Expressions

adopted from KNK C Programming : A Modern Approach

Operators 연산자

- C 는 표현식을 많이 사용함
- 표현식은 변수와 상수와 연산자로 구성됨
- C 에는 연산자의 종류가 다양함
  1. arithmetic operators (수식 연산자)
  2. relational operators (관계 연산자)
  3. logical operators (논리 연산자)
  4. assignment operators (할당 연산자)
  5. increment and decrement operators (증감 연산자)

  그 외에도 더 있음

Arithmetic Operators 수식 연산자

- C 이항 수식 연산자 ***arithmetic operators:***

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	remainder

An operator is ***binary*** if it has two operands.

Ex: A*B, A+B

- 단항 수식 연산자:

`+`	unary plus
`-`	unary minus

```
i = +1;
j = -i;
```
음수 양수 구분용

Binary Arithmetic Operators

- `i % j` 의 값은 `i` 를 `j`로 나눈 나머지.

  `10 % 3` 의 값은 1, `12 % 4` 는 0.

- 이항 수식 연산자는 `%`를 제외하면 정수와 실수 피연산자를 혼용해서 쓸 수 있음

- `int` 와 `float` 형 피연산자가 혼용되면 결과는 `float`.

  `9 + 2.5f` 의 결과는 11.5, `6.7f / 2` 는 3.35.

/ 와 % 연산자

- / 와 % 는 주의를 요하는 연산자:
  - 두 피연산자가 정수면, / 소수점 이하는 버림. 1 / 2 의 결과는 0이다. 0.5가 아님.
  - % 연산자는 정수만 피연산자로 받음; 어느 하나라도 정수가 아니면 컴파일이 안됨
  - 우측 피연산자로 0을 쓸 경우 / 또는 % 는 정의되지 않은 동작을 함

❖ / 과 % 에 음수 피연산자가 사용될 경우 C89의 경우 ***implementation-defined(구현에 따라 다름)***.

❖ C99에서는, 나눗셈의 결과는 항상 0 방향으로 내림, 그리고 i % j의 결과는 i의 부호와 같음.

Operator Precedence (연산자 우선순위)

- `i + j * k` 의 의미가 "`i` 와 `j`를 더하고, 그 결과를 `k`로 곱하라" 일까 "`j` 와 `k`를 곱하고, `i`를 더하라"일까?
- 명확히 하는 방법은 `(i + j) * k` 또는 `i + (j * k)` 처럼 괄호로 묶는 것.
- 괄호가 없으면, C 는 연산자 우선 순위 ***operator precedence*** 규칙에 따라 처리함
- 우선순위를 모를 때는 괄호를 써서 먼저 계산한 것을 표시!

Operator Precedence

- 수식 연산자의 우선 순위는 간략하게 다음과 같음:

  Highest: `+ -` (unary)
  `* / %`
  Lowest: `+ -` (binary)

- Examples:

  `i + j * k` 와 `i + (j * k)`는 동치

  `-i * -j` 와 `(-i) * (-j)` 는 동치

  `+i + j / k` 와 `(+i) + (j / k)` 는 동치

Operator Associativity (연산자 결합)

- ***Associativity*** 결합은 동일한 우선순위의 연산자들이 포함된 수식에서 적용됨
- 연산자들의 결합 방향이 왼쪽에서 오른쪽으로 진행된다면 ***left associative 왼쪽 결합***
- 이항 수식 연산자 (*, /, %, +, –) 는 left associative

  `i – j – k` 와 `(i – j) – k` 는 동치

  `i * j / k` 와 `(i * j) / k` 는 동치
- 연산의 결합 방향이 오른쪽에서 왼쪽이면 ***right associative*** 오른쪽 결합.
- 단항 수식 연산자 (+ , –) 는 right associative

  `– + i` 와 `– (+i)` 는 동치

Assignment Operators 할당 연산자

1. ***Simple assignment 단순 할당:*** 변수에 값 저장에 사용

2. ***Compound assignment 합성 할당:*** 변수에 저장된 값을 갱신할 때 사용

Simple Assignment 단순 할당

- ***v* = *e*** 은**e의 표현식의 값을 v로 복사**하라는 말과 같음
- ***e*** 상수일 수도 또는 복잡합 표현식일 수 있음:

```
i = 5;                /* i is now 5  */
j = i;                /* j is now 5  */
k = 10 * i + j;       /* k is now 55 */
```

- 만약 ***v*** 와 ***e*** 가 같은 형이 아니면 **e의 형은 v의 형으로 변환됨**

```
int i;
float f;

i = 72.99f;    /* i is now 72 */
f = 136;       /* f is now 136.0 */
```

Side Effects

- 피연산자의 값을 변형시키는 연산자를 ***side effect***가 있다고 함
- 단순 할당 연산자는 side effect가 있음: 좌변의 변수를 변경함
- `i = 0` 를 분석해보면 우변의 결과는 0이 됨. 그리고 side effect로 인해 `i`에 `0`을 저장함.
- 할당은 연산자이기 때문에 여러 할당 연산이 한 줄에 사용될 수 있음:

```
i = j = k = 0;
```

- `=` 는 오른쪽 결합이기 때문에 아래와 동치임

```
i = (j = (k = 0));
```

Side Effects

- 그러나 한 줄에 할당을 여러 번 할 경우 값이 변환되는 경우를 주의해야 함:

```
int i;
float f;

f = i = 33.3f;
```

`i` 는 33를 저장,
`f`는33.0 (not 33.3)을 저장.

Side Effects

- *v* = *e* 형태의 할당은 v의 타입이 허용되는 모든 경우에 대해 사용할 수 있음:

```
i = 1;
k = 1 + (j = i); // 수식 내에 할당을 포함할 경우
                 // 읽기 어렵고, 버그의 온상임
printf("%d %d %d\n", i, j, k);
  /* prints "1 1 2" */
```

Lvalues

- 할당 연산자는 왼쪽 피연산자로 ***lvalue*** 를 필요로 함.
- **lvalue** 는 **컴퓨터 메모리에 저장된 객체**로서 상수도 계산 결과도 아님.
- 변수가 lvalues 임; `10` 또는 `2 * i` 는 아님.

```
12 = i;          /*** WRONG ***/
i + j = 0;       /*** WRONG ***/
-i = j;          /*** WRONG ***/
```

- 컴파일러는 이런 경우 *"invalid lvalue in assignment."*라는 오류 메시지를 출력함

Compound Assignment 합성 할당

- 변수의 과거의 값을 사용하여 새로운 값을 다시 원래의 변수에 저장하는 경우는 매우 흔함
- Example:

```
i = i + 2;
```

- += 합성 할당 연산자로 간단하게 표현할 수 있음:

```
i += 2;    /* same as i = i + 2; */
```

Compound Assignment

- 합성 할당 연산자는 소개한 것 외에도 9개가 더 있음. 그 중 일부는: `-=` `*=` `/=` `%=`

  모든 합성 할당 연산자는 동일한 방식으로 동작:

  v `+=` *e* : *v* 를 *e*에 더하고, *v*에 결과 저장

  v `-=` *e* : *v* 를 *e*에서 빼고, *v*에 결과 저장

  v `*=` *e* : *v* 를 *e*와 곱하고, *v*에 결과 저장

  v `/=` *e* : *v* 를 *e*로 나누고, *v*에 결과 저장

  v `%=` *e* : *v* 를 *e*로 나눈 나머지를, *v*에 저장

- *v* `+=` *e* 가 *v* = *v* + *e* *언제나 동치는 아님.*
  - 연산자 우선 수위가 있기 때문에 `i *= j + k` 는 `i = i * j + k`와 동치가 아님

Increment and Decrement Operators 증감, 가감 연산자

- 변수에 가장 자주 사용되는 연산은 "증가" (더하기 1) 그리고 "가감" (빼기 1):
  - C 는 이를 위해 `++` (***증가***) 그리고 `--` (***감소***)연산자 제공.

- `++` 연산자는 피연산자에 1을 더하고, `--` 연산자는 1을 뺌.
  - ***prefix*** 연산자 (`++i`, `--i`) 또는 ***postfix*** 연산자 (`i++`, `i--`).

```
i = i + 1;
j = j - 1;
```

```
i += 1;
j -= 1;
```

합성 할당 연산자

```
i++;  ++i;
j--;  --i;
```

증가 감소 연산자

Increment and Decrement Operators

- `++i` ("먼저 더함")는 `i + 1` 를 뜻하고 side effect로 `i`는 증가됨

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

- `i++` ("후에 더함") 의 결과는 `i`이고, 그 다음 `i`의 값이 증가됨

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

✓`++i` 는 "`i` 를 즉시 증가"를 뜻함

✓`i++` 는 "현재의 `i` 값을 쓰고, 그 후에 `i` 를 증가"를 뜻함

Increment and Decrement Operators

- `--` 연산자도 똑같은 성질을 갖음:

```
i = 1;
printf("i is %d\n", --i);   /* prints "i is 0" */
printf("i is %d\n", i);     /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 0" */
```

Increment and Decrement Operators

- ++ 또는 – 가 한 표현식에서 여러 차례 사용된다면 그 결과를 판단하기 어려워짐
- Example:

```
i = 1;
j = 2;
k = ++i + j++;
```

마지막 문장은 다음과 같음

```
i = i + 1;
k = i + j;
j = j + 1;
```

최종 결과로 i, j, k 는 2, 3, 4 가 됨

Increment and Decrement Operators

- 다음의 문장을 실행 시키면

```
i = 1;
j = 2;
k = i++ + j++;
```

i, j, k 는 2, 3, 3 이 됨.

표현식 평가

- 지금까지 다룬 연산자:

우선 순위	*이름*	*기호*	*결합 방향*
1	increment (postfix)	++	left
	decrement (postfix)	--	
2	increment (prefix)	++	right
	decrement (prefix)	--	
	unary plus	+	
	unary minus	-	
3	multiplicative	* / %	left
4	additive	+ -	left
5	assignment	= *= /= %= += -=	right

표현식 평가

- 괄호를 써서 표현식을 묶는 방식으로 평가할 수 있음
- 가장 높은 우선 순위의 연산자를 중심으로 연산자와 피연산자를 괄호로 묶음
- Example:

	Precedence level
`a = b += c++ - d + --e / -f`	
`a = b += (c++) - d + --e / -f`	1
`a = b += (c++) - d + (--e) / (-f)`	2
`a = b += (c++) - d + ((--e) / (-f))`	3
`a = b += (((c++) - d) + ((--e) / (-f)))`	4
`(a = (b += (((c++) - d) + ((--e) / (-f)))))`	5

하위표현식의 평가 순서

- Example:

```
i = 2;
j = i * i++;
```

- `j` 의 값이 4라고 생각하기 쉬움. 하지만 `j` 는 6일 수도 있음:
  1. 두 번째 피연산자가 (i의 원래 값) 사용되고, `i` 가 증가됨
  2. 첫 번째 피연산자가 (`i`의 새로운 값) 사용됨
  3. `i`의 원래 값과 새로운 값이 곱해져 6이 됨.

Expression Statements

- C 에선 모든 표현식이 문장이 될 수 있음.
- Example:

```
++i;
```

`i` 가 먼저 증가되고, 그 새로운 값을 읽었지만 어디에 쓰이진 않음

- side effect가 있는 경우만 표현식이 문장으로서 가치가 있음:

```
i = 1;          /* useful */
i--;            /* useful */
i * j - 1;      /* not useful */
```