컴퓨터가 음수를 표현하는 방법 (2의 보수)

안녕하세요 여러분! 지난 강좌에서 변수를 이용해서 여러가지 연산을 수행하는 방법에 대해 다루었습니다. 그런데 C 언어에서 아무런 제약 없이 연산을 수행할 수 있는 것은 아닙니다. 왜냐하면 변수마다 각각의 타입에 따라서 **보관할 수 있는 데이터의 크기** 가 정해져 있기 때문이죠.

예를 들어서 `int` 의 경우 -2147483648 부터 2147483647 까지의 정수 데이터를 보관할 수 있습니다.

그렇다면 아마 여러분들은 아래와 같은 질문을 하실 수 있습니다.

> 만약에 변수의 데이터가 주어진 범위를 넘어간다면 어떻게 되나요?

한 번 직접 코드를 작성해봅시다.

```c
#include <stdio.h>

int main() {
  int a = 2147483647;
  printf("a : %d \n", a);

  a++;
  printf("a : %d \n", a);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
a : 2147483647
a : -2147483648
```

와 같이 나옵니다.

```c
int a = 2147483647;
printf("a : %d \n", a);
```

먼저 위와 같이 `a` 라는 변수를 정의한 뒤에 `int` 가 표현할 수 있는 최대값인 2147483647 를 대입하였습니다. 해당 문장은 문제가 없으며 `printf` 로도 2147483647 가 잘 출력되었습니다.

```
a++;
printf("a : %d \n", a);
```

반면에 a++ 을 해서 int 가 표현할 수 있는 최대값을 넘어가버렸습니다. 그런데 놀랍게도 전혀 예상하지 못한 값이 출력되었습니다. 바로 **-2147483648** 이 나온 것이죠. 어떻게 양수에서 1 을 더했는데 음수가 나올 수 있을까요?

이에 대한 대답을 하기 위해선 먼저 컴퓨터에서 어떻게 음수를 표현하는지 알아야 합니다.

음수 표현 아이디어

여러분이 CPU 개발자라면 컴퓨터 상에서 정수 음수를 어떤식으로 표현하도록 만들었을까요? 가장 간단히 생각해보자면 우리가 부호를 통해서 음수 인지 양수 인지 나타내니까, 비슷한 방법으로 부호를 나타내기 위해서 1 비트를 사용하는 것입니다. (예를 들어서 0 이면 양수, 1 이면 음수) 그리고 나머지 부분을 실제 정수 데이터로 사용하면 되겠죠.

예를 들어서 가장 왼쪽 비트를 부호 비트라고 생각하자면 (참고로 아래 표현하는 수들은 모두 이진법으로 작성한 것입니다.)

```
0111
```

은 7 이 될 것이고

```
1111
```

은 맨 왼쪽 부호 비트가 1 이므로 -7 을 나타내게 됩니다. 꽤나 직관적인 방식이기는 하지만 여러가지 문제점이 있습니다. 첫 번째로 0 을 나타내는 방식이 두 개라는 점입니다. 즉

```
0000
```

도 0 이고

```
1000
```

역시 0 이지요. (-0 은 0 이니까) 뭐 *0* 이 두 개일 수 도 있지 라고 생각하실 수 있겠지만 사실 이는 매우 큰 문제 입니다. 왜냐하면 컴퓨터 상에서 어떠한 데이터가 0 인지 아닌지 비교하는 일을 굉장히 많이 하거든요.

0 을 표현하는 방법이 두 가지라면, 어떠한 데이터가 0 인지 확인하기 위해서 +0 인지 -0 인지 두 번이나 확인해야 하게 됩니다. 따라서 **이상한 데이터 표현법 덕분에 쓸데없이 컴퓨터 자원을 낭비하게 됩니다.**

또 다른 문제로는, 양수의 음수의 덧셈을 수행할 때 부호를 고려해서 수행해야 한다는 점입니다. 예를 들어서 `0001` 과 `0101` 을 더한다면 그냥 `0110` 이 되겠지만 `0001` 과 `1001` 을 더할 때에는 `1001` 이 사실은 `-1` 이므로 뺄셈을 수행해야 하죠. 따라서 덧셈 알고리즘이 좀 더 복잡해집니다.

물론 부호 비트를 도입해서 음수와 양수를 구분하는 아이디어 자체는 나쁜 생각은 아닙니다. 여기서는 `int` 와 같은 정수 데이터만 다루지만 `double` 이나 `float` 처럼 소수인 데이터를 다루는 방식에서는 (이를 부동 소수점 표현 이라고 하는데, 나중 강좌에서 자세히 알아봅시다.) 부호 비트를 도입하여서 음수인지 양수인지를 표현하고 있습니다.[3)]

하지만 적어도 정수를 표현하는 방식에서는 부호 비트를 사용하는 방식은 문제점이 있습니다.

2의 보수(2's complement) 표현법

그렇다면 다른 방법을 생각해봅시다. 만약에 어떤 x 와 해당 수의 음수 표현인 $-x$ 를 더하면 당연히도 0 이 나와야 합니다. 예를 들어서 7 을 이진수로 나타내면

```
0111
```

이 되는데 여기에 더해서 `0000` 이 되는 이진수가 있을까요? 이 때 덧셈 시에 컴퓨터가 4 비트만 기억한다고 가정합시다.

그렇다면 -7 의 이진수 표현으로 가장 적당한 수는 바로 `1001` 이 될 것입니다. 왜냐하면 `0111` 과 `1001` 을 더하면 `10000` 이 되는데, CPU 가 4 비트만 기억하므로 맨 앞에 1 은 버려져서 그냥 `0000` 이 되기 때문이지요.[4)]

이렇게 가장 덧셈을 고려하였을 때 가장 자연스러운 방법으로 음수를 표현하는 방식을 바로 2 의 보수 표현이라고 합니다. **2 의 보수 표현 체계 하에서 어떤 수의 부호를 바꾸려면 먼저 비트를 반전시킨 뒤에 1 을 더하면 됩니다.**

예를 들어서 -7 을 나타내기 위해서는, 7 의 이진수 표현인 `0111` 의 비트를 모두 반전시키면 `1000` 이 되는데 여기다 1 을 더해서 `1001` 로 표현하면 됩니다. 반대로 -7 에서 7 로 가고 싶다면 `1001` 의 부호를 모두 반전 시킨뒤 (0110) 다시 1 을 더하면 양수인 7 (0111) 이 나오게 됩니다.

이 체계에서 중요한 점은 `0000` 의 2 의 보수는 그대로 `0000` 이 된다는 점입니다. 왜냐하면 `0000` 을 반전하면 `1111` 이 되는데, 다시 1 을 더하면 `0000` 이 되기 때문이죠!

또한 어떤 수가 음수 인지 양수인지 판단하는 방법도 매우 쉽습니다. 그냥 맨 앞 비트가 부호 비트라고 생각하면 됩니다. 예를 들어서 `1101` 의 경우 맨 앞 비트가 1 이기 때문에 음수 입니다. 따라서 이 수가 어떤 값인지 알고싶다면 보수를 구한 뒤에 (1101 –> 0010 –> 0011) - 만 붙여주면 되겠죠. `0011` 이 3 이므로, `1101` 은 경우 -3 이 됩니다.

3) 실제로 부동 소수점 표현법에서는 -0 과 +0 이 있습니다.

4) 참고로 두 개의 자료형을 더했을 때 범위를 벗어나는 비트는 그냥 버려진다고 생각하시면 됩니다. 마치 왼쪽으로 쉬프트 했을 때 맨 왼쪽에 있는 비트들이 버려지는 것 처럼 말이죠.

이와 같이 2 의 보수 표현법을 통해서

- 음수나 양수 사이 덧셈 시에 굳이 부호를 고려하지 않고 덧셈을 수행해도 되고
- 맨 앞 비트를 사용해서 부호를 빠르게 알아낼 수 있다

와 같은 장점 때문에 컴퓨터에서 정수는 2 의 보수 표현법을 사용해서 나타내게 됩니다.

한 가지 재미있는 점은 2 의 보수 표현법에서 음수를 한 개더 표현할 수 있습니다. 왜냐하면 `1000` 의 경우 음수 이지만 변환 시켜도 다시 `1000` 이 나오기 때문이죠 (1000 –> 0111 –> 1000) 실제로 `int` 의 범위를 살펴보면 -2,147,483,648 부터 2,147,483,647 까지 이죠. 음수가 1 개 더 많습니다.

자 그렇다면 이전 코드를 다시 살펴봅시다.

```
int a = 2147483647;
printf("a : %d \n", a);

a++;
printf("a : %d \n", a);
```

처음에 a 에 `int` 최대값을 집어 넣었을 때 아마 a 에는 `0x7FFFFFFF` (이진수로 `0111 1111 ... 1111`) 라는 값이 들어가있을 것입니다. 그런데 여기서 1 을 더하게 되면 어떻게 될까요?

우리는 a 의 현재 값이 `int` 가 보관할 수 있는 최대값이므로 1을 더 증가 시킨다면 오류를 내뿜게하거나 아니면 그냥 현재 값 그대로 유지하게 하고 싶었을 것입니다.

하지만 CPU 는 그냥 `0x7FFFFFFF` 값을 1 증가 시킵니다. 따라서 해당 `a++` 이후에 a 에는 `0x80000000` (이진수로 `1000 0000 ... 0000`) 이 들어가겠죠. 문제는 `0x80000000` 을 2 의 보수 표현법 체계하에서 해석한다면 반전 하면 (`0111 1111 ... 1111`) 이 되고 다시 1 을 더하면 (`1000 0000 ... 0000`) 이 되므로 `-0x80000000`, 즉 -2147483648 이 됩니다.

따라서 위와 같이 양수에 1 을 더했더니 음수가 나와버리는 불상사게 생기게 되죠. 이와 같이 자료형의 최대 범위보다 큰 수를 대입하므로써 발생하는 문제를 **오버플로우(overflow)** 라고 하며, C 언어 차원에서 오버플로우가 발생하였다는 사실을 알려주는 방법은 없기 때문에 **여러분 스스로 항상 사용하는 자료형의 크기를 신경 써줘야만 합니다!**

주의 사항

실제로 1996년에 발사한 Arian 5 로켓은 제어 프로그램 상에서 발생한 오버플로우로 인해서 가속도를 제대로 계산하지 못해서 추락한 사례가 있습니다. 참고로 해당 로켓의 발사 비용은 3억 7천만 달러 (한화로 대략 4000 억원 정도 되죠) 였다고 합니다.

음수가 없는 자료형은 어떨까요?

unsigned int 의 경우 음수가 없고 0 부터 4294967295 까지의 수를 표현할 수 있습니다. unsigned int 가 양수만 표현한다고 해서 int 와 다르게 생겨먹은 것이 아닙니다. unsigned int 역시 int 와 같이 똑같이 32 비트를 차지 합니다.

다만, unsigned int 의 경우 int 였으면 2 의 보수 표현을 통해 음수로 해석될 수를 그냥 양수라고 생각할 뿐이지요.

따라서 unsigned int 에 예를 들어서 -1 을 대입하게 되면, -1 은 0xFFFFFFFF 로 표현되니까,

```c
#include <stdio.h>

int main() {
  unsigned int b = -1;
  printf("b 에 들어있는 값을 unsigned int 로 해석했을 때 값 : %u \n", b);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
b 에 들어있는 값을 unsigned int 로 해석했을 때 값 : 4294967295
```

와 같이 나옵니다. 참고로 printf 에서 %u 는 unsigned 타입으로 해석하라는 의미 입니다.

물론 unsigned int 상에서도 오버플로우가 발생하지 않으라는 법이라는 없습니다. 예를 들어서 b 에 최대값을 대입한 뒤에 1 을 추가한다면;

```c
#include <stdio.h>

int main() {
  unsigned int b = 4294967295;
  printf("b : %u \n", b);

  b++;
  printf("b : %u \n", b);

  return 0;
}
```

성공적으로 컴파일 하였다면

실행 결과

```
b : 4294967295
b : 0
```

와 같이 나옵니다. 즉, b 에 0xFFFFFFFF (1111 1111 ... 1111) 이 들어가 있다가 1 증가함으로써 (1 0000 ... 0000) 이 되었는데 앞서 이야기 하였듯이 자료형의 크기를 초과하는 비트는 그대로 버려지므로 그냥 0 (0000 0000 ... 0000) 으로 해석된 것입니다.

즉 unsigned int 역시, 아니 C 언어 상에 모든 자료형은 오버플로우의 위험으로 부터 자유롭지 않습니다.

자 그럼 이것으로 이번 강좌를 마치도록 하겠습니다. C 언어에서 코딩을 할 때에는 언제나 오버플로우 문제에 신경써야 합니다. 또한 아니 *int* 에 분명히 양수만 더했는데 왜 음수가 나왔지*?* 와 같은 상황에 당황하지 않고 대처할 수 있겠죠!

뭘 배웠지?

- 컴퓨터 상에서 정수인 음수를 표현하기 위해서 2 의 보수 표현법을 사용합니다.
- 이에 따라 int 상에서 오버플로우가 발생하였을 때 양수에서 값을 증가시켰더니 음수로 바뀌는 기적을 볼 수 있습니다.
- 항상 오버플로우를 조심합시다.

4

문자 입력 받기

지난번 강좌는 잘 이해 되셨는지요? 이번 강좌에서는 제목에서도 볼 수 있듯이 두 가지 내용을 한꺼번에 배우게 됩니다. 바로, 문자를 키보드로 부터 입력을 받는 것이지요. 문자를 입력 받을 수 있다면, 숫자도 당연히 입력 받을 수 있게 됩니다. 즉, 이번 강좌에서는 문자 형식의 변수와 키보드로부터 입력을 받는 입력에 대해 알아 보도록 하겠습니다.

일단, 컴퓨터에서 문자를 처리하는 방식에 대해 생각해 봅시다. 제가 누누히 말하지만 우리의 컴퓨터는 그다지 똑똑하지 못합니다. 아무리 최신 `Intel CPU` 를 장착해도 컴퓨터는 단지 0 과 1 만을 처리할 뿐이죠.

따라서, 2 와 3 같은 숫자도 처리하지 못하는데 어떻게 a, b 가, 나, 韓 과 같은 수 많은 문자를 처리할 수 있겠습니까? 하지만, 방법이 있습니다. 이러한 문자들을 숫자에 대응시키는 것입니다. 그런데, 숫자에 대응시킨다면 컴퓨터가 이 것이 숫자인지, 아니면 문자인지 어떻게 알까요? 물론 알 방법은 없습니다. 단지 이 숫자를 '문자' 형태로 사용하거나 '숫자' 형태로 사용하는 것이지요.

문자를 저장하는 변수는 앞에서 살짝 본 적이 있습니다. 바로 `char` 이지요. `int` 가 `integer` 의 약자였다면 `char` 은 `character` 의 약자 입니다. 변수가 등장하면 어김없이 등장하는 아래의 표를 살펴 봅시다.

Name	Size*	Range*
char	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	1byte	true or false
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

보시는 것과 같이 `char` 은 맨 위에 위치해 있으며 크기는 1 바이트 입니다. 또한, 이를 통해 나타낼

수 있는 숫자의 범위를 알려주고 있는데, 이는 -128 부터 127 까지, 256 가지 입니다.

```c
/* 문자를 저장하는 변수 */
#include <stdio.h>
int main() {
  char a;
  a = 'a';

  printf("a 의 값과 들어 있는 문자는? 값 : %d , 문자 : %c \n", a, a);
  return 0;
}
```

위 소스를 성공적으로 컴파일 했다면

실행 결과

```
a 의 값과 들어 있는 문자는? 값 : 97 , 문자 : a
```

위와 같이 나옵니다. 일단, 소스를 분석해 보겠습니다.

```c
char a;
```

이 부분은 char 형 변수를 선언하는 부분입니다. 기억이 안나시는 분들은 3강을 참조하세요.

```c
a = 'a';
```

이 부분은 a 라는 변수에 문자 a 를 대입하고 있습니다. 이 때, 모든 문자들은 모두 작은 따옴표로 묶어 주어야 합니다. 만약 작은 따옴표로 묶지 않고 그냥 썼다면

```c
a = a;
```

C 컴파일러는 이 a 가 변수 a 라고 착각하여 a 라는 변수의 값을 a 라는 변수에 대입하는 문장으로 인식하게 되죠. 따라서 a 에는 아무런 값이 들어있지 않은 쓰레기 값(NULL) 이 되어 나중에 a 라는 문자를 출력해 보았을 때, 이상한 값이 나오게 됩니다.

문자를 대입하는 것도 숫자를 대입하는 것과 동일합니다. 대입 연산자를 이용하면 되죠.

```c
printf("a 의 값과 들어 있는 문자는? 값 : %d , 문자 : %c \n", a, a);
```

마지막으로, 위 `printf` 문에 대해 보도록 하겠습니다. 앞에서 말했듯이 컴퓨터는 a 가 문자라는 것 자체를 모른다고 했습니다. 단지 우리가 a 를 문자로 보느냐 아니면 숫자를 보느냐에 따라 달라진다고 했는데, 이 말 뜻을 위 `printf` 문을 보면 알 수 있습니다.

일단, %d 는 a 의 값을 숫자 (정수인 10 진수) 라고 출력하라는 뜻입니다. 그 옆의 %c 는 아마 예상했겠지만 a 의 값을 문자로 출력하라는 뜻이지요. 따라서, %c 에는 a 에 저장되어 있던 문자 'a' 가 출력되게 됩니다.

그렇다면 %d 에는 무엇이 출력되었을까요? 앞에서 말했지만 컴퓨터는 문자와 숫자를 일대일 대응시켜서 생각한다고 했습니다. 따라서, %d 에 출력되는 숫자가 바로 a 에 대응되는 숫자를 가리킵니다.

이 때, 각 문자 마다 대응되는 숫자를 아무렇게나 하는 것이 아니라 일정하게 정해져 있는데 현재 우리가 쓰고 있는 컴퓨터에서는 다음과 같이 정의되어 있습니다.

10진수	ASCII	10진수	ASCII	10진수	ASCII	10진수	ASCII
0	NULL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	`	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	`	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	SC2	50	2	82	R	114	r
19	SC3	51	3	83	S	115	s
20	SC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	₩	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

위 표는 미국 표준 학회(ASA) 에서 정한 **아스키**(ASCII, *American Standard Code for Information Interchange*) 코드로 8 비트 데이타를 이용하여 여러 문자에 번호를 붙인 것 입니다. 아까, a 의 숫자 값을 출력하였을 때 97 이 나왔는데 위 표에서 찾아 보면 a 의 값이 97 임을 볼 수 있습니다. 이 때, 위 표의 내용이 0 부터 127 까지 밖에 없는 이유는 위 표준을 정할 당시 그 당시 7 비트 만으로 충분하다고 생각했기 때문이죠. 하지만 IBM 에서 좀 더 많은 종류의 문자가 필요하게 되자 1 비트를 더 추가 시켜서 확장된 아스키 코드(Extended ASCII Code) 를 만들었습니다.

하지만 위 256 개 가지고는 충분하지 못하죠. 왜냐하면 우리 글만 해도 자모음 24 개로 구성되어 있는데, 한 글자당 최대 초성/중성/종성 을 모두 표현해야 합니다. 또한 더욱 심각한 것은 한자와 같은 표의문자의 경우 수만 개가 넘는 한자 데이터들을 가지고 있어야 하는데 이를 256 개 안에 다 표현한다는 것은 불가능하기 때문이죠. 따라서, 컴퓨터가 전세계에 보급되자 좀 더 많은 종류의 문자를 표현해야 한다는 필요성이 대두되었습니다.

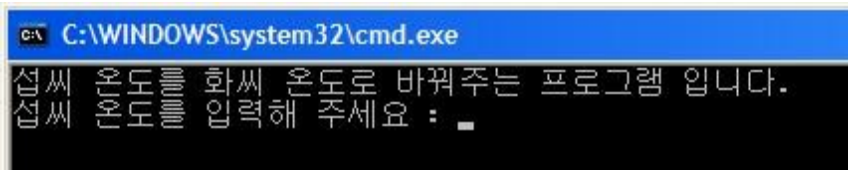
결국에는 **유니 코드(Unicode)** 라는 새로운 형식의 문자 체계를 도입하게 됩니다. 유니코드는 한 문자를 1 에서 4 바이트까지 다양한 길이로 처리합니다. 이는, 기존 아스키 코드의 체계를 유지하면서, 새로운 문자들을 추가하기 위함입니다. 여러분은 아직 유니코드를 직접 다룰 일은 없기 때문에 여기서는 무시하셔도 괜찮습니다.

scanf 의 도입

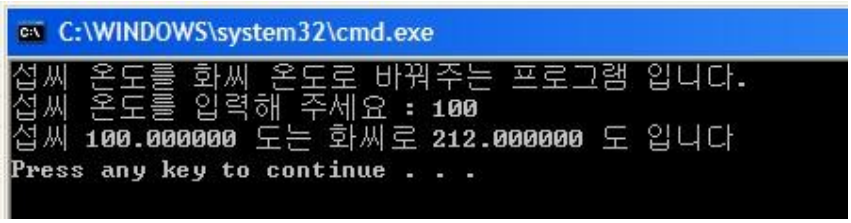
```c
/* 섭씨온도를 화씨로 바꾸기 */
#include <stdio.h>
int main() {
  double celsius;  // 섭씨 온도

  printf("섭씨 온도를 화씨 온도로 바꿔주는 프로그램 입니다. \n");
  printf("섭씨 온도를 입력해 주세요 : ");
  scanf("%lf", &celsius);  // 섭씨 온도를 입력 받는다.

  printf("섭씨 %f 도는 화씨로 %f 도 입니다 \n", celsius, 9 * celsius / 5 + 32);

  return 0;
}
```

위 소스를 성공적으로 컴파일 했다면 아래와 같이 나옵니다. 참고로 scanf_s 를 사용하라며 컴파일 되지 않는다면 여기 에 소개된 방법으로 해당 메세지를 끌 수 있습니다.[1)]

1) scanf_s 를 사용하라는 의미는 scanf 가 입력받는 데이터의 크기를 확인하지 않기 때문에 버퍼 오버플로우 (입력받는 데이터의 크기가 준비된 공간보다 큰 문제) 가 발생할 수 있기 때문입니다. 이 문제에 대해서는 나중에 다루어보도록 하겠습니다.



이 때, 원하는 숫자를 쓴 후 엔터를 누른다면 (예 : 100)



위와 같이 섭씨가 화씨 온도로 변경된 값이 출력됩니다. 와우! 드디어 쓸만한 프로그램을 처음으로 만들어 보게 된 것 같군요. 소스 코드를 찬찬히 살펴 보도록 합시다.

```
double celsius;  // 섭씨 온도
```

일단, celsius 라는 double 형 변수를 선언하였습니다. 변수의 이름을 종전의 a , b 에서 celsius 라고 한 이유는 좀 더 이해하기 편하기 때문이죠. 좋은 소스 코드의 조건은 다른 사람이 이해하기 쉬운 소스 코드 이고, 다른 사람이 이해하기 쉬운 소스코드는 기본적으로 변수 이름을 보고도 변수를 한 눈에 파악하기 쉽게 만드는 것입니다.

```
scanf("%lf", &celsius);  // 섭씨 온도를 입력 받는다.
```

이제, 새로운 것이 등장하였군요. printf 에 이어 등장한 scanf 군. printf 가 화면에 결과를 출력해 주는 함수였다면, scanf 는 화면(키보드) 로 부터 결과를 받아들이는 입력 함수 입니다. 이렇게 흔히 printf 와 scanf 를 가리켜 입출력함수라 하죠. 이 때, scanf 함수는 우리가 어떠한 입력을 하기 전까지 계속 기다립니다. 또한, 입력을 할 때 엔터를 눌러야지만 입력으로 처리됩니다.

scanf 와 printf 는 이름도 비수무리 할 뿐더러, 사용하는 방법도 비슷합니다. printf 에서 각 변수를 출력할 포맷(%d, %f, %c 등) 을 변수마다 다르게 하는 것처럼 scanf 도 각 변수의 타입마다 입력받는 포맷을 달리 해야 합니다.

위 경우 처럼 double 형의 변수를 입력 받으려면 %lf (소문자 LF 이다, if 가 아니다) 로 해야 합니다. 그런데, printf 보다 조금 까다로운 점은 printf 는 double 이나 float 모두 %f 로 출력하지만 이에 경우 float 은 %f 로 무조건 입력 받아야 한다는 점입니다.

마찬가지로 double 형 변수도 무조건 %lf 로만 입력 받아야 합니다. 그 외에도, printf 는 정수형 변수는 모두 %d 로 출력 가능했던 반면에 scanf 는 각 자료형 마다 포맷이 다 정해져 있습니다. 아래 예제에서 잠시 scanf 의 포맷 들에 대해 정리해 보도록 하겠습니다

```
printf("섭씨 %f 도는 화씨로 %f 도 입니다 \n", celsius, 9 * celsius / 5 + 32);
```

마지막으로 위 프로그램의 중요한 부분을 살펴보자. 바로 이 부분에서 섭씨와 화씨의 환산 작업이 이루어 진다. 참고로, 화씨와 섭씨의 변환 공식은 아래와 같습니다.

$$C \cdot \frac{9}{5} + 32 = F$$

따라서, 이 공식을 그대로 C 언어 수식을 바꾼 것이 `9 * celsius / 5 + 32` 인 것입니다. 곱셈과 나눗셈의 우선순위가 높으므로 `9 * celsius / 5` 가 먼저 계산 된 후 32 가 더해지므로 위의 식과 일치합니다. 따라서 `printf` 의 두번째 `%f` 부분에는 위 계산된 화씨의 값이 들어가게 됩니다.

```c
/* scanf 총 정리 */
#include <stdio.h>
int main() {
  char ch;  // 문자

  short sh;  // 정수
  int i;
  long lo;

  float fl;  // 실수
  double du;

  printf("char 형 변수 입력 : ");
  scanf("%c", &ch);

  printf("short 형 변수 입력 : ");
  scanf("%hd", &sh);
  printf("int 형 변수 입력 : ");
  scanf("%d", &i);
  printf("long 형 변수 입력 : ");
  scanf("%ld", &lo);

  printf("float 형 변수 입력 : ");
  scanf("%f", &fl);
  printf("double 형 변수 입력 : ");
  scanf("%lf", &du);

  printf("char : %c , short : %d , int : %d ", ch, sh, i);
  printf("long : %ld , float : %f, double : %f \n", lo, fl, du);
  return 0;
}
```

성공적으로 컴파일 후 (경고가 6 개 정도 나올 수 있는데 무시하세요)


```
printf("char 형 변수 입력 : ");
scanf("%c", &ch);
```

일단, 제일 먼저 문자를 입력 받는 부분을 봅시다. 예전에도 이야기 했지만 한글은 2 바이트 이상을 차지하기 때문에 최대 1 바이트를 차지하는 char 형 변수인 ch 에 한글을 치면 오류가 납니다. 이와 같이 허용된 메모리 이상에 데이터를 집어넣어 발생하는 오류를 **버퍼 오버플로우(Buffer Overflow)** 라고 하며 보안 상 매우 취약합니다.[2] 뿐만 아니라 근처의 데이터가 손상됨에 따라 큰 문제가 발생하게 될 수 도 있습니다. 따라서, 여러분들은 버퍼 오버플로우가 일어나지 않게 허용된 데이터 이상을 집어넣는지 안집어 넣는지 검사해야 합니다.

또한 앞으로 우리가 char 형 변수를 선언할 때 에는 이 사람이 문자를 보관하는 변수를 선언하는 구나 라고 생각하도록 합시다. 왜냐하면 보통 정수 데이터를 보관하는 변수로는 int 를 쓰지 char 을 잘 쓰지 않을 뿐더러 char 이름도 character 에서 따왔을 만큼 문자와 무언가 관련이 있기 때문이죠.

```
printf("short 형 변수 입력 : ");
scanf("%hd", &sh);
printf("int 형 변수 입력 : ");
scanf("%d", &i);
printf("long 형 변수 입력 : ");
scanf("%ld", &lo);
```

이 부분은 여러분들이 무난하게 이해하실 수 있으리라 봅니다. 단지 포맷에 %hd, %d, %ld 로 다른 것 뿐이지요. 참고로 short 형이나 long 형은 아직 다루지는 않았지만 int 와 똑같은 계열의 정수형 변수라고 생각하시면 됩니다.

```
printf("float 형 변수 입력 : ");
scanf("%f", &fl);
printf("double 형 변수 입력 : ");
scanf("%lf", &du);
```

마찬가지로 float 형에서는 %f 로, double 형에서는 %lf 로 사용한다는 것을 기억하시기 바랍니다.

2) 버퍼 오버플로우를 이용해서 공격자들이 프로그램의 정상적인 코드가 아니라, 자신들이 원하는 코드가 실행될 수 있도록 조종할 수 있습니다.

이번 강좌는 지난번 강좌보다는 조금 짧습니다. 하지만 이번 강좌를 통해 응용할 수 있는 것들이 무궁무진해졌습니다. 일단, 기본적으로 연습하실 것은 단위 환산 프로그램을 만들어 보세요! 아니면, 금리와 원금을 입력 받아서 일정 개월 후의 상환할 돈이라 든지 등등... 수 많은 프로그램을 만들 수 있습니다. 지금, 이러한 것들을 만들 수 있는 모든 도구들은 준비되어 있습니다. 이제 여러분이 스스로 창작할 세계가 남아 있을 뿐입니다.

생각 해보기

문제 1

앞서 섭씨를 화씨로 바꿀 때 `9 * celsius / 5 + 32` 라고 하였습니다. 만약에 이를 `9 / 5 * celsius + 32` 로 바꾸면 결과가 달라질까요?

뭘 배웠지?

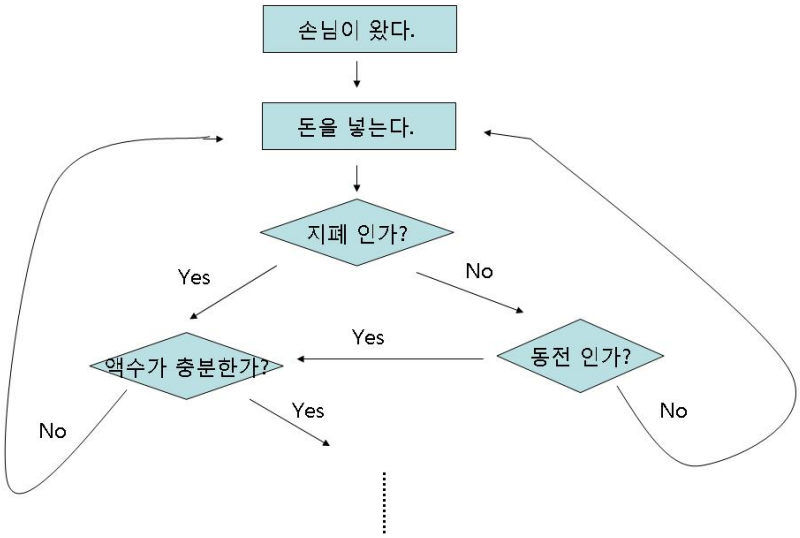
- `char` 은 1 바이트 정수를 저장하는 타입으로, 주로 문자를 저장하는데 사용됩니다.
- 각 문자들은 아스키 테이블이란 표를 통해 특정 정수와 대응되어 있습니다. 예를 들어서 65 는 알파벳 A 와 대응됩니다.
- `scanf` 를 통해 사용자로 부터 데이터를 받을 수 있습니다.
- `%c` 는 문자, `%d` 는 정수, `%f` 는 `float`, `%lf` 는 `double` 을 받습니다.

5

만약에... (조건문)

안녕하세요? 여러분. 이제 저의 강좌도 6강에 이르렀네요. 지난번의 강좌에서 배운 입출력 함수로 여러가지 재미있는 프로그램을 만들었나요? 그러한 프로그램들을 많이 만들 수록 여러분의 실력은 몇 배로 향상된다는 사실을 잊지 마시기 바랍니다. 이번 강좌에는 C 언어에서 매우 중요한 부분인 제어문 - 그 중에서도 조건문이라는 사실을 배우겠습니다.

우리가 자판기에서 음료수를 고를 때 자판기 내부에는 다음과 같은 과정이 수행 됩니다.



돈을 넣는다. 지폐인가? 맞다면 액수가 충분한가? 아니면 더 받는다 .... 이렇게 맞다 아니다 로 갈리게 됩니다. 이러한 것을 조건문 이라고 합니다. 즉 항상 실행되는 것이 아니라 특정한 조건이 맞는 경우에만 실행되는 것이지요

그런데, 위 부분에서 우리가 여태 까지 보지 못했던 특징이 있습니다. 여태까지 우리는 모든 문장들이 순차적으로 실행되어 왔습니다. 예를들어,

```
//.....
```

```
printf("안녕\n");
printf("내 이름은 Psi 야 \n");
printf("너의 이름은 뭐니?\n");
//....
```

와 같은 문장에서 처음에 **안녕** 이 출력되고 그 다음에 **내 이름은 Psi 야**, 그리고 마지막으로 **너의 이름은 뭐니?** 가 출력이 됩니다. 안녕이 출력이 되지 않거나 너의 이름은 뭐니? 가 출력되지 않는 일은 없죠.

뿐만 아니라, 내 이름은 Psi 야가 안녕 보다 먼저 출력되는 경우도 없습니다. 단지 어떠한 조건에서도 안녕, 내 이름은 Psi 야, 너의 이름은 뭐니? 가 차례대로 출력되는 것이지요.

하지만, 위의 그림을 봅시다. **지폐 인가?** 부분을 보면 만약 Yes 라면 **액수는 충분한가?** 를 실행하고 No 라면 **동전 인가?** 를 실행하게 끔 되어 있습니다. 다시말해 어떠한 경우를 만족한다면 이것을 실행하고, 또 다른 경우라면 이 것을 실행하는 꼴이지요. 이런 것을 '조건문' 이라 합니다. 어떠한 조건에 따라 실행되는 것이 달라지는 것이지요.

if 문 시작하기

아래 예제를 봅시다.

```
/* if 문 이란? */
#include <stdio.h>
int main() {
  int i;
  printf("입력하고 싶은 숫자를 입력하세요! : ");
  scanf("%d", &i);

  if (i == 7) {
    printf("당신은 행운의 숫자 7 을 입력했습니다");
  }

  return 0;
}
```

위 예제를 성공적으로 컴파일 한 후, 7 을 입력하였으면

실행 결과

```
입력하고 싶은 숫자를 입력하세요! : 7
당신은 행운의 숫자 7 을 입력했습니다
```

와 같이 나오게 됩니다. 그런데, 7 이 아닌 다른 수를 입력하였을 때 에는,

실행 결과

```
입력하고 싶은 숫자를 입력하세요! : 1
```

처럼 나오게 됩니다. 일단, 위 소스 코드에는 우리가 여태까지 보지 못했던 것이 있으니 찬찬히 뜯어보도록 합시다.

```
if (i == 7) {
  printf("당신은 행운의 숫자 7 을 입력했습니다");
}
```

이는위 소스코드에서 가장 핵심적인 부분입니다. 영어에서 흔히 말할 때 어떨 때, if 라는 단어를 쓰나요. 예를 들어서, *If you are smart enough, please learn C language* (만약 당신이 충분히 똑똑하다면, C 언어를 배워라!) 라는 문장을 보았을 때 **If** 라는 단어는 무슨 역할을 하나요? 아마 영어를 조금이나마 배운 사람들이라면, **if** 는 '만약 ~ ' 이라는 의미를 가진 것임을 바로 알 수 있습니다.

C 언어에서도 마찬가지 입니다. `if` 는 가정을 나타냅니다. 여기서는 무엇을 가정 하였을까요? 바로 `if` 문 안에 있는 `i == 7` 이 그 가정을 나타냅니다. 즉, 만약 i 의 값이 7 이라면 이라는 뜻이지요.

`if` 문은 언제나 괄호 안의 조건이 참 이라면 중괄호 속의 내용을 실행하게 되고, 아니면 중괄호 속의 내용을 실행하지 않고 지나치죠.

따라서, 만약 `i` 의 값이 7 이라면,

```
printf("당신은 행운의 숫자 7 을 입력했습니다");
```

를 실행 한 후 중괄호 밖으로 빠져 나갑니다. 그 후 다시 아래로 순차적으로 실행하게 되죠. 즉, 마지막으로 `return 0;` 가 실행됩니다.

그런데, `i` 의 값이 7 이 아니라면, `if` 문의 중괄호 속의 내용은 실행되지 않고 지나쳐 버립니다. 왜냐하면 `if` 문 에서 `i == 7` 이 '거짓' 이 되기 때문이죠. 따라서 그냥 `return 0` 만 실행이 됩니다.

참고적으로 == 와 같이 어떠한 두 값 사이의 관계를 나타내 주는 연산자를 **관계 연산자** 라고 부릅니다. 이 때, 관계 연산자의 좌측에 있는 부분을 좌변, 우측에 있는 부분을 우변이라 합니다. (즉, `3 == 2` 와 같은 경우 3 은 좌변, 2 는 우변 이라 부릅니다)

또한, 알아야 될 또 한가지 중요한 것은, 사실 관계 연산자는 어떠한 관계를 연산 한 후에, 참 이면 1 을, 거짓이면 0 을 나타내게 됩니다.

다시 말해, `if` 문은 참, 거짓에 따라서 중괄호 속의 내용을 실행 하느냐, 하지 않느냐 결정하는 것 처럼 보이지만 실제로는, **`if` 문 속의 조건이 0 인가 (거짓), 0 이 아닌가 (참) 에 따라서 실행의 유무를 판별하게 되죠**

따라서, `if (0)` 이라 한다면 그 중괄호 속의 내용은 절대로 실행되지 않고, `if(1)` 이라 한다면 그 중괄호 속의 내용은 100% 실행되게 됩니다.

```c
/* 나눗셈 예제 */
#include <stdio.h>
int main() {
  double i, j;
  printf("나누고 싶은 두 정수를 입력하세요 : ");
  scanf("%lf %lf", &i, &j);

  printf("%f 를 %f 로 나눈 결과는 : %f \n", i, j, i / j);

  return 0;
}
```

성공적으로 컴파일 후 10 과 3 을 입력하였다면 아래와 같이 나오게 됩니다

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 10
3
10.000000 를 3.000000 로 나눈 결과는 : 3.333333
```

와우! 성공적으로 되었습니다. 그런데 위에서 나온 소스 코드는 우리가 여태까지 바왔던 코드와 다를 바가 없습니다. 여태까지 배운 기능들만 이용해서 만든 것이지요. 하지만 바로 이 부분에서 문제가 대두됩니다.

컴파일한 프로그램을 다시 실행시켜 1 과 0 을 차례대로 입력해 보세요. 즉, 1 을 0 으로 나누어 보세요.

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 1
0
1.000000 를 0.000000 로 나눈 결과는 : inf
```

위에서 보시는 것과 같이 나눈 결과가 `inf` 이라는 이상한 결과를 내뿜었습니다. 왜 일까요? 왜냐하면 수학에서, (즉 컴퓨터 에서) 어떠한 수를 0 으로 나누는 것은 금지되어 있기 때문이죠. 위 `i` 와 `j` 변수가 `double` 로 선언되어 있어서 망정이지 `i,j` 변수를 `int` 형으로 선언하였다면 프로그램은 애러를 내뿜고 종료 됩니다.

이 문제를 대수롭지 않게 여긴다면 큰 문제입니다. 예를들어서 여러분이 엑셀로 열심히 작업을 하였는데 실수로 어떤 수를 0 으로 나누는 작업을 하였더니 힘들게 작업한 엑셀이 종료되 버려 파일이

날아가 버리면 여러분은 다시는 엑셀을 쓰지 않을 것 입니다. 따라서, 우리는 나누는 수 (제수) 가 0 이 아닌지 확인할 필요성이 있습니다. 즉, 제수가 0 이면 나누지 않고 0 이 아니면 나누는 것이지요.

따라서 위 프로그램을 아래와 같이 수정합시다.

```c
#include <stdio.h>
int main() {
  double i, j;
  printf("나누고 싶은 두 정수를 입력하세요 : ");
  scanf("%lf %lf", &i, &j);

  if (j == 0) {
    printf("0 으로 나눌 수 없습니다. \n");
    return 0;
  }
  printf("%f 를 %f 로 나눈 결과는 : %f \n", i, j, i / j);

  return 0;
}
```

만약 1 을 0 으로 나누었다면

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 1
0
0 으로 나눌 수 없습니다.
```

그리고 다시 10 을 3 으로 나누어 보면

실행 결과

```
나누고 싶은 두 정수를 입력하세요 : 10
3
10.000000 를 3.000000 로 나눈 결과는 : 3.333333
```

로 위와 같이 정상적으로 나타납니다. 그럼 위 소스코드를 뜯어 보기로 합시다.

```c
{
  printf("0 으로 나눌 수 없습니다. \n");
  return 0;
}
```

만약 j 의 값이 0 이라면 중괄호 속의 내용이 실행되며, *0* 으로 나눌 수 없습니다 가 표시되고 프로그램이 종료(return 0) 됩니다.

반면에, j 의 값이 0 이 아니라면 중괄호 속의 내용이 실행되지 않습니다. 즉, 아래의 내용이 실행되게 됩니다.

```
printf("%f 를 %f 로 나눈 결과는 : %f \n", i, j, i / j);

return 0;
```

이렇듯, if 문은 여러 조건에 따른 처리를 위해 사용합니다. 먼저 나왔던 예제는 i 의 값이 7 일 때의 처리를 위해 if 문을 사용하였고 위의 예제는 j 의 값이 0 일 때의 처리를 위해 사용하였습니다.

```
/* 합격? 불합격? */
#include <stdio.h>
int main() {
  int score;

  printf("당신의 수학점수를 입력하세요! : ");
  scanf("%d", &score);

  if (score >= 90) {
    printf("당신은 합격입니다! \n");
  }

  if (score < 90) {
    printf("당신은 불합격 입니다! \n");
  }

  return 0;
}
```

위 소스를 성공적으로 컴파일하였다면 다음과 같이 나옵니다.만약 당신의 수학점수로 91 점을 입력하였다면,

실행 결과

```
당신의 수학점수를 입력하세요! : 91
당신은 합격입니다!
```

만약 당신의 수학 점수로 80 점을 입력하였다면

실행 결과

```
당신의 수학점수를 입력하세요! : 80
당신은 불합격 입니다!
```

와 같이 나타납니다.

```c
if (score >= 90) {
  printf("당신은 합격입니다! \n");
}

if (score < 90) {
  printf("당신은 불합격 입니다! \n");
}
```

위 소스의 핵심이라 할 수 있는 부분은 위 두 부분입니다.일단, `if(score >= 90)` 이라는 부분부터 살펴 봅시다. 이미 짐작은 했겠지만 `>=` 은 ~ 이상, 즉 ~ 보다 크거나 같다 를 의미합니다. 따라서, `score` 의 값이 90 보다 '크거나 같으면' `if` 문 안의 내용이 참 (true) 이 되어 중괄호 속의 내용이 실행됩니다.

따라서, 처음에 우리가 91 점을 입력하였을 때 `score >= 90` 이 참이 되어서

```c
printf("당신은 합격 입니다! \n");
```

가 실행되었습니다. 그런데 여기서 주의해야 할 점은 `score => 90` 이라고 하면 안된다는 것입니다. 이렇게 쓰면 컴파일러는 인식을 하지 못합니다.

이 부분에서 헷갈리는 사람들은 '크거나 같다' 라는 말 그대로 옮겨 적었다고 생각하세요. `>=` 는 크거나 (>) 같다 (=) 를 합친 것이다!

마찬가지로, 아래 `score < 90` 도 보자면 `score` 가 90 미만일 때 참이다 라는 사실을 나타낸 것임을 알 수 있습니다.

```c
/* 크기 비교하기 */
#include <stdio.h>
int main() {
  int i, j;

  printf("크기를 비교할 두 수를 입력해 주세요 : ");
  scanf("%d %d", &i, &j);

  if (i > j)  // i 가 j 보다 크면
  {
    printf("%d 는 %d 보다 큽니다 \n", i, j);
```

```
  }
  if (i < j)  // i 가 j 보다 작으면
  {
    printf("%d 는 %d 보다 작습니다 \n", i, j);
  }
  if (i >= j)  // i 가 j 보다 크거나 같으면
  {
    printf("%d 는 %d 보다 크거나 같습니다 \n", i, j);
  }
  if (i <= j)  // i 가 j 보다 작거나 같으면
  {
    printf("%d 는 %d 보다 작거나 같습니다 \n", i, j);
  }
  if (i == j)  // i 와 j 가 같으면
  {
    printf("%d 는 %d 와(과) 같습니다 \n", i, j);
  }
  if (i != j)  // i 와 j 가 다르면
  {
    printf("%d 는 %d 와(과) 다릅니다 \n", i, j);
  }

  return 0;
}
```

위 내용을 성공적으로 컴파일 후, 10 과 4 를 입력하였다면

실행 결과

```
크기를 비교할 두 수를 입력해 주세요 : 10 4
10 는 4 보다 큽니다
10 는 4 보다 크거나 같습니다
10 는 4 와(과) 다릅니다
```

와 같이 나타나게 됩니다.

이번 예제에서는 소스의 길이가 약간 깁니다. 하지만 따지고 보면 상당히 간단한 구조로 되어 있음을 알 수 있습니다. 일단 위의 예제에 관계 연산자 들의 역할에 대해 알아봅시다.

1. `>=` : 좌변이 우변보다 같거나 크면 참이 됩니다. (`6 >= 3` : 참, `6 >= 6` : 참, `6 >= 8` : 거짓)
2. `>` : 좌변이 우변보다 크면 참이 됩니다. (`6 > 3` : 참, `6 > 6` : 거짓, `6 > 8` : 거짓)
3. `<=` : 좌변이 우변보다 작거나 같으면 참이 됩니다. (`6 <= 3` : 거짓, `6 <= 6` : 참, `6 <= 8` : 참)

4. `<` : 좌변이 우변보다 작으면 참이 됩니다.(`6 < 3` : 거짓, `6 < 6` : 거짓, `6 < 8` : 참)
5. `==` : 좌변과 우변이 같으면 참이 됩니다.(`6 == 3` : 거짓, `6 == 6` : 참, `6 == 8` : 거짓)
6. `!=` : 좌변과 우변이 다르면 참이 됩니다. (`6 != 3` : 참, `6 != 6` : 거짓, `6 != 8` : 참)

따라서, 위 관계연산자에 따라 위 프로그램이 실행이 됩니다. 한 번 여러가지 숫자를 집어 넣으면서 확인해 보세요.

마지막으로 `if` 문의 구조에 대해서 간단히 정리해 보자면

```
if (/* 조건 */) {
  /* 명령 */
}
```

와 같이 됩니다. 잊지 마세요!

if - else 문 시작하기

```
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else {
    printf("그냥 보통 숫자인 %d 를 입력했군요\n", num);
  }
  return 0;
}
```

만약 성공적으로 컴파일 하였다면 7 을 입력했을 때,

실행 결과

```
아무 숫자나 입력해 보세요 : 7
행운의 숫자 7 이군요!
```

그리고, 그 외 7 이 아닌 다른 숫자를 입력하였을 때 에는,

실행 결과

```
아무 숫자나 입력해 보세요 : 100
그냥 보통 숫자인 100 를 입력했군요
```

와 같이 나오게 됩니다. 자, 이제 위 소스를 뜯어 봅시다.

```
if (num == 7) {
  printf("행운의 숫자 7 이군요!\n");
}
```

여태 까지 보왔듯이, 이 부분은 그냥 평범한 `if` 문이군요. 하지만 그 다음 부분에 심상치 않은 것이 등장합니다.

```
else {
  printf("그냥 보통 숫자인 %d 를 입력했군요\n", num);
}
```

이번에는 여태까지 배우지 않은 것인 `else` 라는 것이 등장합니다. 영어를 잘 하시는 분들은 지레 짐작하시고 있었겠지만 `else` 는 '그 외의~ , 그 밖의~' 의 뜻으로 사용되는 단어입니다.

그렇다면 여기서도 그러한 의미를 나타내는 것인가요?

맞습니다. `else` 는 바로 '앞선 `if` 문이 조건을 만족하지 않을 때' 를 나타냅니다. 즉, 앞선 `if` 문이 조건을 만족 안할 때 해야 할 명령을 바로 `else` 문에 써 주는 것이지요. 다시 말해, `else` 문은 떨거지(?) 들을 처리하는 부분입니다.

위의 경우에서도 만약 `num` 의 값이 7 이 아니였다면 `if` 문을 만족 안하게 되는 것입니다. 그러면 자연스럽게 `else` 로 넘어와서 "그냥 보통 숫자인 ... 를 입력했군요" 를 출력하게 되는 것이지요. 하지만, `num` 의 값이 7 이 였다면 `if` 문을 만족하는 것이기 때문에 `else` 는 거들떠 보지도 않고 넘어가게 됩니다.

```
/* if - else 죽음의 숫자? */
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else {
    if (num == 4) {
```

```
      printf("죽음의 숫자 4 인가요 ;;; \n");
    } else {
      printf("그냥 평범한 숫자 %d \n", num);
    }
  }
  return 0;
}
```

이번에 성공적으로 컴파일 한 후, 숫자들을 입력해 보면 비슷한 결과가 나오지만 4 를 입력했을 경우 "죽음의 숫자 4 인가요 ;;; " 가 나오게 됩니다.

실행 결과

```
아무 숫자나 입력해 보세요 : 4
죽음의 숫자 4 인가요 ;;;
```

자, 이제 소스를 뜯어 보기로 합시다.

```
else {
  if (num == 4) {
    printf("죽음의 숫자 4 인가요 ;;; \n");
  } else {
    printf("그냥 평범한 숫자 %d \n", num);
  }
}
```

앞 `if (num == 7)` 부분은 이미 설명 했으니 생략하기로 하고, `else` 문의 구조만 뜯어 보기로 합시다. 만약 `num` 의 값이 4 였다고 합시다. 그렇다면, 처음에 만나는 `if` 문에서 `num == 7` 이 거짓이 되므로 `else` 로 넘어가게 됩니다.

그런데, `else` 의 명령을 실행하려고 하는데 보니, 또 `if` 문이 있네요. 이번에는 `if (num == 4)` 가 나타납니다. 하지만 아까와는 달리 `num == 4` 가 참이므로 그 `if` 문의 중괄호 속의 명령, 즉 "죽음의 숫자 4 인가요 ;;;" 가 출력되게 됩니다.

그리고, 앞 예제에서 설명했듯이 `if(num == 4)` 아래의 `else` 는 무시 하게 되고, 끝나게 되는 것이지요.

그렇다면 이제 아이디어를 확장해서 `num` 이 1 부터 10 일 때 까지 특별한 설명을 달기로 합시다. 그러면 아래와 같이 프로그램을 짜야 되겠죠.

```
/* 쓰레기 코드 */
#include <stdio.h>
int main() {
  int num;
```

```
  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else {
    if (num == 4) {
      printf("죽음의 숫자 4 인가요 ;;; \n");
    } else {
      if (num == 1) {
        printf("첫 번째 숫자!! \n");
      } else {
        if (num == 2) {
          printf("이 숫자는 바로 두번째 숫자 \n");
        } else {
          ......(생략).....
        }
      }
    }
  }
  return 0;
}
```

정말 믿도 끝도 없이 길어져서 나중에는 중괄호가 너무 많아 헷갈리기도 하고, 보기도 불편하게 됩니다. 하지만, C 언어는 위대한지라, 이 문제를 간단히 해결하였습니다.

```
/* 새로쓰는 죽음의 숫자 예제 */
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("행운의 숫자 7 이군요!\n");
  } else if (num == 4) {
    printf("죽음의 숫자 4 인가요 ;;; \n");
  } else {
    printf("그냥 평범한 숫자 %d \n", num);
  }
  return 0;
}
```

위 코드를 실행해 보면 앞선 예제와 똑같이 작동합니다. 하지만 코드도 훨씬 보기 편해 졌고 난잡하던 중괄호도 어느 정도 정리가 된 것 같군요. 그렇다면 정말 하는 일이 똑같을 까요? 네, 똑같습니다. 왜냐하면

```
if (/* 조건 1 */) {
  // 명령 1;
} else {
  if (/* 조건 2 */) {
    // 명령 2;
  } else {
    if (/* 조건 3 */) {
      // 명령 3;
    } else {
      // ....
    }
  }
}
```

위와 같은 코드를 단지 아래 처럼 '간단히' 표현한 것이기 때문이죠.

```
if (/* 조건 1 */) {
  // 명령 1;
} else if (/* 조건 2 */) {
  // 명령 2;
} else if (/* 조건 3 */) {
  //명령 3;
}
....else {
  // 명령 ;
}
```

단지, 보기 편하게 하기 위해 '간략하게' 줄인 꼴과 같다는 것입니다. 마치 `a = a + 10` 을 `a+= 10` 으로 바꾼 것 처럼 말이죠.

```
/* if 와 if- else if 의 차이*/
#include <stdio.h>
int main() {
  int num;

  printf("아무 숫자나 입력해 보세요 : ");
  scanf("%d", &num);

  if (num == 7) {
    printf("a 행운의 숫자 7 이군요!\n");
  } else if (num == 7) {
    printf("b 행운의 숫자 7 이군요! \n");
  }

  // 비교
  if (num == 7) {
    printf("c 행운의 숫자 7 이군요!\n");
```

```
  }
  if (num == 7) {
    printf("d 행운의 숫자 7 이군요! \n");
  }

  return 0;
}
```

성공적으로 컴파일 후, 7 을 입력하였다면

실행 결과

```
아무 숫자나 입력해 보세요 : 7
a 행운의 숫자 7 이군요!
c 행운의 숫자 7 이군요!
d 행운의 숫자 7 이군요!
```

와 같이 나오게 됩니다. 여기서 주목해야 할 점은, 출력되는 문장 앞의 알파벳 (a,c,d) 인데 이는 각 문장이 위 프로그램의 어느 부분에서 출력되는 지 알려줍니다.

우리가 컴퓨터 라고 생각하고 프로그램을 실행해 봅시다. 통상적으로 프로그램은 소스코드의 위에서 부터 아래 방향으로 실행됩니다

```
if (num == 7) {
  printf("a 행운의 숫자 7 이군요!\n");
} else if (num == 7) {
  printf("b 행운의 숫자 7 이군요! \n");
}
```

컴퓨터가 쭉 프로그램을 읽다가 위 부분에 도달하면

"어! `if` 문이군. `num` 의 값이 7 인지 확인해 볼까?" 라고 확인을 합니다. 그런데, 참 이므로"오, `if` 문이 참 이군. 그렇다면 중괄호 속의 내용을 실행해야겠다! " 하며 "a 행운의 숫자 7 이군요" 를 출력합니다.

그런데, 그 다음 부분인 `else if(num == 7)` 에서도 마찬가지로 num 의 값이 7 이므로 `num == 7` 이 참이 되어서 `else if` 가 참이 되어 "b 행운의 숫자 7 이군요! " 도 출력되어야 할 것 같습니다. 하지만, 결과를 보아하니 출력이 되지 않았습니다. 왜 일까요?

왜냐하면, 앞에서 누누히 설명했듯이 `else` 문은 전제 조건이 '앞의 `if` 문이 참이 아닐 때' 실행된다는 사실을 기본으로 깔고 있기 때문이죠. 따라서, 앞의 `if` 문이 참이므로 실행이 될 수 없습니다. 따라서, "b 행운의 숫자 7 이군요! " 도 출력되지 않습니다. 참고적으로 `else` 문은 언제나 `if` 문의 결과에 따라 실행 여부가 결정되므로 언제나 `else` 를 사용하려면 `if` 도 반드시 함께 따라 사용해야 합니다.

다시 컴퓨터가 쭉 읽다가 아래와 같은 문장을 발견했네요.

```
if (num == 7) {
  printf("c 행운의 숫자 7 이군요!\n");
}
if (num == 7) {
  printf("d 행운의 숫자 7 이군요! \n");
}
```

"어! if 문이군. 그런데 num 의 값이 7 이므로 이 if 문은 참이야. 중괄호 속의 내용을 실행해야지" 하면서 "c 행운의 숫자 7 이군요!" 가 출력된다. 마찬가지로 아래도

"어! if 문이군. 그런데 num 의 값이 7 이므로 이 if 문은 참이야. 중괄호 속의 내용을 실행해야지" 하면서 "d 행운의 숫자 7 이군요!" 가 출력되는 것입니다.

```
#include <stdio.h>
int main() {
  float ave_score;
  float math, english, science, programming;

  printf("수학, 영어, 과학, 컴퓨터 프로그래밍 점수를 각각 입력해 주세요 ! : ");
  scanf("%f %f %f %f", &math, &english, &science, &programming);

  ave_score =
      (math + english + science + programming) / 4;  // 4 과목의 평균을 구한다.
  printf("당신의 평균 점수는 %f 입니다 \n", ave_score);
  if (ave_score >= 90) {
    printf("당신은 우등생 입니다. ");
  } else if (ave_score >= 30) {
    printf("조금만 노력하세요!. \n");
  } else {
    printf("공부를 발로 합니까? \n");
  }

  return 0;
}
 ~/Te
```

만약 성공적으로 컴파일 하였다면

실행 결과

```
수학, 영어, 과학, 컴퓨터 프로그래밍 점수를 각각 입력해 주세요 ! : 100
90
90
85
```

```
당신의 평균 점수는 91.250000 입니다
당신은 우등생 입니다.
```

와 같이 나오게 됩니다. 그 외에도, 다른 값들을 입력하면 다른 결과가 출력됨을 알 수 있습니다.

```
ave_score = (math + english + science + programming) / 4;
```

아마, 산술 연산을 까먹으신 분들은 없겠죠? 위 식이 2 초 내로 이해가 되지 않는다면 4 강을 다시 공부하시기 바랍니다. 위 식은, 수학, 영어, 과학, 프로그래밍 점수의 평균을 구해서 ave_score 라는 변수에 대입하는 식입니다.

```
if (ave_score >= 90) {
  printf("당신은 우등생 입니다. ");
} else if (ave_score >= 40) {
  printf("조금만 노력하세요!. \n");
} else if (ave_score > 0) {
  printf("공부를 발로 합니까? \n");
} else {
  printf("인생을 포기하셨군요. \n");
}
```

위 프로그램의 핵심 부분(?) 이라 할 수 있는 이 부분을 잘 살펴봅시다. 만약 내 평균 점수가 93 이라면, "당신은 우등생 입니다" 가 출력되게 되죠. 그리고, 아래 else if 와 else 는 무시하고 종료됩니다.

하지만 평균 점수가 40 이라면, 위의 if (ave_score >= 90) 이 거짓이 되어 다음으로 넘어가죠. 즉, else if (ave_score >= 30) 을 합니다. 이번에는 참이 되므로, 조금만 노력하세요! 가 출력이 되고 종료가 됩니다.

또한, 평균점수가 10 점 이라면 위의 if 와 else if 모두 거짓이지만 else if(ave_score > 0) 은 참이 되어 공부를 발로 합니까? 가 출력 됩니다.

마지막으로 평균점수가 0 점 이하 라면, 떨거지 처리(?) 인 else 에서 참이 되어서 인생을 포기하였군요 가 실행됩니다.

```
/* 크기 비교 */
#include <stdio.h>
int main() {
  int a;
  printf("아무 숫자나 입력하세요 : ");
  scanf("%d", &a);

  if (a >= 10) {
    if (a < 20) {
      printf(" %d 는 10 이상, 20 미만인 수 입니다. \n", a);
```

```
    }
  }
  return 0;
}
```

성공적으로 컴파일 후, 10 이상 20 미만의 수를 입력했다면

실행 결과

```
아무 숫자나 입력하세요 : 10
 10 는 10 이상, 20 미만인 수 입니다.
```

와 같이 나오게 됩니다. 위 소스 코드는 간단합니다. 우리가 여태까지 배운 내용 만으로도 충분히 이해 할 수 있습니다!

```
if (a >= 10) {
  if (a < 20) {
    printf(" %d 는 10 이상, 20 미만인 수 입니다. \n", a);
  }
}
```

처음에, a 의 값이 10 이상이면 참이 되어서 중괄호 속의 내용을 실행하게 되고, 또한 거기서 a < 20 이라는 if 문을 만나게 되는데 이것 조차 참이라면 printf 가 실행되겠지요,.

그런데, 사실 위 문장은 아래와 같이 간단히 줄여 쓸 수 있습니다.

논리 연산자

```
/* 논리 연산자 */
#include <stdio.h>
int main() {
  int a;
  printf("아무 숫자나 입력하세요 : ");
  scanf("%d", &a);

  if (a >= 10 && a < 20) {
    printf(" %d 는 10 이상, 20 미만인 수 입니다. \n", a);
  }

  return 0;
}
```

위 소스를 그대로 컴파일 해 보면 위와 결과가 똑같이 나옵니다. 그렇다면 '&&' 는 무엇일까요? 그것은 바로 **논리 곱(AND)** 라고 불리는 논리 연산자 입니다.

앞에서 우리는 AND 연산에 대해 배운 적이 있었습니다. (기억이 나지 않는다면 여기를 클릭). 이 때, AND 연산의 특징은 바로 오직 `1 AND 1` 만이 결과가 1 이였고, `1 AND 0` 또는 `0 AND 0` 은 모두 결과가 0 이였습니다.

여기서도 마찬가지 입니다. 위에서도 이야기 했지만 '참' 은 숫자 1 에 대응되고 '거짓' 은 숫자 0 에 대응 됩니다. 따라서, `a >= 10` 이 참이라면 '1' 을 나타내고, 거짓이라면 '0' 을 나타냅니다.

마찬가지로, `a < 20` 도 참 이라면 '1' 을 나타내고, 거짓이라면 '0' 을 나타냅니다. 만약 `a >= 10` 도 참이고 `a < 20` 도 참 이라면 `1 AND 1` 을 연산하는 것과 같게 되어서 결과가 1, 즉 참이 되어 `if` 문의 중괄호 속의 내용을 실행하게 되죠.

반면에 `a < 10` 라던지 `a >= 20` 이여서 둘 중 하나라도 조건을 만족하지 않게 된다면 `1 AND 0` 이나 `0 AND 1` 을 하는 것과 같게 되어 결과가 `0`, 즉 거짓이 됩니다. 따라서 중괄호 속의 내용은 실행되지 않게 됩니다.

정리하자면, && 는 두 개의 조건식이 모두 '참' 이 되어야 `if` 문 속의 내용을 실행하는 것이 됩니다.

그렇다면 우리가 여태 알고 있었던 AND 연산 기호인 & 를 쓰지 않고 && 를 쓰는 것일까요? 그 이유는 & 하나는, 숫자 사이의 AND 연산을 사용 할 때 쓰고 논리 곱 연산자인 && 는 두 개 이상의 조건식 사이에서 사용됩니다.

따라서 `if (a >= 10 & a < 20 )` 이나, `0x10 && 0xAE` 와 같은 연산은 모두 틀린 것이 됩니다.

```c
/* 논리 합 */
#include <stdio.h>
int main() {
  int height, weight;
  printf("당신의 키와 몸무게를 각각 입력해 주세요 : ");
  scanf("%d %d", &height, &weight);

  if (height >= 190 || weight >= 100) {
    printf("당신은 '거구' 입니다. \n");
  }

  return 0;
}
```

성공적으로 실행 후, 키를 190 이상으로 입력했거나 몸무게를 100 이상으로 입력했다면

실행 결과

```
당신의 키와 몸무게를 각각 입력해 주세요 : 200 90
```

```
당신은 '거구' 입니다.
```

위와 같이 나오게 되죠.

이번에는 || 라는 것이 등장하였습니다. 앞서 AND 가 && 였다는 것을 보아, || 는 OR 를 나타내는 논리 연산자임을 알 수 있습니다.

기억을 되살려 봅시다. AND 와 OR 의 차이가 뭐였죠? 음.... 아, 기억 나는군요. AND 가 두 조건이 모두 참 일 때, 참을 반환한다면, OR 은 두 조건이 모두 거짓 일 때 만 거짓을 반환합니다. 다시말해 (참) || (거짓) == (참) 이 된다는 것이지요.

```
if (height >= 190 || weight >= 100)
```

그렇다면 위 if 문을 살펴 봅시다. height >= 190 이 참 이라면, OR 연산한 값은 weight 의 크기에 관계없이 무조건 참이 되어서 중괄호 속의 내용을 실행 합니다. 또한 height >= 190 이 거짓이여도, weight >= 100 이 참 이라면 중괄호 속의 내용을 실행하게 되죠.

따라서, OR 논리 연산자는 조건식에서 적어도 어느 하나가 참 이라면 무조건 if 문의 내용을 실행해 주게 됩니다.

```c
/* 논리 부정 */
#include <stdio.h>
int main() {
  int height, weight;
  printf("당신의 키와 몸무게를 각각 입력해 주세요 : ");
  scanf("%d %d", &height, &weight);

  if (height >= 190 || weight >= 100) {
    printf("당신은 '거구' 입니다. \n");
  }
  if (!(height >= 190 || weight >= 100)) {
    printf("당신은 거구가 아닙니다. \n");
  }

  return 0;
}
```

위 소스를 성공적으로 컴파일 한 후, 180 과 80 을 입력하였다면

실행 결과

```
당신의 키와 몸무게를 각각 입력해 주세요 : 180 80
당신은 거구가 아닙니다.
```

와 같이 나옵니다.

위 소스에서 관심있게 살펴 보아야 할 부분은 바로 이 부분이죠.

```
if (!(height >= 190 || weight >= 100)) {
  printf("당신은 거구가 아닙니다. \n");
}
```

(참고로 위의 소스와 다른 점은 `height` 앞에 `!` 가 붙었다는 점 입니다.) 다른 것은 다 똑같은데, 새로 붙은 `!` 가 무슨 역할을 하는 것 같나요?

아마도 예측 하셨겠지만, `!` 는 NOT 을 취해주는 연산자 입니다. 다시 말해, **반전**을 시켜 주는 것이지요. 위 경우, `height >= 190 || weight >= 100` 의 가 **거짓**일 경우에만, 다시 말해서 `height < 190 && weight < 100` 인 경우에만 중괄호 속의 내용이 실행 됩니다. 어때요, 간단하죠?

이것으로 C 언어에서 중요한 부분인 `if` 문에 대해 알아 보았습니다. 이제 마지막으로 할 일은 바로 `if` 문을 가지고 여러가지 프로그램을 만들어 보는 것입니다. 솔직히 말해서, `if` 문 이전까지 배운 내용으로는 그다지 할 것들이 없었습니다.

하지만, 이번 `if` 문을 통해서 컴퓨터에게 '생각(?)' 할 수 있는 능력을 부여 할 수 있게 됩니다. 따라서, 무궁 무진한 것들을 만들어 낼 수 있게 되죠. 뭐, 만들어 볼 것들로 제가 추천하는 것들은 계산기나 성적표라던지, 등등. 아무튼, 힘내세요.

뭘 배웠지?

- `if, else if, else` 가 무엇 인지 알고 있습니다.
- 논리 연산자 `&&, ||` 를 배웠습니다.
- `!` 의 역할을 알고 있습니다.
- `0 <= a <= 1` 을 잘못된 사용 예 입니다. 이 대신 `0 <= a && a <= 1` 과 같이 사용해야 합니다.