1주차 학습자료

코드가 어떻게 작성되고 결과가 나오는가? (컴파일과 실행)

소스 코드(Source Code)

```
#include <stdio.h>

int main() {
    printf("Hello, World!");
}
```

소스 코드 혹은 **원시 코드**는 컴퓨터 프로그램을 사람이 읽을 수 있는 프로그래밍 언어로 작성한 텍스트 파일을 일컫습니다. 사람이 읽을 수 있는 평문으로 작성되며, 프로그래밍을 할 때 맨 처음 작성한 소스 코드를 재료로 컴파일하여 실행 파일을 만들게 됩니다.

소스 코드 파일의 확장자는 언어에 따라 특정 규칙을 가지고 있는데, 그 예시로 C는 *.c*, 파이썬(Python)은 *.py*를 사용하고, 자바(Java)는 *.java*를 사용합니다.

전처리(Preprocessing)

C언어 문법에선 외부의 코드를 현재 소스코드에서 사용하고 싶을 때, `#include` 라는 문법으로 외부 라이브러리 코드를 현재 소스 코드에 포함시킬 수 있도록 할 수 있습니다. 위 소스 코드의 예시와 같이, `#include <stdio.h>` 는 *stdio.h* 라는 라이브러리 파일을 현재 소스 코드에서 사용할 수 있도록 하는 명령어 입니다. 이 때 컴파일 과정에서 라이브러리 파일의 소스 코드를 작성한 소스 코드에 포함 시키는 작업이 필요한데, 이 작업을 수행해주는 프로그램을 **전처리기(Preprocessor)**라고 합니다. C언어에서는 전처리기가 전처리를 할 수 있도록 전처리기 구문을 따로 지원하고 있습니다. 이 전처리기 구문의 특징은 `#` 로 시작한다는 점입니다.

```
#include <stdio.h> // 외부 라이브러리를 포함시킴

#define PI 3.14 // 매크로

#ifdef // 조건부 컴파일
#ifndef
```

컴파일(Compile)

컴퓨터는 인간이 프로그래밍 언어로 작성한 소스 코드가 아닌, 0과 1로 이루어진 기계어만 인식합니다. 그리하여 프로그래밍 언어에 대한 개념이 정착되지 않은 과거엔 **천공카드**를 이용하여 특정 위치에 구멍을 뚫어 기계어를 작성하여 프로그램을 작성하였습니다.

하지만 이러한 방식은 인간에게 매우 복잡하고 컴퓨터 친화적이었기 때문에, 기계어와 1:1 대응이 되는 언어인 **어셈블리어(assembly language)**라고 하는 저급 언어(Low-level Language)를 만들고, 그 위로 C, 파이썬, 자바와 같은 여러 고급 언어(High-level Language)들이 만들어지게 되었습니다.

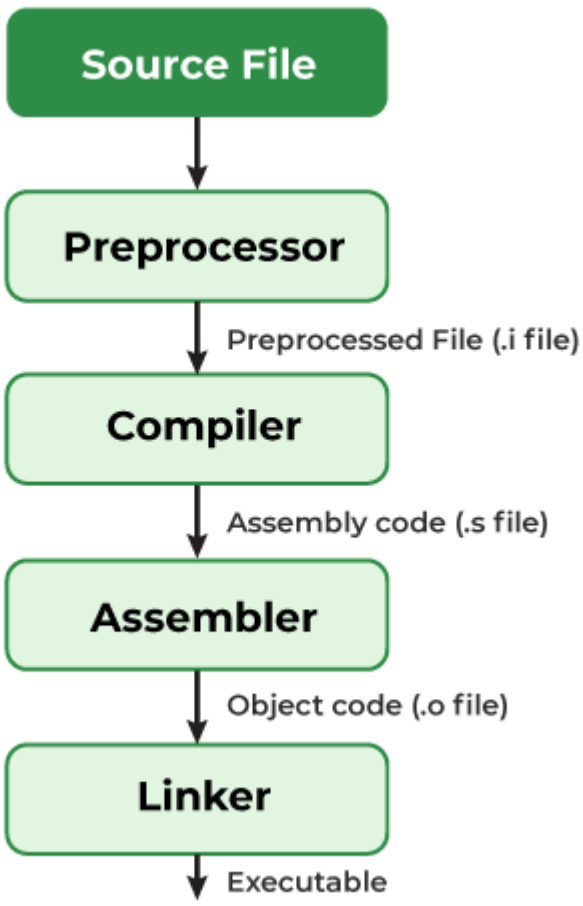
이 때 고급언어로 소스 코드를 작성하게 되면, 사람은 잘 읽을 수 있지만 컴퓨터는 해당 문법을 전혀 해석하지 못하기 때문에 고급 언어를 어셈블리어로 변환하는 과정을 **컴파일**이라고 합니다. 그리고 컴파일을 해주는 도구를 **컴파일러(Compiler)**라고 합니다.

링킹(Linking)

컴파일을 한 뒤에, **어셈블러(Assembelr)**를 통해 어셈블리 코드를 **목적 파일(Object file)**로 변환하고, 해당 목적 파일을 **실행 가능한 파일(Executable file)**로 변환해줘야 최종적으로 실행 파일이 만들어집니다. 이 때 목적 파일을 실행 가능한 파일로 변환해주는 프로그램을 **링커(Linker)**라고 합니다.

작성한 프로그램은 간혹 다른 프로그램이나 다른 프로그램의 라이브러리를 사용하는 경우가 많습니다. 이 때 외부의 라이브러리와 작성한 프로그램을 연결하여 실행 가능한 파일을 만드는 과정을 **링킹(Linking)**이라고 합니다.

위의 과정을 요약하면 아래의 그림과 같습니다.



(출처 : https://www.geeksforgeeks.org/cc-preprocessors/)

디버깅(Debugging)

만약 소스 코드에 문제가 생겨, 컴퓨터가 소스 코드의 의미를 정확하게 파악하지 못하게 된다면 컴파일 과정에서 정확하게 기계어로 번역하지 못했다는 의미가 됩니다.

기계어로 번역을 못했다는 말은 컴퓨터가 요구하는 규칙이나 문법이 틀렸다는 의미와 같습니다..

이러한 오류가 발생한 것을 **버그(bug)**가 발생했다고 하며, 이러한 오류를 수정하는 작업을 **디버그(Debug)**, 혹은 **디버깅(Debugging)** 이라고 합니다.

변수(Variable)

컴퓨터가 일을 하고, 명령어를 처리하려면 다양한 정보를 저장해야 합니다. 다시 말해, 프로그래밍을 하려면 값을 저장해 놓기도 하고, 저장된 값을 가져와서 다시 사용하기도 해야합니다. 여기서 '값을 저장한다'라는 말은 컴퓨터에 있는 하드웨어인 '메모리(memory)'에 저장한다는 의미와 같습니다.

컴퓨터는 모든 데이터를 기계어로 저장합니다. 숫자와 문자도 모두 0과 1로 저장하게 됩니다. 이러한 정보의 값을 저장하려면 정보의 최소 저장 단위인 **비트(bit)**를 사용하게 된다.

이러한 정보를 저장하고 사용하기 위해서 개발자는 **변수**(variable)를 생성하면 됩니다.

```
#include <stdio.h>

int main() {
```

```
    int a;              // a라는 정수형 변수 선언
    a = 5;              // a라는 변수에 5 대입
    printf("%d", a);    // a 변수 출력
}
```

위의 코드를 보면, 개발자는 `int a` 라는 명령어를 사용하여, **정수형(Integer) 변수** a를 선언 하였습니다. 이후 `a = 5` 라는 명령어를 입력하여 a에 5라는 값을 대입하였습니다.

실행 결과는 아래와 같습니다.

```
$ ./ex
5
```

서식 지정자 (Format Specifier)

맨 마지막 줄의 `printf("%d", a)` 를 보자. 기존 `printf("hello world!")` 라는 문자열을 출력하기 위해서는 `printf()` 안에 큰따옴표(")를 사용하여 문자들을 출력하였던 것을 기억할 것입니다. C언어에서 변수를 출력하기 위해서는 **%d** 와 같은 서식 지정자를 사용하여, 해당하는 변수의 값이 화면에 출력되게 됩니다.

```
#include <stdio.h>

int main() {
	int a;
	int b;

	a = 7;
	b = 5;

	printf("a의 값은 %d이며, b의 값은 %d 입니다.\n", a, b);
	printf("a와 b의 합은 %d입니다.\n", a + b);
}
```

(`a+b`와 같이 수식을 사용해도 출력이 가능합니다.)

실행 결과는 아래와 같습니다.

```
$ ./ex
a의 값은 7이며, b의 값은 5 입니다.
a와 b의 합은 12입니다.
```

변수 선언(Variable Declaration)

```
int a;
a = 5;
```

기존에는 이렇게 선언과 초기화를 따로 해줬다면, 아래와 같이 선언과 동시에 초기화 시켜줄 수도 있습니다.

```
int a = 5;
```

다양한 변수를 아래와 같이 동시에 선언도 가능합니다.

```
int a, b, c, d, e;

int a = 3, b, c;     // 이렇게도 사용이 가능합니다.
```

자료형(Data Type)

기존에 설명한 바와 같이, 컴퓨터는 모든 정보를 2진수로 처리한다. 이때 0 또는 1을 저장할 수 있는 공간을 **비트(bit)**라고 합니다. 비트는 컴퓨터가 정보를 저장하는 최소 공간으로, 8개의 비트가 모인 공간을 **바이트(Byte)**라고 합니다.

1비트로 표현이 가능한 숫자는 0과 1밖에 없습니다. 그렇다면 2비트로 표현할 수 있는 2진수 숫자는 00, 01, 10, 11로 10진수의 0,1,2,3과 같습니다. 이를 3비트로 확장하게 된다면, 000,001,010,011,100,101,110,111으로 0~7까지의 숫자들을 저장할 수 있게 됩니다.

이를 확장하게 된다면 n비트로 저장할 수 있는 숫자는 `0 ~ 2^n - 1(n은 비트의 개수)` 의 범위를 가집니다.

예를 들어, 4바이트로 표현할 수 있는 숫자는 `4,294,967,296(=2^32)`개 만큼의 숫자를 표현할 수 있게 됩니다. 그러나, 모든 정수는 양수, 0, 음수를 포함해야 합니다. 따라서 음수까지 범위를 넓히게 된다면, `4,294,967,296 / 2 = 2,147,483,648`로 나누어 `-2,147,483,648~2,147,483,647`의 범위를 가지게 됩니다.

다른 정수형 타입들 및 저장 범위는 아래와 같습니다.

```
#include <stdio.h>

int main() {
    //정수
    char byte_value = 127;         // 1바이트: -128~127
    short small_num = 32767;       // 2바이트: -32,768~32,767
    int standard_num = 2147483647; // 4바이트: -2,147,483,648~2,147,483,647
    long large_num = 2147483647L;  // 4 or 8바이트(시스템에 따라 다름)

    // unsigned 타입(음수 없음)
    unsigned int positive_only = 4294967295U; // 0~4,294,967,295

    printf("char: %d\n", byte_value);
    printf("short: %d\n", small_num);
    printf("int: %d\n", standard_num);
    printf("long: %ld\n", large_num);
    printf("unsigned int: %u\n", positive_only);
}
```

실행 결과는 아래와 같습니다.

```
./ex
char: 127
short: 32767
int: 2147483647
long: 2147483647
unsigned int: 4294967295
```

실수형의 경우는 아래와 같이 사용하며, 서식 지정자는 `%f` 혹은 `%lf` 를 사용하여 출력한다.

```
#include <stdio.h>

int main() {
    // 실수
    float single_precision = 3.14f;       // 4바이트
    double double_precision = 3.14159265; // 8바이트, 더 많은 소숫점 아래 자리의 값까지 저장 가능

    printf("float: %f\n", single_precision);
    printf("double: %.8lf\n", double_precision);  //double은 서식지정자 %lf를 사용합니다.
}
```

```
$ ./ex
float: 3.140000
double: 3.14159265
```

'A'와 같은 문자 뿐 아니라 `!, #, $` 등의 특수 문자도 2진수로 변환이 되지 않습니다. 그렇다면 어떻게 저장할 수 있을까요?

저장하려는 문자에 해당하는 숫자를 각각 지정하고, 메모리에 저장할 때는 그 숫자를 비트 단위로 변환하여 저장하면 해결 될 수 있습니다. 이러한 규칙을 지정하기 위해 만들어진 내용이 **아스키 코드**(ASCII code: American Standard Code for Information Interchange)입니다. (아래가 아스키코드 표 입니다 .)

제어 문자 공백 문자 구두점 숫자 알파벳

10진	16진	문자	10진	16진	문자	10진	16진	문자	10진	16진	문자
0	0x00	NUL	32	0x20	SP	64	0x40	@	96	0x60	`
1	0x01	SOH	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	39	0x27	'	71	0x47	G	103	0x67	g
8	0x08	BS	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	HT	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	LF	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	VT	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	FF	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	CR	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	SO	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	SI	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	DLE	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	60	0x3C	<	92	0x5C	₩	124	0x7C	|
29	0x1D	GS	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	US	63	0x3F	?	95	0x5F	_	127	0x7F	DEL

C에서는 다른 자료형과 구별하기 위하여 문자 변수(character variable)을 `char` 를 통해 선언할 수 있습니다.

문자 변수는 서식 지정자 `%c` 를 통하여 문자 그대로를 출력할 수 있으며, `%d` 를 통하여 저장된 아스키 코드 값을 출력할 수도 있습니다.

코드로 살펴보면 아래와 같습니다.

```
#include <stdio.h>

int main() {
        char letter = 'A';

    printf("문자: %c\n", letter);       // 'A' 출력
```

```
    printf("ASCII 값: %d\n", letter);   // 65 출력
}
```

실행 결과는 아래와 같습니다.

```
$ ./ex
문자: A
ASCII 값: 65
```

연산자(Operator)

우선 연산(operation)이 무엇인지 먼저 고민해보자. 우리는 더하기, 빼기, 곱하기, 나누기 등의 계산을 의미하며, 이 과정에서 사용하는 `+, -, *, /` 등의 기호들은 **연산자(operator)**라고 합니다.

C언어에서는 다양한 연산자를 제공하며, 그 중 많이 사용하는 몇 가지를 알아보고자 합니다.

대입 연산자

`a = 3` 이라고 하면, 수학적으로는 'a는 3과 같다.' 라는 의미를 가지지만, 컴퓨터에서는 'a에 3을 할당한다.' 를 의미하게 됩니다.

대입 연산자는 다음과 같은 문법을 갖습니다.

```
#include <stdio.h>

int main() {
    int a = 3;  // 상수를 변수에 대입
    a = b;      // 변수에 다른 변수를 대입
    a = 3 * b;  // 수식을 변수에 대입
}
```

산술 연산자

산술 연산자는 수학적 계산을 수행하는 연산자로, C 언어에서 기본적인 사칙연산과 나머지 연산을 수행할 수 있습니다.

연산자	설명	예제	결과
`+`	덧셈	`5 + 3`	`8`
`-`	뺄셈	`5 - 3`	`2`
`*`	곱셈	`5 * 3`	`15`
`/`	나눗셈	`5 / 3`	`1` (정수 나눗셈)
`%`	나머지	`5 % 3`	`2`

주의:

- `/` 연산자는 **정수형끼리 연산하면 정수형 결과(몫)**가 나옵니다. (`5 / 3`은 `1`)
- `%` 연산자는 **정수형에만 사용 가능**합니다. (`5.0 % 3`은 오류 발생)

예제 코드는 아래와 같습니다.

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;

    // 산술 연산자

    printf("덧셈: %d + %d = %d\n", a, b, a + b);     // 13
    printf("뺄셈: %d - %d = %d\n", a, b, a - b);     // 7
    printf("곱셈: %d * %d = %d\n", a, b, a * b);     // 30
```

```
    printf("나눗셈: %d / %d = %d\n", a, b, a / b);    // 3 (정수 나눗셈은 몫만)
    printf("나머지: %d %% %d = %d\n", a, b, a % b);   // 1
}
```

실행 결과는 아래와 같습니다.

```
$ ./ex
덧셈: 10 + 3 = 13
뺄셈: 10 - 3 = 7
곱셈: 10 * 3 = 30
나눗셈: 10 / 3 = 3
나머지: 10 % 3 = 1
```

복합 대입 연산자

산술 연산자를 이용한 대입 연산을 단축할 수 있습니다.

연산자	의미	예제	동일한 표현
`+=`	덧셈 후 대입	`a += 5;`	`a = a + 5;`
`-=`	뺄셈 후 대입	`a -= 3;`	`a = a - 3;`
`*=`	곱셈 후 대입	`a *= 2;`	`a = a * 2;`
`/=`	나눗셈 후 대입	`a /= 4;`	`a = a / 4;`
`%=`	나머지 후 대입	`a %= 2;`	`a = a % 2;`

증감 연산자 (`++` , `-`)

변수 값을 1씩 증가하거나 감소시킬 때 사용합니다.

연산자	설명	예제
`++a`	전위 증가	`int a = 5; int b = ++a;` (`a=6, b=6`)
`a++`	후위 증가	`int a = 5; int b = a++;` (`a=6, b=5`)
`--a`	전위 감소	`int a = 5; int b = --a;` (`a=4, b=4`)
`a--`	후위 감소	`int a = 5; int b = a--;` (`a=4, b=5`)

전위(`++a`)와 후위(`a++`)의 차이는 아래 코드로 확인해봅시다.

```
#include <stdio.h>

int main() {
    int a = 5;

    // 증감 연산자

    printf("원래 값: %d\n", a);         // 5
    printf("전위 증가(++a): %d\n", ++a); // 6 (먼저 증가 후 사용)
    printf("후위 증가(a++): %d\n", a++); // 6 (사용 후 증가)
    printf("증가 후: %d\n", a);         // 7

    printf("전위 감소(--a): %d\n", --a); // 6 (먼저 감소 후 사용)
    printf("후위 감소(a--): %d\n", a--); // 6 (사용 후 감소)
    printf("감소 후: %d\n", a);         // 5

    return 0;
}
```

실행 결과는 아래와 같습니다.

```
$ ./test3
원래 값: 5
전위 증가(++a): 6
후위 증가(a++): 6
증가 후: 7
전위 감소(--a): 6
후위 감소(a--): 6
감소 후: 5
```

관계 연산자

관계 연산자는 두 값을 비교하고, 참(`1`) 또는 거짓(`0`)을 반환하는 연산자입니다.

관계 연산자의 종류

연산자	설명	예제	결과
`==`	두 값이 같은지 비교	`5 == 3`	`0` (거짓)
`!=`	두 값이 다른지 비교	`5 != 3`	`1` (참)
`>`	왼쪽 값이 더 큰지 비교	`5 > 3`	`1` (참)
`<`	왼쪽 값이 더 작은지 비교	`5 < 3`	`0` (거짓)
`>=`	왼쪽 값이 크거나 같은지 비교	`5 >= 5`	`1` (참)
`<=`	왼쪽 값이 작거나 같은지 비교	`5 <= 3`	`0` (거짓)

코드로 보면 다음과 같습니다.

```
#include <stdio.h>

int main() {
    int a = 5, b = 3;

    printf("a == b: %d\n", a == b);
    printf("a != b: %d\n", a != b);
    printf("a > b: %d\n", a > b);
    printf("a < b: %d\n", a < b);
    printf("a >= b: %d\n", a >= b);
    printf("a <= b: %d\n", a <= b);

    return 0;
}
```

실행 결과는 아래와 같습니다.

```
$ ./ex
a == b: 0
a != b: 1
a > b: 1
a < b: 0
a >= b: 1
a <= b: 0
```

논리 연산자

논리 연산자는 `참(1)`과 `거짓(0)`을 조합하여 복합적인 조건을 만들 때 사용합니다.

논리 연산자의 종류

연산자	설명	예제	결과
`&&`	논리 AND (둘 다 참이면 참)	`(5 > 3) && (3 > 2)`	`1` (참)
`||`	논리 OR (둘 중 하나만 참이면 참)	`(20 > 15) || (5 > 15)`	`1` (참)
`!`	논리 NOT (참이면 거짓, 거짓이면 참)	`!(5 > 3)`	`0` (거짓)

예제 코드로 보면 아래와 같습니다.

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;

    printf("(x > 5) && (y > 15): %d\n", (x > 5) && (y > 15));
    printf("(x > 15) || (y > 15): %d\n", (x > 15) || (y > 15));
    printf("!(x > 5): %d\n", !(x > 5));

    return 0;
}
```

실행 결과는 아래와 같습니다.

```
$ ./ex
(x > 5) && (y > 15): 1
(x > 15) || (y > 15): 1
!(x > 5): 0
```