Basic C Programming

Variable & Operator

4th Week

Goal

1. 변수에 대한 이해
2. 연산자에 대한 이해
3. 형변환에 대한 이해
4. 잘못된 연산에 대한 이해

학습 목표

목차

변수의 이해

연산자에 대한 이해

형변환에 대한 이해

오버플로우와 언더플로우에 대한 이해

변수

기초 사용법

변수란 값을 담는 공간

변수를 사용하기 위해선?

변수를 사용하겠다는 **선언(정의)**이 필요

변수의 선언이 끝나면 사용 가능

변수를 선언할 때 변수 이름은 유일해야 함

이미 선언한 이름으로 변수를 또 선언할 수 없음

변수의 선언(정의)

변수 이름

```
int value = 10;
```

변수의 자료형 변수의 값

정수형의 변수 value는 10의 값을 가진다

변수의 대입(할당)

변수 이름

```
value = 20;
```

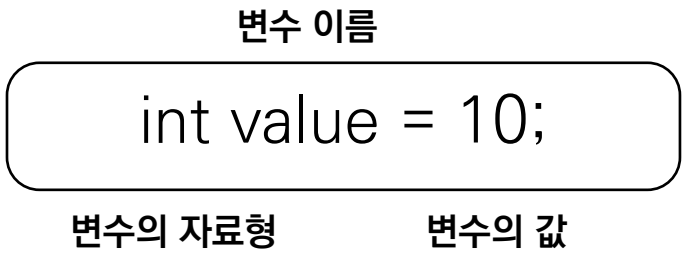
변수의 값

(정수형의) 변수 value는 20의 값을 가진다

변수

기초 사용법

예시




```c
int main() {
    int value = 10;

    int year = 2024;

    int month = 3;

    month = 4;

    int month = 5;  // Error (중복된 이름)

    int day = 1  // Error (세미콜론 없음)

    return 0;
}
```

변수

기초 사용법

변수를 선언할 때 변수의 값은 필수가 아님

변수 이름

```
int value = 10;
```

변수의 자료형 변수의 값

하지만 그 경우 변수에 어떤 값이 들어갈 진 모름

변수 이름

```
int value;
```

변수의 자료형

그러니 항상 변수를 선언할 때 값을 넣어주자
(변수 선언 시 초기화)

```c
int main() {
    int value;

    int year;

    int month;

    month = 4;

    int month;  // Error (중복된 이름)

    int day  // Error (세미콜론 없음)

    return 0;
}
```

변수

기초 사용법

변수란 값을 담는 공간

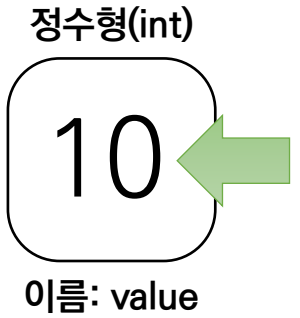
공간에 값을 넣을 수 있으니

공간으로부터 값을 꺼내 쓸 수도 있어야 맞다

변수 이름

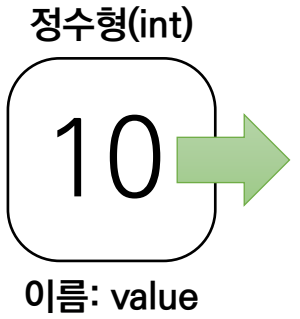
```
int value = 10;
```

변수의 자료형 변수의 값



변수 이름

```
value
```



변수의 값

변수 이름 변수 이름 값

```
value = value + 10;
```

= 10

이 때 value가 10으로 바뀌는 것을
value가 평가(evaluated)되었다고 표현

변수

기초 사용법

예시

```c
int main() {
    int value = 10; // 10

    value = 1 + 2; // 3

    int sum = 1 + 2 - 3 + 0; // 0

    sum = value + 10; // 3 + 10 = 13

    sum = value + value; // 3 + 3 = 6

    sum = sum + sum; // 6 + 6 = 12

    return 0;
}
```

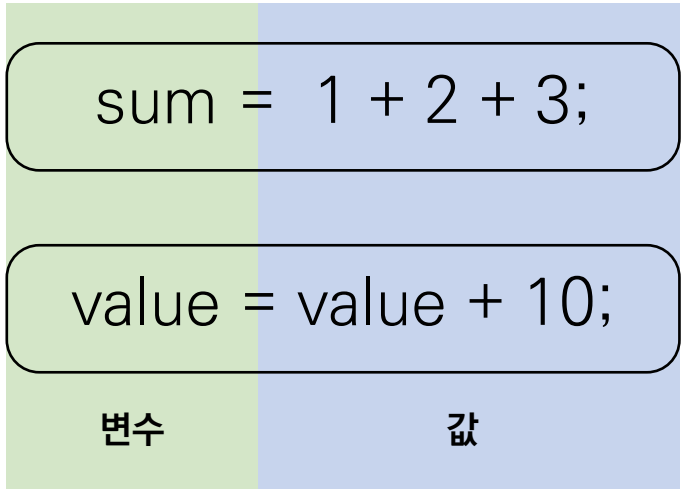
변수

기초 사용법

변수 할당의 해석

- 등호 좌측엔 변수
- 등호 우측엔 값

등호 우측에 무엇이 위치하더라도 결국에는 값으로 평가(evaluated)됨



변수

자료형과 변수

변수는 자료형을 가진다

정수형 변수
실수형 변수
문자형 변수
…

자료형 (Revisit)

요약

자료형	유형	크기	최솟값	최댓값
signed char	문자	1B	-128	127
unsigned char	문자	1B	0	255
signed short	정수	2B	-32,768	32,767
unsigned short	정수	2B	0	65,535
signed int	정수	4B(32-bit) or 8B(64-bit)	-21억(32-bit) or -922경(64-bit)	21억(32-bit) or 922경(64-bit)
unsigned int	정수	4B(32-bit) or 8B(64-bit)	0	42억(32-bit) or 1,844경(64-bit)
signed long	정수	4B(32-bit) or 8B(64-bit)	-21억(32-bit) or -922경(64-bit)	21억(32-bit) or 922경(64-bit)
unsigned long	정수	4B(32-bit) or 8B(64-bit)	0	42억(32-bit) or 1,844경(64-bit)
signed long long	정수	8B	-922경	922경
unsigned long long	정수	8B	0	1,844경
float	실수	4B	6~7자리 십진 유효숫자	
double	실수	8B	15~16자리 십진 유효숫자	
long double	실수	8B or 10B or 16B		

변수

자료형과 변수

- 정수
- 실수
- 문자
- 문자열

변수

자료형과 변수

- 정수
- 실수
- 문자
- 문자열

```c
int main() {
    int normal_value = 10;

    signed int signed_value = 20;

    unsigned int unsigned_value = 30;

    long long normal_long_long = 10;

    signed long long signed_long_long = 20;

    unsigned long long unsigned_llong = 30;

    return 0;
}
```

변수

자료형과 변수

- 정수
- 실수
- 문자
- 문자열

```c
int main() {
    float float_value = 1.1;

    double double_value = 2.2;

    long double long_double_value = 3.3;

    return 0;
}
```

변수

자료형과 변수

정수

실수

문자

문자열

```c
int main() {
    char character = '1';

    signed char character = '2';

    unsigned char character = '3';

    return 0;
}
```

변수

자료형과 변수

정수

실수

문자

문자열

```
int main() {
    char* string = "abcde";

    return 0;
}
```

연산자

연산의 종류

- 산술 연산
- 비교/관계 연산
- 논리 연산
- 비트 연산
- 복합 할당 연산
- 멤버 및 포인터 연산
- 기타 연산

연산자

1. 산술 연산

연산을 한 후 변수에 저장하지 않으면?

연산 결과가 사라진다

정수에 대한 나누기 연산은 실수 결과를 만든다

모듈러 연산은 나머지를 반환한다

전위/후위 증감 연산은

변수의 값을 1만큼 증가, 감소시킨 후
다시 변수에 값을 저장하는 연산이다

그런데 전위와 후위를 나누는 이유가 뭘까?

```c
int main() {
    int a = 10;
    int b = 3;

    a + b;  // 13
    a - b;  // 7
    a * b;  // 30
    a / b;  // 3.333... (주의!)
    a % b;  // 1 (모듈러/나머지 연산)

    a = a + 1;  // 11

    a++;  // 12, 후위 증가연산
    a--;  // 11, 후위 감소연산

    ++a;  // 12, 전위 증가연산
    --a;  // 11, 전위 감소연산

    return 0;
}
```

연산자

1. 산술 연산

전위/후위 증감 연산

++a와 --a는 전위 연산

전위 연산은 증감을 수행한 후 결과를 반환

a = ++a의 해석

++a를 실행하여 a의 값이 1 증가 (11)

수정된 a의 값을 반환하여 a = 11이 됨

a++와 a—는 후위 연산

후위 연산을 결과를 먼저 반환하고 증감을 수행

a = a++의 해석

a++를 실행하여 a의 값이 1 증가 (11)

수정되기 전 a의 값을 반환하여 a = 10이 됨

```c
int main() {
    int a = 10;
    a = ++a;  // 11

    a = 10;
    a = --a;  // 9

    a = 10;
    a = a++;  // 10

    a = 10;
    a = a--;  // 10

    return 0;
}
```

연산자

1. 산술 연산

악랄한 전위/후위 증감 연산

코드를 절대 이렇게 작성하지 말자

좋은 코드는 효율적인 코드가 아니라
제3자가 쉽게 이해할 수 있는 코드

효율성을 따져야 하는 상황이 아니라면
이런 코드는 무척 나쁜 코드이다

```c
int main() {
    int a = 10;
    a = ++a + ++a;  // 12 + 12 = 24

    a = 10;
    a = --a + --a;  // 8 + 8 = 16

    a = 10;
    a = a++ + a++;  // 10 + 11 = 21

    a = 10;
    a = a-- + a--;  // 10 + 9 = 19

    return 0;
}
```

연산자

2. 비교/관계 연산

대입 연산(=)과 동등 비교 연산(==) 혼동 주의!

논리값(Boolean)을 표현할 때
- C에서는 일반적으로 1을 True, 0을 False로 표현

대입 연산자(=)가 복합되어 있는 경우
- 대입 연산자를 가장 마지막에 쓴다
- 그래야 컴퓨터가 헷갈리지 않음

=!가 아니라 !=
=〉, =〈 가 아니라 〉=, 〈=

```c
int main() {
    int a = 10;
    int b = 20;

    a == b;  // False (0)

    a != b;  // True (1)

    a > b;  // False (0)

    a < b;  // True (1)

    a >= b;  // False (0)

    a <= b;  // True (1)

    return 0;
}
```

연산자

3. 논리 연산

논리값(Boolean)을 표현할 때

C에서는 일반적으로 1을 True, 0을 False로 표현

논리 연산(True, False)에서는 &&, ||를 사용

&, |가 아니니 혼동 주의

논리학

참의 반대 (not) -> 거짓
거짓의 반대 (not) -> 참
참이면서 거짓 (and) -> 거짓
참이거나 거짓 (or) -> 참

```c
int main() {
    int a = 1;
    int b = 0;

    !a;  // True의 반대 = False

    !b;  // False의 반대 = True

    a && b;  // True and False = False

    a || b;  // True or False = True

    return 0;
}
```

연산자

4. 비트 연산

비트 연산은 정수형 자료를 이진수로 전제함
  비트(이진) 연산은 이진수에서만 가능하기 때문

비트 연산(0, 1)에서는 &와 |를 사용
  &&, ||가 아니니 혼동 주의

비트 연산
  1의 반대 (not) -> 0 (False)
  1이고 0 (and) -> 0 (False)
  1이거나 0 (or) -> 1 (True)
  1과 0은 다르다 (xor) -> 1 (True)
  1과 1은 다르다 (xor) -> 0 (False)

  비트 시프트: 좌/우로 비트를 한 칸 씩 밀어내는 것

```
int main() {
    int a = 1;
    int b = 0;

    // 이해가 쉽게 이진수를 괄호로 표기함

    ~a;  // not (1) = 0 (0)

    a & b;  // (1) and (0) = 0 (0)

    a | b;  // (1) or (0) = 1 (1)

    a ^ b;  // (1) xor (0) = 1 (1)

    a << 1;  // (1) << 1 = 2 (10)

    a >> 1;  // (1) >> 1 = 0 (0)

    return 0;
}
```

연산자

4. 비트 연산

비트 연산은 정수형 자료를 이진수로 전제함
  비트(이진) 연산은 이진수에서만 가능하기 때문

비트 연산(0, 1)에서는 &와 |를 사용
  &&, ||가 아니니 혼동 주의

비트 연산
  1의 반대 (not) -> 0 (False)
  1이고 0 (and) -> 0 (False)
  1이거나 0 (or) -> 1 (True)
  1과 0은 다르다 (택) -> 1 (True)
  1과 1은 다르다 (택) -> 0 (False)

  비트 시프트: 좌/우로 비트를 한 칸씩 밀어내는 것

```
int main() {
    int a = 3;  // 11
    int b = 2;  // 10

    // 이해가 쉽게 이진수를 괄호로 표기함

    ~a;  // not (11) = 0 (00)

    a & b;  // (11) and (10) = 2 (10)

    a | b;  // (11) or (10) = 3 (11)

    a ^ b;  // (11) xor (10) = 1 (01)

    a << 1;  // (11) << 1 = 6 (110)

    a >> 1;  // (11) >> 1 = 1 (1)

    return 0;
}
```

연산자

5. 복합 할당 연산

기존에 있었던 연산들을 할당 연산(=)과 복합

대입 연산자(=)가 복합되어 있는 경우
- 대입 연산자를 가장 마지막에 쓴다
- 그래야 컴퓨터가 헷갈리지 않음

=!가 아니라 !=
=>, =< 가 아니라 >=, <=

정수의 나눗셈은 항상 실수이니 조심할 것

```
int main() {
    int a = 3;
    int b = 2;

    a += b;  // 덧셈
    a -= b;  // 뺄셈
    a *= b;  // 곱셈
    a /= b;  // 나눗셈 (주의!)
    a %= b;  // 나머지 (모듈러)

    a &= b;  // and 비트 연산
    a |= b;  // or 비트 연산
    a ^= b;  // xor 비트 연산

    a <<= b;  // 왼쪽 비트시프트 연산
    a >>= b;  // 오른쪽 비트시프트 연산

    return 0;
}
```

연산자

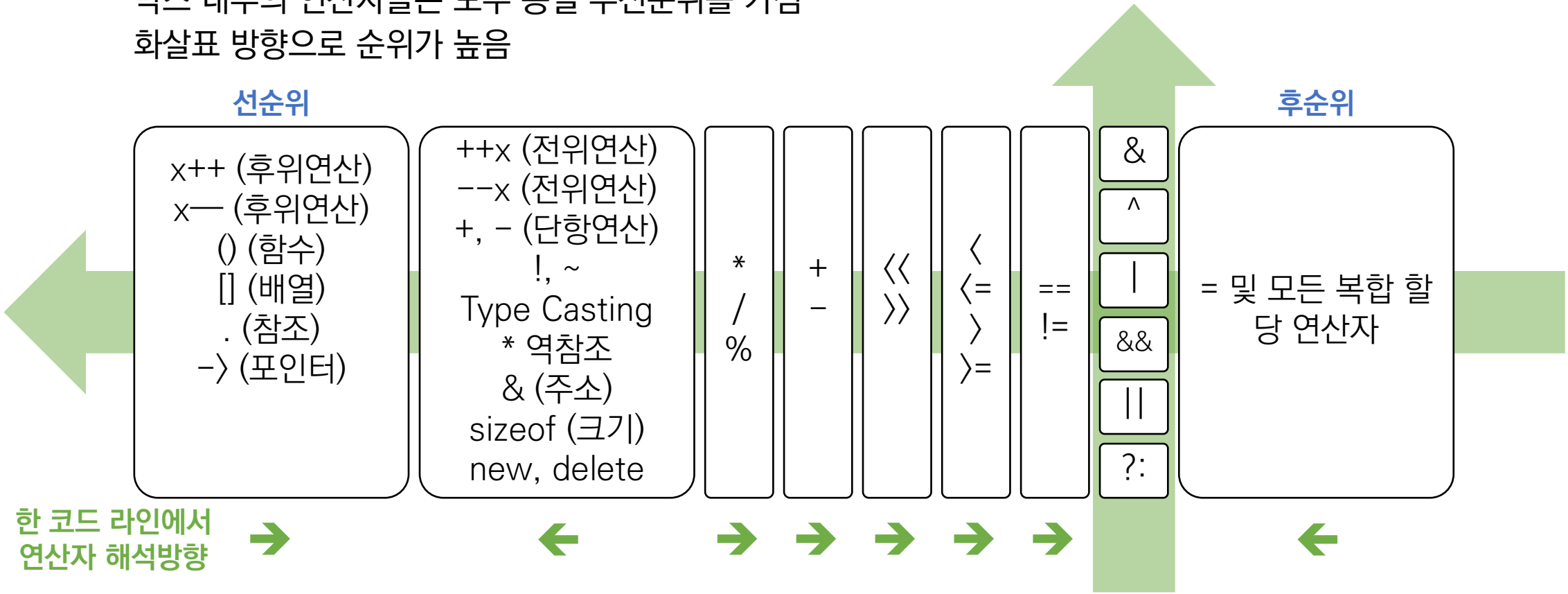
우선순위

후위연산, 전위연산 순으로 빠르고
산술, 비교, 비트, 논리 순으로 빠르고
할당이 가장 느리다

연산에는 우선순위가 존재

박스 내부의 연산자들은 모두 동일 우선순위를 가짐

화살표 방향으로 순위가 높음



연산자

우선순위

연산에는 우선순위가 존재

- 좋은 코드는 이해하기 쉬운 코드
- 증감 연산자를 남발하지 말자

```
int main() {
    int value = (2*1 + 1) + (2 - 1*3)*3;  // 0
    int comparison = (value > 10) && (value <= 0);  // 0

    value = value++;  // 0
    value += value++;  // 1

    return 0;
}
```

형변환

형변환이란

어떤 자료의 자료형을 다른 자료형으로 변환하는 것 (Type Casting)

자료형을 변환하는 이유?

1. 개발자의 의도

2. 자료형이 서로 다른 자료를 연산하고 싶을 때

자료형을 의도적으로 변환할 때 발생하는 **강제성** 형변환

명시적 형변환 (Explicit Type Casting)

자료형을 의도적으로 변환하지 않고 자료형이 서로 다른 자료끼리 연산할 때 **자연스럽게 발생**하는 형변환

암묵적 형변환 (Implicit Type Casting)

형변환

형변환이란

형변환의 방향

자료형 간 형변환은 항상 가능

주의할 점

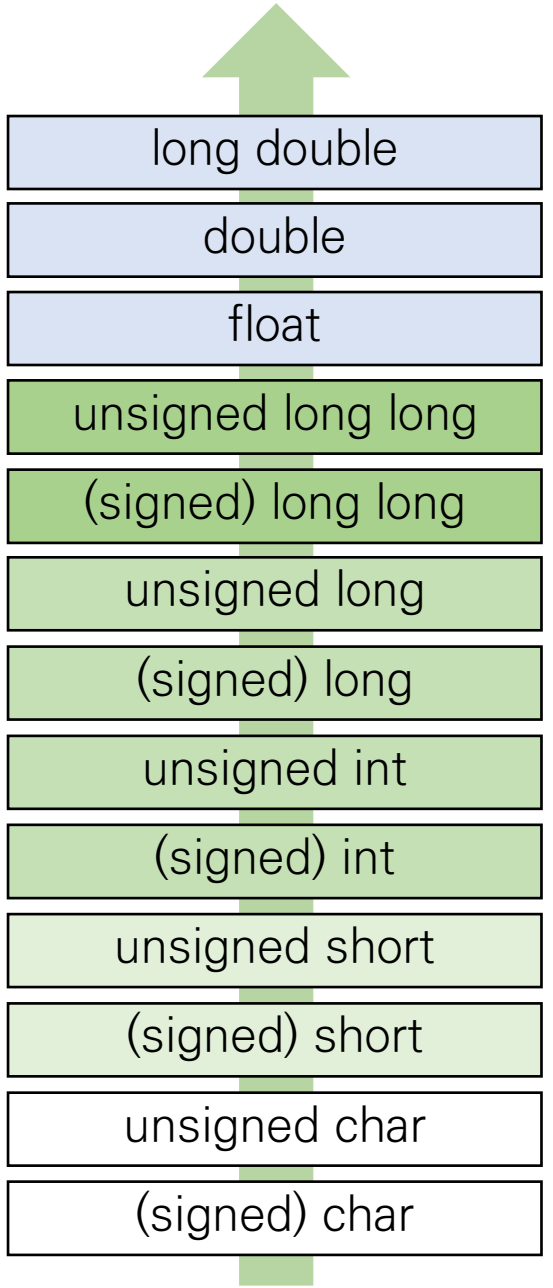
하위의 자료형에서 상위의 자료형으로 형변환될 때는 큰 문제가 발생하지 않음

하지만 signed -> unsigned는 주의할 것

상위의 자료형에서 하위의 자료형으로 형변환될 때는 큰 문제가 발생

데이터 유실 가능성 존재

e.g. 3.14를 정수형으로 형변환하면 3이 됨



형변환

명시적 형변환

명시적으로 형변환하려면 값 앞에 (자료형)을 사용
변수도 값이 될 수 있다는 점을 기억하자

```c
int main() {
    int iValue = 1;
    unsigned int uiValue = 10;

    uiValue = (unsigned int)iValue;  // 10

    char cValue = 'A';  // 'A' = 65
    iValue = (int)cValue;  // 65

    float fValue = 0.0;
    fValue = (float)iValue;  // 65.0

    // 아래 형변환에는 손실이 발생

    fValue = 3.14;
    iValue = (int)fValue;  // 3

    long long int llValue = 10000000000;
    iValue = (int)llValue;  // 1410065408

    return 0;
}
```

형변환

암묵적 형변환

암묵적 형변환에는 아무런 명시도 필요하지 않음
- 서로 다른 자료형 간 연산 과정에서 발생

모든 데이터는 숫자에 불과함
- 문자, 정수, 실수 모두 0101덩어리
- 그렇기에 서로 말도 안되는 형변환이 가능한 것

하지만 암묵적 형변환은 바람직하지 않음
- 개발자가 실수할 가능성을 높이는 안 좋은 습관

```
int main() {
    char cValue = 65;  // 'A'
    int iValue = 'A';  // 65
    long long llValue = 1.5;  // 1
    float fValue = 10;  // 10
    double dValue = 'A';  // 65

    iValue = 'A' + 1;  // 65
    iValue = 1 + 1.5;  // 2
    fValue = 1 + iValue;  // 3.0
    dValue = fValue + cValue;  // 68.0

    return 0;
}
```

오버/언더플로우

정의

오버플로우

자료형이 표현할 수 있는 범위를 벗어난 수를 처리할 때 발생하는 현상

e.g. 너무 큰 수, 너무 작은 수

언더플로우

자료형이 표현할 수 있는 범위를 벗어난 정밀도를 처리할 때 발생하는 현상

e.g.무한소수, 0은 아니지만 0에 매우 근사한 수

자료형 (Revisit)

요약

자료형	유형	크기	최솟값	최댓값
signed char	문자	1B	-128	127
unsigned char	문자	1B	0	255
signed short	정수	2B	-32,768	32,767
unsigned short	정수	2B	0	65,535
signed int	정수	4B(32-bit) or 8B(64-bit)	-21억(32-bit) or -922경(64-bit)	21억(32-bit) or 922경(64-bit)
unsigned int	정수	4B(32-bit) or 8B(64-bit)	0	42억(32-bit) or 1,844경(64-bit)
signed long	정수	4B(32-bit) or 8B(64-bit)	-21억(32-bit) or -922경(64-bit)	21억(32-bit) or 922경(64-bit)
unsigned long	정수	4B(32-bit) or 8B(64-bit)	0	42억(32-bit) or 1,844경(64-bit)
signed long long	정수	8B	-922경	922경
unsigned long long	정수	8B	0	1,844경
float	실수	4B	6~7자리 십진 유효숫자	
double	실수	8B	15~16자리 십진 유효숫자	
long double	실수	8B or 10B or 16B		

오버/언더플로우

오버플로우 예시

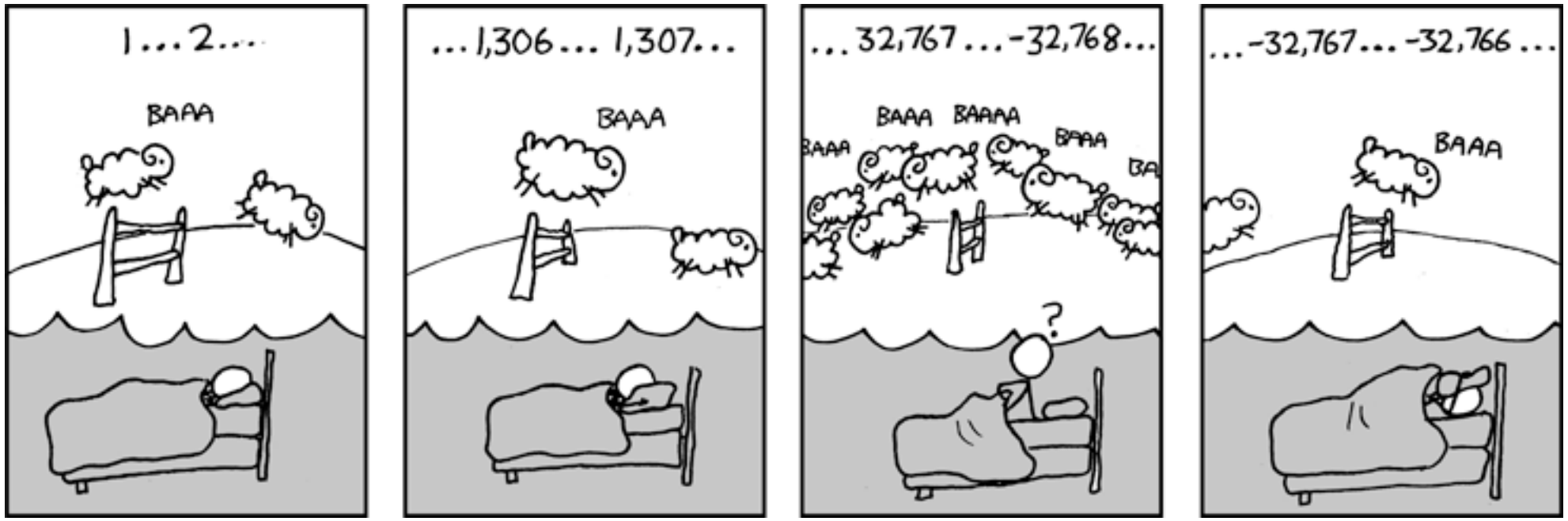


figure from https://xkcd.com/571/

오버/언더플로우

오버플로우 예시

cf. int형이 표현할 수 있는 최솟값은 -2,147,483,648이다

오버/언더플로우

코드 예시

1e-100은 10의 -100제곱을 의미

초보 개발자들이 흔히 하는 실수

지금은 기초 단계이니 개념만 보고 넘어가나

향후에는 이런 일이 일어나지 않게 코딩해야 함

이러한 안전한 코딩 방법론을 **시큐어 코딩**이라 함

```
int main() {
    // 오버플로우 예시
    char cValue = 10000;  // 16
    int iValue = 10000000000;  // 1410065408

    // 언더플로우 예시
    float fValue = 1e-100;  // 0.000...

    return 0;
}
```

변수

과제 01

signed int 변수를 화면에 출력하는 방법

printf(“%d\n”, value);

unsigned int 변수를 화면에 출력하는 방법

printf(“%u\n”, value);

```c
int main() {
    signed int value = 10;
    printf("%d\n", value);

    unsigned int value = 10;
    printf("%u\n", value);

    return 0;
}
```

변수

과제 01

signed int 변수를 세 개 만들어 서로 곱해보고 결과 출력하기
signed int 변수를 크게 만들어 오버플로우 시켜서 결과 출력하기
signed int 변수를 작게 만들어 오버플로우 시켜서 결과 출력하기

unsigned int 변수를 세 개 만들어 서로 곱해보고 결과 출력하기
unsigned int 변수를 크게 만들어 오버플로우 시켜서 결과 출력하기
unsigned int 변수를 작게 만들어 오버플로우 시켜서 결과 출력하기