ARDUINO

Open Physical Computing Platform

오탈자, 문의 및 보완이 필요한 내용은 hgycap@hotmail.com으로 알려주세요.

Chapter 0. 아두이노를 위한 C/C++ 언어

아두이노의 스케치는 C/C++ 언어로 작성한다. C/C++ 언어는 Java와 함께 프로그래밍 언어 중에서 가장 많이 사용되는 언어 중 하나로 Java가 인터넷의 보급에 따라 다양한 운영체제에서 동작할 수 있는 호환성을 중시한 언어라면, C/C++은 Java의 기반이 된 언어로 시스템 프로그래밍을 위해 많이 사용되는 언어이다. Windows와 Linux같은 운영체제도 C언어로 작성되었으며 아두이노와 같은 마이크로컨트롤러 제어와 관련된 프로그래밍을 위한 언어로는 C가 거의 유일한 언어이기도 하다. Java와 C/C++은 어느 언어가 더 우수하다고 말하기는 어려우며 목적에 맞게 사용하여야 한다. 참고로 아두이노의 조상이 되는 Processing 언어는 Java를 기반으로 하고 있다.

스케치를 작성하기 위해서는 C/C++ 언어를 사용할 수 있어야 하는 것은 당연하다. 일반적으로 간단한 스케치는 C 스타일로 작성하지만 아두이노에서 사용하는 라이브러리는 C++ 스타일의 클래스로 작성되어 배포되므로 클래스의 개념 또한 이해하고 있어야 아두이노 라이브러리를 활용하는데 어려움이 없다. C/C++ 언어를 모두 설명하는 것은 이 책의 범위를 벗어나는 일이므로 이 장에서는 스케치 작성을 위해 이해하고 있어야 할 C/C++ 언어의 기본적인 내용들을 요약해서 제시한다. 보다 자세한 설명은 C/C++ 언어에 대한 책을 참고하기 바란다.

1. C/C++ 언어 테스트 환경

아두이노 프로그램에는 시리얼 모니터가 포함되어 있다. 시리얼 모니터는 아두이노와 컴퓨터 사이에 직렬 통신을 통해 데이터를 주고받을 수 있도록 해주므로 아두이노에서의 실행 결과를 모니터 하기 위한 용도로 사용할 수 있다. 이는 시리얼 모니터가 일반적으로 C/C++ 언어를 학습하는 과정에서 출력을 확인하기 위한 콘솔(console)과 같은 용도로 활용할 수 있음을 의미한다. 따라서 이 장에서 C/C++ 언어 설명을 위해 모든 출력을 시리얼 모니터로 전송하여 확인하도록 한다. 시리얼 모니터는 아두이노의 기본 클래스 중 하나로 콘솔 용도로 사용하기 위해 이 장에서는 줄바꿈이 없는 출력 함수인 Serial.print()와 줄바꿈이 있는 Serial.println()을 사용한다. Serial.print()와 Serial.println() 함수는 문자열 및 숫자를 시리얼 모니터에 출력할 수 있다. 출력 문자열 포맷을 위해서는 String 클래스를 주로 사용한다. Serial 및 String 클래스에 대한 자세한 내용은 해당 장을 참고하기 바란다. 시리얼 모니터를 콘솔로 활용하여 출력을 확인하는 과정은 다음과 같다.

① 먼저 코드 1을 아두이노 프로그램의 편집기에 입력한다. 코드 1에서 붉은색으로 표시된 부분은 이 장의 대부분의 예에서 공통된 내용으로 loop 함수 내의 코드들을 주의해서 살펴보기 바란다.

코드 1

```
void setup()                // 초기화 함수
{
  Serial.begin(9600);       // 직렬 포트 초기화
}

void loop()                 // 반복 실행 함수
{
  Serial.print("Without 'New Line' character... ");   // 줄바꿈 없는 출력
  Serial.println("With 'New Line' character");        // 줄바꿈 있는 출력
  Serial.println("See the difference...");

  while(true);
}
```

② '스케치 → 확인/컴파일' 메뉴 또는 'Ctrl + R' 키로 입력한 코드의 오류를 확인한다.

③ '파일 → 업로드' 메뉴 또는 'Ctrl + U' 키로 실행 파일을 아두이노로 다운로드 한다.

④ '도구 → 시리얼 모니터' 메뉴 또는 'Ctrl + Shift + M' 키로 시리얼 모니터 프로그램을 실행시켜서 실행 결과를 확인한다.

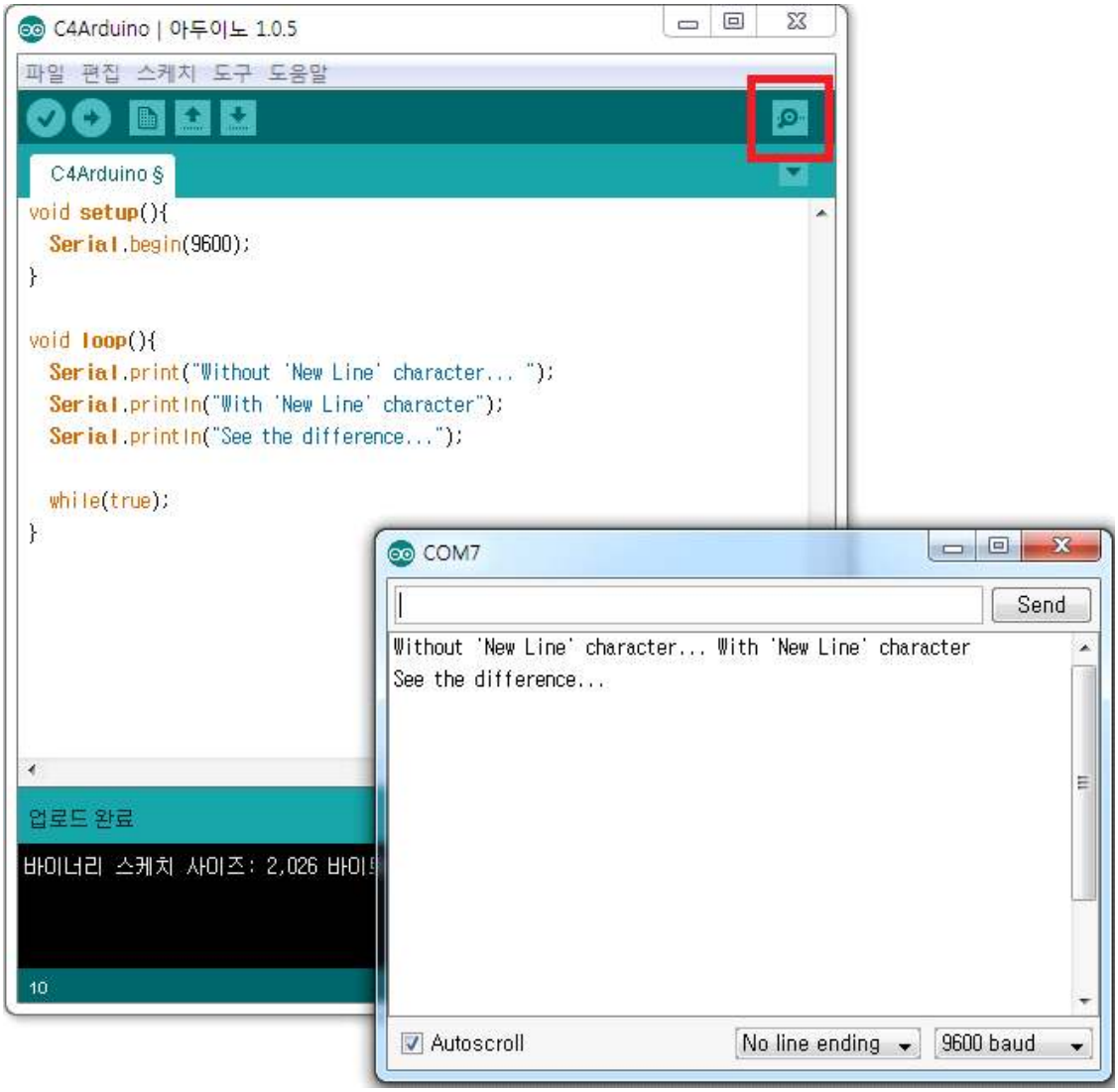


그림 1. 시리얼 모니터를 통한 출력 결과 확인

1. main 함수

C는 함수 중심의 언어인 반면 C++은 연관된 함수들과 변수들을 묶어 놓은 클래스를 기본 단위로 하는 객체 지향 언어이다. 이러한 개념적인 차이에도 불구하고 C와 C++의 공통점 중 하나는 프로그램의 시작점으로써 main 함수를 반드시 필요로 한다는 점이다. main 함수는 프로그램의 진입점 역할을 하며 프로그램이 실행될 때 가장 먼저 실행되는 함수에 해당한다. 하지만 스케치에는 main 함수가 존재하지 않으며 프로그램이 처음 실행될 때 한 번만 실행되는 setup 함수와 아두이노가 꺼질 때까지 반복 호출되는 loop 함수를 기본으로 하고 있다. 아두이노는 프로그래밍에 익숙하지 않은 사람들이 쉽게 프로그램을 작성할 수 있도록 C/C++에

대한 지식은 최소로 하고 직관적인 프로그래밍을 위해 이러한 구조를 채택하고 있지만 C/C++에 경험이 있다면 당황스러울 수도 있다. 스케치에 main 함수는 보이지 않는 곳에 숨겨져 있으므로 신경 쓰지 않아도 될 뿐이다.

2. 데이터 타입

데이터 타입은 메모리에 저장되는 값을 해석하는 기준을 제시하기 위해 필요하다. 예를 들어 65라는 값이 메모리에 저장되어 있다고 가정해 보자. 만약 이 값이 정수라면(int 형 변수로 선언되었다면) 숫자 65로 해석될 것이고 문자라면(char 형 변수로 선언되었다면) 대문자 'A'로 해석될 것이다. 이처럼 메모리에 동일한 값이 저장되어 있더라도 해석하는 방식에 따라 다른 의미가 되므로 해석의 기준을 데이터 타입으로 제시한다. 메모리에 데이터를 저장할 때 데이터 타입에 따라 메모리에서 차지하는 크기 역시 다르다. 예를 들어 ASCII 코드를 사용하는 문자형(char 형)의 경우 512글자까지 서로 다른 문자를 지정할 수 있으므로 1 바이트($2^8 = 512$)면 모든 글자를 표현할 수 있다. 반면 정수형(int 형)의 경우 1,000을 저장하기 위해서는 최소한 10비트가 필요하며 사용하는 메모리의 크기가 크면 클수록 더 큰 정수를 저장할 수 있다. 값에 따라 서로 다른 메모리 크기를 사용한다면 값의 해석에 혼란이 올 수 있으므로 데이터 타입에 따라 고정된 크기의 메모리를 할당해서 사용한다.

아두이노 프로그램은 C/C++을 기반으로 하고 있으므로 C/C++에서 사용되는 데이터 타입과 크게 다르지 않다. 다만 AVR 기반의 보드(Arduino UNO)는 8비트 마이크로프로세서를 사용하는 반면 ARM 기반의 보드(Arduino Due)는 32비트 마이크로프로세서를 사용하고 있어 데이터 타입에 따라 필요로 하는 메모리의 크기가 다른 점에 주의하여야 한다. 표 1은 아두이노 프로그래밍에서 사용되는 주요 데이터 타입을 요약한 것이이며 아래 설명은 ATmega 마이크로컨트롤러를 사용하는 보드를 기준으로 한다.

데이터 타입	크기 (byte)		설명	비고
	ATmega 기반	ARM 기반		
boolean	1		논리형	true 또는 false
char	1		문자형	
byte	1		부호 없는 정수형	
int	2	4	정수형	
word	2	4	부호 없는 정수형	
long	4		정수형	
short	2		정수형	
float	4		단정도 실수형	
double	4	8	배정도 실수형	

표 1. 데이터 타입

- boolean

true나 false의 논리값을 저장하기 위해 사용한다. 논리값 저장은 1비트로 가능하지만 boolean 타입은 1 바이트 메모리를 차지한다는 점에 유의하여야 한다.

- char

하나의 문자를 저장하기 위해 사용하며 1 바이트 메모리를 차지한다. char 타입이 문자를 저장하기 위해 사용되지만 내부적으로는 ASCII 코드값에 해당하는 숫자로 저장되므로 char 타입에 대한 산술연산이 가능하다. 따라서 다음 두 문장은 동일한 의미를 가진다.

```
char ch1 = 'A';
char ch2 = 65;
```

char 타입은 부호가 있는 타입이므로 숫자 -128에서 127로 저장된다. 따라서 char 타입을 문자로 변환하여 출력하는 경우 128~255 범위의 값을 가지는 확장 ASCII 코드의 경우 예상치 못한 결과를 가져올 수 있다. 부호 없는 1 바이트 데이터 타입을 위해서는 byte 타입을 사용하면 된다.

- unsigned char

부호 없는 문자형 타입으로 1 바이트의 메모리를 차지하며 byte 타입과 동일하다. 0에서 255 사이의 숫자로 저장된다. 아두이노 프로그래밍에서는 byte 타입을 많이 사용하므로 byte 타입 사용을 권장한다.

- byte

1 바이트의 부호 없는 숫자 데이터를 저장하기 위해 사용한다. 0에서 255 ($=2^8-1$) 사이의 숫자를 저장할 수 있다.

- int

정수형 데이터 저장을 위한 데이터 타입이다. Arduino UNO를 포함하여 ATmega 기반의 보드에서 int 타입은 2바이트 크기의 메모리를 차지하며 -32,768 (-2^{15})에서 32,767 ($2^{15}-1$) 사이의 값을 저장할 수 있다. 음수는 2의 보수 형태로 저장된다. Arduino Due는 ARM 아키텍처를 사용하는 보드로 int 타입이 4 바이트 메모리를 차지한다. 따라서 -2^{31}에서 $2^{31}-1$ 사이의 값을 저장할 수 있다. int 타입을 사용하여 정수형 값을 연산할 경우 그 범위에 유의하여야 한다. 저장 가능한 최대값이나 최소값을 넘어서는 경우 오버플로에 의한 롤오버가 발생한다.

```
int x;
x = -32768;
x = x - 1;        // x = 32,767 (Arduino UNO의 경우)

x = 32767;
x = x + 1;        // x = -32,768 (Arduino UNO의 경우)
```

- unsigned int

Arduino UNO를 포함하여 ATmega 기반의 보드들에서는 2 바이트의 부호 없는 정수형 ($0 \sim 2^{16}-1$)을 저장하기 위해 사용된다. Arduino Due에서는 int와 마찬가지로 4 바이트 크기를 가지며 $0 \sim 2^{32}-1$ 범위의 값을 저장할 수 있다. int 형의 경우 2의 보수를 사용하므로 MSB(Most Significant Bit)는 부호를 나타내는 부호 비트(sign bit)로 1이면 음수가 되지만 unsigned int의 경우에는 전체를 크기(magnitude)를 나타내기 위해 사용한다.

• float

단정도 부동 소수점 방식의 실수를 저장하기 위해 사용된다. 부동 소수점을 표현하기 위해 4 바이트 메모리를 필요로 하며 -3.4028234×10^{38}에서 3.4028235×10^{38} 사이의 값을 표현할 수 있다.

float 타입은 6~7 자리의 유효 자릿수를 가지므로 연산 결과는 정확하지 않을 수 있다. 따라서 실수값을 비교하는 (6.0 / 3.0 == 2.0)은 잘못된 결과를 반환할 수 있으므로 오차를 고려하여 (abs(6.0 / 3.0 - 2.0) < 1e-5)와 같이 사용하는 것이 안전하다.

실수 연산은 정수 연산에 비해 연산 속도가 느리므로 타이밍이 중요한 코드를 실행하고자 하는 경우에는 가능한 피하는 것이 좋다.

• double

배정도 부동 소수점 방식의 실수를 저장하기 위해 사용된다. Arduino UNO를 포함하여 ATmega 기반의 보드에서는 4 바이트의 메모리를 차지하므로 float와 동일하다. Arduino Due에서는 8 바이트의 메모리를 차지한다.

• string - char 배열

문자열은 두 가지로 표현 가능하며 그 중 하나가 C/C++에서 사용하는 char 배열을 이용하여 표현하는 방법이다. C/C++에서와 같이 다양한 방법으로 문자열 저장을 위한 char 배열을 생성하고 초기화할 수 있다.

```
char Str1[15]; // 초기화 없음
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'}; // 명시적인 널 문자 추가
char Str4[ ] = "arduino"; // 배열 크기 지정 없이 초기화
char Str5[8] = "arduino";
char Str6[15] = "arduino"; // 초기값보다 큰 배열 지정
```

배열을 이용하여 문자열을 저장하는 경우 문자열의 끝에는 항상 널 문자(NULL character, ASCII code 0)를 이용하여 문자열의 끝을 표시하여야 한다. 이를 위해 문자열 저장을 위한 배열은 실제 문자열의 길이보다 최소한 1 크게 설정하여야 한다. 널 문자 없이 문자열을 저장

하는 것도 가능하지만 대부분의 기존 함수들이 널 문자로 문자열의 종료를 확인하므로 추천하지 않는다. char 타입과 char 배열은 각각 작은따옴표와 큰따옴표를 이용하여 초기화 하므로 주의하여야 한다.

```
char ch = 'A';          // 문자. 1 바이트 메모리 소요
char str[] = "A";       // 문자열. 2 바이트 메모리 소요 (NULL 문자 포함)
```

- String

String 클래스는 문자열을 다루기 위한 전용 클래스로 문자형 배열로 문자열을 다루는 것과 비교할 때 다양한 작업을 보다 쉽게 사용할 수 있도록 해준다. String 클래스에는 문자열 연결, 검색, 바꾸기 등 여러 문자열 조작을 지원하지만 문자 배열에 비해 많은 메모리를 필요로 하는 단점이 있다. 큰 따옴표로 표시되는 문자열 상수는 내부적으로 String 클래스가 아니라 문자 배열로 다루어진다. String 클래스의 멤버 함수는 String 클래스 장을 참고하면 된다.

코드 2는 Arduino UNO에서 각 데이터 형이 실제 필요로 하는 메모리 크기를 시리얼 모니터로 출력하는 코드이며 실행 결과는 그림 2와 같다. 코드 2에서 sizeof는 각 데이터 타입이 필요로 하는 메모리의 크기를 반환하는 연산자이다.

코드 2

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

void loop()
{
  Serial.println("boolean : " + String(sizeof(boolean)));
  Serial.println("char    : " + String(sizeof(char)));
  Serial.println("byte    : " + String(sizeof(byte)));
  Serial.println("int     : " + String(sizeof(int)));
  Serial.println("word    : " + String(sizeof(word)));
```

```
    Serial.println("long      : " + String(sizeof(long)));
    Serial.println("short     : " + String(sizeof(short)));
    Serial.println("float     : " + String(sizeof(float)));
    Serial.println("double    : " + String(sizeof(double)));

    while(true);
}
```

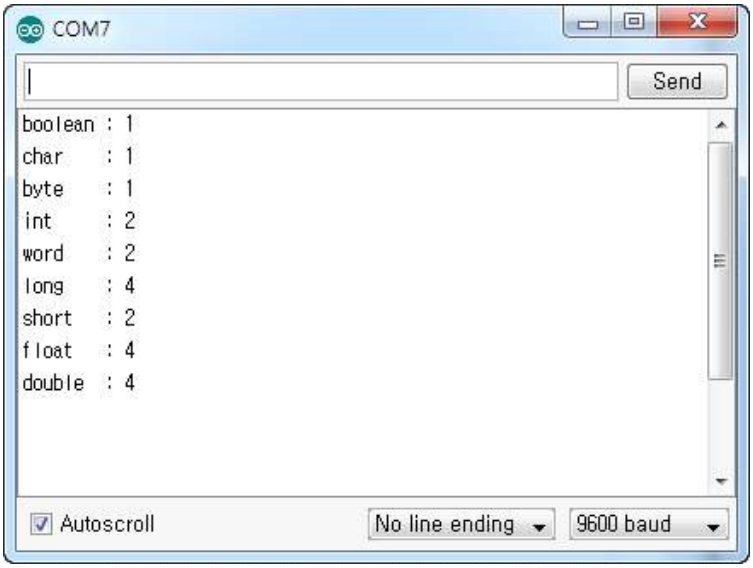


그림 2. 코드 2 실행 결과

각 데이터 타입으로 선언된 변수는 유효 범위(scope)를 가진다. 유효 범위는 변수에 저장된 값을 볼 수 있는 범위를 나타낸다. 변수는 크게 지역변수(local variable)와 전역변수(global variable)로 나누며 지역변수의 경우 가장 가까이에 있는 중괄호 쌍 내에서만 유효하다. 반면 전역변수는 코드 전체에서 유효하다. C/C++에서 중괄호 내에 위치하는 코드는 하나의 블록을 형성하고 하나의 블록은 여러 개의 하위 블록을 포함할 수 있으며 블록들은 계층적 구조를 가진다. 블록이 중첩된 경우 유효 범위가 가장 좁은 변수가 우선하며 이는 동일한 이름의 변수가 여러 개 존재할 때 유효한 변수를 결정하기 위해 사용된다. 하지만 동일한 수준의 블록 내에서 동일한 이름의 변수가 두 개 이상 존재할 수 없다. 코드 2는 변수의 유효 범위를 보여주는 예이다.

코드 3

```
int A = 0;                          // 전역 변수. Level 0

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

void loop()
{
  // 전역 변수 A 참조. 변수는 사용 이전에 선언되어야 하며
  // 동일한 레벨에서 선언되어 있지 않으면 상위 레벨의 변수를 사용한다.
  Serial.println("Level 0 : " + String(A));

  // loop 함수 내에서만 유효하다. Level 1
  int A = 1;
  Serial.println("Level 1 : " + String(A));

  if(A == 1){
    // loop 함수 내의 if 블록에서만 유효하다. Level 2
    int A = 2;
    Serial.println("Level 2 : " + String(A));

    {
      // loop 함수 내, if 블록 내의 중괄호({ }) 블록에서만 유효하다. Level 3
      int A = 3;
      Serial.println("Level 3 : " + String(A));
    }
  }

  // loop 함수 내에 정의된 변수 A 참조한다.
  // 동일한 레벨의 블록 내에 선언된 변수를 먼저 참조한다.
  Serial.println("Level 1 : " + String(A));

  while(true);
}
```

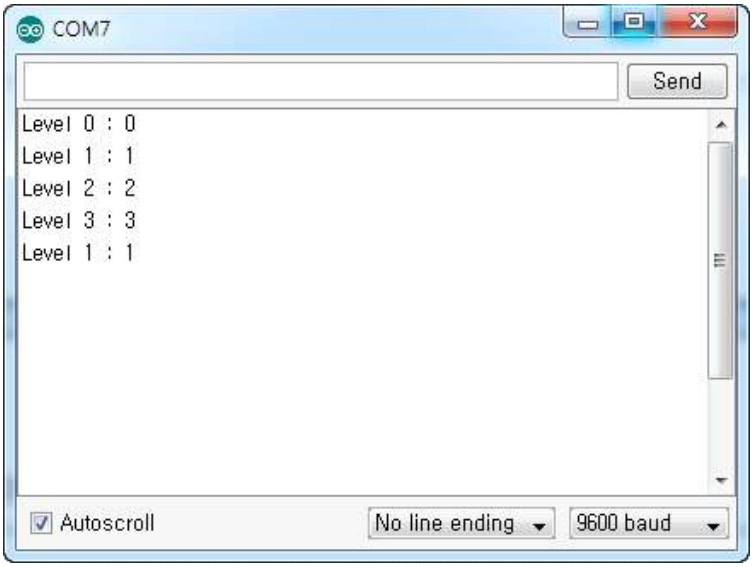


그림 3. 코드 3 실행 결과

변수는 유효 범위 이외에도 몇 가지 지정자를 사용할 수 있다. 스케치에서 사용하는 지정자로는 static, volatile, const가 있다.

- static

일반적으로 지역 변수는 블록 내에서 생성되고 블록을 빠져나가면, 즉, 변수의 유효 범위를 벗어나면 소멸된다. 하지만 변수를 static으로 선언하면 변수는 유효 범위를 벗어나도 소멸되지 않고 그 값을 유지한다. 하지만 static 변수도 일반 변수와 선언 위치에 따라 동일한 유효 범위를 가진다.

코드 4

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

void loop()
{
  int A = 0;                    // 일반 지역 변수
  static int B = 0;             // 정적 지역 변수
```

```
    A = A + 1;                    // 일반 지역 변수 1 증가
    B = B + 1;                    // 정적 지역 변수 1 증가

    Serial.println("A = " + String(A));
    Serial.println("B = " + String(B));

    delay(1000);                  // 1초 대기
}
```

그림 4. 코드 4 실행 결과

그림 4의 실행 결과에서 알 수 있듯이 일반 지역 변수는 loop 함수가 호출될 때마다 생성되어 0으로 초기화되므로 매번 1을 출력한다. 반면 정적 변수는 최초에 loop 함수가 호출될 때 생성되고 이후 동일한 변수가 사용되므로 값이 증가하는 것을 볼 수 있다.

- volatile

volatile 키워드는 컴파일러가 스케치를 컴파일 할 때 최적화에서 제외하도록 하는 역할을 한다. 최적화 과정은 실행 파일의 크기를 줄이거나 속도를 높이는 기능을 수행하지만 완벽하지는 않다. 특히 실행 과정 중에서 정상적인 처리 순서에서 벗어나 명령어를 처리하는 경우 최적화로 인해 잘못된 결과가 나올 수 있으며 아두이노 프로그램의 경우 인터럽트에 의해 정상

적인 처리 순서를 벗어나 인터럽트 처리 루틴으로 이동하는 경우가 이에 해당한다. 따라서 인터럽트 서비스 루틴에서 값을 변경하는 변수는 volatile로 선언해주는 것이 안전하다.

- const

const는 상수(constant)를 나타내며 선언에서 값이 할당된 이후에는 그 값을 바꿀 수 없는 변수를 나타낸다. 상수를 선언하는 또 다른 방법은 전처리 명령어인 #define을 사용하는 방법이 있지만 #define은 전처리기에서 처리하는 명령어로 유효 범위가 없고 메모리를 차지하지 않으므로 값을 대입할 수 없다. 반면 const에 의한 상수 변수는 선언할 때 한 번 값을 대입할 수 있지만 이후 변경이 불가능하고 일반 변수와 동일하게 메모리를 차지하고 유효 범위를 가지므로 구조적인 코드 작성이 가능한 장점이 있다. 코드 4에서 #define에 의한 CONST1에 1을 더하는 코드의 주석을 제거하면 “lvalue required as left operand of assignment”라는 오류가 발생한다. 이는 CONST1이 메모리를 차지하지 않기 때문에 값을 대입할 수 없으므로 발생하는 오류이다. 반면 CONST2에 1을 더하는 코드의 주석을 제거하면 “assignment of read-only variable ‘CONST2’”라는 오류가 발생한다. 이는 CONST2 변수가 상수 변수로 값을 변경할 수 없기 때문에 발생하는 오류이다.

코드 5

```
#define CONST1 5                    // 상수 정의 (단순히 CONST1을 5로 대체한다.)

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

void loop()
{
  const int CONST2 = 10;            // 상수 변수. 이후 값을 변경할 수 없는 변수

  Serial.println("Constant 1 by #define : " + String(CONST1));
  Serial.println("Constant 2 by const keyword : " + String(CONST2));

  // CONST1 = CONST1 + 1;
  // CONST2 = CONST2 + 1;
```

```
    while(true);
}
```

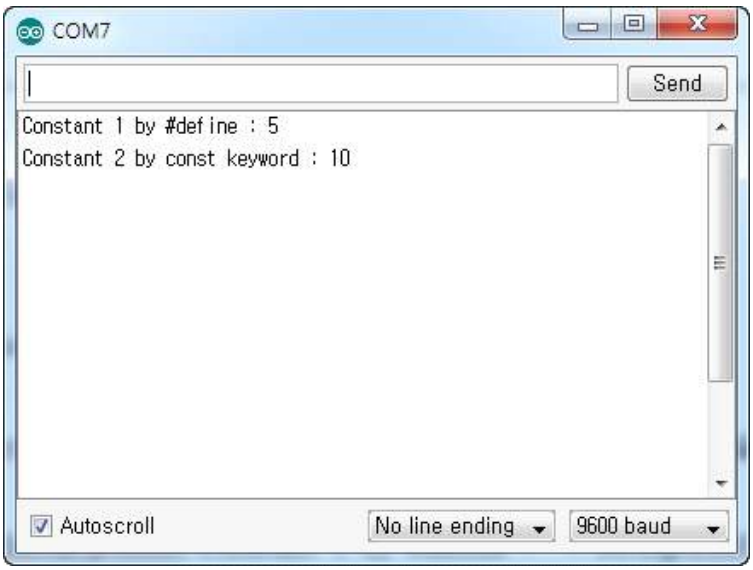


그림 5. 코드 4 실행 결과

3. 연산자

아두이노 프로그래밍에서 사용할 수 있는 연산자는 C/C++에서 사용하는 연산자들을 모두 사용할 수 있으며 산술 연산자, 비교 연산자, 논리 연산자, 비트 연산자, 복합 연산자 등으로 나눌 수 있다.

- 산술 연산자

C/C++에서는 정수형 및 실수형 값에 대하여 표 2와 같은 산술 연산자를 제공한다.

연산자	의미	사용 예	비고
+	더하기	a = b + 3;	
-	빼기	a = b - 3;	
*	곱하기	a = b * 3;	
/	나누기	a = b / 3;	
%	나머지	a = b % 3;	정수형만 가능
=	대입		비교 연산자 '=='와 구별됨

표 2. 산술 연산자

사칙연산을 수행함에 있어 정수형 사이의 연산은 그 결과가 정수로 나온다. 따라서 '3 / 2'를 계산하면 1.5가 아닌 1의 결과를 얻는다. 1.5를 얻기 위해서는 '3.0 / 2.0'을 계산하여야 한다. 예에서 알 수 있듯이 소수점이 있는 경우에는 실수로, 소수점이 없는 경우에는 정수로 간주되므로 연산 과정에서 주의가 필요하다.

프로그램 작성 중 흔히 발생하는 논리 오류 중 하나는 대입 연산자인 '='와 등치 연산자인 '=='의 혼돈으로 인한 오류이다. 수학에서와 다르게 '='는 오른쪽의 연산 결과를 왼쪽으로 대입하는 것을 나타낸다. 수학에서 동일한 값에 해당하는 C/C++ 연산자는 '=='임에 유의하여야 한다.

코드 6

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

void loop()
{
  int a = 10, b = 3;                // 정수형 변수
  float fa = 10.0, fb = 3.0;        // 실수형 변수

  Serial.println("a % b = " + String(a % b));        // 나머지
  Serial.println("a / b = " + String(a / b));        // 정수 나눗셈
  Serial.print("fa / fb = ");
```

```
    Serial.println(fa / fb);                    // 실수 나눗셈

    while(true);
}
```

코드 6에서 볼 수 있듯이 String 클래스는 실수값을 문자열로 변환하는 함수를 제공하지 않지만 Serial 클래스는 실수값을 출력하는 기능을 제공하고 있음에 유의하여야 한다.

그림 6. 코드 6 실행 결과

- 비교 연산자

비교 연산자는 두 개의 피연사자를 비교하는 연산자로 관계 연산자(relational operator)라고도 불린다. 비교 연산의 결과는 true 또는 fals로 주어지며 조건문이나 반복문에서 흔히 사용된다. 표 3은 비교 연산자를 요약한 것이다. 비교 연산자의 사용에서 주의할 점은 '크거나 같다'에서와 같이 두 개의 기호로 구성되는 연산자의 경우 반드시 붙여서 써야한다는 점이다. '> ='와 같이 띄어 쓰면 오류가 발생한다. 또 한 가지는 두 값이 같은지 비교하는 연산자는 '==' 이며 수학에서 사용하는 '='은 대입의 의미한다는 점이다.

연산자	의미	수학식
x > y	크다	$x > y$
x >= y	크거나 같다	$x \geq y$
x < y	작다	$x < y$
x <= y	작거나 같다	$x \leq y$
x == y	같다	$x = y$
x != y	다르다	$x \neq y$

표 3. 비교 연산자

코드 7은 대입 연산자와 등치 연산자를 혼돈해서 사용한 예를 보여준다. 데이터 유형에서도 언급하였듯이 논리형은 1비트로 저장할 수 있으나 실제로는 1바이트에 저장된다. 따라서 0은 false로 간주되고 0이 아닌 모든 값은 true로 간주된다.

코드 7

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

void loop()
{
  int a = 10;

  if(a == 10){                    // a가 10인지 비교
    Serial.println("a is ten !!");
  }

  if(a = 5){
    // a에 5를 대입하면 a는 0이 아닌 값을 가지므로 논리값 true를 가진다.
    // 따라서 if 블록의 코드가 실행된다.
    Serial.println("a is five ??");
  }
```

```
    while(true);
}
```

그림 7. 코드 7 실행 결과

- 논리 연산자

C/C++ 언어에는 불 논리에서의 기본 연산인 AND, OR, NOT에 대응하는 연산자가 제공된다. 논리 연산자는 비교 연산자와 마찬가지로 그 결과가 true 또는 false로 나타나며 논리 연산자의 피연산자 역시 true나 false 값을 가지는 논리값이어야 한다. C/C++에서 제공되는 논리 연산자는 표 4와 같다.

논리 연산자	의미	사용 예
&&	AND	(a > 100) && (a < 200)
||	OR	(a == 100) || (b == 200)
!	NOT	!(a < 100)

표 4. 논리 연산자

- 비트 연산자

마이크로프로세서는 모든 자료를 이진값으로 처리하며 처리의 최소 단위는 비트이다. C/C++ 언어에서는 논리 연산자와 더불어 비트 값을 처리할 수 있는 비트 연산자들을 통해 비트 단위

의 논리 연산을 수행할 수 있도록 해준다. 한 가지 주의할 점은 비트 연산자들이 비트 단위의 연산이 가능하도록 해주지만 실제로 연산은 최소한 8개의 비트를 묶어서 바이트 단위로 이루어진다는 점이다. 표 5는 C/C++ 언어에서 제공하는 비트 연산자를 나타낸다. 비트 AND 및 OR 연산자는 논리 AND 및 OR 연산자와 유사하므로 주의하여야 한다.

구분	연산자	종류	결과
비트 논리 연산자	a & b	비트 AND	a와 b의 비트 단위 AND
	a | b	비트 OR	a와 b의 비트 단위 OR
	a ^ b	비트 XOR	a와 b의 비트 단위 XOR
	~a	비트 NOT	a의 비트 단위 NOT (1의 보수)
비트 이동 연산자	a << n	왼쪽으로 이동	a를 n비트 왼쪽으로 이동하고 오른쪽은 0으로 채움
	a >> n	오른쪽으로 이동	a를 n비트 오른쪽으로 이동하고 왼쪽은 0으로 채움

표 5. 비트 연산자

마이크로프로세서 프로그래밍에서는 일반 컴퓨터 프로그래밍과는 다르게 비트 연산을 많이 사용한다. 이는 마이크로프로세서 프로그래밍 과정에서는 해당 레지스터를 조작하는 작업이 필수적이며 레지스터는 비트별로 그 의미가 다르기 때문에 비트 단위 연산을 통해 값을 바꾸어 주는 작업이 필요하기 때문이다. 아두이노의 경우 이러한 레지스터 조작을 대부분 추상화하여 함수 형태로 제공하고 있지만 비트 연산을 이해함으로써 코드의 이해도를 높일 수 있다. 일반적으로 특정 비트를 1로 만드는 작업을 '세트(set)', 특정 비트를 0으로 만드는 작업을 '클리어(clear)'라고 표현하며 마이크로컨트롤러 프로그래밍 과정에서 많이 사용되는 비트 연산에는 다음과 같은 것들이 있다.

- 특정 비트 세트
- 특정 비트 클리어
- 특정 비트 반전
- 특정 비트 검사

먼저 숫자 표기 방법을 살펴보자. 일반적인 C/C++ 프로그래밍에서 숫자는 10진수로 표현되는 것이 일반적이다. 하지만 마이크로프로세서 프로그래밍에서는 비트별 의미 파악을 쉽게 할 수

있도록 16진수를 많이 사용한다. 한 가지 유의할 점은 C/C++ 프로그래밍에서는 2진수 표현을 위한 표기법이 별도로 제공되지 않지만 아두이노에서는 16진수가 '0x'로 시작되는 것과 유사하게 '0b'로 시작되는 숫자는 2진수로 취급된다. 2진수로 값을 표현하면 위치별 비트 값 파악이 쉬운 장점은 있지만 숫자 표현이 길어지고 읽거나 쓸 때 잘못될 가능성이 있으므로 주의하여야 한다. 십진수 36을 진법에 따라 표현하는 방법은 다음과 같다.

```
a = 36;
b = 0x24;              // 16진수 표현, '0x'로 시작
c = 0b00100100;        // 2진수 표현, '0b'로 시작
```

① 특정 비트 세트

특정 비트 세트는 특정 위치의 비트를 1로 설정하고 나머지는 현재 값을 그대로 유지하는 것을 말한다. 이를 위해서는 비트 OR 연산을 사용할 수 있다. 비트 OR 연산은 그림 8에서와 같이 비트 단위로 OR 연산을 수행하며 1과 OR 시키면 결과는 항상 1이 되고 0과 OR 시키면 현재 값이 그대로 유지되는 것을 볼 수 있다.

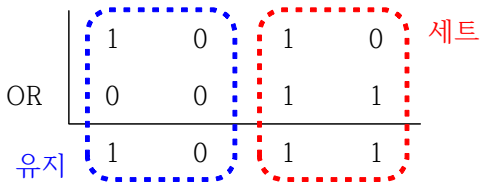


그림 8. 비트 OR 연산

따라서 a = 0bxxxxxxxxx; 값이 저장된 경우 3번째 비트만은 1로 설정하고 나머지 비트들은 현재 값을 그대로 유지하기 위해서는 그림 11에서와 같이 0b00000100과 OR 연산을 수행하면 된다.

	x	x	x	x	x	x	x	x
OR	0	0	0	0	0	1	0	0
	x	x	x	x	x	1	x	x

그림 9. 특정 비트 세트

그림 9을 코드로 나타내면 다음과 같다.

```
a = a | 0b00000100;
a = a | 0x04;
```

위의 코드에서 0b00000100와 0x04는 동일한 값으로 '마스크(mask)'라 불린다. 마스크를 직접 숫자로 기입하는 것은 종종 오류의 원인이 되므로 비트 이동 연산을 통해 마스크를 만들어 사용하기도 한다. 마스크 생성을 위해서는 왼쪽 비트 이동 연산자인 '<<'를 사용하며 오른쪽 빈 칸은 0으로 채워진다.

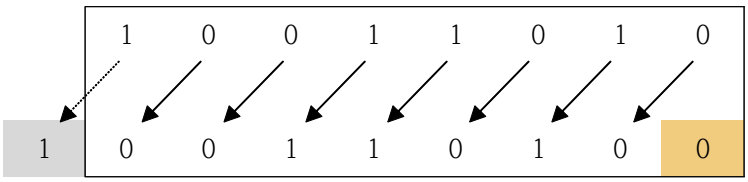


그림 10. 왼쪽 비트 이동 연산

그림 10의 비트 이동 연산을 사용하면 3번째 비트 세트 연산은 다음과 같이 표시할 수 있다.

```
a = a | 0b00000100;
a = a | 0x04;
a = a | (0x01 << 2);
```

위의 코드에서 (1 << 2) 부분이 0x04 마스크를 만들어내는 부분으로 3번째 비트만을 1로 설정하기 위해서는 2 비트만큼 왼쪽으로 이동시킨다. 이 때 세 번째 비트가 특별한 의미를 가지는 경우, 예를 들어 3번째 비트를 1로 세트하는 것이 3번째 LED를 키는 동작인 경우 코드 가독성을 높이기 위해 #define 문장을 사용하기도 한다.

```
#define LED_3_ON        2
a = a | (0x01 << LED_3_ON);
```

② 특정 비트 클리어

특정 비트 클리어는 특정 위치의 비트를 0으로 설정하고 나머지는 현재 값을 그대로 유지하는 것을 말한다. 이를 위해서는 비트 AND 연산을 사용할 수 있다. 비트 AND 연산은 그림 11과 같이 비트 단위로 AND 연산을 수행하며 0과 AND 시키면 결과는 항상 0이 되고 1과 AND 시키면 현재 값이 그대로 유지되는 것을 볼 수 있다.

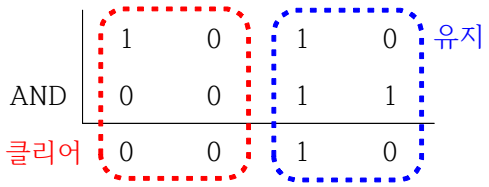


그림 11. 비트 AND 연산

따라서 a = 0bxxxxxxxx; 값이 저장된 경우 5번째 비트만은 0으로 설정하고 나머지 비트들은 현재 값을 그대로 유지하기 위해서는 그림 12에서와 같이 0b11101111과 AND 연산을 수행하면 된다.

	x	x	x	x	x	x	x	x
AND	1	1	1	0	1	1	1	1
	x	x	x	0	x	x	x	x

그림 12. 특정 비트 클리어

그림 12를 코드로 나타내면 다음과 같다.

```
REG = REG & 0b11101111;
REG = REG & 0xEF;
REG = REG & ~(0x01 << 4);
```

비트 클리어의 경우 클리어하는 비트 값을 0으로 하고 나머지 비트들은 1로 설정해야 한다. 이는 비트 세트와 반대의 경우로 비트 이동 연산자를 사용하는 경우 이를 구현하기 위해서는 그림 10에 주어진 것처럼 비트 세트에서와 동일한 방식으로 마스크를 만들고 이를 반전(~, 비트 NOT)시켜 사용한다.

③ 특정 비트 반전

특정 비트 반전은 특정 위치의 비트를 반전시키고 나머지는 현재 값을 그대로 유지하는 것을 말한다. 이를 위해서는 비트 XOR 연산을 사용할 수 있다. XOR 연산은 두 피연산자가 동일한 값을 가질 때 0이 되며 다른 값을 가질 때 1이 된다. 비트 XOR 연산은 그림 13과 같이 비트 단위로 XOR 연산을 수행하며 1과 XOR 시키면 결과는 반전된 값이 되며 0과 XOR 시키면 현재 값이 그대로 유지되는 것을 볼 수 있다.

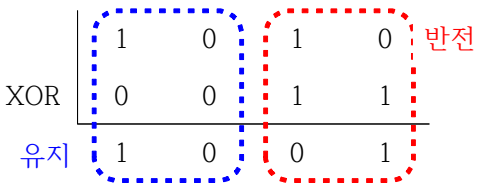


그림 13. 비트 XOR 연산

따라서 a = 0bxxxxxxxx; 값이 저장된 경우 4번째 비트만은 반전시키고 나머지 비트들은 현재 값을 그대로 유지하기 위해서는 그림 14에서와 같이 0b00001000과 XOR 연산을 수행하면 된다.

	x	x	x	x	x	x	x	x
XOR	0	0	0	0	1	0	0	0
	x	x	x	0	$\bar{x}$	x	x	x

그림 14. 특정 비트 반전

그림 12를 코드로 나타내면 다음과 같다.

```
REG = REG ^ 0b00001000;
REG = REG ^ 0x08;
REG = REG ^ (1 << 3);
```

④ 특정 비트 검사

특정 비트 검사는 특정 위치의 비트 값을 구하는 것을 말한다. 특정 비트 클리어에서와는 반대로 구하고자 하는 비트 위치에 해당하는 마스크 값만을 1로하고 나머지 비트 값들을 0으로 설정하고 비트 AND 연산을 수행함으로써 특정 위치의 비트 값을 알아낼 수 있다. a = 0bxxxxxxxx; 값이 저장된 경우 6번째 비트 값을 알아내기 위해서는 그림 15에서와 같이 0b00001000과 AND 연산을 수행하면 된다.

	x	x	x	x	x	x	x	x
AND	0	0	1	0	0	0	0	0
	0	0	x	0	0	0	0	0

그림 15. 특정 비트 검사

그림 15를 이용하여 특정 비트를 검사하는 코드는 다음과 같다.

```
if( (REG & 0x20) == 0x20 ){ // 6번째 비트 검사
        // 비트가 1인 경우
}
else{
        // 비트가 0인 경우
}
```

위의 예에서 나타나 있듯이 특정 비트를 검사한 결과는 마스크와 동일한 값으로 비교하는 것이 안전하다. C/C++ 언어에서는 0이 아닌 모든 정수 값은 논리 참(true)으로 간주하므로 마스크 값과 비교하지 않아도 결과는 동일하지만, 이 경우 정수값과 논리값이 혼재되어 나타나므로 추천하지 않는다.

```
if( REG & 0x20 ){ // 6번째 비트 검사
        // 비트가 1인 경우
        // 실제 if 내의 식은 32의 값을 가지는 정수값이므로
        // 논리값과 정수값의 혼돈을 유발할 수 있으므로 추천하지 않는다.
}
else{
        // 비트가 0인 경우
}
```

특정 비트를 검사하는 결과는 종종 오른쪽 비트 이동(>>) 연산자를 이용하여 결과를 0 또는 1로 만들어 사용하기도 한다. 오른쪽 비트 이동은 그림 10의 왼쪽 비트 이동과 유사하며 왼쪽 빈 칸이 0으로 채워진다.

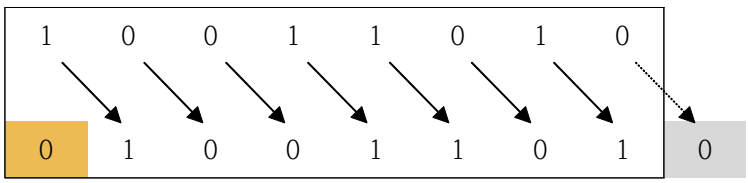


그림 16. 오른쪽 비트 이동 연산

오른쪽 비트 이동 연산을 사용하여 위의 코드는 다음과 같이 수정될 수 있다.

```
if( ((REG & 0x20) >> 5) == 1 ){ // 6번째 비트 검사
        // 비트가 1인 경우
}
else{
        // 비트가 0인 경우
}
```

또는 값을 먼저 이동시킨 후 비트 연산을 수행하여도 동일한 결과를 얻을 수 있다.

```
if( ((REG >> 5) & 0x01) == 1 ){ // 6번째 비트 검사
        // 비트가 1인 경우
}
else{
        // 비트가 0인 경우
}
```

- 복합 연산자

복합 연산자는 기존의 연산자들을 결합하여 짧게 표현할 수 있도록 한 축약 형태이다. 복합 연산자에는 대입 연산자와 산술 또는 비트 연산자를 결합한 복합 대입 연산자와 증가 또는 감소를 나타내는 증감 연산자가 있다.

분류		연산자	의미
복합 대입 연산자	산술 연산자	a += b;	a = a + b;
		a -= b;	a = a - b;
		a *= b;	a = a * b;
		a /= b;	a = a / b;
		a %= b;	a = a % b;
	비트 연산자	a &= b;	a = a & b;
		a |= b;	a = a | b;
		a ^= b;	a = a ^ b;
		a <<= b;	a = a << b;
		a >>= b;	a = a >> b;
증감 연산자		a++;	a = a + 1;
		a--;	a = a - 1;

표 6. 복합 연산자

4. 제어문

C/C++ 코드는 위에서 아래로 순차적인 실행을 기본으로 한다. 하지만 프로그램 실행 중에 실행 순서는 필요에 따라 바뀔 수 있으며 이처럼 실행의 흐름을 조절하기 위해 필요한 문장에는 조건문과 반복문이 있다. 조건문은 주어진 조건을 만족시키는 경우에만 특정 블록을 한 번 실행하도록 하기 위해 사용되며 if-else 문과 switch-case 문이 있다. 반복문은 특정 블록의 문장을 지정한 횟수만큼 또는 주어진 조건을 만족하는 동안 실행하기 위해 사용되며 while 문, do-while 문, for 문이 있다.

- if-else

if 문은 주어진 조건을 만족하는지에 따라 실행할 문장이 다를 때 사용한다. if 문의 조건은 true나 false의 값을 갖는 문장이 사용되며 조건을 만족시키지 못하는 경우 실행할 문장은 else 문을 통해 지정할 수 있다. 그림 17은 if-else 문의 흐름을 나타낸 것이다.

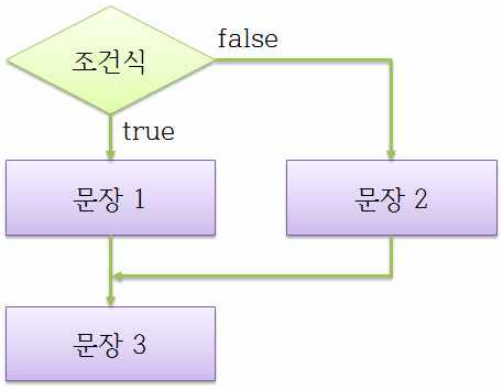


그림 17. if-else 문

그림 17을 코드로 나타내면 다음과 같다.

```
if( 조건식 ){
        // 문장 1
}
else{
        // 문장 2
}
// 문장 3
```

- switch-case

if 문장의 경우 조건의 만족 여부에 따라 두 개의 흐름으로 나눈다. switch 문의 경우에는 if 문장과 달리 true나 false 값을 갖는 조건식이 아닌 정수값을 갖는 정수식으로 주어지므로 필요한 개수만큼의 흐름으로 분리할 수 있다. 그림 18은 switch-case 문의 흐름을 나타낸 것으로 3개의 흐름으로 분리하고 있다.

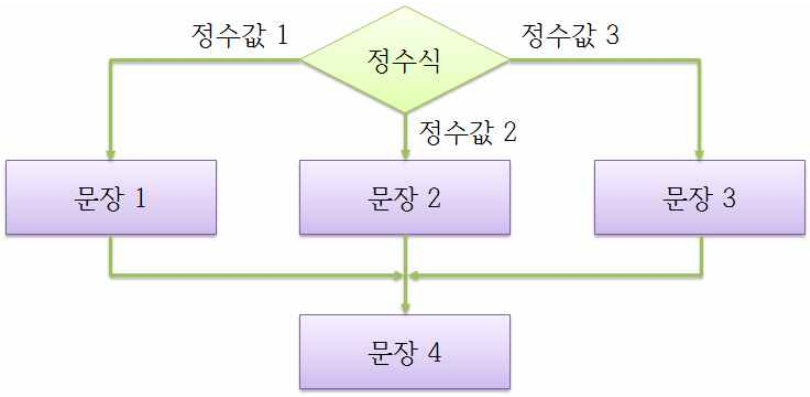


그림 18. switch-case 문

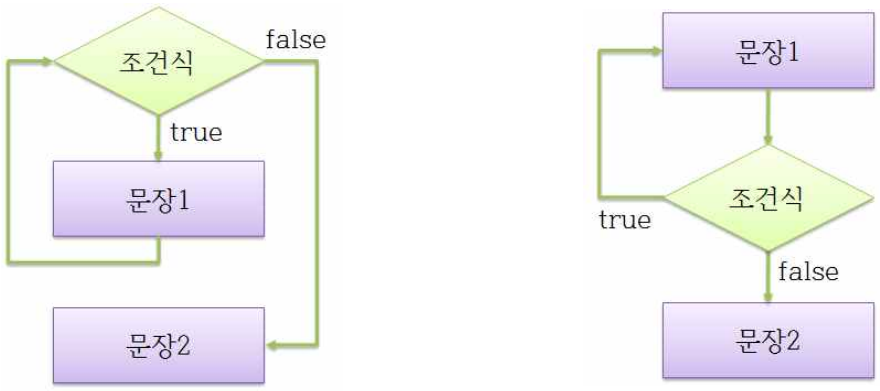
그림 18을 코드로 나타내면 다음과 같다.

```
switch( 정수식 ){
        case 정수값1:
                // 문장 1
                break;
        case 정수값2:
                // 문장 2
                break;
        case 정수값3:
                // 문장 3
                break;
}

// 문장 4
```

- while과 do-while

while 문과 do-while 문은 주어진 조건식을 만족하는 동안 반복해서 시행할 문장을 지정하기 위해 사용한다. 두 문장의 차이는 조건식을 검사하는 위치에서 찾아볼 수 있다. while은 조건 검사가 처음에 이루어지므로 while 문 내에 포함된 문장은 한 번도 실행되지 않을 수 있다. 반면 do-while 문은 조건 검사가 마지막에 이루어지므로 최소한 한 번 실행된다. while 문과 do-while 문의 흐름을 비교한 것이 그림 19이다.



(a) while 문 (b) do-while 문

그림 19. while 문과 do-while 문

그림 19를 코드로 나타내면 다음과 같다.

```
while( 조건식 ){
        // 문장 1
}
// 문장 2
```

```
do{
        // 문장 1
}while ( 조건식 );
// 문장 2
```

- for

for 문은 초기 조건, 반복 조건 등을 한 번에 지정할 수 있어 다양한 제어가 가능할 뿐만이 아니라 유연성이 높아 반복문 중에서도 가장 많이 사용되는 문장이다. 하지만 다양한 형식을 지원하므로 그만큼 이해하기 어려운 점도 사실이다. for 문을 사용하면 while 문이나 do-while 문장으로 표현되는 반복문을 보다 간략하게 표현할 수 있다. for 문의 흐름을 나타낸 것이 그림 20이다.

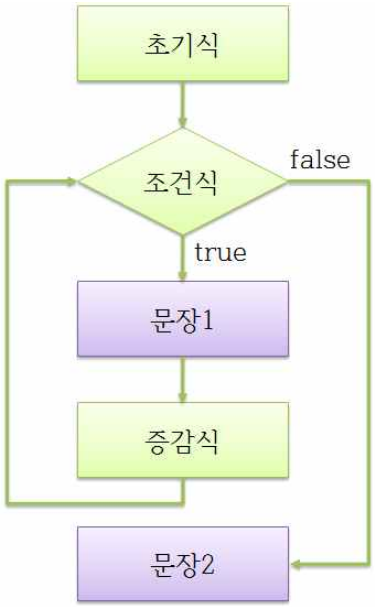


그림 20. for 문

그림 20을 코드로 나타내면 다음과 같다.

```
for( 초기식 ; 조건식 ; 증감식 ){
        // 문장 1
}
// 문장 2
```

아두이노 프로그램에서 종종 무한 반복되는 반복문을 loop 함수 내에서 볼 수 있다. loop 함수 자체가 무한히 반복 호출되는 함수이므로 loop 내 문장들은 계속해서 실행된다. 하지만 테스트 목적이나 기타의 이유로 loop 함수 내의 문장들을 한 번 실행하고 대기하기 위해서는 loop 함수의 마지막에 무한 반복문을 추가하게 된다. 무한 반복문은 모든 반복문으로 만들 수 있지만 while을 이용하는 것이 간단하면서도 이해하기 쉽다.

while(true);	do{ }while(true);	for(; ;);

코드 8은 1부터 10까지 더하는 프로그램을 세 가지 반복문을 사용하여 구현한 예이다. 각 반복문의 구조를 비교하여 살펴보기 바란다.

코드 8

```
void setup() {
  Serial.begin(9600);                          // 직렬 포트 초기화
}

void loop()
{
  int count, sum;

  // do-while 문 사용
  sum = 0; count = 0;
  do{
    count++;
    sum += count;
  }while(count < 10);
  Serial.println("do-while : " + String(sum));

  // while 문 사용
  sum = 0; count = 0;
  while(count < 10){
    count++;
    sum += count;
  }
  Serial.println("while    : " + String(sum));

  // for 문 사용
  for(count = 1, sum = 0; count <= 10; count++){
    sum += count;
  }
  Serial.println("for      : " + String(sum));

  while(true);
```

```
}
```

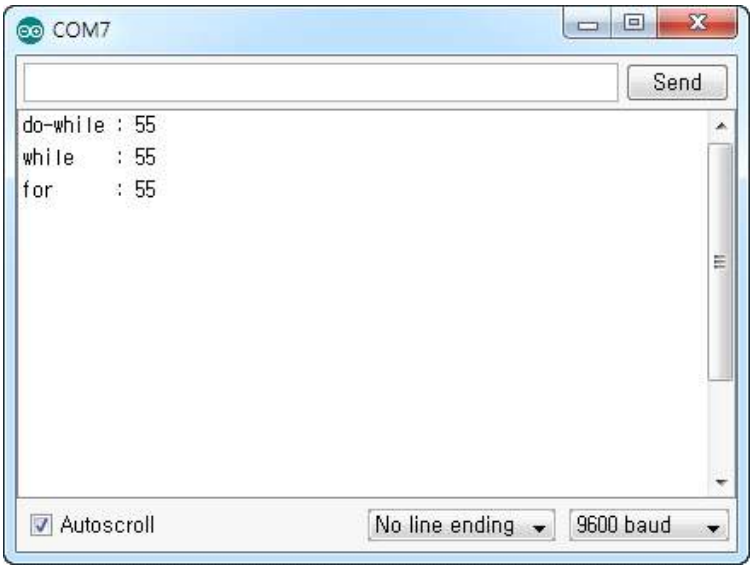


그림 21. 코드 8 실행 결과

이외에도 반복문을 강제로 탈출하는 break, 반복문 내부의 문장 실행을 생략하도록 하는 continue, 함수의 실행을 종료하고 함수를 호출한 곳으로 귀환하는 return 등의 문장이 제어문에 포함된다.

5. 포인터

프로그램은 명령어와 데이터로 구성되며 이들 모두는 실행 시에 메모리에 저장된다. C/C++ 언어에서 사용하는 변수의 값은 메모리의 특정 위치에 저장되며, 변수 선언은 변수의 값을 저장할 메모리 공간을 시스템에 요구하고 할당받는 과정이라 할 수 있다. 따라서 변수는 변수가 저장되는 메모리 내의 '위치'와 그 위치의 메모리에 저장되는 '값'으로 이루어진다. C/C++ 언어에서 가장 이해하기 어려운 부분 중 하나가 바로 포인터(pointer)로 Java에서는 모든 포인터를 암시적으로 처리하도록 하여 포인터 사용에 대한 어려움을 줄이고 있다.
포인터는 어떤 값이 저장되는 위치를 가리키는 메모리의 주소(address)라고 할 수 있다. 정수형 변수 a가 선언된 경우를 살펴보자.

```
int a = 100;
```

변수 이름 a는 변수의 값 100이 저장되어 있는 메모리 위치의 다른 이름이다. 아두이노에서 메모리 위치, 즉 번지는 2바이트로 표현되는 숫자로 기억하고 사용하는데 어려움이 있으므로 사용이 간편한 변수 이름을 대신 사용한다. 실제 100이 저장되어 있는 메모리의 주소는 번지 연산자(&)를 사용하여 알아낼 수 있다. 코드 9는 일반 변수와 포인터 변수가 차지하는 메모리의 크기 및 저장되는 내용의 차이를 보여준다. 포인터 변수는 메모리 번지를 저장하기 위해 사용되며 Arduino UNO의 경우 2바이트 크기를 가진다.

코드 9

```
void setup() {
  Serial.begin(9600);                    // 직렬 포트 초기화
}

void loop()
{
  char ch;
  char *pch;

  ch = 'A';
  pch = &ch;

  // char 형 변수가 필요로 하는 메모리 크기
  Serial.println("Size of character type : " + String(sizeof(ch)));
  // char 포인터 형 변수가 필요로 하는 메모리 크기
  Serial.println("Size of character pointer type : " + String(sizeof(pch)));

  // char 형 변수에 저장되는 내용
  Serial.println("Value of character type variable : " + String(ch));
  // char 포인터 형 변수에 저장되는 내용 (메모리 주소가 저장됨)
  Serial.println("Value of character pointer type : " + String((int)pch, HEX));
```

```
    while(true);
}
```

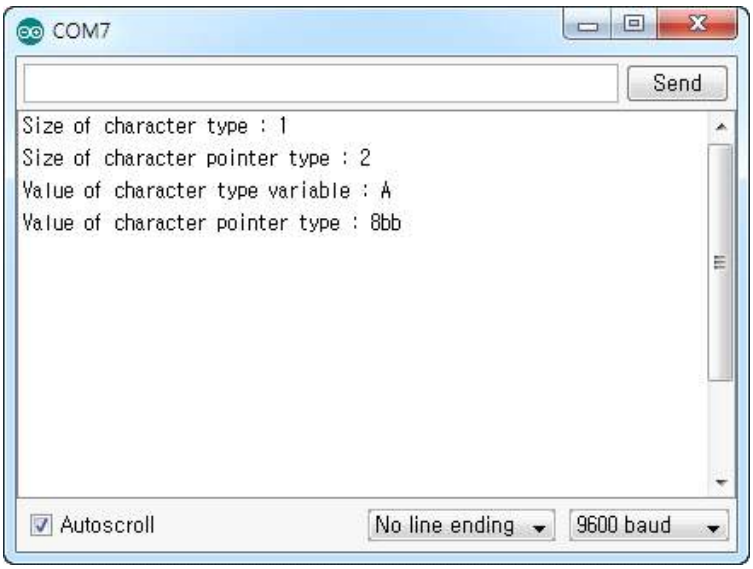


그림 22. 코드 9 실행 결과

코드 9에서 볼 수 있듯이 포인터 변수(char *pch)는 내용으로 주소를 담고 있는 변수로 주소가 2바이트로 표시되므로 메모리 역시 2바이트를 필요로 하며, 내용은 메모리 주소를 가지고 있다. 모든 포인터 변수들은 그 크기가 동일하다. 포인터 변수에 자료형을 명시하는 까닭은 지정한 번지에 들어있는 내용을 해석하기 위함이다. 예를 들어, char 형 변수이면 지정한 번지부터 1바이트 내용을 읽어 문자로 해석하면 되고, int 형 변수이면 지정한 번지부터 2바이트 내용을 읽어 정수로 해석하면 된다.

일반 변수(char ch)는 내용으로 해당 유형의 데이터를 담고 있으며 크기는 자료형에 따라 미리 결정된 값을 가진다. 변수 이름으로부터 변수가 저장된 메모리 주소를 얻어오기 위해 '&'를 사용한 것과 반대로 주소에 저장된 값을 얻어오기 위해서는 '*'를 사용한다.

함수의 인자로 포인터가 전달되는 경우를 살펴보자. 코드 10은 일차원 배열로 선언된 배열을 함수로 전달하는 방법으로 실제로는 포인터가 전달되어 메모리의 내용을 바꾸는 것을 볼 수 있다. 일반적으로 함수를 통해 전달된 매개변수는 값으로 전달되어 함수에서 귀환하면 그 값은 사라지는 것과 대조된다.

코드 10

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

void function1(int a[10])
{ // 매개변수가 배열 형태인 경우
  a[5] = 0;
}

void function2(int *a)
{ // 매개변수가 포인터 형태인 경우
  a[6] = 0;
}

void print_value(int *a)
{
  int i;
  for(i = 0; i < 10; i++)
    Serial.print(String(a[i]) + " ");
  Serial.println();
}

void loop()
{
  int a[10];
  int i;

  for(i = 0; i < 10; i++){
    a[i] = i;
  }

  print_value(a);
  function1(a);   // 배열 이름과 포인터는 동일하므로 function2와 차이는 없다.
  print_value(a); // 함수 내에서 변경한 값이 반영된다.
  function2(a);
```

```
    print_value(a); // 함수 내에서 변경한 값이 반영된다.

    while(true);
}
```

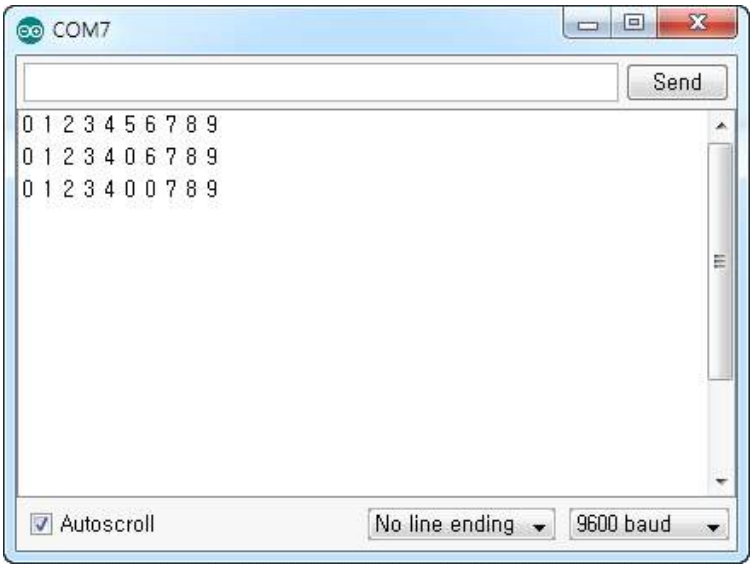


그림 23. 코드 10 실행 결과

코드 11은 배열이 아닌 변수를 일반 변수와 포인터 변수를 통해 전달하는 예로 포인터 변수로 전달한 경우 함수 내에서 변경한 값이 귀환한 이후에도 그대로 반영되는 것을 볼 수 있다. 함수의 경우 반환값은 return을 통해 하나만 가능하지만 포인터 변수를 매개변수로 사용하는 경우에는 return을 통해 값을 반환하는 것과 유사한 효과를 낼 수 있다.

코드 11

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

void function1(int a)
{ // a는 지역변수로 함수가 끝나면 사라진다.
```

```
  a = 0;
}

void function2(int *pa)
{ // pa는 지역변수이나 메모리에 직접 값을 기록하므로 함수가 끝나도 사라지지 않는다.
  *pa = 0;
}

void loop()
{
  int a = 3;

  Serial.println(a);

  function1(a);          // 일반 변수를 이용한 함수 호출
  Serial.println(a);
  function2(&a);         // 포인터 변수를 이용한 함수 호출
  Serial.println(a);

  while(true);
}
```

그림 24. 코드 11 실행 결과

6. 클래스

C는 함수 중심의 절차적인 프로그래밍 언어이다. 이에 비해 C++은 객체 중심의 객체지향 프로그래밍 언어이다. C++ 언어에 대한 가장 큰 오해 중 하나는 C++을 C의 확장판으로 생각하는 점이다. 크게 틀린 말은 아니지만 정확하게는 '객체지향 프로그래밍에서 핵심적인 개념인 객체를 표현하기 위해 C 언어의 문법을 이용하는 언어'라고 이야기 하는 것이 옳다. 즉, C와 C++은 전혀 다른 언어로 단지 비슷한 문법과 키워드를 사용하는 언어일 뿐이다. 동일한 문법을 사용하는 서로 다른 언어가 과연 존재할까?

찌아찌아(Ciacia)말은 인도네시아의 소수 민족 중 하나인 찌아찌아족이 사용하는 고유 언어이지만 찌아찌아족에게는 자신들의 언어를 표현할 문자가 없다. 이에 찌아찌아족은 인도네시아어, 영어, 아랍어 등 다른 언어의 문자를 사용해 찌아찌아말을 표현하고자 하였으나 한계를 느끼고 2009년부터 배우기 쉽고 별도의 발음 기호 없이 소리 나는 데로 표기할 수 있는 한글을 공식 표기 문자로 채택하여 사용하고 있다.

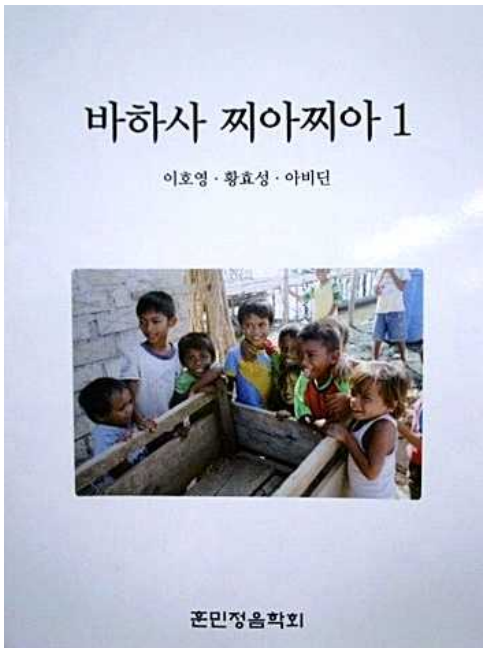


그림 25. 찌아찌아어

찌아찌아족의 예에서 보듯이 찌아찌아말과 우리말은 전혀 다르다. 하지만 표기 문자로 한글을 사용한다는 점에서 동일하다. C와 C++ 역시 마찬가지다. C 언어는 객체지향 개념이 등장하기 전 가장 많이 사용되는 프로그래밍 언어였으므로 객체지향을 표현하기 위해 C 언어가 선택된

것은 자연스러운 일이었다고 할 수 있다. 그렇다면 객체지향이란 무엇일까? 흔히 프로그래밍 관련 수업에서 C++의 문법 몇 가지를 배우고 C++을 배웠다고 이야기하지만 C++의 핵심은 그 개념에 있지 문법에 있지 않다. 객체지향의 개념을 한 마디로 설명하기는 어려우며 많은 책들에서 다양한 방식으로 객체지향의 개념을 설명하고 있다. 하지만 객체지향의 개념에서 가장 중요한 것은 객체(object)인 것은 당연하며 C++에서 객체는 클래스(class)로 표현된다. 객체지향에 대한 개념을 설명하기는 쉬운 일이 아니므로 이 절에서는 C 스타일과 C++ 스타일로 작성한 코드를 비교해 보고자 한다. 코드 12는 1부터 10까지의 정수에 대하여 소수인지를 검사하는 프로그램을 C 언어로 작성한 예이며 코드 13은 동일한 문제를 객체지향 기법으로 작성한 예이다.

코드 12

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

boolean is_prime(int no)
{
  int i;

  for(i = 2; i < no; i++){
    if(no % i == 0) break;
  }

  if(i == no)
    return true;
  else
    return false;
}

void loop()
{
  int no[10], i;
```

```
  for(i = 0; i < 10; i++){
    no[i] = i + 1;
  }

  for(i = 0; i < 10; i++){
    if(is_prime(no[i])){
      Serial.println(String(no[i]) + " is a prime number.");
    }
    else{
      Serial.println(String(no[i]) + " is NOT a prime number.");
    }
  }

  while(true);
}
```

코드 13

```
class NUM
{
private:
  int no;

public:
  void set_number(int _no) { no = _no; }
  int get_number(void) { return no; }
  bool is_prime(void)
  {
    int i;

    for(i = 2; i < no; i++){
      if(no % i == 0) break;
    }

    if(i == no)
```

```
        return true;
      else
        return false;
  }
};

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

void loop()
{
  NUM no[10];
  int i;

  for(i = 0; i < 10; i++){
    no[i].set_number(i + 1);
  }

  for(i = 0; i < 10; i++){
    if(no[i].is_prime()){
      Serial.println(String(no[i].get_number()) + " is a prime number.");
    }
    else{
      Serial.println(String(no[i].get_number()) + " is NOT a prime number.");
    }
  }

  while(true);
}
```

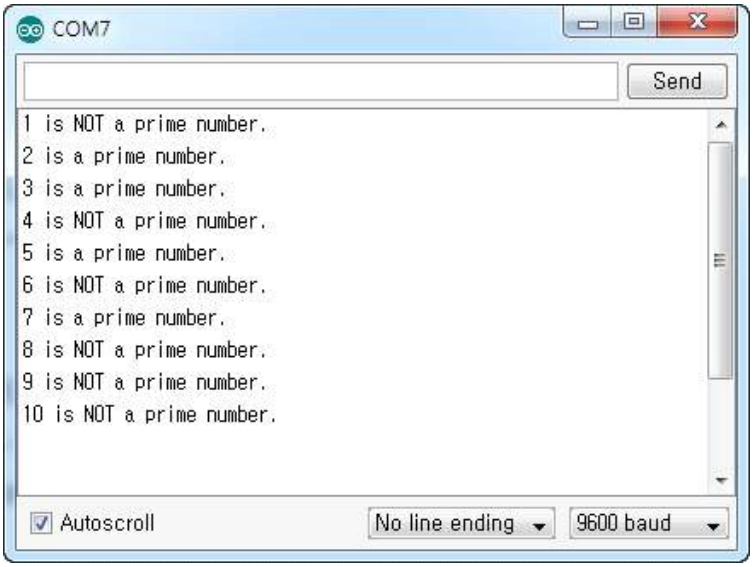


그림 26. 코드 12, 13 실행 결과

코드 12와 13을 비교해보면 크게 차이점이 없는 것처럼 보일 수 있으나 그 개념 자체에서는 큰 차이가 있다. 클래스 NUM은 int나 char와 같은 데이터 유형이라고도 볼 수 있는 것은 그림 13에 나타난 바와 같이 int나 char형 배열을 정의하는 것과 동일한 방식으로 객체를 정의하고 있기 때문이다.

```
NUM no[10];     // 객체 배열의 정의
int a[10];      // 정수형 배열의 정의
```

눈에 띄게 다른 점은 함수가 클래스 정의 내에 포함된 점이다. 객체는 종종 독자적으로 생존 가능한 생명체와 비교하여 설명한다. 고양이의 예를 들어보자. 고양이는 나이와 색깔 등의 특징을 가지고 생존을 위해 이동하고 먹이를 사냥하는 등 외부와 상호작용한다. 이 중 나이와 색깔 등의 특징은 변수로 클래스 내에 포함되고 '이동'이나 '사냥' 등은 함수로 클래스 내에 포함된다. 고양이를 집에 놓아두면 혼자 돌아다니면서 먹이를 먹고 집안의 환경과 상호작용하면서 살아간다. 이처럼 클래스로 만들어지는 객체는 다른 객체들과 상호작용하면서 프로그램이 실행되는 동안 존재한다. 코드 13의 숫자 클래스 NUM 역시 정수값을 특징으로 가지면서 외부에서의 질문, 소수인지 아닌지를 묻는 질문에 대답하면서 프로그램이 종료될 때까지 존재한다.

간단한 몇 마디 말로 객체지향을 설명할 수는 없으며 짧은 몇 페이지의 글로 객체지향을 이해할 수는 없다. 다만, 객체지향 언어인 C++은 C와는 전혀 다른 언어라는 점, C++과 C의 차이는 문법이 아닌 개념에서 찾아야 한다는 점을 이해하고 아두이노에서 외부 라이브러리를 사용함에 있어 클래스 구조를 이해할 수 있도록 객체의 표현 방법에 익숙해져야 할 것이다.