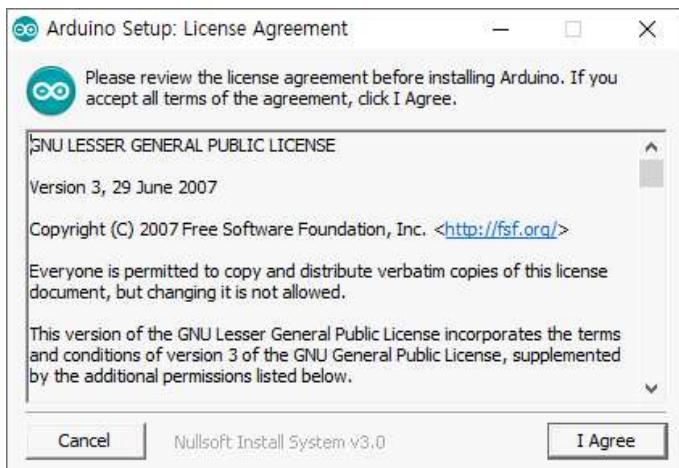


그림 24 설치2

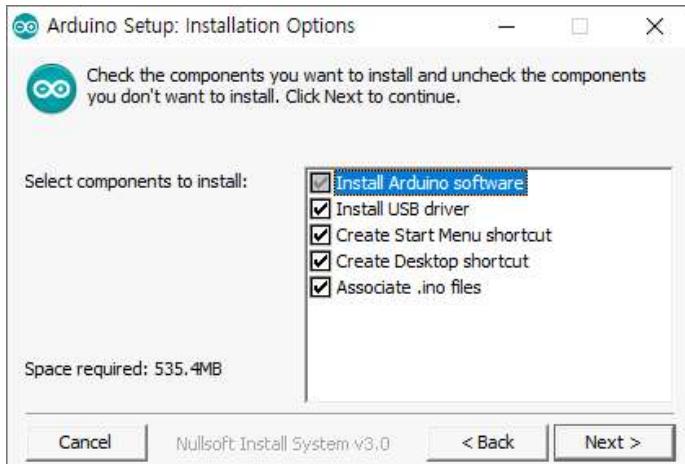


그림 25 설치 - 드라이버

Next 를 눌러 설치를 계속 진행한다.

설치할 폴더를 선택한다. 별도로 선택하지 않으면 Program Files (x86) 아래 Arduino 폴더가 생기면서 그곳에 아두이노 IDE 를 설치한다.

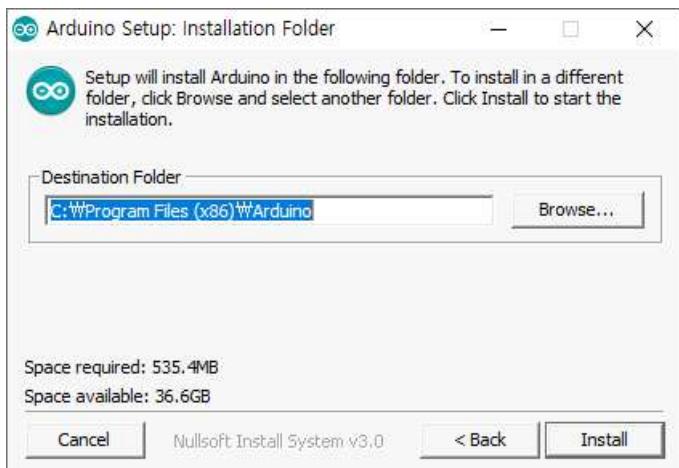


그림 26 설치 – 폴더

설치가 진행되는 과정이 나온다.

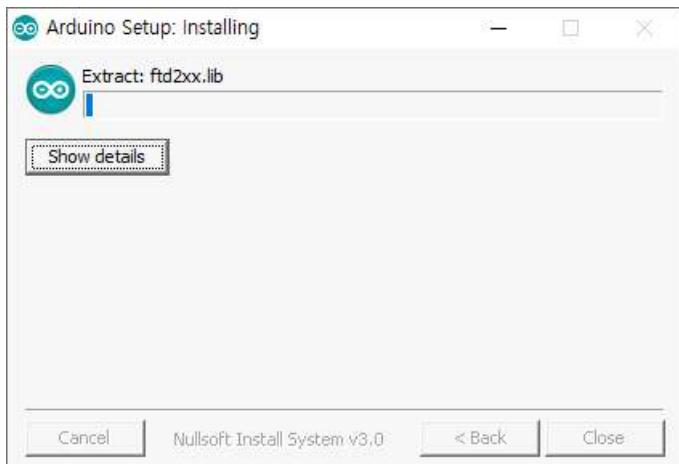


그림 27 설치 – 진행과정

장치 소프트웨어 설치를 위한 대화창이 뜨면 모두 [설치(I)]를 눌러준다. USB를 사용한 통신이 가능하도록 PC에 COM 포트를 만들고, 아두이노와 연결하기 위해 드라이버를 설치하는 과정이다. 만약 여기서 설치하지 않으

면 PC 는 아두이노와의 연결을 허가받지 않은 해킹으로 여겨 접속이 되지 않는다. PC 와 아두이노의 연결이 정상적인 허가를 받았다는 것을 여기서 PC쪽에 알려줌으로서 이후에 PC 와 아두이노가 서로간에 신호를 주고 받을 수 있게 한다.



그림 28 설치 – 드라이버 설치1

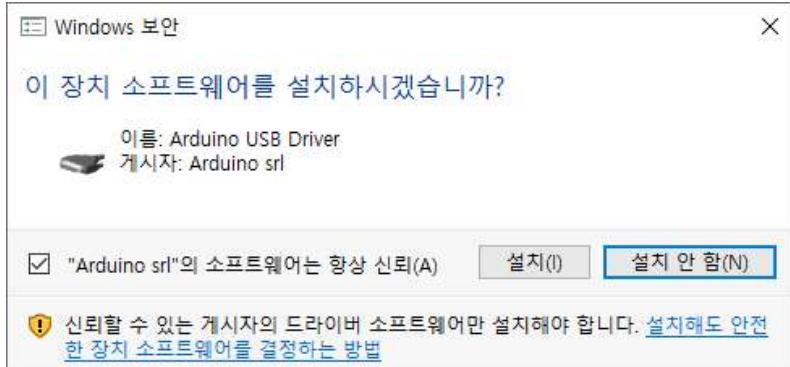


그림 29 설치 – 드라이버 설치2

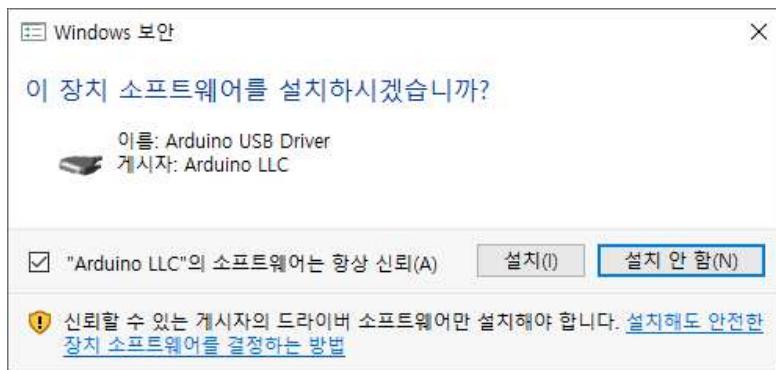


그림 30 설치 – 드라이버 설치3

설치가 마쳐지면 [Close] 를 눌러서 설치과정을 마친다.

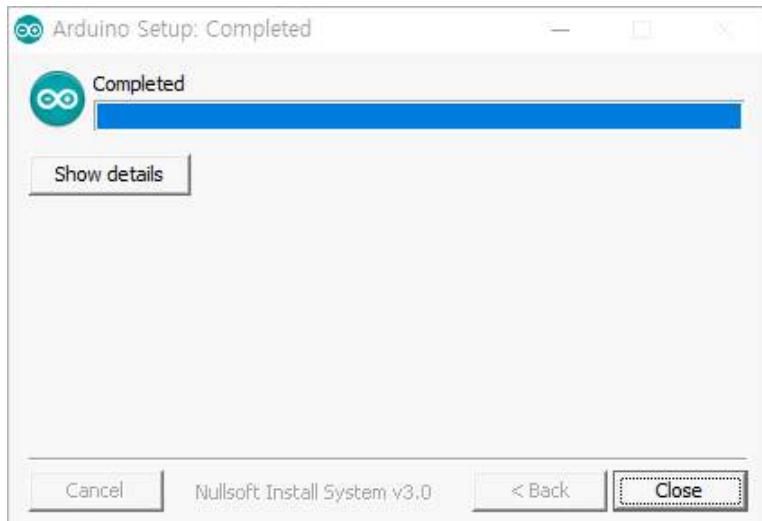


그림 31 설치 – 완료

아두이노IDE를 실행시킨다. 처음 실행시키면 윈도우 방화벽에서 일부기능이 차단된다는 것을 알리고 액세스를 허용할 것인지를 묻는다. [액세스 허용(A)] 을 클릭해 준다.

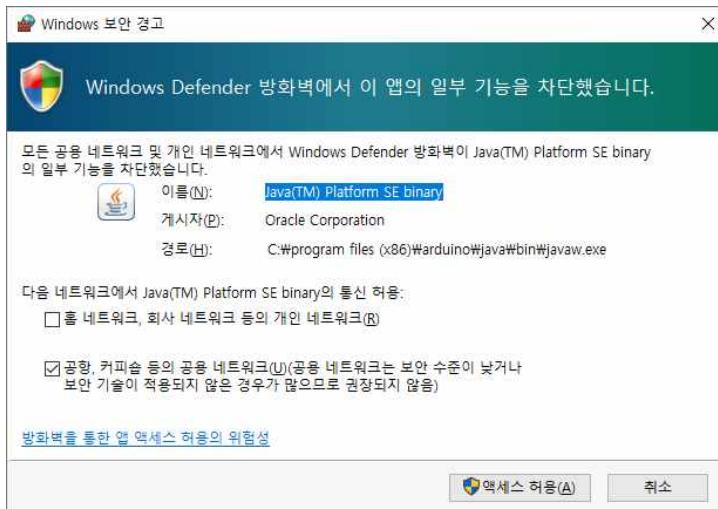


그림 32 설치

아두이노 IDE 가 실행되어 기본 편집기가 화면에 나온다.



그림 33 아두이노 IDE

2.4. CH340 드라이버 설치

기본적으로 아두이노 정품은 ATMEL 의 ATmega16U2 가 장착되어 있다. ATmega16u2 에는 PC의 USB 포트를 통해 아두이노와 연결하고 프로그램을 PC에서 아두이노쪽으로 보내서 플래시메모리를 쓸수 있게 하는 기능이 들어있다. 이 기능을 제대로 쓰기 위해서 별도로 설치할 프로그램은 없다. 그저 아두이노 IDE 만 설치하면 알아서 ATmega16U2 에 관련된 것이 모두 함께 설치된다. 즉, 정품 혹은 정품과 동일하게 제작된 아두이노 호환품을 사용할 경우 아두이노 IDE 외에는 따로 설치할 것이 없다.

하지만 아두이노를 저렴하게 만들기 위해서 ATmega16u2 대신 CH340 이나 CH341 등의 저가형 중국산 IC 를 사용한 것들이 많다. CH340 칩을 사용한 아두이노 호환품을 사용하기 위해서 CH340 드라이버를 별도로 설치해 주어야 한다.

여기서 드라이버라는 말을 처음 들어보는 이들도 있을 것이다. 여기서 말하는 드라이버는 컴퓨터와 연결하는 물리적 장치를 컴퓨터에서 인식할 수 있게 해주는 작은 프로그램이라고 생각하라. 즉, 내가 가지고 있는 PC에 카메라를 연결하고 싶다고 카메라를 그냥 PC의 USB에 물리적으로 꽂기만 하면 되는 것이 아니다. PC와 카메라를 물리적인 USB선을 통해 연결하면서 동시에 PC에서 카메라에 연결할 수 있는 프로그램을 설치해주어야 한다. 이것을 '드라이버'라고 부른다. 드라이버는 PC의 OS가 물리적으로 연결된 외부 장치에 접속해서 사용하기 위한 작은 접속용 프로그램으로 이해하면 된다. 일반적으로 드라이버 프로그램은 하드웨어를 생산하는 제조업체에서 제품과 함께 제공해준다. 프린터를 판매하는 회사에서 프린터 드라이버를 만들어서 제공하고, 웹캠을 판매하는 회사에서 웹캠의 드라이버를 제공해준다. 마찬가지로 16U2 나 CH340 칩을 만드는 회사에서 칩의 드라이버를 제공한다. 아두이노 IDE 에는 16U2 드라이버가 포함되어 있다.

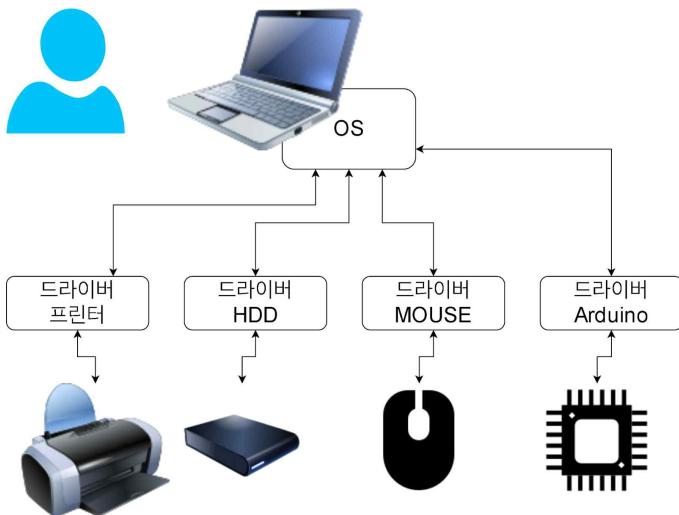


그림 34 외부 장치 구동 드라이버

아두이노 우노, 메가 2560 등에 사용된 16U2 칩의 드라이버는 아두이노 IDE를 설치 프로그램안에 포함되어 있으므로 별도 설치가 필요하지 않다. 하지만 중국산 저가 드라이버인 CH340 을 사용하는 아두이노 호환보드의 경우 아두이노 IDE 를 설치할 때 함께 설치되는 16U2 드라이버로는 사용 할 수 없으므로 별도로 설치해 주어야 한다.

CH340 드라이버는 인터넷 검색으로 찾아 다운로드 후 설치한다. 일반적으로 구글에서 "아두이노 ch340 드라이버"라고 검색하면 관련 자료를 쉽게 찾을 수 있다. 혹은 필자의 블로그에 올려두었으므로 그곳에서 다운받아도 된다. [블로그 링크 <https://electoy.tistory.com/328>]

인터넷에서 검색하여 다운을 받거나 혹은 필자의 블로그에서 다운을 받은 후 압축을 풀면 다음과 같은 파일들이 나온다. 여기서 SETUP.EXE 를 실행한다.

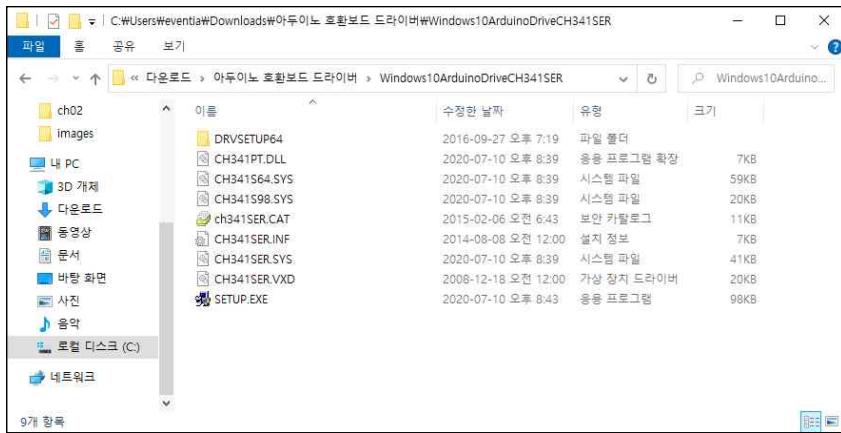


그림 35 드라이버 설치 파일

DriverSetup(X64) 화면에서 INSTALL 을 클릭하면 CH34x 드라이버가 설치된다. 설치가 끝나면 드라이버 설치가 성공했다는 메시지창이 뜬다.

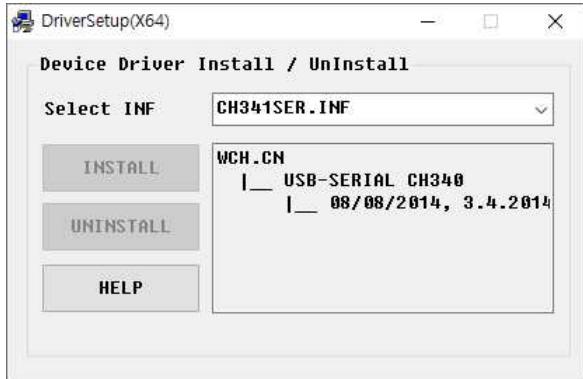


그림 36 CH340 드라이버 설치 화면

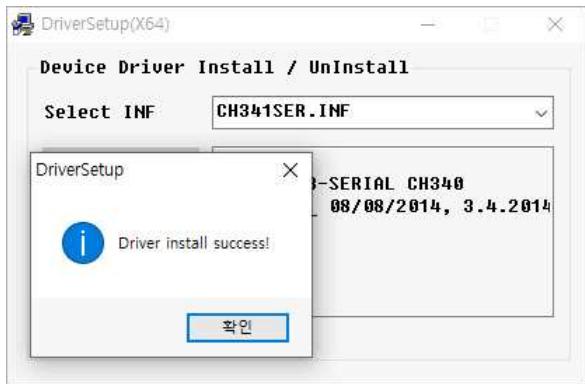


그림 37 설치 성공

확인버튼을 누르고, 다시 오른쪽위의 X 표시를 눌러 창을 완전히 닫아준다. 드라이버 설치는 별도의 프로그램이 작동하지 않는다. 설치가 끝나면 창을 닫아주면 완료된다.

드라이버가 제대로 설치되었는지 확인해 보겠다. 우선 CH340 을 사용한 아두이노를 PC와 USB 선을 이용해 연결해 준다. 윈도우 검색창에 '장치 관리자'를 입력하여 검색한다. 윈도우 검색창은 왼쪽 하단에 있는 검색창이다. 만약 검색창이 없다면 윈도우키를 클릭한 다음 '장치 관리자'를 키보드에 치면 된다.

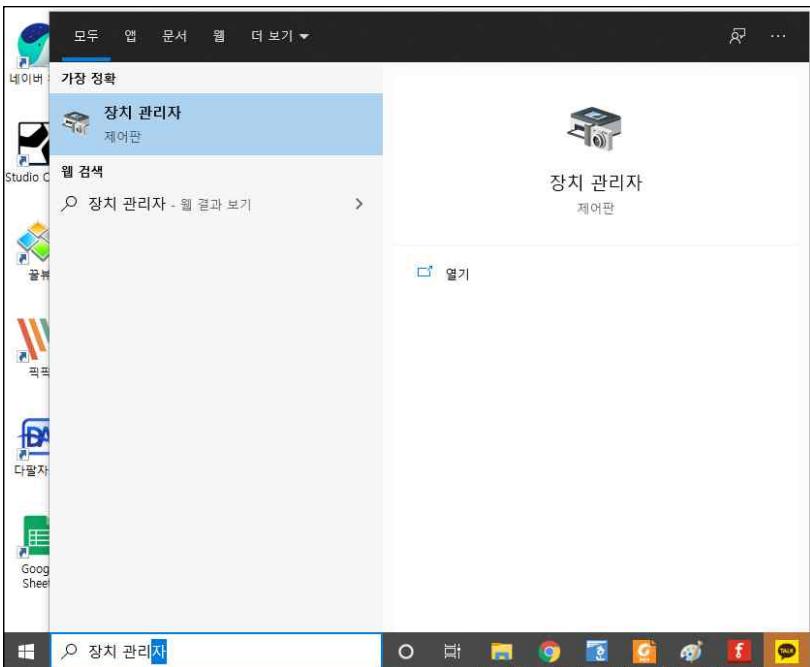


그림 38 장치관리자1

장치관리자에서 포트(COM & LPT) 항목 왼쪽에 있는 작은 꺽쇠를 클릭하면 새롭게 접속한 CH340 드라이버를 사용한 아두이노와 어떤 COM 포트를 이용해서 접속할 수 있는지 보여준다.

CH340 을 사용하는 아두이노는 CH340 으로 표시된다. 16U2 를 사용하는 아두이노의 경우 이 포트 이름이 Arduino Uno 라는 이름으로 표시된다. 아래는 CH340 을 사용하는 아두이노와 16U2 를 사용한 아두이노를 모두 PC에 연결한 상태다. 16U2 를 사용한 아두이노는 COM3 에 연결되었고, CH340 을 사용한 아두이노는 COM5 에 연결되었다.

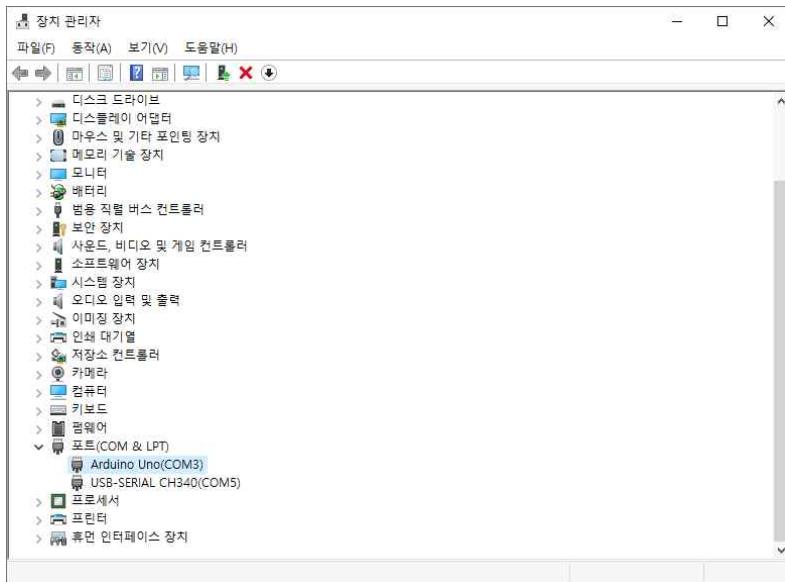


그림 39 장치관리자2

아두이노와 PC가 연결된 USB 연결을 끊으면 포트는 보이지 않게 된다. 16U2를 사용한 아두이노와의 연결을 끊어보았다. 다음과 같이 하나의 포트만 보이게 된다.

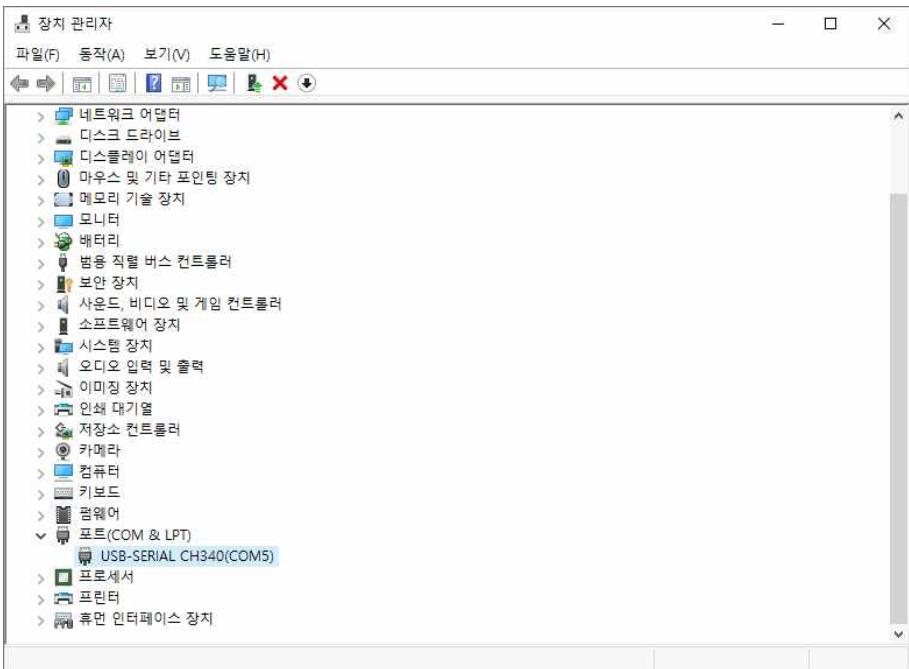


그림 40 장치관리자3

2.5. 아두이노 시뮬레이터

아두이노를 시뮬레이션 할 수 있는 프로그램들이 있다. 아두이노 시뮬레이터가 있으면 아두이노 없이 간단한 프로그램을 작성해서 오류가 있는지 여부를 없는지 확인할 수 있다. 아두이노 시뮬레이터는 온라인으로 가능한 것부터 실제 개인용 PC에 설치해서 사용할 수 있는 것까지 다양한 종류가 나와있으며 여기서는 그중 사용하기 편한 온라인 시뮬레이터 두 가지를 소개하고자 한다.

첫번째는 가볍게 사용할 수 있는 오토데스크사에서 공개한 Circuits 가 있다.

두번째는 <https://wokwi.com>에서 제공하는 Wowki가 있다.

이 외에도 구글에서 "arduino simulator"로 검색하면 다양한 시뮬레이터가 나온다. 때로 어떤 서비스는 금방 사라지기도 한다. 충분한 시간을 두고 서비스가 가능한 곳으로 여기서 [tinker.com](https://tinkercad.com)으로 접속해 사용할 수 있는 Circuits 사용법을 간단히 소개한다.

<https://tinkercad.com>으로 접속하고, 로그인한 후 왼편 메뉴에서 Circuits를 선택한다. 그 다음 [새 회로 작성]을 클릭하면 전자회로를 만들 수 있는 화면이 나온다.

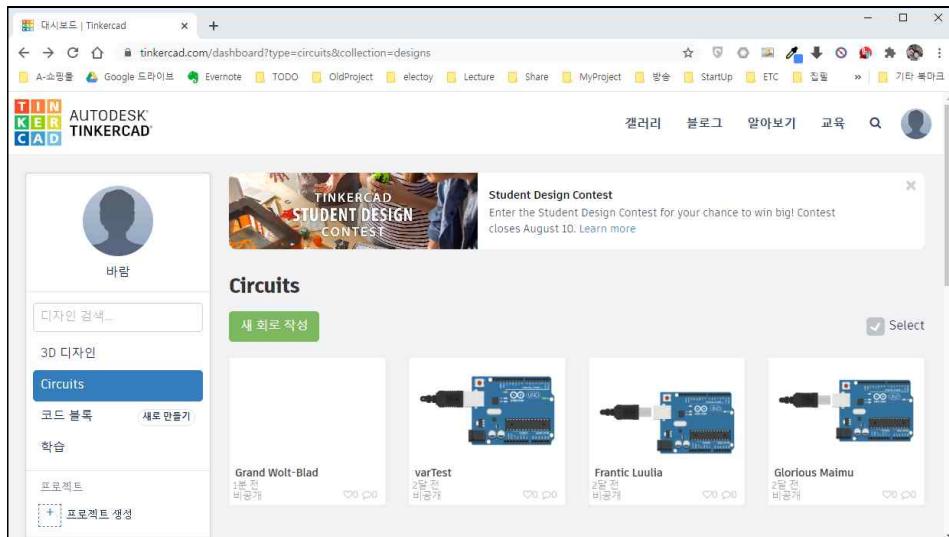


그림 41 tinkerCAD 접속화면

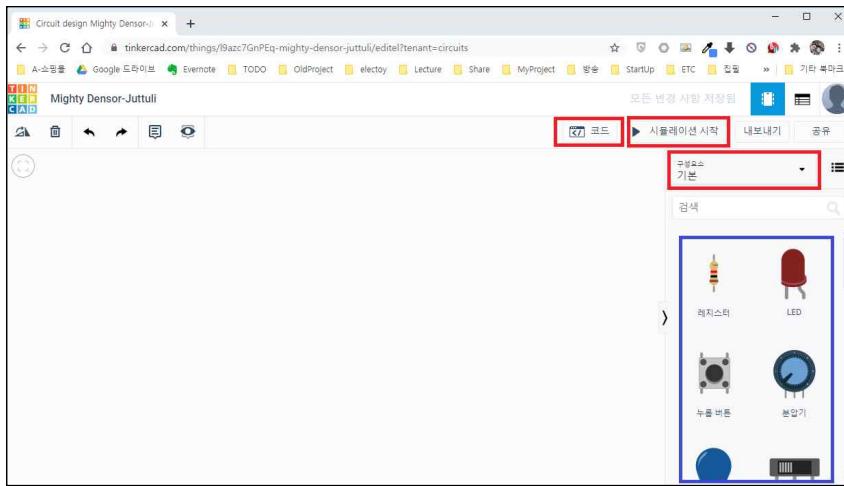


그림 42 tinkerCAD 화면

이 화면에서 오른쪽 파란색 박스에 있는 전자 부품들을 가운데로 가져와서 전자회로를 만들 수 있다. 이때 [구성요소] 를 클릭하면 기본, 모두, Arduino, 회로조립품 등을 선택할 수 있다. Arduino 를 선택하면 아두이노 보드가 나온다. '깜빡임'이라고 이름 붙여진 그림을 클릭하면 아두이노 우노 와 LED 1개가 저항과 연결된 상태로 화면에 나온다.

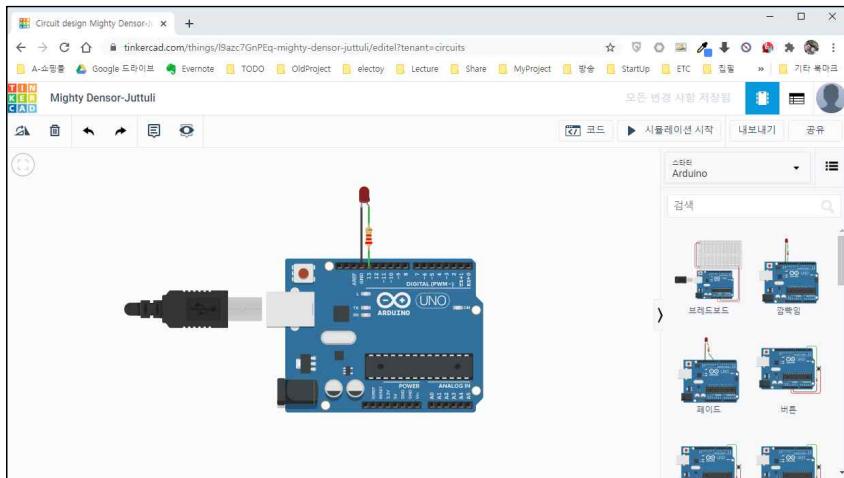


그림 43 tinkerCAD 프로젝트 화면

여기서 간단한 프로그램을 넣어보도록 한다. [코드] 버튼을 클릭하면 아두이노 프로그래밍 환경이 나온다.

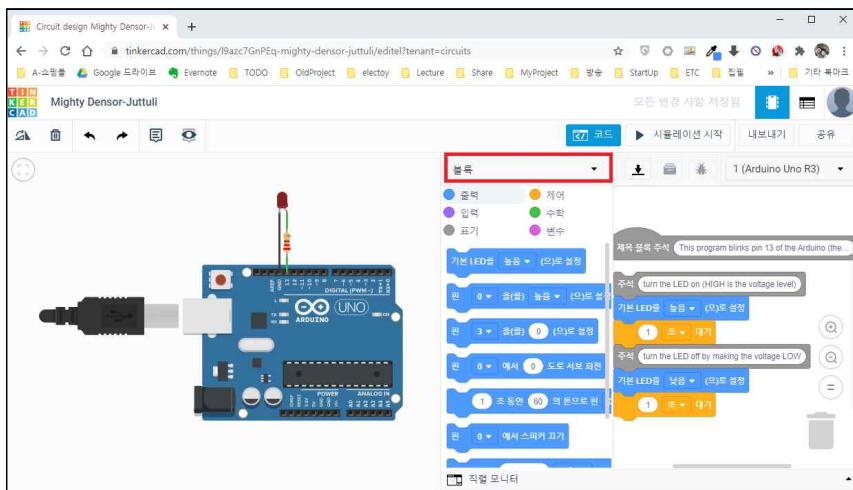


그림 44 tinkerCAD 코드편집

스크래치 형태의 블록형 코딩을 사용할 수도 있지만 우리는 텍스트코딩을 사용하도록 한다. '블록'이라고 된 부분을 클릭해서 '문자'로 바꾼다. 화면에 모든 블록이 손실된다고 경고표시가 나지만 [계속]을 눌러준다.

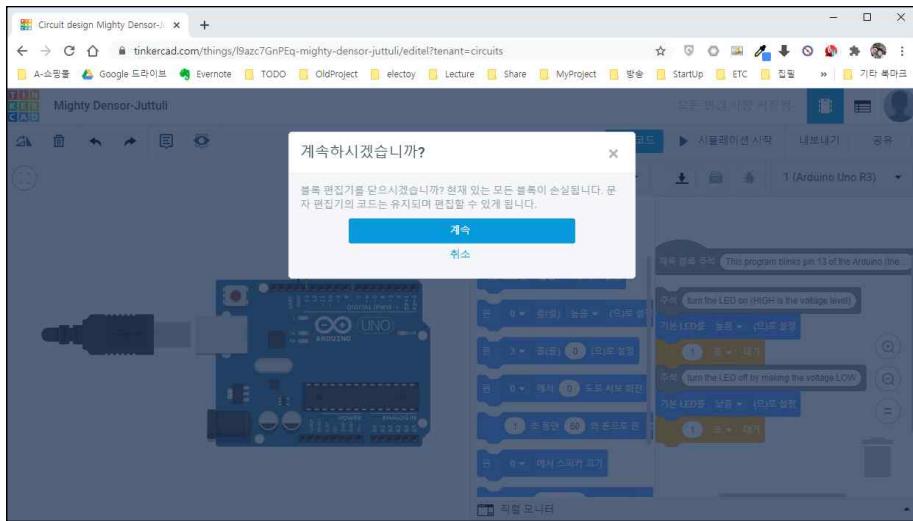


그림 45 tinkerCAD 텍스트코드 전환

블록대신 텍스트로 된 C 소스가 나왔다. 이제 여기서 아두이노를 C로 프로그래밍하고 그 결과를 화면에서 바로 확인할 수 있다. 화면에 있는 소스는 아두이노를 시작할 때 처음 다루게 되는 Blink라는 프로그램이다. LED가 1초 간격으로 깜박이는 프로그램이다.

소스는 1초 간격이지만 0.2 초 간격으로 바꿔보도록 하자.

delay(1000); 이라고 된 부분을 모두 delay(200); 으로 바꿔본다. 1000 을 200 으로 고치기만 한다.

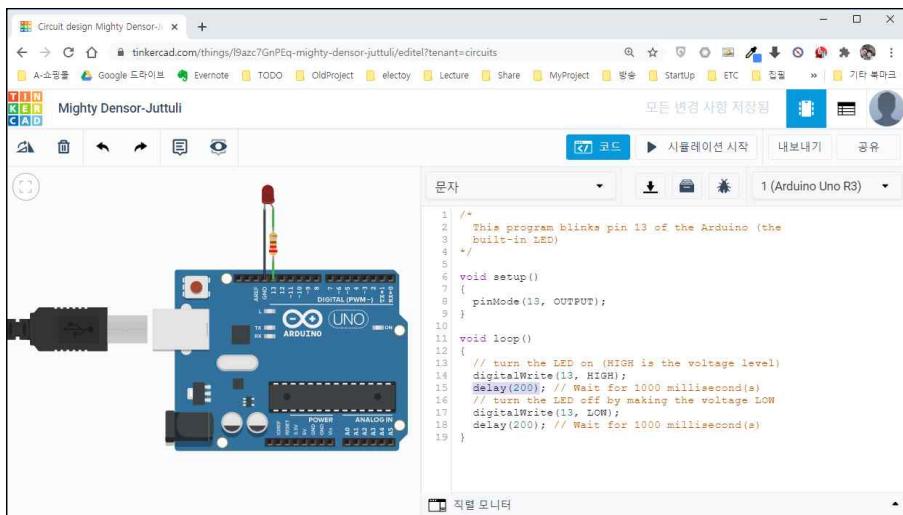


그림 46 코드 설정

이제 제대로 작동하는지 확인하고 싶다면 [시뮬레이션 시작] 버튼을 클릭해 준다.

LED에 짧은 간격으로 불이 켜지고 꺼지는 것을 확인할 수 있을 것이다.

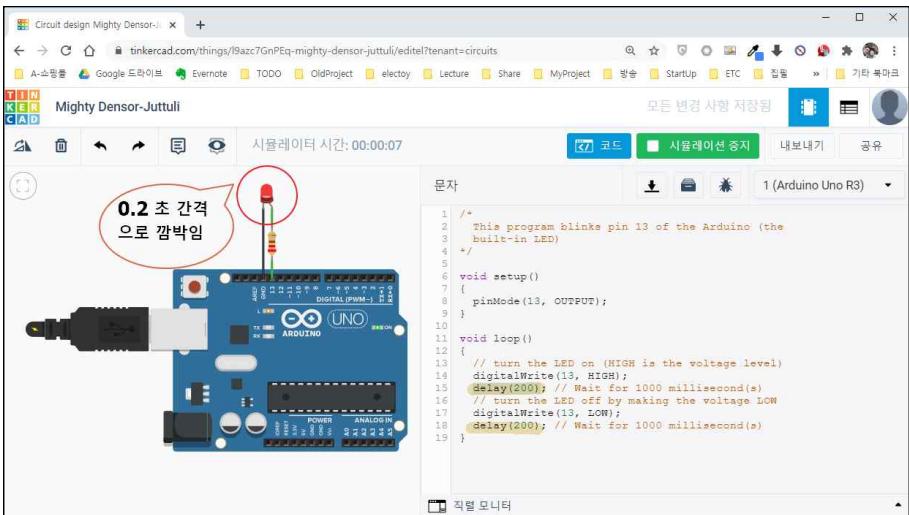


그림 47 시뮬레이션 작동 화면

간단하지만 디버거와 직렬 모니터도 사용해볼만 하다. 급하게 아두이노 프로그램을 작성해보고 싶지만 가지고 있는 아두이노가 바로 옆에 없을 때, tinkercad.com에서 Circuits 서비스를 이용할 수 있다.

2.6. 온라인 C 컴파일러

C는 아두이노와 같은 MCU에서 주로 사용된다. 하지만 일반 PC 환경에서도 사용된다. 일반 PC에 설치할 수 있는 C 컴파일러들이 있다. Dev C++이나 CodeBlocks 같은 컴파일러들이 많이 사용된다. 우선 설치하기 쉽고, 사용도 간편한 편이다. 조금 더 전문적인 C 컴파일러로는 Visual Studio가 있다. 여기서는 PC에 설치하지 않고 사용할 수 있는 온라인 C 컴파일러를 소개한다.

<https://glot.io>
https://www.tutorialspoint.com/compile_c_online.php
<https://repl.it/languages/c>
https://www.onlinegdb.com/online_c_compiler

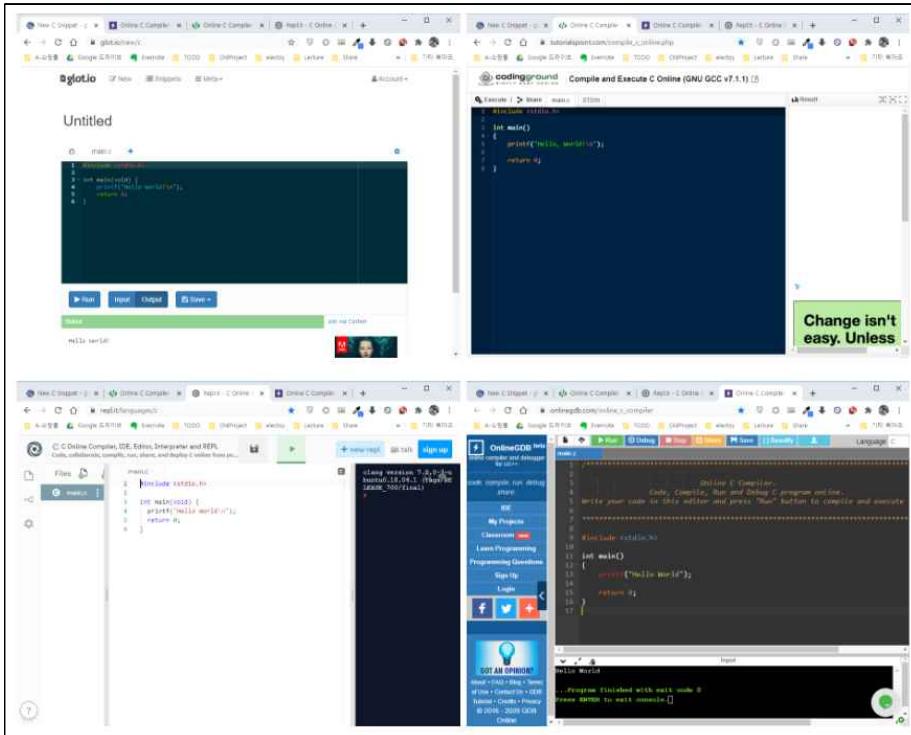


그림 48 다양한 온라인 C컴파일러

온라인 컴파일러는 이 외에도 많이 있다. glot.io 나 repl.it 를 사용하면 C 언어만이 아닌 다른 언어들을 온라인상에서 공부할 수 있다. PC 에 별도의 C 컴파일러를 설치하지 않고 온라인 C 컴파일러를 이용해서 이후에 나오는 임베디드용 아두이노 C 와 PC 기반의 C 에 대해서 비교하며 살펴보도록 하겠다.

Chapter 03

주요 학습 목표

- 프로그래밍에 대한 이해
- C언어 기본 이해

Chapter 03. C와의 만남

C 는 상당히 오랜 역사를 가지고 있는 언어다. C 가 처음 등장할 때만 해도 C 는 대체불가한 최고의 프로그래밍 언어였다. 하지만 역사는 흐르고, 새로운 프로그래밍 언어들이 계속 나오고 있다. 지금 C 의 위치는 예전 같지는 않다. PC 쪽에서는 C#, 자바등의 언어들이 주를 이루고 있고, 웹 환경에서는 자바스크립트 계열의 언어들이 대세다. 파이썬이나 루비같은 처음 프로그래밍을 접하는 이들도 쉽게 시작할 수 있는 언어들이 계속 나오고 있다. 그럼 이 시점에서 C언어를 배운다는 것은 어떤 의미가 있는가?

PC 에서 단순한 계산을 하는 경우라면 굳이 C 를 사용할 필요가 없다. 더 쉽고 가벼운 언어들이 많이 나오고 있다. PC 에서의 언어로 C 는 예전의 자리를 잃어가고 있다. 하지만 IoT 가 대세가 되어버린 지금 수많은 MCU 들이 여기 저기에 사용되고 있다. 현관을 열면 만나는 감지센서를 사용하는 현관등, 인터폰을 들어가는 MCU, 전기밥솥과 전기포트, 그 옆에 있는 전자렌지와 믹서기, 냉장고와 TV, 인터넷을 사용하기 위해 가져다 둔 공유기와 허브, 게이트웨이, 로봇청소기와 에어컨 등 우리 주변에는 너무나 많은 MCU 들이 사용되고 있다. 이런 곳에 사용되는 MCU 들은 필요한 일을 하기 위한 프로그래밍이 되어 있고, 그 프로그래밍에 사용되는 언어는 단연 C 가 주가 된다. C 는 저차원부터 고차원까지 감당할 수 있는 프로그래밍 언어다. 현대에 새롭게 만들어지는 언어들은 저차원에 대한 접근은 하지 않는다. 왜냐하면 PC 는 충분히 많은 램과 하드디스크를 가지고 있으며, PC 에 있는 CPU 는 예전에 비해 아주 빨라졌기 때문이다. 하지만 전기밥솥에

사용되는 MCU 는 여전히 작고, 적은 메모리와 느린 속도를 가지고 있다. 그것만으로도 충분하기 때문이다.

3.1 프로그래밍이란 무엇인가?

위키피디아에 따르면 프로그래밍이란 특정 연산 결과에 도달하기 위해 실행 가능한 컴퓨터 프로그램을 설계하고 구축하는 프로세스로 정의된다. 프로그래밍에는 분석, 알고리즘 생성, 언어로 알고리즘 구현등의 작업이 포함된다. 이때 알고리즘을 구현하는 소스코드는 CPU에서 직접 실행하는 기계어가 아닌 프로그래머가 이해할 수 있는 언어로 작성된다.

마이크로컨트롤러는 기본적으로 초소형 컴퓨터이고, 이러한 프로그래밍에 대한 정의는 아두이노를 포함한 MCU 프로그래밍에도 적용된다.

키보드를 쳐서 소스코드를 만들기 전, 프로그램을 설계하는 것이 우선이다. MCU에서 수행하기 원하는 모든 과정을 글이나 그림등으로 그리는 과정이 선행되어야 한다. 이것을 의사코드(pseudo code)라고 부른다.

의사코드는 실제코드와 유사할 필요는 없다. 다만, 사람이 읽을 때 명확하게 전달되어야 하고, 코드를 통해 최종적으로 만들어 내고자 하는 결과를 구체화시킬수 있어야 한다. 예를 들면 다음과 같이 의사코드를 만들수 있다.

- 1) 현재의 온도를 측정한다.
- 2) 온도가 43도 이상이면, 부저로 소리를 낸다.
- 3) 그렇지 않다면 (온도가 43도 미만이라면), 초록색 램프를 켠다.
- 4) 맨 앞으로 돌아가서 모든 단계를 반복한다.

이렇게 의사코드를 만든 다음 이것을 구체적인 프로그래밍 언어를 사용해 코드로 변환시킨다. 그러면 위의 의사코드를 사용해 실제 소스코드를 만드는 것은 어렵지 않게 할 수 있다. 의사코드는 주관적이고 표준화되어 있지 않다. 어떤 의사코드이든지 처리절차를 잘 설명할 수 있으면 된다.

하지만 이 글을 여기까지 읽은 분 이들 중 일부는 여기서 한가지 의문점이 생길것이다. 과연 제대로 작동하면 될까? 하는 의문이 들수 있다. 저 의사코드를 작성할때 목욕물을 생각하며 의사코드를 썼다. 43도가 넘어가면 소리를 내고, (위에는 적혀있지 않지만) 온수 가열을 중단한다. 43도 미만이라면 온수가열을 하면서 초록색 램프로 켜서 뜨겁지 않음을 보인다.

그런데 이런 경우가 발생할 수 있다. 지금 온도는 42도다. 그러면 초록색 램프가 켜진다. 그러다 43도를 넘어서면 부저로 소리를 낸다. 그런데 초록색 램프를 끄라는 지시가 없다. 램프는 계속 초록색으로 남아있게 된다. 그리고 부저의 소리도 뭔가 문제가 있어보인다. 언제 소리를 꺼야할까? 그 부분이 나와있지 않다. 즉, 이 시점에서 이 프로그램은 무언가 문제가 있다는 것이 발견된다. 의사코드는 실제 소스코드에서 발견하기 어려운 논리적 결함을 보다 쉽게 볼수 있게 해준다.

문법적 결함은 컴파일러가 잘 찾아주기 때문에 오히려 쉽게 고칠수 있다. 하지만 위와 같은 논리적 결함은 컴파일러가 찾을 수 없다. 문제를 정의하고, 프로그램을 설계한 사람만이 논리적 결함을 찾을 수 있다. 이러한 과정을 디버깅이라고 부른다. 디버깅을 통해 문법적 결함만이 아니라 논리적 결함을 찾아야 한다.

위 내용을 고치면 다음과 같다.

- 1) 현재의 온도를 측정한다.
- 2) 온도가 43도 이상이면, 부저로 소리를 내고 초록색 램프를 끈

다.

- 3) 그렇지 않다면 (온도가 43도 미만이라면), 부저를 끄고, 초록색 램프를 켠다.
- 4) 맨 앞으로 돌아가서 모든 단계를 반복한다.

3.2. 프로그래밍 패러다임

사회학에서 패러다임은 토마스 쿤의 제안에 따르면 한 시대 사람들의 의견이나 사고를 근본적으로 규정하고 있는 테두리로서의 인식체계, 사물에 대한 이해의 관점을 의미한다. 그리스어 παράδειγμα (파라데이그마)에서 파생된 패러다임은 안경으로 비유되기도 한다. 패러다임은 세상을 보는 시각이다. 이런면에서 세상을 보는 관점이 시대에 따라 변하는데 그 관점을 유지하는 것과 반발하는 것 사이에서 패러다임 역시 변할 수 있다는 것이다.

프로그래밍을 하는 사람들에게도 이러한 패러다임이 있다. 가장 대표적인 프로그래밍 패러다임은 절차지향과 객체지향이다. 과거 C 언어와 함께 구조적 프로그래밍이 주류의 패러다임이었다면 지금은 객체지향으로 패러다임의 변화가 일어난 상태다.

A. 절차지향

절차지향 프로그래밍은 순서대로 명령을 받고, 문제를 해결하는 식이다. 초기의 프로그래밍언어들은 성능이 낮은 하드웨어에서 최대한의 속도를 끌어내야 했기 때문에 절차지향 프로그래밍은 좋은 패러다임이었다.C 는 프로그래머에게 거의 모든 권한을 주는 언어다. 기존의 다른 언어들과는 비교할 수 없을 정도의 빠른 속도를 내는 언어이며, 지금도 여전히 임베디드, 사물인터넷 등의 저사양의 프로세서를 사용하는 곳에서 강점을 보인다. C 는 인라인 어셈블리를 사용할 수 있지만, 최근의 C 컴파일러는 사용자가 어셈블러로 만든것과 거의 동일한 정도의 최적화가 이루어지고 있다.

B. 객체지향

객체지향프로그래밍은 프로그램을 작게 나누고 각각의 기능을 모아 문제를 해결하는 방식이다. 기능이 다양해지면 코드는 복잡해진다. 절차지향방법만으로 해결하기 어려운 많은 요구를 작은 부분으로 나누어 각각 기능을 부여하고 각 객체가 알아서 자신의 기능을 하도록 만드는 방식이 객체지향이다. 절차지향은 위에서 아래로 순서대로 명령을 수행해나가는 구조가 중요했다면, 객체지향은 개별 객체를 하나의 유기체처럼 다루어 어떤 입력에 대해 어떤 반응을 보일지를 개별적으로 프로그램하는 방식이다. 한명의 천재가 완벽한 프로그램을 만드는 것 보다 여러명의 범재가 단순한 기능을 만들고, 그 기능으로 문제를 해결하는 방식이다. 프로젝트가 커지고, 기능이 많아지면서 객체지향프로그래밍은 지금은 필수적인 패러다임이 되어가고 있다.

이 외에도 함수형 프로그래밍, 반응형 프로그래밍등 다양한 프로그래밍 패러다임이 있다.

3.3. 프로그래밍 스타일

프로그래밍 스타일, 혹은 코딩스타일은 소스 코드를 작성할 때의 사회적 약속과 같다. 특히 객체지향프로그래밍 설계를 할때 여러 사람이 협업을 하게 되므로 기존의 코드나 타인이 작성할 코드를 이해하기 어려워서 일정한 패턴을 만들고 그것을 사용하기로 약속하기도 한다. 디자인패턴이란 이름으로 많은 자료들이 나와있으므로 객체지향프로그래밍을 하는 사람들은 디자인패턴을 검색해 보기 바란다.

C는 객체지향 프로그램이 아니므로 디자인패턴을 적용할 필요는 없다. 대

신 아래 세가지 규칙을 나름대로 정하고 함께 협업을 하는 이들과 공유하는 것이 좋다.

A. 변수, 함수, 상수의 이름

변수와 함수의 이름을 만들때 이름 정의하는 규칙이 필요하다. C의 문법으로는 예약어를 제외하고, 영문으로 시작하고 영문자, 숫자, 특자문자로는 '_' 가 포함되는 범위에서 이름을 정하면 된다. 하지만 다른 이들과 함께 프로젝트를 진행한다면 이것만으로는 부족하다. 함수의 이름, 변수의 이름, 상수의 이름이 모두 같은 식으로 만들어지면 곤란한 상황에 처할 수 있다.

이름 명명법으로 다음 세가지가 주로 사용된다.

a. 카멜 표기법(Camel Case, camelCase)

여러 단어를 이어서 이름을 만들때 각 단어의 첫자는 대문자로 하되 첫문자는 소문자로 한다. 이렇게 만들어진 이름의 형상이 중간중간 대문자때문에 낙타 혹이 튀어나온 듯 보여서 카멜표기법(Camel Case)라고 부른다. 예를 들면 myBankAccount, manAges 처럼 사용한다.

b. 파스칼 표기법(Pascal Case, PascalCase)

카멜표기법과 유사하다. 맨 앞에 오는 문자도 대문자로 표기한다. 카멜표기법과 파스칼표기법을 혼합해서 사용하는 경우가 많다. 변수 이름으로는 카멜표기법을 쓰고, 클래스 이름으로 파스칼표기법을 쓰는 경우가 많다. 예를 들면 CarList, KidAge 처럼 사용한다.

c. 스네이크 표기법(Snake Case, snake_case)

단어 사이에 언더바(_)를 넣어서 분리한다. 단어는 전부 소문자로 하거나 대문자로 할수 있고, 첫자만 대문자를 쓸수도 있다. 보통 전체를 대문자로 하는 경우는 상수이름을 정할때 사용한다. MAX_AGE, my_height, Survivor_Titanic 과 같이 사용할 수 있다.

이 외에도 헝가리안 표기법이라는 것이 있다. strName 처럼 이름의 앞에 자료형을 붙이는 방식인데, 초기에는 많이 사용되었지만 지금은 거의 사용되지 않는다. 프로젝트 도중에 자료형이 바뀌면 모든 변수명이 수정해야 하는 상황이 발생할 수 있다.

필자는 변수와 함수는 카멜표기법으로 클래스(C는 클래스가 없다)는 파스칼표기법으로, 상수는 대문자를 사용하는 스네이크 표기법을 주로 사용한다. 물론 이 모든 것은 함께 작업하는 이들과 함께 논의해서 약속을 만들어 사용하는 것이 우선이다.

B. 중괄호의 위치

C 언어는 중괄호를 사용해서 블록을 구분한다. 괄호의 위치에 따라 크게 세가지 방법이 존재한다. 각 방법은 장단점이 있기 때문에 어떤 것이 가장 좋은 것이라고 주장하기는 어렵다. 단, 둘 이상을 혼용하여 사용하는 것은 좋지 않다. 하나의 방법을 선택했으면 그것을 계속 사용하는 것이 좋다.

a. GNU 괄호 포맷

```
if (i==tmp)
```

```
{  
    process();  
}
```

블록을 if 문 아래 작성하고, 블록에 if 에 속했음을 보이기 위해 들여쓰기 처리를 한다. 장점은 구조가 한눈에 잘 보인다는 것이고, 단점은 들여쓰기를 해서 수평으로 코드 작성할 공간이 적어진다는 점이다.

b. K&R 팔호 포맷

```
if (i==tmp){  
    process();  
}
```

초창기 C언어에서 주로 사용되던 스타일이다. 여는 블록을 if 구문과 같은 행에 배치한다. 코드 줄 수를 절약할 수 있어서 한눈에 여러줄을 코드를 볼 수 있고 한줄에 최대한 많은 코드를 넣을 수 있다는 장점이 있다. 단, 여는 팔호와 닫는 팔호의 위치가 달라서 편집이나 수정시 가독성이 떨어진다.

K&R 은 1978년 브라이언 커니핸(Kernighan)과 데니스 리치(Ritchie)의 이름 첫자를 따서 불리는 이름으로 그들이 만든 “The C Programming Language” 책은 비공식적으로 초창기 C의 표준으로서의 역할을 했다. 그 책에서 서술되는 C를 "K&R C"라고 이름 붙였고, 이후 C 표준을 이야기할 때 자주 등장한다.

c. BSD 팔호 포맷

```
if (i==tmp)  
{
```

```
    process();
}
```

GNU 스타일의 장점인 블럭의 시작과 끝부분의 가독성과 K&R의 장점인 수평 코딩 작성시의 효율을 가져와 결합한 스타일이다.

이 책에서는 괄호포맷으로 BSD 스타일을 주로 사용하고 있으며, 필요시에 K&R 스타일을 사용한다.

K&R 과 BSD 로 작성된 소스를 보고 비교해 보자.

```
// K&R Format
if (price > 50) {
    for (i=0; i<100; i++) {
        if (tax < 5) {
            process1();
        } else if (tax < 10) {
            process2();
        }
    }
}
```

```
// BSD Format
if (price > 50)
{
    for (i=0; i<100; i++)
    {
        if (tax < 5)
```

```
{  
    process1();  
}  
else if (tax < 10)  
{  
    process2();  
}  
}  
}
```

BSD 포맷이 아래로 길어지는 것을 확인할 수 있다. 가능한 적은 지면에 코드를 출력하기 원하는 출판사의 입장에서는 BSD 보다는 K&R을 선호할 것이다. 초창기 C 언어를 쓰는 이들이 K&R 포맷을 사용했기에 많은 이들이 K&R에 익숙해졌다. 하지만 코드를 길게 쓰더라도 가독성은 BSD나 GNU가 나아 보인다. K&R는 중괄호를 열고 닫는 부분이 동일 열상에 있지 않기 때문에 바로 확인하기가 어렵다.

C. 블록과 들여쓰기

들여쓰기를 탭(Tab)으로 할 것인지 스페이스로 할 것인지도 여러 사람이 함께 협업할 때는 약속으로 정해 두어야 할 사안이다. 탭을 쓰면 편하지만 다른 협업자가 작성할 때 사용한 것과 다른 에디터와 설정을 가지고 있다면 탭은 다르게 보일 수 있다. 실제로 현장에서는 탭보다는 스페이스를 사용하는 것이 일반적이다. 다만, 탭과 스페이스를 함께 쓰는 것은 피해야한다. 특정 언어는 탭과 스페이스가 함께 있을 때 매우 곤란한 에러를 발생하곤 한다.

스페이스를 사용해서 들여쓰기를 한다면 남은 문제는 2칸인가 4칸인가다. 이 부분은 전적으로 프로그래머들에게 달려있다. 4칸을 써서 가독성을 높일 것인지, 2칸을 써서 가로줄 코드를 한자라도 더 넣는 방향으로 갈 것인

지는 함께 논의해서 약속을 하고 그 약속을 지키면 된다. 요즘 사용되는 대부분의 에디터는 스페이스를 몇칸으로 할 것인지 설정할 수 있다.

HTML 은 가로줄에 넣어야 할 코드가 많기 때문에 2칸으로 사용하는 경우가 많다. C 는 4칸과 2칸이 비슷하다.

```
// BSD Format, indent 2
if (price > 50)
{
    for (i=0; i<100; i++)
    {
        if (tax < 5)
        {
            process1();
        }
        else if (tax < 10)
        {
            process2();
        }
    }
}

// BSD Format, indent 4
if (price > 50)
{
    for (i=0; i<100; i++)
    {
        if (tax < 5)
        {
            process1();
        }
    }
}
```

```
else if (tax < 10)
{
    process2();
}
}
```

들여쓰기를 2로 했을 때와 4로 했을 때의 가독성 차이를 보라. 왼쪽의 코드를 보고 크게 불편하지 않다면 2칸으로 들여쓰기를 사용하라. 하지만 왼쪽의 코드의 중괄호의 위치가 어긋났는지 거슬린다면 4칸 들여쓰기를 사용하는 것이 좋다.

3.4. C 와 C++

프로그래밍언어는 700개 이상 존재한다. 이중 일반적으로 알려지고 사용되고 있는 프로그래밍언어를 다양한 사이트에서 조사해서 발표한다. 이중 TIOBE에서 발표한 2020년 프로그래밍언어 순위는 다음과 같다. 조사결과를 보면 2020년 7월에 C언어가 1위를 차지했다. 자바, 파이썬 등을 제치고 C가 2020년에 1위 언어가 될 것이라고는 생각하지 못했다. C언어는 꾸준히 사용되던 언어면서 동시에 사물인터넷(IoT)의 소형디바이스나 차량용 임베디드 시스템에 많이 사용되면서 1위 언어로 등극했다. 불과 2020년 초반까지만해도 자바가 1위였지만 지금은 C가 1위 언어가 되었다. 물론 언어의 순위는 언제든 바뀔수 있다. C언어는 예전부터 지금까지 꾸준한 인기를 가지고 있다. 즉, 그만큼 사용되는 곳이 많다는 것을 의미하는 것이기도 하다.

Jul 2020	Jul 2019	Change	Programming Language	Ratings	Change
1	2	▲	C	16.45%	+2.24%
2	1	▼	Java	15.10%	+0.04%
3	3		Python	9.09%	-0.17%
4	4		C++	6.21%	-0.49%
5	5		C#	5.25%	+0.88%
6	6		Visual Basic	5.23%	+1.03%
7	7		JavaScript	2.48%	+0.18%
8	20	▲	R	2.41%	+1.57%
9	8	▼	PHP	1.90%	-0.27%
10	13	▲	Swift	1.43%	+0.31%
11	9	▼	SQL	1.40%	-0.58%
12	16	▲	Go	1.21%	+0.19%
13	12	▼	Assembly language	0.94%	-0.45%
14	19	▲	Perl	0.87%	-0.04%
15	14	▼	MATLAB	0.84%	-0.24%
16	11	▼	Ruby	0.81%	-0.83%
17	30	▲	Scratch	0.72%	+0.35%
18	33	▲	Rust	0.70%	+0.36%
19	23	▲	PL/SQL	0.68%	-0.01%
20	17	▼	Classic Visual Basic	0.66%	-0.35%

그림 49 프로그래밍 언어 순위

아두이노는 C 혹은 C++언어를 사용할 수 있다. C는 1972년 벨 연구소에서 유닉스개발을 맡은 데니스 리치에 의해 만들어졌다. 그당시만 해도 컴퓨터의 성능은 지금과 비교해 매우 떨어졌기 때문에 프로그램은 크기가 작으면서 빠르게 동작하도록 만들어져야했다. 이에 맞는 언어는 어셈블러밖에 없었고, 어셈블러는 특정 프로세서에 종속적이기 때문에 여러 기종에 사용할 수 없었다. 데니스리치는 이런 상황속에서 어셈블러의 강력한 기능과 함께 고급언어가 가지는 다양한 기종에 대한 이식성을 함께 갖는 프로그램언어의 필요성을 알고, C언어를 개발했다.

1972년에 C가 등장한 이후 약 10년후 C++ 언어가 나온다. C++ 은 비아르네 스트로우스루프(Bjarne Stroustrup)가 C언어를 바탕으로 만들었다. C의 문법중 ++ 는 1을 더해서 원래의 변수에 더한다는 의미가 있다. C++ 은 C 를 그대로 두고 한단계 업그레이드했다는 의미가 있다. C++ 문법에 대해서는 이 책에서 깊이있게 다루지 않을 예정이다. 기본적으로 C++ 은 C를 포함하고 있다. 그래서 C를 사용해서 만든 모든 것은 C++ 에서 정상적으로 작동한다. C++ 의 자체문법과 객체지향에 대한 내용들이 있지만 아두이노라는 열악한 메모리를 가진 환경에서는 C++ 문법을 사용한다는 것은 다소 과하기 때문이다. 32KB 의 작은 플래시메모리와 2KB 의 작은 램 용량을 사용해서 프로그램을 운영할 때 필요한 것은 가능한한 프로그램을 작게 만들고, 램을 적게 사용하도록 하는 것이다. 즉, 객체지향적 프로그램관점을 아두이노에는 적용할 수 없다.

아두이노는 오픈소스 C++ 컴파일러인 AVR-GCC 를 사용한다. C 와 C++ 문법을 사용할 수 있다. 이 책에서는 아두이노 우노와 메가에 적합한 C 언어를 기준으로 설명하지만 다른 종류의 MCU 에서도 동일하게 사용할 수 있다.

3.5. 아두이노 검색하기

아두이노 프로그램을 시작하면 C 문법과 함께 아두이노에서 제공하는 기본적인 라이브러리와 함수를 사용하게 된다. C 문법에 대해서는 다양한 책들을 참고할 수 있지만 아두이노에서 제공하는 라이브러리와 함수들에 대해서는 아두이노 사이트에서 검색으로 찾을수 있다. 단, 아두이노 IDE 를 사용하는 경우 그 안에 있는 아두이노 라이브러리에 있는 함수나 예약어들에 대해서는 검색기능을 사용할 수 있다.

아두이노 IDE 를 실행시킨 후 예제 중 하나를 선택해서 내부의 소스를 살

펴보기로 하자.

[파일] – [예제] – [01.Basics] – [AnalogReadSerial] 을 선택하면 아두이노 A0 핀의 값을 읽어서 시리얼모니터창에 값을 보여주는 프로그램 예제가 나온다.

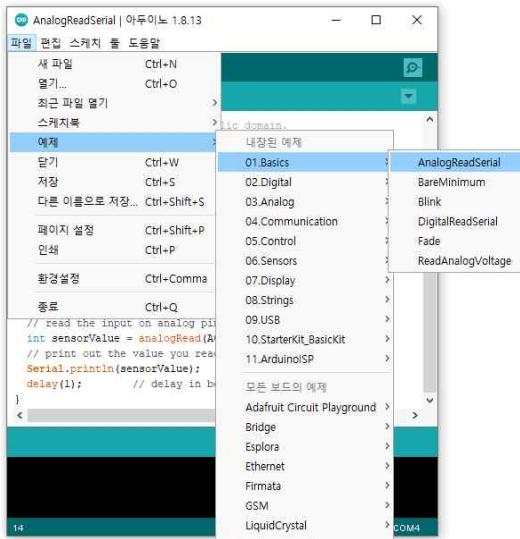


그림 50 IDE 예제

This example code is in the public domain.
<http://www.arduino.cc/en/Tutorial/AnalogReadSerial>

```
// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

// the loop routine runs over and over again forever:
void loop() {
  // read the input on analog pin 0:
  int sensorValue = analogRead(A0);
  // print out the value you read:
  Serial.println(sensorValue);
  delay(1); // delay in between reads for stability
}
```

Arduin Uno on COM4

그림 51 IDE 창

여기서 아두이노 기본 라이브러리에 속해 있다면 검색이 가능하다. `analogRead`라는 글씨 위에 마우스를 두고 우클릭한 후 [참조에서 찾기]를 클릭하면 arduino.cc 사이트에서 관련된 도움말이 나온다.

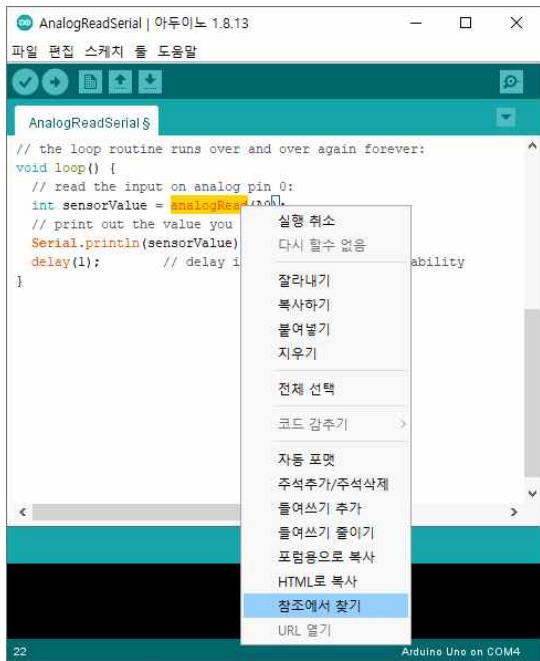


그림 52 analogRead() 도움말

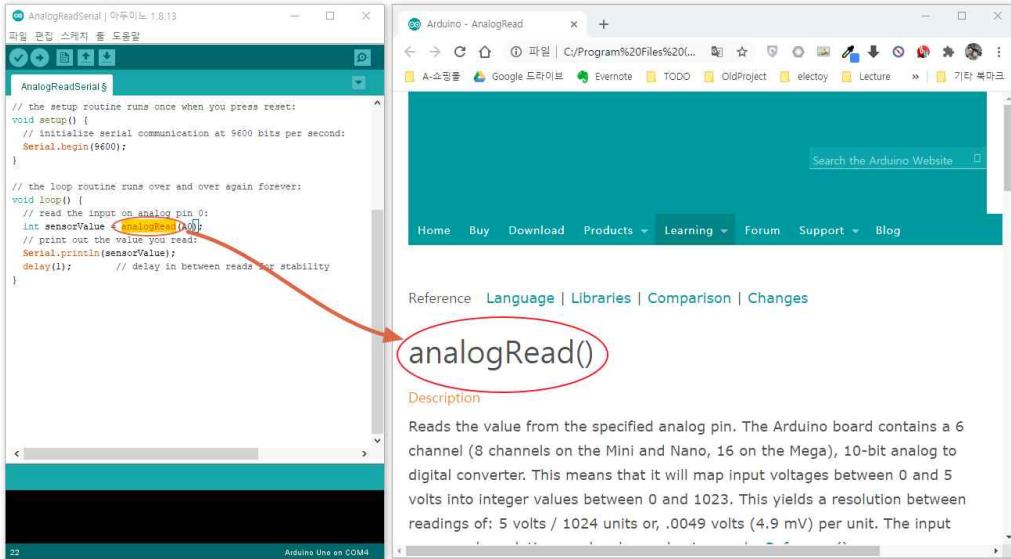


그림 53 도움말

아두이노 예제를 열어 보고 그 안에 있는 함수나 예약어들이 어떤 기능을 하는지 잘 모르겠다면 검색기능을 활용하면 된다. 최근 웹 브라우저들은 번역기능을 제공하는 것도 있어서 필요시 영어 – 한글 번역을 사용하면 꽤 놀라운 정도로 번역이 되어 나온다.

3.6 Hello World 그리고 Hello Led

프로그래밍 언어를 처음 접할때 첫번째 프로그램으로 “Hello World”를 화면에 출력한다. OnLine 컴파일러를 써서 PC에서 “Hello World” 출력해보자.

우선 온라인 컴파일러를 제공하는 사이트에 접속한다. 필자는 repl.it 을 주로 사용한다.

온라인으로 접속하고 C언어를 선택한 후 아래의 코드를 입력한다.

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

C의 기본 구조를 이해하기 위해서 한줄씩 살펴보겠다.

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

첫줄은 stdio.h 라는 파일을 가져오라는 의미다. #include 를 사용해서 확장자가 h 인 파일(헤더파일이라고 부른다) 중 stdio.h 를 가져오라는 뜻이다. stdio 는 STAnDard Input Output, 표준입출력을 의미한다. 표준입출력은 제대로 이해하려면 조금 복잡하다. 여기서는 키보드와 모니터를 이용한 입력과 출력을 의미한다는 정도로만 이해하고 넘어가자. 키보드를 사용하는 표준입력 장치와 모니터를 사용하는 표준 출력장치에 관련된 함수들이 stdio.h 안에 모여있다. 키보드와 모니터를 사용하기 위해서는 stdio.h 를 불러와서 사용해 주어야 한다.

C 언어는 함수로 모든 것이 만들어진다. 그중 가장 먼저 실행되는 함수의 이름이 main 이다.

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

메인함수는 main 이름 뒤의 괄호안에 main에서 사용할 외부 입력 값을 넣을 수 있다. 들어갈 값이 없을 경우 void를 써준다. main 이름 앞은 main 함수를 실행한 다음 나온 결과값을 전달하게 되는데, 이 전달되는 값의 타입을 의미한다. int는 정수형이고, main 함수가 끝날 때 정수값이 전달된다는 의미이다. 전달되는 값은 return이라는 명령어 뒤에 붙여서 쓴다. 모든 명령은 한번에 하나씩 사용한다. 명령의 끝에는 끝을 알리기 위해서 세미콜론(;)을 붙인다.

```
printf("Hello World\n");
```

메인함수 안에 printf() 함수가 들어있다. printf() 함수는 모니터로 원하는 값을 출력하기 위해 사용한다. 여기서는 " "로 묶여진 내용이 화면에 출력된다. 마지막 \n은 C에서 사용하는 일종의 약속이다. 일반 문자와 달리 특수문자들을 출력하거나 저장하기 위해서 텨, 엔터, 줄바꿈 등을 \t, \r, \n 등으로 표시한다. 문자열 안에 \n이 있으면 이것을 아래줄로 내리라는 특수문자인 개행문자로 인식하게 된다.

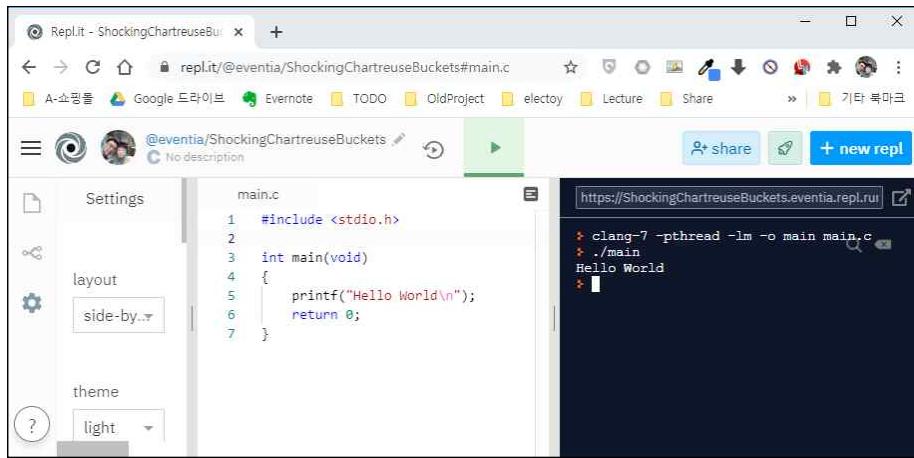


그림 54 온라인 C컴파일러

아두이노식으로 코드를 짜면 다음과 같다.

```
void setup()
{
    pinMode(LED_BUILTIN, OUTPUT);
}
void loop()
{
    digitalWrite(LED_BUILTIN, HIGH);
    delay(1000);
    digitalWrite(LED_BUILTIN, LOW);
    delay(1000);
}
```

아두이노는 키보드나 모니터가 없으므로 LED 를 켜고 끄는 것으로 간단한 프로그램을 시작한다. 위의 코드는 1초간 LED 를 켜고, 다시 1초간 LED 를 끄는 것을 반복한다.

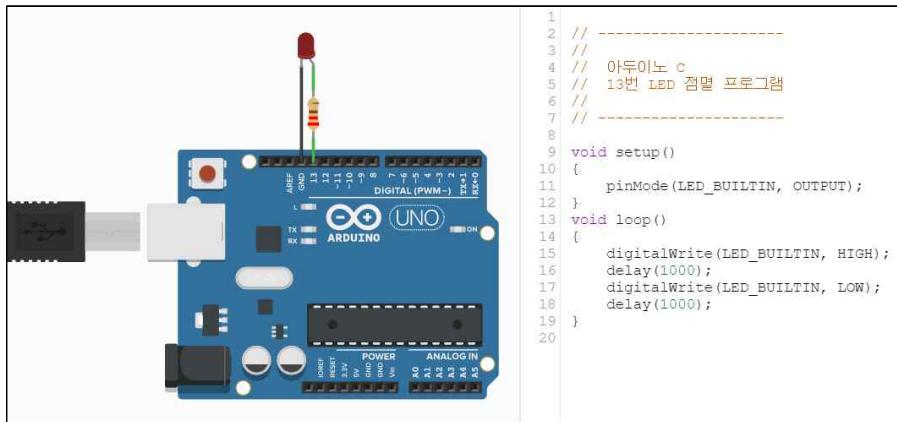


그림 55 아두이노 코드

3.7. 아두이노 코드의 특징

아두이노 코드는 기존의 C 언어 문법과 약간의 차이점이 보인다.

A. main() 함수가 없다.

항상 C는 main() 함수를 만들어서 시작했는데 아두이노는 특이하게도 main() 함수 자체가 없다. 마시모밴지는 MCU 프로그래밍의 특징이 한번 시작한 프로그램은 별다른 일이 없는 한 계속 작동한다는 것에 착안해서 main() 함수를 다음과 같이 미리 만들어 두고 사용자가 변경할 필요가 없도록 했다. 아두이노에서 미리 만들어 둔 main 함수는 C 드라이브에 프로그램이 설치되었다면 아래 위치에서 찾을 수 있다.

"C:\Program
Files
(x86)\Arduino\hardware\arduino\avr\cores\arduino\main.cpp"

파일 내부를 문서 편집기로 살펴보면 다음과 같다. 아래는 전체 소스가 아닌 편집된 일부분이다.

```
#include <Arduino.h>
int main(void)
{
    init();
    setup();          // setup() 함수가 들어가는 곳
    for (;;)
    {
        loop();       // loop() 함수가 들어가는 곳
        if (serialEventRun) serialEventRun();
    }
    return 0;
}
```

여기서 살펴볼 것은 setup(); 과 loop(); 가 있는 곳이다. 아두이노 IDE 에서 작성된 setup() 함수와 loop() 함수가 들어갈 위치다. 세부적인 내용은 이후 함수를 공부하면서 다루어 보도록 하겠다. 즉, setup() 과 loop() 를 외부에 만들어두면 그 내용이 위 처럼 적용되면서 main() 함수 안에서 setup() 함수는 1회만 실행되고, 이후에 loop() 함수는 무한반복된다. 이처럼 아두이노는 사용자가 main() 함수를 작성하지 않을 뿐이지 실제로 main() 함수는 존재한다.

B. #include <stdio.h> 가 없다.

PC에서 C를 사용할 때는 기본 입출력 방식으로 키보드와 모니터를 사용한다. 이것을 표준입출력이라고 하는데 아두이노는 키보드나 모니터를 사용하지 않고 작동하게 된다. 그래서 별도의 표준입출력과 관련된 라이브러리를 포함해서 PC에서 사용되는 라이브러리들을 사용하지 않는 편이다. 대신 아두이노에 필요한 헤더파일을 #include 해서 사용한다. main.cpp와 같은 곳에 위치한 Arduino.h 헤더파일은 아두이노 사용에 필요한 상수와 여러 함수들이 선언되어 있다. Arduino.h 헤더파일은 main.cpp에서 #include 명령으로 포함되어 있으므로 사용자가 신경쓸 필요는 없다.

C. printf() 함수를 사용하지 않는다.

PC 기반으로 작성된 함수와 라이브러리는 많은 리소스를 차지할 뿐 아두이노 환경에서는 그다지 도움이 되지 않는다. 위에 사용한 "Hello World"를 화면에 보여주는 간단한 프로그램을 만들때 실행파일의 크기가 128KB가 된다. PC에서 128KB는 작은 용량이지만 아두이노에서는 부담스러운 크기다. 아두이노 우노가 담을 수 있는 실행파일의 크기는 최대 32KB이고, 이중 0.5KB는 이미 부트로더를 담고 있기 때문에 사용할 수 없다. 게다가 아두이노는 키보드도 없고, 모니터도 없다. 시리얼통신을 통해 아두이노의 신호를 보는 것은 아두이노에 붙어있는 모니터를 보는 것이 아니라 아두이노에서 PC로 보낸 신호를 PC의 모니터로 보는 것이다. 즉 아두이노는 printf() 함수를 가질 여유도 없을 뿐더러 가져야 할 이유도 없다.

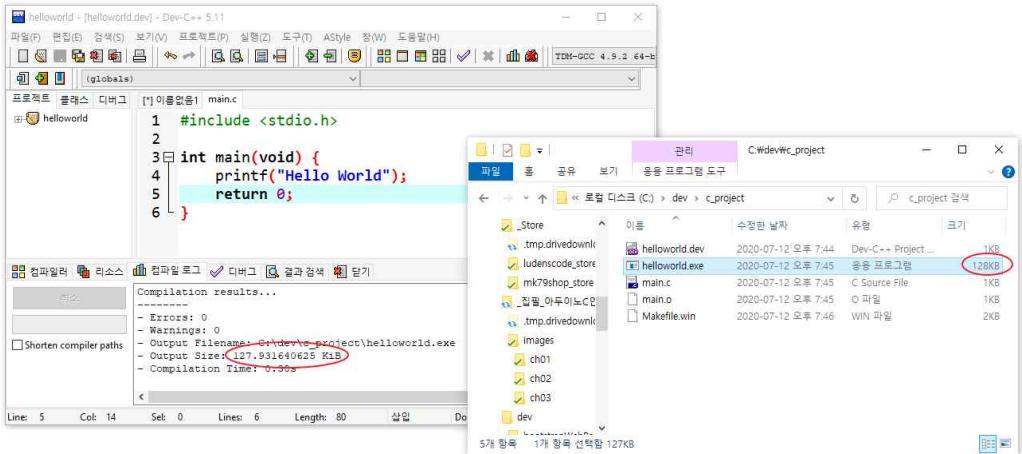


그림 56 실행파일 위치

D. digitalWrite() 와 delay() 등, PC 에서 사용하지 않는 함수들이 있다.

아두이노는 PC 와 달리 표준 입출력 장치인 키보드나 모니터를 사용하지 않고, 아두이노에 연결된 개별 장치들을 사용한다. PC 에서는 표준화된 키보드를 사용하기 때문에 어느 회사의 키보드를 사용하더라도 표준규격에 맞춰 제작했다면 신경쓸 필요가 없다. 그냥 키보드를 사서 컴퓨터에 연결하기만 하면 자동으로 연결된다. 반면 MCU 들은 대부분 이렇게 규격화되고 표준화된 장치가 없다. 그래서 개별적으로 연결해서 사용해야만 한다. 그때 사용되는 보다 낮은 단계의 함수들이 존재한다. 이런 함수들이 정의되고 선언된 곳이 Arduino.h 이다. pinMode(), digitalWrite(), delay() 등의 함수들을 사용해서 프로그래머가 필요로 하는 기능을 구현한다.

3.8. 다양한 라이브러리

오픈소스하드웨어 답게 아두이노에는 다양한 사람들이 만들어 둔 라이브러리가 있다. 아두이노가 나오기 전까지만 해도 이런 라이브러리들은 쉽게 구

할 수 없었고, 유료로 판매되는 것들이었다. 아두이노를 사용해 EEPROM, Ethernet, WIFI, SD카드, 서보모터등의 라이브러리를 사용할 수 있다. 현재 Arduino Library List 사이트(<https://www.arduinolibraries.info>)에 따르면 아두이노 라이브러리는 3098개 이지만 계속 추가되면서 이 숫자는 더 늘어날 것이다. 3098 이란 숫자는 아두이노 라이브러리 매니저에 등록된 숫자로 여기에 등록되지 않고 압축파일 등으로 제공되는 라이브러리를 포함하면 전체 라이브러리의 숫자는 훨씬 증가한다. 이런 라이브러리를 모두 배운다는 것은 불가능하다. 가장 기본적인 라이브러리를 익힌 다음, 제조사에서 제공하는 하드웨어(보통 아두이노와 연결되도록 만들어 둔 실드)와 함께 제공되는 라이브러리를 익히는 방식으로 자신에 필요한 것을 익혀 나가면 된다.

3.9. 시리얼 모니터

아두이노를 사용해서 프로그래밍을 한다는 것은 기본적으로 하드웨어를 제어한다는 것을 의미한다. 하드웨어를 직접 제어한다는 부분이 일반 PC 용 프로그램을 만드는 것과 다른 부분이다. PC 는 대부분 운영체제가 있다. 윈도우라는 운영체제를 사용한다는 것은 하드웨어 위에 윈도우라는 프로그램이 있고, 그 윈도우라는 프로그램에서 모든 하드웨어를 제어하게 된다. 즉, PC 프로그램을 하는 경우 하드웨어 제어는 프로그래머가 할 필요가 없다. 프로그래머는 어떤 모니터가 연결되는지, 어떤 키보드가 연결되는지 신경쓰지 않고 그저 printf() 나 scanf() 명령어로 모니터에 출력한 문자나 키보드로부터 받을 문자를 보내거나 받기만 하면 된다. 그러면 나머지는 윈도우라는 운영체제에서 알아서 하게 된다.

반면 아두이노는 이런 운영체제가 없다. 8비트의 32KB 플래시메모리에 2KB 램을 사용하는 아두이노는 다양한 하드웨어를 적절히 제어할 운영체제 없이 프로그래머가 직접 하드웨어를 제어해야만 한다. 그래서 아두이노와 같은 MCU 프로그램은 소프트웨어로 분류되기도 하지만 일반적으로는 펌웨어(firmware)라고 부른다.

아두이노는 하드웨어를 직접 제어해야 하고 이를 위해 필요한 기본 기능으로 시리얼통신을 이용한 모니터를 사용한다. PC에서 아두이노와 UBS 케이블을 사용하여 연결한 다음 시리얼포트를 통해서 PC와 아두이노간 신호가 오갈 수 있게 한다. 이걸로 PC에서 아두이노에 신호를 보내거나 혹은 아두이노로부터 오는 신호를 볼수 있다. 이것을 아두이노 시리얼 모니터라고 한다. 이후로는 C 언어를 공부하면서 그 결과를 시리얼통신을 통해 화면에 출력해서 확인하게 된다. 자세한 시리얼 통신에 대한 내용은 별도의 장을 통해 공부할 것이며 이곳에서는 간단한 설정 방법만 알아보기로 한다.

3 x 4 를 계산하는 프로그램을 만들어 본다. 아두이노 소스코드는 다음과 같다.

```
void setup()
{
    int i = 3;
    int j = 4;
    int k;
    k=i*j;
}
void loop()
{}
```

두개의 변수 i, j 에 각각 3과 4를 넣은 다음 변수 k 에 i 와 j를 곱한 결과를 저장했다. 그런데 이 결과가 제대로 되었는지 확인할 수가 없다. 아두이노 MCU 안에는 결과값이 저장되어 있겠지만 지금 우리는 그 값이 제대로 된 값인지 혹은 틀린 값인지를 확인할 방법이 없다. 그래서 아두이노와 PC를 연결해서 결과값을 가져와 보겠다.

여기서는 loop() 함수를 사용할 필요가 없으므로 비워둔 채로 진행한다.

setup() 함수 안에 시리얼통신을 시작하겠다는 것을 알린다.

```
Serial.begin(9600);
```

이 문장은 9600이라는 속도의 시리얼 통신을 PC와 아두이노 사이에 연결하겠다는 것이다.

이제 시리얼통신이 시작되었으므로 아두이노에서 PC쪽으로 신호를 보낼 수 있다. 신호를 보낼 때 Serial.println() 함수를 사용했다.

```
Serial.println(k);
```

위 문장은 k 변수에 있는 값을 시리얼통신으로 보내라는 명령이다. 이제 PC는 k의 값을 받게 된다.

위 내용을 끓어서 정리하면 다음과 같다.

```
void setup()
{
    Serial.begin(9600);
    int i = 3;
    int j = 4;
    int k;
    k=i*j;
```

```

    Serial.println(k);
}

void loop()
{
}

```

아두이노 IDE에서 실행한 결과와 Circuits (tinkercad.com의 Circuits)에서 실행한 결과는 다음과 같다. 아두이노 IDE의 경우 오른쪽 위에 있는 시리얼모니터 아이콘을 클릭하면 시리얼모니터 창이 나온다. Circuits의 경우 코드창이 보이는 상태에서 오른쪽 아래에 있는 [직렬 모니터]를 클릭하면 아래 화면과 같이 시리얼통신 창이 열린다.



그림 57 아두이노 시리얼모니터

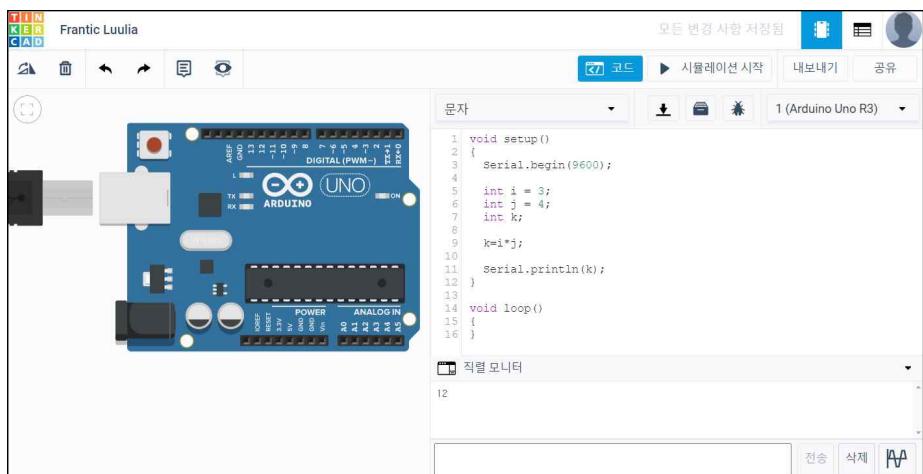


그림 58 tinkerCAD 에서의 시리얼모니터

Chapter 04

주요 학습 목표

- 주석에 대해 이해한다
- 변수의 필요성과 초기화 방법을 이해한다

Chapter 04. 주석, 변수, 상수

4.1. 아두이노 기본구조

아두이노 프로그램의 기본 구조는 2개의 블록으로 구분된다. 첫번째 블록은 실제 일을 하기 전 준비운동을 하는 단계로 `setup()`이라는 이름의 함수다. 두 번째 블록은 실제 일을 하는 단계로 `loop()`라는 이름의 함수다. `setup()`과 `loop()`은 정해진 이름으로 마음대로 바꿔서 쓸 수 없다. 3.7 절에서 `main()` 함수가 없는 아두이노 특성을 말하면서 `setup()`과 `loop()`로 분리해서 프로그래밍을 한다는 것을 보였다. 아두이노 이후로 `setup()`, `loop()`처럼 초기설정과 반복작업을 분리하는 경우들이 늘어나고 있다. TI는 최근 MCU들을 프로그래밍할 때 아두이노와 유사하게 적용하고 있다.

```
void setup()
{
    // 여기에 [준비운동]에 필요한 내용을 넣는다.
    // 초기 설정에 해당하는 코드를 넣는다.
}

void loop()
{
    // 여기에 [실제 할 일]을 넣는다.
    // MCU가 실제로 할 작업 코드를 넣는다.
```

```
}
```

A. setup()

setup() 함수는 아두이노가 시작되는 처음단계에서 한번만 실행된다. 원하는 작업을 하기 위해 13번 핀에 LED 를 연결해서 출력으로 사용하려면 먼저 그 핀을 출력으로 쓴다고 알려줘야 한다. 이때 pinMode() 라는 함수를 사용하고, pinMode() 는 setup() 함수 안에서 실행한다.

```
void setup()
{
    pinMode(13, OUTPUT);
}
```

명령의 끝은 항상 ; 로 마무리한다. 세미콜론(;)을 만나기 전까지 컴파일러는 명령이 끝나지 않았다고 판단한다. 처음 C 언어를 배우는 이들이 하는 실수는 ; 를 제대로 사용하지 않는 것이다.

B. loop()

loop() 함수는 반복적인 작업을 하는 명령을 담는 곳이다. setup() 에서 설정한 다음 실제 반복수행할 프로그램을 담는 곳이다.

4.2. 주석

주석(Comment)은 프로그램의 이해를 돋기 위해 프로그래머가 추가한 설명문이다. 즉, 주석은 사람을 위한 것이지 컴파일러가 읽을 수 있는 것이 아니다. 따라서 컴파일러는 주석을 만나면 무시한다.

C 에서는 두 종류의 주석이 가능하다. 슬래시 2개를 사용한 한줄 주석 // 과 /* 으로 시작해서 */ 으로 마치는 주석이 있다. 최초 C 언어에서는 한줄 주석은 없었다. 한줄 주석은 C++ 에서만 사용되었으나 최근 표준안에서는 한줄 주석도 C에 포함된다.

```
/*
여러줄 주석
아두이노 C 에서 주석의 사용법
*/
```

```
// 한줄 주석
```

주석에 대해서는 다양한 생각들이 공존한다. 기본적으로 유념할 것은 주석은 가능한 쓰지 않는 것이 좋다. 이 말은 주석을 무조건 쓰지 말라는 것이 아니라 주석이 필요없을 정도로 프로그래밍 코드를 쉽게 읽힐 수 있도록 만들라는 뜻이다. 지나치게 많은 주석은 그 자체로 코드를 읽기 어렵게 만든다. 주석이 없이도 한눈에 이해할 수 있도록 만드는 코드가 가장 좋은 코드다.

하지만 주석이 필요없을 정도로 코드를 만든다는 것이 쉬운 일은 아니다. 결국 주석은 코드의 이해를 위해 필요악적인 면이 있다. 가능한 적게 쓰면 좋지만 쓰지 않을 수 없는 것이 주석이다.

그래서 주석을 쓸때는 가능한 짧고 간결하게 쓰고, 핵심을 바로 알수 있도록 쓰는 것이 좋다. 단, 코드의 맨 앞부분에는 전체 프로그래밍 코드에 대한 간단한 요약과 저작권정보, 소유권정보를 넣어두는 것이 좋다.

아래처럼 코드의 시작부분에 아두이노 프로그램 간단한 요약과 저작권, 소유권 정보를 담아두는 것이 좋다.

```
/*
MyBlink

LED를 1초간 켜고 끄는 것을 반복한다.
LED_BUILTIN 은 아두이노 우노, 메가에서 13번핀과 연결되어 있다.

created 13 July 2020
by eventia@gmail.com
http://www.ludenscode.com/en/Tutorial/Blink
*/
```

위의 코드를 아래처럼 적기도 한다. 모두 같은 형식이다. 간단한 것을 원하면 위의 형식을 사용하면 되고, 조금 다듬어진 사각형 테이블 형식에 가까워지기 원한다면 아래 형식을 쓰면 된다. 위의 경우는 주석이 길어지면 중간부분이 주석인지 아닌지 빠르게 판단되지 않을 수 있지만, 아래의 경우는 줄의 앞부분에 표시가 되어 있으므로 주석의 처음과 끝을 한눈에 알수 있게 된다. 단, 왼벽하게 갖추어진 사각형을 만들어서 사용하는 것은 피하는 것이 좋다.

```
*****
* MyBlink
*
* LED를 1초간 켜고 끄는 것을 반복한다.
* LED_BUILTIN 은 아두이노 우노, 메가에서 13번핀과 연결되어 있다.
*
* created 13 July 2020
* by eventia@gmail.com
* http://www.arduino.cc/en/Tutorial/Blink
*****/
```

아래는 피하는 것이 좋다. 3가지 경우 모두 동일한 텍스트이고, 폰트만 달리 쓴 경우다. PC 가 달라지면 보이는 것이 조금 달라진다. 일반적으로는 큰 신경을 쓸 필요가 없지만 아래처럼 사각형 테두리를 만들면 문제가 달라진다. 오른쪽 끝 부분의 열이 맞지 않아 시각적으로 거슬린다.

```
*****  
* MyBlink  
*  
* LED를 1초간 켜고 끄는 것을 반복한다.  
*****
```

그림 59 폰트 - Courier New

```
*****  
* MyBlink  
*  
* LED를 1초간 켜고 끄는 것을 반복한다.  
*****
```

그림 60 폰트 - Consolas

```
*****  
* MyBlink  
*  
* LED를 1초간 켜고 끄는 것을 반복한다.  
*****
```

그림 61 폰트 - 나눔고딕코딩

주석으로 사각형 테두리를 만들어 사용하는 것을 피하는 첫째, 한글 폰트의 경우 영문과 달리 마음에 드는 고정폭폰트를 찾기가 어렵다. 그래서 정확한 사각형을 맞추기가 힘들다. 결국 우측에 비뚤어진 선은 계속 프로그

래머의 마음을 건드려서 코딩에 집중하지 못하게 한다. 코딩에 사용되는 한글 고정폭폰트로는 '나눔고딕코딩'이나 'D2 Coding' 등이 있다. 다만, 이 폰트가 설치되어 있지 않은 곳에서는 여전히 한글글꼴이 고정폭이 되지 못한다. 두번째로 주석부분을 수정할 때 오른쪽 테두리가 있는 경우 다시 테두리가 사각형에 맞게 고쳐야한다. 주석을 쓰면서 지나치게 많은 에너지를 소모하는 것은 바람직하지 않다/. 그래서 위의 주석은 아래처럼 쓰는 것이 좋다.

```
*****  
* MyBlink  
*  
* LED 를 1초간 켜고 끄는 것을 반복한다.  
*****
```

위에 추천된 폰트를 포함해서 코딩용 폰트는 알파벳 소문자 o, 대문자 O, 숫자 0 이 잘 구분되어야 하고 알파벳 대문자 I, 소문자 l, 숫자 1, 느낌표 !, 수직선(Vertical Bar) | 도 잘 구분되어야 한다. 특히 I, l , 1 이 한눈에 바로 구분되는 것이 중요하다. 이런 구분이 잘 되는 폰트를 다운받아서 사용하거나 아니면 기본 폰트중에서 Courier New 혹은 Consolas 를 사용한다. 필자도 자리를 옮기면 코딩을 하는 경우가 많고, 종종 내 PC 가 아닌 다른 PC 에서 수정작을 하는 경우가 많다. 그래서 기본으로 탑재된 위 두 폰트를 즐겨사용한다. Courier New 와 Consolas 는 한글폰트 없이 고정폭 영문폰트만 있다.

한줄 주석은 // 을 사용한다. 한줄로 간단하게 라이브러리, 함수가 무엇인지 설명하고, 왜 이렇게 동작하는지 설명한다.

```
// 이 함수는 초음파센서로부터 받은 거리를 출력한다
```

```
func( );
```

함수 설명을 한다. func() 함수가 어떤 함수인지를 설명한다.

```
// 리셋스위치가 눌려져 초음파 센서의 상대 거리가 초기화된다  
length = 0
```

length 가 0 이 된다는 것은 코드를 봐도 알 수 있다. 주석에 "거리가 0이 된다" 는 설명을 굳이 할 필요가 없다. 대신 그 의미나 이유를 기록하는 것 이 도움이 된다. 위 주석을 통해 왜 length 가 0 이 되었는지 알 수 있게 된다.

4.3. 자료형

프로그램에서 사용하는 데이터의 종류를 자료형(data type)이라고 부른다. C 에서는 크게 숫자와 문자로 나뉘고, 숫자는 정수형과 실수형으로 나뉜다. 그리고 boolean 이라는 0 과 1 로 나눌수 있는 자료형이 있다. boolean 은 C에서는 공식적으로 지원하지는 않지만 대부분의 C언어는 C++ 컴파일러를 사용하고 있기 때문에 아두이노에서도 boolean 자료형을 사용할 수 있다. 아래는 아두이노에서 사용하는 기본 자료형들이다.

자료형 (크기) : 설명

- **boolean (8 bits)** : 참, 거짓(true, false)
- **byte (8 bits)** : 0 에서 255 사이의 부호없는 정수
- **char (8 bits)** : -128 에서 127 사이의 부호있는 정수 또는 그 값을

가지는 문자

- `unsigned char (8 bits)` : byte 와 동일
- `word (16 bits)` : 0 에서 65,535 사이의 부호없는 정수
- `unsigned int (16 bits)` : word 와 동일
- `int (16 bits)` : -32,768 에서 32,767 사이의 정수
- `unsigned long (32 bits)` : 0 에서 4,294,967,295 사이의 부호없는 정수
- `long (32 bits)` : -2,147,483,648 에서 2,147,483,647 사이 부호있는 정수
- `float (32 bits)` : -3.4028235E38 에서 3.4028235E38 사이 부호있는 부동소수형 실수
- `double (32 bits)` : 아두이노 우노와 메가2560 에서는 `float` 과 동일,
- 아두이노 DUE 는 64bits 사용, PC의 경우도 `float` 의 2배의 비트인 64 bits 사용

위의 자료형은 기본 자료형으로 배열, 포인터, 문자열등을 다루지 않았다.

4.4. 변수

A. 메모리와 변수

변수는 데이터를 저장할 때 사용되는 기억 장소(RAM 메모리)의 이름이다. 변수에 데이터를 저장하고 읽고 변경할 수 있다. 쉽게 설명하면 변수는 '변할 수 있는 수'로, 어떤 값을 저장할 수 있는 공간을 의미한다. 변수는 빈 상자와 같아서 사과를 담거나 배를 담을 수 있다. 어떤 상자는 크기가 작아서 계란 하나만 담을 수 있다. 이 상자에는 배나, 사과를 담을 수는 없지만 담았던 계란을 꺼내고 다른 계란을 담을 수 있다.

이렇게 변수는 컴퓨터 램 메모리에 차지하는 공간을 의미한다. 작은 값을 저장할 수 있는 작은 변수도 있고, 큰 값을 저장할 수 있는 큰 변수도 있다.

변수를 쓰려면 우선 변수를 쓰겠다고 알려야 한다. 이걸 '선언'이라고 한다. C언어는 변수를 사용하기전에 반드시 변수를 사용하겠다는 것을 '선언'해서 컴파일러가 미리 준비할 수 있게 해 두어야 한다.

```
char c;  
int i;
```

변수를 선언할 때는 아래줄과 같이 하면 된다.

자료형 변수이름;

변수는 선언되면서 메모리에 정해진 크기 만큼의 공간을 확보한다. 이것을 '기억공간을 할당'한다고 말하는데, 위처럼 int value;라는 명령을 수행하는 순간 int 가 16 bits 의 기억공간을 필요로 하기 때문에 메모리에 비어있는 16 bits 만큼을 확보한다는 것이다. 이렇게 확보된 16 bits 의 공간은 다른 곳에서 함부로 읽거나 쓰지 못하게 잠근다. 이렇게 확보된 공간에 접근하는 일반적인 방법은 변수를 사용하는 것이다(이 책의 후반부에서 다른 방법 – 포인터 사용 등 – 도 알게 된다).

주소	내용								변수 이름
416									
417									i
418									
419									c
420									
421									

그림 62 메모리와 변수

int 데이터형으로 선언된 변수 i 는 16 bits 만큼의 공간을 확보하고, char 형으로 선언된 변수 c 는 8 bits 만큼의 공간을 확보한다.

기본적으로 8 bits 는 디지털공간에서는 1 byte 로 부른다. 앞으로는 8 bits 의 크기를 1 byte 로 부르겠다. 그러면 char 은 1 byte, int 는 2 byte, long 과 float 은 4 byte 가 된다.

B. 변수의 식별자

변수, 상수, 함수등의 이름을 식별자(identifier)라고 부른다. 3장에서 카멜 표기법, 파스칼 표기법, 스네이크 표기법을 살펴보았다. 그 표기법을 사용해서 변수의 이름을 붙이면 된다. 참고로 아직 셋중에 정한 것이 없다면 변수 이름은 카멜표기법을 사용할 것을 권한다. 이유는 다수의 프로그래머들이 변수이름으로 카멜표기법을 사용하고 있기 때문이다.

변수 이름을 지을 때 주의할 점이 몇 가지 있다. 식별자는 문자와 숫자로 구성되면 첫글자는 반드시 문자여야 한다. 문자는 밑줄문자 ‘_’를 포함, 영문 소문자와 대문자를 의미한다. 한글이나 특수 문자, 공백등이 사용되면 안된다. 대소문자는 다른 문자로 구분된다. APPLE, Apple, apple 는 모두 다른 이름이다.

잘못된 예	바른 예
25x	x25
Z#	Z
99adress	address99
sun*	sun
-space	_space
class-a	class_a

그림 63 변수 이름의 예

위 [표4-2]에서 잘못된 예를 보였지만 더 좋은 것은 변수의 이름 그 변수가 어떤 변수인지 바로 알 수 있도록 짓는 것이다. 명사, 형용사, 부사의 조합으로 만드는 것이 일반적이며 동사나 관사, 전치사는 가능한 생략한다.

user, users, userLists 처럼 변수 이름을 보고 바로 알 수 있게 만든다. numberOfRows 의 경우 중간에 있는 Of 라는 전치사가 중요하지 않으면서도 뺏어보인다. 이런 경우 userCounts 정도로 쓰는 것이 좋다. 단수라고 해서 a 라는 관사를 사용하지 않는 것이 일반적이고, of 같은 전치사는 가능한 사용하지 않고 명사+명사, 형용사+명사로 만드는 것이 좋다. 중고차가 격을 담는 변수는 usedCarPrice 정도면 좋겠고, 신차가격은 newCarPrice

면 적당하다. camelCase 대신 snake_case 를 사용할 경우 used_car_price 도 좋다. 다만, 변수이름으로 PascalCase 는 잘 사용되지 않는다.

물론 이 모든 것보다 우선되어야 할 것은 함께 일하는 이들과 약속을 하는 것이다. 이미 정해진 이름 규칙(Naming Rule)을 가지고 있는 집단에 들어가는 경우 그들이 가지고 있는 규칙을 따르는 것이 좋다. 로마에서는 로마의 법을 따라야한다.

C. 변수 초기화

변수는 선언하면서 초기화를 함께 할 수 있다. 이를 변수의 초기화라 한다. 변수 초기화는 필요에 의해 하는 선택사항이다. 초기화되지 않은 변수는 그 안에 어떤 값이 들어가 있는지 확신할 수 없다. 대부분의 컴파일러는 초기 값으로 0 을 사용하지만 그렇지 않을 수도 있다. 그래서 가능한 변수는 선언과 동시에 초기값을 넣어주는 것이 일반적이다.

```
int sum = 0;
```

동일한 타입의 데이터형이라면 한 줄에 둘 이상의 변수를 선언하고 초기화 할 수 있다. 아래는 두번째 count 변수가 0으로 초기화되고, 나머지 2개의 변수는 초기화되지 않았다.

```
int sum, count = 0, result;
```

정수형 변수의 선언과 초기화 사용 예제를 살펴보겠다.

아래 예제를 보면 다양한 자료형을 사용하여 변수를 만들어보고, 그 변수를

출력하는 연습을 진행해 보라. 참고로 사용가능한 범위를 넘어가는 값을 변수에 넣어서 출력하면서 어떻게 되는지를 확인해 보라

[프로그램 4.1]

```
/*********************************************
* CH0401_Variable.ino
* 정수 변수 사용
********************************************/
void setup()
{
    Serial.begin(9600);

    char c = 'X';
    int i = 416;
    long l = 2000000000;
    float f = 3.141592;

    Serial.print("c = ");
    Serial.println(c);
    Serial.print("i = ");
    Serial.println(i);
    Serial.print("l = ");
    Serial.println(l);
    Serial.print("f = ");
    Serial.println(f);
}

void loop()
{}
```

[실행결과]

```
i = 416  
c = X  
l = 2000000000  
f = 3.14
```

D. sizeof() 함수

변수의 크기를 확인하고 싶다면 sizeof() 함수를 사용하면 된다. sizeof() 함수를 이용해 변수의 크기, 즉 변수가 몇바이트로 구성되어 있는지 확인할 수 있다.

```
*****  
* CH0402_VariableSize.ino  
* 변수 크기 구하기  
*****  
  
void setup()  
{  
    Serial.begin(9600);  
  
    Serial.print("char : ");    Serial.print(sizeof(char));  
    Serial.print(", int : ");  Serial.print(sizeof(int));  
    Serial.print(", long : "); Serial.print(sizeof(long));  
    Serial.print(", float : "); Serial.print(sizeof(float));  
}
```

```
void loop()
{
}
```

[실행결과]

```
char : 1, int : 2, long : 4, float : 4
```

위의 소스에서 한 줄에 두개의 명령을 함께 넣었다. 한 줄에 두 명령을 넣는 것은 그다지 바람직하지 않다. 다만, 위처럼 더 명료한 특별한 경우에만 한 줄에 둘 이상의 명령을 넣는 것을 고려하는 것이 좋다.

세미콜론(;)을 만나면 명령의 끝을 의미하고, 그 뒤에 나오는 문자는 다시 명령어 첫부분을 의미하게 된다.

```
Serial.print("char : ");    Serial.print(sizeof(char));
```

명령은

```
Serial.print("char : ");
Serial.print(sizeof(char));
```

과 완벽하게 동일하다.

Serial.print() 는 () 안의 내용을 시리얼통신으로 보내는 것이고, Serial.println() 는 () 안의 내용의 마지막 부분에 line feed 인 \n 개행 문자를 덧붙인다.

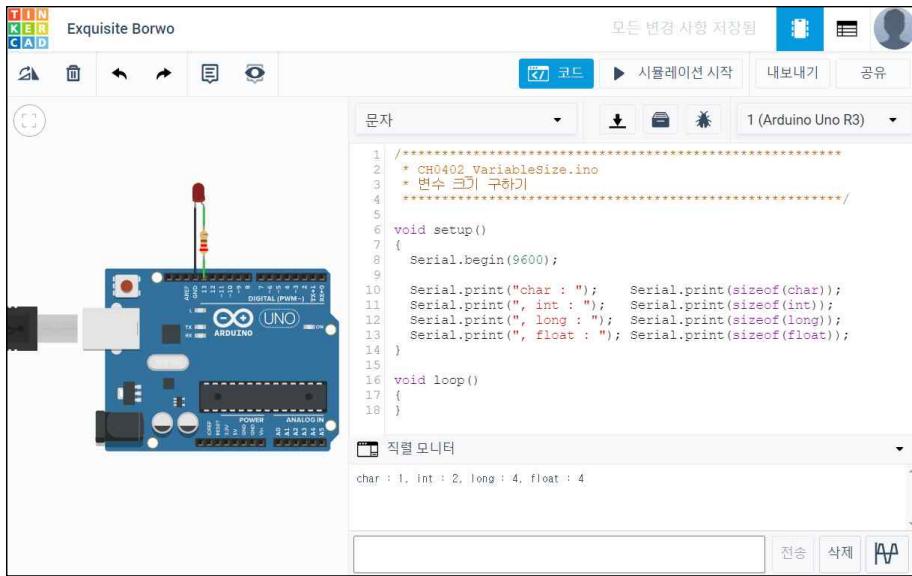


그림 64 변수의 크기

변수의 값을 변화시킬 때 대입연산자를 사용한다. '=' 는 수학기호로는 좌변과 우변이 같다는 의미로 사용되지만 프로그래밍 언어에서는 우측에 있는 값을 좌측의 공간에 넣으라는 의미로 사용된다. 즉 i 에 1 더한 값을 넣을 때는 다음과 같이 한다.

$i = i+1$

이 명령을 자세히 살펴보면 첫째, '=' 의 우측에 있는 값($i+1$)을 계산한 다음 둘째, 그 값을 변수 i에 넣어준다. 즉, i 가 10 이었다면 이후에 위 명령을 마친 다음 i 는 11 이 된다. 변수는 값을 저장하는 (메모리)공간이라 는 것이 가장 중요하다.

E. 전역변수, 지역변수

변수가 있는 위치에 따라 전역변수와 지역변수로 나뉜다.

C 언어는 기본적으로 함수를 기반으로 하고 있다. 함수안에서 생성된 변수는 그 함수 안에서만 유효하다. 즉, 함수 바깥에서는 함수 안에서 생성된 변수는 사용할 수 없다. 다음과 같이 사용할 수 없다.

```
void setup()
{
    int i = 10;
    Serial.begin(9600);
}

void loop()
{
    Serial.println(i);
}
```

setup() 함수 안에서 만들어진 변수 i 는 setup() 함수 안에서만 유효하다. 즉, setup() 함수 안에서 만들어진 변수는 loop() 함수안에서는 사용할 수 없다. 이런 변수를 지역에 종속된 변수라 하여 지역변수라 부른다. 지역변수는 만들어진 공간안에서만 사용할 수 있다. 위의 소스를 실행시켜 려하면 문법오류가 발생해서 컴파일이 되지 않는다.

반면 어느 곳에서든 사용하기 위해서 함수 외부에서 만드는 변수를 전역변수라고 한다. 변수 i 를 setup() 함수 외부에 만들면 다음과 같다.

```

int i = 10;      // 함수의 외부에서 만들어진 전역변수

void setup()
{
    Serial.begin(9600);
    Serial.println(i); // i 는 전역변수이기 때문에 사용가능
}

void loop()
{
    Serial.println(i); // 전역변수
    delay(30000);
}

```

변수 i 는 setup(), loop() 의 밖에서 만들어졌기 때문에 전역변수다. 전역변수는 어디에서도 사용할 수 있다. setup() 과 loop() 내부에서도 i 를 사용할 수 있다. 어디에서도 사용할 수 있기 때문에 편리하지만 동시에 위험하기도 하다. 전역변수로 만들어진 변수는 어느 곳에서도 수정이 가능하기 때문에 오류가 발생했을 때 그 오류를 찾기가 어려워진다. i 라는 변수를 전역변수로 만들었고, i 변수를 100 개의 함수가 사용하고 있다면 그 모든 함수를 다 뒤져가며 오류를 찾아야한다. 그래서 전역변수는 가능한 사용하지 않는것이 대부분의 프로그래머들이 가지고 있는 불문율이다. 다만, 빠른 프로그래밍이 필요한 알고리즘 대회나 메모리가 작고 전체 프로그램 코드가 짧은 MCU 프로그램에서는 전역변수의 사용이 다소 관대한 편이다.

4.5. 상수

상수는 변수와 달리 항상 일정한 값, 변화되지 않는 정해진 값을 의미한다. 초창기 어셈블러를 사용하던 시절의 변수는 RAM 메모리의 공간을 의미했고, 상수는 ROM 메모리의 공간을 의미했다. RAM은 쓰고 지울 수 있는 메모리라면 ROM은 읽기만 할 수 있는 메모리였다. ROM에 최초에 한번 기록된 값은 읽기만 할 수 있고, 쓰거나 변경할 수 없었다. 여기에 PI(=3.141592...)의 값이나 혹은 프로그램 수행에 필요한 정해진 값을 넣어두었다. 반면 계속 바뀌는 값은 RAM 메모리에 변수로 기록해 두었다. 이것이 C언어를 사용하면서도 어느 정도 초창기에는 유지되었다. 변수는 RAM 메모리를 사용하지만 상수는 RAM의 공간을 사용하지 않고 ROM 메모리에 기록했다. 아두이노에 사용되는 MCU들은 프로그램코드를 기존의 (EP)ROM 대신 플래시메모리에 저장한다. 즉, 아두이노에서도 상수는 RAM 메모리를 사용하지 않는다.

A. F() 매크로

F() 매크로 함수는 RAM 대신 플래시메모리에 데이터를 저장할 수 있게 해준다. F()를 사용한 것과 사용하지 않은 경우에 대해 메모리 사용량을 아래 코드를 통해 살펴보라.

변수사용

```
void setup()
{
    Serial.begin(9600);
    Serial.print("Is this string in flash?");
}

void loop()
{
```

```
}
```

[컴파일 결과]

스케치는 프로그램 저장 공간 1480 바이트(4%)를 사용. 최대 32256 바이트. 전역 변수는 동적 메모리 210바이트(10%)를 사용, 1838바이트의 지역 변수가 남음. 최대는 2048 바이트.

상수사용

```
void setup()
{
    Serial.begin(9600);
    Serial.print(F("Is this string in flash?"));
}

void loop()
{}
```

[컴파일 결과]

스케치는 프로그램 저장 공간 1492 바이트(4%)를 사용. 최대 32256 바이트. 전역 변수는 동적 메모리 184바이트(8%)를 사용, 1864바이트의 지역 변수가 남음. 최대는 2048 바이트.

램 메모리가 부족한 MCU 에서는 문자열을 다루는 것이 상당한 부담이 된다. 기본적으로 문자열은 하나의 문자가 1 byte 를 차지하고, 문자의 크기 만큼 메모리를 차지하게 된다. 2GB 이상의 램을 가지는 PC 에서는 몇 천 byte 도 부담스럽지 않은 메모리지만 아두이노 우노같이 2KB 의 램을 가지는 MCU 는 몇십 byte 만 되도 상당한 부담이 된다. 이때 램보다 상대적으로 큰 플래시 메모리를 사용하는 방법이 있다. 문자열은 특성상 고정된 문구가 나오고 반복해서 같은 문구가 나오게 된다. 이 문자열을 램 메모리

대신 플래시메모리에 저장해서 사용하게 한다. 단 이 경우 주의할 것은 플래시메모리에 있기 때문에 수정할 수 없다는 점이다.

PC 의 C 프로그램에서는 거의 다루지 않는 방법이지만 8비트 MCU 를 사용하는 현대의 개발자들은 기본적으로 코드메모리(플래시메모리)에 상수를 등록하는 방법을 사용한다. 아두이노는 AVR MCU 를 사용하므로 PROGMEM 을 사용해서 플래시메모리에 상수를 기록해서 읽어내는 방법을 사용한다. 가장 쉽게는 F() 매크로를 사용한다.

```
Serial.print(F("Is this string in flash?"));
```

시리얼 통신을 사용하는 경우 문자열 대신 문자열을 F() 매크로를 사용하여 전달하면 된다. 문자열을 한번만 출력하는 경우 F() 매크로를 사용하는 것이 편하다. 하지만 동일한 문자열을 여러곳에서 출력한다면 그때마다 F() 매크로를 사용하면 사용한 횟수만큼 플래시메모리를 사용한다.

B. const 와 함께 PROGMEM 사용

const 는 C언어에서 상수를 정의할 때 사용되는 키워드다. int a = 1; 는 a라는 이름의 변수를 만들어 거기에 1을 넣는 것이라면 const int b = 1; 명령은 b라는 이름으로 상수를 만들고 상수의 값을 1로 고정시킨다. 상수는 변수와 달리 한번 설정된 값이 변경되지 않는다. 즉, 변경시켜서는 안 되는 값을 사용할 때 안전을 위해 변수보다는 상수를 사용하는 것이 좋다. 이를테면 pi라는 변수를 만들어 3.141592라는 값을 저장하는 것은 이후 누군가에 의해 실수로 그 값이 바뀔 가능성을 가진다. 하지만 PI 이름으로 상수를 만들어 3.141592를 저장하면 이후 PI = 3.14;라는 명령이 주어질 때 PI는 상수이므로 값을 변경시킬 수 없다는 메시지가 나오게 되고, 실수나 오류에 의해 PI 값이 변경될 위험이 사라지게 된다.

하지만 C언어는 기본적으로 상수도 RAM 영역에 저장해서 사용된다.

아두이노 우노의 플래시메모리는 32KB로 PC의 일반적인 보조기억장치인 HDD, SSD와 비교해본다면 상당히 작지만 램 용량인 2KB 보다는 많다. 그래서 한번 정의되고 다시 수정되지 않는 값인 상수를 램이 아닌 플래시 메모리에 저장하는 것이 가능하다. 즉, RAM을 사용은 최소화하는 것이 좋다. 그때 PROGMEM을 사용해서 아래와 같이 프로그램할 수 있다.

```
*****
* CH0406_PROGMEM_F.ino
* 플래시메모리 사용
*****  
  
#define G(s) ((const __FlashStringHelper*)(s))  
const char hello[] PROGMEM = "Hello World";  
  
void setup()  
{  
    Serial.begin(9600);  
    Serial.println(F("Hello World!")); // F() 매크로 사용  
    Serial.println(G(hello)); // 사용자 정의한 G() 매크로 이용  
}  
  
void loop()  
{  
}
```

[결과 : 프로그램저장공간 1556, 동적메모리 188]

프로그램 저장 공간(플래시메모리) 1556 바이트, 동적 메모리(램) 188 바이트가 사용된다. 프로그램 저장 공간과 동적 메모리의 크기는 컴파일러 버전이 달라지면 바뀔 수 있고, OS 와 OS가 32비트인지 64비트인지에 따라서 달라질 수 있다.

setup() 을 아래처럼 F() 매크로를 여러번 사용하는 것으로 수정하면 프로그램 저장 공간과 동적 메모리 사용량은 달라진다.

PROGMEM 을 사용해서 플래시 메모리에 저장된 내용을 print() 로 출력할 때 직접 출력을 하면 정상적인 결과가 나오지 않는다. 이 경우 __FlashStringHelper 을 사용하여 형변화 내용을 G() 매크로 함수로 형변환을 해주는 것이 필요하다.

```
void setup()
{
    Serial.begin(9600);
    Serial.println(F("Hello World!")); // F() 매크로 사용
    Serial.println(F("Hello World!")); // F() 매크로 사용
    Serial.println(F("Hello World!")); // F() 매크로 사용
    Serial.println(G(hello)); // 사용자 정의한 G() 매크로 이용
}
```

[결과 : 프로그램저장공간 1598, 동적메모리 188]

동적메모리 사용량은 동일하고 프로그램 저장공간이 42 바이트 늘었다. 이번엔 G(hello) 사용을 3번 늘려보겠다.

```
void setup()
{
    Serial.begin(9600);
    Serial.println(F("Hello World!")); // F() 매크로 사용
    Serial.println(G(hello)); // 사용자 정의한 G() 매크로 이용
    Serial.println(G(hello)); // 사용자 정의한 G() 매크로 이용
    Serial.println(G(hello)); // 사용자 정의한 G() 매크로 이용
}
```

[결과 : 프로그램저장공간 1572, 동적메모리 188]

프로그램저장공간이 16 바이트 늘었다. F() 매크로를 사용하는 것 보다 26만큼 적게 사용되었다. 한번 사용할 때는 F() 를, 반복해서 사용할 때는 PROGMEM에 저장한 후 저장된 것을 사용하는 것이 효율적이다.

F() 매크로를 사용할 때는 문자열을 반복횟수만큼 플래시메모리에 쓰기 때문에 그만큼 플래시 메모리 사용량이 늘었지만, PROGMEM을 사용해서 플래시메모리에 문자열을 저장해 두고 그것을 G()로 불러 사용했기 때문에 명령은 반복되지만 데이터는 같은 주소의 것을 사용해서 결과적으로 플래시메모리의 소모를 줄였다.

PC 와 달리 메모리가 부족한 MCU에서 프로그램을 할때는 PROGMEM 등을 제대로 사용해야 한다.

4.6. 정수형

우리가 사는 세상에서 수를 배울때 자연수, 정수, 유리수, 분수 등으로 확장해가며 수의 개념을 배운다. 기본적으로 C에서 사용하는 수는 정수와 실수다. 정수는 음의 정수, 0, 양의 정수가 포함되고, 실수는 정수가 포함된

모든 수를 의미한다. 이런 정수와 실수를 저장하기 위해서 C는 메모리의 특정한 크기를 사용한다. 이 크기에 따라 8비트, 16비트, 32비트 64비트 등으로 구분된다. 아래 표는 정수를 비트의 크기로 구분한 표이다.

아두이노는 8비트 Atmega328p라는 MCU를 사용하므로 32비트 기반의 MCU나 PC와는 자료형이 다른 부분이 있다.

정수자료형	비트수	최소값	최대값
char	8-bit (1-byte)	$-2^7 = -128$	$2^7 - 1 = 127$
byte	8-bit (1-byte)	0	$2^8 - 1 = 255$
unsigned char	8-bit (1-byte)	0	$2^8 - 1 = 255$
int	16-bit (2-byte)	$-2^{15} = -32,768$	$2^{15} - 1 = 32,767$
short	16-bit (2-byte)	$-2^{15} = -32,768$	$2^{15} - 1 = 32,767$
word	16-bit (2-byte)	0	$2^{32} - 1 = 65,535$
unsigned int	16-bit (2-byte)	0	$2^{32} - 1 = 65,535$
unsigned short	16-bit (2-byte)	0	$2^{32} - 1 = 65,535$
long	32-bit (4-byte)	$-2^{31} = -2,147,483,648$	$2^{31} - 1 = 2,147,483,648$
unsigned long	32-bit (4-byte)	0	$2^{32} - 1 = 4,294,967,295$

그림 65 정수자료형

A. 정수표현

정수 자료형에 char 자료형을 포함시켰다. 일반적으로 C 언어에서는 char 자료형은 정수자료형이 아닌 문자자료형으로 분류된다. 8 비트로 표현할 수 있는 수는 총 256개로 0을 중심으로 음수와 양수로 나누면 -128에서 127 까지의 수가 된다. 이 범위는 상당히 작은편에 속하므로 일반적으로 사칙연산등을 하는 숫자에 적용되지 않는다. 그래서 char 자료형은 문자

형으로 구분되고, 정수 숫자를 사용하는 가장 작은 단위로 int 형이 사용된다.

아두이노는 메모리의 크기가 작으므로 최대한 작은 공간을 사용하도록 프로그램하는 것이 좋다. 숫자를 저장하지만 숫자가 127을 넘지 않는다면 int 대신 char 형을 사용할 수 있고, 음수가 없다면 byte 나 unsigned char 를 사용할 수 있다. 위의 표에서 보이듯 byte 는 아두이노만 가지고 있는 특수한 자료형이다. 실제로 byte 는 unsigned char 자료형과 동일하다. unsigned char 의 이름만 바꾼것이 byte 이다.

이를테면 한반의 학생들의 성적을 저장하는 변수를 사용한다면 0 점에서 100점까지만 저장하면 된다. 이때 2 Byte 를 사용하는 int 대신 1 Byte 를 사용하는 char 이나 byte 를 사용하면 int 를 사용하는 것에 비해 메모리를 반으로 줄일 수 있다.

가능한 저장할 수의 크기에 따라 적절한 자료형을 사용하는 것이 좋다. byte 로 저장해도 될 성적데이터를 4 Byte 짜리 long 을 사용해서 저장하는 것은 자원의 낭비다. 단, 값은 범위가 확실하지 않을때는 충분히 큰 자료형을 사용해야한다.

비트로 정수의 크기를 알수 있다. 간단한 8비트 byte 자료형을 살펴보겠다.

byte 변수는 0 부터 255 까지의 정수 숫자를 저장할 수 있다. byte 변수는 8비트, 즉 1개의 바이트를 사용한다. 1 바이트는 8개의 비트의 모음이다. 일반적으로 아두이노 같은 마이크로프로세서는 8비트, 즉 1바이트를 기본 단위로 한다. 1 비트는 0 또는 1을 보관할 수 있는 최소의 공간이다. 물질 세계를 구성하는 기본 단위가 아톰(Atom) 이라면 정보세계를 구성하는 기본단위는 비트(Bit)다. 정보가 보관된 곳을 1로 표시하고 비어있는 것을 0 으로 표시하면 다음과 같다.

이진수	십진수	계산
0000 0000	0	$= 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
0000 0001	1	$= 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
0000 0010	2	$= 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
0000 0011	3	$= 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
0000 0100	4	$= 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
...
1111 1101	253	$= 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
1111 1110	254	$= 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
1111 1111	255	$= 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

그림 66 이진수와 십진수 관계

8개의 빙, 8개 비트를 가지고 셀 수 있는 수는 0부터 255까지 총 256 개가 된다. byte에 담을 수 있는 최대 수는 255다. 이보다 더 큰수가 들어오면 다시 0부터 시작된다. 아래 코드를 실행해보면 8비트에 담길 수 없는 값이 들어갈 때 어떻게 되는지 알수 있다. 실행하기 전에 화면에 나올 숫자를 예상해 보라. 그리고 그 예상이 맞는지 확인해 보라.

```
void setup()
{
    Serial.begin(9600);
    byte maxNumber = 255;
    Serial.println(maxNumber);
    maxNumber = maxNumber + 1;
```

```

    Serial.println(maxNumber);
}
void loop(){ }

```

[결과]

255

0

maxNumber에 십진수 255가 들어가면 이진수로 변환되어 메모리에 저장된다. 십진수 255는 이진수 1111 1111이다. 즉, 8개의 비트가 모두 꽉찬 상태가 된다. 여기에 1을 더하면 다음과 같다.

$$\begin{array}{r}
 1111 1111 \\
 + \quad \quad \quad 1 \\
 \hline
 1 0000 0000
 \end{array}$$

그림 67 이진수 덧셈

이진수의 덧셈이므로 $1 + 1$ 은 2가 아니라 10이 된다.

그래서 이진수 1111 1111 + 1은 9개의 비트가 필요한 1 0000 0000이 되고, 8개의 비트밖에 없는 byte 변수는 가장 왼쪽에 있는 하나의 비트에 담긴 값을 읽지 못하게 된다. 즉, 1 0000 0000이 계산 결과지만 화면에 쓰이는 것은 0000 0000이 되는 것이다. byte 변수는 맨 왼쪽 1을 버린 나머지 8개의 비트값을 저장한다. 그래서 위의 코드를 실행하면 위에서 255가 아래에서 0이 출력된다.

B. 이진수와 십진수 변환

십진수가 인간이 사용하는 수를 세는 방법이라면 이진수는 전기의 흐름을 있다와 없다로 판단하는 컴퓨터가 수를 세는 방법이다. 십진수는 각 자리마다 0에서 9까지 총 10개의 단계를 가진다. 반면 이진수는 각 자리마다 0과 1의 총 2개의 단계를 가진다. 그래서 이진수라 부른다.

이진수는 다음과 같이 10진수로 바꿀 수 있다.

$$\begin{aligned} \text{(이진수) } & \text{XXXX XXXX} = (\text{십진수}) X \times 2^7 + X \times 2^6 + X \times 2^5 + X \times 2^4 + X \times 2^3 + X \times 2^2 + X \times 2^1 + X \times 2^0 \\ \text{(이진수) } & 0000\ 1001 = 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ & = (\text{십진수}) 9 \end{aligned}$$

그림 68 이진수를 십진수로 변환

C에서 십진수와 이진수를 쓰는 표기법이 있다. 그냥 숫자를 쓰면 십진수로 C언어는 인식한다. 숫자의 앞에 0B 혹은 0b 를 붙여 쓰면 이진수로 인식한다. 즉, 10은 십진수 "십"이고, 0B10은 이진수 "일공"으로 인식한다.

C. 10진수, 8진수, 16진수, 2진수 표기법

아두이노는 2진수와 16진수를 자주 사용한다. 가끔 8진수를 사용할 때도 있다. 각 진법을 사용할 때 다음의 규칙으로 표시한다.

	표기	십진수 변환 값
2진수	0B11	$1 * 2^1 + 1 * 2^0 = 3$
8진수	011	$1 * 8^1 + 1 * 8^0 = 9$
16진수	0x11	$1 * 16^1 + 1 * 16^0 = 17$
10진수	11	$1 * 10^1 + 1 * 10^0 = 11$

그림 69 변환방법

위 4개의 값을 시리얼로 출력하는 코드를 만들어보면 다음과 같다.

```
void setup()
{
    Serial.begin(9600);

    int binNum = 0B11;      // 2진수 표현
    int octNum = 011;      // 8진수 표현
    int hexNum = 0x11;      // 16진수 표현
    int decMun = 11;        // 10진수 표현

    Serial.println(binNum);
    Serial.println(octNum);
    Serial.println(hexNum);
    Serial.println(decMun);
}

void loop(){}
```

[결과]

3
9
17
11

D. 2의 보수

음수를 표현하기 위해서 우리는 숫자의 앞에 음수기호(minus sign)인 '-'를 붙인다. 아두이노와 같은 MCU 는 음수를 표현하기 위해 2의 보수라는 개념을 사용한다.

이번에는 byte 와 같은 1개의 바이트를 사용하는 char 형 변수를 만들어 사용해 보도록 하겠다. byte 는 부호없는 변수이므로 음수여부를 고려하지 않고 모두 양수 혹은 0인 경우만 생각하면 되었다. 하지만 char 변수는 -1, -2 같은 값도 저장할 수 있어야 한다. 이를 위해서 재미있는 실험을 해보도록 하자.

1 과 더해서 0 이 되는 수는 무엇인가? -1 이다. 즉, 이진수에서 1 과 더해서 0 이 되는 수를 찾으면 된다. 앞에서 byte 변수에 최대값 255 를 넣은 다음 1을 더했을때 0이 된 것을 기억해보라. 그러면 다음의 식이 성립한다.

$$\begin{array}{r} & & 1 \\ + & 1111 & 1111 \\ \hline & 1 & 0000 & 0000 \end{array}$$

그림 70 오버플로우

즉, 1 에 1111 1111 을 더했더니 0이 되었다. 1111 1111 은 -1 과 같다. 이것이 2의 보수다. C언어에서 음수는 연산 효율성을 위해 2의 보수형태로 나타낸다. 이는 다른 모든 프로그래밍언어에서 동일한다.

음수를 표현하는 2의 보수를 만드는 방법은 다음과 같다.

첫째. 음수를 표현하고자 하는 이진수의 0과 1을 바꿔준다.

둘째, 그 결과에 1을 더한다.

2의 보수를 사용해서 -9 를 표현해보도록 하자. 우선 십진수 9는 0000 1001 이다. 0 과 1 을 바꾸면 0000 1001 은 1111 0110 이 된다. 여기에 1 을 더하면 1111 0110 은 1111 0111 이 된다.

1111 0111 이 -9를 표현한 2의 보수다.

이를 표로 나타내보면 다음과 같다.

char 변수를 사용한 양수와 음수의 표현

양수	이진수	음수	이진수
0	0000 0000		
1	0000 0001	-1	1111 1111
2	0000 0010	-2	1111 1110
3	0000 0011	-3	1111 1101
...
126	0111 1110	-126	1000 0010
127	0111 1111	-127	1000 0001
		-128	1000 0000

그림 71 양수의 음수 표현

변수에 저장할 수 있는 수보다 더 큰 수를 넣으면 결과 값이 담을 수 있는 그릇보다 커져서 그릇을 넘치게 한다. 이를 오버플로우라고 한다. int 형 변수는 최대 32767 까지 담을 수 있다. 여기에 1이 추가되면 어떻게 될까? 오버플로우가 발생하고, 결과값은 우리가 예상하지 못하는 값이 나오고 만다.

```

void setup()
{
    Serial.begin(9600);
    int intMax = 32767;
    Serial.println(intMax);
    intMax++;
    Serial.println(intMax);
}
void loop(){}

```

[결과]

32767

-32768

16 비트 변수인 intMax 에 32767 을 넣으면 이진수로 0111 1111 1111 1111 이 된다. 여기에 1 이 더해지면 1000 0000 0000 0000 이 된다. 첫 번째 비트가 1인 것은 음수라는 것을 뜻한다. 2의 보수로 계산해보면 -32768 이 된다.

아두이노를 포함해서 C로 프로그래밍할 때 자주 발생하는 실수 중의 하나가 변수의 값을 증가하다가 오버플로우를 발생시키는 경우다. 오버플로우가 발생하면 갑자기 최고값이 음의 최저값으로 바뀐다.

4.7. 실수형

8비트 MCU 인 아두이노에서의 실수형은 float 하나만 있다. double 형도 있기는 하지만 float 과 동일하다. float 변수는 $-3.4028235E+38$ 부터 $3.4028235E+38$ 까지의 실수 숫자를 저장하고 4바이트를 사용합니다.

실수는 일상 사용하듯이 3.14, 0.01, -4.16 등으로 표현할 수 있고, 4.16E16 처럼 지수표기식을 사용할 수도 있다. 4.16E16 는 4.16×10^{16} 이다.

실수자료형	비트수	표현값의 범위	비고
float	32-bit (4-byte)	7개 유효숫자로 $\pm 3.4E+38$ 사이	8 bit MCU, 16 MCU
double	32-bit (4-byte)	7개 유효숫자로 $\pm 3.4E+38$ 사이	8 bit MCU (Uno, Mega)
double *	64-bit (8-byte)	15개 유효숫자로 $\pm 1.79E+308$ 사이	16 bit MCU (Due) 에 해당

그림 72 실수 자료형의 범위

아두이노에서 실수 연산은 속도를 느리게 만든다. 그리고 실수연산은 정확도가 떨어진다. 가능하면 float 변수를 사용하지 않는 것이 좋다. float 변수는 총 32비트 중 부호 1비트, 지수부 8비트, 가수부 23비트를 사용한다. 그래서 십진수로 따지면 7자리숫자가 유효숫자가 된다. float 변수는 $-3.4028235 \times 10^{38} \sim 3.4028235 \times 10^{38}$ 사이의 숫자를 표현할 수 있다. 32개 비트는 다음과 같은 위치에 저장됩니다.

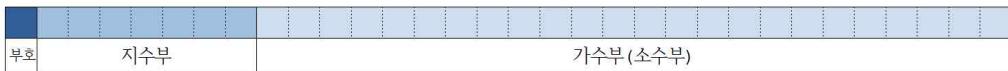


그림 73 부동소수점 표기

C 는 이진수를 사용하고, 그것은 실수 표현에도 동일하게 적용된다. float 형으로 실수를 저장할 때 10진수를 이진수로 변환하고 그것을 다시 가수부와 지수부를 나눠서 표기하게 된다. 이 과정에서 필연적으로 값의 손실이 발생한다. 이를테면 0.1 을 저장하려고 0.1 을 이진수로 변환하면 0.001100110011001100 ... 이 된다. 0011 이 무한 반복되는 것이다.

여기서 유효숫자 7개까지만 저장되고 나머지는 버려지게 되므로 십진수 0.1 은 이진수로 정확히 변환되지 않고 근사치로 변환된다. 이것을 10번 반복해서 더하면 1 이 되지 않고 1에 근사한 값이 된다. 이런 이유로 반복문, 조건문, 조건연산자 등을 사용할 때 실수형 변수의 사용은 조심해야 한다.

4.8. 문자형

문자는 C의 기본 자료형이다. ‘a’, ‘b’, ‘?’ 같은 문자 리터럴(character literal)은 단일 인용부호를 사용하여 표현한다. C는 문자표현으로 ASCII

코드를 주로 사용한다. 128개의 문자를 7비트를 사용하여 표현된 초기의 아스키코드는 8비트로 확장되어 서유럽에서 사용되는 알파벳을 반영하였으며, 다음의 문자들을 표현할 수 있다.

- 영문 대문자
- 영문 소문자
- 숫자
- 공백문자
- 구두점 (콜론, 세미콜론 등)
- 특수문자(&, |, \ 등)
- 제어문자(개행문자, 널문자 등)
- 악센트 문자(Extended ASCII Character)

ASCII (아스키 제어 문자) 테이블에 포함된 값은 다음과 같다.

ASCII TABLE											
Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	'
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	S	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	,	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

A. 이스케이프 시퀀스

\(백슬래시) 뒤에 문자나 숫자가 오는 조합을 이스케이프 시퀀스라고 부른다. 제어문자를 표현할 때 이스케이프 시퀀스를 사용한다. 아래는 자주 사용되는 이스케이프 시퀀스를 표로 나타낸 것이다.

이스케이프 시퀀스	설명
\b	백스페이스
\n	개행(Line Feed)
\r	복귀(Carriage Return)
\t	탭
\w	백슬래시
\'	작은따옴표
\"	큰따옴표

그림 75 이스케이프 시퀀스

문자형은 정수처럼 다루어진다. 문자는 문자로 화면에 출력할 수도 있고, 정수처럼 코드값을 화면에 보이게 할 수도 있다. 문자를 정수처럼 int 형 변수에 저장하고 출력하는 것도 가능하다.

문자를 다루는 방식은 PC 와 아두이노에서 상당히 달라진다. PC 는 기본적으로 모니터와 키보드가 있기 때문에 이 둘을 표준입출력장치로 사용하게 된다. PC 에서 프로그래밍을 할 때는 기본적으로 OS 를 활용하기 때문에 키보드 입력이나 모니터 출력을 위한 프로그래밍을 할 필요는 없다. 하지만 아두이노는 키보드도 모니터도 없기 때문에 하드웨어에 관한 프로그래밍을 직접 해야만 한다.

우선 PC 에서 하나의 문자를 변수에 담아 출력하는 프로그램코드는 다음과 같다.

```
#include <stdio.h>

int main(void)
{
    char myAlphabet = 'a';
    printf("%c %d\n", myAlphabet, myAlphabet);
    return 0;
}
```

PC 기반으로 C 언어를 배울때 처음 다루는 것이 printf() 함수다. printf() 함수는 모니터(정확하게는 표준출력인 stdout)에 결과를 출력시켜주는 기능을 한다.

printf() 함수 호출은 다음의 형식으로 한다.

```
printf("제어할 문자열", 표현식1, 표현식2, ... );
```

$2 + 3$ 의 결과를 표현할 때 다음과 같이 한다.

```
printf("%d + %d = %d\n", 2, 3, 2+3);
```

위 식의 결과는

$2 + 3 = 5$

가 된다.

제어할 문자열 안에는 서식지정자(format specifier)가 포함될 수 있고, 그 형식코드는 정수, 실수, 문자 등에 따라 다르게 사용된다.

4.9. 문제

1. 간단한 시를 출력하는 프로그램을 작성해 보세요
2. RAM 메모리를 최소로 사용하도록 위 코드를 수정해 보세요.

Chapter 05

주요 학습 목표

- 문자열에 대한 이해
- 아두이노 시리얼 통신 이해

Chapter 05. 문자열과 입출력 함수

C언어에서 문자열을 처리하는 것은 쉽지 않다. 실제로 C언어 이후에 나오는 프로그래밍 언어들은 문자열 처리에 탁월한 효과를 가진 것이 많다. 반면 C는 문자열처리에 대해서는 최근의 언어들에 비해서는 상당히 까다롭고, 번거롭다. 게다가 아두이노와 같은 MCU 는 PC 의 표준입출력 함수를 사용할 수 없기 때문에 문자열 처리에 더 신경을 써야 한다.

그 이유는 C 언어는 상대적으로 낮은 하드웨어 사양에서 프로그램이 가능한 언어이기 때문이고, 그런 점에서 조금 불편한 디버깅 방법으로 시리얼모니터를 아두이노는 사용하고 있다. 시리얼모니터를 통해 아두이노에서 오는 신호를 PC에서 받아 화면에 보여주게 된다. 이 방식을 통해 디버깅이 이루어지기 때문에 문자열 관련 함수는 아두이노와 같은 MCU 를 다룰 때 꼭 필요한 부분이기도 하다.

이번 장에서는 문자열 처리가 쉽지 않다는 것을 염두에 두고 아두이노 C의 문자열 처리를 살펴보겠다. 또한 PC 에서의 문자열 처리에 대해서도 짧게 살펴본다. PC 환경과 아두이노의 환경의 차이점을 알고 그에 따라 다른 방식으로 입출력을 고려해야 한다.

고전적인 C언어의 문자열 처리는 char 변수를 배열로 사용하고 그것을 포인터로 처리한다. 포인터에 대한 세부 내용은 별도의 챕터에서 다룬다. 포

인터는 변수가 담겨 있는 메모리의 주소를 가리키는 것을 의미한다. 문자열은 0개 이상의 문자들이 모여있는 열(row)을 인증인용부호로 감싸서 나타낸다. 문자 하나는 8비트의 char 또는 byte 타입의 변수에 저장될 수 있다. 하나의 문자를 이중 인용부호로 표시하면 문자열이 되고, 단일 인용부호로 표시하면 문자가 된다. 즉, 다음에서 "A" 는 문자열이고, 'A' 는 문자다.

"Hello World"

"A"

'A'

C는 문자열을 위한 자료형을 지원하지 않는다. 문자열은 문자 배열 형태로 저장된다. 문자열의 개별 문자의 ASCII 코드 값으로 저장되고, 문자열의 마지막 원소는 NULL 문자인 '\0'이 들어간다.

"Arduino" 문자열은 다음과 같이 저장된다.

A	r	d	u	i	n	o	\0
---	---	---	---	---	---	---	----

그림 76 문자열저장

마지막 '\0' 은 역슬래시와 숫자 0이 들어간다는 의미가 아니라 NULL 문자인 ASCII 코드 테이블에 있는 0 번 기호를 의미한다. NULL 은 화면에 표시되지 않고, 특별한 의미로만 사용된다. NULL 은 0을 의미하지 않는다. "없다"를 의미한다. 메모리 주소를 표시할 때 NULL 은 해당하는 메모리가 없다는 것을 의미하고, 문자열안에 있는 NULL 은 더 이상 문자열이 없다는 뜻이 된다. 즉, 문자열을 읽다가 NULL 을 만나는 순간 더 이상 읽기를

중지하고 문자열의 끝을 표시할 때 사용되는 특수 부호다.

메모리를 사용할 때 다른 값으로 초기화하지 않으면 기본적으로 NULL로 초기화된다. 즉, “Arduino”를 입력하기 전 초기화가 진행되었으면 메모리는 NULL로 가득채워진다. NULL의 Ascii Code는 0번이므로, Hex에 디터로 메모리의 모습을 본다면 0으로 채워진 것을 볼 수 있다.

Symbol	\0	\0	\0	\0	\0	\0	\0	\0
ASCII(Hex)	00	00	00	00	00	00	00	00

그림 77 비어있는 메모리 내부

문자열 “Arduino”를 메모리에 넣으면 다음과 같이 메모리의 공간이 채워진다. 테이블은 각 문자에 해당되는 아스키코드로 채워지고, NULL 문자가 들어간 부분은 0으로 채워진다.

Symbol	A	r	d	u	i	n	o	\0
ASCII(Hex)	41	72	64	75	69	6E	6F	00

그림 78 문자가 들어간 메모리 내부

C에서는 문자열을 위한 자료형이 없지만 C++에서는 별도의 자료형을 만들어서 사용할 수 있다. 아두이노는 String 자료형을 사용할 수 있고, 최근 나오는 대부분의 MCU에 사용되는 C는 C++의 연장선에서 String 자료형을 대부분 적용받고 있다.

5.1. 입출력장치와 시리얼 모니터

대부분의 C 책은 문자열을 책의 후반부에 둔다. 문자열을 다루기 위해서는 배열과 포인터를 알아야 하기 때문에 이와 같은 구성이 일반적이다. 하지만 PC 와 달리 아두이노는 표준입출력을 지원하지 않는다. 키보드와 모니터를 통한 입력과 출력이 기본적으로 제공되는 PC의 OS 와 달리 아두이노는 하드웨어에 맞는 입출력이 고려되어야 한다. 아두이노는 이를 위해 시리얼 모니터를 사용한다. 대부분의 MCU 에 있는 시리얼통신을 이용해 PC와 통신 할 수 있게 한 후 이를 이용해 간단한 문자열을 주고 받을 수 있도록 했다.

이것이 문자열과 시리얼 통신을 이용한 입출력을 책의 초반부에 넣은 이유다. PC 에서 간단한 프로그램을 진행하며 모니터를 통해 결과를 쉽게 확인하는 것처럼 아두이노에서도 시리얼통신을 통해 원하는 결과가 나오는지 확인할 수 있다. 이를 위해 필요한 설정과 함수들을 살펴보도록 하자.

A. 시리얼통신의 특징

MCU 를 사용하는 대부분의 시스템은 주변 장치들과 신호를 주고 받기 위한 통신방식을 가지고 있다. MCU 는 병렬(Parallel)방식과 직렬(Serial)방식으로 데이터를 주고 받을수 있다. 이중 시리얼통신은 직렬방식의 통신으로 하나의 신호선을 통하여 신호를 주고 받는다. 시리얼통신은 병렬통신방식과 비교해 적은 수의 신호선으로 데이터를 주고 받기 때문에 회로가 간결해지는 이점이 있다.

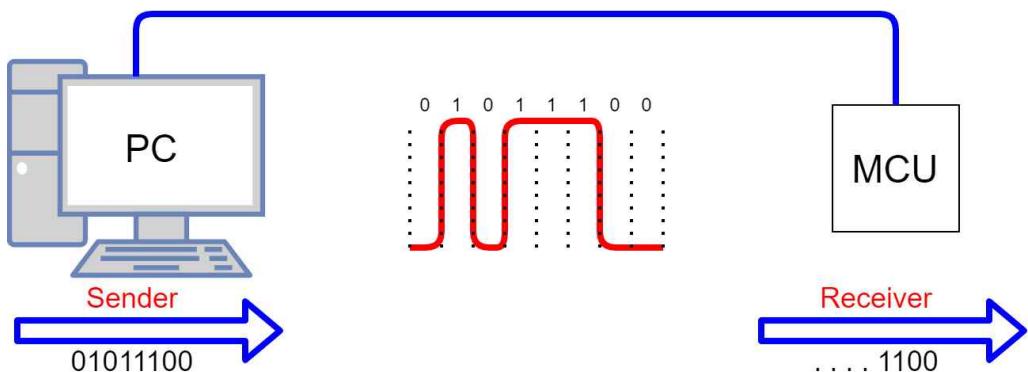


그림 79 직렬(시리얼)통신

이런 직렬 통신 방식 중 보편적으로 많이 사용되는 것은 RS-232C 와 RS-485 이다. 이중 아두이노가 기본 지원하는 직렬 통신은 RS-232C 방식이다. 산업용으로 많이 사용되는 RS-485 는 다자간 통신이 가능한 반면 하드웨어적인 종단저항이 사용된다. RS-232C 는 상대적으로 간단한 통신 방식이지만 1:1 통신만 지원된다.

PC 와 아두이노는 시리얼통신을 위해 Rx, Tx 선을 사용한다. Tx 선으로 신호를 보내고 Rx 선으로 신호를 받는다. 즉, 두 기기를 시리얼통신을 위해 연결할 때는 Rx 와 Tx 를 아래 그림처럼 연결 한다. 한 기기의 Rx 는 다른 기기의 Tx 와 연결된다.

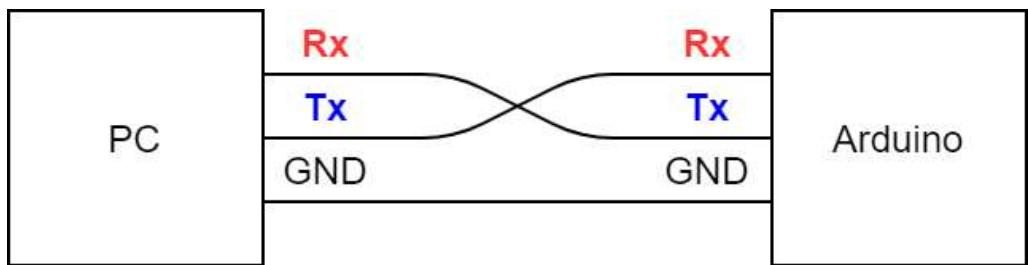


그림 80 RS232 통신을 위한 아두이노와 PC 핀 연결

B. 보드레이트와 BPS

하나의 전송라인을 통해 데이터를 주고 다른 하나의 전송라인으로 신호를 받기 때문에 두 기기 사이에 일정한 약속이 필요하다. 그중 모뎀의 속도를 bps(bits per second)로 표시하는데, 이것은 1초당 전송할 수 있는 비트의 수를 의미한다. BPS 와 혼용해서 사용되는 보드레이트(baud rate)가 있다. 보드레이트는 1초 동안 전송되는 신호의 수를 나타낸다. 즉, 아두이노와 같은 디지털 시스템에서 bps 와 보드레이트는 같은 의미를 가진다. 아두이노 IDE 를 실행한 후 시리얼모니터창을 열면 오른쪽 아래에 보드레이트를 선택할 수 있다.

아두이노와 PC간 시리얼통신을 위해 설정해야 할 것은 아두이노의 보드레이트값과 PC의 보드레이트값을 동일하게 맞춰주어야 한다.



그림 81 Arduino baurate 설정

5.2. 문자열 처리 방법과 String

A. string

문자열은 문자들의 모음 후 끝에 NULL 문자를 조합시키는 것으로 만들어진다. C 는 문자열을 위해 char 변수를 배열의 형태로 사용한다. 아래는 String 자료형을 사용하지 않고 문자열 "Arduino"를 저장하는 전통적인 방법이다.

```
char str1[10];
char str2[8] = {'A', 'r', 'd', 'u', 'i', 'n', 'o'};
```

```
char str3[8] = {'A', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};  
char str4[] = "Arduino";  
char str5[8] = "Arduino";  
char str6[10] = "Arduino";
```

str1 처럼 초기화 없이 문자 배열로 선언할 수 있다. 이후 str1 배열에 원하는 내용을 추가한다.

str2 는 8개의 저장공간을 확보한 후 각각의 공간에 필요한 문자를 넣는 방식이다. 문자는 Arduino 로 7개가 되고, 마지막 8번째 공간에는 '\0' 인 NULL 문자가 들어간다. 즉, 7개의 문자를 저장하는 문자열은 1을 더한 8 개의 공간이 필요하다.

str3 처럼 NULL 문자까지 포함해서 문자열을 만들 수 있다. str2 와 완벽하게 동일한 결과가 나온다.

str4 처럼 배열의 크기를 지정하지 않고 문자열 배열을 만들 수 있다. 이 경우 문자열이 할당되면서 자동으로 배열의 크기가 결정된다. 일반적으로 많이 사용되는 방식이다.

str5 는 배열의 크기를 정하고 문자를 할당한다. 이 경우 문자의 갯수보다 1 크게 만들어주어야만 한다. 그렇지 않으면 NULL 문자를 만나지 못해서 심각한 오류가 발생할 수 있다.

str6 는 문자의 크기보다 넉넉하게 여분의 공간을 주는 경우다. 이 경우 아무런 문제없이 초기화와 문자열 할당이 이루어진다. 다만 지나치게 많은 여유공간을 주면 메모리 부족현상이 나올 수 있다.

한줄에 표시하기 어려운 긴 문자열은 다음과 같이 쓸 수 있다.

```
char myString[] = "Lorem ipsum dolor sit amet,"  
                  "consectetur adipiscing elit, "  
                  "sed do eiusmod tempor incididunt";
```

여러줄에 걸치더라도 문장의 끝의 의미하는 세미콜론(;)을 만나기 전까지는 끝나지 않는다. 즉, 위에서는 세줄에 걸쳐 각각 이중 인용부호로 묶인 문자열이 합쳐진 하나의 문자열이 되어 저장된다.

실행을 위한 전체 코드는 다음과 같다.

```
void setup()  
{  
    char myString[] = "Lorem ipsum dolor sit amet,"  
                      "consectetur adipiscing elit, "  
                      "sed do eiusmod tempor incididunt";  
    Serial.begin(9600);  
    Serial.println(myString);  
}  
void loop() {}
```

결과는 다음과 같다.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
 tempor incididunt

문자열 하나가 아닌 여러 문자열을 사용하는 경우가 많다. 이 틀테면 LCD
 디스플레이등을 사용할 때 정해진 문자열이 순서대로 나오게 하는 경우에

문자열의 배열을 사용할 수 있다. 배열은 변수를 여러개 모아서 사용하는 방법으로 이후에 다루게 된다. 배열은 변수들을 모아 번호를 붙이고, 번호로 해당되는 내용을 부르는 방식이다.

다음은 char 자료형 뒤에 * 를 사용한 포인터이다. 포인터와 배열은 이후에 별도의 장에서 다루게 된다. 여기서는 간단한 사용방법만 익히도록 하자.

문자열을 원소로 가지는 배열은 다음과 같이 만들 수 있다.

```
char *문자열배열이름[] =  
{  
    "문자열 1",  
    "문자열 2",  
    "문자열 3"  
};
```

사용할 때는 앞에서부터 순서대로 0 부터 1 씩 증가하며 인덱스를 붙여 사용한다.

문자열배열이름[0] 은 "문자열 1" 을 의미한다.

문자열배열이름[1] 은 "문자열 2" 을 의미한다.

문자열배열이름[2] 은 "문자열 3" 을 의미한다.

즉, Serial.print(문자열배열이름[0]) 과 같이 문자열을 출력할 수 있다. 아래는 6개의 문자열을 배열에 담아 한줄씩 차례대로 출력하는 예제이다.

```

char *multiString[] =
{
    "This is string 1",
    "This is string 2",
    "This is string 3",
    "This is string 4",
    "This is string 5",
    "This is string 6"
};

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    for (int i = 0; i < 6; i++)
    {
        Serial.println(multiString[i]);
        delay(500);
    }
}

```

B. String

String 은 C가 아닌 C++ 클래스에 속한다. String 을 사용하면 상당히 간편하게 문자열을 처리할 수 있다. String() 을 이용하면 수와 문자, 문자열을 String 객체로 변환해서 사용될 수 있다. 단, 위에 소개한 전통적인 배

열과 포인터를 이용하는 방식에 비해 메모리 사용이 많아지므로 지나치게 많이 사용하는 것은 주의해야 하고, UNO 처럼 기본 메모리가 작은 경우 더 많은 신경을 써야 한다.

String(value, base|number)

value : String 객체로 바꾸려는 대상(수, 변수, 문자(열) 등)
base : 2진수(BIN), 8진수(OCT), 10진수(DEC), 16진수(HEX)
number : 소수점 자리수

String() 클래스는 value 를 문자열로 변환시켜준다. value 가 정수일 때 base 에 따라서 변환이 이루어진다. value 가 소수점이 포함된 실수라면 number 에 따라서 정해진 자리수까지 문자열로 변환한다.

아래에 몇 가지 변환 식을 통해 string() 의 사용법을 익혀본다.

```
String str1 = "Hello Arduino";
```

우선 str1 변수를 String 타입으로 만든다. 그 다음 str1 에 "Hello Arduino" 문자열을 할당한다. 이는 다음과 같이 사용할 수도 있다.

```
String str2 = String("Hello Arduino");
```

또한 다음처럼 문자 하나 혹은 숫자를 String() 을 이용해 변환할 수 있다.

```
String str3 = String('A');
```

```
String str4 = String(2020);
```

기존의 문자열에 새로운 문자열을 추가해서 새로운 문자열을 만들 때 다음과 같이 사용할 수 있다.

```
String str5 = String("Arduino");
String str6 = String("Hello " + str1);
```

다음처럼 정수를 10진수, 16진수, 2진수로 변환하여 문자열로 저장할 수 있다.

```
String str7 = String(46, DEC);
String str8 = String(46, HEX);
String str9 = String(46, BIN);
```

실수의 소수점 자리수를 포함하여 다음과 같이 문자열로 저장할 수 있다.

```
String str10 = String(1.23456789, 3);
```

str1 을 출력하면 1.235 가 출력된다.

위의 내용을 실행하면 다음과 같다.

```
void setup()
{
    Serial.begin(9600);
```

```
String str1 = "Hello Arduino";
String str2 = String("Hello Arduino");
String str3 = String('A');
String str4 = String(2020);
String str5 = String("Arduino");
String str6 = String("Hello " + str1);
String str7 = String(46, DEC);
String str8 = String(46, HEX);
String str9 = String(46, BIN);
String str10 = String(1.23456789, 3);

Serial.println(str1);
Serial.println(str2);
Serial.println(str3);
Serial.println(str4);
Serial.println(str5);
Serial.println(str6);
Serial.println(str7);
Serial.println(str8);
Serial.println(str9);
Serial.println(str10);
}

void loop()
{ }
```

실행결과

```
Hello Arduino
Hello Arduino
A
```

```
2020
Arduino
Hello Hello Arduino
46
2e
101110
1.235
```

5.3. PC 와 표준입출력

PC에서 C를 사용하는 경우 표준입출력을 이용할 때 일반적으로 printf()와 scanf()를 사용한다. 물론 이 외에도 문자열을 다루는 함수들은 많이 있지만 여기서는 printf()와 scanf()에 한정해서 사용법을 익혀보도록 한다. 4장에서 간단한 언급했던 printf() 함수는 다음과 같은 특징을 가진다.

A. printf()

PC에서 사용되는 printf()는 인수로 주어진 데이터의 형식에 맞춰 문자열로 변환시킨 후 화면에 출력한다. 이 인수를 포맷 스트링(format string)이라고 한다.

```
*****
* standard_output.c
*
* printf 함수의 기초적인 사용법
*****
```



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

```
int main(void)
{
    int number = 2;
    float pi = 3.14;
    char name = 's';

    printf("number = %d\n", number);
    printf("pi = %f\n", pi);
    printf("name = %c\n", name);

    return 0;
}
```

[실행결과]

```
number = 2
pi = 3.140000
name = s
```

위의 예제에 사용된 printf()의 포맷 스트링을 정리하면 다음과 같다.

포맷스트링	내용
%d, %i	십진수 정수 출력, 값이 음수이면 앞에 - 가 붙음
%o	부호없는 8진 정수 출력
%x, %X	부호없는 16진 정수 출력
%u	부호없는 10진 정수 출력
%c	문자 출력
%s	문자열 출력
%f	고정 소수점 표기로 실수 출력
%e, %E	지수 표기로 실수 출력
%g, %G	%f, %e 중 표현이 더 짧은 것을 선택하여 출력
%p	포인터 형태로 출력
%%	% 부호를 출력

그림 82 printf() 포맷
자주 사용되는 것은 %d, %f, %c 이다. C99 표준에서는 %lf 가 추가되었고, 최신 C컴파일러는 double 타입의 실수를 다룰때 %lf 를 사용하는 것이 표준이 되었다. 기존의 방식대로 %f 를 사용할 수도 있다.

printf() 는 문자열로 변환한 후 출력이 이루어진다. 이렇게 문자열로 변환될 때 문자열 폭을 지정할 수 있다. 지정하고자 하는 폭은 포맷 스트링의 %뒤 숫자로 표시한다. 십진수를 나타낼 때 문자열 폭을 일정 간격으로 지정하여 출력시 가독성을 높일 수 있다.

아래는 2020년 4월 16일의 연도, 월, 일을 4개의 문자폭을 주어 화면에 출력한 예제다. 출력폭을 명시하면 가독성이 좋아진다.

```
*****
* print_width.c
*
```

```

* printf 출력 폭 지정
***** */

#include <stdio.h>

int main(void)
{
    int y = 2020;
    int m = 4;
    int d = 16;

    printf("y = %4d\n", y);
    printf("m = %4d\n", m);
    printf("d = %4d\n", d);

    return 0;
}

```

[실행 결과]

```

y = 2020
m =     4
d =     16

```

실수 출력시 경우 소수점 이하 정밀도를 설정할 수 있다. 폭의 넓이를 나타내는 숫자 뒤에 .(dot)을 붙인 후 그 뒤에 소수점 이하 자리수를 표시한다.

`%10.2f`

`%10.2f` 는 실수를 나타내면서 전체 자리수는 10으로, 소수점이하는 2자리 까지 표시한다.

```

*****
* print_width.c
*
* printf 출력폭 지정
*****
#include <stdio.h>

int main(void)
{
    float pi = 3.1415926535897932;

    printf("pi = %10.2f\n", pi);
    printf("pi = %10.4f\n", pi);
    printf("pi = %10.6f\n", pi);

    return 0;
}

```

[실행결과]

```

pi =      3.14
pi =      3.1416
pi =    3.141593

```

B. scanf()

scanf() 는 키보드로 입력한 문자열을 전달받는 함수다. 키보드는 표준입력의 대표격이지만 키보드외에 다른 방식으로 문자열을 전달받을 수 있다. 예

기서는 키보드 입력에 한해서만 다루기로 한다.

scanf() 는 입력을 다루는 함수로 출력을 다루는 printf() 의 반대편에 있다. scanf() 로 정수를 입력받아 입력된 정수의 제곱값을 출력하는 코드는 다음과 같다.

```
*****
* scanf_square.c
*
* printf 출력 폭 지정
*****
```

```
#include <stdio.h>

int main(void)
{
    int num;

    scanf("%d", &num);
    printf("input number      = %d\n", num);
    printf("input number*number = %d\n", num*num);

    return 0;
}
```

scanf() 를 실행시키면 커서가 멈춰서 키보드 입력을 기다린다. 이때 정수를 입력하고 엔터키를 치면 그 값이 변수 i에 입력된다. 키보드로 25 를 치고 엔터키를 누르면 다음과 같다.

[실행결과]

25[Enter]

```
input number      = 25
input number*number = 625
```

포맷스트링	내용
%d	십진수 정수
%o	부호없는 8진 정수
%x	부호없는 16진 정수
%u	부호없는 10진 정수
%c	문자
%s	공백이 포함되지 않은 문자열
%f, %e, %g	고정 소수점, 부동 소수점 실수
%p	포인터 형태로 출력

그림 83 scanf() 포맷

scanf()의 포맷 스트링은 printf()와 유사하지만 사용방법은 다르다.

위에서 scanf("%d", &num); 는 첫번째로 입력 받은 값은 문자열로 인식된다. 그 다음 문자열을 포맷 스트링 "%d"에 따라서 정수로 변환한다. 변환된 정수값은 num 변수에 저장되는데 이를 위해 변수 num의 주소값 (&num)이 필요하다.

"scanf("%d", &num);;" 을 정리하면 다음과 같다.

1. 키보드로 입력을 받는다 – scanf()

2. 입력받은 문자열을 정수로 변환한다 – %d
3. 변환된 정수를 변수 num에 저장한다 – &num

5.4. MCU 시리얼통신을 이용한 입출력

아두이노는 키보드와 모니터가 없기 때문에 표준입출력 함수를 사용하는 것이 큰 의미가 없고, printf()나 scanf() 대신 시리얼통신을 이용한 몇 가지 함수를 제공하고 있다. 이렇게 시리얼통신을 이용한 입출력 함수는 아래에 분류되어 있다. 모든 함수는 "Serial."을 붙여 사용할 수 있다. Serial이라는 클래스에 속한 함수는 다음과 같다.

시작과 종료 : begin(), end()
상태 : available()
입력 : read()
출력 : print(), println(), write()

A. 시작과 종료

시리얼 통신 시작을 위해서 begin() 함수를 사용한다.

```
Serial.begin(9600);
```

9600 속도로 시리얼통신을 시작하라는 명령을 아두이노가 실행한다. 이 명령 후에 아두이노의 Tx, Rx 핀은 시리얼통신을 위해 신호를 주고 받는 용도로 사용된다. 이 책에서는 기본 연결속도는 9600을 사용한다. 선의 상태나 주변 환경이 전자파 노이즈가 많은 경우 속도를 낮추면 오작동하지 않는다. 통상적으로 시리얼통신 속도는 9600에서 57600 사이의 값이 사

용된다.

아두이노 우노는 시리얼통신을 위한 Rx, Tx 핀이 각각 0번과 1번이다. 0번과 1번핀은 PCB안에서 Atmega 16u2 에 연결되어 USB 케이블을 통해 PC로 연결된다.



그림 84 아두이노 우노의 Rx, Tx

같은 방식으로 아두이노 메가2560도 0번과 1번핀이 Rx, Tx 로 PC 와 연결된다. 우노는 하나의 Rx, Tx 만 존재하지만 메가2560은 (Rx, Tx), (Rx1, Tx1), (Rx2, Tx2), (Rx3, Tx3) 로 총 4개의 시리얼포트를 사용할 수 있다. 각각 Serial, Serial1, Serial2, Serial3 클래스로 이루어져 있다. (Rx, Tx) 핀은 PC 와의 통신을 위해 사용되고, 나머지 3개의 시리얼 포트를 이용해 다른 기기와 1:1 로 통신할 수 있다.

메가 2560에서 4개의 시리얼포트로 통신을 시작하는 명령은 다음과 같다.

```
Serial.begin(9600);  
Serial1.begin(9600);  
Serial2.begin(9600);  
Serial3.begin(9600);
```

Serial1, Serial2, Serial3가 사용하는 함수는 Serial과 동일하다.

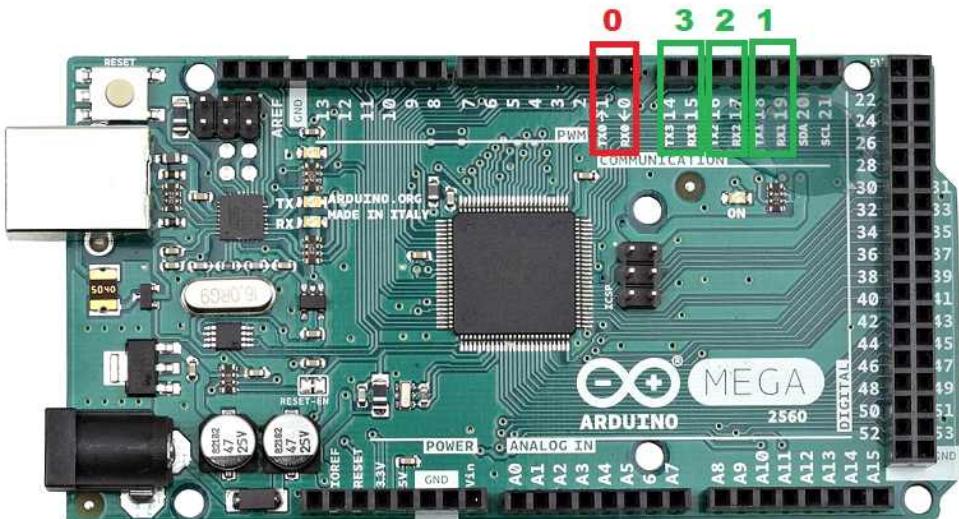


그림 85 메가2560의 시리얼통신 핀

시리얼 통신 종료시에는 end() 함수를 사용한다.

```
Serial.end();
```

보통 end() 함수는 잘 사용되지 않는다. 시리얼 통신을 통해 계속적으로

모니터링 하는 것이 일반적이고, 특별한 경우에만 시리얼 통신을 지원하는 경우에 Serial.end() 를 사용한다.

```
/*********************  
* serial_end.c  
*  
* end() 함수 사용, 시리얼 종료  
******/  
  
void setup()  
{  
    Serial.begin(9600);  
    Serial.println("아두이노 시리얼통신 작동");  
    Serial.println(".....");  
    Serial.println("시리얼 종료 - 0번,1번핀 사용가능");  
    Serial.end();  
}  
  
void loop() {}
```

위 코드를 실행하고 나면 시리얼 통신을 이용해 메시지를 보낸 후 시리얼 통신에 사용되었던 0번핀과 1번핀인 Rx, Tx 핀이 해제 되어 일반적인 입출력 용도로 재사용 가능해진다. end() 함수는 더 이상 시리얼 통신을 하지 않을때 사용한다.

[실행결과]

```
아두이노 시리얼통신 작동  
.....  
시리얼 종료 - 0번,1번핀 사용가능
```

B. 상태

available() 함수는 버퍼에 수신된 데이터의 바이트(문자) 수를 가져온다. 시리얼 통신을 위해 수신쪽 버퍼는 64바이트 크기를 가지고 있으며 아두이노로 들어온 신호는 수신버퍼에 쌓이게 된다. 수신 버퍼는 바이트 단위로 읽은 후 비워진다. 일반적으로 available() 함수는 수신버퍼에 들어온 데이터가 있는지 여부를 확인할 때 사용된다.

```
...
if (Serial.available() > 0) {
    // 시리얼 수신 버퍼에 신호가 들어오면 할 일:
    // ...
}
```

수신 버퍼에 데이터가 있다면 데이터를 읽기위해 read() 함수를 사용한다.

C. 입력

시리얼 입력으로 read() 함수를 사용한다.

시리얼로 두 기기가 연결된 상태에서 상대쪽에서 보내온 데이터가 수신 버퍼에 들어온 후 버퍼의 내용을 바이트 단위로 읽는 명령이다. 더 이상 읽을 데이터가 없으면 -1 을 반환한다. 기본 예제는 다음과 같다.

```
*****
```

```

* serial_read.c
*
* read() 함수 사용, 시리얼 데이터 읽기
***** */

int myByte = 0;      // 시리얼통신으로 전송받은 데이터를 저장할 변수

void setup()
{
    Serial.begin(9600);    // 시리얼포트를 열고 속도를 9600으로 세팅
}

void loop()
{
    if (Serial.available() > 0)
    {
        myByte = Serial.read();
        Serial.print("I received: ");
        Serial.println(myByte);
    }
}

```

Serial.read() 를 사용하기 전에 먼저 수신버퍼에 데이터가 있는지를 확인한다. Serial.available() 을 이용해서 수신버퍼를 확인한 후 Serial.read() 를 사용한다.

D. 출력

출력을 위해 아두이노에서 사용되는 함수는 print(), println(), write() 등이 있다.

`Serial.print()` 는 아스키코드를 이용해서 문자열을 전송한다.
`Serial.println()` 함수와 차이점은 문자열의 마지막 부분에 개행문자를 붙이지 않는 것이다.

`Serial.print()`, `Serial.println()` 함수는 다음과 같은 형식으로 사용된다.

```
Serial.print(value)  
Serial.print(value, format)
```

정수 숫자는 모두 아스키 문자를 사용하여 전송되고, 실수는 기본 소수점 이하 두자리로 아스키 문자열로 전송된다. 실수 전달시 소수점 자리수를 지정할 수 있다. 문자와 문자열은 그대로 전송된다.

```
Serial.println(416)    ---> "416"  
Serial.println(1.23456) ---> "1.23"  
Serial.println('A')   ---> "A"  
Serial.println("Hi Arduino") ---> "Hi Arduino"
```

문자열로 표현할 형식을 지정할 수 있다. 가능한 형식은 BIN, OCT, DEC, HEX 와 실수의 경우 소수의 자릿수를 사용할 수 있다.

```
Serial.println(78, BIN) ---> "1001110"  
Serial.println(78, OCT) ---> "116"  
Serial.println(78, DEC) ---> "78"  
Serial.println(78, HEX) ---> "4E"  
  
Serial.println(1.23456, 0) ---> "1"  
Serial.println(1.23456, 4) ---> "1.2346"  
Serial.println(1.23456, 6) ---> "1.234560"
```

```

void setup()
{
    Serial.begin(9600);

    Serial.print("NO FORMAT");
    Serial.print("\t");
    Serial.print("DEC");
    Serial.print("\t");
    Serial.print("HEX");
    Serial.print("\t");
    Serial.print("OCT");
    Serial.print("\t");
    Serial.print("BIN");
    Serial.println();

    for (int x = 0; x < 32; x++)
    {
        Serial.print(x);          // 출력 x
        Serial.print("\t\t");
        Serial.print(x, DEC);    // print as an ASCII-encoded decimal
        Serial.print("\t");      // prints a tab
        Serial.print(x, HEX);   // print ASCII-encoded hexadecimal
        Serial.print("\t");      // prints a tab
        Serial.print(x, OCT);   // print as an ASCII-encoded octal
        Serial.print("\t");      // prints a tab
        Serial.println(x, BIN); // print as an ASCII-encoded binary
        // then adds the carriage return with "println"
        delay(200);            // delay 200 milliseconds
    }
    Serial.println();           // prints another carriage return
}

```

```

void loop()
{
}

```

[실행결과]

NO	FORMAT	DEC	HEX	OCT	BIN
0		0	0	0	0
1		1	1	1	1
2		2	2	2	10
3		3	3	3	11
4		4	4	4	100
5		5	5	5	101
6		6	6	6	110
7		7	7	7	111
8		8	8	10	1000
9		9	9	11	1001
10		10	A	12	1010
11		11	B	13	1011
12		12	C	14	1100
13		13	D	15	1101
14		14	E	16	1110
15		15	F	17	1111
16		16	10	20	10000
17		17	11	21	10001
18		18	12	22	10010
19		19	13	23	10011
20		20	14	24	10100
21		21	15	25	10101
22		22	16	26	10110
23		23	17	27	10111
24		24	18	30	11000
25		25	19	31	11001
26		26	1A	32	11010
27		27	1B	33	11011
28		28	1C	34	11100
29		29	1D	35	11101
30		30	1E	36	11110
31		31	1F	37	11111

출력시 문자열은 기본 RAM 메모리를 사용하므로 플래시 메모리를 사용하도록 F() 매크로를 사용해서 RAM 대신 플래시 메모리를 사용할 수 있다.

```
Serial.println(F("Hello Arduino"))
```

아스키 문자로 변환하지 않고 데이터를 전송하려면 Serial.write() 를 사용하면 된다.

```
Serial.write(value);           // 단일 바이트  
Serial.write(string);         // 문자열  
Serial.write(buffer, length); // 배열과 바이트 수
```

Serial.write() 함수는 리턴값으로 사용한 바이트 수를 반환한다.

```
void setup()  
{  
    Serial.begin(9600);  
    Serial.write(72); // 72, 'H'  
    // 문자열 "ello" 과 개행문자를 출력하고, 문자열의 길이를 반환  
    int bytesSent = Serial.write("ello\n");  
  
    Serial.println("String Length : "+String(bytesSent));  
  
    char myStr[] = "Hi Arduino";  
    Serial.write(myStr, 2); // "Hi", 버퍼에 있는 2개문자 출력  
}  
  
void loop() {}
```

[실행결과]

```
Hello  
String Length : 5  
Hi
```

5.5. 문자열

String 을 사용한 문자열 처리를 간략히 살펴보자. 시리얼통신 사용시 String 클래스는 유용하게 사용될 수 있다. String 클래스에 대해서는 5.2 절에서 간략히 살펴보았다. 여기서는 그외 유용한 함수를 살펴보도록 한다.

A. 인덱스

String 클래스에 속한 charAt(index) 함수는 정해진 순서의 문자를 반환한다. 다음과 같이 “abcdef” 라는 문자열에 대해서 charAt(index) 는 index 번째 문자를 요청하는 명령이다. index 는 0부터 시작되므로 만약 index 가 2라면 2번째 문자가 출력된다. 단, index 는 0부터 시작된다.

문자열 "abcdefg"						
a	b	c	d	e	f	g
0번째	1번째	2번째	3번째	4번째	5번째	6번째

그림 86 문자열 저장 형태

String.charAt(index)

전통적인 방법으로는 [] 를 사용해서 인덱스 번호를 지정해준다. 즉

String.charAt(index)

String[index]

는 동일한 결과를 나타낸다.

```

void setup()
{
    Serial.begin(9600);

    String myStr = String("abcdefg"); // myStr 객체 생성
    Serial.println(myStr.charAt(2)); // myStr에서 2번째 문자 출력
    Serial.println(myStr[2]);      // myStr에서 2번째 문자 출력
}

void loop() { }

```

[실행결과]

```
c
c
```

indexOf() 함수는 문자열에서 특정 문자나 문자열을 찾아 인덱스를 전달해 준다. 문자열안에 찾고자 하는 문자나 문자열이 없을 경우 -1 을 전달해 준다. "Arduino World"라는 문자열변수 myString에서 'n'이라는 문자의 위치인 인덱스를 찾고자 한다면 myString.indexOf("n") 으로 찾을 수 있다.

```

String myString = "Arduino World";
int idx = myString.indexOf("n")

```

처럼 사용할 수 있다. 위에서 idx 값은 5 가 된다. 만약 찾고자 하는 문자나 문자열이 없다면 -1 을 반환한다.

B. 비교

String 객체에 저장된 문자열과 string 의 비교하는 함수로 같으면 0을, 다

르면 int 타입 숫자를 반환한다. 실제로 비교해서 나오는 값이 0이 아닌 경우에 대해서는 큰 의미를 부여할 수 없다. 0인지 아닌지에 대해서만 확인하면 된다.

compareTo() 대신 == 연산자를 사용할 수 있다. == 연산자를 사용할 경우 양쪽이 같으면 1 을, 다르면 0 을 반환한다.

```
String.compareTo(string)
String == string

void setup()
{
    Serial.begin(9600);
    String myStr = String("42");      // 기본데이터 "42"

    int comp;
    comp = myStr.compareTo("42");     // 비교데이터 "42" -> 0
    Serial.println(comp);             // 출력 0 : 같다
    comp = myStr.compareTo("44");     // 비교데이터 "44" -> 음수
    Serial.println(comp);             // 출력 -2 : 다르다

    bool comp2, comp3;
    comp2 = (myStr == "42");
    comp3 = (myStr == "44");
    Serial.println(comp2);           // 출력 1 : True
    Serial.println(comp3);           // 출력 0 : False
}

void loop()
{
```

```
}
```

[실행결과]

```
0  
-2  
1  
0
```

5.6. 합성, 추출, 교체

A. 문자열 합치기 : +, concat()

두개의 문자열을 하나로 합치는 기능은 concat() 함수를 쓰는 방법과 + 연산자를 사용하는 방법이 있다. + 연산자를 사용하면 두 문자열을 합친 후 새로운 문자열을 만들게 된다. 반면 concat() 을 사용하면 하나의 문자열에 다른 문자열을 덧붙이게 된다. 기존 문자열을 보존할 필요가 있다면 + 연산자를 사용하고, 기존의 문자열을 수정해서 사용하려면 concat() 을 사용한다.

```
myString3 = myString1 + myString2
```

이미 있는 기존의 문자열 2개(myString1 과 myString2)를 이어서 새로운 문자열을 만들어 myString3 에 저장한다.

```
myString1.concat(myString2)
```

myString1 문자열의 마지막 부분에 myString2 문자열을 이어 붙인다. 이

명령후에 myString1 을 출력해보면 이어붙여진 문자열로 수정되어 있는 것을 확인할 수 있다.

```
void setup()
{
    Serial.begin(9600);
    String myString1 = String("Hello ");
    String myString2 = String("Arduino");
    String myString3;

    myString3 = myString1 + myString2;
    Serial.println("myString1 , myString2, myString3 is ...");
    Serial.println("myString1 = " + myString1);
    Serial.println("myString2 = " + myString2);
    Serial.println("myString3 = " + myString3);

    myString1.concat(myString2);
    Serial.println("myString1 , myString2 is ...");
    Serial.println("myString1 = " + myString1);
    Serial.println("myString2 = " + myString2);
}

void loop()
{}
```

[실행결과]

```
myString1 , myString2, myString3 is ...
myString1 = Hello
myString2 = Arduino
myString3 = Hello Arduino
```

```
myString1 , myString2 is ...  
myString1 = Hello Arduino  
myString2 = Arduino
```

B. 문자열 추출 : substring(), length()

substring() 을 사용하면 String 문자열 객체의 시작점과 끝점을 지정하여 문자열을 복사, 새로운 객체를 만들 수 있다. startNum 번째부터 endNum 번째까지 복사할 때 다음과 같이 명령을 써주면 된다. 끝점인 endNum 은 포함되지 않고, 그 앞까지 추출된다. endNum 은 생략할 수 있으면, endNum 을 생략하면 문자열의 끝까지 추출하여 복사한다.

```
String.substring(startNum, endNum)
```

문자열의 크기를 알고자 할 때 length() 를 사용할 수 있다.

"Hello Arduino !" 라는 문자열에서 "Arduino" 만을 추출해서 따로 저장하고 싶다면 6 번째 인덱스에서 시작해서 12번까지 추출하거나 혹은 6번째에서 시작, 끝에서 2번째까지 추출하면 된다.

string	H	e	1	1	o		A	r	d	u	i	n	o		!	\0
앞에서	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
15 - ?	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	

그림 87 문자열 "Hello Arduino !"

다음과 같이 쓸 수 있다.

```
String myString1 = String("Hello Arduino !");  
myString1.substring(6, 13);
```

문자열의 크기(길이)를 알기 위해서 length() 를 사용할 수 있는데, 그 방법은 다음과 같다.

```
int len = myString1.length();
```

"Hello Arduino !"는 공백문자와 '!' 문자까지 포함해서 총 15개의 문자로 구성되어 있고, myString1.length() 는 15를 반환한다. 이것을 사용하여 맨 끝에서 두번째 자리까지 출력하게 할때 myString1.length()-2 를 사용할 수 있다. 즉, 아래 두 명령은 동일하다.

```
myString1.substring(6, 13);  
myString1.substring(6, myString1.length()-2);
```

문자열의 크기가 작으면 앞에서부터 인덱스를 계산해도 큰 무리가 없지만 크기가 조금만 커져도 앞에서부터 인덱스를 세는 것은 오류가 들어올 가능성이 커진다. 이때 뒤에서부터 계산하는 것이 좋다. length() 를 쓰면 문자열 전체의 크기를 쉽게 알수 있으므로 이것을 이용하여 뒷자리부터 몇번 째 자리까지 추출할 것인지를 프로그래밍 할 수 있다.

```
void setup()  
{  
    Serial.begin(9600);  
  
    String myString1 = String("Hello Arduino !");  
    String outString;
```

```

int strLen = myString1.length();
outString = myString1.substring(6, strLen-2);
Serial.println(outString);
}

void loop()
{
}

```

[실행결과]

Arduino

C. 문자열 교체

replace() 를 이용하면 전체 문자열에서 원하는 부분 문자열을 교체할 수 있다.

기존 문자열에서 일부분을 교체할 때 다음과 같이 명령할 수 있다.

```

String.replace(char1, char2)
String.replace(String1, String2)

```

"Hello Arduino !" 에서 "Arduino" 대신 "World" 로 교체할 때 replace("Arduino", "World") 를 사용하면 된다. 앞의 문자열을 모두 뒤의 문자열로 교체한다.

```

void setup()
{

```

```

Serial.begin(9600);

String myString1 = String("Hello Arduino !");
myString1.replace("Arduino", "World");
Serial.println(myString1);
}

void loop()
{
}

```

[실행결과]

Hello World !

5.7. 기타 함수

그 외에 문자열과 함께 자주 사용되는 함수들이다. 여기에 소개된 것 외에도 많이 있지만 모든 함수를 다 알 필요는 없다. 우선 이곳에 소개된 것을 숙지한 후 프로그래밍을 하면서 필요한 부분을 별도로 찾아보면 된다.

- **toInt()** : 문자열이 정수형 숫자를 나타낼 때 실제 정수값으로 변환하여 반환, 숫자가 아닌 경우 0을 반환
- **toFloat()** : 문자열이 소수점을 포함하여 실수를 나타낼 때 실제 실수값으로 변환하여 반환, 숫자가 아닌 경우 0.0 을 반환
- **toLowerCase()** : 모든 알파벳 문자를 소문자로 변환
- **toUpperCase()** : 모든 알파벳 문자를 대문자로 변환
- **trim()** : 문자열의 앞과 뒤에 있는 공백 문자를 제거

- `toCharArray(buffer, size)` : `String` 타입의 문자열을 `size`에 맞춰 `char` 형 `buff`에 저장

사용방법은 아래의 예제를 참고한다.

`myString1` 은 문자열의 시작부분과 끝부분에 공백문자가 3개씩 들어있고, 대문자와 소문자가 함께 쓰인 문장이다. 이것을 `toLowerCase()`, `toUpperCase()`, `trim()` 을 이용해 대문자, 소문자로 변환하고 문장의 앞과 뒤에 있는 불필요한 공백문자를 제거했다.

`myString2` 와 `myString3` 는 각각 정수와 실수 형태를 담은 문자열이다. 이것을 각각 `toInt()` 와 `toFloat()` 로 정수와 실수로 변환한 후 10을 곱하여 출력해보았다. 문자열이라면 * 연산을 할 수 없지만 정수와 실수는 * 연산을 할 수 있다.

```
void setup()
{
  Serial.begin(9600);

  String myString1 = String("  Hello Arduino !  ");
  String myString2 = String("123");
  String myString3 = String("0.456");

  myString1.toLowerCase();
  Serial.println("Lower Text : " + myString1);
  myString1.toUpperCase();
  Serial.println("Upper Text : " + myString1);
  Serial.print("String to Int (x10) : ");
  Serial.println(10*myString2.toInt());
  Serial.print("String to Float (x10) : ");
  Serial.println(10*myString3.toFloat());
```

```

Serial.print("before trim() Length is ");
Serial.println(myString1.length());
Serial.println(myString1);

myString1.trim();
Serial.print("after trim() Length is ");
Serial.println(myString1.length());
Serial.println(myString1);
}

void loop()
{
}

```

[실행결과]

```

Lower Text :    hello arduino !
Upper Text :    HELLO ARDUINO !
String to Int (x10) : 1230
String to Float (x10) : 4.56
before trim() Length is 21
    HELLO ARDUINO !
after trim() Length is 15
    HELLO ARDUINO !

```

`toCharArray()` 는 `String` 에 붙어서 다음과 같이 사용할 수 있다.

`myString.toCharArray(buf, len)`

'buf' 는 `char` 타입의 배열이 들어가고, `len`에는 자릿수+1 이 들어간다. 추

가되는 +1은 NULL(\0) 이 들어가는 자리다. "1234" 라는 문자열을 buf에 입력하려면 char buf[5] 라는 5 크기의 문자형 배열이 있어야하고(마지막 자리는 \0 이 들어감), len 은 \0 을 포함한 크기인 5가 된다.

다음과 같이 문자열 "2014" 를 char buf[5] 에 입력할 수 있다.

char buf[] 를 사용하면 atoi() 함수를 사용할 수 있다. atoi() 는 char 배열에 있는 문자열을 정수로 변환할 수 있다. 위에서 언급한 toInt() 를 myString 에 직접 사용해도 되고, char 타입의 배열로 변환한 뒤에 atoi() 를 사용해도 된다.

그럼 간단하게 시리얼모니터로 숫자를 치면 그 숫자에 1 을 더한 수를 시리얼모니터창에 출력하는 프로그램을 만들어보겠다. 위에서 언급한 String, toInt(), atoi() 등을 사용하면 된다.

PC 에서 C 를 사용할때는 표준입출력을 사용할 수 있으므로 printf() 와 scanf() 를 사용해서 어렵지 않게 모니터 화면을 사용할 수 있다. 반면 아두이노는 시리얼모니터 창을 통해 아두이노에서 PC 쪽으로 데이터를 보내고, 그 데이터를 PC에서 시리얼모니터 창을 통해 확인하는 방식을 사용한다. 세부적인 부분은 차이가 있겠지만 일반적으로 다음과 같은 과정으로 아두이노와 PC 는 시리얼통신을 하게 된다.

1. 시리얼 통신 시작한다.
2. PC에서 키보드를 통해 입력을 받는다.
3. 입력이 끝난 문자열을 아두이노로 보낸다.
4. 아두이노에서 시리얼 통신으로 들어오는 문자를 하나씩 받는다.
5. 아두이노는 하나씩 받은 문자를 하나로 합친다.
6. 문자열을 그대로 사용하거나 혹은 정수나 실수로 변환하여 사용한다.
7. 연산이 마친 다음 아두이노에서 PC 쪽으로 문자열을 보낸다.

이러한 과정으로 PC 의 키보드 입력을 시리얼통신으로 아두이노에 전달하면, 아두이노는 그 문자열을 받고 정수로 변환한 다음 1을 더하여 다시 PC 로 전달하는 코드는 다음과 같다.

```
*****  
* simpleInOut.ino  
*  
* 시리얼 통신을 이용한 간단한 입출력 프로그램  
*****  
  
void setup()  
{  
    Serial.begin(9600);  
}  
  
void loop()  
{  
    char c, myChar[5];  
    String myString = "";  
    int num;  
  
    while(Serial.available())  
    {  
        c = Serial.read();  
        myString.concat(c);  
    }  
  
    if (myString != "")  
    {  
        myString.toCharArray(myChar, 5);  
        num = atoi(myChar);  
    }  
}
```

```
    Serial.println(num+1);
}
delay(100);
}
```

위의 코드에서 atoi() 대신 toInt() 를 사용할 수도 있다. 그 경우 아래와 같이 if 구문을 조정하면 된다.

```
if (myString != "")
{
    myString.toCharArray(myChar, 5);
    num = atoi(myChar);
    Serial.println(num+1);
}
```

```
if (myString != "")
{
    num = myString.toInt();
    Serial.println(num+1);
}
```

이상으로 아두이노에서 문자열을 처리하는 방법에 대해서 간략하게 살펴보았다. 아두이노와 같은 MCU를 사용하다보면 시리얼모니터를 이용해서 디버깅하는 방법에 익숙해져야 한다. 특히 아두이노 같은 8비트 MCU 는 디버깅을 위해 별도의 장치를 갖추는 것이 쉽지 않다. 그래서 시리얼모니터를 통한 디버깅이 주로 이용된다. 시리얼모니터를 통한 디버깅시 문자열 처리에 익숙하면 보다 편한 디버깅이 가능하다. 이번 장에서 다룬 내용들을 잘 익혀두기 바란다.

Chapter 06

주요 학습 목표

- C 연산자의 이해
- 연산자 우선 순위를 이용한 코딩

Chapter 06. 연산자

6.1. 연산자란

연산자는 정의된 연산을 수행하는 기호로 연산자의 종류로는 산술, 논리, 대입, 관계, 조건, 비트연산자등이 있다. 연산자와 피연산자들의 조합으로 수식을 표현한다.

연산자의 간단한 예로 `+`, `-`, `*`, `/` 등이 있으며 이러한 연산의 대상이 되는 피연산자로는 상수, 변수, 다른 수식등이 있다.

아두이노 C에서 사용되는 연산자는 다음과 같다.

연산자		의미	용례	비고
산술연산자	+	더하기	$y = a + b$	
	-	빼기	$y = a - b$	
	*	곱하기	$y = a * b$	
	/	나누기	$y = a / b$	
	%	나머지	$y = a \% b$	정수형만 가능
관계연산자	==	같다	$y == x$	
	!=	같지 않다	$y != x$	
	>	크다	$y > x$	
	>=	크거나 같다	$y >= x$	
	<	작다	$y < x$	
	<=	작거나 같다	$y <= x$	

그림 88 연산자1

연산자		의미	용례	비고
할당연산자	=	대입	$y = x$	
	+=	덧셈 후 할당	$y += x$	
	-=	뺄셈 후 할당	$y -= x$	
	*=	곱셈 후 할당	$y *= x$	
	/=	나눗셈 후 할당	$y /= x$	
	%=	나머지 후 할당	$y %= x$	
논리연산자	&&	AND	$(y>0) \&\& (Y<10)$	
		OR	$(y>10) (Y<0)$	
	!	NOT	$!(y>0)$	
비트 논리연산자	&	비트 AND	$x \& y$	
		비트 OR	$x y$	
	^	비트 XOR	$x ^ y$	
	~	비트 NOT	$\sim x$	1의 보수
비트 시프트연산자	<<	왼쪽 이동	$x << 2$	오른쪽 0 채움
	>>	오른쪽 이동	$x >> 2$	왼쪽 0 채움

그림 89 연산자2

책의 앞부분에서 프로그래밍 코드의 내용에 '=' 기호가 종종 사용되었다. '='의 정식명칭은 대입연산자로, 대입연산자는 다음과 같은 형태로 가지고 사용된다.

변수(variable) = 표현식(expression);

'=' 오른쪽의 표현식은 결과로 어떤 값을 만들어 낸다. 표현식의 결과로 만들어진 값(value)이 메모리 공간에 있는 변수(variable)에 저장된다. 명령의 끝은 세미콜론(;)이고, 여기서 중요한 명령인 연산자는 '='이다. 대입연산자는 대입연산자의 오른쪽과 왼쪽의 값이 같다는 것을 나타내는 것이 아니라, 오른쪽의 값을 왼쪽의 공간에 넣으라는 명령이다. 즉, 대입연산자는

다음과 같이 사용할 수 있다.

$$y = y + 1;$$

위 식은 오른쪽의 표현식($y+1$)의 값을 구한 뒤, 그 값을 왼쪽의 공간(y)에 넣으라는 뜻이다. 즉, 변수 y 의 값을 구하고 거기에 1을 더한 다음 그 값을 변수 y 에 넣으라는 명령으로 변수 y 를 1 증가시킨다.

6.2. 산술 연산자

산술연산자는 정수와 실수에 대해서 사칙연산과 나머지 연산을 할 때 사용된다.

연산자		의미	결과	비고
덧셈	+	$a + b$	합	
뺄셈	-	$a - b$	차	
곱셈	*	$a * b$	곱	
나눗셈	/	a / b	a, b 정수 : 몫	
		a / b	a, b 실수 : 나눈 값	
나머지	%	$a \% b$	나머지	
단항 플러스	+	+a		
단항 마이너스	-	-a	a의 부호 변경	

그림 90 산술연산자

단항 플러스와 단항 마이너스는 양수와 음수를 표현할 때 사용된다. 변수 x 에 저장된 값을 부호를 바꿔서 y 에 저장하고 싶으면 다음과 같이 하면 된다. 산술연산자를 사용할 수 있는 피연산자는 정수나 실수여야한다. 즉, 실제 숫자를 사용하거나 숫자를 담을 수 있는 변수를 사용할 수 있다.

```
y = -x;
```

A. 덧셈, 뺄셈, 곱셈 연산자

덧셈, 뺄셈, 곱셈에 대한 산술연산자를 사용해보면 다음과 같다.

```
void setup()
{
    Serial.begin(9600);
    int a, b, x, y;
    a = 10;
    b = 4;

    // 산술연산자 +
    y = a + b;
    Serial.println(y);

    // 산술연산자 -
    y = a - b;
    Serial.println(y);

    // 산술연산자 *
    y = a * b;
    Serial.println(y);
}

void loop() {}
```

결과는 다음과 같다. 정수형 변수 a, b에 각각 10과 4를 넣고 덧셈, 뺄셈, 곱셈을 하면 다음과 같은 결과가 나온다.

14

6

40

나눗셈 연산의 경우 조금 주의할 부분이 있다. 정수와 정수의 연산인 경우와 실수가 포함된 연산인 경우 연산결과가 달라진다. 정수와 정수의 연산결과는 정수형으로 나온다. 하지만 하나로도 실수가 포함되어 있으면 연산결과는 실수형으로 나온다. 즉, 정수 10 을 정수 4 로 나누면 그 결과가 정수형으로 저장되어 2 가 된다. 하지만 정수 10 을 실수 4.0 으로 나누면 그 결과가 실수형으로 저장되고 2.5 가 된다.

아래 실행결과를 확인해보자.

```
void setup()
{
    Serial.begin(9600);
    int a, b, y;
    float fa, fb, fy;

    a = 10;
    b = 4;
    fa = 10.0;
    fb = 4.0;

    // 산술연산자 정수 나눗셈
    y = a / b;
```

```

Serial.println(y);

// 산술연산자 실수 나눗셈
fy = fa / fb;
Serial.println(fy);
}

void loop() {}

```

[실행결과]

2
2.50

정수와 정수의 연산이 정수형으로 결과가 나오는 것을 확인해 두자. 그러면 정수형 변수에 담긴 값을 실수형으로 확인하고자 하면 어떻게 하면 될까? 그때 형변환이라는 방법을 사용해서 정수를 실수로 바꿀 수 있다. 정수형 변수 a 에 담긴 10 이라는 정수를 실수인 10.0 으로 바꾸려면 float(a) 를 하면 된다. 덧셈, 뺄셈, 곱셈에서는 정수와 정수의 연산이 정수로 나온다. 하지만 나눗셈은 정수와 정수의 연산이 반드시 정수로 나오지 않는다. 이런 경우 형변환을 사용하여 산술연산을 진행한다.

B. 형변환

명시적 형변환

정수형 변수 a, b 에 10 과 4 가 저장되어 있을 때 a / b 를 실수로 저장하기 위해 a 또는 b 중의 하나를 실수로 형변환시켜준다.

```
float(a) / b
```

위와 같이 하나의 값을 강제로 다른 타입으로 변화시키는 것을 명시적 형변환이라고 한다. 명시적 형변환 때 주의할 것이 있다. 하나의 자료형이 다른 자료형을 포함하는 여부에 따라 제대로 변환될 수도 있지만 경고가 발생할 수도 있다. 정수형을 실수로 변환하는 경우 문제없이 변환되지만 반대로 실수로 정수로 변환할 경우 소수점이하 자리를 버리는 결과가 되어 캄파일시 경고가 뜨게 된다.

```
char → int → long → float → double : 형확장[변환 이루어짐]  
char ← int ← long ← float ← double : 형축소[경고 발생]
```

형변환은 위의 키워드를 사용하여 기존의 타입을 새로운 타입으로 변환한다. int 타입으로 정의된 변수 a 를 float 으로 변환하려면 float(a) 를 하면 된다. 마찬가지로 실수 b 를 정수 int 로 변환한다면 int(a) 를 하면 된다. 실수 10.7 을 int 로 변환하면 소수점 이하를 버린다(반올림이 아니다).

```
/*********************************************  
* type_cast.ino  
*  
* 형변화  
* 10, 10.7 을 각각 float, int 로 변환  
*****  
void setup()  
{  
    Serial.begin(9600);  
  
    int a = 10;
```

```
float b = 10.7;  
  
Serial.println(float(a));  
Serial.println(int(b));  
}  
  
void loop() {}
```

[실행결과]

```
10.00  
10
```

묵시적 형변환

명시적 형변환이 강제적인 형변환이라면 묵시적 형변환은 다른 타입의 피연산자들이 연산되면서 자동적으로 이루어지는 형변환이다. 묵시적 형변환은 사용자가 어떤 조처를 할 필요없이 컴파일러 내부에서 이루어진다. C는 두개 이상의 피연산자를 가지는 연산에서 필요한 기억 장소의 크기를 피연산자들 중 가장 큰 쪽으로 자동변환한다.

자료형은 기계(Windows, Mac, Linux, Unix 등)마다 다를 수 있고, 같은 기계내에서도 컴파일러에 따라 달라질 수 있다. Windows 운영체제에서 사용되는 C도 컴파일러에 따라 자료형의 크기가 달라진다. 많이 사용되는 Microsoft Visual C++ 6.0에서의 자료형은 다음과 같다.

MSVC 6.0 정수 자료형 [크기(바이트수)] =>
char [1], short [2], int [4], long [4],

```
float [4], double [8], long double [8]
```

온라인 컴파일러인 onlinedb 를 사용해서 출력한 자료형 크기는 다음과 같다. MSVC 6.0 에 비해 일부 자료형의 크기가 커진 것을 확인할 수 있다.

```
*****
* data_size.c
*
* 자료형 크기 출력 (char, short, int, long,
* float, double, long double)
*
*****
#include <stdio.h>
int main()
{
    printf("%ld \n", sizeof(char));           // 1
    printf("%ld \n", sizeof(short));          // 2
    printf("%ld \n", sizeof(int));            // 4
    printf("%ld \n", sizeof(long));           // 8
    printf("%ld \n", sizeof(float));          // 4
    printf("%ld \n", sizeof(double));         // 8
    printf("%ld \n", sizeof(long double));    // 16
    return 0;
}
```

[계산결과]

```
1
2
4
```

8
4
8
16

OnlineGDB 정수 자료형 [크기(바이트수)] =>
char [1], short [2], int [4], long [8],
float [4], double [8], long double [16]

아두이노의 기본 자료형 크기는 다음과 같다.

32비트, 64비트 PC 와는 달리 8비트 마이크로프로세서를 사용하는 아두이노는 기본 자료형의 크기가 작은 것을 확인할 수 있다. 아래에서 확인할 수 있듯이 아두이노는 실수 연산에서 float 과 double 사이에 차이가 없다.

```
*****
* data_size.ino
*
* 자료형 크기 출력 (char, short, int, long,
* float, double, long double)
*****
void setup()
{
    Serial.begin(9600);
    Serial.println(sizeof(char));           // 1
    Serial.println(sizeof(short));          // 2
    Serial.println(sizeof(int));            // 2
    Serial.println(sizeof(long));           // 4
    Serial.println(sizeof(float));          // 4
    Serial.println(sizeof(double));         // 4
```

```
Serial.println(sizeof(long double)); // 4
}
void loop() {}
```

Arduino 정수 자료형 [크기(바이트수)] =>
char [1], short [2], int [2], long [4],
float [4], double [4], long double [4]

자동 형변환은 항상 작은 곳에서 큰 곳으로 이동한다. 즉, int 형과 long 형이 피연산자로 들어가면 int 형은 자동적으로 long 형으로 바뀐 후 결과값을 만들어 낸다. int 형 정수와 float 형 실수의 연산의 int 형 정수가 float 형 실수로 자동 형변환이 일어나고 그 후에 float 형 실수로 결과가 나오게 된다.

두 정수의 나눗셈 결과를 실수로 출력하기 위해 하나를 실수로 형변환 한 다음, 산술연산을 한 코드는 다음과 같다. a 만을 float 형 실수로 변환했지만 / 연산자에 사용되는 b 도 컴파일러 내에서 자동으로 float 형으로 형변환이 일어난 후 결과가 만들어진다.

```
void setup()
{
    Serial.begin(9600);
    int a, b;
    float fy;

    a = 10;
    b = 4;

    // 산술연산자 / 와 형변환
    fy = float(a) / b;
```

```
    Serial.println(fy);
}

void loop() {}
```

[실행결과]

2.50

C. 나머지 연산자

산술연산자 중 나머지연산자도 많이 사용되는 연산자다.

나머지 연산은 첫번째 피연산자를 두번째 피연산자로 나눈 나머지 정수값이 된다. 결과의 부호는 컴파일러에 따라 다를수 있으며 대부분의 컴파일러와 아두이노에서는 분자에 해당하는 첫번째 피연산자의 부호를 따른다. 나머지 연산은 두개의 피연산자가 모두 정수형이어야 한다.

```
*****
* remainder_operator.ino
*
* 나머지연산
* 10, -10 을 4, -4 로 나눈 나머지 구하기
*****
void setup()
{
    Serial.begin(9600);

// 산술연산자 중 나머지연산 %
```

```

Serial.println(10%4);
Serial.println((-10)%4);
Serial.println(10%(-4));
Serial.println((-10)%(-4));
}

void loop() {}

```

[실행결과]

```

2
-2
2
-2

```

나머지 연산의 결과 부호값은 정리하면 다음과 같다. 첫번째 연산자의 부호와 연산 결과의 부호가 같다는 것을 기억해두기 바란다.

```

10 % 4 = 2
-10 % 4 = -2
10 % -4 = 2
-10 % -4 = -2

```

D. 단항 마이너스, 단항 플러스

일반적으로 단항 플러스 연산자는 거의 사용되지 않는다. 단항 플러스는 이전 값 그대로를 보존하기 때문에 붙이는 것이 아무 의미가 없다. 변수 a의 값을 그대로 사용할 때 '+' 를 앞에 붙일 수 있으며, 기본값으로 '+' 가 생략되어 사용된다.

$a = +a$

$b = +a$

단항 마이너스 연산자는 원래의 값의 부호를 바꾼다. 변수 a 에 4라는 값이 담겨 있다면 $-a$ 는 부호가 바뀐 -4 가 된다. 아래처럼 부호를 바꿀 때 사용할 수 있다.

$b = -a$

$a = -a$

6.3. 할당 연산자, 대입 연산자

앞에서 대입연산자를 살펴보았다. 대입연산자 '='은 오른쪽의 값을 왼쪽의 공간에 넣으라는 명령이다. 대입연산자와 다른 연산자를 결합시켜 할당연산자를 만들어 사용할 수 있다.

기본 산술연산자와 대입 연산자의 결합은 다음과 같다. 한줄을 짧게 쓰면서 의미를 담기 위해서 사용되는 문법이지만, 최근에 나오는 프로그래밍 언어는 이런 할당 연산을 피하고 있다.

할당연산자		사용법	의미	비고
덧셈	$+=$	$y += a$	$y = y + a$	
뺄셈	$-=$	$y -= a$	$y = y - a$	
곱셈	$*=$	$y *= a$	$y = y * a$	
나눗셈	$/=$	$y /= a$	$y = y / a$	
나머지	$\%=$	$y \%= a$	$y = y \% a$	정수만 가능

그림 91 할당연산자

y += a 는 y의 값에 a의 값을 더한 후 그 결과를 y에 저장한다.
y -= a 는 y의 값에 a의 값을 뺀 후 그 결과를 y에 저장한다.
y *= a 는 y의 값에 a의 값을 곱한 후 그 결과를 y에 저장한다.
y /= a 는 y의 값에 a의 값을 나눈 후 그 결과를 y에 저장한다.
y %= a 는 y의 값을 a의 값으로 나눈 나머지를 y에 저장한다.

할당연산을 사용한 예는 다음과 같다.

```
*****
* comp_assign_op.ino
*
* 할당연산자 사용 (+=, -=, *=, /=, %=)
*****
```

```
void setup()
{
    Serial.begin(9600);
    int a = 3;
    int y = 10;

    Serial.println(y += a);    // y = 13 <- 10+3
    Serial.println(y -= a);    // y = 10 <- 13-3
    Serial.println(y *= a);    // y = 30 <- 10*3
    Serial.println(y /= a);    // y = 10 <- 30/3
    Serial.println(y %= a);    // y = 1 <- 10%3
}

void loop() {}
```

위에 제시된 것 외에 비트연산자와 결합된 할당연산자도 있다. 비트연산자와 결합된 할당연산자는 비트연산자를 다루면서 살펴보도록 한다.

6.4. 증감연산자

값을 하나 증가 혹은 감소시킬 때 사용한 특별한 연산자를 C에서 제공한다. 증감연산자는 증가연산자와 감소연산자를 함께 부르는 말로 증가연산자(increment operator)는 변수의 값을 1 증가시키는 연산자이고, 감소연산자(decrement operator)는 변수의 값을 1 감소시키는 연산자이다. 증감연산자는 피연산자의 값을 변경시키기 때문에 피연산자는 상수가 아닌 변수이다.

변수++

++변수

증가연산자는 위와 같은 형태로 사용되며, 위 두 수식은 변수를 1만큼 증가시킨다. ++ 대신 -- 를 사용하면 감소연산자가 되고 변수를 1 감소시킨다.

변수--

--변수

증감연산자는 C 이외의 프로그래밍 언어에서는 잘 지원하지 않는다. 증감연산자를 대입연산자로 풀어서 사용할 수 있고, 복잡한 연산식 내부에 증감연산자가 사용되었을 경우 우선순위에 따른 연산결과가 예상하는 것과 다르게 나올 수 있기 때문에 증감연산자의 사용은 주의할 필요가 있다.

원 연산식	같은 의미 연산식
a++; 혹은 ++a;	a+=1;
	a=a+1;
a--; 혹은 --a;	a-=1;
	a=a-1;

그림 92 증감연산자

증가연산자가 피연산자의 뒤에 위치할 때 후치(postfix) 형태라고 하고 앞에 위치할 때 전치(prefix) 형태라고 한다. 즉, $a++$ 와 $++a$ 의 형태는 연산에 영향을 미친다.

$$\begin{aligned} y &= a++ \quad \text{----- (1)} \\ y &= ++a \quad \text{----- (2)} \end{aligned}$$

위의 두 식은 다음과 같은 순서로 실행된다.

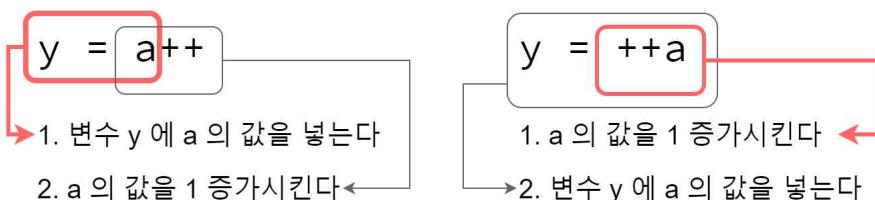


그림 93 증감연산자 전치와 후치의 연산순위

결과적으로 $y = a++$ 에서 y 는 변수 a 가 가진 이전값(1증가 되기 전의 값)을 가지게 되고, $y = ++a$ 는 y 는 a 가 가진 값보다 1 큰 값(1증가 된 후의 값)을 가지게 된다.

이렇게 실행순서가 직관적으로 인식되지 않기 때문에 다음과 같은 식을 쓸 때 종종 원치 않는 오류를 가지게 된다.

$$y = 4 - a++ \quad \text{----- (1)}$$

$$y = 6 + --a \quad \text{----- (2)}$$

$$y = 2 * a++ * 3 \quad \text{----- (3)}$$

(1), (2), (3)의 경우는 어떻게 연산 결과가 나오는지를 이해하는 것보다 가급적 이런식으로 코드를 만들지 않도록 하자. 다른 연산자와 함께 증감연산자를 사용하게 되면 가독성을 현저하게 떨어지게 되고, 논리오류가 발생할 가능성이 매우 커진다. 가능하면 위의 세 경우는 다음과 같이 풀어서 쓰는 것이 좋다. 한두줄 코드가 늘어나지만 가독성을 훨씬 좋아진다.

$y = 4 - a++ ;$	$y = 6 + --a ;$	$y = 2 * a++ * 3 ;$
↓	↓	↓
$y = 4 - a ;$ $a++ ;$	$--a ;$ $y = 6 + a ;$	$y = 2 * a * 3 ;$ $a++ ;$

그림 94 증감연산자 사용법

이런 문제들 때문에 최근에 나오는 프로그래밍 언어에서는 증감연산자가 사용되지 않는 경우가 많다. 반복문 등에서 변수의 값을 1씩 증가 혹은 감소시킬 때 증감연산자는 유용하다. 이러한 경우를 제외하고 증감연산자를 무분별하게 사용하는 것은 자제하는 것이 좋다. 증감연산자를 사용하는 이유가 코드수를 줄여 간결하게 만들어 보기 쉽게 하려는 것인데 오히려 증감연산자때문에 코드의 가독성이 떨어진다면 차라리 쓰지 않는 것이 나을

수도 있다. 즉, 증감연산자를 사용해서 만들어진 코드가 가독성이 좋지 않다면, 증감연산자는 사용하지 않는 것이 좋다.

```
*****
* inc_dec_op.ino
*
* 증감연산자 : 사용 예
*****
void setup()
{
    Serial.begin(9600);
    int y, a;

    a = 4; y = a++;
    Serial.println(y);      // 4, 좋은 예

    a = 4; y = ++a;
    Serial.println(y);      // 5, 좋은 예

    a = 4; y = 4 - a++;
    Serial.println(y);      // 0, 조심해야 할 예 (비추천)

    a = 4; y = 6 + --a;
    Serial.println(y);      // 9, 조심해야 할 예 (비추천)

    a = 4; y = 2 * a++ * 3;
    Serial.println(y);      // 24, 조심해야 할 예 (비추천)
}

void loop() {}
```

6.5. 관계 연산자, 비교 연산자

관계연산자는 두 피연산자의 값의 관계를 비교하는 연산자다. 두 피 연산자의 값이 같은지 그렇지 않은지를 검사하는 동등 연산자와 값의 대소를 비교하는 크기 비교 연산자가 있다. C 언어에는 값들의 동등, 대소와 관련해서 다음과 같은 관계연산자가 있다. '관계 연산자'라는 말 대신 '비교 연산자'라는 단어도 자주 사용된다.

C 문법에서 관계연산자는 $==$, $!=$, $<$, $<=$, $>$, $>=$ 의 6가지가 있다. 각 관계연산자의 연산결과는 다음표와 같다.

연산	연산자	결과값	비고
같다	$x == y$	1 (참), 0 (거짓)	x, y 가 같을 때 참
다르다	$x != y$	1 (참), 0 (거짓)	x, y 가 다를 때 참
작다	$x < y$	1 (참), 0 (거짓)	x 가 y 보다 작을 때 참 (같으면 거짓)
작거나 같다	$x <= y$	1 (참), 0 (거짓)	x 가 y 보다 작거나 같으면 참
크다	$x > y$	1 (참), 0 (거짓)	x 가 y 보다 클 때 참 (같으면 거짓)
크거나 같다	$x >= y$	1 (참), 0 (거짓)	x 가 y 보다 크거나 같으면 참

그림 95 관계연산자

관계연산자는 반복문, 조건문과 함께 사용되는 경우가 많다. 관계연산자의 결과는 참이나 거짓이지만 C에서는 별도로 논리형이 없기 때문에 참은 1, 거짓은 0으로 표현한다.

관계연산자를 사용한 결과를 출력한 예제는 다음과 같다.

```

/***********************
* relation.ino
*
* 관계연산자 사용 (==, !=, <, <=, >, >=)
***********************/

void setup()
{
    Serial.begin(9600);
    int x = 4;
    int y = 6;

    Serial.println(x == y);      // 0
    Serial.println(x != y);     // 1
    Serial.println(x > y);     // 0
    Serial.println(x >= y);    // 0
    Serial.println(x < y);      // 1
    Serial.println(x <= y);    // 1
}

void loop() {}

```

6.6. 논리 연산자

C 언어에는 3개의 논리연산자가 있다. AND, OR, NOT 논리연산자는 각각 `&&`, `||`, `!` 기호를 사용한다.

`&&` 연산자는 논리 AND 연산을 수행한다. 논리 AND 연산은 "논리곱" 이라고 불리며, 두개의 피연산자를 가지는 이항연산자로 두개의 피연산자가

모두 참일때 결과가 참이 된다. 참이라는 것은 0 이 아닌 값을 의미한다.

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

그림 96 논리 AND 연산자

|| 연산자는 논리 OR 연산을 수행한다. 논리 OR 연산은 "논리합" 이라고 불리며, 두개의 피연산자를 가지는 이항연산자로 두개의 피연산자가 하나라도 참이라면 결과가 참이 된다.

a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

그림 97 논리 OR 연산자

! 연산자는 논리 NOT 연산을 수행한다. 논리 NOT 연산은 "논리부정" 으로 불리며, 피연산자가 하나인 단항연산자로 연산의 결과 값은 다음과 같다.

a	!a
0	1
1	0

그림 98 논리 NOT 연산자

입력으로 들어온 온도값이 37 도와 39 도 사이에 있는 알기 위해서 다음과 같이 논리연산자와 관계연산자를 사용할 수 있다.

```
temp >= 37.0 && temp <= 39.0
```

위의 결과가 1 이면 temp 는 37.0 이상 39.0 이하라는 뜻이고, 0이라면 그 외의 범위라는 뜻이다.

6.7. 윤년 판단 예제

위에서 나온 관계연산자와 조건연산자를 사용해서 입력받은 값이 윤년인지 여부를 출력하는 함수를 만들수 있다. 반복문이나 조건문 없이 연도를 입력받은 후 입력받은 연도가 윤년이라면 1 을 아니라면 0 을 출력하는 프로그램을 만들어보자.

윤년은 4로 나누어 떨어지고, 100 으로 나누어 떨어지지 않는다. 단, 400 으로 나누어 떨어지는 것은 윤년이다.

```
*****
```

```

* leafYear.ino
*
* 윤년 계산하기 : 윤년이면 1, 아니면 0 출력
***** */
void setup()
{
    Serial.begin(9600);
}
void loop()
{
    char c;
    String myString = "";
    while(Serial.available())
    {
        c = Serial.read();
        myString.concat(c);
    }
    if(myString != "" )
    {
        char yearData[5];
        myString.toCharArray(yearData, 5);
        int year = atoi(yearData);

        int leapYear = (year % 400 == 0) ||
                      (((year % 4) == 0) && (year % 100 != 0) );
        Serial.println(leapYear);
    }
    delay(100);
}

```

6.8. 조건 연산자

조건연산자는 삼항연산자로 조건이 참이 경우와 거짓인 경우에 대한 각각의 명령이 존재한다. C 의 유일한 삼항연산자로 항이 3개인 구조라 복잡해 보이지만 자세히 살펴보면 그리 어려운 구조는 아니다. 삼항연산자는 if 조건문을 사용해서 풀어 사용할 수 있으며, 간단한 조건문의 경우 한줄로 조건에 맞는 경우와 그렇지않은 경우로 모두 쓸수 있어 줄수를 줄이고자 할 때 주로 사용된다.

“조건식” ? “참인 경우 실행할 명령” : “거짓인 경우 실행할 명령”

조건식은 관계연산자를 주로 사용한다. 변수의 a 의 값을 읽어서 짹수라면 "even" 을 출력하고, 홀수라면 "odd"를 출력하는 프로그램을 조건연산자를 사용해서 만들어보면 다음과 같다.

```
/****************************************************************************
 * cond_op.ino
 *
 * 조건연산자 : 짹수이면 "even" 홀수이면 "odd" 출력
 */
void setup()
{
    Serial.begin(9600);
    int a = 16;
    (a % 2 == 0) ? Serial.println("even") : Serial.println("odd");
}

void loop() {}
```

삼항연산자를 다른 연산식 안에서 사용할수 도 있다. 아래 코드는 4자리수

를 가진 자연수 입력 myNum 변수가 0보다 크면 그대로 내보내고 myNum 변수의 값이 0 이하라면 0이 되게 하는 삼항연산자의 사용예다. 즉, 2020 이 들어오면 그대로 2020 이 출력되고, -2 가 들어오면 0이 출력된다.

```
*****
* right_year_op.ino
*
* 조건연산자 : 연도를 입력받아 출력하기,
* 입력받은 해가 0보다 크면 그대로, 0이하면 0으로 출력
*****
```

```
void setup()
{
    Serial.begin(9600);
}

void loop()
{
    char c;
    String myString = "";
    while(Serial.available())
    {
        c = Serial.read();
        myString.concat(c);
    }
    if(myString != "" )
    {
        char yearData[5];
        myString.toCharArray(yearData, 5);
        int year = atoi(yearData);

        year = year>0 ? year: 0;
```

```

    Serial.println(year);
}
delay(100);
}

```

[입력]	[출력]
1024	1024
2048	2048
-10	0
0816	816

그림 99 윤년 검사 입력과 출력

6.9. 콤마 연산자

콤마(",")연산자는 피연산자 두개를 받는 이항 연산자로 콤마의 좌측에 있는 첫번째 식을 수행한 다음 우측의 식을 수행한다. 콤마연산자는 우선 순위가 낮아서 일반적으로 괄호와 함께 사용된다. 콤마연산자의 사용은 주의 할 부분이 있다.

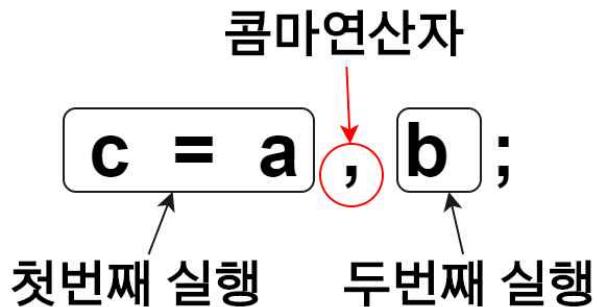


그림 100 콤마연산자

콤마연산자 앞에 있는 부분을 우선 수행하기 때문에 c 에 a 의 값이 들어 가게 된다. 그 다음 콤마 뒷 부분을 실행하는데 이 경우 b 의 값을 호출만 하고 사용하지 않기 때문에 위의 경우 b 는 아무런 일도 하지 않는다.

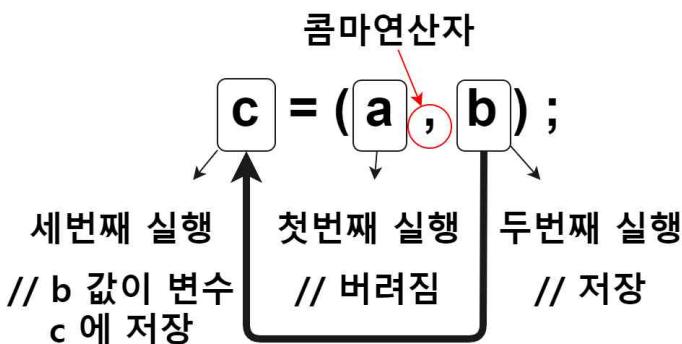


그림 101 콤마연산자 실행순서

여기에 괄호를 사용하게 되면 괄호의 우선순위가 우선이기 때문에 a,b 를 실행한 결과인 b의 값을 가져온 후, 그 b 의 값을 변수 c 에 넣는다. 콤마연산자의 우선순위는 낮기 때문에 대부분의 연산자들 보다 나중에 수행된다.

```
*****
* comma_op1.ino
*
* 콤마연산자 : 사용예
*****
void setup()
{
    Serial.begin(9600);
    int a=1, b=2, c, d;
```

```
c = a , b;  
d = (a , b);  
Serial.println(String(c) + String(d));  
}  
  
void loop() {}
```

a, b 는 콤마 앞의 a 를 먼저 다룬 다음 b 를 다루게 된다. 여기서 콤마연산자보다 대입연산자("=")가 우선순위가 있으므로 변수 c 에 a 의 값이 입력된다. 그 후 b 를 처리하는데 b 를 사용하는 곳이 없으므로 콤마 이후의 b 는 없는 것과 같다.

```
c = a, b;
```

즉, 위 식은 다음과 같다.

```
c = a;  
b;
```

반면 d = (a, b) 의 경우 대입연산자 "=" 보다 () 가 우선순위가 높으므로 (a, b) 가 먼저 수행된다. (a, b) 는 a 를 실행한 후 b 를 실행한다. a를 실행한 결과는 별도로 저장되지 않고, b를 실행한 결과만 남는다. 즉 (a, b) 는 b 와 같다.

```
d = (a, b);
```

식은

d = b;

가 된다.

일반적으로 콤마연산자는 변수 선언시와 반복문안에서 변수의 값을 수정할 때 사용되지만, 그 이외의 경우는 가독성, 최적화 문제, 휴면 에러의 가능성 때문에 사용을 자제하는 것이 좋다.

콤마연산자를 사용하는 경우 두 수식의 실행 순서가 강제되기 때문에 컴파일러는 코드의 실행순서를 최적화하지 못할 수도 있다. 즉, 아래의 수식의 경우 어느쪽을 먼저하든 관계없지만 콤마연산자를 사용할 경우 전반부가 우선적으로 실행된다.

```
++i; ++j; // ++i 와 ++j 는 동시에 실행될 수 있다  
++i, ++j; // ++i 가 ++j 보다 우선 실행된다
```

콤마연산자를 사용하면 두개의 표현식을 하나의 문장으로 만들수 있기 때문에 중괄호 없이 사용할 수 있다. 코드의 줄수를 줄일 수 있다는 점에서 장점으로도 볼 수 있겠지만 바로 눈에 보이지 않는 모호성때문에 실수할 가능성이 커진다. 가독성이 떨어져 원하는 결과를 만들어내지 못할 수도 있다.

```
if (x) a = 1, b = 1;
```

위의 코드는 중괄호를 사용해서 아래처럼 쓰는 것이 일반적이다.

```
if (x)
{
    a = 1;
    b = 1;
}
```

아래의 코드의 출력 결과를 예상해보라.

```
void setup()
{
    Serial.begin(9600);
    int a=1, b=2, c, d;
    c, d = a , b;

    Serial.println(c);
    Serial.println(d);
}
void loop() {}
```

위 코드에서

```
c, d = a , b;
```

부분에 대한 실행은 다음과 같이 진행된다.

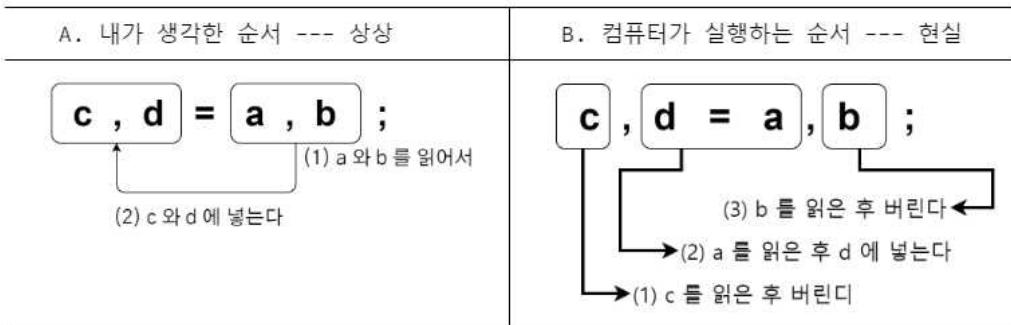


그림 102 콤마연산자 주의할 실행 순서

위의 코드를 생각한(의도한) 순서는 A 처럼 a 와 b 을 값을 읽어 변수 c 와 d 에 각각 넣으려는 것이었지만, 실제로 실행은 B 처럼 콤마를 기준으로 세개의 명령으로 나뉘게 되어 의도와는 다른 실행이 이루어진다. 문제는 위 코드가 문법적으로 문제가 없기 때문에 컴파일러에서 에러메시지가 출력되지 않는다는 것이다. 이러한 오류를 논리오류라고 하고 실제로 디버깅 시에서 발견하기 어려운 형태의 오류가 발생하게 한다.

6.10. 비트 연산자

비트연산자는 아두이노와 같은 임베디드 시스템을 다루는 C에서 반드시 알아야 하는 연산자다. 일반적인 PC 환경에서는 비트연산자를 사용할 일이 많지는 않지만 임베디드 시스템에서는 비트 연산자를 필수적으로 이해하고 있어야 한다. C 언어는 시스템 프로그래밍이 가능한 언어로 설계되었고, 그래서 어셈블러에서나 가능했던 연산들이 가능하다. 비트연산은 레지스터등을 다루거나 다양한 방법으로 하드웨어에 관계되어 사용된다.

비트연산자는 비트단위의 정보를 다룬다. 1 바이트는 8 비트이고, 2바이트는 16비트다. 즉, 변수가 1바이트, 2바이트, 4바이트 등의 여부에 따라 각각 8비트, 16비트, 32비트에 대한 연산이 이루어진다. 아두이노 C는 정수형 중 가장 큰 값을 담을 수 있는 자료형이 long 이고, long 은 4 바이트를 사용한다.

비트 연산자는 비트를 좌나 혹은 우로 이동시키는 이동 연산자와 AND, OR, XOR, NOT 에 대한 논리 연산자가 있다. 비트연산은 정수연산만 가능하다.

비트연산자		내용	결과
비트 이동 연산자	a << 2	왼쪽으로 이동	a 를 2비트 왼쪽으로 이동, 오른쪽 2자리는 0으로 채움
	a >> 2	오른쪽으로 이동	a 를 2비트 오른쪽으로 이동, 왼쪽 2자리는 0으로 채움
비트 논리 연산자	a & 2	비트 AND	a 와 2(0B0010)의 비트 AND
	a 2	비트 OR	a 와 2(0B0010)의 비트 OR
	a ^ 2	비트 XOR	a 와 2(0B0010)의 비트 XOR
	~a	1의 보수	a 의 비트를 반전시킴

그림 103 비트연산자

비트 이동연산자는 비트를 좌 혹은 우로 이동시킨다. 비트가 이동될 때 넘치는 부분(벗어나는 부분)과 모자란 부분(빈 부분)이 발생한다. 8비트로 00001111 이 들어있는 값을 좌로 2비트 이동시키면 다음과 같다.

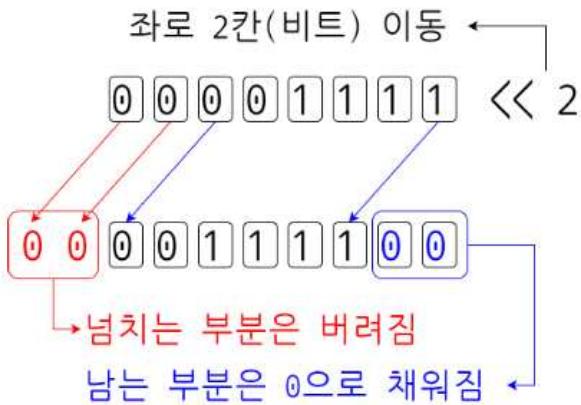


그림 104 비트연산자 이동위치

아래 코드를 실행시켜서 비트가 이동될 때 어떤 결과가 일어나는지 살펴보자.

이진수 0B1111 을 좌, 우로 2비트씩 이동시키며 출력하기 위해 0B1111 << 2 연산을 실행하면 0B111100 이 되고, 0B1111 >> 2 는 0B11 이 된다. 이를 확인해보는 코드는 다음과 같다.

```
*****  

* bit_shift_op1.ino  

*  

* 비트 이동연산자 : 사용 예  

*****  

void setup()  

{  

    Serial.begin(9600);  

    int a=0B00001111;  

    Serial.println(a, DEC); // 15  

    Serial.println(a, HEX); // F
```

```
Serial.println(a, BIN); // 1111 : 0000 0000 0000 1111

int b = (a << 2);
Serial.println(b, DEC); // 60
Serial.println(b, HEX); // 3C
Serial.println(b, BIN); // 111100 : 0000 0000 0011 1100

int c = (a >> 2);
Serial.println(c, DEC); // 3
Serial.println(c, HEX); // 3
Serial.println(c, BIN); // 11 : 0000 0000 0000 0011
}

void loop() {}
```

비트 이동연산에서 한가지 더 알아두어야 할 것은 좌로 비트를 하나 이동할 때마다 2배가 된다는 것이다. 우로 비트를 하나 이동시키면 절반인 2로 나눈 값이 된다. 어떤 수 n 에 2의 10승을 곱하고 싶다면 n << 10 을 사용하면 된다. 아래는 비트연산자를 이용해서 2에 29 을 곱하는 과정을 표로 나타낸 것이다.

2 << n		결과
0000 0000 0000 0010	2	2
0000 0000 0000 0100	2 << 1	4
0000 0000 0000 1000	2 << 2	8
0000 0000 0001 0000	2 << 3	16
0000 0000 0010 0000	2 << 4	32
0000 0000 0100 0000	2 << 5	64
0000 0000 1000 0000	2 << 6	128
0000 0001 0000 0000	2 << 7	256
0000 0010 0000 0000	2 << 8	512
0000 0100 0000 0000	2 << 9	1024

그림 105 비트연산자 이동후 값 변화

이런 비트연산자를 이용한 곱셈은 FFT 등에 사용된다. 정수형 곱셈 중 2의 배수가 필요한 경우 비트연산을 쓰면 빠른 곱셈이 가능하다.

비트 논리연산은 일반적인 PC 프로그래밍에서는 거의 사용되지 않지만 하드웨어를 직접 제어하는 임베디드에서는 자주 사용하는 연산이다. 비트 논리연산은 산술적인 값을 계산하기 보다는 레지스터의 값을 다룰 때 주로 사용된다. 비트 논리연산에는 비트 AND, 비트 OR, 비트 XOR, 비트 NOT 연산자가 있다. 비트 NOT 연산자는 단항연산자로 개별 비트의 값을 반전시켜서 1의 보수(원래 비트의 0과 1을 각각 반전시킨 것)를 만들어준다.

a = 0B00001111;

라면

$\sim a$ 는 0B11110000 이 된다. 맨 앞 비트가 1이므로 2의 보수의 모습을 가진 음수가 된다. 즉 이 수는 1) 1의 보수를 만들고 (00001111) 2) 여기에 1을 더한 후 (00010000, 십진수 16) 3) 마이너스 기호를 붙이면 된다 (-16).

```
void setup()
{
    Serial.begin(9600);
    char a=0B00001111;
    Serial.println(~a); // -16
}
void loop() {}
```

비트 AND, OR 연산은 레지스트를 다루거나 하나의 바이트안에 둘 이상의 정보를 담아서 사용할때 일반적으로 사용된다. 실제 그 외의 용도로 사용되는 경우는 많지 않다. PC 의 경우 메모리가 부족하지 않기 때문에 비트연산을 사용하는 일은 드물다. 하지만 아두이노 같이 메모리가 넉넉하지 않은 경우 효율적인 메모리 활용을 위해 비트연산을 사용하게 된다.

거리센서로 부터 들어오는 값이 0에서 255 사이의 값이고, 밝기 센서로부터 들어오는 값이 0에서 255 사이의 값이라면 이 두 값을 저장하기 위해 두개의 변수를 사용하는 것이 일반적이다. 하지만 이때 비트 연산을 사용하면 하나의 변수에 두개의 값을 담아 사용할 수 있다.

우선 비트 논리 연산자의 기본 사용법은 다음과 같다.

$$\begin{array}{r} 0000 \ 1100 \\ & \& 0000 \ 0101 \\ \hline 0000 \ 0100 \end{array} \quad \begin{array}{r} 0000 \ 1100 \\ | \ 0000 \ 0101 \\ \hline 0000 \ 1101 \end{array} \quad \begin{array}{r} 0000 \ 1100 \\ ^ \ 0000 \ 0101 \\ \hline 0000 \ 1001 \end{array}$$

그림 106 비트연산 and, or, not

- 비트 & 연산은 두개의 비트가 모두 1 일때 결과가 1이 된다. 그외는 0 이 된다.
- 비트 | 연산은 두개의 비트중 하나라도 1 이면 결과가 1이 된다.
- 비트 ^ 연산은 하나가 0이고 다른 하나가 1일때만 결과가 1이 된다.

레지스터의 특정 비트를 수정할 때 비트 논리연산을 사용할 수 있다.

```
ADCSRC = ADCSRC | 0x40 ;
```

혹은 위 코드를 다음과 같이 쓸 수 있다.

```
ADCSRC |= 0x40 ;
```

0x40 은 0B 0100 0000 이므로 8개의 비트 중 왼쪽에서 두번째 비트를 1 로 만들때 비트 OR 연산을 사용한다.

정수형 변수 i 를 선언한 다음 i 에 둘 이상의 값을 넣을 수 있다.

```
int i ;
```

거리센서의 값이 0에서 255 까지의 값을 가지고, 밝기센서의 값도 0에서 255 까지의 값을 가지고, 거리센서의 값이 len에 담겨있고 밝기센서의 값이 brt에 담겨있다.

0에서 255 까지의 값을 8개 비트를 사용하면 표현이 가능하다.

```
0 : 0000 0000  
255 : 1111 1111
```

int 타입은 2바이트, 즉 16비트를 사용하므로

```
i = □□□□ □□□□ □□□□ □□□□  
(□ 는 0 또는 1)
```

이다. 여기서 전반 8개 비트와 후반 8개 비트로 나누어 사용한다. 전반 8개 비트에는 len에 담긴 거리값을 저장하고, 후반 8개 비트에는 brt에 담긴 밝기값을 저장한다.

```
i = len << 8 | brt ;
```

위의 내용이 들어간 전체 소스는 다음과 같다.

```

/***********************
* bit_op2.ino
*
* 비트 이동연산자 : 사용 예
***********************/

void setup()
{
    Serial.begin(9600);
    unsigned int i;

    byte len=4, brt = 19;
    i = len << 8 | brt;

    Serial.println(i, BIN);
    Serial.println(i >> 8);
    Serial.println(i&0x00FF);
}

void loop() {}

```

[실행결과]

10000010011

4

19

이진수로 출력한 *i*의 값은 '10000010011'이다. 이 값을 8개의 비트로 나누면 '100', '00010011'가 된다. 즉 상위 8개 비트는 '00000100'이고, 하위 8개 비트는 '00010011'이다. 각각을 10진수로 변환하면 4와 19가 된다.

6.11. 연산자의 우선 순위와 결합 규칙

C언어 연산자들의 우선순위를 정리하면 다음 표와 같다.

순위	연산자	연산	결합 순서
1	() [] -> . (후위)++ (후위)--	식	왼쪽에서 오른쪽으로→
2	++(전위) --(전위) ! ~ + - * & (형)	단항	←오른쪽에서 왼쪽으로
3	* / %	사칙(*)	왼쪽에서 오른쪽으로→
4	+ -	사칙(+)	왼쪽에서 오른쪽으로→
5	<< >>	비트이동	왼쪽에서 오른쪽으로→
6	< <= > >=	관계	왼쪽에서 오른쪽으로→
7	== !=	관계	왼쪽에서 오른쪽으로→
8	& ^	비트논리	왼쪽에서 오른쪽으로→
9	&&	논리	왼쪽에서 오른쪽으로→
10	? :	조건	←오른쪽에서 왼쪽으로
11	= += -= *= /= %= <<= >>= &= = ^=	할당	←오른쪽에서 왼쪽으로
12	,	콤마	왼쪽에서 오른쪽으로→

그림 107 연산자 우선순위

연산자는 우선순위가 있어서 기본적으로 괄호없이 사용할 수 있다. 하지만 연산자의 우선순위가 모호하거나 낮은 연산자를 먼저 수행할 필요가 있을 때에는 괄호연산자'()'를 사용하면 된다. 나머지는 상식적인 선에서 이해하면 된다. 사칙연산에서 곱셈은 덧셈보다 우선순위가 높고, 할당연산자는 다른 연산자들의 계산이 끝난 다음에 이루어지는 것이 상식적으로 타당하다. 콤마연산자는 왼쪽의 모든 계산식이 마쳐진 다음 오른쪽에 있는 식을 수행한다.

상식적인 선에서 자주 사용되는 연산자들의 우선순위를 이해한 다음 나머지는 굳이 암기할 필요없이 먼저 연산하고자 하는 곳에 괄호 '()'를 사용하면 된다. 괄호는 연산자와 피연산자의 관계를 나타내는데 괄호를 사용하면 강제적으로 연산순서를 변경할 수 있다.

$$y = a + b * c; \quad \text{----- (1)}$$

$$y = (a + b) * c; \quad \text{----- (2)}$$

(1) 은 b 와 c 를 곱한 다음 그 결과에 a 를 더해서 y 에 저장한다. 반면 (2)는 a 와 b 를 먼저 더한 후 그 결과에 c 를 곱해서 y 에 저장한다. 비트연산이나 논리연산에서 AND 연산이 OR 연산보다 우선순위가 높다. 이런 세부 사항을 기억하지 못한다면 괄호를 써서 정확하게 의도하는 바를 드러내면 된다.

$$y = a \& b | c \& d ;$$

위의 식에서 & 와 | 사이의 우선순위에 대해 정확히 암기되어 있지 않을 때 다음식처럼 괄호를 사용하면 된다.

$$y = (a \& b) | (c \& d) ;$$

모든 우선순위를 암기하는 것이 쉽지 않고, 우선순위와 다르게 연산을 해야 할 경우 괄호연산을 사용하면 된다.

6.12. Mini Project 화씨 온도를 섭씨로 바꾸기

간단한 프로젝트를 진행해보자. 키보드로 화씨온도를 입력받으면 섭씨온도로 바꾸는 프로그램 코드를 작성해 본다. 화씨 온도와 섭씨 온도의 변환공식은 다음과 같다.

$$\text{섭씨온도} = (\text{화씨온도} - 32) * 5 / 9$$

위의 공식과 키보드로 입력받은 화씨온도를 이용해서 섭씨 온도를 출력하는 프로그램 코드는 다음과 같다. 코드안에 지금까지 배웠던 시리얼통신, 문자열과 문자열 함수, 연산자와 우선순위 변경등이 포함되어 있다.

```
*****  
* temp_fc.ino  
*  
* 시리얼 통신을 이용한 간단한 입출력 프로그램  
*****  
  
void setup()  
{  
    Serial.begin(9600);  
}  
  
void loop()  
{  
    char c, myChar[5];  
    String myString = "";  
    int f_temp;  
    float c_temp;  
  
    while(Serial.available())  
    {
```

```
c = Serial.read();
myString.concat(c);
}

if (myString != "")
{
    myString.toCharArray(myChar, 5);
    f_temp = atoi(myChar);
    c_temp = (f_temp - 32) * 5 / 9.0;
    Serial.println(c_temp);
}
delay(100);
}
```


Chapter 07

주요 학습 목표

- 제어문을 이해한다
- if를 이용한 조건문을 이해한다
- for, while 을 이용한 반복문을 이해한다

Chapter 07. 제어구조

7.1. 제어흐름

프로그램에서 실행되는 문장의 실행순서를 제어 흐름이라 부른다. C 언어는 기본적으로 main 함수내의 첫번째 문장부터 시작해서 한번에 하나씩 문장씩 실행하고, 마지막 문장에 이르기까지 순차적으로 실행된다.

이런 순차적인 구조만으로는 조건에 따른 작업이나 반복적인 작업을 수행 할 수 없기 때문에 조건과 반복이라는 내용으로 순차적 실행구조를 제어하게 된다.

조건문 : if 문, if-else 문, switch 문

반복문 : while 문, for 문, do-while 문

조건문은 조건에 따라 실행된 내용을 선택할 수 있는 구조로 C의 조건문은 if 문, if-else 문, switch 문등이 있다. 조건은 참과 거짓의 둘 중의 하나의 결과가 나오는 부울식(boolean expression)으로 표현된다.

조건문과 함께 반복문은 제어구조에서 가장 핵심적인 내용이다. 컴퓨터 프로그래밍을 하는 큰 이유는 컴퓨터의 계산 속도 때문이다. 반복적인 작업을 처리하는 속도가 매우 빠르기 때문에 우리는 컴퓨터를 사용한다. 동일한 내용의 반복 작업을 하는 문장을 반복문이라 부르고 반복문 역시 조건문과 비슷한 일정 조건하에서 반복을 정해진 횟수나 정해진 조건에 따라 반복될 수 있다.

7.2. if 문

if 문은 if 뒤에 조건식이 있고, 그 뒤에 조건이 참인 경우에 대한 명령 문장이 오는 구조로 만들어진다. 조건식은 일반적으로 괄호안에 표현되고, '참' 또는 '거짓'으로 판정된다. 조건식이 '참'인 경우 if 블록에 있는 문장이 실행된 후 if 블록을 빠져나간다. 조건식이 '거짓'인 경우 if 블록을 실행하지 않고 그 블록을 건너뛴다. 이후 if 블록 뒤의 내용을 실행하게 된다.

```
if (조건식)
{
    // 조건이 '참'일 때 실행될 문장들
    // 조건이 '참'일 때 실행될 문장들
}

// 조건의 '참', '거짓' 여부와 상관없는 문장들
```

if 조건식 다음에 블록의 시작을 알리는 중괄호 '{'로 조건문의 블록을 시작할 수 있다. 다만, 블록안의 문장이 한줄인 경우 중괄호 블록을 생략할 수 있다.

```
if (조건식)
```

```
// 조건이 '참'일 때 실행될 문장
```

```
// 조건의 '참', '거짓' 여부와 상관없는 문장들
```

중괄호 블록 시작이 없을 경우 if 조건문은 바로 다음에 나오는 한줄을 조건식이 참인 경우 실행할 문장으로 인식한다. 중괄호 없이 사용하면 간단한 한줄짜리 조건문을 만들 수 있다.

```
if (x<0) Serial.println("x is negative");
```

작은 값을 찾는 프로그램 코드를 만든다면 다음과 같은 조건식이 사용될 수 있다.

```
if (x <= min) min = x;
```

C언어에서 정수 0은 '거짓'으로 처리되고, 나머지 값은 '참'으로 처리된다. 이것을 이용하면 다음과 같이 쓸수 있다.

```
if (x) Serial.println("x is not zero");
if (x != 0) Serial.println("x is not zero");
```

x 가 0 일때만 '거짓'이고, 나머지는 '참'이기 때문에 x 가 0 인지 아닌지를 판단할때는 ($x \neq 0$) 또는 (x) 로 사용할 수 있다.

7.3. if-else 문

조건이 맞을 때와 조건이 틀릴 때에 별도로 실행할 명령을 구분할 때 if-else 를 사용한다. if 문이 조건이 맞은 경우에 대해서만 별도의 실행 구문을 둔 것과 구별해서 조건이 틀린 경우에 실행할 내용이 있을 경우 if-else 는 유용하다.

```
if (조건식)
{
    // 조건이 '참'일 때 실행될 문장들
    // 조건이 '참'일 때 실행될 문장들
}
else
{
    // 조건이 '거짓'일 때 실행될 문장들
    // 조건이 '거짓'일 때 실행될 문장들
}

// 조건의 '참', '거짓' 여부와 상관없는 문장들
```

다음은 if-else 를 이용하여 두 수 중 큰 값과 작은 값을 프로그램 코드이다.

```
if (a<b)
{
    min = a;
    max = b;
}
else
{
    min = b;
    max = a;
}
```

위 코드를 줄이면 다음과 같이 쓸 수 있다. 코드의 줄 수를 줄일수 있는 장점은 있지만 위의 모습이 가독성이 더 좋다.

```
if (a<b) min = a, max = b;  
else min = b, max = a;
```

이처럼 중괄호로 둘러싸인 문장들의 집합을 복합문이라 부르며, 복합문도 하나의 문장처럼 사용된다. 복합문 안에 변수와 상수를 선언할 수 있다.

```
{  
    // 변수와 상수의 선언  
    // 실행할 문장들  
}
```

7.4. 다중 if 문

다중 if 문 혹은 중첩 if 문이라고 불리는 구조를 이용하면 두가지 이상의 조건에 대한 실행이 가능해진다. 짝수여부를 묻는 첫번째 조건과 양수와 음수를 묻는 두번째 조건으로 구분하면 수는 다음과 같이 분류할 수 있다.

		x는 짝수	
		참	거짓
x는 양수	참	(1) 짝수이면서 양수 2, 4, 6, ...	(3) 짝수가 아닌 양수 1, 3, 5, ...
	거짓	(2) 짝수이면서 양수가 아님 ..., -6, -4, -2, 0	(4) 짝수가 아니고 양수가 아님 ..., -5, -3, -1

그림 108 중첩된 조건문의 예

위의 표를 다중 if 구문을 이용해 구현하면 다음과 같다. 우선 이전에 사용 하던 키보드 입력을 그대로 화면에 출력하는 프로그램 코드를 가지고 와서 변형해서 사용하도록 한다.

```
if (x%2 == 0)
{
    if (x>0)
    {
        // (1) x 가 짝수이면서 양수일 때
    }
    else
    {
        // (2) x 가 짝수이면서 양수가 아닐 때
    }
}
else
{
    if (x>0)
    {
        // (3) x 가 짝수가 아니면서 양수일 때
    }
    else
    {
        // (4) x 가 짝수가 아니면서 양수가 아닐 때
    }
}
```

PC의 키보드 입력을 시리얼 통신으로 아두이노로 보내고, 아두이노에서 처리한 내용을 시리얼 통신으로 PC로 보내서 모니터에 출력하는 프로그램에 위의 다중 if 구문을 적용해보면 다음과 같다.

```

/***********************
* nested_if_op.ino
*
* 다중 if 구문 예제, 짝수와 양수 구별
***********************/

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    char c, myChar[5];
    String myString = "";
    int inputNum;

    while(Serial.available())
    {
        c = Serial.read();
        myString.concat(c);
    }

    if (myString != "")
    {
        myString.toCharArray(myChar, 5);
        inputNum = atoi(myChar);
        if (inputNum%2 == 0)
        {
            if (inputNum>0)
            {
                // (1) inputNum 가 짝수이면서 양수일 때
                Serial.println(String(inputNum) + " : even positive");
            }
        }
    }
}

```

```

else
{
    // (2) x 가 짝수이면서 양수가 아닐 때
    Serial.println(String(inputNum) + " : even negative");
}
else
{
    if (inputNum>0)
    {
        // (3) x 가 짝수가 아니면서 양수일 때
        Serial.println(String(inputNum) + " : odd positive");
    }
    else
    {
        // (4) x 가 짝수가 아니면서 양수가 아닐 때
        Serial.println(String(inputNum) + " : odd negative");
    }
}
delay(100);
}

```

위 코드를 실행한 후 시리얼모니터로 숫자를 넣으면 동일한 값이 시리얼모니터로 출력된다. 여기서 if 구문을 살펴보면 myString이라는 변수에 아무런 값이 들어있지 않은 경우가 아니라면(값이 들어있다면), 중괄호 안의 복합문이 실행된다.

위 코드에서 if 조건문 구문을 보면 다음과 같다.

```
if (myString != "")
```

```
{  
    myString.toCharArray(myChar, 5);  
    inputNum = atoi(myChar);  
    Serial.println(inputNum);  
}
```

조건은 myString 이 "" (비어있음) 이 아니라면이고, 그때 { } 안의 복합문이 실행된다.

조건 : myString 이 ""(비어있음) 이 아니다.

다중 if 구문은 아래처럼 여러 선택지를 두고 선택할 때도 유용하다.

- 만약 성적이 90점 이상이면 , 학점은 A
- 그렇지 않고 만약 성적이 80점 이상이면, 학점은 B
- 그렇지 않고 만약 성적이 70점 이상이면, 학점은 C
- 그렇지 않고 만약 성적이 60점 이상이면, 학점은 D
- 그렇지 않으면(위 모든 경우에 해당되지 않으면), 학점은 F

					if x는 90점 이상
	참				거짓
					else if x는 80점 이상
	참				거짓
					else if x는 70점 이상
		참			거짓
					else if x는 60점 이상
		참			거짓
					else
학점 =	A	B	C	D	F

그림 109 다중 조건문 도식도

이런 경우 아래와 같이 if – else if – else 를 이용한 다중 if 구문을 사용할 수 있다.

```

if (x >= 90)
{
    score = 'A';
}
else if (x >= 80)
{
    score = 'B';
}
else if (x >= 70)
{
    score = 'C';
}
else if (x >= 60)

```

```

{
    score = 'D';
}
else
{
    score = 'F';
}

```

명령어가 한줄인 경우 중괄호를 이용한 복합문을 사용하지 않아도 된다.
즉, 위의 내용은 다음과 같이 짧게 쓸 수 있다.

```

if (x >= 90) score = 'A';
else if (x >= 80) score = 'B';
else if (x >= 70) score = 'C';
else if (x >= 60) score = 'D';
else score = 'F';

```

키보드로 성적을 입력하면 학점이 나오는 프로그램을 다중 if 구문을 이용해서 만들어 보면 다음과 같다.

```

*****
* nested_if_score_op2.ino
*
* 다중 if 구문 예제, 성적으로 학점 출력
*****

```

```

void setup()
{
    Serial.begin(9600);
}

```

```

void loop()
{
    char c, myChar[5];
    String myString = "";
    int inputNum;
    char score;

    while(Serial.available())
    {
        c = Serial.read();
        myString.concat(c);
    }

    if (myString != "")
    {
        myString.toCharArray(myChar, 5);
        inputNum = atoi(myChar);

        if (inputNum>=90)      score = 'A';
        else if (inputNum>=80) score = 'B';
        else if (inputNum>=70) score = 'C';
        else if (inputNum>=60) score = 'D';
        else                  score = 'F';

        Serial.println(String(inputNum) + " : " + String(score));
    }
    delay(100);
}

```

[실행 결과]

(키보드) 입력값	화면 출력
99	99 : A
85	85 : B
54	54 : F

그림 110 실행결과

7.5. switch 문

조건문으로 if 문과 함께 switch 문이 있다. if 조건문이 '참'과 '거짓'의 양 자택일의 선택이라면 switch 문은 다수의 경우에 대한 선택이 가능하다. 즉, switch 문은 if-else if-else if-else if... 로 된 다중 if 문 대신 사용 할 수 있다.

switch 문의 기본 형식은 다음과 같다.

```
switch (표현식)
{
    case c1 :
        명령구문들;
        break;
    case c2 :
        명령구문들;
        break;
    case c3 :
        명령구문들;
        break;
    default :
        명령구문들;
        break;
}
```

}

switch 문은 제어식 혹은 표현식이라 불리는 수식을 사용한다. 표현식의 결과는 문자형을 포함해서 정수형 수식만 사용할 수 있다. case에는 정수형 상수가 올 수 있으며 각 case 절에는 중괄호 없이 여러 문장이 올 수 있다. case 문의 마지막에 break 문을 사용한다. case 문의 마지막에 break가 없으면 그 아래로 이어서 계속 문장이 실행되므로 주의가 필요하다. 표현식이 case에 있는 상수 c1, c2, c3와 비교해서 같은 case로 가서 명령구문들이 실행된다. 만약 c1, c2, c3가 모두 표현식과 같지 않으면 default에 있는 명령구문들이 실행된다.

switch - case 문에서 사용된 break는 switch 문을 벗어나는 기능을 가진다. break를 만나면 즉시 switch에서 벗어나 그 다음 줄로 이동하여 실행된다.

기본적으로 모든 case에 대해 break 문이 필수적으로 들어가는 것이 좋다. 특별한 경우에 break 문을 의도적으로 배제시키는 경우가 있다. 아래의 경우처럼 특별한 경우를 제외하면 case는 항상 break와 묶어서 사용된다.

```
case 1:  
case 2 :  
    명령구문들;  
    break;
```

제어식의 값이 1 또는 2라면 case 1 혹은 case 2로 가서 명령구문들이 실행된다. 이때 case 1의 경우 break가 없기 때문에 case 2로 바로 넘어가서 거기 있는 명령구문들이 실행되고 그 후에 break를 만나서 실행을 멈추게 된다. 즉, 표현식이 1이거나 2이거나 상관없이 동일한 명령구문들이 실행된다.

카드의 숫자를 입력했을때 카드의 크기를 나타내는 프로그램코드를 만들어 보자. 1이면 "Ace" 가 출력되고, 2부터 10까지는 숫자 그대로, 11이면 "Jack", 12이면 "Queen", 13이면 "King" 이 출력되게 만들어보자. 13보다 큰 숫자는 입력되지 않는다.

```
*****  
* switch_op.ino  
*  
* switch 이 용한 카드 출력  
*****  
  
void setup()  
{  
    Serial.begin(9600);  
}  
  
void loop()  
{  
    char c, myChar[5];  
    int inputNum;  
    String myString = "";  
  
    while(Serial.available())  
    {  
        c = Serial.read();  
        myString.concat(c);  
    }  
    if (myString != "")  
    {
```

```

myString.toCharArray(myChar, 5);
inputNum = atoi(myChar);
switch(inputNum)
{
    case 1 : Serial.println("Ace"); break;
    case 2 :
    case 3 :
    case 4 :
    case 5 :
    case 6 :
    case 7 :
    case 8 :
    case 9 :
    case 10 : Serial.println(inputNum); break;
    case 11 : Serial.println("Jack"); break;
    case 12 : Serial.println("Queen"); break;
    case 13 : Serial.println("King"); break;
    default : break;
}
}
delay(100);
}

```

switch 문은 세로로 길어질 수 있다. 그래서 case 문을 줄이는 것이 종종 좋은 효과를 내기도 한다. case 문과 짧은 명령구문, break 까지 하나의 줄에 넣을 수 있다면 나쁘지 않다.

위의 case 문을 수정하면 다음과 같이 더 줄일 수 있다.

```

case 1 : Serial.println("Ace");      break;
case 11 : Serial.println("Jack");   break;
case 12 : Serial.println("Queen"); break;

```

```
case 13 : Serial.println("King"); break;  
default : Serial.println(inputNum); break;
```

default 를 잘 사용하면 case 2 부터 case 10 까지의 값을 한꺼번에 처리 할 수 있다.

switch 문에 사용되는 case 식별자는 정수형 상수이어야 하므로 정수형 상수가 아닌 경우는 switch 문을 사용할 수 없고 다중 if 를 사용해야한다. 기술적으로 여러가지 경우에 대한 조건을 다룰때 switch 문을 사용할 수 있다면 사용하는 것이 효율적이다.

산술 계산기

두 수를 입력받고, 덧셈, 뺄셈, 곱셈, 나눗셈을 입력받아 그 결과를 출력하는 간단한 산술 계산기를 만들어보자. 앞장에서 다루었던 문자열 함수를 사용하여 "10 + 20" 과 같은 사칙연산부호가 포함된 문자열을 받아서 두개의 피연산자와 연산자를 구분하고, 결과를 화면에 표시하는 프로그램을 만들어보자.

우선 시리얼모니터를 통한 키보드 입력을 받기 위해서 앞에서 사용한 기본 구문을 그대로 사용한다. 입력된 문자열에서 사칙연산기호에 해당하는 "+", "-", "*", "/" 의 위치를 indexOf() 함수를 사용해서 가져온다. indexOf() 함수는 문자열에 원하는 문자나 문자열이 있다면 인덱스번호를 반환하고, 없으면 -1 을 반환한다. 사칙연산에 해당하는 4가지 기호 중 최대값을 가진 값을 사용한다. 나머지는 -1 이기 때문에 버린다.

```
int plusSign = myString.indexOf("+");  
int minusSign = myString.indexOf("-");  
int multiSign = myString.indexOf("*");
```

```

int divSign = myString.indexOf("/");

int signIndex = max(max(plusSign, minusSign), max(multiSign,
divSign));

```

키보드 입력으로 받은 문자열에서 처음부터 사칙연산기호의 인덱스번호까지 첫번째 숫자로 저장하고, 사칙연산기호 인덱스 번호 다음부터 끝까지 두 번째 숫자로 저장한다.

```

myString.substring(0,signIndex).toCharArray(myChar, 7);
num1 = atoi(myChar);

myString.substring(signIndex+1, len).toCharArray(myChar, 7);
num2 = atoi(myChar);

```

switch 문을 사용해서 문자열에 있는 사칙연산부호에 따라 다른 연산을 진행한다.

```

switch(myString[signIndex])
{
    case '+' : result = num1 + num2; break;
    case '-' : result = num1 - num2; break;
    case '*' : result = num1 * num2; break;
    case '/' : result = num1 / num2; break;
}

```

위 내용을 모두 사용해서 만들어진 사칙연산 계산기 프로그램 소스는 다음과 같다. 하나의 긴 줄을 여러개의 짧은 줄로 바꿀때는 매 줄의 끝에 역슬래시 '\'를 넣어주면 된다. 즉,

```
Serial.println(String(num1)+myString[signIndex]+String(num2)+"="+String(result));
```

은

```
Serial.println(String(num1)+\
myString[signIndex]+\
String(num2)+"="+\
String(result));
```

과 동일하다. 줄이 너무 길 경우 중간중간 적당한 곳에 역슬래시를 넣어 가독성이 좋도록 여러줄로 수정할 수 있다.

```
/****************************************
* switch_op.ino
*
* switch 이 용한 카드 출력
****************************************/
void setup()
{
    Serial.begin(9600);
}
void loop()
{
    char c, myChar[7];
    int num1, num2;
    long result;
    String myString = "";
    while(Serial.available())
    {
```

```

c = Serial.read();
myString.concat(c);
}
if (myString != "")
{
    int plusSign = myString.indexOf("+");
    int minusSign = myString.indexOf("-");
    int multiSign = myString.indexOf("*");
    int divSign = myString.indexOf("/");

    int signIndex = max(max(plusSign, minusSign), max(multiSign,
divSign));
    int len = myString.length();
    myString.substring(0,signIndex).toCharArray(myChar, 7);
    num1 = atoi(myChar);
    myString.substring(signIndex+1, len).toCharArray(myChar, 7);
    num2 = atoi(myChar);

    switch(myString[signIndex])
    {
        case '+' : result = num1 + num2; break;
        case '-' : result = num1 - num2; break;
        case '*' : result = num1 * num2; break;
        case '/' : result = num1 / num2; break;
    }
    Serial.println(String(num1)+\
        myString[signIndex]+\
        String(num2)+"="+\
        String(result));
}
delay(100);
}

```

[실행결과]

키보드 입력	출력 결과 (시리얼 모니터)
10 + 10 ⁴	10+10=20
10+10	10+10=20
55 * 12	55*12=660
54-51	54-51=3

그림 111 실행결과

7.6. while 문

컴퓨터는 인간과 달리 매우 빠른 계산을 수행할 수 있다. 하지만 단순하고 순차적인 계산은 계산량이 비교적 적기 때문에 빠른 컴퓨터의 효능을 제대로 발휘하기 어렵다. C 언어를 포함한 대부분의 언어는 반복적인 작업을 할 수 있는 문법구조를 가지고 있다. while 문은 C 언어가 가지고 있는 기본적인 반복문으로 일정조건이 만족되는 범위내에서 지속적으로 일정명령을 반복하는 구조이다.

while 문은 if 처럼 조건식을 통해 그 값이 '참(TRUE)' 인지 '거짓(FALSE)' 를 계산한 뒤, 참이면 중괄호 내부의 복합문을 실행시킨다. 중괄호 내부의 복합문 실행이 완료되면 다시 조건식을 검사해서 조건식이 참인지 거짓인지를 판단하여 참이면 중괄호 내부의 복합문을 실행시키고 거짓이면 whiel 문에서 빠져나간다. 조건이 참인 한 계속 반복되는 것이 while 문의 특징이다. 아래는 이러한 과정을 보인다.

```

while (조건식)
{
    // 조건식이 참인 경우 실행될 문장들
}

```

while 문은 조건식이 참인 동안만 반복되는 구조이므로 조건식이 거짓이라면 한번도 실행되지 않게 된다.

기본적인 while 문의 사용법을 알아보기 위해 1 부터 100 까지의 수를 모두 더한 결과를 출력하는 프로그램 코드를 만들어보자. count 변수를 1부터 시작해서 2씩 증가되도록 하자. count 는 100 이하일때 while 문 안에서 result 에 count 를 계속 더하게 한다.

```

/****************************************************************************
 * sum_1_100_op.ino
 *
 * while 문을 이용한 누적합 계산
 */
void setup()
{
    int count = 1, result = 0;
    Serial.begin(9600);

    while(count <= 100)
    {
        result = result + count;
        count = count + 1;
    }

    Serial.println(result);
}

```

```
}
```

```
void loop(){}  
  

```

[실행결과]

5050

변수 count 와 result 를 각각 초기값 1과 0으로 만들어둔다. count 는 더 하려는 숫자이고, result 는 더한 결과를 저장하는 곳이다. 더하려는 숫자는 1 부터 시작하고, 더한 결과는 0 부터 시작된다.

```
int count = 1, result = 0;
```

while 문에 들어갈 조건식은 “count 가 100 이하” 이다. 그러면 더하려는 숫자인 count 가 100보다 작거나 같은 동안 조건식은 참(true)이 되고, 조건식인 참인 동안 while 블록에 있는 명령행들이 실행된다.

while 블록은 while(조건식) 뒤에 중괄호로 묶여 있는 부분이다. 여러 명령 줄이 함께 있는 복합문이 일반적이지만 하나의 행으로 이루어질 수도 있다. 이때에만 특별히 중괄호없이 사용할 수 있다.

```
while(count <= 100)  
{  
    result = result + count;  
    count = count + 1;  
}
```

count ≤ 100 인 조건에 맞는 동안 result 에 result + count 를 넣고, count 를 1 증가시켜준다. 이렇게 조건식이 참(true)인 동안 실행된 내용을

중괄호 안에 넣어 준다. 만약 한줄이라면 중괄호를 다음과 같이 생략할 수 있다. `result = result + count` 와 `count = count + 1` 을 대입연산자와 증감연산자를 이용하여 다음과 같이 한줄로 줄일 수 있다.

```
while(count <= 100)
    result += count++;
```

`result += count++;` 을 `while(count <= 100)` 옆으로 붙여서 써도 된다.

```
while(count <= 100) result += count++;
```

이렇게 하면 5줄 이었던 while 문이 1줄로 줄어든다. 프로그램의 줄수가 많아서 보기 힘들때 이렇게 줄수를 줄이면 도움이 된다. 단 위의 경우 대입연산자와 증감연산자가 하나의 수식에 같이 들어있기 수식을 한눈에 바로 알아보기 어려워진다. 이렇게 줄일수는 있지만 추천하지는 않는다. 짧아지는 줄 수로 인한 효율성보다 증감연산자가 대입연산자와 함께 있어서 생기는 가독성이 나빠지는 비효율성이 더 크기 때문이다. 위의 식은 충분히 검토해서 사용한 문제없는 식이지만 만약 아래처럼 쓰여져 있을때 어떤 결과가 나올지 바로 파악하기가 어려워지고, 문법적으로 아무런 문제가 없는 문장이기 때문에 오류가 발생하지 않는다.

```
while(count <= 100) result += ++count;
```

이런 오류를 최소화하기 위해 가급적 증감연산자는 대입연산자와 함께 사용하는 것을 삼가하는 것이 좋다. 생각보다 위와 같은 오류는 전문적인 프로그래머들 사이에서도 종종 발생하는데 이러한 논리오류는 가능한 발생되지 않도록 프로그래밍 하는 것이 좋다.

그래서 while 문은 다음을 참고하여 줄수는 늘어나더라도 증감연산자와 대

입연산자가 하나의 수식에 포함되지 않게 하는 것이 좋다.

```
while(count <= 100)
{
    result += count;      // (1) result 에 count 를 더해준다.
    count++;              // (2) count 에 1을 더한다.
}
```

약간의 프로그래밍 팁을 말한다면 C 에서는 세미콜론 ';' 을 만나기 전까지 문장을 끝이 아니므로 위의 코드는 줄바꿈과 띄어쓰기를 살짝 조정하면 다음과 같다.

```
while(count <= 100) { result += count; count++; }
```

위 두 while 문은 완벽하게 동일한 문장이다. 주석은 컴파일러에서 읽지 않는 부분이고, 눈에 보이는 엔터키를 쳐서 보이는 줄바꿈도 컴파일러에서는 별 의미가 없다. C 컴파일러는 세미콜론 ';' 을 만날 때 명령의 끝을 인식한다. 그 점을 활용하면 위처럼 여러 줄을 하나의 줄에 쓸 수 있다. 위의 경우는 짧은 2개의 줄을 하나로 만들면서 세미콜론으로 분리되어있기 때문에 가독성 면에서도 4줄을 사용하는 것과 비교해서 많이 떨어지지는 않는다. 짧은 코드 2줄로 된 명령줄은 위처럼 하나의 줄에 세미콜론을 사용하여 함께 사용하는 방법도 있다.

즉,

```
while(count <= 100) result += count++;
```

보다는

```
while(count <= 100) { result += count; count++; }
```

을 사용하기를 추천한다.

합과 평균 구하기

학생들의 키를 입력받아 합과 평균을 구하는 프로그램을 만들어보자. 키는 모두 정수 세자리수로 이루어지고, 0 을 입력 받으면 입력이 종료된다. 합은 들어온 모든 입력값을 더하면 되고, 평균은 들어온 입력값을 사람의 숫자로 나누면 된다. 이때 0 을 입력받은 것은 제외해야 한다. 계속되는 입력 값이 있기 때문에 loop() 함수를 사용할수도 있지만 여기서는 setup() 함수 안에서 while 문을 사용하기로 하자.

코드를 잘 이해해 보기 바란다. 반드시 코드를 손으로 직접 타이핑해보고 결과를 확인하면서 다른 방법이 없는지 생각해보고 제시된 것과 다른 다양한 방법으로 코드를 만들어볼 것을 권한다.

코드를 실행하면 화면에 "Input the height => " 라는 문자열이 출력된다. 이후 키를 정수형으로 입력하고 엔터키를 치면 각 사람의 키가 입력되고, 최종적으로 0을 입력하면 입력이 끝난다. 모든 키의 합과 키의 평균이 출력된다.

책의 3장에서 아두이노는 main() 함수가 없는 것이 아니라 숨겨져 있으며, main() 함수 내부에 1회 실행될 setup() 과 반복 실행될 loop() 가 있음을 알고 있다. setup() 만을 사용하여 프로그래밍 해보자.

`setup()` 함수 내에 `while()` 문을 사용해서 들어오는 입력값을 계속 더하고, 입력값이 0 인 경우 반복문을 그치는 코드를 만들었다.

```
setup()
{
    // 시리얼통신 시작
    while(num)
    {
        // PC 키보드로 입력 받은 값을 시리얼통신을 통해 가져온다.
        // 0 이 입력되면 반복문은 종료
        // 입력된 값을 result 에 저장
        // 입력 횟수를 count 에 저장
    }
    // result 와 result / count 를 출력
}
```

시리얼통신으로 입력값이 들어오기까지 기다리는 코드는 다음과 같다.

```
while (!Serial.available())
{
}
```

`Serial.available()` 함수는 시리얼 버퍼에 값이 담겨있는지를 검사하는 함수다. 즉, 시리얼통신으로 어떤 값이 수신되지 않았다면 거짓(false)인 0이 나오고, 수신된 값이 있다면 참(true)이 나온다. `!Serial.available()` 는 논리 NOT 연산자를 사용해서 "시리얼 수신 버퍼가 수신되지 않았다면"의 의미가 된다. 즉, PC로 부터 아무런 신호를 받지 않은 상태라면 `while` 문의 블록 내부를 실행시키라는 뜻이다. `while` 블록은 아무것도 채워져 있지 않으므로, 아무런 실행도하지 않고 시간을 보내게 된다. PC에서 어떤 신

호가 시리얼통신으로 들어오게 되면 조건식 `!Serial.available()` 이 거짓 (false)이 되어 while 반복문을 빠져나온다. 위 문장을 한줄로 단축시킨 것이 다음 문장이다. 참고로 아래와 같이 쓸 경우 팔호의 종류를 잘 파악하는 것이 중요하다. 맨 끝의 팔호가 중괄호 "{}"이고 나머지 앞부분의 팔호는 일반괄호 "()"이다.

```
while (!Serial.available()) {}
```

시리얼 버퍼에 신호가 있을 때 그 신호를 받아들이는 코드는 다음과 같다.

```
String myString = "";
while (Serial.available())
{ c = Serial.read(); myString.concat(c);}
```

앞에서 시리얼 버퍼가 비어있을 때 잠시 멈춰있는 코드는 시리얼 버퍼에 신호가 하나라도 들어오면 바로 벗어나게 된다. 즉, while (`!Serial.available()`) {} 코드는 시리얼 버퍼에 첫번째 신호가 들어오는 순간 더 이상 참이 아니기 때문에 while 반복문을 종료하고 그 다음 줄로 넘어가게 된다. 이때 시리얼 버퍼에 수신되는 신호가 쌓이는 시간과 실행되는 시간을 비교하면 신호가 쌓이는 시간보다 실행되는 시간이 더 빨라서 이로 인한 원치 않는 동작이 진행될수 있다. 즉, 키보드를 통해 PC에서 아두이노로 전달되는 신호는 "174" 인데, 이중 "1"이 들어간 상태에서 아래 코드가 실행되게 된다.

```
while (Serial.available())
{
    c = Serial.read();
    myString.concat(c);
}
```

1	<code>Serial.available()</code>	시리얼 버퍼에 신호가 들어와 있으므로 참(true)이므로 아래 블록 내용을 실행한다.
2	<code>c = Serial.read();</code>	시리얼 수신버퍼에 있는 한 바이트를 읽어 c에 저장하고, 수신버퍼에서 읽은 한 바이트를 지운다.
3	<code>myString.concat(c);</code>	c에 저장된 값을 myString에 붙여넣는다.

그림 112 시리얼 흐름

1 -> 2 -> 3 단계를 거치면서 처음 들어온 값 "1"을 읽고 버리게 된다. 이때 읽고 버리는 시간이 시리얼포트를 통해 두번째 값 "7"을 읽어오는 시간보다 빠르게 되면 위의 코드는 시리얼 창을 통해 "174"를 입력했지만 "1"만을 읽고, 수신 버퍼가 비어있는 상태가 되어 반복문에서 빠져나가게 된다. 결과적으로 이로 인해 원치 않는 결과가 발생한다.

즉 시리얼 통신을 속도에 따라 "1", "7", "4" 가 들어오는 시간과 들어온 버퍼를 처리하는 시간의 차이를 비교해서 후자가 더 빠른 경우 키보드로 입력한 "174"는 "1"만을 처리한 채 끝나게 된다. 아래의 ①의 경우와 같다.

시리얼통신	"1" 전달		"7" 전달		"4" 전달					
①	"1" 처리	??	??							
②			delay, 시간지연				"1" 처리	"7" 처리	"4" 처리	

그림 113 시리얼통신 흐름 도식

그래서 시리얼 버퍼에 일정한 데이터가 모일때까지 시간지연을 둔 다음 처리를 하는 ② 방식을 사용한다. 시리얼로 "174"가 들어올 때 약간의 delay를 주면 그 동안 버퍼에 신호가 모이게 되고 그 후에 버퍼에 모인 값들을

읽어서 처리한다. delay(200) 은 0.2 초 동안 시간지연을 두고 그 후에 버퍼에 모인 값을 읽게 된다.

```
delay(200);
```

시간지연 후 myString 변수에 시리얼 수신 버퍼에 담긴 값을 저장한다. myString 변수는 매번 실행할 때마다 새롭게 만들어지도록 로컬변수로 만든다. while 블록 내부 로컬변수가 되면 while 을 매번 실행할 때마다 초기화된다.

```
String myString = "";
while (Serial.available())
{
    c = Serial.read();
    myString.concat(c);
}
```

while 문을 짧게 한줄로 만들어 보면 다음과 같다.

```
while (Serial.available()) { c = Serial.read(); myString.concat(c); }
```

while 문 안에서 result 와 count 값을 다음과 같이 변경한다.

```
result = result + num;
count++;
```

전체 프로그램 소스는 다음과 같다. 실행시킨 후 시리얼모니터 창에서 키를 입력하고, 마지막에 0 을 입력하면 전체 합과 평균을 출력한다.

```

*****
* while_height_op.ino
*
* while 문을 이용한 키의 합과 평균 구하기
*****
void setup()
{
    char c, myChar[5];
    int num=1, result=0, count=0;
    Serial.begin(9600);

    Serial.println("Input the height => ");

    while(num)
    {
        while (!Serial.available()) {} // wait 200 ms until input
        delay(200);
        String myString = "";
        while (Serial.available()) { c = Serial.read();
        myString.concat(c);}
        myString.toCharArray(myChar, 5);
        num = atoi(myChar);
        result = result + num;
        count++;
    }
    Serial.println("Sum : "+String(result));
    Serial.println("Average: "+String(result/float(count-1)));
}

void loop(){}

```

[실행결과]

입력	출력
170 180 174 0	Input the height => Sum : 522 Average: 174.66

그림 114 실행결과

7.7. do-while 문

while 문은 조건이 루프의 시작에 위치하기 때문에 조건을 검사한 후 조건이 참인 동안 반복문이 실행된다. 만약 조건을 검사했을 때 조건식이 거짓이라면 반복문은 한번도 실행되지 않는다. 반면 do-while 문은 종료 조건을 본체의 마지막 부분에 두어 조건식의 참, 거짓을 검사하게 된다.

검사할 조건식의 중요한 변수들이 반복문안에 포함되어 있을 경우 do-while 을 쓰면 보다 깔끔하게 코드가 만들어진다. while 문으로 작성할 수 있는 코드를 굳이 do-while 을 써서 만들 필요는 없지만 다음과 같은 구조를 가지고 있다면 do-while 을 쓰는 것이 좋다.

```
int a = 1;
```

명령문A

```
while (a>0)
{
    명령문A
    a = func();
}
```

프로그래머는 명령문A를 실행하고, 이후에 같은 명령문A를 반복문의 조건식에 따라 반복하여 실행할 것인지 결정하게 한다. 그래서 반복문의 외부에 명령문 A를 작성하고, 반복문 내부에도 같은 코드를 다시 한번 더 작성하게 된다. 이런 코드는 같은 명령문을 반복문의 외부에 내부에 중복해서 작성하게 되기 때문에 실수가 만들어진 확률이 커지고 이후 유지보수에 비효율적인 코드가 만들어진다. 이와 같은 문제가 발생할 수 있기 때문에 이런 경우에는 단순한 while 문이 아닌 do-while 문을 사용하는 것이 좋다. do-while 을 사용하면 중복이 없어지고 보다 논리적이고 구조적인 프로그래밍이 된다.

```
int a = 1;  
  
do  
{  
    명령문A  
    a = func();  
}  
while (a>0);
```

여기서 명령문A는 한번만 나온다. while 조건식이 만족되지 않더라도 한번은 실행된다. 앞의 경우와 비교하면 같은 결과가 나오지만 작성된 코드는 중복이 제거되었으므로 보다 깔끔하고 보기 편한 구조를 가지게 된다.

간단히 정리하면 while 문은 "조건식을 실행할 후 반복을 결정"해야 하는 경우에 사용되고 do-while 문은 "반복을 시작한 후 조건식을 실행"할 경우에 사용된다고 볼수 있다.

주의할 것은 do-while 문의 경우 "while(조건식);;" 처럼 마지막에 세미콜론이 붙는다.

다음은 간단한 do-while 예제로 0부터 9까지 출력하는 프로그램 코드다.

```
void setup()
{
    int i = 0;
    Serial.begin(9600);
    do
    {
        Serial.println(i);
        i++;
    }
    while (i<10);
}
void loop(){}  
 
```

[실행결과]

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
 
```

7.8. for 반복문

while 과 함께 자주 사용되는 반복문이 for 문이다. 일반적으로 while 은 반복횟수가 정해져 있지 않을 때 사용되고, 반복횟수가 정해져 있을 때는 for 를 사용한다.

for 문은 다음과 같이 세개의 수식을 세미콜론으로 구분해서 사용한다.

```
for (수식1 ; 수식2; 수식3)
{
    // 실행될 명령들
}
```

수식1 은 초기화 부분으로 반복을 시작하기 전에 실행된다. 일반적으로 수식1 에는 이후 변경될 변수의 초기값을 넣어준다. 수식2 는 불린(boolean) 조건식으로 반복문 안에 있는 명령들을 실행하기 전에 계산하여 조건이 참(true)이면 내부 명령들을 실행하고 거짓이면 for 반복문에서 벗어난다. 수식3 은 조건식에 사용된 변수를 증가시키거나 감소시키기 위해 사용되는 부분으로 내부 명령어들이 모두 실행된 후에 실행된다.

for 문의 실행 순서는 다음과 같다.

1. 수식1을 수행한다.

2. 수식2의 참과 거짓을 확인한다.

수식2가 참이라면 -> 중괄호 안의 명령어들을 실행한다.[3으로 간다]

수식2가 거짓이라면 -> for 문을 벗어난다.[종료]

3. 수식2가 참이고 중괄호 안의 명령어들이 모두 실행되었다면, 수식3을 실행한다.[2로 간다]

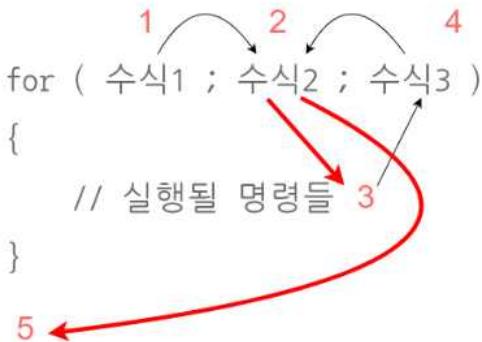


그림 115 for 실행순서

for 문을 이용해서 1부터 10까지의 정수의 합을 계산하는 간단한 코드를 만들어보면 다음과 같다.

```

int sum = 0;
for (int i=1; i<=10; i++)
{
    sum = sum + i;
}

```

위와 같이 간단한 하나의 문장 혹은 한줄로 코드가 표현될 경우 이전의 if 나 while 과 같이 중괄호를 생략할 수 있다.

```

int sum = 0;
for (int i=1; i<=10; i++) sum = sum + i;

```

for 반복문에서 3개의 수식은 생략할 수 있다. 두번째 수식이 생략되었을 때는 조건식은 항상 참이 된다. 즉 for(;;) 는 while(1) 과 동일하다.

for 를 이용해서 7의 배수를 10개 출력하는 프로그램을 만들어보자. 초기 값으로 정수형 변수 i 를 만들고 1을 넣어준 다음, 1씩 더하면서 11이 되

기 전까지 총 10번 반복하면 변수 i 는 1부터 10까지의 값을 가지게 된다.
여기에 7을 곱해서 출력한다.

이것을 코드로 만들어보면 다음과 같다.

```
*****  
* for_seven_op.ino  
*  
* for 문을 이용하여 7을 배수 출력  
*****  
  
void setup()  
{  
    Serial.begin(9600);  
    for (int i = 1; i<11; i++)  
    {  
        Serial.print(String(i*7)+" ");  
    }  
}  
  
void loop(){}
```

[실행결과]

7 14 21 28 35 42 49 56 63 70

for 와 while 의 관계

for 문과 while 문은 어느 한쪽을 사용해서 다른 쪽을 만들수 있다. 그래서 프로그래머들은 for 혹은 while 중 어느 한쪽을 더 자주 사용한다.

```
for (초기화식; 조건식; 증감식)
{
    명령문들;
}
```

위의 for 문은 아래의 while 문으로 사용할 수 있다.

```
초기화식;
while(조건식)
{
    명령문들;
    증감식;
}
```

7.9. 중첩 루프, 무한 루프

중첩 루프

반복문 안에 다른 반복문이 포함될 때 이것을 중첩루프(nested loop)라고 부른다. 반복문을 루프(loop)라고도 부른다. while 안에 while 이 포함되거나 for 안에 for 가 포함되는 식으로 중첩루프가 구성된다. 바깥쪽 반복문

의 개별 반복에 대해서 안쪽 반복문이 실행되기 때문에 2차원적인 반복작업을 수행할 수 있다.

for 반복문을 사용한 구구단 프로그램은 중첩루프를 다뤄보는 좋은 예제다. 바깥쪽 반복은 변수 i를 사용, 2부터 9까지 반복하게 하고 안쪽 반복은 변수 j를 가지고 1부터 9까지 반복하게 한다. 총 반복 횟수는 $8 * 9$ 인 72회가 된다.

```
*****  
* for_multiply_op.ino  
*  
* for 문을 이용한 구구단 출력  
*****  
  
void setup()  
{  
    Serial.begin(9600);  
    for (int i=2; i<10; i++)  
    {  
        for (int j=1; j<10; j++)  
        {  
            Serial.print(i);  
            Serial.print("*");  
            Serial.print(j);  
            Serial.print("=");  
            Serial.println(i*j);  
        }  
    }  
}  
void loop(){}  
*****
```

위의 식을 줄이면 다음과 같이 쓸수 있다.

```

*****
* for_short_multiply_op.ino
*
* for 문을 이용한 구구단 출력
*****
void setup()
{
    Serial.begin(9600);
    for (int i=2; i<10; i++)
        for (int j=1; j<10; j++)
            Serial.println(String(i)+"*"+String(j)+"="+String(i*j));
}
void loop(){}

```

무한루프

반복문은 언젠가는 끝나야한다. 즉 반복문이 끝나지 않는 무한루프에 빠진다는 것은 프로그래머가 코드를 잘못 만들었다는 것을 의미한다. 반복문이 영원히 반복해서 실행되는 상황을 무한루프라고 부르는데 루프의 조건이 항상 참인 상황이 무한루프가 된다. 다음의 무한루프는 전원이 꺼지지 않는 한 계속해서 반복하게 된다.

```
while(1) Serial.println("infinite loop");
```

아래의 경우는 count 값을 1씩 증가시키며 1부터 N 까지의 값을 더해서 출력하는 프로그램이다.

```

*****

```

```

* initfinite_op.ino
*
* 무한루프
*****void setup()
{
    Serial.begin(9600);
    int sum=0, count=0;
    while(count <= 10)
    {
        sum += count;
        count++; // count--
        Serial.println(sum);
    }
}
void loop(){}

```

while 문의 count++ 를 count-- 로 쓰게 되면 이때 while 은 무한 루프에 빠지게 된다. 이런 논리적 오류는 문법 오류보다 찾기 어렵다. 논리 오류로 만들어지는 무한 루프는 발견하기 어렵다.

7.10.break, continue

switch 문에서 빠져나오는 방법으로 break 를 사용했다. break 는 반복문 안에서 강제종료를 할 때도 사용된다. 반복문을 마치는 방법은 조건문의 조건식이 거짓(false, 0)이 되는 것과 함께 조건문안에서 break 가 실행되는 경우도 포함된다. 다음 코드에서 break 문은 읽은 값이 0 이면 루프를 끝내게 된다.

```
while()
```

```
{  
    // 명령문들  
    if (i==0)  
        break;  
    result = input/i;  
}
```

break 문을 사용하는 것이 꼭 필요한 것이 아니라면 사용을 절제하는 것이 좋다. switch 문에서의 break 는 필요하기 때문에 사용해야 하지만 while이나 for 반복문에서는 break 없이 조건문과 내부 명령어들을 잘 사용하면 break 문의 사용을 줄이거나 없앨 수 있다. break 는 기본 구조를 벗어나 강제적인 개입이 되는 명령이므로 가능한 줄이는 것이 나중 유지보수를 위해 좋다.

continue 문은 루프에서 벗어나지 않고 루프의 처음 부분으로 돌아간다. 루프 조건을 다시 판단한 후 조건이 참이면 명령을 처음부터 실행한다. break 와 같이 continue 문도 많이 사용하는 것은 프로그램의 구조를 난해하게 만들기 때문에 자제하는 것이 좋다.

```
while()  
{  
    i++;  
    if (i % 2 == 0)  
        continue;  

```

위의 코드는 홀수만 계속 더하는 프로그램 구문이다. 짹수인 경우 continue 명령으로 이후의 명령을 실행하지 않고 반복문의 앞부분으로 돌아가게 한다. 전체 소스코드는 다음과 같다. 10 이하의 홀수를 더하는 프로그램을 continue 를 이용해서 만들어 본 예제이다.

```

*****
* continue_op.ino
*
* continue 예제
*****
void setup()
{
    Serial.begin(9600);
    int sum=0, i=0;
    while(i < 10)
    {
        i++;
        if (i % 2 == 0) continue;
        sum += i;
    }
    Serial.println(sum);
}

void loop(){}

```

7.11. goto

ANSI C 표준에 속해있기는 하지만 goto 문은 현재 프로그래밍에서 거의 사용되지 않는다. 즉, 이후로 설명하는 goto 문은 문법적으로만 이해하고 있고, 누군가 사용했을때 의미를 파악하는 정도로만 알아주기를 바란다. 실제로 C언어에서 goto 문을 사용하는 경우는 거의 없다.

C에서 사용할 수 있는 낮은 수준의 제어문이 goto 문이고, 다음의 형태를 가진다.

```
goto LABEL;  
...  
LABEL:  
// 명령어들
```

goto 문은 구조적이지 않아서 goto 문을 사용하게 되면 프로그램의 구조가 쉽게 망가질수 있다. 이 책은 goto 문을 사용하지 않으며, 필자 주변에 goto 문을 사용하는 사람을 본적이 없음을 밝힌다. 아주 특별한 경우, 시스템 프로그래밍이나 디바이스 드라이버 등을 다루면서 가독성보다 성능을 최우선으로 할때 goto 는 극히 제한적으로 사용된다. goto 는 어셈블러에서 주소번호로 점프하는 것과 같고, 이는 컴퓨터나 아두이노등이 이해하는 가장 기본적인 방식이다.

아래에 goto 문을 사용한 예제가 있다. 이 예제를 살펴보고 한번 따라해보되 가급적 goto 문을 사용하지 않기를 바란다.

```
*****  
* goto_op.ino  
*  
* goto 예제  
*****  
void setup() { Serial.begin(9600); }  
void loop()  
{  
    int num = 0; char myData[5];  
  
    //シリ얼로 키보드 입력 받기  
    char c; String myString = "";  
    while(Serial.available())  
    { c = Serial.read(); myString.concat(c); }
```

```

delay(200);

// 문자열을 숫자정수로 변환
if (myString != "")
{
    myString.toCharArray(myData, 5);
    num = atoi(myData);
}

// goto 예문
if      (num==1)  goto LABEL_ONE;
else if (num==2)  goto LABEL_TWO;
else          goto LABEL_EXIT;

LABEL_ONE:
Serial.println("No.1 Processing");
goto LABEL_EXIT;

LABEL_TWO:
Serial.println("No.2 Processing");
goto LABEL_EXIT;

LABEL_EXIT:
delay(100);
}

```

시리얼로 정수 입력하기 예제 :

```

void setup() { Serial.begin(9600); }
void loop() {
    if (Serial.available()) {

```

```
    int value = Serial.parseInt();  
}  
}
```

문제

1. 세개의 막대가 있다, 각 막대의 길이는 a, b, c 일때 이 3개의 막대를 사용해서 삼각형을 만들수 있는지 여부를 프로그래밍하라.

(삼각형의 조건 : 세변의 길이 중 가장 큰 변의 길이가 나머지 두변을 더 한 것보다 작아야 한다)

a, b, c 가 각각 다음과 같을때 삼각형을 만들수 있는지 판단하는 코드를 만드세요.

- 1) 10, 15, 20
- 2) 10, 10, 10
- 3) 10, 20, 30
- 4) 10, 12, 12

Chapter 08

주요 학습 목표

- 아두이노 실드에 대해 이해한다
- 실드를 이용한 프로그래밍을 한다

Chapter 08. 아두이노 실습환경

이전 장까지 C 언어의 기본내용들을 살펴보았다. 이 책의 전체적인 내용을 따라가기 위해서 아두이노 우노 혹은 아두이노 메가 2560 을 사용하면 된다. 아두이노, MCU 의 특별한 하드웨어를 다루어야한다. 여기서는 아두이노와 어울리는 몇개의 아두이노 쉴드(Arduino Shield)를 소개한다. 쉴드는 원보드에 연결해서 성능을 확장시키거나 특정목적을 위한 부가적인 기능을 위해 만들어진 보드를 의미한다.

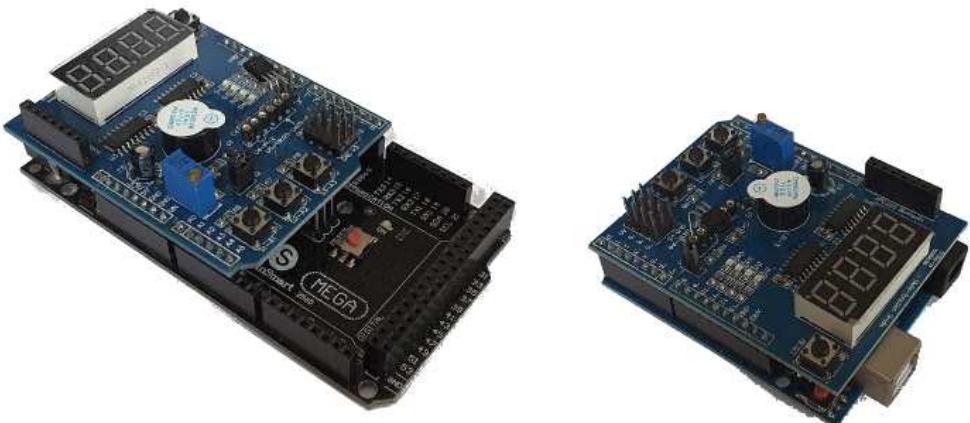


그림 116 아두이노 메가2560과 우노 + MFS

아두이노를 공부할 때 일반적으로 아두이노와 몇가지 센서들을 키트로 사용한다. 센서와 아두이노를 브래드보드에 연결하고 아두이노에 프로그램을

넣어서 실습을 진행하는 것이 아두이노를 공부하는 기본적인 방법이다. 다만 이렇게 하드웨어를 주로 다루게 되면 그만큼 소프트웨어, 언어 자체에 대한 비중이 소홀해지게 된다. 그래서 이 책은 기본 아두이노와 브레드보드와 듀얼케이블을 이용한 연결을 최소화하기 위해 이미 판매되고 있는 아두이노 쉴드를 사용한다.

아두이노는 우노 혹은 메가 2560 을 사용하고, 쉴드로는 멀티펑션쉴드를 사용한다. 쉴드는 쉴드를 만든 제조사에서 제공하는 라이브러리가 있을수 있으며, 그 라이브러리를 이용하는 방법도 살펴보도록 한다. 여기서부터는 아두이노 시뮬레이터를 이용한 코딩이 불가능해진다, 라이브러리에 포함된 코드는 tinkercad 에서 작동하지 않는다. 아두이노 IDE 에서 라이브러리를 포함시키고 아두이노에 쉴드로 올린 상태에서 작동한다.

8.1. Multi Function Shield

멀티펑션 쉴드(Multi Function Shield)는 이름에 붙어있는 그대로 다양한 기능을 제공한다. 대부분의 쉴드가 특정한 기능에 전문화되어 있는 반면 멀티펑션쉴드는 교육용으로 아두이노와 C언어를 익히기에 적합하게 만들어져 있다. 버튼 입력, LED 디스플레이 출력, 부저를 이용한 알람등 기본적인 IO 기능을 아두이노에서 사용할 수 있도록 하나의 보드에 넣었다.

멀티펑션쉴드를 이용하면 아두이노로 C언어를 익힐 때 하드웨어 세팅에 많은 시간을 들이지 않고 프로그래밍을 공부할 수 있다. 멀티펑션쉴드를 이용한 아두이노 학습을 진행해 본 경험으로 브레드보드와 센서들을 듀얼케이블로 연결해서 학습을 진행했던 것 보다 안정적이고 임베디드 시스템을 보다 깊이 가르칠 수 있었다.

멀티펑션쉴드의 사용법에 대한 유튜브 영상 강의가 함께 제공된다. 책의 부록에 있는 링크를 통해 필요한 영상강의를 무료로 시청할 수 있다.

멀티평선쉴드 라이브러리 설치

TimerOne 라이브러리 설치

멀티평선쉴드의 라이브러리를 설치하기 전에 우선 아두이노에서 TimerOne 라이브러리를 설치한다. 멀티평선쉴드는 아두이노의 인터럽트 Timer1 을 사용하고, 이를 제대로 사용하기 위해서 TimerOne 라이브러리를 설치해야 한다.

우선 아두이노 IDE 를 열고, [툴] 에서 [라이브러리 관리]를 클릭한다.

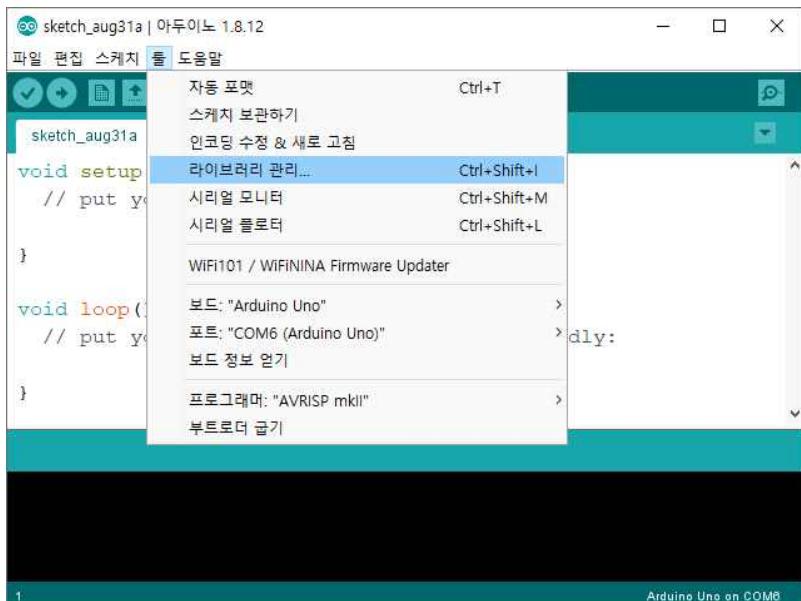


그림 117 아두이노 IDE 라이브러리 관리 메뉴

[라이브러리 매니저] 창이 화면에 뜨면 검색창에 TimerOne 을 키보드를

친다. 그러면 아두이노 서버에 있는 많은 라이브러리 중 TimerOne 검색 결과가 나온다. 여기서 "TimerOne" 을 마우스로 클릭하여 설치를 진행한다.

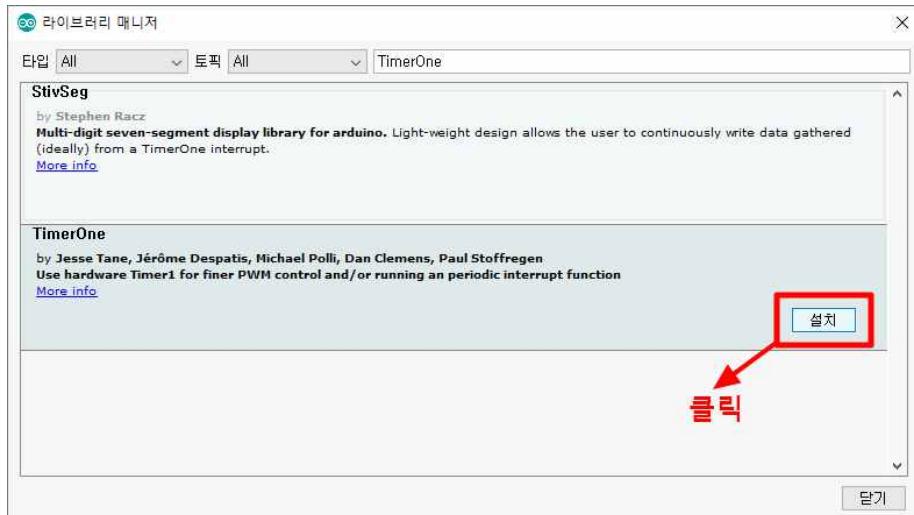


그림 118 라이브러리 매니저

검색되어 나온 TimerOne 위에 마우스를 대면 [설치] 버튼이 나온다. TimerOne 을 설치한다.

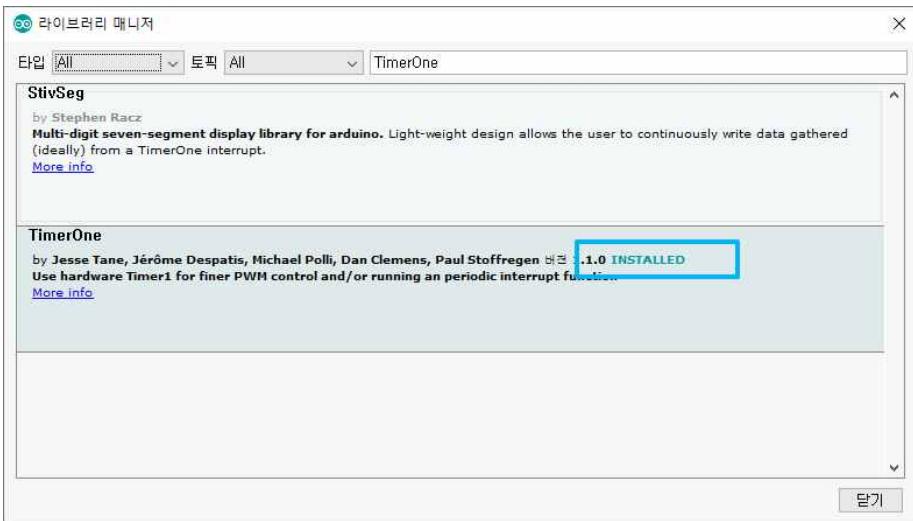


그림 119 타이머1 라이브러리 설치

설치가 끝나면 [INSTALLED]라는 표시가 나온다. 설치가 완료 되었으니 오른쪽 아래에 있는 [닫기] 버튼을 눌러 창을 닫는다.

이 책에서는 [툴] – [라이브러리 관리]로 들어가서 TimerOne 을 설치하였지만 [스케치] – [라이브러리 포함하기] – [라이브러리 관리]로 들어가도 된다. 두가지 방법 모두 [라이브러리 관리] 메뉴를 호출한다.

멀티평션라이브러리 설치

멀티평션쉴드 라이브러리는 TimerOne 처럼 서버에 등록되어 있지 않아 별도로 파일을 다운받은 후 설치해야 한다.

멀티평션쉴드 라이브러리는 몇군데서 만들어진 것이 있고, 인터넷에서 구글 검색을 통해 어렵지 않게 찾을 수 있다. 이 책에서는 "Cohesive

Computing"에서 제공한 라이브러리를 사용하기로 한다. 라이브러리 파일의 링크는 https://files.cohesivecomputing.co.uk/MultiFuncShield-Library-1_3.zip 이고, 관련 사이트 링크는 <https://www.cohesivecomputing.co.uk/hackatronics/arduino-multi-function-shield/> 이다. bit.ly를 이용한 짧은 주소는 다음과 같다.

라이브러리 파일 옆의 주소에서 파일을 다운받는다. 혹은 구글링을 통해 직접 다운받는다.

라이브러리 파일 : bit.ly/34R9VB1
사이트 주소 : bit.ly/2YQWzB8

다운받으면 "MultiFuncShield-Library-1_3m.zip" 파일이 다운받아진다. 이 글을 쓰는 시점에서 버전 1.3이 최신이고, 이 책에서 다루는 버전이 1.3이므로 가급적 같은 버전의 라이브러리를 사용할 것을 권장한다.

파일을 다운받은 곳의 위치를 기억해 둔 다음 아두이노 IDE를 열어서 [스케치] – [라이브러리 포함하기] – [ZIP 라이브러리 추가]를 클릭한다.

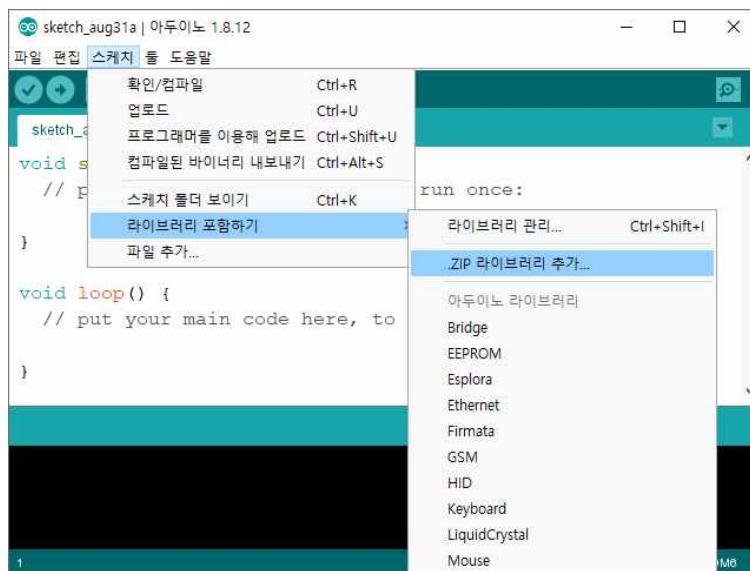


그림 120 라이브러리 포함

나오는 창에서 ZIP 파일이 있는 위치를 찾아서 "MultiFuncShield-Library-1_3m.zip" 파일을 열어준다.

[TIP] 참고로 ZIP 으로 압축된 라이브러리 파일은 다음의 포맷에 따라 압축이 되어 있다.

```
압축파일이름.zip
+---- 라이브러리이름
+---- filename1.ext
+---- filename2.ext
+---- filename3.ext
+---- ...
+---- filename1.ext
```

그림 121 압축파일 내부

Multi Function Shield Library 는 1.3 버전까지 나왔는데 1.3 버전의 경우 압축파일로 만들어지면서 바로 ZIP 파일로 설치하기 어렵게 되어 있다. 이미 이전 버전을 사용하고 있던 사람들은 압축을 풀어 내부에 있는 MultiFuncShield.cpp 파일과 MultiFuncShield.h 파일을 기존 라이브러리가 설치된 디렉토리에 풀어주면 된다. 하지만 처음 1.3 버전을 설치하는 사람은 위에 수정된 "MultiFuncShield-Library-1_3m.zip" 파일을 다운받아서 설치하거나 혹은 "MultiFuncShield-Library-1_3.zip" 파일의 압축을 푼 다음 다시 위의 구조로 재압축해서 사용하면 된다. 즉, 아래처럼 상위 하나의 디렉토리를 제거하고 다시 ZIP 으로 압축하면 된다.

[기존 라이브러리 압축파일]
MultiFuncShield-Library-1_3.zip

[PATH]
+-- MultiFuncShield-Library-1_3
 +-- MultiFuncShield-Library-1_3
 +--- MultiFuncShield-Library
 +--- I2C.cpp
 +--- I2C.h
 +--- II2C.h
 +--- I2C.cppkeywords.txt
 +--- I2C.cpplibrary.json
 +--- I2C.cppMPU6050.cpp
 +--- I2C.cppMPU6050.h
 +--- I2C.cppMultiFuncShield.cpp
 +--- I2C.cppMultiFuncShield.h

[수정된 라이브러리 압축파일]
MultiFuncShield-Library-1_3m.zip

[PATH]
+-- MultiFuncShield-Library-1_3m
 +--- MultiFuncShield-Library
 +--- I2C.cpp
 +--- I2C.h
 +--- II2C.h
 +--- I2C.cppkeywords.txt
 +--- I2C.cpplibrary.json
 +--- I2C.cppMPU6050.cpp
 +--- I2C.cppMPU6050.h
 +--- I2C.cppMultiFuncShield.cpp
 +--- I2C.cppMultiFuncShield.h

그림 122 압축파일 내부 수정

만약 위 과정이 복잡하게 여겨진다면 위에서 링크로 알려준 패스가 수정된 라이브러리("MultiFuncShield-Library-1_3m.zip")를 바로 이용할 것을 권한다.

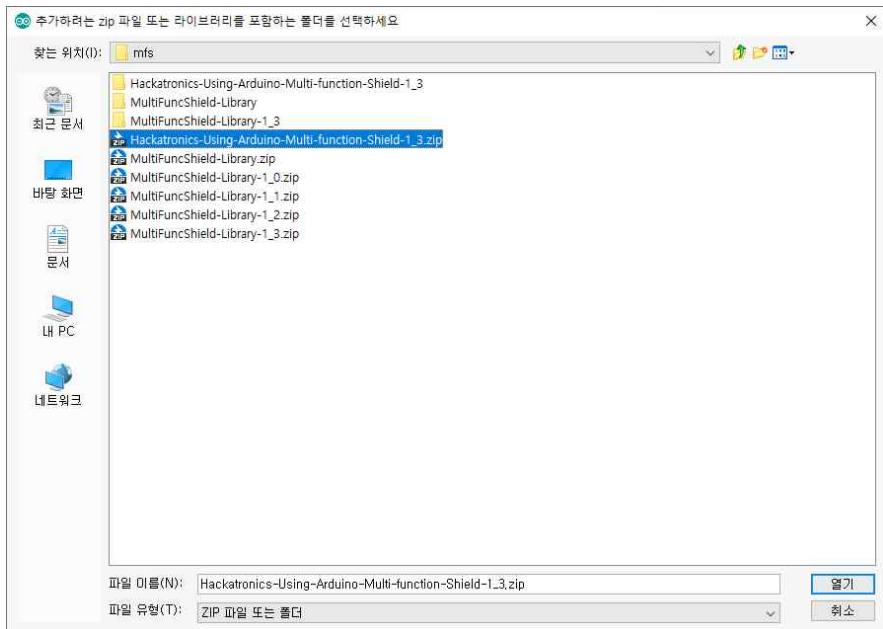


그림 123 zip 파일 선택

어느 정도 익숙해지면 라이브러리 없이도 쉴드를 사용할 수 있다. 하지만 라이브러리를 사용하는 이유는 첫째, 보다 빠르게 프로그래밍을 하기 위해서이고, 둘째, 복잡한 내부를 굳이 알 필요없이 프로그래밍이 가능하기 때문이다. 즉, 주어진 라이브러리를 사용하면 어떻게 연결되어 있는지를 알지 못해도 프로그래밍이 가능해진다.

MFS 라이브러리 1.2 까지는 TimerOne 라이브러리가 필요했지만 1.3 버전에서는 TimerOne 없이 사용가능하게 수정되었다. 다만 예전 버전을 MFS 라이브러리를 사용하거나 MFS 라이브러리 없이 MFS 를 사용하는 경우도 있기 때문에 TimerOne 라이브러리는 설치해 두도록 한다.

8.2. MFS 하드웨어 설정

멀티평션쉴드(MFS)는 아두이노 레오나르도에 맞춰서 만들어졌다. 하지만 아두이노 우노나 메가 2560과 함께 사용하기에 아무런 기술적 어려움이 없다. 아두이노 레오나르도는 사진처럼 마이크로 USB를 사용한다. 프린터와 동일한 규격의 USB B 타입을 사용하는 우노나 메가2560과 USB 케이블이 다르다.



그림 124 아두이노 레오나르도

아두이노 레오나르도는 마이크로 USB 커넥터를 사용하기 때문에 작은 커넥터 위에 충분한 공간이 생긴다. 반면 아두이노 우노와 아두이노 메가 2560은 USB B 타입 커넥터를 사용하기 때문에 커넥터와 MFS 사이에 공간이 없다. USB 커넥터는 금속으로 되어 있어 전기를 통하는 도체이기 때문에 그 위에 PCB의 납땜한 부분이나 전기가 흐르는 곳이 닿으면 그로 인해 문제가 발생할 수 있다. 아두이노는 5V은 낮은 전압을 사용하기 때문에 인체에 유해한 일이 발생하지는 않지만 LED가 제대로 작동하지 않거나 아두이노가 꺼지는 등의 일이 발생할 수 있고, 심하면 USB 선을 통해 PC가 고장을 일으킬 수도 있다.

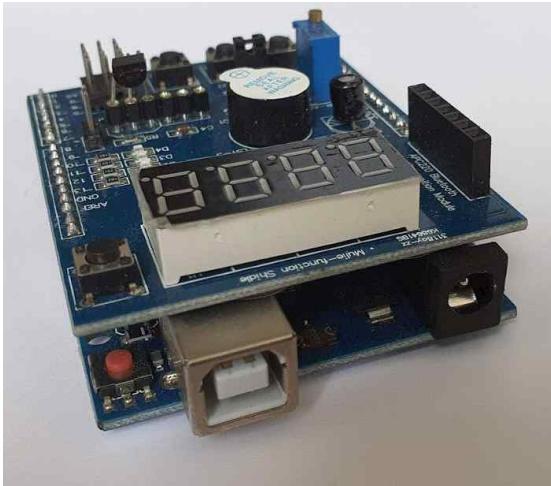


그림 125 우노와 결합한 MFS

그래서 MFS 를 사용하기 전에 USB 커넥터와 닿는 부분을 정리해주도록 한다. USB 커넥터와 닿는 납땜부위는 최대한 얇게 잘라준다. 집에 니퍼가 있으면 니퍼를 쓰면 좋고, 없다면 일반 가위를 사용하면 된다. 단, 섬세한 공예에 사용되는 가위는 날이 상할 수 있으므로 가급적 공작용 저렴한 가위를 쓰면 된다. 두꺼운 종이를 오릴 수 있는 가위면 충분하다. 계속해서 아두이노와 관련된 프로그램과 개발을 하려는 사람이라면 전자부품 다리를 자를 수 있는 작은 니퍼를 구입해두면 좋다. PLATO 니퍼는 저렴하면서 얇은 전자부품 다리를 자르기에 좋다. PLATO 니퍼를 쓰면 아주 얇고 만족할 만한 커팅을 할 수 있다.



그림 126 우노와 MFS

MFS 의 뒷면이다 7 Segment LED 가 있는 바로 뒤쪽에 있는 납땜되어 있는 선들을 깊게 만들어준다. 12개의 선중, 오른쪽 아래에 있는 3개를 깊게 만든다. 실제로 USB 커넥터에 닿는 것을 오른쪽 아래의 2개다.



그림 127 뒷면 수정사항

깊게 자른 뒤에 USB 커넥터와의 합선을 피하기 위해 절연테일이나 유리판 미끄럼 방지 받침등을 붙인다. 외관을 생각하면 유리판 미끄럼 방지 받침

가장 작은 것을 구입해서 사용하면 되고, 아니면 검정색 절연 테잎을 사용하면 된다.

이제 합선의 우려가 없어졌으므로 아두이노 우노 혹은 아두이노 메가 2560 과 MFS 를 결합시킨다. 헤더핀은 좌우가 다르게 배치되어 있어 정확히 맞는 방법은 하나밖에 없으니 잘 살펴보면서 결합시키면 된다.

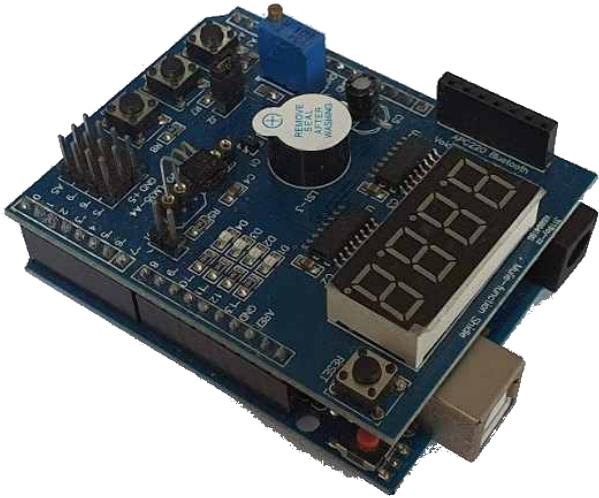


그림 128 우노와 결합된 MFS

7 Segment LED 뒷쪽에 튀어나온 핀을 자르고 절연작업을 거친 후 우노와 결합된 MFS이다.

8.3. MFS 부저 알람 프로그래밍

간단한 부저 알람 프로그램을 만들어 라이브러리가 제대로 작동하는지 확인해보자. 다음 코드를 넣어 컴파일시 문제가 없는지, 부저에서 소리가 제

대로 나는지 확인해보라.

```
/*********************  
* mfs_beep_op.ino  
*  
* 멀티펑션쉴드 부저 예제  
*****  
#include <MultiFuncShield.h>  
void setup()  
{  
    MFS.initialize();  
    MFS.beep();      // 부저소리  
    delay(1000);  
    MFS.beep(5, 5, 4, 3, 50);  
        // (5,5)-(부저소리나는 시간, 정지하는시간 ms)  
        // (4)-(앞의 소리를 4번반복)  
        // (3,50)-(앞 전체과정을 3회반복, 반복시 50ms 정지)  
}  
void loop() {}
```

라이브러리가 제대로 설치되었다면 에러없이 부저 소리가 "삑~ 삑삑삑삑
삑삑삑삑" 이렇게 나올것이다. MultiFunctionShield 사용과 이후
C 문법에 대해서는 다음장에서 계속 진행된다. 이곳에서는 라이브러리가
제대로 설치된 것과 MFS 라이브러리 사용에 대한 기초적인 내용을 확인한
다.

8.4. MFS 라이브러리 사용방법

MFS 라이브러리를 이용하기 위해서 소스의 맨 윗부분에

```
#include <MultiFuncShield.h>
```

를 넣는다. #include 는 전처리기로 외부에 존재하는 파일을 그 자리에 넣는 기능을 한다. 이미 우리가 설치한 MFS 라이브러리에 있는 MultiFuncShield.h 라는 파일을 읽은 다음 그 내용을 #include <MultiFuncShield.h> 자리에 넣는 것이다. 이제부터 라이브러리에 들어있는 기능을 사용할 수 있다.

```
MFS.initialize();
```

위 명령은 MFS 라이브러리를 설치했기 때문에 사용할 수 있는 명령이다. C 문법이 아닌 C++ 문법이지만 이정도는 많이 사용되는 내용이기 때문에 잘 봐두도록 하자. "." 의 앞에는 몸체가 오고 "." 의 뒤에는 몸체에 포함된 어떤 내용이나 기능이 온다. 몸체는 클래스를 이용하여 만들어지며 보통 오브젝트, 객체라고 부른다. 클래스와 객체에는 변수와 함수가 포함되어 있다. 즉, "." 을 사용하면 몸체에 속해있는 변수나 함수를 불러서 사용할 수 있다.

MFS.initialize() 는 MFS 라는 객체 혹은 클래스에 속해 있는 initialize()라는 함수를 실행시키라는 뜻이다.

MFS.initialize() 는 쉴드를 초기화시켜준다. 쉴드가 초기화된 후에 MFS 를 다룰수 있게 된다.

```
MFS.beep();
```

MFS 에 있는 beep() 함수를 실행시킨다. 200 ms 동안 알람소리가 나온다.

```
MFS.beep(5, 5, 4, 3, 50);
```

beep() 를 사용하는 방법으로 단순히 200 ms 동안 짧은 경고음이 나오는 것과 함께 알람을 직접 만들수 있다. 만드는 방법은

```
beep(알람지속시간, 알람정지시간, 반복회수, 전체 반복회수, 일시멈춤시간);
```

과 같다. 위의 숫자를 넣어서 확인해 보면 다음과 같다.

- "A" 는 5 ms 동안 부저소리가 나는 것으로 하고, "-" 는 5 ms 동안 조용한 상태라 하면
- "A-" 는 5ms 동안 소리가 나오고 그 다음 5ms 동안 조용한 상태를 의미 한다. beep(5,5, ...)
- 이러한 상태를 4번 반복한다.
- "A-A-A-A-" 는 beep(5,5,4,...) 를 의미한다.
- 그 다음 여기까지를 3번 반복하는데 반복할 때마다 50ms 씩 휴지기를 둔다.
- "A-A-A-A-----A-A-A-A-----A-A-A-----" 는 전체를 3번 반복하는 신호다.

```
MFS.beep(5, 5, 4, 3, 50);
```

는 "A-A-A-A-----A-A-A-A-----A-A-A-A-----" 와 같은 알람을 나타낸다.

라이브러리에 있는 함수 요약은 다음과 같다.

MFS.beep() : 부저 : 부저 소리 제어

MFS.getButton() : 버튼 : 버튼 누름 상태 제어

MFS.write() : 7 Segment LED : 7 Segment LED 제어

MFS.blinkDisplay() : 7 Segment LED : 7 Segment LED 제어

MFS.blinkLeds() : LED : LED 제어

MFS 예제

Basic Input Output	Reading Sensors	Applications
Beeper	MPU6050	Count_Down_Timer
Buttons	Pulse_Counter	Heart_Monitor
Led_Display	Pulse_Counter_2	Level_Indicator
Led_Lights	Sonar_Module	Sonar_Ranger
Preset_Pot	Temp_Sensor	Speedometer
		_24_hr_Alarm_Clock

그림 129 MFS 기능별 분류 색인

Buttons.ino 예제

```
#include <MultiFuncShield.h>
```

```
void setup()
{
```

```

// put your setup code here, to run once:
Serial.begin(9600);
MFS.initialize();           // initialize multi-function shield
library
}

void loop()
{
    // put your main code here, to run repeatedly:

    byte btn = MFS.getButton(); // Normally it is sufficient to
    compare the return          // value to predefined macros, e.g.
    BUTTON_1_PRESSED,           // BUTTON_1_LONG_PRESSED etc.

    if (btn)
    {
        byte buttonNumber = btn & B00111111;
        byte buttonAction = btn & B11000000;

        Serial.print("BUTTON_");
        Serial.write(buttonNumber + '0');
        Serial.print("_");

        if (buttonAction == BUTTON_PRESSED_IND)
        {
            Serial.println("PRESSED");
        }
        else if (buttonAction == BUTTON_SHORT_RELEASE_IND)
        {
            Serial.println("SHORT_RELEASE");
        }
    }
}

```

```
else if (buttonAction == BUTTON_LONG_PRESSED_IND)
{
    Serial.println("LONG_PRESSED");
}
else if (buttonAction == BUTTON_LONG_RELEASE_IND)
{
    Serial.println("LONG_RELEASE");
}
}
```

8.5. MFS에서 LED와 7 Segments 사용

A. 7 세그먼트 LED 제어

MFS를 이용한 7 Segments의 사용법은 다음과 같다.

MFS의 숫자 표시는 인터럽트를 사용하여 백그라운드에서 수행되므로 응용 프로그램은 계속해서 주된 작업을 수행할 수 있다. 아래 스케치에 표시된 것처럼 문자열, 정수 및 부동값이 디스플레이를 이용하여 출력된다.

7 Segments를 사용하기 위해서는 우선 MFS.initialize()를 통해 쉴드를 초기화 한다.

이후 MFS.write() 함수를 이용해 원하는 문구, 숫자를 화면에 표시한다.

```
MFS.write("HI"); // 원하는 내용 출력  
MFS.write(-314); // 정수값 출력  
MFS.write(3.141592, 2); // 소수점이하 2째 자리까지 출력
```

다음은 간단한 예제 소스이다.

```
#include <MultiFuncShield.h>  
  
/*  
 * MFS 이용한 7 Segments 예제  
 */  
  
void setup() {  
    // put your setup code here, to run once:  
  
    MFS.initialize(); // initialize MFS library  
  
    MFS.write("Hi");  
    delay(2000);  
    MFS.write(-273);  
    delay(2000);  
    MFS.write(3.141, 2); // display to 2 decimal places.  
    delay(2000);  
}  
  
int counter=0;  
byte ended = false;  
  
void loop() {  
    // put your main code here, to run repeatedly:  
  
    if (counter < 200)
```

```

{
    MFS.write((int)counter);
    counter++;
}
else if (!ended)
{
    ended = true;
    MFS.write("End");
    MFS.blinkDisplay(DIGIT_ALL, ON);
}
delay(50);
}

```

B. MFS 쉴드의 LED 제어

MFS 쉴드의 LED 제어를 위해 다기능 쉴드 라이브러리를 반드시 사용할 필요는 없다. 하지만 기본 깜박임 동작을 수행하기 위해 LED 제어를 위한 지원은 제공된다. 인터럽트를 사용하여 백그라운드에서 깜박임을 제어한다.

```

#include <TimerOne.h>
#include <Wire.h>
#include <MultiFuncShield.h>

void setup() {
    // put your setup code here, to run once:

    Timer1.initialize();
    MFS.initialize(&Timer1); // initialize multi-function shield
    library

    MFS.writeLeds(LED_ALL, ON);

```

```

delay(2000);
MFS.blinkLeds(LED_1 | LED_2, ON);
delay(2000);
MFS.blinkLeds(LED_1 | LED_2, OFF);

MFS.blinkLeds(LED_3 | LED_4, ON);
delay(2000);
MFS.blinkLeds(LED_ALL, ON);
delay(2000);
MFS.blinkLeds(LED_ALL, OFF);
MFS.writeLeds(LED_ALL, OFF);
}

void loop() {
    // put your main code here, to run repeatedly:
}

```

8.6. MFS 에서 센서 사용

MFS 는 내부에 간단한 센서를 가지고 있다. 우선 가변저항인 포텐셔미터를 이용한 아날로그 입력값을 읽을 수 있다. 손잡이를 돌려 조정한 저항 값을 읽어 다기능 실드에 표시할 수 있다. 다음 예제를 업로드한 후 전위차계의 나사를 돌려 7 Segments 디스플레이의 판독값 변화를 확인해보라.

```

#include <TimerOne.h>
#include <Wire.h>
#include <MultiFuncShield.h>

void setup()
{

```

```

    Timer1.initialize();
    MFS.initialize(&Timer1); // initialize multi-function shield
library
}
void loop()
{
    MFS.write(analogRead(POT_PIN));
    delay(100);
}

```

아날로그 값을 읽어서 7 Segments 화면에 표시한다.

온도와 습도 측정 하기

MFS 실드는 LM35 센서를 부착할 수 있는 소켓을 가지고 있다. LM35 온도 센서를 외부에서 구입하여 장착하면 온도를 읽을 수 있다.

다기능 실드에는 LM35 온도 센서를 수용할 수 있는 소켓이 있다. 이 소켓에 LM35를 위치를 맞추어 바르게 삽입한다. 만약 그렇지 않으면 아두이노나 USB로 연결된 컴퓨터가 복구 불가능하게 손상될 수 있다. 센서가 잘못 연결된 경우 센서와 주변이 매우 뜨거워지기 때문에 바로 알 수 있다. 만약 센서 부분이 뜨거워지면 바로 전원을 끊고 잘못된 부분을 살펴봐야한다. 다기능 실드 라이브러리는 이 센서의 판독값을 평활화하기 위한 SMOOTHING_NONE, SMOOTHING_MODERATE, SMOOTHING_STRONG의 세 가지 수준의 필터링을 제공한다.

```

#include <TimerOne.h>
#include <Wire.h>
#include <MultiFuncShield.h>

```

```
// NOTE: make sure jumper J1 is removed from shield,  
// and that LM35 is inserted correctly.  
  
void setup()  
{  
    Timer1.initialize();  
    MFS.initialize(&Timer1); // initialize MFS library  
  
    // Initialize using a low pass filter.  
    // Choose either: SMOOTHING_NONE, SMOOTHING_MODERATE  
    // or SMOOTHING_STRONG  
    MFS.initLM35(SMOOTHING_MODERATE);  
}  
  
void loop()  
{  
    int tempCentigrade = MFS.getLM35Data();  
    // get centigrade in 1/10 of degree.  
    MFS.write((float)tempCentigrade / 10, 1);  
    // display temp to 1 decimal place.  
    delay(100);  
}
```


Chapter 09

주요 학습 목표

- C언어의 구조적 기능과 함수에 대한 이해
- 함수 선언과 정의

Chapter 09. 함수

이번 장에서는 함수의 개념을 이해하고 만들어져 있는 함수를 사용하는 것과 함수를 만드는 것을 다뤄본다. C 언어는 기본적으로 모든 것이 함수로 되어 있다고 해도 과언이 아니다. C 언어는 main() 함수를 호출하면서 main() 함수에 포함된 명령들을 순차적으로 실행하고 마친다. 실제로 아두이노는 setup() 과 loop() 함수로 구성되어 있지만 조금 더 깊이 들여다보면 main() 함수가 다음과 같이 구성되어 있다.

```
#include <Arduino.h>
int main(void)
{
    init();
    setup();      // setup() 함수가 들어가는 곳
    for (;;)
    {
        loop();    // loop() 함수가 들어가는 곳
        if (serialEventRun) serialEventRun();
    }
    return 0;
}
```

main() 함수를 실행시키면 main() 함수는 다시 init() 함수부터 시작해서

`setup()` 함수를 실행시킨 후 무한루프로 `loop()` 함수를 실행시킨다. `for(;;)`는 `while(1)` 과 동일하고 무한루프임을 앞장에서 공부했다. `loop()` 를 실행하면서 시리얼 이벤트가 있으면 `serialEventRun()` 함수를 실행시키는 구조로 되어 있다.

이처럼 절차지향, 구조적 프로그래밍인 C 언어는 기본적으로 함수로 구성되어 있다.

이번 장에서는 함수를 다루는 문법을 익힌다.

9.1. 함수란

어린 시절 배웠던 사칙연산 상자가 있다. 상자의 위에 구멍이 있어 그리로 어떤 숫자가 적힌 구슬이 들어가게 되어 있다. 상자에는 $x 2$ 라고 적혀있다. 상자의 아래쪽에 구멍이 있어 그리로 들어간 구슬보다 2배 더 큰 숫자가 적힌 구슬이 나온다.

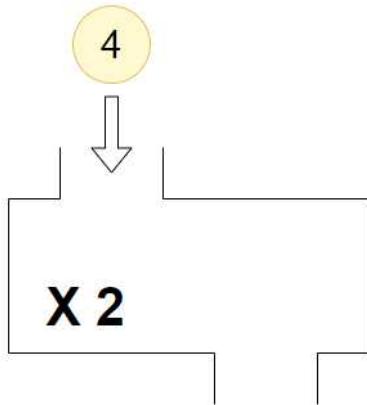


그림 130 함수상자

함수는 위의 그림과 같다. 어떤 값이 블랙박스 안으로 들어가고, 블랙박스에 안에서 처리된 값이 블랙박스 밖으로 나온다. 블랙박스 안에서 어떤 일이 일어나는지 우리는 관심이 없다. 단, 이 블랙박스는 들어간 값을 2배 해

서 보내준다는 것만 기억하면 된다.

이 함수는 특정 작업을 수행하는 명령어들의 구조화된 집합이다. 큰 공장에서 컨베이어 벨트를 통해 제품이 순서대로 일정한 과정을 거치며 완성되듯이 C 언어는 함수를 통해 시작부터 끝까지 하나씩 조립되어 간다. 아두이노는 main() 함수 내부에 setup(), loop() 가 있으며, setup() 과 loop() 함수를 사용자가 만들어서 프로그램을 완성하게 된다. 이런 함수 즉, 블랙박스를 때로는 있는 것을 사용해야하기도 하고, 때로는 직접 만들어야 하기도 한다.

Serial.println() 같은 이미 만들어져 있는 함수는 그냥 사용하면 된다. C 언어를 사용해서 프로그래밍하기 위해서는 이미 만들어져서 사용할 수 있는 함수들을 어느정도 알고 있어야 할 뿐만 아니라, 아직 없는 기능을 직접 함수로 만들수도 있어야 한다.

아두이노를 포함해서 임베디드 시스템에서 사용하는 C는 PC에서 사용하는 표준입출력 함수를 거의 사용하지 않는다. 대신 시스템이 가지고 있는 자체적인 하드웨어에 특화된 함수를 사용하거나 혹은 개발자가 스스로 함수를 만들어 사용하게 된다.

이미 우리는 Serial.println() 부터 몇몇 제공된 함수들을 사용해 보았다. 이번 장에서는 아두이노 내장 함수들과 함께 사용자 정의 함수를 만들어 보도록 한다.

함수를 만들때는 (1) 함수선언, (3) 함수호출, (2) 함수정의의 과정을 거친다. 함수선언 없이 바로 함수정의 후에 함수 호출을 하기도 하지만 일반적인 과정은 선언-호출-정의의 순서를 따른다.

- 함수 선언 : 컴파일러에게 함수의 이름과 간단한 소개를 알려줌

- 함수 호출 : 함수를 불러서 사용함
- 함수 정의 : 함수의 정확한 내용을 기록함

9.2. 함수선언

함수 선언은 컴파일러에게 함수의 간략한 정보와 소개를 하는 것으로, 다음과 같은 형식을 가진다. 위의 2배 블랙박스를 함수로 선언한다면 다음과 같다.

```
int ix2(int i);
```

함수 선언은 꼭 필요한 것만 다룬다. 실제 함수 내부의 모습은 제외하고 들어가는 값과 나오는 값, 그리고 함수의 이름이 포함된다. 변수가 아닌 함수라는 것을 나타내기 위해서 함수의 이름뒤에 괄호()를 붙인다. 괄호안에 있는 매개변수는 불필요할 경우 생략할 수도 있다. 문장의 끝에는 세미콜론 ";" 을 붙인다.

```
int ix2 ( int i ) ;  
| | |  
| | +--- 블랙박스로 들어가는 값 (정수)  
| +--- 함수의 이름 (함수를 호출할때 필요)  
+--- 블랙박스에서 나오는 값 (정수)
```

그림 131 함수로 들어가는 값

함수 선언은 반드시 필요하지는 않다. 함수 선언을 생략하고 함수 정의가 있어도 된다. 단, 이 경우 함수 정의는 반드시 함수 호출보다 앞에 있어야 한다.

컴파일러는 기본적으로 위에서 아래로, 왼쪽에서 오른쪽으로 하나씩 읽는다. 함수 선언이나 함수 정의가 없는 상태에서 함수 호출이 이루어지면 컴파일러는 에러를 발생한다. 상식적으로 "더하라"라는 명령이 무슨 뜻인지 모르는 상대에게 명령을 하기 위해서는 먼저 "더하라"라는 말의 뜻을 알려줘야하고(함수정의), 그 후에 "더하라"는 명령을 내려야 한다. C 언어 역시 함수를 사용하기 전에 함수를 알려줘야한다. 함수 정의로 알려주거나 혹은 함수선언이라도 해서 들어가는 값의 종류와 나오는 값의 종류(타입)를 알려줘야한다.

9.3. 함수정의

함수를 만드는 것을 함수정의라고 한다. 다음은 ix2 라는 함수를 정의한 예이다.

```
int ix2(int i)
{
    int tmp;
    tmp = i*2;
    return tmp;
}
```

한줄씩 살펴보면 다음과 같다.

```

int ix2 ( int i )
=====
| | +-- 매개변수 목록
| +-- 함수이름
+-- 반환형

```

그림 132 ix2 함수

함수는 '매개변수'를 가질 수 있고, 함수를 호출하면서 '인자'를 전달하면 그 인자값이 매개변수에 담겨서 함수를 실행하게 된다. 함수의 실행이 끝난 후 만들어진 결과값을 '반환값'이 되어 돌려보내진다.

함수 이름 뒤 괄호 () 로 둘러쌓인 부분에 매개변수를 넣을 수 있다. 만약 함수에 매개변수가 필요하지 않다면(입력값이 필요없다면), 매개변수를 생략할 수 있다. 또한 연산이 끝난 후 반환되는 값이 필요없다면 반환값이 비어있다는 뜻으로 void 를 int 자리에 써준다.

매번 아두이노에서 새파일을 만들면 seup()과 loop() 가 만들어진다. 그때 잘 보면 setup() 과 loop() 의 앞에 void 가 붙고 괄호안은 비어있음을 알 수 있다. 즉, setup 함수는 함수를 호출할 때 아무런 입력이 필요없고, loop 도 함수호출시에 입력이 필요없다는 뜻이다. 호출시에 입력이 필요없다는 것이 함수 실행중에 아무런 입력이 들어오지 않는다는 의미는 아니다. 함수를 시작할 때 어떤 특정값을 필요로 하지 않는다는 것이다. 즉, 함수 안에서 외부의 신호를 읽는것과 함수 자체를 실행할때 특정값을 필요로 하는 것은 별도로 이해해야한다.

setup(), loop() 함수를 사용하면서 내부에 Serial.println() 함수를 사용해서 시리얼 통신으로 문자열을 보내는 프로그램을 만들어보았고, Serial.available() 함수를 써서 외부에서 시리얼 포트를 통해 들어오는 입

력값을 받아들이는 프로그램도 만들었다. 즉, 함수 내부에서 외부 신호를 받거나 보내는 것과 함수를 만들때 매개변수로 어떤 값을 받아들이는 것은 다르게 이해해야 한다.

```
void loop()
{
    while (Serial.available() > 0)
    {
        int number = Serial.parseInt();
    }
    // 생략
}
```

위의 코드에서 loop() 함수는 매개변수가 없다. 하지만 시리얼통신을 통해 들어오는 값을 red라는 변수에 저장한 후 number 변수를 사용한다.

```
int ix2(int i)
{
    int tmp;
    tmp = i*2;
    return tmp;
}
```

i2x() 함수는 시리얼통신을 통해 i 값을 받는 것이 아니라 i2x 함수를 호출하면서 인자를 함께 전달하면, 그 인자가 매개변수에 전달되어 매개변수 i에 전달받은 값이 저장된다. 함수 i2x() 내부 연산이 끝난 다음 어떤 값을 이 함수를 호출한 쪽으로 보내는데 그 값은 int 타입의 정수가 된다.

즉, i2x(3)을 호출하면 3이라는 '인자'가 i2x라는 이름의 함수를 호출하면서 함께 전달되고, 전달된 인자값 3은 i2x(int i) 함수의 매개변수 i에 저장된다.

이때 함수의 세부내용은 중괄호 안에 들어간다.

```
{  
    int tmp;  
    tmp = i*2;  
    return tmp;  
}
```

중괄호 안의 함수 세부 내용은 위에서 아래로 한 줄씩 순차적으로 실행된다.

```
int tmp; // tmp 변수를 만든다.  
tmp = i*2; // tmp 변수에 매개변수 i에 담긴 값을 2배하여 저장한다.
```

매개변수의 값으로 들어온 i 값에 2를 곱한 후 그 결과값을 tmp에 저장했으면 이제 그 값을 함수를 불렀던 곳으로 돌려보내야 한다.

```
return tmp;
```

return 은 그 뒤에 있는 값을 함수를 호출한 쪽으로 반환하라는 명령이다. 즉, 윗 줄은 tmp 변수에 있는 값을 함수 ix2()를 호출한 쪽으로 돌려주라는 뜻이다.

함수의 구조

리턴타입 함수이름 (매개변수) { 연산 ; return 반환값 ; }	int ix2(int i) { int tmp; tmp = i*2; return tmp; }
--	---

그림 133 함수의 구조

9.4. 함수호출

함수를 선언하거나 함수가 정의되었으면 이제 함수를 사용할 수 있다. 함수를 사용하는 것을 함수호출이라고 한다. 함수호출시에는 함수선언과 비슷한 절차로 명령을 내리면 된다.

```
int a = ix2(4);
```

위 명령을 보면 대입연산자 '=' 뒤에 함수 ix2() 가 있으며, ix2함수의 매개변수로 4라는 정수가 들어있는 것을 알수 있다. 즉 ix2() 함수를 호출하면서 매개변수로 4를 함수에 전달해 준다. ix2() 함수는 4를 전달받으면서 매개변수 i 값으로 4를 넣고, 계산결과를 int 타입의 정수로 전달한다. 즉, 대입연산자 '='의 우측에 함수 호출 후 전달받은 값 8이 들어간다. 정수형 변수 a 는 함수 호출 결과로 반환된 값 8이 저장된다.

```

int a = ix2(4);
| | +--(1) 함수호출 |
| | +-- (2) 함수에 들어갈 매개변수 4
| +-- (4) 함수호출 후 반환값이 저장될 곳
+-- (3) 변수 타입 (함수의 반환형과 동일)

```

그림 134 함수 호출의 순서

간단한 ix2() 함수를 사용해 4를 2배로 만들어 화면에 출력하는 코드는 다음과 같다.

```

/****************************************************************************
 * myfunc_op.ino
 *
 * 사용자함수 만들기
 */
int ix2(int i) // 함수정의 - 호출보다 먼저라면 함수선언 필요없음
{
    int tmp;
    tmp = i*2;
    return tmp;
}

void setup()
{
    Serial.begin(9600);
    int tmp = ix2(4); // 함수호출
    Serial.println(tmp);
}

void loop(){}

```

위의 코드는 함수선언 없이 함수정의가 이루어진 형태다. 함수를 호출하기 전에 함수정의가 이루어진다면 별도의 함수 선언이 필요하지 않다. 만약 함수정의가 함수호출 아래쪽에 들어가게 된다. 함수선언을 미리 해두어야 한다.

```
*****  
* myfunc_op2.ino  
*  
* 사용자함수 만들기  
*****  
  
int ix2(int i);           // 함수선언  
  
void setup()  
{  
    Serial.begin(9600);  
    int tmp = ix2(4);      // 함수호출  
    Serial.println(tmp);  
}  
  
void loop(){}  
  
int ix2(int i)           // 함수정의  
{  
    int tmp;  
    tmp = i*2;  
    return tmp;  
}
```

위의 코드에서 함수 정의가 함수 선언없이 함수 호출보다 아래쪽으로 가게 된다면 컴파일러에 따라 문제가 생길 수 있다. 컴파일러에 따라 에러메시지를 내고 컴파일이 되지 않거나 경고메시지가 뜨게 된다. 아두이노의 경우

이런 부분을 다듬어서 경고없이 컴파일이 되고 있지만 이후 아두이노가 아닌 다른 임베디드 시스템을 사용할 경우 함수 선언을 미리 하지 않는 습관을 들여두면 나중에 어려움에 처할 수 있다. 항상 함수 선언을 소스코드의 선두에 미리하는 습관을 갖는 것이 좋다.

다양한 C 컴파일러들이 만들어져 있다. 아두이노와 같은 MCU들을 위해서도 KEIL, IAR, CodeVision, Atmel Studio, GCC, AVR Toolchain 등 다양한 컴파일러가 시중에 나와있으며 각각의 컴파일러들은 회사나 만든 이들의 생각과 이념이 담겨 있다. 그러다보니 C 언어의 표준이 있음에도 불구하고 더 나은 그들의 가치관과 개발 과정에서 어쩔수 없이 타협하는 부분이 있다. 현재 C 언어는 1999년을 기점으로 C99라는 당시의 현실에 적합한 표준을 만들었고, 2011년 C11을 발표했다. K&R로 시작해서 ANSI C, 지금의 C99와 C11에 이르기까지의 표준화 과정이 지금 우리가 사용하는 C 컴파일러에 적용되었는가하면 그렇지 않다. 대부분 C99, C11에 기반을 두지만 여전히 이전에 사용되던 것을 폐기하지 못하고 넣어두는 경우가 많다. 그 하나가 함수정의가 함수호출보다 뒤에 있을때, 문법적 에러를 발생시키지 않고 경고만 뜯 채 컴파일이 되는 현상이다.

즉, 위에서 설명한 대로라면 아래는 쓰면 안되는 구문이다.

```
void setup()
{
    Serial.begin(9600);
    int tmp = ix2(4); // 함수호출 - 모르는 상태에서 호출했다!
    Serial.println(tmp);
}

void loop(){}
int ix2(int i)          // 함수정의
{
    int tmp;
```

```
    tmp = i*2;  
    return tmp;  
}
```

하지만 위 구문은 제대로 컴파일되고 실행된다. 이유는 이전 K&R 때는 함수선언이나 정의없이 호출하면 호출 구문을 통해 함수의 형태를 추측하고 그것을 사용하도록 했다. 그것이 C11이 표준이 된 지금까지 계속 이어져오고 있는 것이다. 이로 인해서 발생할 수 있는 문제는 분명히 있다. 즉, 위처럼 써도 컴파일은 된다. 하지만 절대로 위처럼 함수선언이나 함수정의없이 함수를 호출하는 코드를 만들지 않기를 바란다. 적어도 의도적으로 위처럼 만들어서는 안된다.

9.5. 매개변수 없는 함수

함상 함수를 더하기 상자로 시작하다보니 처음 C를 시작하는 이들은 함수안으로 들어오는 값이 없으면 의아하게 생각하곤 한다. 함수를 쉽게 이해하기 위해서 더하기 상자의 예를 든것인지 그것이 함수의 전부가 아님을 이해해야한다. 즉, C에서의 함수는 들어오는 값이나 나가는 값이 없어도 함수가 될수 있다.

다음과 같이 인사말을 출력하는 코드를 생각해보자.

```
Serial.println("Hello! ");  
Serial.println("Welcome to the Arduino Course.");  
Serial.println("If you work hard, you will surely have good  
results.");  
Serial.println("If you have any questions, please use the  
question board.");  
Serial.println("We are also answering questions using the
```

YouTube channel.”);

이 구문을 프로그램중에 여러곳에서 사용해야한다. 그때마다 위 구문을 복사해서 넣어도 된다. 사실 이렇게 하는 이들이 많다. 처음 아두이노를 시작하는 이들에게는 구문을 복사해서 원하는 곳에 넣는 것도 쉬운 일은 아니고, 이렇게라도 해서 원하는 결과를 만들어 내는 것은 격려해 줄일이다. 하지만 조금만 더 생각해본다면 같은 명령을 여러번 사용할때마다 복사해서 붙여넣는다는것은 좋은 방법은 아니다.

위 코드를 함수로 만들어서 필요한 곳에 함수를 호출하는 식으로 사용하면 된다.

```
void sayNotice()
{
    Serial.println("Hello! ");
    Serial.println("Welcome to the Arduino Course.");
    Serial.println("If you work hard, you will surely have good
results.");
    Serial.println("If you have any questions, please use the
question board.");
    Serial.println("We are also answering questions using the
YouTube channel.");
}
```

syaNotice() 함수는 매개변수와 반환값이 없다. syaNotice() 함수는 매개변수 없이 사용하고 함수 수행 후 결과를 따로 반환하지 않는 함수다.

9.6. 매개변수 있는 함수

위에서 만들어 본 ix2() 함수를 응용해서 idiv2() 함수를 만들어보자. 들어오는 입력값은 정수가 되도록 하고 나가는 출력값은 입력값을 2로 나눈 실수가 되도록 하자. 즉, 함수의 선언은 다음과 같다.

```
float idiv2(int i);
```

함수의 내용은 다음과 같다.

```
float result; // 실수를 저장할 변수 result 을 만든다.  
result = i/2.0; // i를 실수 2.0 으로 나누고 결과를 저장  
return result; // 변수 result 의 값을 반환
```

함수를 제대로 만들면 다음과 같다.

```
float idiv2(int i)  
{  
    float result;  
    result = i/2.0;  
    return result;  
}
```

어느정도 익숙해지면 다음처럼 간결하게 만들어도 된다.

```
float idiv2(int i)  
{  
    return i/2.0;  
}
```

앞에서 배웠던 것처럼 정수 i를 실수 2.0 으로 나누면 결과는 실수가 되고, 그 결과값인 실수를 return 으로 반환한다.

위에서 만든 함수를 사용해서 전체 코드를 만들어보면 다음과 같다.

```
*****  
* myfuncs_op3.ino  
*  
* 사용자함수 만들기  
*****  
  
int ix2(int i);      // 곱하기2 함수선언  
float idiv2(int i); // 나누기2 함수선언  
void sayNotice();  
  
void setup()  
{  
    Serial.begin(9600);  
    sayNotice();  
    int tmp = ix2(4);      // 함수호출  
    Serial.println(tmp);  
    float tmp2 = idiv2(4); // 함수호출  
    Serial.println(tmp2);  
    sayNotice();  
  
}  
  
void loop(){}
  
  
int ix2(int i)      // 함수정의
{
    int tmp;
```

```

    tmp = i*2;
    return tmp;
}

float idiv2(int i)          // 함수정의
{
    return i/2.0;
}

void sayNotice()
{
    Serial.println("Hello! ");
    Serial.println("Welcome to the Arduino Course.");
    Serial.println("If you work hard, you will surely have good
results.");
    Serial.println("If you have any questions, please use the
question board.");
    Serial.println("We are also answering questions using the
YouTube channel.");
}

```

매개변수가 둘 이상인 함수도 당연히 가능하다.

두 수를 입력받아 합을 구하는 함수를 만들어보면 다음과 같다. 매개변수 a 와 b 에 각각 인자값을 넣어주면 두 값을 더한 후 그 합을 반환한다.

```

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

```

두 수를 넣으면 큰 값을 찾는 함수를 만들어보면 다음과 같다.

```
int maxNumber(int a, int b)
{
    if (a<b)
        return b;
    else
        return a;
}
```

인자, 매개변수와 인수의 차이

함수를 정의하거나 선언할 때 () 안에 들어가는 변수를 '매개변수' 혹은 '인자'라고 하고 영어로는 'parameter'라고 한다. 매개변수는 변수의 일종으로 메모리상 공간을 차지하고 어떤 값을 저장하는 기능을 한다. 그때 이 변수로서의 매개변수에 들어갈 구체적인 값을 '인수'라고 하고 영어로는 'argument'라고 부른다.

바로 앞에서 만든 `int maxNumber(int a, int b)` 함수는 매개변수로 `int a` 와 `int b` 즉, 정수타입 값을 저장할 수 있는 `a` 와 `b`라는 변수를 가진다. `maxNumber(3, 5)`처럼 `a` 와 `b`에 3과 5를 넣어서 함수를 실행시킬 때 들어간 3과 5는 인수라고 부른다.

9.7. 반환값

`return` 문을 만나면 함수는 그 지점에서 종료되고 함수를 호출했던 곳으로

복귀한다. 그때 복귀하면서 return 바로 뒤의 수식의 처리 결과로서의 값을 복귀하는 곳으로 가지고 갈 수 있는데 이 값을 반환값이라고 한다. 즉, 함수는 더 이상 실행할 곳이 없을 때까지 간 다음 종료되거나 혹은 return 문을 만나면 즉시 반환값을 돌려주고 종료된다. return 문을 만나서 종료될 때, return 뒤의 값을 함수를 호출한 곳으로 돌려준다.

반환값은 함수 선언이나 정의에 사용된 타입과 동일해야 한다.

```
int lineSum(int num)      // 반환값은 int 타입이라고 알려준다.  
{  
    int sum = 0;  
    for (int i = 1; i<=num; i++)  
    {  
        sum += i;  
    }  
    return sum;      // sum 은 int 타입의 값을 가지고 있으며, 그 값을 반  
    환한다.  
}  
  
void setup()  
{  
    Serial.begin(9600);  
    int tmp = lineSum(4);      // 함수호출 - 함수를 모르는 상태에서 호  
    출했다!  
    Serial.println(tmp);  
}  
  
void loop(){}  
 
```

9.8. void 함수

반환값이 없는 함수를 void 함수라고 부른다. void 함수는 기존의 다른 프로그래밍 언어에서는 '프로시저'라고 불리던 명령어들의 집합으로 특정 명령어들을 실행하되 그 결과값은 반환하지 않는 서브프로그램이다. C 는 함수기반의 프로그래밍 언어로 프로시저를 함수의 형태로 사용할 수 있다. 결과값을 반환하지 않는 함수를 void 함수라고 하고 반환타입을 void 로 한다. void 의 뜻은 '없는', '텅 비어있는' 으로 반환값이 없는 경우 사용된다.

void 함수를 사용하면 값을 리턴할 필요가 없는 여러 문장을 하나의 단위로 묶을 수 있다. 1 부터 n 까지의 합을 구해서 리턴할 필요 없이 화면에 결과를 출력하는 함수를 void 함수로 만들면 다음과 같다.

```
void lineSum(int num)      // 인자가 int 타입, 반환값은 void 라고 알려준다.  
{  
    int sum = 0;  
    for (int i = 1; i<=num; i++)  
    {  
        sum += i;  
    }  
    Serial.println(sum);  
}    // 함수의 끝부분에 return 이 필요없다.  
  
void endString()  
{  
    Serial.println("PROGRAM END");  
}  
  
void setup()  
{  
    Serial.begin(9600);
```

```
    lineSum(4);      // 함수호출 - 함수를 모르는 상태에서 호출했다!
    endString();
}
```

```
void loop(){}  
  
lineSum(), endString() 함수는 반환값이 필요없는 함수이므로 void 함수가  
된다. 아두이노에서 새파일을 만들어보면 기본으로 만들어지는 setup() 과  
loop() 함수도 void 함수인 것을 알 수 있다. main() 함수에서 setup() 과  
loop() 함수를 호출하지만 호출된 이후 setup() 과 loop() 는 반환값을 돌  
려주지 않는다.
```

```
void setup()
{
    // put your setup code here, to run once:  
  
}  
  
void loop()
{
    // put your main code here, to run repeatedly:  
  
}
```

9.9. 변수와 범위

4장에서 변수에 대해 살펴며 지역변수와 전역변수에 대해 소개하였다. 코드가 실행되는 지점에 따라 특정한 값을 저장하고 불러오는 변수의 역할과 기능이 보다 다양한 변화를 가지게 된다. 이번 함수를 살펴보며 함수 내부의 공간과 외부의 공간을 기준으로 변수가 어떻게 사용되는지 살펴본다.

A. 지역변수(local variable)

함수가 만들어지는 곳은 기준의 명령어들의 순차적 실행이 이루어지는 곳과는 별도의 공간이다. 즉, 일반적으로 함수를 사용하는 경우 다음과 같은 구조를 가진다.

라인	설명	코드
1	함수정의	<pre>int lineSum(int num) { int sum = 0; for (int i = 1; i<=num; i++) sum += i; Serial.println(sum); }</pre>
...		
10	명령어	<code>int sum = 0;</code>
11	명령어	<code>sum = 1</code>
12	함수호출	<code>lineSum(4);</code>
13	명령어	<code>Serial.println(sum)</code>

그림 135 함수 정의와 실행

여기서 함수가 정의되는 공간과 명령어들이 실행되는 공간은 별도의 공간으로 구분된다. 위와 같은 구조로 프로그램이 만들어져 있다면 코드는 라인 1에 있는 함수정의부터 실행된다. 라인 1부터 차례대로 실행하면서 메모리에 lineSum() 함수에 대한 내용을 저장해 두게 된다. 라인 1에 있는 함수를 보면 sum이라는 변수를 정수타입으로 만들어 사용하고 있다. 실제 lineSum() 함수가 호출된 것이 아니기 때문에 lineSum() 함수 안에 있는 sum이라는 변수는 아직 만들어지지 않는다.

이후 10번째 라인에서 C언어는 sum이라는 변수를 만들고 변수에 0을 담게 된다. 이때 sum 변수는 함수 lineSum()에 있는 변수와 다른 공간에 만들어지는 변수다. 이처럼 제한된 공간에서 만들어지고 실행된 후 공간이 더 이상 필요없어지면 소멸되는 변수를 지역변수(local variable)라고 한다. 함수 lineSum() 안에 있는 지역변수 sum은 지역변수로 함수가 호출되어 사용되는 동안만 값을 가지게 되고, 함수가 종료되면 변수도 메모리에서 삭제된다.

아래 완성된 코드를 보면 코드의 진행순서를 확인해보도록 하자.

```
void lineSum(int num)
{
    int sum = 0;
    for (int i = 1; i<=num; i++)
    {
        sum += i;
    }
    Serial.println(sum);
}

void setup()
{
    Serial.begin(9600);
    int sum = 0;
    sum = 1;
    lineSum(4);
    Serial.println(sum);
}

void loop(){}  
-
```

아두이노 IDE 를 통해 만들어지는 소스코드는 main() 함수를 보여주지는 않지만 setup() 을 먼저 실행(호출)한다. setup() 함수는 다음 세줄을 순차적으로 실행한다.

```
Serial.begin(9600);
int sum = 0;
sum = 1;
```

시리얼통신을 위한 함수를 실행하면서 통신속도로 9600 으로 설정한 다음 sum 변수를 만들고 0 을 값으로 넣는다. 세번째 줄에서는 sum 변수에 있는 값을 1로 바꾼다.

```
lineSum(4);
```

그다음 lineSum() 함수를 실행하면서 인수로 4를 넣는다. lineSum() 함수를 호출하면서 함수안에 있는 sum 변수를 만든다. lineSum() 함수안에 만들어지는 변수는 함수 외부에서 만들어진 변수와 이름은 같지만 서로 다른 공간에 만들어지므로 별개의 변수로 취급된다. 마치 서울에 사는 '길동이'와 부산에 사는 '길동이'는 이름이 같지만 다른 사람이 것처럼 setup() 함수 안에서 만들어진 sum 변수와 lineSum() 함수안에 만들어진 sum 변수는 같은 이름 다른 변수가 된다.

B. 전역변수(global variable)

지역변수가 변수가 만들어진 특정 공간 안에서만 의미가 있는 것이라면 전역변수(global variable)는 한번 만들어지면 프로그램이 종료될 때까지 없어지지 않고 계속 사용되는 것이다. 일반적으로 전역변수는 함수 바깥에서

만들어진다.

```
int gSum = 0;

void lineSum(int num)
{
    for (int i = 1; i<=num; i++)
    {
        gSum += i;
    }
    Serial.println(gSum);
}

void setup()
{
    Serial.begin(9600);
    lineSum(4);
    Serial.println(gSum);
}

void loop(){}
```

함수의 바깥에서 전역변수 gSum 을 선언했다. 함수의 외부에서 선언된 gSum 은 어떤 함수에서도 사용할 수 있는 전역변수가 된다. 지역변수가 함수의 내부에서 선언되어 함수 안에서만 의미있는 값을 가질 수 있었다면 함수의 외부에서 선언된 전역변수는 프로그램이 종료될 때까지 계속 살아 있는 변수가 된다.

그리면 지역변수와 전역변수 중 어느것을 사용하는 것이 좋을까? 사용의 편리성을 따진다면 전역변수가 월등하다. 한번 선언하면 그 다음부터는 신경쓸 필요없이 언제나 어디서든지 사용할 수 있다. 반면 지역변수는 선언된 장소를 벗어나면 더 이상 변수가 살아있지 않기 때문에 다른 방법들을 사

용해서 필요한 정보를 전달해야만 한다. 그래서 상대적으로 짧은 코드를 사용하는 마이크로프로세서의 경우 전역변수를 종종 사용하게 된다.

반면 전역변수를 사용함으로 발생하는 문제도 만만치 않다. 우선 여러 함수를 사용하고, 여러 라이브러리를 사용하게 되면 중복되는 변수이름이 발생할 가능성이 커진다. 여러명의 프로그래머가 각자의 분야를 맡아서 코드를 만들 때 A 가 사용한 sum 이라는 변수를 B나 C도 사용할 가능성이 있다. 이때 sum 이라는 변수를 전역변수로 만들었다면 상당히 곤란한 경우를 만날수 있다. A 가 sum 이라는 변수에 어떤 값을 넣었을때 B 가 다시 sum 을 수정하게 되면 A 가 만든 코드는 원치 않는 값이 들어가서 결국은 에러를 발생하게 된다.

또 전역변수의 경우 프로그램이 종료되기 전까지 메모리를 계속 유지한다는 것도 문제가 된다. 함수안에 있는 변수는 함수가 호출되고, 종료될 때 메모리에 있는 모든 것이 사라진다. 즉, 함수가 사용한 메모리는 함수가 종료되면서 모두 지워지고, 그 공간을 다른 함수가 사용할수 있게 비워진다. 하지만 전역변수로 만들어진 변수는 메모리를 차지하고 프로그램이 종료되기까지 자리를 계속 차지하게 된다. 메모리를 많이 사용하는 코드의 경우 전역변수를 사용하게 되면 메모리문제가 발생하게 된다.

[지역변수 사용]

함수호출 --- 함수 내부의 지역 변수 사용 : 메모리 사용

함수종료 --- 함수가 사용했던 메모리를 삭제하여 빈 공간으로 만들어 줌

[전역변수 사용]

전역변수 생성 --- 메모리 사용

함수호출 --- 전역변수 사용

함수종료 --- 함수가 종료되어도 전역변수는 메모리에 그대로 남아있음

이런 특성을 때문에 전역변수의 사용은 가능한 자체하는 것이 C프로그래밍의 기본이다. 코딩테스트 등에서 짧은 시간에 알고리즘문제를 풀어야하는 특별한 경우를 제외하고는 가능한 전역변수의 사용을 자체하는 것이 좋다.

C. 동적변수(dynamic variable)

동적변수(dynamic variable)는 프로그램 수행 중에 동적으로 생성되거나 소멸되는 변수를 의미한다. 반대되는 개념으로 정적변수가 있다. 정적변수는 동적변수와 달리 프로그램 수행시 만들어진 후 종료될 때까지 소멸되지 않는다. 정적변수는 전역변수의 특성을 가지고, 동적변수는 지역변수의 특성을 가진다.

기본적으로 함수내부에 만들어지는 변수인 지역변수는 동적변수가 된다. 함수 호출과 함께 만들어지는 동적변수는 함수의 공간에 생성되고, 함수가 종료되면 소멸된다. 동적변수를 표시하기 위한 별도의 키워드로 auto 가 있지만 auto 를 붙이지 않아도 자동으로 동적변수로 취급된다. auto 는 지역변수에 사용되는 것이므로 전역변수의 앞에는 붙을 수 없다.

```
int funct()
{
    auto int dnum=0;
    return dnum;
}
```

위 함수는 auto 를 생략해서 다음과 같이 쓸 수 있다.

```
int funct()
```

```

{
    auto int dnum=0;
    return dnum;
}

```

C 의 auto 키워드와 C++ 에서의 auto 키워드는 다르다. 그래서 실제 위에 사용했던 auto int dnum=0; 를 아두이노에서 사용하면 문법오류가 발생한다. 이 오류는 auto 키워드에 대한 것으로 아두이노는 C++ 컴파일러를 사용하고 있기 때문에 발생한다.

즉, 다음과 같이 가장 왼쪽은 C 에서는 바른 문법이지만 C++ 에서는 오류를 발생시킨다.

(1) C 고유 문법	(2) C / C++ 바른 문법	(3) C++ 고유 문법
auto int dnum = 0;	int dnum = 0;	auto dnum = 0;
auto float fnum = 10.0;	float fnum = 10.0;	auto fnum = 10.0f;
...

그림 136 C에서는 되지만 C++에서는 안되는

많이 사용되는 것은 아니지만, C++ 에서 auto 키워드를 사용하면 뒤에 나오는 자료형에 따라 변수의 타입이 자동으로 결정된다. auto 키워드는

D. 정적변수(static variable)

함수 내부에 선언된 지역변수가 아닌 모든 변수는 정적변수(static variable)가 된다. 또한 함수 내부에서 static 으로 선언된 변수도 전역변수의 특성을 가지게 된다. 단, static 으로 선언된 변수는 동일한 범위 내에서만 참조할 수 있다.

```

int staticNumIncrese()
{
    static int num = 0;
    num++;
    return num;
}

void setup()
{
    Serial.begin(9600);

    staticNumIncrese();
    staticNumIncrese();
    Serial.println(staticNumIncrese());
}

void loop() {}

```

num 변수는 static 변수이기 때문에 staticNumIncrese() 함수가 호출될 때 한번만 초기화된다. 즉, staticNumIncrese() 가 처음 호출될 때는 num 이라는 변수를 만든 후 0 이라는 값을 넣고, 바로 그 다음줄 num++; 를 실행하면서 num에 1이 저장된다. static 변수로 선언되었기 때문에 num 은 함수가 종료되더라도 변수가 사라지지 않는다. 이후에 다시 staticNumIncrese() 함수가 호출되면 초기화 부분을 건너뛰고 나머지를 실행시킨다. staticNumIncrese() 함수가 실행될 때마다 num 값이 1 씩 증가 한다.

staticNumIncrese() 함수를 처음 실행하면 (1) 정수형인 num 변수를 static 으로 만들고 0 이라는 초기값을 넣는다. (2) num++ 을 실행시켜서 num 변수의 값을 1 증가시킨다(num=1). 그다음 다시 staticNumIncrese() 함수를 실행하면 (3) 이미 num 변수가 만들어져 있기 때문에 다시 변수를

만들지 않고 (4) num++ 를 실행시켜서 num 변수를 다시 1 증가시킨다 (num=2).

```
int staticNumIncrease()
{
    static int num = 0;
    num++;
    return num;
}

void setup()
{
    Serial.begin(9600);
    staticNumIncrease(); // ----->>
    staticNumIncrease(); // ----->>
    Serial.println(staticNumIncrease()); //--->>
}

void loop() {}
```

```
static int num = 0;
num++;
return num;
```

```
static int num = 0;
num++;
return num;
```

```
static int num = 0;
num++;
return num;
```

그림 137 staticNumIncrease() 함수 구동

정적변수를 함수 안에 위치시키면 함수내부에서만 정적변수에 접근할 수 있다. 함수 밖에 정적변수를 위치시키면 전역변수와 비슷하게 사용할 수 있다. 단, 이 경우 정적변수가 선언된 파일 내부에서만 사용할 수 있다.

즉 static 변수는 변수에 접근하는 방식에 있어서는 지역변수와 같고, 변수의 값을 저장하는 방식에 있어서는 전역변수와 같다. static 으로 선언된 변수는 프로그램 종료시까지 소멸되지 않고 메모리를 차지하고 있지만 static 변수는 어디서나 접근하지는 못하고 변수가 선언된 공간 (보통 함수 내부)에서만 접근해서 읽거나 수정하는 작업을 할 수 있다.

E. volatile 변수

변수의 앞에 volatile 을 붙이면 최적화에서 제외된다. 어떤 변수가 최적화에서 제외되면 그 변수를 사용할 때마다 메모리에서 직접 읽어오게 된다.

```
volatile int num = 0;
```

마이크로프로세서를 사용하는 경우 시간함수를 사용하면서 일정시간마다 특정한 신호를 만들기 위해 volatile 변수를 사용하게 된다. volatile 이 없으면 컴파일러는 최적화를 진행한다. 이를테면 다음과 같은 반복문이 있는 경우다.

```
for (i=0; i<=10000; i++);
```

위의 반복문이 실행되는 구조를 앞에서 보아서 알고 있다. i 는 0 부터 시작해서 1 씩 커지면서 10000이 될때까지 증가한다. for 문의 특성상 그 뒤에 명령구문이 있어야 하는데 위의 코드를 보면 아무것도 없다. 즉, i 는 0 부터 10000 까지 1씩 증가하되, 다른 일은 전혀 하지 않는 코드다. 이런 코드는 일반적으로 PC 에서는 사용하지 않고, PC 에서 컴파일할때는,

```
i = 10000;
```

으로 대체되어도 무관하다. 오히려 컴파일러는 가능한 계산을 적게하면서 동일한 결과를 만들어 내려고 하기 때문에 위의 for 문을 만나면 아래 i=1000; 의 대입문으로 치환해버린다. 이를 최적화라한다. 굳이 피해도 되는 반복작업을 피하고 결과를 빨리보여주는 것이 일반적으로 좋은 컴파일

러다. 동일한 결과를 보이는데 A 컴파일러는 0.1초가 걸리지만 B 컴파일러는 10초가 걸린다면 당연히 0.1초로 결과를 보여주는 컴파일러의 성능이 더 좋은 것이다.

PC 기반의 C 컴파일러들은 이런 최적화작업을 알아서 진행하고, PC 기반에서 프로그램을 만들때는 volatile 을 사용하는 경우는 극히 드물다. 반면, 마이크로프로세서를 사용할 때는 두가지 정도의 이유로 volatile 변수를 사용하게 된다.

첫째는 위에서 잠시 언급했던 의도적 지연효과를 이용하는 delay() 함수를 만들어서 사용할 때이다. OS 위에서 프로그래밍을 하는 컴퓨터와는 달리 마이크로프로세서는 하드웨어에 직접 접속하게 된다. 즉, 어떤 특별한 신호를 주거나 혹은 받을 때 필요한 타이밍을 직접 만들어서 제어해야 한다. 이를테면 가속도신호를 받을 수 있는 A 칩에서 신호를 읽을 때 신호와 신호 사이에 일정 시간의 간격을 두어야만 한다면 그 시간을 정확히 만들어낼 때 사용된다. MPU-6000 같은 가속도센서의 값을 읽어올 때 하드웨어적으로 정확히 맞추어야 하는 시간들이 있다. 대부분 10ns 에서 500ns 정도의 값 을 가지게 된다. 이러한 시간을 만들고자 할때 volatile 을 사용한다.

둘째는 인터럽트를 사용할 때다. 인터럽트는 MCU 안에 있지만 별도로 움직이는 시스템이라고 보아도 무방하다. 인터럽트에 대해서는 이후 별도의 장에서 소개한다. 이곳에서는 간단하게 volatile 에 관해서만 짧게 설명하겠다.

인터럽트는 미리 정해둔 몇가지 특수한 경우에 대해서 기존 MCU 가 하던 일을 강제로 일시정지시키고 인터럽트로 들어온 일을 우선처리한다. 인터럽트는 가능한 빨리 일을 처리하려고 하기 때문에 꼭 필요한 일들만 빨리 처리하려고 한다. 인터럽트로 처리되는 일은 메모리에 저장될 수 있다. 만약 인터럽트가 메모리에 신호를 저장하기 이전에 MCU 가 메모리에 있던 값 은 별도의 레지스터에 담아두고 있었다면 레지스터에 담아있는 값과 메모

리에 있는 값은 다른 값이 된다. MCU에서 메모리의 값을 읽어올 때, MCU에서 그 메모리의 값을 수정한 적이 없다면 MCU는 메모리 대신 메모리가 임시 저장된 레지스터에서 값을 읽어오게 된다.

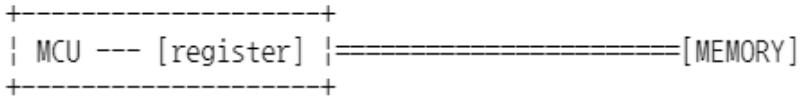


그림 138 레지스터와 메모리

레지스터를 간단한 이해하기 위해서 컴퓨터의 캐시메모리를 생각하면 된다. 컴퓨터에서 가장 많이 시간이 걸리는 것은 외부에서 신호를 읽어오는 것으로 HDD나 SSD에서 데이터를 가져오는 것이다. 그래서 컴퓨터는 중요한 데이터를 HDD나 SSD 대신 RAM에 넣어두고 사용한다. 실제로 컴퓨터가 프로그램을 실행시킨다는 것은 HDD/SDD에 있는 실행파일의 내용을 RAM으로 옮긴 후 RAM에 있는 내용을 실행시키는 것이다. 이때 RAM에 있는 내용을 빈번하게 자주 읽어오는 데이터는 별도의 캐시메모리에 저장한다. 캐시메모리는 RAM이 아닌 CPU에 포함되어 있는 가장 빠른 메모리로 처리속도는 CPU와 거의 동일하다.

MCU에 있는 레지스터는 CPU의 캐시메모리와 유사한 역할을 한다. MCU가 자주 읽어서 사용하는 데이터를 레지스터에 저장해 둔다. MCU는 데이터가 수정되면 램 영역을 수정하고, 수정된 값을 다시 레지스터로 가져온다. 이때 인터럽트가 발생하면 MCU와는 별도의 시스템이 작동해서 메모리를 수정하게 된다. 하지만 인터럽트는 레지스터를 수정하지는 않는다. 이로 인해서 레지스터에 있는 값과 메모리에 있는 값이 달라질 수 있다. 이 경우가 MCU를 사용할 때 문제가 되는 부분이고, 이러한 이유 때문에 인터럽트를 사용할 때는 volatile 변수를 사용해서 최적화를 하지 않게 된다. 최적화를 하지 않으면 MCU는 레지스터 변수를 사용하지 않고 직접 메모리에 있는 값을 읽어오게 된다. 이로 인해 약간의 시간 손실은 발생하지만 데이터가 충돌하지 않고, 제대로 신호를 읽고 쓸 수 있게 된다.

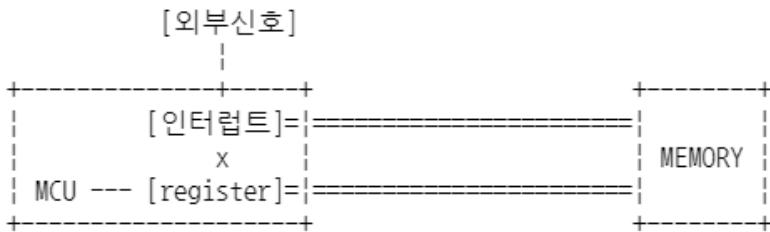


그림 139 인터럽트

`volatile` 변수를 사용해서 최적화를 하지 않는 경우에 대한 코드는 다음과 같다.

```

void setup()
{
    long i, ltime;
    volatile long vi;

    Serial.begin(9600);
    delay(1000);

    // volatile 변수 사용하지 않은 경우 (i)
    ltime = micros();
    for (i=0;i<=1000;i++);
    Serial.println(micros()-ltime);      // 4

    ltime = micros();
    for (i=0;i<=10000;i++);
    Serial.println(micros()-ltime);      // 4

    ltime = micros();
    for (i=0;i<=100000;i++);
    Serial.println(micros()-ltime);      // 4
}

```

```

// volatile 변수 사용한 경우 (vi)
ltime = micros();
for (vi=0;vi<=1000;vi++);
Serial.println(micros()-ltime);    // 2224

ltime = micros();
for (vi=0;vi<=10000;vi++);
Serial.println(micros()-ltime);    // 22096

ltime = micros();
for (vi=0;vi<=100000;vi++);
Serial.println(micros()-ltime);    // 220100
}

void loop() {}

for 문을 사용해서 1000, 10000, 100000 번 반복할 때 걸리는 시간은 다음과 같다. 이 테스트는 아두이노 우노를 사용하였으며, 아두이노 IDE 1.8.13 을 사용하였다. 아두이노 버전에 따라 컴파일 옵션이 달라질 수 있고, 최적화 정도가 달라질 수 있으며 이로 인한 약간의 시간차는 발생할 수 있다.

```

반복횟수	시간 : long	시간 : volatile long
1,000	4	2,224
10,000	4	22,096
100,000	4	220,100

그림 140 volatile 최적화 회피

반복횟수가 1,000 번이든 100,000 번이든 volatile 변수를 사용하지 않아 최적화가 발생한 경우의 for 문 실행시간은 모두 $4\mu\text{s}$ 인것을 알 수 있다. 반면 volatile 변수를 사용한 경우 반복횟수가 10배씩 늘어남에 따라 걸린 시간도 10배씩 증가한 것을 알 수 있다.

9.10. 재귀함수

함수는 다른 함수를 호출할 수 있다. 이전 단계에서 우리는 setup() 이라는 함수 안에서 hello() 라는 함수나 hi() 와 같은 함수를 호출하는 것을 배웠다. 함수는 다른 함수를 호출할 수 있다. 그 뿐만 아니라 함수는 자신을 호출할 수도 있다. 이런 함수를 재귀함수라고 한다. 즉, 함수안에 자신을 다시 호출하는 명령이 포함되어 있으면 이를 재귀함수라고 한다.

재귀함수의 대표로 팩토리얼을 구하는 함수가 있다. 팩토리얼의 수학적 정의는 다음과 같다.

$$\begin{aligned} n! &= n * (n-1)! \\ 0! &= 1 \end{aligned}$$

함수를 만들때 $n!$ 의 반환값은 $n*(n-1)!$ 이 된다. 즉, fact(n) 라는 함수의 반환값은 $n*fact(n-1)$ 이다. 이때 n 이 0 이라면 1 을 반환한다. 이를 코드로 만들면 다음과 같다.

```
long fact(int num)
{
```

```
if (num==0) return 1L;
return num * fact(num-1);
}

void setup()
{
    int num = 3; // 팩토리얼함수에 들어갈 인자값
    Serial.begin(9600);
    long lnum = fact(num);
    Serial.println(lnum);
}

void loop() {}
```

코드를 통해 보듯이 함수가 자기 자신을 다시 호출하는 구조를 가지고 있다. 이런 구조를 가진 함수를 재귀함수라고 부른다. 모든 함수는 순차적 실행을 한다. 만약 실행 도중 다른 함수를 호출하게 되면 호출된 함수가 모두 완료된 다음 호출했던 자리로 돌아가서 남은 부분을 끝까지 실행하게 된다.

fact(3) 을 실행하면 다음과 같다.

[fact(3)] n=3 return 3*fact(2)	[fact(2)] n=2 return 2*fact(1)	[fact(1)] n=1 return 1*fact(0)	[fact(0)] return 1
return 3*2	return 2*1	return 1*1	

그림 141 fact(3) 실행

fact(3) 재귀함수를 호출하면 $3*fact(2)$ 를 반환해야 하는데 fact(2) 를 알기 위해 다시 fact() 함수를 호출하면서 인수 2를 주게된다. 이때 fact(3) 는 아직 종료되지 않은 상태로, fact(3) 의 반환값인 $3*fact(2)$ 에서 fact(2) 가 아직 결정되지 않은 상태다. fact(3) 는 종료하지 않고 지금까지의 과정을 메모리에 저장해두고 fact(2) 를 호출한다. fact(2) 를 호출하면 $2*fact(1)$ 을 반환해야 하는데 여기서 다시 fact(1)이 결정되지 않은 상태이므로 지금까지와 과정을 메모리에 저장해 둔 상태로 fact(1) 을 다시 호출한다. 이렇게 fact(0)까지 호출하면 fact(0) 에 반환값 1이 결정된다. fact(0)의 반환값 1을 fact(1) 에서 $1*fact(0)$ 으로 반환해준다. fact(1) 은 연산결과인 $1*1$ 을 fact(2) 로 반환하고, fact(2) 는 $2*1$ 을 fact(3) 로 반환한 후 fact(3) 는 $3*2$ 값을 반환한다.

이 과정 중 함수 호출과 종료에 대한 부분은 다음과 같다.

fact(3) 를 호출한다.

fact(2) 를 호출한다.

fact(1)을 호출한다

fact(0)을 호출한다.

fact(0)에서 반환값 1을 fact(1) 으로 반환하고 fact(0) 은 소멸한다.

fact(1)에서 반환값 1을 fact(2) 으로 반환하고 fact(1) 은 소멸한다.

fact(2)에서 반환값 2를 fact(3) 으로 반환하고 fact(3) 은 소멸한다.

fact(3)에서 반환값 6을 fact(3)을 호출한 쪽으로 반환하고 fact(3) 은 소멸한다.

재귀함수의 경우 재귀의 깊이가 깊어질수록 메모리의 사용량이 커진다. fact(10) 은 fact() 함수를 11번 호출한 후 다시 역순으로 소멸된다. 만약 fact(1000) 이 되면 fact() 함수를 1001 번 호출해야하고, 이는 상당히 많은 메모리를 사용하게 됨을 의미한다. 실제로 아두이노 우노와 같은 작은 메모리를 가진 MCU 의 경우 이러한 재귀함수는 금방 메모리가 고갈되므로 재귀함수의 사용은 주의깊게 사용해야한다.

9.11. 함수 이용

함수에 대해 간략히 알아보았다. 이제 함수를 사용해보도록 한다. 첫째 우리가 사용할 수 있는 기본 제공되는 함수들이 있다. 이 함수는 이미 만들어져서 제공되는 것이므로 그저 사용하기만 하면 된다. 대표적으로 이미 앞에서 사용해왔던 pinMode() 가 있다.

pinMode() 함수는 다른 C 컴파일러에서는 존재하지 않는다. 오직 아두이노에서만 사용할 수 있다. 즉, 이렇게 아두이노 컴파일러가 내부적인 사용을 위해 이미 만들어서 제공하는 함수를 내부함수라고 부른다. 내부함수는 다시 표준 C언어가 제공해주는 함수들과 아두이노에서 제공해주는 함수로 구분할 수 있다. 표준 C언어가 제공하는 함수들은 어떤 컴파일러에서도 사용할 수 있다. 하지만 아두이노에서 제공하는 내부 함수는 다른 환경에서는 사용할 수 없다. 즉, 아두이노에서 사용가능한 pinMode() 함수는 PC에서

프로그램을 할 때는 사용할 수 없다.

또한 아직 존재하지 않는 기능을 사용자가 임의로 편집하여 작성한 사용자 정의 함수가 있다. 위에서 우리가 만들었던 함수들은 모두 사용자 정의 함수가 된다. 즉, 이미 만들어져서 제공되는 함수는 그 사용법을 익혀서 사용하면 된다. 필요하지만 아직 존재하지 않는 기능은 사용자 정의 함수로 만들어서 사용하면 된다. 사용자 정의 함수는 만들고(선언과 정의), 사용(호출)하는 방법은 이미 위에서 다루웠기 때문에 여기서는 아두이노 내부 함수와 수학 함수 중 자주 사용되는 것을 다루도록 한다.

A. 아두이노 내부 함수

a. pinMode(핀번호, 핀상태)

내부 함수인 pinMode() 함수는 다음과 같이 사용된다.

```
pinMode(13, OUTPUT); // 13번 핀을 출력으로 설정하라
```

pinMode() 함수는 일반적으로 void setup() 안에서 특정 핀의 입력과 출력을 지정하기 위해 사용된다.

```
pinMode(13, OUTPUT); // 13번 핀을 출력으로 설정하라  
pinMode(12, INPUT); // 12번 핀을 입력으로 설정하라  
pinMode(11, INPUT_PULLUP); // 11번 핀을 풀업된 입력으로 설정하라
```

여기서 풀업, INPUT_PULLUP의 의미는 아두이노 MCU의 내부에서 20k Ω 의 저항이 핀과 VCC(5V) 사이에 들어간다는 것을 의미한다. Pull Up과 Pull Down은 입력 핀이 HIGH 또는 LOW에 연결되지 않은 애매한 상황

인 경우 기본 값을 HIGH로 할지 LOW로 할지를 결정해준다. 이는 아두이노와 같이 직접적으로 전기를 사용하는 하드웨어에 필요한 내용이다.

약간의 부가적인 설명을 더하면 pinMode() 함수는 2개의 매개변수를 필요로 한다. 첫번째 매개변수에는 핀의 번호가 들어가고 두번째 매개변수에는 핀을 출력으로 사용할지, 입력으로 사용할지의 여부가 들어간다. 첫번째 매개변수는 핀의 번호이므로 정수값이 들어가면 된다. 하지만 두번째 매개변수는 핀의 상태를 나타내는 것이므로 이미 약속된 값이 들어가야 한다. 즉, 입력으로 쓰겠다고 내 마음대로 "입력"이라고 쓰거나 "Input", "input", "IN" 등으로 쓰면 안된다는 뜻이다. 이미 약속된 "INPUT" 외에는 사용할 수 없다. 그리고 여기 사용된 INPUT은 Arduino.h 파일에 정의되어 있다.

Arduino.h 파일의 내용

```
...
#define INPUT 0x0
#define OUTPUT 0x1
#define INPUT_PULLUP 0x2
...
```

상수에서 설명하지 않는 부분이다. #define 은 그 바로 뒤에 하나의 상수 이름과 그 상수에 들어갈 값을 가진다. #define 의 보다 복잡한 사용법은 별도로 공부하기로 하자. 여기서는 #define 은 단순한 매크로상수를 정의하는 기능만 다루도록 한다.

그리고 #define 은 컴파일 되는 시간의 측면에서 볼때는 전처리기로 분류된다. 즉, #define 이 있는 줄은 컴파일이 일어나기 전에 미리 처리되는 구문이다. #define INPUT 0x0 을 만나면 컴파일하기 전에 문서 전체에 대해서 INPUT이라는 단어가 있는지를 찾은 후 그 단어대신 0x0 으로 치환한다. 그래서 pinMode(13, INPUT) 은 pinMode(13, 0x0) 이 된다. 0x0

은 정수 0을 16진 표기로 쓴 것이라면 pinMode(13, 0) 과 같다.

b. digitalWrite(핀)

아두이노 입력으로 설정된 핀의 디지털값을 읽는다. 일반적으로 핀에 걸린 전압이 VCC 이면 1 을, GND 이면 0 을 반환한다.

```
value = digitalRead(12); // 12번 핀을 읽어서 value 변수에 저장  
단, digitalRead()를 쓰기 전에 pinMode()를 사용해서 핀의 출력을 설정해  
두어야한다.
```

```
pinMode(12, INPUT);  
value = digitalRead(12);  
// 12번 핀을 읽어서 value 변수에 저장  
// value에는 0 또는 1이 저장됨
```

c. digitalWrite(핀, 값)

아두이노 출력핀에 정해진 값을 출력한다. 값은 0 과 1 둘 중의 하나이다.

```
digitalWrite(11, 1); // 11번 핀에 1(+5V)을 출력한다.  
// 이것을 다음으로 쓸 수 있다.  
digitalWrite(11, HIGH); // HIGH는 1과 같고, LOW는 0과 같다.
```

d. analogRead(핀)

아날로그 핀에 연결된 전압값을 읽는다. 아두이노 우노와 메가2560에서는 0에서 5V사이의 전압을 2의 10승인 1024단계로 나눈 0에서 1023사이의 단계값으로 읽는다.

```
value = analogRead(A0); // 핀 A0에 연결된 전압을 읽어온다.  
// value에 저장되는 값은 최소 0에서 최대 1023 사이 정수이다.
```

아두이노에서 아날로그 값을 읽는 것은 0V에서 5V 사이의 어떤 전압값을 읽는 것이다. 아두이노는 MCU의 분해능에 따른 비트로 값을 읽는다. 우노와 메가2560은 AVR 8비트 MCU를 사용하고 있으며 자체적으로 10개 비트로 값을 분해할 수 있는 ADC를 가지고 있다. 즉, 10개 비트로 분해된 단계를 감지한다. 10개의 비트는 2의 10승을 의미하고 이는 총 1024 가 된다. 최소값은 0이고 최대값은 1023이다.

아두이노에서 아날로그 입력을 받을 때 주의할 것은 아날로그 입력핀에 들어오는 전압이 0V에서 5V 사이가 되어야 한다는 것이다. 이 값을 벗어난 전압이 들어올 경우 아두이노는 MCU가 망가질 수 있다.

e. analogWrite(핀, 값)

핀으로 아날로그 값을 보낸다. 정확하게는 아날로그값을 대체하는 PWM 신호를 보낸다. 이때 핀은 PWM 출력이 가능한 핀만 가능하다. PWM은 보드의 번호 옆에 물결(~) 표시가 있는 핀으로 정해진 핀들에서만 PWM 출력을 낼수 있다. UNO의 경우 3,5,6,9,10,11 핀이 pwm 이 가능하여 analogWrite()로 값을 쓸 수 있다.

```

analogWrite(3,127);
// 아날로그 출력핀 3 번에 127 의 값인 2.5V를 출력

```

쓸 수 있는 값은 0 부터 255 까지로 PWM 의 분해능은 8 비트이다. 최대 256 단계까지 가능하다. PWM 은 2ms 의 시간 안에서 256 단계로 나누어 일부는 1 을, 나머지는 0 을 출력하는 방식으로 2ms 의 주기로 반복된다.

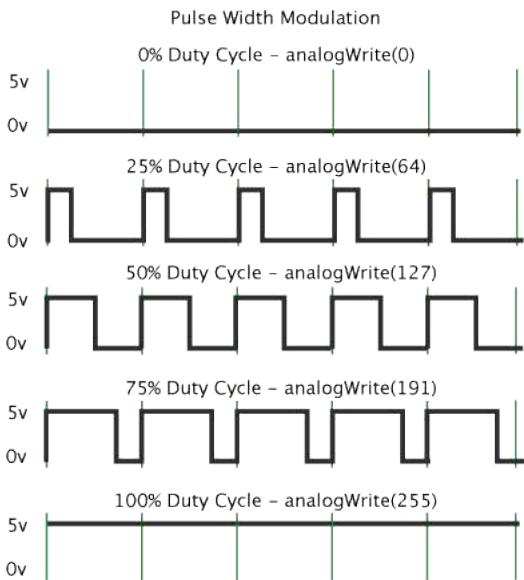


그림 142 PWM 과정

PWM 이란 Pulse Width Modulation 의 약자로 펄스의 폭을 조절하는 것이다. 아두이노에서는 2ms 즉, 0.002 초 안에서 ON 과 OFF를 나눠서 나가는 출력을 조절한다. 그래서 염밀하게는 Digital Analog Converter 가 아니지만 모터를 제어하거나 온도를 제어하거나 LED 의 빛의 세기를 제어할 때는 PWM을 DAC 로 대체하여 사용할 수 있다.

f. delay(시간)

`delay()` 함수는 는 주어진 시간만큼 아무것도 하지 않는다. 시간의 단위는 ms 이고, 1000ms 는 1초이다.

```
delay(1000); // 1000ms 즉 1초 동안 기다림
```

`delay()` 함수를 쓸 경우 주의할 점은 함수 실행 중인 시간동안 다른 작업을 하지 못한다는 것이다. 만약 스위치입력을 받아야 한다면 `delay()` 함수는 사용을 피하는 것이 좋다. `delay()` 되는 시간 동안 스위치 작동이 있다면 아두이노에서는 스위치 입력을 인식하지 못하게 된다.

g. `millis()`

현재 시간을 반환한다. 반환되는 시간은 아두이노에 전원이 들어온 뒤로 흐른 시간을 의미한다. 단위는 ms이고, `milis()` 함수를 사용해서 시간을 저장할 때는 `unsigned long` 타입의 변수를 일반적으로 사용한다.

```
unsigned long value;  
value = millis();
```

`milis()` 함수를 사용하면 `delay()`를 대체할 수 있다. 아래 소스는 `delay()`를 사용하지 않고도 동일한 효과를 내는 방법을 보여준다.

```
unsigned long previousMillis = 0;  
unsigned long currentMillis = millis();
```

```

if (currentMillis - previousMillis >= 1000) {
    previousMillis = currentMillis;
    if (ledState == LOW) { ledState = HIGH; }
    else { ledState = LOW; }
}

```

1초마다 한번씩 LED 가 켜지고 꺼지는 것을 반복하지만 delay() 함수를 사용하지 않았기 때문에 다른 작업을 같이 할 수 있다. 이를테면 센서로부터 값을 읽어오거나 스위치 입력을 받거나 모터를 구동시키는 등의 작업을 동시에 할 수 있게 된다.

B. 다양한 수학 함수

`min(x,y)` : 둘 중 작은 수를 반환

`max(x,y)` : 둘 중 큰 수를 반환

`randomSeed(value)` : `random()` 함수의 시작지점을 결정할 때 사용
보통 오픈 상태의 `analogRead()` 값을 사용

`random(max); 0` 과 `max` 사이의 값 중 하나를 무작위로 만든다.

`random(min,max); min` 과 `max` 사이의 값 중 하나를 무작위로 만든다.

```

int number;
randomSeed(analogRead(A0)); // random() 초기화
number = random(1000);
    // 0 과 1000 사이의 값을 무작위(random)로 만듭니다.
number = random(50,200);
    // 50 과 200 사이의 값을 무작위(random)로 만듭니다.

```

`analogRead(A0)` 는 A0 핀에 들어오는 아날로그값 (0V ~ 5V)이다. A0에

아무것도 연결하지 않은 플로팅 상태에서 노이즈에 의한 값이 들어오면 그 값으로 random() 함수를 초기화시킨다.

이 외에 sin(), cos() 등의 함수도 사용 가능하지만 아두이노 우노와 메가2560은 8비트 MCU를 사용하고 있고 실수연산은 매우 느리기 때문에 권하지 않는다. 실수 연산이 필요해도 아두이노 우노와 메가2560에서는 정수로 변환한 다음 연산을 진행하는 것이 좋다. 실수 연산이 반드시 필요하면 그때는 우노와 메가2560을 사용하지만 말고 실수 연산을 지원하는 MCU를 가진 아두이노를 쓰는 것이 좋다. 이를테면 아두이노 듀에나 최근에 나온 아두이노 우노 r4는 32비트 Arm 계열의 MCU를 사용하고 있으며 실수 연산을 지원하므로 이런 종류의 보다 고급형 MCU가 탑재된 아두이노를 사용하기 바란다.

Chapter 10

주요 학습 목표

- 배열에 대한 이해
- 일차원, 이차원, 다차원 배열 연습

Chapter 10. 배열

10.1. 배열이란

배열을 이용하면 같은 자료형의 변수들을 한꺼번에 다룰 수 있다. C에서 는 집합형 자료구조로 배열이 사용된다. 온도를 측정해서 최근 온도 10개 를 사용하는 프로그램을 만들때 10개 온도를 저장하기 위해 변수가 10개 필요하다. 이때 일일이 하나씩 이름을 붙여서 변수를 만드는 대신 크기 10 짜리인 배열을 사용하면 된다. 같은 방법으로 크기 100짜리, 크기 1000짜 리 배열을 만들 수 있다.

배열(array)이란 같은 자료형의 변수들을 하나의 이름으로 만들어 순서에 따른 인덱스 번호로 접근할 수 있게 만든 것이다.

```
int t[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
```

위처럼 t라는 이름으로 크기 10인 배열을 만들게 되면 다음과 같이 메모 리를 사용하여 배열이 만들어진다.

t	10	11	12	13	14	15	16	17	18	19
	t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]

그림 143 배열

초기값이 없는 배열은 그냥 선언만 하면 된다.

```
int t[10];
```

배열의 첨자(인덱스)는 1이 아닌 0부터 시작한다. 배열 원소를 참조하기 위해 사용한 대괄호[]는 첨자연산자(index operator)라고 불리며 배열의 특정 원소에 접근할 때 사용된다. n번째 요소를 참조할 때 0부터의 거리를 사용하므로 n-1을 사용한다.

배열 t의 10번째 요소 표기법 : t[9]

배열은 우선 기준의 변수를 여러개 사용하던것을 깔끔하게 만들어준다. 위처럼 10개의 변수를 사용하고자 할 때 기준의 방식을 사용하면,

```
int temp0 = 10, temp1 = 11, temp2 = 12, temp3 = 13, ....
```

처럼 매번 변수의 이름과 초기값을 넣어주어야 한다. 또한 개별 값을 읽거나 수정할 때도 변수의 이름이 항상 들어가야만 한다. 변수의 시작값을 20으로 수정하고 1씩 증가하도록 프로그램하면 다음과 같다.

```
temp0 = 20;
temp1 = 21;
```

```
temp2 = 22;  
...  
temp9 = 29;
```

반면 배열을 쓰면 위와 같은 경우 다음과 같이 간단하게 코드를 만들 수 있다.

```
for(int i=0; i<10; i++) temp[i] = 20+i;
```

배열을 사용하여 숫자 5개를 저장한 후 이를 앞에서 뒤로 출력하고, 반대로 뒤에서 앞으로 출력하는 프로그램을 만들어 보면 다음과 같다.

```
void setup()  
{  
    Serial.begin(9600);  
    int iarr[5] = {1,3,5,7,9};  
    for (int i=0; i<5; i++) Serial.print(iarr[i]);  
    Serial.println();  
    for (int i=4; i>=0; i--) Serial.print(iarr[i]);  
    Serial.println();  
}  
  
void loop(){}  
 
```

[실행결과]

```
13579  
97531
```

10.2. 배열 선언과 초기화

배열초기화

변수는 선언과 동시에 초기화하는 것이 좋다. 사용자가 초기화 하지 않은 변수는 특정 컴파일러에서는 특정값으로 초기화하는 경우도 있다. 초기화하지 않은 변수로 인해 발생할 수 있는 위험을 고려한다면 가능한 한 최선을 다해 변수들을 초기화하는 것이 바람직하다.

변수 선언과 함께 초기값을 넣어줌으로써 사용자의 의도가 명확하게 드러날 수 있고, 초기화되지 않은 변수가 가진 쓰레기값 때문에 발생할 수 있는 오류나 위험에서 벗어날 수 있다.

배열은 변수 이름 뒤에 [] 를 사용하여 크기를 결정한다. 배열을 선언하면서 값을 초기화할 때는 { } 를 사용한다. 다음은 배열을 생성하는 몇 가지 방법이다.

```
int myInts[6];
int myPins[] = {2, 4, 8, 3, 6};
int mySensVals[6] = {2, 4, -8, 3, 2};
char message[6] = "hello";
```

배열 선언시 크기가 정해져야한다.

배열은 기본적으로 크기가 결정되어야만 한다. int myInts[6]; 는 크기가 6인 int 형 배열이다. myInts[6] 는 int 형 변수 6개를 담을 수 있는 배열로, 배열의 이름은 myInts 이다. 크기는 결정되었지만 배열에 담기는 값은

결정되지 않았기 때문에 어떤 값이 들어가 있는지는 모르는 상태다. 이렇게 크기만 가지고 만들어진 배열은 내부적으로 의미없는 값(쓰레기값, garbage 라고도 불림)이 들어간다.

myInts	?	?	?	?	?	?
index	[0]	[1]	[2]	[3]	[4]	[5]

그림 144 배열 생성 직후 내부 상태

크기를 명시하는 대신 초기값을 넣으면 배열을 정의할 수 있다.

int myPins[] = {2, 4, 8, 3, 6}; 처럼 대입연산자(=) 오른쪽에 {}로 묶인 초기값을 쓰게 되면 배열의 초기화가 이루어지면서 자동으로 배열의 크기가 결정된다. 즉, myPins 배열은 정수형(int) 배열로 인덱스 번호 0부터 순서대로 2, 4, 8, 3, 6 이 들어간다. 초기화되면서 자연스럽게 크기가 5인 배열이 된다.

myPins	2	4	8	3	6
index	[0]	[1]	[2]	[3]	[4]

그림 145 초기화된 배열

배열의 부족한 초기값은 숫자 0 또는 널문자(\0)로 채워진다.

int mySensVals[6] = {2, 4, -8, 3, 2}; 처럼 배열의 크기와 초기값이 모두 포함된 상태로 배열을 만들 수 있다. 이때 배열의 크기가 6이고, 초기화되는 인덱스의 번호가 5개이면 마지막 하나는 0으로 초기화된다. 정수형 변수의 경우 0, 실수형 변수의 경우 0.0, 문자형 변수의 경우 \0 인 NULL

문자로 초기화된다.

```
int mySensVals[6] = {2, 4, -8, 3, 2};
```

mySensVals	2	4	-8	3	2	0
index	[0]	[1]	[2]	[3]	[4]	[5]

그림 146 배열 초기화

배열의 크기를 6으로 하고 5개 원소로 초기화하면 남은 하나의 자리에는 0이 들어간다. 정수형 배열일때는 정수 0이 들어가고, 실수형 배열일때는 실수 0.0이 들어간다.

```
float myTempVals[6] = {1.0, 2.1, -1.1};
```

myTempVals	1.0	2.1	-1.1	0.0	0.0	0.0
index	[0]	[1]	[2]	[3]	[4]	[5]

그림 147 실수 배열 초기화

문자열을 배열에 저장할 경우 남은 곳에 아스키코드 0x00 번인 NULL(혹은 NUL) 문자가 들어간다. NULL 문자는 NUL 또는 \0 으로도 표기한다. NULL 문자는 특수문자로 문자열을 다룰때 사용된다. 문자열 내부에서 NULL 문자를 만나는 순간 그 지점을 문자열의 끝으로 인식한다. 크기가 7인 char 타입의 배열을 만들고 5개의 문자로 초기화하면 남은 2개의 공간은 NULL 문자로 채워진다.

이렇게 만들어진 myHello 배열을 출력하면 index 5 인 \0(NULL문자)를

출력하고 종료한다. 이때 myHello 의 6번째 인덱스에 다른 문자를 넣어도 그 문자는 출력되지 않는다.

```
char myHello[7] = "Hello";
```

myHello	H	e	l	l	o	\0	\0
index	[0]	[1]	[2]	[3]	[4]	[5]	[6]

그림 148 문자 초기화

10.3. 배열의 크기

배열의 크기를 구할 때 sizeof 연산자를 사용한다.

```
sizeof(배열이름);
```

위처럼 배열의 이름을 sizeof 연산자에 넣으면 배열이 차지하고 있는 전체 크기를 알 수 있다. 배열의 전체 크기를 배열 요소 하나의 크기로 나누면 요소의 개수, 즉 배열의 크기를 구할 수 있다.

```
sizeof(배열이름) / sizeof(배열이름[0]);
```

```
int arr[3] = {10, 11, 12};  
int arrsize = sizeof(arr)/sizeof(arr[0]);
```

10.4. 2차원 배열, 3차원이상 배열

2차원 배열은 1차원 배열을 요소로 하는 배열이다. 2차원 배열은 엑셀의 시트와 같은 형태로 행과 열로 구성된 가로축과 세로축으로 구성된 평면상 데이터 구조를 가진다. 2차원 배열 선언은 [] 를 두번 사용한다.

자료형 배열이름 [행크기][열크기];

3차원 배열은 2차원 배열을 요소로 하는 배열이다. 배열의 요소로 배열이 될 수 있기 때문에 몇차원이든 고차원 배열을 만들 수 있다. 일반적으로 2 차원 이상의 고차원 배열은 잘 사용하지 않는다.

```
int arr[3][3] = {  
    {0, 1, 2},  
    {3, 4, 5},  
    {6, 7, 8}  
}
```

10.5. 배열 연습

일반적으로 배열을 사용할 때 for 반복문을 사용한다. 아두이노 LED 를 깜박여서 좌우로 불빛이 움직이도록 프로그래밍해보자.

```
int pinArray[] = {2, 3, 4, 5, 6, 7};  
int count = 0;  
int timer = 100;  
  
void setup()  
{
```

```

for (count=0;count<6;count++)
{
    pinMode(pinArray[count], OUTPUT);
}
}

void loop()
{
    for (count=0;count<6;count++)
    {
        digitalWrite(pinArray[count], HIGH);
        delay(timer);
        digitalWrite(pinArray[count], LOW);
        delay(timer);
    }
    for (count=5;count>=0;count--)
    {
        digitalWrite(pinArray[count], HIGH);
        delay(timer);
        digitalWrite(pinArray[count], LOW);
        delay(timer);
    }
}

```

C언어의 배열은 문자열을 처리할 때 많이 사용된다. 하지만 아두이노와 같은 임베디드 시스템에서는 메모리 용량이 적기 때문에 PC와 같은 방식으로 문자열 처리를 하는 경우는 거의 없다. 우리는 앞으로 배열을 다룰 때 문자열의 문자들을 나눠서 담은 후 문자열 처리 함수를 사용하는 방식을 사용하지 않을 것이다. 일반적으로 아두이노와 같이 저장 공간이 적은 임베디드 시스템에서의 배열은 문자열을 담는 것이 아닌 계산을 위한 필요한 임시 숫자들을 저장하고 나열하는 방식으로 사용하게 된다.

또한 배열의 이름은 배열이 위치한 메모리상의 주소를 의미하게 된다. 이를 이용해서 C는 포인터라는 문법을 가지고 있으며 포인터를 이용해 배열에 접근할 수 있다. 즉, 배열의 요소에 접근하는 방법은 배열이름과 첨자를 이용한 방법과 포인터를 이용한 방법이 있다. 포인터를 이용한 배열 접근은 이 장에서는 다루지 않는다. 포인터는 매우 강력한 칼과 같다. 마치 관우의 청룡언월도와 같다고 보면 된다. 이는 잘 쓰면 엄청난 위력을 발휘하지만 잘못 사용하면 아군에게 막대한 피해를 입힐수 있다. 즉, 작은 실수로 시스템이 모두 큰 문제에 빠지게 될 수도 있는 것이 바로 포인터이다. 다음장에서 포인터의 기본 내용을 익히게 된다. 하지만 아두이노를 사용할 때는 굳이 포인터를 쓰지 않는 것이 좋다. 꼭 필요한 경우가 아니라면 포인터의 사용은 가능한 남용하지 않는 것이 프로그램의 안정성을 유지하는 길이다.

Chapter 11

주요 학습 목표

- 포인터의 이해
- 포인터 연산과 주의사항

Chapter 11. 포인터

모든 데이터는 기억장치, 즉 RAM에 저장된다. 기억장치에 저장되는 정보는 1) 정보의 값과 2) 정보의 위치를 가져야 한다. 이렇게 저장된 값은 변수에 담기게 되고, 변수는 저장된 정보의 위치에 대한 정보를 가지게 된다.

C언어에서 포인터는 상당한 중요한 위치를 차지한다. 포인터에 대해서만 별도의 책이 나와있고, 포인터를 이해하면 C언어를 다 알았다고 말해도 과언이 아니다. 다만 일반 PC환경에서 사용하는 포인터와 달리 마이크로프로세서, 특히 아두이노에서는 포인터의 사용이 그다지 일반적이지는 않다. 우선 아두이노의 시작이 일반인들, 컴퓨터나 프로그래밍을 전공하지 않은 이들을 대상으로 누구든지 생활속에서 익히고 만들 수 있는 활동을 지원하기 위함이므로 복잡한 배열, 포인터, MCU마다 다른 인터럽트등에 대해서는 자세히 다루지 않는다.

이번 장에서 우리는 C언어의 포인터를 다룰것이다. 하지만 아두이노에서 사용할 수 있는 정도로 가볍게 다루고 넘어가려한다. 더 깊은 포인터의 세계를 들여다보고 싶은 이들은 포인터를 다루고 있는 별도의 교재나 자료를 통해 공부하기를 바란다.

여기서 다룰 것은 포인터의 기본 개념과 포인터를 사용하여 함수와 배열을 사용하는 방법을 다룬다.

11.1. 포인터의 선언

포인터는 변수의 종류 하나의 이름이다. 우리는 이미 앞에서 정수를 저장하는 변수, 실수를 저장하는 변수 등을 배웠다. 그런데 정수나 실수가 아닌 조금 특별한 데이터를 저장하는 변수가 있다. 이 변수는 변수들이 위치한 메모리의 주소값을 저장하는 변수이고, 그래서 이 변수를 주소를 저장하는 변수, 포인터 변수라고 부른다.

```
int i = 125;
```

위의 코드에서 i라는 이름의 변수는 125 값을 가진다. 그럼으로 표현하면 다음과 같다.

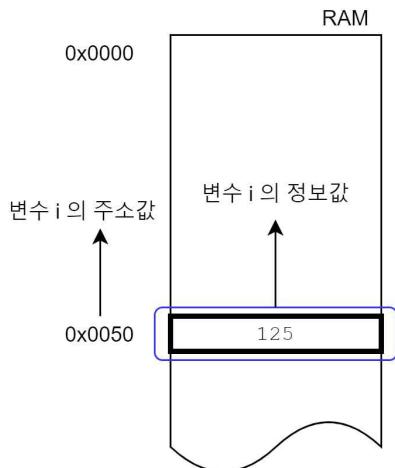


그림 149 변수와 포인터

지금까지는 변수 *i*에 저장된 값을 읽거나 혹은 변수 *i*에 어떤 값을 입력하는 방법을 익혔다. 이번 포인터에서는 변수 *i*의 주소값을 이용하는 방법을 익

하게 된다.

변수의 주소를 읽을 때 사용하는 주소 연산자는 &이고, &i 처럼 변수의 앞에 붙여서 사용한다. 위의 그림을 참고하면 &i 는 i변수의 주소인 16진수 0x50 을 반환한다.

변수의 주소를 저장할 때 사용되는 변수를 포인터, 혹은 포인터 변수라고 부른다. int 가 정수형 변수의 앞에 붙는 형태라면 int * 는 정수형 포인터 변수의 앞에 붙는 형태이다. 즉, 정수형 변수는 정수를 담는 변수이고, 정수형 포인터 변수는 포인터주소를 담는 변수다. 두 변수는 어떤 것의 값과 주소라는 다른 값을 담는다.

```
int i = 125;  
int *pi = &i;
```

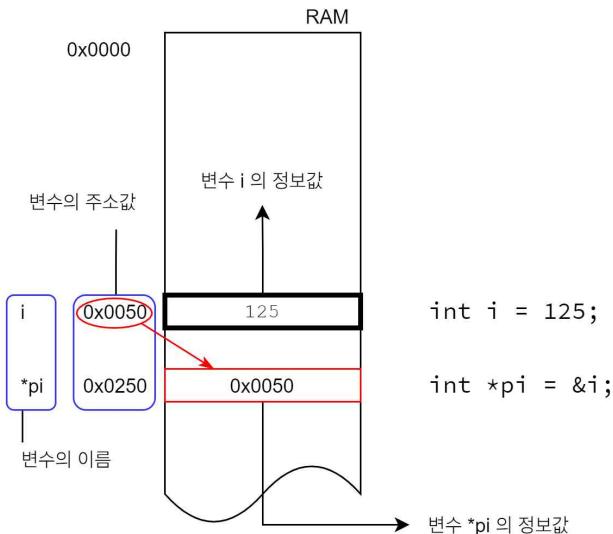


그림 150 변수와 주소, 포인터

포인터 변수를 사용해서 변수 i의 주소를 다음과 같이 가져올 수 있다. 변수 이름 앞에 * 을 붙이면 그 변수는 주소를 가리키는 포인터(변수)가 된다. 변수는 정수, 실수, 문자등 다양한 크기를 가질 수 있지만 포인터는 주소를 담기 때문에 동일한 크기가 된다. 실수를 가리키는 포인터변수나 정수를 가리키는 포인터변수나 문자를 가리키는 포인터변수는 모두 동일한 크기가 가진다. 아두이노 우노의 RAM 크기는 2k Bytes(2048 bytes)이고 아두이노 메가2560의 RAM 크기는 8kByte 이다. 주소변지를 저장하기 위해 2 Bytes 만큼 할당하면 2의 16승인 64k Bytes 만큼의 주소를 구분할 수 있게 된다. 그래서 다음과 같이 포인터 변수의 크기를 확인할 수 있다.

```
...
int *pi;
long *lpi;
Serial.print("sizeof(pi, lpi)=");
Serial.print(sizeof(pi));
Serial.print(", ");
Serial.print(sizeof(lpi));
...
```

결과

`sizeof(pi, lpi)=2, 2`

A. 주소연산자 &

주소연산자는 변수의 이름 앞에 붙어 해당 변수의 주소값을 반환하는 연산

자로 & 기호를 사용한다. 위의 그림에서 포인터변수 *pi는 0x0250이라는 주소값을 가지고, 그 주소값에는 0x0050이라는 실제 정보(다른 변수의 주소값)를 담고 있다. 이것을 한번 더 포인터로 만들어보면 *pi의 주소값을 담는 포인터 변수를 다시 만들수 있고, 이를 포인터의 포인터라고 부른다. 포인터의 포인터는 이 책에서 다루지 않을 것이다. 다만 포인터는 다른 변수들처럼 1)주소와 실제 값을 가지고 있고, 2)포인터 변수가 가지는 실제 값은 다른 변수나 배열등이 가지는 실제 값이 담겨 있는 주소값임을 기억하기 바란다. 포인터를 너무 어렵거나 특별하게 대하지 말고 변수 중 하나로 대하되, 담기는 내용은 메모리의 주소정보가 된다. 정수형 변수가 정수를 담고, 실수형 변수가 실수를 담듯이 포인터 변수는 기억장치의 주소를 담을 수 있는 변수이다.

변수 i에 125를 저장했다면 125는 기억장치인 RAM의 어딘가에 저장되었다는 뜻이다. 이때 그 위치를 알고 싶다면 주소연산자를 사용하여 &i의 값을 활용하면 된다.

PC의 경우 printf() 함수를 사용하면서 주소값을 출력하는 옵션인 %p를 사용할 수 있다. 다음은 온라인 C 컴파일러인 <https://repl.it/>에서 테스트한 결과이다.

```
#include <stdio.h>

int main()
{
    int i = 125;
    int *pi;
    pi = &i;

    printf("%p %d\n", &i, i);
    printf("%p %d\n", pi, *pi);
```

```
    return 0;  
}
```

출력은 다음과 같이 나온다.

```
0x7fff2c34d6c8 125
```

```
0x7fff2c34d6c8 125
```

두 줄 모두 동일하게 출력이 되었다는 것을 확인하라. 동일한 코드를 아두이노에서 작성하면 다음과 같다. 단 아두이노는 printf() 함수가 없고, %p라는 옵션을 붙일 수 없다. 그래서 Serial.println() 함수를 사용하고 포인터 변수를 int 를 이용해 정수로 형변환하여 출력한다.

아두이노 코드는 다음과 같다.

```
void setup()  
{  
    Serial.begin(9600);  
  
    int i = 125;  
    int *pi;  
    pi = &i;  
  
    Serial.print("i=");  
    Serial.print(i);  
    Serial.print(", &i=");  
    Serial.print(int(&i));  
  
    Serial.print(", pi=");  
    Serial.print(int(pi));
```

```
Serial.print(", *pi=");
Serial.print(int(*pi));
}

void loop() { }
```

실행결과

i=125, &i=2294, pi=2294, *pi=125

B. 간접 참조연산자 *

간접 참조연산자는 포인터 변수의 이름이나 주소값 앞에 붙여서 사용하는 연산자로 * 기호를 사용한다. * 기호는 a * b 처럼 두개의 피연산자 사이에 있으면 곱셈을 의미하는 수학연산자로 사용되고, *a 처럼 하나의 피연산자 앞에 있으면 참조연산자로 사용된다.

다음 코드를 실행하면 변수 a에 저장된 값이 출력된다.

```
int a = 8;
int *p;
p = &a;
Serial.println(*p);
// 실행결과 8
```

간접 참조연산자를 이용하여 데이터를 변경할 수 있다.

```
int a = 8;  
int *p;  
p = &a;  
*p = 12;  
Serial.println(*p);  
// 실행결과 12
```

C. 포인터 사용시 주의사항

포인터는 C가 가지고 있는 매우 강력한 기능이지만 잘못 사용하면 큰 오류가 발생할 수 있다. 포인터 사용시 주의사항은 다음과 같다.

a. 포인터의 초기화

초기화되지 않은 포인터는 메모리에 잘못된 접근을 할 수 있고 이는 시스템 전체에 심각한 손상을 줄수 있다. 포인터 초기화를 반드시하고 초기화되지 않은 포인터는 사용하지 않도록 한다.

```
int *a; // 초기화 전  
*a = 100;
```

보다는

```
int *a = NULL; // 초기화  
*a = 100;
```

로 사용하도록 한다.

b. 자료형과 포인터 타입의 불일치

포인터 선언 시 지정 자료형과 일반 변수의 자료형이 일치되어야한다. 일치하지 않으면 오류가 발생한다. 포인터로 참조되는 데이터의 크기와 자료형을 결정하는 것은 포인터의 타입이다.

```
void setup()
{
    Serial.begin(9600);

    double a = 100.1;
    int *pa = NULL;

    Serial.println((int)pa);
    Serial.println(*pa);

    pa = &a;      // 자료형 불일치로 인한 에러 발생
    Serial.println((int)pa);
    Serial.println(*pa);
}

void loop() { }
```

c. 절대주소값 참조 자제

포인터를 선언할 때 절대 주소값은 시스템이 관리하게 된다. 이를 강제적으로 접근하기 위한 절대주소값을 참조하는 것은 특별한 경우가 아니고서는

허용하지 않는 것이 바람직하다.

11.2. 포인터 연산

포인터는 주소변지를 담는 변수이므로 일반적인 사칙연산을 수행하는 것은 허용되지 않는다. 포인터의 연산은 주소를 활용하기 위한 연산으로 증가와 감소만 있으며 주소까리의 연산이 아닌 특정 위치를 기준으로 앞과 뒤로 이동하기 위한 연산이 된다.

포인터 연산자의 종류

유형	종류	활용
산술연산자	$+, -, ++, --$	$*p+1, *p++$
주소 참조 연산자	$\&, *$	$\text{int } *p; p=\&a$
대입 연산자	$=, +=, -=$	$p=\&a, a += (*p)++$
비교 연산자	$==, !=$	$p == \text{NULL}$

그림 151 포인터 연산자

A. 포인터에 정수 더하기

포인터 p 에 정수 j 를 더하면 p 가 현재 가리키는 원소에서부터 j 만큼 떨어진 원소를 가리킨다. 즉, p 가 $a[i]$ 를 가리키게 되면 $p + j$ 는 $a[i + j]$ 를 가리킨다.

아래 예제는 다음 선언을 가정한다.

```

int a[10];
int* p;
int* q;

p = &a[4];

```

배열 a 의 4번 인덱스의 주소를 포인터 변수 p 에 넣는다. 포인터 p는 a[4] 주소를 가리킨다.

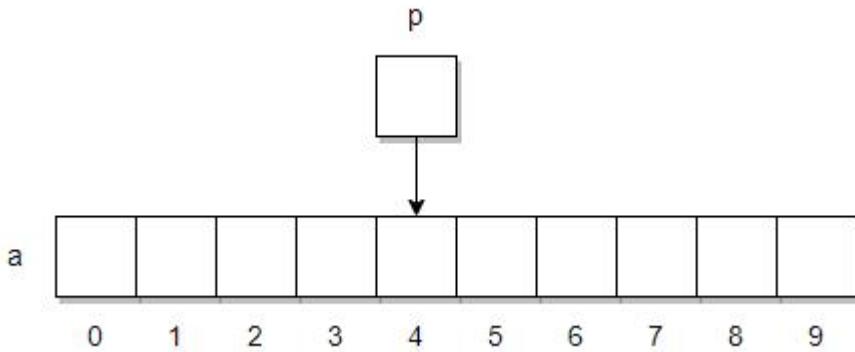


그림 152 p = &a[4]

q = p + 2;

p에 담긴 주소의 타입이 int 이다. p에서 int 크기만큼 두번 오른쪽으로 간 주소를 q 포인터 변수에 담는다. q는 a[6] 을 가리킨다.

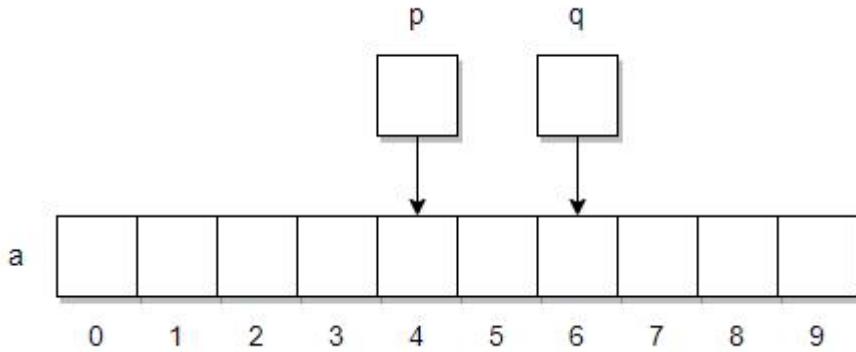


그림 153 $q = p + 2$

$p += 4;$

포인터 p 를 오른쪽으로 4칸 이동시킨다. p 는 $a[8]$ 을 가리키게 된다.

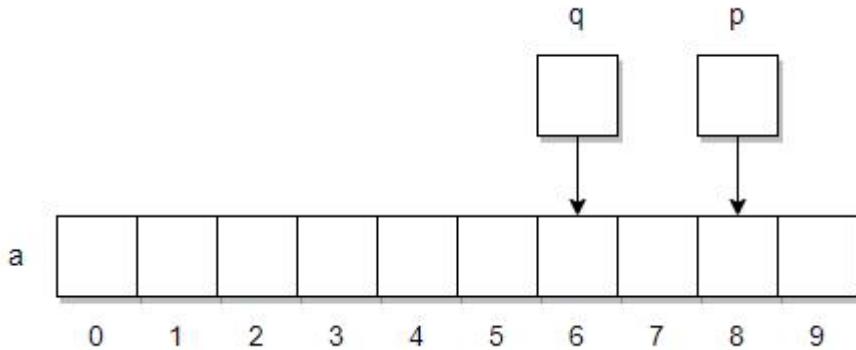
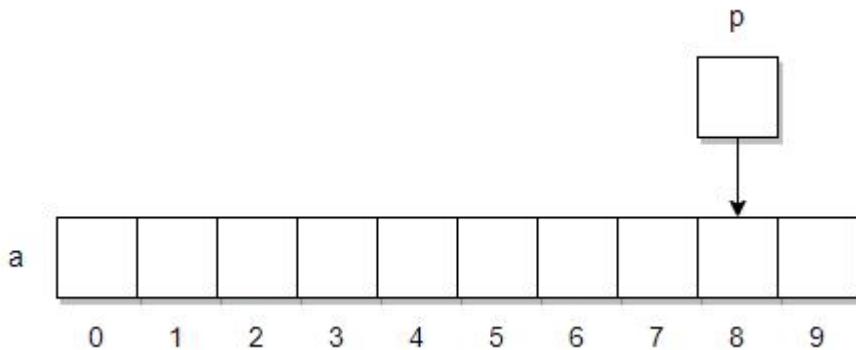


그림 154 $p += 4$

B. 포인터에 정수 빼기

A와 동일하게 포인터를 이동시킨다. 빼기를 이용해 왼쪽으로 이동시킨다.

```
p = &a[8];
```



```
그림 155 p = &a[8];
```

```
q = p - 2;
```

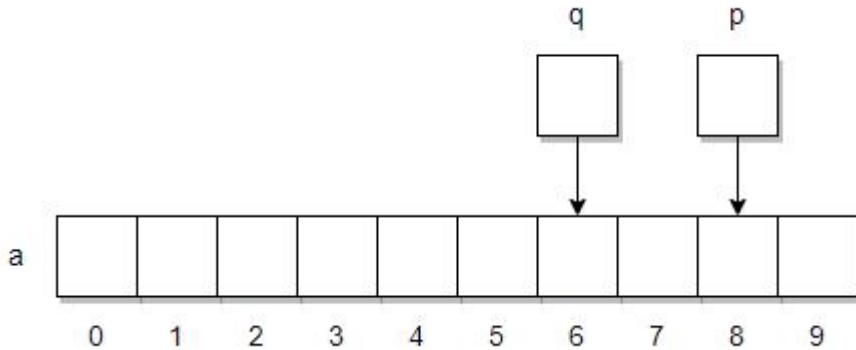


그림 156 $q = p - 2;$

$p -= 5;$

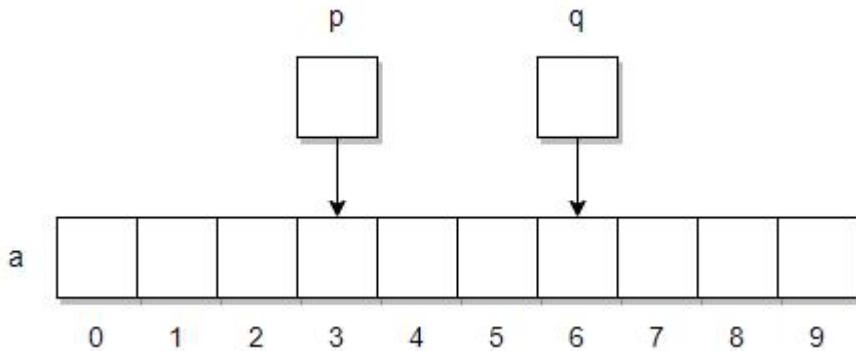


그림 157 $p -= 5;$

11.3. 함수의 포인터

함수포인터는 함수를 가리키는 포인터이다. 함수를 정의하고 실행하면 메모리에 코드 형태로 존재하게 된다. 이때 메모리 공간의 차지하고 주소를 가지게 된다. 이 주소를 포인터에 넣어서 활용하는 것이 함수의 포인터이다. 함수 포인터는 함수가 저장된 공간의 시작 주소를 가리킨다.

함수 포인터의 타입은 함수의 반환값과 매개변수에 의해 결정된다. 해당 함수에 맞는 함수 포인터를 만들기 위해 함수 원형을 알아야한다.

함수 원형은 다음과 같다.

```
void Func(int, int);
```

이에 대한 함수의 포인터는 다음과 같다.

```
void (*ptr_func)(int, int);
```

함수 포인터 사용 형식을 예를 들어 살펴보면 다음과 같다. 매개변수 2개를 정수형으로 반환하는 hap() 함수를 호출하려면 다음과 같이 선언한다.

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

```
void loop()
{
    int result;
    int (*fp)(int, int);
    ..
    fp = hap;
    result = fp(100,200);
    ..
}
```


Chapter 12

주요 학습 목표

- 구조체의 기본 개념 이해
- typedef 구조체 선언

Chapter 12. 구조체

이 장에서는 구조체, 공용체, 열거형을 익힌다. 구조체는 서로 다른 형을 가지는 집합이다. 공용체와 구조체는 유사하고, 각 요소가 같은 저장 공간을 공유한다. 공용체는 한 번에 동시에 모든 구성원을 저장할 수 없고, 한 요소만 저장한다. 열거형은 새로운 타입을 선언하면서 동시에 그 타입이 가질 수 있는 정수형 상수값도 같이 명시한다.

이중 구조체의 사용빈도가 가장 높다. 이 장의 대부분을 구조체에 할애하고 있다. 구조체 변수를 선언하고 간단한 연산을 수행하는 법, 구조체형을 정의하는 법, 구조체 입력변수를 받거나 구조체를 반환하는 함수를 작성하는 법을 배운다.

12.1. 구조체 기본 개념

구조체(Structure)는 C 언어에서 데이터를 묶어 하나의 새로운 데이터 형태로 정의하는 방법 중 하나로 서로 다른 데이터 타입을 가진 여러 변수를 하나의 단위로 묶어서 관리하고 처리할 수 있게 해준다.

구조체는 다음과 같이 정의된다.

```

struct name {
    type1 name1;
    type2 name2;
    // 추가 멤버들...
};

```

`struct`는 구조체 정의 키워드이고 `name1`, `name2` 는 사용자가 지정한 구조체의 이름이다. 이어지는 중괄호 `{}` 내부에는 멤버 변수들이 정의된다. 각 멤버 변수는 데이터 타입과 이름을 가지며, 멤버 변수들은 쉼표로 구분된다.

예를 들어, 정수로 이루어진 `x` 좌표와 `y` 좌표로 점의 위치를 나타내면 다음과 같다.

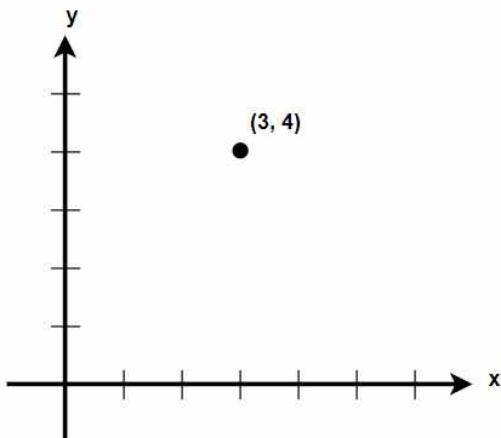


그림 158 공간위 점의 위치 (x,y)

점의 구성요소인 두 원소는 다음과 같이 선언된 구조체에 들어 갈 수 있다.

```
struct point {  
    int x;  
    int y;  
}
```

struct 명령어로 구조체를 시작한다. 구조체 이름은 struct라는 명령이 뒤에 온다. 구조체의 이름을 structure tag라고도 하는데, 복잡한 선언 대신 사용된다. 구조체에 속하는 변수는 멤버(member)로 부른다. 구조체 멤버나 이름이 아닌 보통의 변수들은 의미상 구별이 가능하기 때문에 멤버와 일반 변수의 이름이 동일해도 상관없다. 동일한 이름의 멤버를 다른 구조체에서도 사용할 수 있다. struct 선언은 형을 정의하는 것으로 멤버의 끝을 나타내는 우측 중괄호 뒤에 콤마로 분리하여 변수를 사용하면 여러 변수를 동시에 정의할 수 있다.

```
struct { ... } x, y, z;
```

이 struct 구문은 x, y, z를 특정한 형의 변수로 선언하고 각각에 기억장소를 배정

한다는 점에서 기존의 변수 선언과 유사하다.

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

변수열이 따르지 않는 구조체 선언은 구조체의 형만을 선언한다. 그러나 만약 구조체에 이름 tag가 붙어 있다면 그 이름은 나중에 실제 구조체의 선언에 사용될 수 있다. 위의 point 선언의 경우는,

```
struct point pt;
```

변수 pt를 point 형의 구조체로 선언하는 것을 뜻한다. 구조체의 경우 다음과 같이 각 요소에 대한 초기값을 열거, 정의함으로써 초기화할 수 있다.

```
struct point maxpt = { 320, 200 };
```

구조체형 지동변수는 구조체의 선언과 같은 형의 값을 리턴하는 함수를 부르거나 지정에 의해 초기화할 수 있다.

구조체의 멤버는 다음과 같이 사용된다.

구조체이름.멤버

구조체 멤버 연산자 “.”로 구조체의 이름과 멤버 이름을 연결시킨다. 점 pt의 좌표 값을 출력하기 위해 다음과 같이 쓸 수 있다.

```
printf("%d,%d", pt.x, pt.y);
```

점 pt 원점에서의 거리를 구하기 위해 다음과 같이 사용할 수 있다.

```
double dist, sqrt(double) ;
dist = sqrt ((double)pt.x * pt.x + (double)pt.y * pt.y) ;
```

구조체의 구조체도 가능하다. 사각형은 대각선에 위치한 본 두 점을 지정해서 표현할 수 있다.

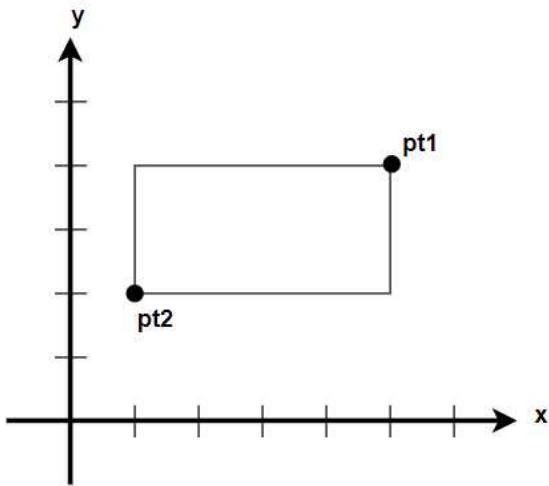


그림 159 평면위 사각형의 위치, pt1, pt2

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

rect 구조체는 두 개의 point 구조체를 요소로 가진다. 만일 screen이 다음과 같이 선언되어 있다면

```
struct rect screen;
```

screen의 pt1이라는 멤버의 x 좌표는 다음과 같이 나타낼 수 있다.

```
screen.pt1.x
```

각 구조체는 새로운 스코프를 갖는다. 해당 스코프 안에 선언한 이름은 프

로그램의 다른 이름과 충돌을 일으키지 않는다. 각 구조체는 구성원들에 대해 서로 다른 이름 공간 name space를 갖는다. 예를 들어 다음은 같은 코드안에 존재할 수 있다.

```
struct {
    int number;
    char name[NAME_LEN + 1];
    int on_hand;
} part1, part2;

struct {
    char name[NAME_LEN + 1];
    int number;
    char sex;
} employee1, employee2;
```

앞에서 설명한 것처럼 part1과 part2 구조체들의 number와 name 구성원들은 employee1과 employee2의 number와 name 구성원과 충돌하지 않는다.

12.2. 구조체 변수 초기화

구조체 변수를 선언시에 초기화하려면 구조체에 저장할 값의 목록을 준비하고 이를 괄호로 감싸주면 된다.

```
struct {
    int number;
    char name[NAME_LEN + 1];
```

```
    int on_hand;
} part1 = {510, "Arduino Uno", 10},
part2 = {620, "ESP32", 5};
```

초기값들은 반드시 구조체의 멤버순서와 동일순서로 주어져야 한다. part1의 number 구성원은 510이 되고, name 구성원은 "Arduino Uno"가 될 것이다. 초기화 이후 part1은 다음과 같다.

```
number : 510
name : Arduino Uno
on_hand : 10
```

구조체 초기자는 배열 초기자와 유사한 규칙을 따른다. 구조체 초기자 내부에 사용된 표현식은 상수여야 한다. part1의 on_hand 구성원 초기화에 변수는 사용될 수 없다(단, 이 제한은 C99에서 완화되었다). 초기자는 초기화할 구조체의 구성원 수보다 적은 구성원을 가질 수 있다. 배열처럼 나머지 구성원들은 초기값으로 0이 주어진다. 특히 문자형 배열의 나머지 바이트들은 비어있는 문자열이 된다.

12.3. 구조체 연산

배열 연산은 첨자로, 원소를 위치 기반으로 선택한다. 같은 방식으로 구조체의 일반적인 연산은 그 멤버 중 하나를 선택하는 것으로 보아도 무방하다. 구조체 구성원은 위치가 아닌 이름으로 접근한다.

구조체 내의 구성원에 접근하려면 우선 구조체의 이름을 작성하고, 점을 찍은 뒤 구성원의 이름을 작성한다. 예를 들어 다음 구문은 part1의 구성원의 값을 출력할 것이다.

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

구조체의 구성원은 왼쪽 값으로 좌측에 위치하거나 증감식의 피연산자로 사용할 수 있다.

```
part1.number = 5; /* part1의 부품 번호를 바꾼다*/
part1.on_hand++; /* part1의 현재 보유 수를 하나 증가한다*/
```

구조체 구성원에 접근할 때 사용하는 점은 C 언어의 연산자다. ++, -- 연산자와 같은 연산 순서를 가지므로 대부분의 연산자보다 우선 순위가 높다.

```
scanf("%d", &part1.on_hand);
```

표현식 &part1.on_hand는 두 개의 연산자(&, .)를 갖는다. "." 연산자는 "&" 연산자보다 우선순위가 높으므로 &은 의도대로 part1.on_hand의 주소를 반환한다.

다른 주요 구조체 연산에는 할당이 있다:

```
part2 = part1;
```

i) 구문은 part1.number을 part2.number에 복사하고, part1.name을

part2.name에 복사하는 식의 효과를 본다.

배열은 = 연산자로 복사될 수 없지만 구조체는 가능하다. 구조체 안에 존재하는 배열 또한 해당 구조체가 복사될 때 같이 복사된다. 프로그래머들은 이를 이용해 이후 배열을 복사하기 위해 배열을 포함하는 일종의 "임시" 구조체를 만들어 사용한다.

```
struct { int a[10]; } a1, a2;  
a1 = a2; /* a1과 a2는 구조체이므로 = 연산자 사용이 가능 */
```

= 연산자는 오로지 호환 가능한 구조체 간에만 사용할 수 있다. 동시에 선언된 두 구조체는 서로 호환 가능하다. 같은 구조체 태그를 가지고 선언된 구조체 혹은 같은 형명을 갖는 구조체는 호환된다.

구조체를 이용하여 학생과 학점을 함께 다루는 예제

```
struct student // 학생 구조체 선언  
{  
    int id; // 학번  
    char name[30]; // 이름  
    double grade; // 학점  
};  
...  
void setup(void)  
{  
    ...  
    struct student s1 = { 21, "강산", 2.4 }, // 구조체 변수  
    선언과 초기화
```

```

s2 = { 22, "유미", 3.7 },
s3 = { 23, "하늘", 4.4 };

struct student max; // 최고 학점을 저장할 구조체 변수
max = s1; // s1을 최고 학점으로 가정
if(s2.grade>max.grade) max = s2; // s2가 높으면 max에
if(s3.grade>max.grade) max = s3; // s3가 높으면 max에 ...
}

```

12.4. **typedef** 구조체 선언

typedef를 사용하여 구조체를 선언할 수 있다. 다음과 같이 방법으로 Part라는 이름의 형을 정의한다.

```

typedef struct {
    int number;
    char name[20];
    double grade;
} Part;

```

주의해야 할 점은 Part가 struct 다음이 아니라 { } 중괄호의 끝에 온다는 점이다.

이제 Part를 다른 기본형처럼 사용하면 된다. 다음과 같이 변수를 선언한다.

```

Part part1;
Part part2;

```

Part는 **typedef** 이름이므로 struct Part로 쓰면 안 된다. 그리고 Part형 변수

는 어디서 선언되었든 각자 호환된다.

구조체명을 정해야할 때 보통 구조체 태그나 typedef 를 중 하나를 사용한다.

12.5. 공용체

공용체(Union)는 다양한 데이터 형식을 하나의 메모리 공간에서 공유하도록 허용하는 데이터 구조이다. 공용체는 구조체(Structure)와 유사하게 멤버 변수들을 포함하며, 각 멤버 변수는 하나의 메모리 영역을 공유한다.

공용체는 주로 다양한 데이터 형식의 데이터를 저장하고 필요한 데이터 형식을 동적으로 선택할 때 유용하다. 공용체를 사용하여 정수, 실수, 문자열 등 다양한 데이터 형식을 하나의 메모리 영역에서 처리할 수 있다. 이렇게 메모리 사용을 최적화하고 다양한 데이터 형식을 처리할 수 있게 된다.

다음은 C 언어에서 공용체를 정의하고 사용하는 간단한 예제이다.

```
union SampleUnion {
    int intValue;
    float floatValue;
    char stringValue[20];
};

...
void setup() {
    ...
    union SampleUnion data;
```

```
data.intValue = 42;
Serial.println(data.intValue); // 정수값

data.floatValue = 3.14;
Serial.println(data.floatValue); // 실수값

...
}
```

공용체 SampleUnion을 사용하여 서로 다른 데이터 형식의 값을 저장하고 출력한다. 주의할 점은 공용체의 멤버 중 하나를 수정하면 나머지 멤버의 값도 변경된다는 것이다.

12.6. 열거형

열거형(Enumeration)은 상수 값들의 집합을 정의하는 데이터 형식이다. 열거형은 주로 프로그램에서 가독성을 높이고 코드를 이해하기 쉽게 만들기 위해 사용된다.

열거형을 정의할 때, 각 상수는 고유한 이름(기호)을 가지며, 기호에 대응하는 정수 값을 가진다. 이러한 상수는 일반적으로 연속적으로 증가하는 정수 값을 가지며, 첫 번째 상수는 0부터 시작하고 1씩 증가한다. 그러나 초기 값을 설정하여 상수의 값을 지정할 수도 있다.

다음은 간단한 열거형을 정의하는 예제이다.

```

...
enum Days {
    SUN,      // 0
    MON,      // 1
    TUE,      // 2
    WED,      // 3
    THU,      // 4
    FRI,      // 5
    SAT       // 6
};

...
void setup()
{
    ...
    enum Days today = WED;
    Serial.print("Today is ");
    Serial.println(today );
    ...
}

```

예제에서는 "Days"라는 열거형을 정의하고, "SUN", "MON", "TUE" 등의 상수를 포함시켜 각 요일을 나타냈습니다. "today" 변수에 "WED" 상수를 할당하고, "today" 변수를 출력하여 현재 요일을 표시한다.

열거형은 주로 상태, 옵션 또는 선택지와 관련된 정수값을 표현하는 데 사용된다. 이를 통해 코드를 읽기 쉽게 만들고 오류 방지에 도움이 된다.

Chapter 13

주요 학습 목표

- 파일 입출력에 대한 기본 이해
- 파일 열기, 닫기, 읽기, 쓰기

Chapter 13. 파일입출력

일반적으로 아두이노와 같은 임베디드 시스템은 OS를 가지고 있지 않기 때문에 파일입출력을 위해서는 별도의 하드웨어 장치와 관련 드라이버가 필요하다. 여기서는 PC를 기반으로 간단한 파일 입출력 예제를 들기로 한다.

파일 입출력은 C 프로그래밍에서 중요한 부분 중 하나이다. 파일 입출력은 데이터를 파일로 저장하고 파일에서 데이터를 읽어오는 프로세스를 관리하며, 데이터를 영구적으로 보존하고 공유하는 데 유용한다. 여기서는 C 언어에서 파일을 다루는 기본 개념과 함수에 대해 설명한다.

13.1. 기본 개념

C 언어에서 파일 입출력은 다음 기본 개념으로 이루어집니다:

A. 파일 포인터 (File Pointer): 파일을 읽고 쓸 때, 파일의 현재 위치를 추적하기 위한 파일 포인터가 사용됩니다. 파일 포인터는 파일의 처음, 끝, 또는 중간 위치로 이동할 수 있습니다.

B. 파일 열기와 닫기: 파일을 사용하기 전에 파일을 열어야 하며 사용이 끝나면 파일을 닫아야 합니다. 파일을 열 때, 해당 파일의 경로와 열기 모드(읽기, 쓰기, 추가 등)를 지정해야 합니다.

C. 파일 읽기와 쓰기: 파일을 열었으면, 파일에서 데이터를 읽거나 데이터를 파일에 쓸 수 있습니다. 파일 읽기와 쓰기는 파일 포인터를 조절하여 수행됩니다.

13.2. 파일 열기 및 닫기

파일을 열기 위해서는 `fopen()` 함수를 사용합니다. `fopen()` 함수는 파일 포인터를 반환하며, 파일을 열 때 사용할 파일 경로와 열기 모드를 지정해야 합니다. 예를 들어:

```
FILE *file;
file = fopen("example.txt", "w");
// "w"는 쓰기 모드로 파일을 열 때 사용
if (file == NULL) {
    perror("파일 열기 실패");
    return 1;
}
```

파일을 사용한 후에는 `fclose()` 함수를 사용하여 파일을 닫아야 합니다. 파일을 닫지 않으면 데이터 손실이 발생할 수 있습니다.

```
fclose(file);
```

13.3. 파일 읽기와 쓰기

파일에서 데이터를 읽을 때는 fread() 함수를 사용하고, 파일에 데이터를 쓸 때는 fwrite() 함수를 사용합니다. 이 함수들은 파일 포인터와 데이터 크기, 읽거나 쓸 데이터의 양을 인자로 받습니다.

```
// 파일에서 데이터 읽기  
fread(buffer, sizeof(char), 100, file);  
  
// 파일에 데이터 쓰기  
fwrite(data, sizeof(int), 5, file);
```

13.4. 파일 입출력 예제

다음은 파일에서 텍스트를 읽고 파일에 텍스트를 쓰는 간단한 예제입니다:

```
#include <stdio.h>  
  
int main() {  
    FILE *file;  
    char text[100];  
  
    // 파일 열기  
    file = fopen("example.txt", "w");  
    if (file == NULL) {  
        perror("파일 열기 실패");
```

```

    return 1;
}

// 파일에 텍스트 쓰기
fprintf(file, "안녕하세요, 파일 입출력!");

// 파일 닫기
fclose(file);

// 파일 열기
file = fopen("example.txt", "r");
if (file == NULL) {
    perror("파일 열기 실패");
    return 1;
}

// 파일에서 텍스트 읽기
fgets(text, sizeof(text), file);
printf("파일 내용: %s\n", text);

// 파일 닫기
fclose(file);

return 0;
}

```

이 예제에서는 파일을 열고 텍스트를 쓰고 읽는 과정을 보여줍니다. 파일 입출력을 통해 데이터를 파일에 저장하고 읽어올 수 있습니다.

Chapter 14

주요 학습 목표

- 인공지능을 이용한 코딩
- ChatGPT 활용 방법

Chapter 14. 마무리

최근 인공지능의 열풍이 거세지고 있다. 이런 환경은 개발자들에게 도전을 주고 있다. 기존의 정보는 책에서 인터넷 검색으로 넘어왔고, 이제는 인공지능을 이용하는 AI Literacy가 큰 흐름이 되고 있다. 즉, 인공 지능 (Artificial Intelligence, AI)과 관련된 기술, 개념 및 응용 분야를 이해하고 활용하는 능력은 우리의 일상 생활과 사회 전반에 더욱 중요한 역할을 하고 있으며 특히 개발자들에게는 필수적인 스킬 중 하나가 되었다.

임베디드 시스템을 개발하는 입장에서 C 언어를 사용하는 경우, ChatGPT를 효과적으로 활용하기 위한 몇 가지 방법이 있다.

첫째, C 언어와 관련된 질문이나 문제에 직면했을 때 ChatGPT에게 도움을 요청할 수 있다. 예를 들어, 특정 C 언어 함수의 사용 방법이나 프로그램 디버깅과 관련된 질문을 물어보는 것이 가능하다.

둘째, C 코드 작성에 도움이 필요한 경우 ChatGPT에게 원하는 작업을 수행하는 간단한 C 코드 예제를 요청할 수 있다. 이를 통해 코드 작성의 기초를 이해하고 빠르게 시작할 수 있다.

셋째, 임베디드 시스템 설계나 알고리즘 구현과 관련하여 ChatGPT로부터 도움을 받을 수 있다. 여러 가지 알고리즘 및 설계 원칙에 대한 정보와 조

언을 얻을 수 있다.

넷째, ChatGPT를 활용하여 C 코드의 디버깅과 최적화에 관한 질문을 하여 프로젝트의 성능을 향상시키고 코드 버그를 해결할 수 있다.

이러한 방법을 통해 ChatGPT를 활용하여 C 언어와 임베디드 시스템 개발 관련 문제의 도움을 얻고, 프로젝트를 원활하게 진행할 수 있다.

14.1. 인공지능과 개발자

인공지능(AI)은 현대 기술의 중심에 서 있으며, 그 발전은 디지털 시대를 혁신적으로 변화시키고 있다. 이 기술은 컴퓨터 시스템에 인간과 유사한 지능을 부여하거나, 데이터로부터 패턴을 학습하여 다양한 작업을 자동화하는 데 사용된다. AI는 머신 러닝과 딥 러닝과 같은 하위 분야를 포함하며, 이미 우리의 일상 생활에서 널리 사용되고 있다.

AI의 역할과 영향

AI는 다양한 분야에서 혁신과 변화를 가져오고 있다. 의료 분야에서는 AI를 사용하여 질병 진단 및 치료 계획을 개선하고, 자율 주행 자동차 분야에서는 운전 보조 시스템을 개발하며, 금융 분야에서는 거래 감시와 위험 분석에 활용된다. 또한 AI는 언어 번역, 음성 인식, 이미지 분석 및 예측 모델링과 같은 다양한 작업에 사용된다.

AI 개발자의 역할

AI의 성공적인 구현에는 뛰어난 AI 개발자가 필요합니다. AI 개발자는 AI

모델을 디자인하고 훈련시키는 작업을 담당하며, 이를 통해 시스템이 원하는 작업을 수행할 수 있게 만든다. AI 개발자는 데이터 수집 및 전처리, 모델 선택 및 최적화, 그리고 윤리적 고려 사항을 다루는 등 다양한 업무를 수행한다.

AI 개발자가 가져야 할 역량에는 프로그래밍, 수학, 통계학, 머신 러닝 알고리즘 이해, 문제 해결 능력 등이 포함된다. 뿐만 아니라 AI 개발자는 윤리적 고려 사항을 이해하고 AI 시스템의 안전성을 보장하기 위한 조치를 취해야 한다.

AI를 개발하는 개발자가 있는 반면 AI를 이용하는 개발자도 있다. ChatGPT 와 같은 인공지능을 제대로 이용하면서 지나치게 휘둘리지 않아야하고, 동시에 주도적으로 개발과 프로젝트를 진행할 수 있어야한다. 잘못된 답변이나 할루시에이션 같은 허상을 믿는 대신 빠르게 바른 판단을 내릴 수 있어야하고 지나치게 AI에 얹매여서는 안된다. 이런 것들을 전제로 ChatGPT를 이용한 코딩을 도움받는 것이 AI를 이용한 임베디드 시스템 개발자가 해야 할 역할이 된다.

14.2. ChatGPT를 이용한 C코딩

코드 예제 요청:

ChatGPT를 사용하여 C 코딩을 시작하려면 먼저 코드 예제를 요청할 수 있다. 예를 들어, "C 언어를 사용하여 간단한 문자열 처리 프로그램을 작성하는 방법을 보여주세요"와 같은 질문을 할 수 있다. ChatGPT는 해당 작업에 필요한 기본 코드 예제를 생성하여 제공할 수 있으며, 이를 통해 코드 작성의 출발점을 얻을 수 있습니다.

다음은 “아두이노 우노 프로그램을 만들고 싶다. 12번 핀을 200ms 간격으

로 깜박이게 하는 코드를 작성하시오.”라는 요청을 ChatGPT에게 준 결과로 만들어진 코드이다.

```
// 12번 핀을 출력으로 설정
int ledPin = 12;

void setup() {
    pinMode(ledPin, OUTPUT);
}

void loop() {
    // 200ms (0.2초) 동안 LED를 켜
    digitalWrite(ledPin, HIGH);
    delay(200);

    // 200ms (0.2초) 동안 LED를 끄
    digitalWrite(ledPin, LOW);
    delay(200);
}
```

아두이노 LED를 켜는 간단한 코드는 무리없이 예제를 생성해 낸다. 스위치 입력에 대한 코드 생성 요청도 이와 비슷하다. 다음과 같은 코드를 ChatGPT가 생성한다. 문제없이 잘 작동한다.

```
// 11번 핀을 입력으로 설정하고 내부 풀업 저항 사용
int switchPin = 11;
int ledPin = 13;

void setup() {
    pinMode(switchPin, INPUT_PULLUP); // 11번 핀을 입력으로 설정하고 풀업 저항 사용
```

```

pinMode(ledPin, OUTPUT);           // 13번 핀을 출력으로 설정
}

void loop() {
    // 스위치가 눌렸을 때 (LOW 상태) LED를 켜
    if (digitalRead(switchPin) == LOW) {
        digitalWrite(ledPin, HIGH); // LED를 켜
    } else {
        digitalWrite(ledPin, LOW); // LED를 끄
    }
}

```

코드 디버깅과 문제 해결:

C 언어 프로그래밍 중 발생하는 버그나 오류를 해결하기 위해 ChatGPT에 도움을 청할 수 있다. 코드 조각을 제공하고 "왜 이 부분에서 오류가 발생하는 걸까요?"와 같은 질문을 하면, ChatGPT는 코드 오류의 원인을 찾는데 도움을 줄 수 있다. 디버깅 과정에서 ChatGPT의 조언을 활용하여 문제를 해결할 수 있다.

알고리즘 및 최적화 고려:

C 코딩에서는 종종 효율적인 알고리즘과 최적화 기술이 필요하다. ChatGPT에게 "C 언어로 작성한 정렬 알고리즘을 개선하는 방법은 무엇인가요?"와 같은 질문을 하면, 알고리즘 개선 또는 코드 최적화에 대한 제안을 얻을 수 있다. 마찬가지로 임베디드 시스템에 대한 최적화관련 질문을 통해 코드의 성능을 향상시키고 효율성을 개선할 수 있다.

라이브러리 및 툴 사용:

C 언어 프로젝트를 개발하는 과정에서 필요한 라이브러리 및 개발 도구에 대한 정보를 얻을 수도 있다. ChatGPT에게 "C 언어로 Arduino 마이크로컨트롤러를 제어하는 방법을 알려주세요"와 같은 질문을 하면, 특정 라이브러리나 툴 사용법에 대한 안내를 받을 수 있습니다. 이를 통해 프로젝트를 더 효과적으로 진행할 수 있습니다.

"아두이노에 초음파센서를 연결해서 사용하려고 한다. 어떤 라이브러리를 사용하는 것이 좋은가?"라고 질문을 하면 "아두이노에서 초음파 센서를 사용하기 위해 HC-SR04와 같은 초음파 센서를 연결하려면 "NewPing" 라이브러리를 사용하는 것이 좋습니다. "NewPing" 라이브러리는 초음파 센서와의 상호 작용을 단순하게 만들어주고, 거리 측정에 편리한 함수를 제공합니다...."라는 답변을 준다. 실제로 newping이라는 라이브러리는 존재하고 초음파센서를 사용하는 라이브러리이다.

관련된 예제까지 보여준다.

```
#include <NewPing.h>

#define TRIGGER_PIN 9 // 초음파 센서의 Trig 핀 (변경 가능)
#define ECHO_PIN 10    // 초음파 센서의 Echo 핀 (변경 가능)
#define MAX_DISTANCE 200 // 측정할 최대 거리 (센서의 사양에 따라 조정)

NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE);

void setup() {
    Serial.begin(9600); // 시리얼 통신 초기화
}

void loop() {
```

```
delay(100); // 측정 주기를 조절하기 위한 딜레이  
unsigned int distance = sonar.ping_cm(); // 거리 측정 (센치미터  
단위)  
Serial.print("Distance: ");  
Serial.print(distance);  
Serial.println(" cm");  
}
```

ChatGPT를 활용하여 C 코딩에 관한 질문과 도움을 얻으면, 프로그래밍 과정을 더욱 원활하게 진행하고 코드 작성 및 디버깅 과정에서 더 나은 결과를 얻을 수 있다.

남기는 말

이 책은 기준의 C언어가 PC 위주로 쓰여진 것과 현재 C언어는 임베디드 시스템에 주로 사용되고 있는 현실 사이의 괴리감 때문에 시작했습니다.

그러다보니 C언어에서 중요하게 여겨지던 것 중 임베디드 시스템에서는 잘 사용하지 않는 부분도 있고, 메모리가 풍부한 PC에서는 굳이 신경쓰지 않아도 되는 부분을 작은 메모리를 가진 아두이노 MCU에서는 신경써야하는 경우들이 있습니다.

이 책은 그런 부분을 짚고 싶어서 쓰여졌습니다. 그래서 장점도 있지만 단점도 있습니다. C가 가지고 있는 충분한 문법에 대한 설명을 하지 못하고 지나갔습니다. 특히 배열과 포인터에 대해서는 아쉬운 부분이 있습니다.

하지만 아두이노를 기준으로 한 임베디드 시스템에서 포인터가 거의 사용되지 않는 점과 현대 프로그래밍의 방향이 포인터와 같이 하드웨어를 직접적으로 제어하는 부분을 더 이상 프로그래머가 아닌 시스템과 언어에 맡기고 있습니다. 이러한 경향을 바탕으로 C언어 중 포인터에 대해서는 가능한 간략하게 넘어갔습니다. 추후 더 배움을 원하는 이들은 시중에 있는 많은 C언어 책을 공부하기를 권합니다.

임베디드 시스템은 단지 프로그래밍만 잘한다고 해결되지 않습니다. 하드웨어의 특수성을 이해해야하고 때로는 하드웨어에 따라서 프로그래밍이 달라져야 합니다.

임베디드 시스템을 공부하는 이들에게 조언하는 것은 단순한 언어의 문법적 특성을 암기하기보다는 하드웨어와 함께 특성을 만들어가는 경험을 쌓기를 바랍니다. 특히 인터럽트와 같은 부분은 PC에서는 거의 다루지 않는

부분이지만 아두이노와 같은 MCU에서는 조금만 깊이 들어가면 반드시
마주치는 분야이기도 합니다. 문법적으로도 비트연산자는 PC에서 잘 사용
되지 않지만 MCU에서는 레지스터를 다룰 때 비트연산자가 큰 역할을 합니
다.

모쪼록 이 책을 통해 작은 영감을 얻었기를 바라고 더 깊고 찬란한 세상속
으로 들어가는 길잡이의 작은 등불이 되기를 바랍니다.

<제목 차례>

Chapter 01. 임베디드시스템과 마이크로컨트롤러의 이해	1
1.1 컴퓨터의 구조	1
1.2. 마이크로프로세서와 마이크로컨트롤러	6
1.3. 마이크로프로세서 분류방법	10
1.4. 마이크로컨트롤러 동향	17
Chapter 02. 개발 환경 설정	20
2.1. 아두이노에 대한 이해	20
2.2. 아두이노 우노, 메가 2560, 멀티 평선 실드	21
2.3. 아두이노 IDE 설치	24
2.4. CH340 드라이버 설치	31
2.5. 아두이노 시뮬레이터	38
2.6. 온라인 C 컴파일러	44
Chapter 03. C와의 만남	47
3.1 프로그래밍이란 무엇인가?	48
3.2. 프로그래밍 패러다임	50
3.3. 프로그래밍 스타일	51
3.4. C 와 C++	58
3.5. 아두이노 검색하기	60
3.6 Hello World 그리고 Hello Led	63
3.7. 아두이노 코드의 특징	67
3.8. 다양한 라이브러리	70
3.9. 시리얼 모니터	71
Chapter 04. 주석, 변수, 상수	77
4.1. 아두이노 기본구조	77
4.2. 주석	78
4.3. 자료형	83
4.4. 변수	84
4.6. 정수형	100
4.7. 실수형	110
4.8. 문자형	111
4.9. 문제	115

Chapter 05. 문자열과 입출력 함수	117
5.1. 입출력장치와 시리얼 모니터	119
5.2. 문자열 처리 방법과 String	123
5.3. PC 와 표준입출력	131
5.4. MCU 시리얼통신을 이용한 입출력	138
5.5. 문자열	148
5.6. 합성, 추출, 교체	151
5.7. 기타 함수	156
Chapter 06. 연산자	163
6.1. 연산자란	163
6.2. 산술 연산자	166
6.3. 할당 연산자, 대입 연산자	177
6.4. 증감연산자	179
6.5. 관계 연산자, 비교 연산자	183
6.6. 논리 연산자	184
6.7. 윤년 판단 예제	186
6.8. 조건 연산자	187
6.9. 콤마 연산자	190
6.10. 비트 연산자	195
6.11. 연산자의 우선 순위와 결합 규칙	204
6.12. Mini Project 화씨 온도를 섭씨로 바꾸기	205
Chapter 07. 제어구조	209
7.1. 제어흐름	209
7.2. if 문	210
7.3. if-else 문	211
7.4. 다중 if 문	213
7.5. switch 문	221
7.6. while 문	229
7.7. do-while 문	240
7.8. for 반복문	242
7.9. 중첩 루프, 무한 루프	246
7.10. break, continue	249
7.11. goto	251

Chapter 08. 아두이노 실습환경	256
8.1. Multi Function Shield	257
8.2. MFS 하드웨어 설정	264
8.3. MFS 부저 알람 프로그래밍	268
8.4. MFS 라이브러리 사용방법	269
8.5. MFS 에서 LED 와 7 Segments 사용	274
8.6. MFS 에서 센서 사용	277
Chapter 09. 함수	281
9.1. 함수란	282
9.2. 함수선언	284
9.3. 함수정의	285
9.4. 함수호출	289
9.5. 매개변수 없는 함수	293
9.6. 매개변수 있는 함수	294
9.7. 반환값	298
9.8. void 함수	299
9.9. 변수와 범위	301
9.10. 재귀함수	316
9.11. 함수 이용	319
Chapter 10. 배열	329
10.1. 배열이란	329
10.2. 배열 선언과 초기화	332
10.3. 배열의 크기	335
10.4. 2차원 배열, 3차원이상 배열	335
10.5. 배열 연습	336
Chapter 11. 포인터	340
11.1. 포인터의 선언	341
11.2. 포인터 연산	349
11.3. 함수의 포인터	353
Chapter 12. 구조체	357
12.1. 구조체 기본 개념	357
12.2. 구조체 변수 초기화	362
12.3. 구조체 연산	363

12.4. typedef 구조체 선언	366
12.5. 공용체	367
12.6. 열거형	368
Chapter 13. 파일입출력	371
13.1. 기본 개념	371
13.2. 파일 열기 및 닫기	372
13.3. 파일 읽기와 쓰기	373
13.4. 파일 입출력 예제	373
Chapter 14. 마무리	376
14.1. 인공지능과 개발자	377
14.2. ChatGPT를 이용한 C코딩	378
남기는 말	383