2

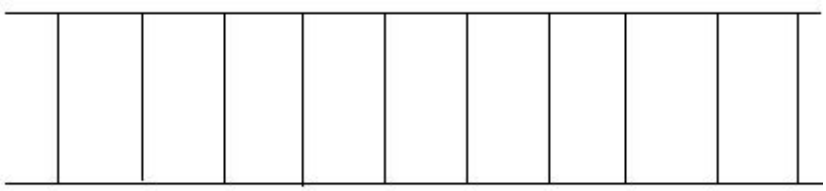
변수 (variable)

안녕하세요? 여러분. 잘 지내셨나요. 지난 번에 처음으로 C 코드를 분석한 것은 이해가 잘 되셨나요? 이해가 잘 안되셨다도 괜찮습니다. 점점 C 언어를 배워감에 따라, 기존이 이해가 안 되었던 것들도 언젠가는 '아 이래서 그랬구나' 하는 순간이 오게 됩니다. 이 단계에서 여러분이 취해야 할 자세는 일단 이해가 안되는 것은 일단, **암기** 하고, 포기하지 않는 것이 필요합니다.

변수란 무엇인가?

컴퓨터는 많은 내용을 기억 해야 합니다. 정확히 말하면, 컴퓨터의 '메모리' 라는 부분에 전기적인 신호를 써 놓는 것이죠. 컴퓨터가 무엇을 기억해야 되냐고 생각할 수 있지만, 우리가 많이 하는 게임인 스타크레프트만 보아도 일단, 각 유닛의 체력과 마나, 그리고 실시간으로 바뀌는 미네랄과 가스, 뿐만 아니라 유닛의 위치, 유닛의 데미지 등 모든 것을 기억해야지 우리가 게임을 제대로 즐길 수 있게 되겠지요. 만약 컴퓨터가 미네랄의 양을 제대로 기억 못한다면 미네랄이 갑자기 100 에서 0 이 되거나 10 에서 9999 로 바뀌는 참사가 발생합니다.

그렇다면 컴퓨터는 이러한 데이터들을 어떻게 기억할까요? 바로 컴퓨터의 메모리, 즉 **램(RAM) 이라는 특별한 기억공간**에 이를 기록합니다. 보통 우리는 흔히 램 을 설명할 때 아래 처럼 표시합니다.

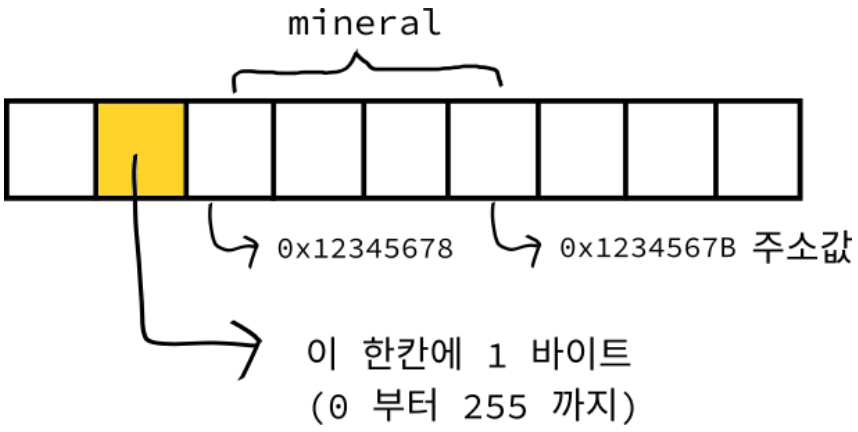


마치, 각 방에 데이터들이 저장됩니다. 이 때, 컴퓨터는 각 방에 이름을 붙이는데 단순하게 숫자로

이름을 붙입니다. 0 번, 1 번, 2 번, .. 와 같이 말입니다. 우리 대부분이 사용하는 32 비트 CPU 에서는 최대 2ş$ 개(4GB) - 총, 42 억개 달하는 방을 가질 수 있게 되겠지요. 참고로 32 비트 숫자를 매번 쓰는게 매우 힘들기 때문에, 대개 16진법으로 주소값을 나타냅니다.

예를 들어서, 컴퓨터가 `0x12345678` 부터 `0x1234567B` 부분에 내가 캔 미네랄의 양에 관한 정보를 저장했다고 합시다. (이 한칸에는 1 바이트, 즉 -128 부터 127 까지의 수 데이터를 저장할 수 있습니다) 만약 우리가 건물을 지을 때, 내가 가진 미네랄의 양이 충분한 지 확인하기 위해, 내가 캔 미네랄의 양에 관한 정보가 필요합니다. 그런데, 이렇게 미네랄에 관한 정보가 필요로 할 때 마다 이 길고 알아보기 힘든 복잡한 주소를 일일이 써야 한다면 상당히 힘들겠지요.[1)]

하지만 다행히도 C 언어에는 **변수** 라는 것이 있어서, 이 모든 작업을 쉽게 할 수 있습니다. 예를들어, 내가 캔 미네랄의 양을 `mineral` 이라는 변수에 저장했다고 합시다. 그렇다면 컴퓨터는 '알아서' 메모리의 어딘가에 `mineral` 의 방을 주고 그 내용을 저장합니다. 예를들어서, 컴퓨터가 이 `mineral` 이라는 변수에게 4 칸의 자리를 할당해 주었다고 합시다. 이는 아래 그림처럼 메모리 상에 표시됩니다.



이 때, 우리가 미네랄을 더 캐서 8 을 추가해야한다고 봅시다. 만약 이전에 8 을 추가한다면 `0x12345678` 부터 `0x1234567B` 까지의 모든 내용을 불러와서 8 을 더한 후, 다시 집어넣는 작업을 일일이 손으로 써 주어야 되었을 것입니다.

하지만, 이제는 단순히 `mineral = mineral + 8` 과 같이 써 주기만 한다면 `mineral` 에 8 이 더해지는 것이죠. (만약 `mineral = mineral + 8` 이라는 식이 이해가 안되도 그냥 넘어가세요. 이 처럼 간단해 진다는 것을 말해주고 싶었을 뿐입니다)

이와 같이 C 언어에서, **바뀔 수 있는 어떤 값을 보관하는 곳을 변수** 라고 합니다. 영어로는 *Variable* 이라 하는데, 말 그대로 바뀔 수 있는 것들 이라는 뜻입니다.

1) 0x12345678, 0x12345679, 0x1234567A, 0x1234567B 이렇게 4 개를 사용합니다.

변수 선언하기

```c
/* 변수 알아보기 */
#include <stdio.h>
int main() {
  int a;
  a = 10;
  printf("a 의 값은 : %d \n", a);
  return 0;
}
```

프로젝트를 만들어 위의 내용을 적은 후, 컴파일 해봅시다. 까먹었다면 1 강을 참조하세요. 만약 성공적으로 하였다면

실행 결과

```
a 의 값은 : 10
```

와 같이 나옵니다.

일단, 이번에도 역시 생소한 것들이 나왔기 때문에 한 문장씩 차근차근 살펴 봅시다.

```c
int a;
```

음, 이게 무엇일까요? 이전에 `int main()` 에서 보았던 `int` 가 다시 나타났군요.사실 이 문장에 뜻은 `a` 라는 변수를 우리가 쓰겠다고 컴파일러에게 알리는 것입니다. 만약 이러한 문장이 없다면 우리가 `x` 가 뭐고 `y` 가 뭔지 알려주지도 않은 채, 친구에게 `x + y` 가 얼마냐? 하고 물어보는 것과 똑같은 격이 되는 것이지요.

이 때, `a` 앞에 붙은 `int` 라는 것은 **int** 형의 데이터를 보관한다는 뜻으로, `a` 에 -2147483648 에서부터 2147483647 까지의 정수를 보관 할 수 있게 됩니다. 따라서, 만약 중간의 문장을

```c
a = 10000000000000;
```

와 같이 한다면 아마 `a` 의 값을 출력하였을 때, 이상한 결과가 나오게 됩니다. 왜냐하면 보관할 수 있는 범위를 초과하는 수를 보관했기 때문이죠.

그럼 이제, 걱정이 생깁니다. `a` 에 고작 10 밖에 안 넣을 거 면서, 굳이 2147483647 까지 표현할 수 있는 `int` 형의 변수를 왜 사용했냐고 물을 수 있습니다. 물론, `int` 형 보다 작은 범위의 숫자 데이터 만을 가지는 형식이 있기는 하지만 (`char` 등등), **일반적인 경우 정수 데이터를 보관할 때 `int` 형 변수를 사용합니다.**

또한, 2147483647 보다 큰 수를 사용하려면 어떻게 해야되냐는 궁금증도 생기지요. 물론 이 보다도 훨씬 큰 숫자를 처리하는 데이터 형식이 있습니다. 아래의 표를 참조하세요.

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)

http://www.cplusplus.com/doc/tutorial/variables/에서 가져왔습니다.

세번째 열인 Range 를 보시면, unsigned 와 signed 라고 나뉜 것이 있는데, 보통 int 라 하면 signed int 를 뜻합니다. 이는 음수와 양수 모두 표시할 수 있는 대신에 양수로 표현할 수 있는 범위가 줄어듭니다.

반면에 unsigned int 는 양수만을 표현할 수 있는 대신에, 양수로 표현할 수 있는 범위가 두 배로 늘어납니다. 또한 마지막에 보면 float, double, long double 이 있는데 이들은 '실수형' 자료형으로 소수(0.1, 1.4123 등) 을 표현 할 수 있습니다. 뿐만 아니라 double 의 경우, $\pm 2.3 \times 10^{-308} \backsim \pm 1.7 \times 10^{308}$ 의 수들을 표현 할 수 있습니다. (이에 대한 정확한 설명은 후에 다루겠습니다.)

```
a = 10;
```

위 문장은 무엇을 의미할까요? 언뜻 보기에도 감이 오시겠지만, 변수 a 에 10 을 집어넣는 다는 것입니다. 따라서 나중에 a 의 값을 출력한다면 10 이 나올 것입니다. 이와 같은 형태의 문장은 뒤에서 연산자에 대해 다룰 때 다시 알아보도록 하겠습니다.

```
printf("a 의 값은 : %d \n", a);
```

마지막으로, 지난번에도 보았던 printf 입니다. 그런데, 약간 다른 것이 있습니다. %d 가 출력되는 부분에 써져 있습니다. 그런데, 프로그램을 실행시켜 보았을 때 %d 는 컴퓨터에서 출력되지 않았습니다.

그 대신, %d 가 출력될 자리에 무언가 다른 것이 출력되었는데, 바로 a 의 값 입니다. 따라서,%d 는 **a 의 값 (정확히는 처음 "" 다음에 오는 첫 번째 변수) 을 *10* 진수 로 출력하라** 라는 뜻이 됩니다.

또 다른 예제를 봅시다.

```c
/* 변수 알아보기 2*/
#include <stdio.h>
int main() {
  int a;
  a = 127;
  printf("a 의 값은 %d 진수로 %o 입니다. \n", 8, a);
  printf("a 의 값은 %d 진수로 %d 입니다. \n", 10, a);
  printf("a 의 값은 %d 진수로 %x 입니다. \n", 16, a);
  return 0;
}
```

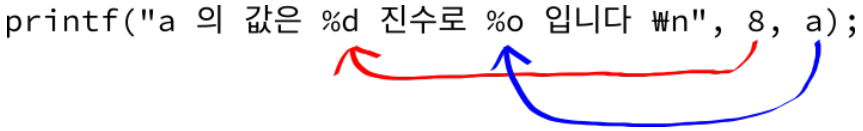
프로그램을 제대로 짰다면 아래와 같은 결과를 볼 수 있을 것입니다.

실행 결과

```
a 의 값은 8 진수로 177 입니다.
a 의 값은 10 진수로 127 입니다.
a 의 값은 16 진수로 7f 입니다.
```

일단, 위 코드를 보고 생기는 궁금증은 2 가지 있습니다. % 달린게 2 개나 있는데, 이를 어떻게 해야되냐와, %d 말고도 %o 와 %x 는 무엇인가 입니다.

먼저, printf 의 작동 원리에 대해 봅시다.



printf 출력시에, 큰 따옴표로 묶인 부분 뒤에 나열된 인자들 (8, a) 가 순서대로 큰 따옴표 안의 % 부분으로 들어감을 알 수 있습니다. 따라서 , 예를들면 printf("%d %d %d %d", a,b,c,d); 와 같은 문장은 a, b, c, d 의 값이 순서대로 출력되겠죠.

이제, %o 와 %x 는 무엇일까요? 이는 인자(a)의 값을 출력하는 형식 입니다. 즉, %o 는 a 의 값을 8 진수로 출력하라라는 뜻이고, %x 는 16 진수로 출력하라는 뜻 이죠.

실수형 변수

앞서 말했듯이, 실수형에는 float 와 double 이 있습니다. double 의 경우 int 형에 비해 덩치가 2 배나 크지만 그 만큼 엄청난 크기의 숫자를 다룰 수 있습니다. 그 대신, 처음 15 개의 숫자들만 정확하고 나머지는 10 의 지수 형태로 표현됩니다. 또한 float 과 double 의 장점은 소수를 표시할 수 있다는 점인데, 정수형 변수에서 소수를 넣는다면 (예를들어 int a; a = 1.234;), 소수 부분은 다 잘린 채, 나중에 a 의 값을 표시해 보면 1 이 나올 것 입니다.

```c
/* 변수 알아보기 3*/
#include <stdio.h>
int main() {
  float a = 3.141592f;
  double b = 3.141592;
  printf("a : %f \n", a);
  printf("b : %f \n", b);
  return 0;
}
```

실행해 본다면 아래와 같이 나오게 됩니다.

실행 결과

```
a : 3.141592
b : 3.141592
```

일단, 위 코드를 보면서 궁금한 점이 생기지 않았나요?

```c
float a = 3.141592f;
double b = 3.141592;
```

왜, `float` 형 변수 `a` 를 선언할 때 에는 숫자 뒤에 `f` 를 붙였는데 `double` 형 에서는 `f` 를 안 붙였는 지요. 왜냐하면, 그냥 `f` 를 안 붙이고 `float a = 3.141592` 로 하면 이를 `double` 형으로 인식하여 문제가 생길 수 있습니다. 따라서, `float` 형이라는 것을 확실히 표시해 주기 위해 `f` 를 끝에 붙이는 것입니다.

```c
printf("a : %f \n", a);
printf("b : %f \n", b);
```

이제, 마지막으로 `%d, %o, %x` 도 아닌 `%f` 가 등장하였습니다. 만약, 여기서 `a` 를 `%d` 형식으로 출력하면 어떻게 될까요? 한 번 해보세요. 아마 이상한 숫자가 나오게 될 것입니다. 왜냐하면 `a` 는 지금 정수형 변수가 아니기 때문 입니다. 설사, 우리가 `a = 3f; b = 3;` 라고 해도, 이미 `a` 와 `b` 를 실수형 변수로 선언하였기 때문에 컴퓨터는`a ,b` 를 절대 정수로 보지 않습니다.

따라서, 우리는 실수형 변수를 출력하는 형식인 `%f` 를 사용해야 합니다.

참고로 주의할 사항은 `printf` 에서 `%f` 를 이용해 수를 출력 할 때 다음과 같이 언제나 소수점을 뒤에 붙여 주어야 한다는 점입니다. 예를 들어서

```c
printf("%f", 1);
```

을 하면 화면에 이상한 값 (아마도 0 이 출력될 것입니다) 이 나오지만

```
printf("%f", 1.0);
```

을 하면 화면에 제대로 1.0 이 출력됩니다.

printf 의 또 다른 형식

```
/* printf 형식 */
#include <stdio.h>
int main() {
  float a = 3.141592f;
  double b = 3.141592;
  int c = 123;
  printf("a : %.2f \n", a);
  printf("c : %5d \n", c);
  printf("b : %6.3f \n", b);
  return 0;
}
```

만약 위 소스를 성공적으로 쳤다면 실행시 아래와 같이 나오게 됩니다.

실행 결과

```
a : 3.14
c :   123
b :  3.142
```

```
printf("a : %.2f \n", a);
```

이번에는 %f 가 아니라 %.2f 로 약간 다릅니다. 그렇다면 .2 가 뜻 하는 것은 무엇일까요? 대충 짐작했듯이, 무조건 소수점 이하 둘째 자리 까지만 표시하라 란 뜻입니다. 따라서, 위의 경우 3.141592 중 3.14 까지만 출력되고 나머지는 잘리게 되죠.

여기서 '무조건' 이라는 것은 %.100f 로 할 경우에도, 3.141592000000....00 을 표시해서 무조건 100 개를 출력하게 합니다.

```
printf("c : %5d \n", c);
```

이번에는 %d 가 아닌 %5d 입니다. 여기서 .5 가 아님을 주의합시다. 이 말은, 숫자의 자리수를 되도록 5 자리로 맞추라는 것입니다. 따라서, 123 을 표시할 때, 5 자리를 맞추어야 하므로 앞에 공백을 남기고 그 뒤에 123 을 표시했습니다.

그런데, 123456 을 표시할 때, %5d 조건을 준다면 어떻할까요? 이 때는 그냥 123456 을 다 표시합니다. 앞서 .?f 는 ? 의 수 만큼 무조건 소수점 자리수를 맞추어야 하지만 이 경우는 반드시 지켜야 되는 것은 아닙니다

```
printf("b : %6.3f \n", b);
```

마지막으로, 위에서 썼던 두 가지 형식을 모두 한꺼번에 적용한 모습입니다. 전체 자리수는 6 자리로 맞추되 반드시 소수점 이하 3 째 자리 까지만 표시한다는 뜻입니다.

변수 작명하기

앞서, 보았듯이 변수를 선언하는 것은 어려운 일이 아닙니다. 단지, 아래의 형태로 맞추어 주기만 하면 됩니다.

```
(변수의 자료형) 변수1, 변수2, .....;
/* 예를 들어 */
int a, b, c, hi;
float d, e, f, bravo;
double g, programming;
long h;
short i;
char j, k, hello, mineral;
```

이 때, 변수 선언시 주의해야 할 점이 있습니다. 만약에 여러분이 오래된 버전의 C 언어 (C89) 를 사용한다면, 변수 선언시 반드시 최상단에 위치해야 합니다. 하지만, 여러분이 지금 사용하고 있는 최신 버전의 C 의 변수 사용하기 전 아무데나 변수를 선언해도 상관 없습니다.

```
/* 변수 선언시 주의해야 할 점 */
#include <stdio.h>
int main() {
  int a;
  a = 1;
  printf("a 는 : %d", a);
  int b;  // 괜찮음!
  return 0;
}
```

두 번째로, 사람의 이름을 지을 때, 여러가지를 고려하듯이 변수의 이름에서도 여러가지 조건들이 있습니다. 아래 예제를 보세요.

```c
/* 변수 선언시 주의해야 할 점 */
#include <stdio.h>
int main() {
  int a, A;  // a 와 A 는 각기 다른 변수 입니다.
  int 1hi;
  // (오류) 숫자가 앞에 위치할 수 없습니다.
  int hi123, h123i, h1234324;  // 숫자가 뒤에 위치하면 괜찮습니다.
  int 한글이좋아;
  /*
  (오류)
  변수는 오직 알파벳, 숫자, 그리고 _ (underscore)로만으로 이루어져야 합니다. */
  int space bar;
  /*
  (오류)
  변수의 이름에는 띄어쓰기하면 안됩니다.  그 대신 _ 로 대체하는 것이 읽기
  좋습니다.*/
  int space_bar;  // 이것은 괜찮습니다.
  int enum, long, double, int;
  /* (오류)
     지금 나열한 이름들은 모두 '예약어' 로 C 언어에서 이미 쓰이고 있는
     것들입니다. 따라서 이러한 것들은 쓰면 안됩니다. 이를 구분하는 방법은
     예약어들을 모두 외우거나 '파란색' 으로 표시된 것들은 모두 예약어라 볼 수
     있습니다  */

  return 0;
}
```

이 안에 모든 내용이 들어 있습니다. 변수의 이름은 반드시

- 숫자가 앞에 위치하면 안됩니다. 그러나 중간이나 뒤는 괜찮습니다.
- 변수명은 오직 영어, 숫자, _ 로 만 구성되어 있어야 합니다.[2)]
- 변수의 이름에 띄어쓰기가 있으면 안됩니다.
- 변수의 이름이 C 언어 예약어 이면 안됩니다. 보통 예약어를 쓰면 에디터에서 다른 색깔로 표시되어 예약어를 썼는지 안썼는지 알 수 있습니다.

또한 C 언어는 대소문자를 구분합니다(이를 영어로 *case sensitive* 하다고 합니다). 따라서, `VARiable` 과 `Variable` 은 다른 변수 입니다. 왠지, 조건이 많아 변수명을 지을 때, 까다로울 것 같지만 그냥 평범하게 짓다보면 예약어와 겹칠일 도 없고, 숫자가 앞에 오는 경우도 별로 없습니다.

2) 사실 비주얼 스튜디오와 같은 컴파일러에서는 유니코드(한글 포함)로 변수 이름을 지어도 괜찮지만 관습상 코드는 모두 영어로 작성하는 것이 맞습니다.

자, 이제 우리는 C 언어에서 중요한 부분인 변수에 대해서 알아보았습니다. 현재 우리는 수를 다루는 변수들만 다루었지만, 다음 강좌에서는 변수에 대한 산술 연산과, 문자를 다루는 변수에 대해 알아보도록 하겠습니다.

뭘 배웠지?

- 변수는 데이터를 임시로 저장하는 곳이며 자유롭게 쓰고 지울 수 있습니다.
- 각 변수에는 형(type) 이 있어서 해당 형에 맞는 데이터를 보관할 수 있습니다.
- 변수의 형으로는 정수값을 보관하는 `char`, `int` 등이 있고, 실수값을 보관하는 `float` 과 `double` 이 있습니다. 각각의 형들은 저장하는 데이터의 크기가 다릅니다.
- `int a = 10;` 의 문장의 의미는 `a` 라는 정수형 변수를 정의한 뒤에, 해당 변수에 10 의 값을 대입한다 라는 뜻입니다.
- 변수의 이름을 정하기 위해서는 여러가지 규칙이 있습니다. 이 규칙에 알맞게 변수의 이름을 정해야 되며 그렇지 않을 경우 컴파일 오류가 발생합니다.

3

계산 하기

안녕하세요 여러분. 지난 강의에서 모두들 변수에 대해 감이 잡혔을 것이라 믿고 강의를 진행하도록 하겠습니다.

최초의 컴퓨터는 무엇을 하기 위해 태어났을까요? 오락용? 영화 시청? (물론 그 때에는 불가능했을 터이지만). 아닙니다. 최초의 컴퓨터라 일컫어 지는 에니악(ENIAC.. 물론 에니악이 최초의 컴퓨터이냐 아니냐에 관한 논쟁은 길다. 한편에서는 콜로서스라는 주장도 있는데 아무튼) 은 포탄을 어떤 각도로 발사했을 때, 어디에 떨어질 지를 예측하는 기계였습니다.

물론 지금도 컴퓨터의 가장 중요한 역할은 인간이 할 수 없는 복잡한 수식을 계산하는 것입니다. 다시말해, 컴퓨터는 **빠른 계산**을 위해 태어난 기계인 것입니다.

산술 연산자, 대입 연산자

이번 강좌에서는 C 언어에서 컴퓨터에 어떻게 연산 명령을 내리는지 살펴보도록 하겠습니다.

일단, '계산' 이라 하면 머리속에 가장 먼저 떠오르는 것은 사칙연산, 즉 +, −, Œ, œ 을 의미합니다. 보통 코딩 시에 Œ 와 œ 기호를 쓰기 힘들기 때문에, 그 대신 `*` 와 `/` 를 사용합니다. 즉, 8 Œ 5 는 `8 * 5` 로 표현하고, 10 œ 7 은 `10 / 7` 로 표현합니다.

또한, 색다른 연산자로 `%` 가 있는데 이는 나눈 나머지를 의미합니다. 예를들어 `10 % 3` 은 1 이 됩니다. 왜냐하면 10 을 3 으로 나눈 나머지가 1 이기 때문이죠. 이러한 `+, -, *, /, %` 를 **산술 연산자(Arithmetic Operator)** 라고 합니다.

```c
/* 산술 연산 */
#include <stdio.h>
int main() {
  int a, b;
  a = 10;
  b = 3;
  printf("a + b 는 : %d \n", a + b);
  printf("a - b 는 : %d \n", a - b);
```

```
  printf("a * b 는 : %d \n", a * b);
  printf("a / b 는 : %d \n", a / b);
  printf("a %% b 는 : %d \n", a % b);
  return 0;
}
```

만약 위 코드를 잘 컴파일 했다면 아래와 같이 나옵니다. (컴파일 하는 방법을 까먹은 사람들은 1강을 참조하세요)

실행 결과

```
a + b 는 : 13
a - b 는 : 7
a * b 는 : 30
a / b 는 : 3
a % b 는 : 1
```

그렇다면 코드를 살펴보도록 합시다.

```
a = 10;
b = 3;
```

지난 강좌를 잘 이해하셧더라면 위 문장이 무슨 역할을 하는지 쉽게 이해하실 수 있을 것입니다. a 라는 변수에 10 을 대입하는 것이고, 두 번째 문장은 b 에 3을 대입하는 것입니다. 그렇다면 아래 문장을 살펴보세요.

```
10 = a;
3 = b;
```

언뜻 보기에 맞는 문장인 것 같습니다. 왜냐하면, 실제 수학을 공부한 사람이라면 `a = 10` 이나 `10 = a` 나 별반 다를 것이 없기 때문이죠. 하지만, C 언어 컴파일러는 '=' 라는 기호를 뒤에서 부터 해석합니다. 즉, `a = 10` 은 *10* 을 *a* 에 대입하라 라는 문장이 되지만, `10 = a` 는' a 의 값을 10 에 대입하라' 라는 이상한 문장이 되서 오류가 발생하게 됩니다.

이렇게 '=' 를 **대입 연산자(Assignment Operator)** 라고 합니다. 왜냐하면 우측의 값을 좌측에 **대입** 하는 것이기 때문이죠.

따라서,

```
a = 5;
b = 5;
```

```
c = 5;
d = 5;
```

라는 문장이나,

```
a = b = c = d = 5;
```

라는 문장은 완전히 같은 것이 됩니다. 왜냐하면, 앞에서 말했듯이 = 는 뒤에서 부터 해석한다고 했으므로, 제일 먼저 d = 5 를 해석한 후, 그 다음에 c = d, b = c, a = b 로 차례대로 해석해 나가기 때문에 a = 5; b = 5; c = 5; d = 5; 라는 문장과 같은 것이지요.

```
printf("a + b 는 : %d \n", a + b);
printf("a - b 는 : %d \n", a - b);
printf("a * b 는 : %d \n", a* b);
printf("a / b 는 : %d \n", a / b);
printf("a %% b 는 : %d \n", a % b);
```

자, 이제 산술 연산자들에 대해 살펴보도록 합시다. 일단, 한 눈에 보게 a + b, a - b, a * b, a / b 는 각각 덧셈, 뺄셈, 곱셈, 나눗셈을 하여서 그 값이 %d 에 들어가 출력된 것 같습니다. 그런데, a + b, a - b, a * b 는 각각 계산 결과가 13, 7, 30 이 나온 사실을 쉽게 받아들일 수 있지만, a / b 가 왜 3 이 나왔는지는 이해하기 힘듭니다. 왜, a / b 가 3 이 되었을까요?

사실, 3 강에서 말했지만 a 와 b 는 모두 int 형으로 선언된 변수 입니다. 즉, a 와 b 는 오직 '정수' 데이터만 담당합니다. 즉, a 와 b 는 모두 정수 데이터만 처리하기 때문에 a 를 b 로 나누면, 즉 10 을 3 으로 나누면 3.3333... 이 되겠지만 정수 부분인 3 만 남기게 되는 것 입니다. 따라서, 값은 3 이 출력됩니다.

마지막으로 생소한 % 라는 연산자에 대해 살펴봅시다. +, -, *, / 연산자는 모두 정수, 실수형 데이터에 대해서 모두 연산이 가능한데, % 는 오직 정수형 데이터에서만 연산이 가능합니다. 왜냐하면 % 는 나눈 나머지를 표시하는 연산자 이기 때문이죠. a % b 는 a 를 b 로 나눈 나머지를 표시합니다. 즉, 10 % 3 = 1 이 됩니다.

이 때,

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

%% 는 % 를 '표시'하기 위한 방법입니다. 왜냐하면 % 하나로는 %d , %f 같이 사용될 수 있기 때문이 표시가 되지 않습니다.

나눗셈 시에 주의할 점

```c
#include <stdio.h>
int main() {
  int a, b;
  a = 10;
  b = 3;
  printf("a / b 는 : %f \n", a / b);  // 해서는 안될 짓
  return 0;
}
```

컴파일 후 (아마 경고 메세지가 뜰 것입니다), 실행한다면 아래와 같이 이상한 결과가 나옵니다.

실행 결과

```
a / b 는 : 0.000000
```

3 강 에서 우리는 %f 가 오직 실수형 데이터 만을 출력하기 위해 있는 것이라 하였습니다. 그런데, a / b 가 10 나누기 3 이므로 3.3333... 이 되서 해서 실수형 데이터가 되는 것이 아닙니다.

(정수형 변수) (연산) (정수형 변수) 는 언제나 (정수) 으로 유지됩니다. 따라서, 실수형 데이터를 출력하는 %f 를 정수형 값 출력에 사용하면 위와 같이 이상한 결과가 나오게 됩니다.

그렇다면 아래의 경우 어떻까요?

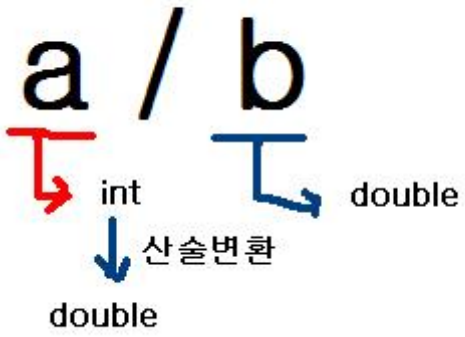
```c
/* 산술 변환  */
#include <stdio.h>
int main() {
  int a;
  double b;

  a = 10;
  b = 3;
  printf("a / b 는 : %f \n", a / b);
  printf("b / a 는 : %f \n", b / a);
  return 0;
}
```

만약 제대로 컴파일 했다면 아래와 같이 나오게 됩니다.

실행 결과

```
a / b 는 : 3.333333
b / a 는 : 0.300000
```

a 는 정수형 변수, b 는 실수형 변수 입니다. 그런데, 이들에 대해 연산을 한 후에 결과를 실수형으로 출력하였는데 정상적으로 나왔습니다. 그 것은 왜 일까요? 이는 컴파일러가 **산술 변환** 이라는 과정을 거치기 때문입니다. 즉, 어떠한 자료형이 다른 두 변수를 연산 할 때, 숫자의 범위가 큰 자료형으로 자료형들이 바뀝니다.



즉, 위 그림에서도 보듯이 a 가 int 형 변수이고 b 가 double 형 변수인데, double 이 int 에 비해 포함하는 숫자가 더 크므로 큰 쪽으로 산술 변환됩니다.

일단, 정수형 변수와 실수형 변수가 만나면 무조건 실수형 변수쪽으로 상승되는데, 이는 실수형 변수의 수 범위가 int 보다 훨씬 넓기 때문입니다. 위와 같은 산술 변환을 통해 애러가 없이 무사히 실행 될 수 있었습니다. 또한 double 형태로 산술 변환 되므로 결과도 double 형태로 나오기 때문에

```
printf(" a / b 는 : %d \n", a / b);
```

와 같이 하면 오류가 생기게 됩니다. 왜냐하면 %d 는 정수형 값을 출력하는 방식이기 때문이죠.

대입 연산자

```
/* 대입 연산자  */
#include <stdio.h>
int main() {
  int a = 3;
  a = a + 3;
  printf("a 의 값은 : %d \n", a);
  return 0;
}
```

위 결과를 컴파일 하면 아래와 같이 나옵니다.

실행 결과

```
a 의 값은 : 6
```

일단, 변수 선언 부분 부터 살펴 봅시다.

```
int a = 3;
```

위 문장을 아마 무슨 뜻일지 감이 바로 오실 것입니다. "음... a 라는 변수를 선언하고 a 변수에 3 의 값을 집어 넣는구나". 맞습니다. 사실 위 문장이나 아래 문장이나 다를 바가 없습니다.

```
int a;
a = 3;
```

그냥, 타이핑 하기 귀찮아서 짧게 써 놓은 것 뿐입니다.

```
a = a + 3;
```

그 다음 부분은 대입 연산자와 산술 연산자가 함께 나와 있습니다. 만일, 우리가 방정식에 대해서 공부해 본 사람이라면 다음과 같이 이의를 제기할 수 도 있습니다.

$a = a + 3$ 따라서 양변에서 a 를 빼면 $0 = 3$???

물론, 위는 수학적으로 맞지만 C 언어 에서 의미하는 바는 다릅니다. 위에서 말했듯이, = 는 등호가 아닙니다. '대입' 연산자 입니다. 무엇을 대입하냐구요? 오른쪽의 값을 왼쪽으로 대입합니다. 즉, a + 3 의 값(6) 을 a 에 대입합니다. 따라서, a = 6 이 되는 것이지요.

이 때, 이와 같이 계산 될 수 있는 이유는 + 를 = 보다 먼저 연산하기 때문입니다. 즉, a + 3 을 먼저 한 후(+), 그 값을 대입(=) 하는 순서를 거치기 때문에 a 에 6 이라는 값이 들어갈 수 있게 됩니다. 이러한 것을 **연산자 우선순위** 라고 하는데, 밑에서 조금 있다가 다루어 보도록 하겠습니다.

```
/* 더하기 1 을 하는 방법 */
#include <stdio.h>
int main() {
  int a = 1, b = 1, c = 1, d = 1;

  a = a + 1;
  printf("a : %d \n", a);
  b += 1;
  printf("b : %d \n", b);
  ++c;
```

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

  return 0;
}
```

위 코드를 컴파일 하면 아래와 같이 나옵니다.

실행 결과

```
a : 2
b : 2
c : 2
d : 2
```

음, 모두 2 가 되었군요. 사실 위에 나온 4 개의 코드는 더하기 1 을 한다는 점에서 모두 같습니다. 일단, 하나하나 차례대로 살펴봅시다.

```
a = a + 1;
```

가장, 기초적으로 1 을 더하는 방법입니다. 위 문장은 "a 에 a 에 1 을 더한 값을 대입한다." 라는 뜻을 가지고 있죠?

```
b += 1;
```

이게 뭔가요! 처음 본 연산인 += 입니다. 이러한 연산을 복합 대입연산 이라 하며, b = b + 1 과 같습니다. 이렇게 쓰는 이유는 단지, b = b + 1 을 쓰기 귀찮아서 간략하게 쓰는 것입니다. 물론, b = b + 1 과 b += 1 은 엄밀히 말하자면 같은 것은 아니지만 이에 대해서는 나중에 다루어 보도록 하겠습니다(우선 순위에서 약간 차이가 있습니다). 복합 대입 연산은 아래와 같이 여러 가지 형태로 이용될 수 있습니다.

```
b += x; // b = b + x; 와 같다
b -= x; // b = b - x;와 같다
b *= x; // b = b * x;와 같다
b /= x; // b = b / x;와 같다
```

마지막으로, 비슷하게 생긴 두 부분을 함께 살펴 보도록 하겠습니다.

```
++c;
d++;
```

위와 같은 연산자(++)를 증감 연산자라고 합니다. 둘 다, c 와 d 를 1 씩 증가시켜 줍니다. 그런데, ++ 의 위치가 다릅니다. 전자의 경우 ++ 이 피연산자(c) 앞에 있지만 후자의 경우 ++ 이 피연산자(d) 뒤에 있습니다.

++ 이 앞에 있는 것을 **전위형(prefix)**, ++ 이 뒤에 있는 것을 **후위형(postfix)** 라 하는데 이 둘은 똑같이 1 을 더해주만 살짝 다릅니다. 전위형의 경우, 먼저 1 을 더해준 후 결과를 돌려주는데 반해, 후위형의 경우 결과를 돌려준 이후 1 을 더해줍니다. 이 말만 가지고는 이해가 잘 안될테니 아래를 보세요.

```c
/* prefix, postfix  */
#include <stdio.h>
int main() {
  int a = 1;

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

  a = 1;
  printf("a++ : %d \n", a++);
  printf("a : %d \n", a);

  return 0;
}
```

위 소스를 성공적으로 컴파일 했다면 아래와 같이 결과가 나온다.

실행 결과

```
++a : 2
a++ : 1
a : 2
```

분명히, 위에서 ++c 나 d++ 이나 결과를 출력했을 때 에는 결과가 1 이 잘 더해져서 2 가 나왔는데 여기서는 왜 다르게 나올까요? 앞서 말했듯이 ++ a 는 먼저 1 을 더한 후 결과를 반환한다고 했고 a++ 은 먼저 결과를 반환 한 후, 그 후에 1 을 더한 다고 했습니다.

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

즉, 위의 경우 a 에 먼저 1 을 더한 값인 2 를 printf 함수에 반환하여 %d 에 2 가 들어가게 됩니다. 그런데,

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

이 경우, 먼저 a 의 값을 printf 에 반환하며 %d 에 1 이란 값이 '먼저' 들어 간 뒤, 1 이 출력된 이후 a 에 1 이 더해집니다. 따라서, 다시 printf 문으로 a 의 값을 출력하였을 때 에는 2 라는 값이 나오게 되는 것입니다. 참고로, 위 4 개의 연산 중에서 가장 빨리 연산되는 것은 a++ 과 같은 증감 연산입니다[1] 하지만, 요즘의 컴파일러는 최적화가 잘 되어 있어, a = a + 1 같은 것은 a++ 로 바꾸어 컴파일 해버립니다.

비트 연산자

마지막으로 **비트 연산자**라고 불리는 생소한 연산자들에 대해 이야기 해보겠습니다. 이 연산자들은 정말 비트(bit) 하나 하나에 대해 연산을 합니다. 비트는 컴퓨터에서 숫자의 최소 단위로 1 비트는 0 혹은 1 을 나타내죠. 쉽게 말해 이진법의 한 자리라 볼 수 있습니다.

보통, 8개의 비트(8 bit) 를 묶어서 1 바이트(byte) 라고 하고, 이진법으로 8 자리 수라 볼 수 있습니다. 따라서, 1 바이트로 나타낼 수 있는 수의 범위가 0 부터 11111111 로 십진수로 바꾸면 0 부터 255 까지 나타낼 수 있습니다.

비트 연산자에는 & (And 연산), | (\위에 있는 것. 영문자 i 의 대문자가 아닙니다. Or 연산), ^ (XOR 연산), <<, >> (쉬프트 연산) , ~ (반전) 등이 있습니다. 일단, 각 연산자가 어떠한 역할을 하는지 살펴보도록 합시다.

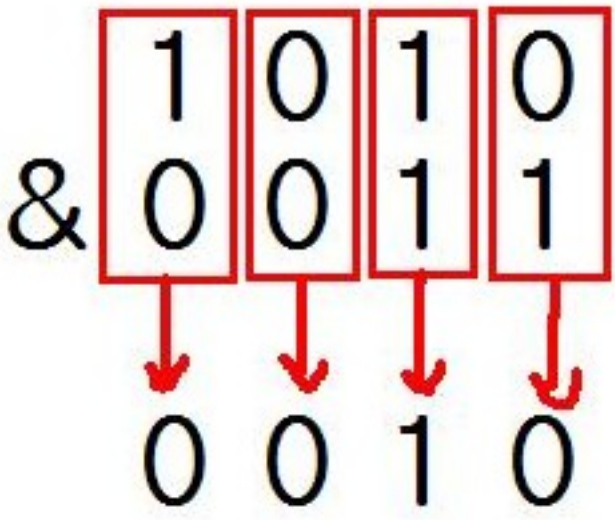
AND 연산 (&)

AND 연산은 아래와 같은 규칙으로 연산됩니다.

		결과
1	1	1
1	0	0
0	1	0
0	0	0

비트 연산은 각 자리를 연산하는데, 예를들어, 1010 & 0011 의 경우

1) 왜냐하면, a = a + 1 의 경우 ADD a 1 로 하지만, a++ 은 INC a 로 좀 더 빨리 계산된다. 자세한 내용은 나중에



위와 같이 한자리 한자리 각각 AND 연산하여, 위에 써 놓은 규칙대로 연산이 됩니다. 만약 두 숫자의 자리수가 맞지 않을 경우, 예를들어 1111100 과 11 을 AND 연산 할 때 에는 11 앞에 0 을 추가하여 자리수를 맞추어 줍니다. 즉, 1111100 과 0000011 의 연산과 같습니다.

OR 연산 (|)

		결과
1	1	1
1	0	1
0	1	1
0	0	0

OR 연산은 AND 연산과 대조적입니다. 어느 하나만 1 이여도 모두 1 이 되는데, 예를들어 `1101 | 1000` 은 결과가 1101 이 됩니다.

XOR 연산 ()

		결과
1	1	0
1	0	1
0	1	1
0	0	0

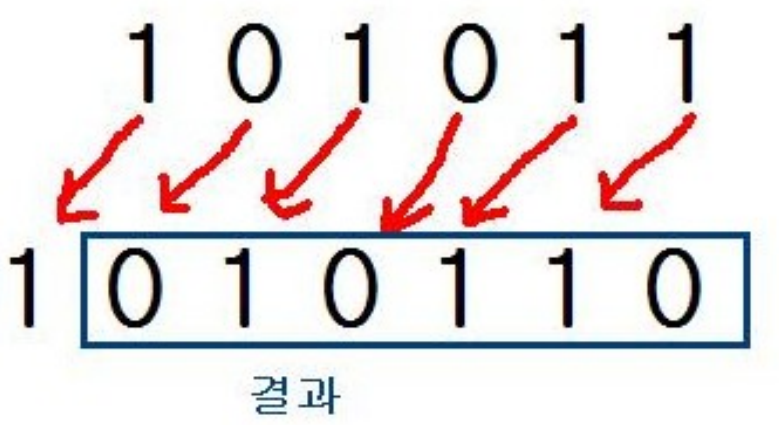
XOR 연산은 특이하게도 두 수가 달라야지만 1 이 됩니다. 예를들어, `1100 ^ 1010` 의 경우 결과가 0110 이 됩니다. 마치 두 비트를 더한 다는 식으로 생각하시면 됩니다.

반전 연산(~)

반전연산은 간단히 말에 0 을 1 로 1 을 0 으로 바꿔주는 것입니다. 예를들어서 `~1100` 을 하면 그 결과는 0011 이 됩니다.

« 연산 (쉬프트 연산)

위 연산 기호에서 느껴지듯이 비트를 왼쪽으로 쉬프트(Shift) 시킵니다. 예를 들어, 101011 를 1 만큼 쉬프트 시키면 (이를 `a << 1` 이라 나타냅니다)



위 처럼 결과가 010110 이 됩니다. 이 때, << 쉬프트 시, 만일 앞에 쉬프트된 숫자가 갈 자리가 없다면, 그 부분은 버려집니다. 또한 뒤에서 새로 채워지는 부분은 앞에서 버려진 숫자가 가는 것이 아니라 무조건 0 으로 채워집니다.

» 연산

이는 위와 같은 종류로 이는 << 와 달리 오른쪽으로 쉬프트 해줍니다. 이 때, 오른쪽으로 쉬프트 하되, 그 숫자가 갈 자리가 없다면 그 숫자는 버려집니다. 이 때, 무조건 0 이 채워지는 << 연산과는 달리 앞부분에 맨 왼쪽에 있었던 수가 채워지게 되죠. 예를들어서 `11100010 >> 3 = 11111100` 이 되고, `00011001 >> 3 = 00000011` 이 됩니다.[2)]

비트 연산자를 자세히 다룬 이유는 이 연산자가 여러분들이 아마 여태까지 살면서 다루어왔던 연산자들 (덧셈, 뺄셈 같은 애들) 과는 조금 다르기 때문입니다. 또한, 처음에 비트 연산자를 접할 때, 저런거 뭐에다 쓰지? 라는 생각이 들기도 합니다. 아직은 그 쓰임새를 짚고 넘어가기 어렵지만 나중에 종종 등장할 때가 있을 테니, 잘 기억해놓으시기 바랍니다.

```
/* 비트 연산 */
#include <stdio.h>
```

2) 참고로 어떤 시스템에서는 » 쉬프트의 경우에도 무조건 맨 왼쪽에 0 이 채워질 수 있습니다.

```c
int main() {
  int a = 0xAF;  // 10101111
  int b = 0xB5;  // 10110101

  printf("%x \n", a & b);   // a & b = 10100101
  printf("%x \n", a | b);   // a | b =10111111
  printf("%x \n", a ^ b);   // a ^ b = 00011010
  printf("%x \n", ~a);      // ~a = 1....1 01010000
  printf("%x \n", a << 2);  // a << 2 = 1010111100
  printf("%x \n", b >> 3);  // b >> 3 = 00010110

  return 0;
}
```

위를 성공적으로 컴파일 했다면

실행 결과

```
a5
bf
1a
ffffff50
2bc
16
```

위와 같이 나오게 됩니다. 일단, 첫 세줄은 그럭 저럭 이해가 잘 갑니다. 그런데, 네 번째 줄인 ~a 연산에 대해 의문을 품는 사람들이 많습니다.

```c
printf("%x \n", ~a);  // ~a = 1....1 01010000
```

우리의 기억을 되돌려 3 강으로 가 봅시다. 강의 중간쯤에 보면 여러가지 자료형 들에 대한 설명과 함께 작은 표가 있을 텐데 말이죠. 이를 다시 아래에 불러와 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)

`int` 형 변수에 대한 설명을 보니 옆에 **Size** 이라 표시된 것이 있습니다. 이는 `int` 형 변수의 크기를 나타내는데 4 바이트라고 되어 있군요. 맞습니다. `int` 형 변수는 하나의 데이터를 저장하기 위하여 메모리 상의 4 바이트 - 즉 32 비트를 사용합니다. (1 byte = 8 bits) 아까, 하나의 비트가 0 과 1 을 나타낸다고 했으므로 (즉 1 개의 비트가 2 진수의 한 자리를 나타내게 되죠), 하나의 `int` 형 변수는 32 자리의 이진수라고 볼 수 있습니다. 예를들어 우리가 `a = 1` 이라 한 것은 실제로 컴퓨터에는 `a = 00000000 00000000 00000000 00000001` 이라 저장되는 것과 같게 되는 거죠.

즉, 우리가 `int a = 0xAF;` 라고 한 것은 `a = 10101111;` 이 맞지만 사실 컴퓨터 메모리 상에서는 `a` 가 `int` 형이기 때문에 `a = 00000000 00000000 00000000 10101111` (`10101111` 앞에 0 이 24 개 있다) 이라 기억하는 것이 됩니다. 따라서, 이 숫자를 반전 시키게 되면`a = 11111111 11111111 11111111 01010000`, 즉 `0xFFFFFF50` 이 되는 것이지요. 마찬가지로 생각해 보면,

```
printf("%x \n", a << 2);  // a << 2 = 1010111100
printf("%x \n", b >> 3);  // b >> 3 = 00010110
```

이 두 문장도 사실은 각각 `00000000 00000000 00000000 10101111` 과 `00000000 00000000 00000000 10110101` 을 쉬프트 연산한 것과 같습니다.

따라서 `a` 의 경우 `00000000 00000000 00000000 10101111` 을 왼쪽으로 2 칸 쉬프트 하면 `00000000 00000000 00000010 10111100` 이 되어 `0x2BC` 가 됩니다

`b` 의 경우 마치 앞에 1 이 있는 것 같지만 실제로는 `int` 형 데이터에 저장되어 있으므로 4 바이트로, `00000000 00000000 00000000 10110101` 이므로 맨 왼쪽의 비트는 0 입니다. 따라서 쉬프트를 하게 되면 왼쪽에 0 이 채워지면서 `00000000 00000000 00000000 00010110` 이 되어 `0x16` 이 됩니다.

복잡한 연산

마지막으로 여러 연산이 중첩된 혼합 연산에 대해 살펴 보도록 합시다. 우리가 연산을 하는데 에도 순서가 있듯이 컴퓨터에도 연산을 하는데 무엇을 먼저 연산을 할 지 우선 순위가 정해져 있을 뿐더러 연산 방향 까지도 정해져 있습니다. 이를 간단히 살펴 보자면 아래와 같습니다.

1	( ) [ ] -> . (expr)++ (expr)--	왼쪽 우선
2	! ~ +-(부호) *p & sizeof 캐스트--(expr) ++(expr)	오른쪽 우선
3	*(곱셈) / %	왼쪽 우선
4	+ -(덧셈, 뺄셈)	왼쪽 우선
5	<< >>	왼쪽 우선
6	< <= > >=	왼쪽 우선
7	== !=	왼쪽 우선
8	&	왼쪽 우선
9	^	왼쪽 우선
10	|	왼쪽 우선
11	&&	왼쪽 우선
12	||	왼쪽 우선
13	? :	오른쪽 우선
14	= 복합대입	오른쪽 우선
15	,	왼쪽 우선

이와 같이 순위가 매겨져 있습니다. 이 때, 눈여겨 보아야 할 점은 괄호들이 제 1 우선 순위에 위치하였다는 점 입니다. 따라서, 어떠한 연산이라도 괄호로 감싸 주게 되면 먼저 실행 됩니다.

마지막으로, 결합 순위에 대해 잠시 다루어 보도록 하겠습니다. 표의 오른쪽을 보면 결합 순위가 나와 있는데, 대부분이 '왼쪽 우선' 이지만 몇 개는 '오른쪽 우선' 입니다. 이 말이 뜻하는 바가 무엇이냐면, 아래와 같은 문장을 수행할 때 계산하는 순위를 이야기 합니다.

```
a = b + c + d + e;
```

위 표에서 보듯이, 덧셈의 결합 순서가 왼쪽 우선이므로 위 계산과정은 아래의 순서대로 진행됩니다.

1. `b + c` 를 계산하고 그 결과를 반환(그 결과를 `C` 라 하면)
2. `C + d` 를 계산하고 그 결과를 반환(그 결과를 `D` 라 하면)
3. `D + e` 를 계산하고 그 결과를 반환(그 결과를 `E` 라 하면)

따라서, 위 식은

```
a = E
```

가 되죠. 따라서, a 에 E 의 값, 즉 b + c + d + e 의 값이 들어가게 됩니다.

또한, 위 표에서 몇 안되는 '오른쪽이 우선' 인 대입 연산자(=) 를 살펴봅시다. 만약 대입 연산자가 왼쪽 우선이였다면 아래의 식이 어떻게 계산될 지 생각해 봅시다.

```
a = b = c = d = 3;
```

만약 왼쪽 우선이였다면 a = b; b = c; c = d; d = 3 의 형식으로 계산되어 a, b, c 에는 알 수 없는 값이 들어가겠죠. 하지만 오른쪽이 우선이므로 위 식은 d = 3, c = d, b = c, a = b 의 형식으로 계산되어 a,b,c,d 의 값이 모두 3 이 될 수 있었습니다.

자, 이제 연산자에 대한 강의가 끝났습니다. 연산자는 C 언어에서 가장 기초적인 부분이라 할 수 있습니다. 마치 수학에서 덧셈, 뺄셈을 가장 처음에 배우는 것 처럼 말이죠.

이번 강좌에서는 특별히 예제를 많이 만들어 보지는 않았지만 여러분 께서 C 언어를 통해 복잡한 수식의 계산을 하거나, 복잡한 수식을 보고 이러한 연산은 이 순서로 연산될 것이다 라고 예측해 보는 것도 우선순위를 이해하는데 도움이 될 것입니다. 보통, 우선순위를 잘못 고려하여 나는 오류들은 찾기가 매우 힘들기 때문에 **애초에 헷갈릴 만한 부분은 괄호를 통해 확실하게 하는 것이 좋습니다**.

뭘 배웠지?

- +, -, /, *, =, % 가 무슨 역할을 하는 연산자 인지 배웠습니다.
- &, |, <<, >> 가 무슨 역할을 하는 연산자 인지 배웠습니다.
- 연산자 우선 순위에 대해 다루었습니다.
- 우선 순위가 헷갈리거나, 복잡한 수식이면 괄호를 적극 활용합니다.