Expressions & Selection Statment

adopted from KNK C Programming : A Modern Approach

Expressions

Operators

- C emphasizes expressions rather than statements.
- Expressions are built from variables, constants, and operators.
- C has a rich collection of operators, including

  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)

  and many others

Arithmetic Operators

- C provides five binary ***arithmetic operators:***

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

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

Ex: A*B, A+B

- There are also two ***unary*** arithmetic operators:

`+`	unary plus
`-`	unary minus

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

음수 양수 구분용

Binary Arithmetic Operators

- The value of `i % j` is the remainder when `i` is divided by `j`.

  `10 % 3` has the value 1, and `12 % 4` has the value 0.

- Binary arithmetic operators—with the exception of `%`—allow either integer or floating-point operands, with mixing allowed.
- When `int` and `float` operands are mixed, the result has type `float`.

  `9 + 2.5f` has the value 11.5, and `6.7f / 2` has the value 3.35.

The / and % Operators

- The / and % operators require special care:
  - When both operands are integers, / “truncates” the result. The value of 1 / 2 is 0, not 0.5. (두 수가 정수이면 소수점은 버림)
  - The % operator requires integer operands; if either operand is not an integer, the program won’t compile. (정수만 가능)
  - Using zero as the right operand of either / or % causes undefined behavior. (0으로 나눌수 없음)

❖The behavior when / and % are used with negative operands is ***implementation-defined(구현에 따라 다름)*** in C89.

❖In C99, the result of a division is always truncated toward zero and the value of i % j has the same sign as i. (결과는 항상 내림, i의 부호를 따름

Operator Precedence (연산자 우선순위)

- Does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`" or "multiply `j` and `k`, then add `i`"?
- One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`.
- If the parentheses are omitted, C uses ***operator precedence*** rules to determine the meaning of the expression.
- 우선순위를 모를 때는 괄호를 써서 먼저 계산한 것을 표시!

Operator Precedence

- The arithmetic operators have the following relative precedence:

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

- Examples:

`i + j * k` is equivalent to `i + (j * k)`

`-i * -j` is equivalent to `(-i) * (-j)`

`+i + j / k` is equivalent to `(+i) + (j / k)`

Operator Associativity (연산자 결합)

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be ***left associative*** if it groups from left to right.
- The binary arithmetic operators (`*`, `/`, `%`, `+`, and `–`) are all left associative, so

  `i - j – k` is equivalent to `(i - j) - k`
  `i * j / k` is equivalent to `(i * j) / k`
- An operator is ***right associative*** if it groups from right to left.
- The unary arithmetic operators (`+` and `–`) are both right associative, so

  `- + i` is equivalent to `- (+i)`

Assignment Operators

1. ***Simple assignment:*** used for storing a value into a variable

2. ***Compound assignment:*** used for updating a value already stored in a variable

Simple Assignment

- The effect of the assignment ***v* = *e*** is to evaluate the <u>expression *e* and copy its value into *v*.</u>
- ***e*** can be a constant, a variable, or a more complicated expression:

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

- If ***v*** and ***e*** don't have the same type, then the value of ***e* is converted to the type of *v***

```
int i;
float f;

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

Side Effects

- An operators that modifies one of its operands is said to have a ***side effect.***
- The simple assignment operator has a side effect: it modifies its left operand.
- Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.
- Since assignment is an operator, several assignments can be chained together:

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

- The `=` operator is right associative, so this assignment is equivalent to

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

Side Effects

- Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;

f = i = 33.3f;
```

`i` is assigned the value 33,
then `f` is assigned 33.0 (not 33.3).

Side Effects

- An assignment of the form *v* = *e* is allowed wherever a value of type *v* would be permitted:

```
i = 1;
k = 1 + (j = i); // Embedded assignments
                 // source of bugs
                 // hard to read
printf("%d %d %d\n", i, j, k);
  /* prints "1 1 2" */
```

Lvalues

- The assignment operator requires an ***lvalue*** as its left operand.
- An **lvalue** represents **an object stored in computer memory**, not a constant or the result of a computation.
- Variables are lvalues; expressions such as `10` or `2 * i` are not.

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

- The compiler will produce an error message such as *"invalid lvalue in assignment."*

Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common.
- Example:

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

- Using the += compound assignment operator, we simply write:

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

Compound Assignment

- There are nine other compound assignment operators, including the following: `-=` `*=` `/=` `%=`

  All of them work in much the same way:

  v `+=` *e* adds *v* to *e*, storing the result in *v*

  v `-=` *e* subtracts *e* from *v*, storing the result in *v*

  v `*=` *e* multiplies *v* by *e*, storing the result in *v*

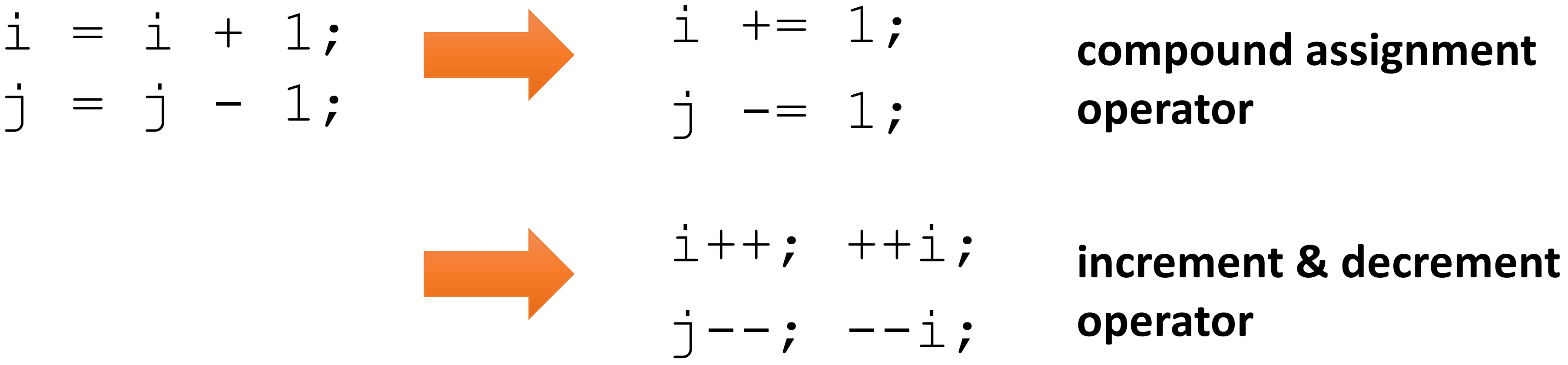
  v `/=` *e* divides *v* by *e*, storing the result in *v*

  v `%=` *e* computes the remainder when *v* is divided by *e*, storing the result in *v*

- *v* `+=` *e* isn't "equivalent" to *v* = *v* + *e*.
  - One problem is operator precedence: `i *= j + k` isn't the same as `i = i * j + k`.

Increment and Decrement Operators

- Two of the most common operations on a variable are "incrementing" (adding 1) and "decrementing" (subtracting 1):
- C provides special `++` (***increment***) and `--` (***decrement***) operators.
- The `++` operator adds 1 to its operand. The `--` operator subtracts 1.
  - They can be used as ***prefix*** operators (`++i` and `--i`) or ***postfix*** operators (`i++` and `i--`).



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

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

compound assignment operator

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

increment & decrement operator

Increment and Decrement Operators

- Evaluating the expression `++i` (a “pre-increment”) yields `i + 1` and—as a side effect—increments `i`:

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

- Evaluating the expression `i++` (a “post-increment”) produces the result `i`, but causes `i` to be incremented afterwards:

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

✓`++i` means “increment `i` immediately”

✓`i++` means “use the old value of `i` for now, but increment `i` later.”

Increment and Decrement Operators

- The -- operator has similar properties:

```
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

- When `++` or `--` is used more than once in the same expression, the result can often be hard to understand.
- Example:

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

The last statement is equivalent to

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

The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

Increment and Decrement Operators

- In contrast, executing the statements

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

  will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

Expression Evaluation

- Table of operators discussed so far:

Precedence	***Name***	***Symbol(s)***	***Associativity***
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

Expression Evaluation

- The table can be used to add parentheses to an expression that lacks them.
- Starting with the operator with highest precedence, put parentheses around the operator and its operands.
- Example:

`a = b += c++ - d + --e / -f`

	Precedence level
`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

Order of Subexpression Evaluation

- Example:

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

- It’s natural to assume that `j` is assigned 4. However, `j` could just as well be assigned 6 instead:

  1. The second operand (the original value of i) is fetched, then `i` is incremented.
  2. The first operand (the new value of `i`) is fetched.
  3. The new and old values of `i` are multiplied, yielding 6.

Expression Statements

- C has the unusual rule that any expression can be used as a statement.
- Example:

```
++i;
```

`i` is first incremented, then the new value of `i` is fetched but then discarded.

- Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect:

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

HW: 짧지만 복잡한 표현 만들기

- 실습반 수업 전까지 계산 가능한 복잡한 수식 5개 만들어 오기
  - 단, 작성한 본인이 정답을 구하고, 검토해서 와야 함
- 양식에 맞게 출력해오기

- Follow up activity in the Lab time.
  - 가장 복잡한 수식 작성자로 선정된 사람에게 초코바 1개
  - 해당 수식을 가장 짧은 시간 내에 정답을 찾는 사람에게 초코바 1개
  - 단, 제출자 제외

Selection Statements

Statements

- So far, we've used `return` statements and expression statements.
- Most of C's remaining statements fall into three categories:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while`, `do`, and `for`
  - ***Jump statements:*** `break`, `continue`, and `goto`. (`return` also belongs in this category.)
- Other C statements:
  - Compound statement
  - Null statement

Logical Expressions

- Several of C's statements must test the value of an expression to see if it is "**true**" or "**false**."
- For example, an `if` statement might need to test the expression `i < j`; a true value would indicate that `i` is less than `j`.
- A comparison such as `i < j` yields an integer: **either 0 (false) or 1 (true).**

Relational Operators

- C's ***relational operators:***

`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

→ produce **0 (false)** or **1 (true)** when used in expressions.

- The precedence of the relational operators is lower than that of the arithmetic operators.
  - For example, `i + j < k - 1` means `(i + j) < (k - 1)`.
- The relational operators are left associative.

`i < j < k` → `(i < j) < k`

The 1 or 0 produced by `i < j` is then compared to `k`

Equality Operators

- C provides two ***equality operators:***

`==` equal to   left associative

`!=` not equal to   **0 (false)** or **1 (true)** as result

- The equality operators have lower precedence than the relational operators

```
i < j == j < k   →   (i < j) == (j < k)
```

Logical Operators

- More complicated logical expressions can be built from simpler ones by using the ***logical operators:***

`!`	logical negation	(unary)
`&&`	logical *and*	(binary)
`||`	logical *or*	(binary)

Result

- 0 means false
- 1 means true

Operand

- 0 is treated as false
- > 0 is treated as true

- Behavior of the logical operators:

!expr has the value 1 if *expr* has the value 0. 참이면 거짓, 거짓이면 참

expr1* && *expr2 has the value 1 if the values of *expr1* and *expr2* are both nonzero. 둘 다 참이면 참, 아니면 거짓

expr1* || *expr2 has the value 1 if either *expr1* or *expr2* (or both) has a nonzero value. 둘 중 하나가 참이면 참, 둘 다 거짓이면 거짓

Logical Operators

- Both `&&` and `||` perform “short-circuit” evaluation: they first evaluate the left operand, then the right one. 왼쪽 먼저 검사, 검사 후 판단 가능하면 오른쪽 검사 안함
- Example:

```
(i != 0) && (j / i > 0) // 영으로 나누기 방지
```

Logical Operators

- The `!` operator has the same precedence as the unary plus and minus operators.
- The precedence of `&&` and `||` is lower than that of the relational and equality operators.
  - For example, `i < j && k == m` means `(i < j) && (k == m)`.
- The `!` operator is right associative; `&&` and `||` are left associative.

The `if` Statement

- The `if` statement allows a program to choose between two alternatives by testing an expression.
- In its simplest form, the `if` statement has the form

  `if (` *expression* `)` *statement*

- When an `if` statement is executed, *expression* is evaluated; if its value is nonzero, *statement* is executed.
- Example:

```
if (line_num == MAX_LINES)
  line_num = 0;
```

The `if` Statement

- Confusing `==` (equality) with `=` (assignment) is perhaps the most common C programming error.
- The statement

  ```
  if (i == 0) …
  ```

  tests whether `i` is equal to 0.
- The statement

  ```
  if (i = 0) …
  ```

  assigns 0 to `i`, then tests whether the result is nonzero.

The `if` Statement

- Often the expression in an `if` statement will test whether a variable falls within a range of values.
- To test whether 0 ≤ `i` < `n`:

```
if (0 <= i && i < n) …
```

- To test the opposite condition (`i` is outside the range):

```
if (i < 0 || i >= n) …
```

Compound Statements

- In the `if` statement template, notice that *statement* is singular, not plural:

  `if (` *expression* `)` *statement*

- To make an `if` statement control two or more statements, use a ***compound statement.***

- A compound statement has the form

  `{` *statements* `}`

- Putting braces around a group of statements forces the compiler to treat it as a single statement.

Compound Statements

- Example:

```
{ line_num = 0; page_num++; }
```

- A compound statement is usually put on multiple lines, with one statement per line:

```
{
  line_num = 0;
  page_num++;
}
```

- Example of a compound statement used inside an `if` statement:

```
if (line_num == MAX_LINES) {
  line_num = 0;
  page_num++;
}
```

The `else` Clause

- An `if` statement may have an `else` clause:

  `if (` *expression* `)` *statement* `else` *statement*

- The statement that follows the word `else` is executed if the expression has the value 0.

- Example:

```
if (i > j)
  max = i;
else
  max = j;
```

The `else` Clause

- It’s not unusual for `if` statements to be nested inside other `if` statements: 중첩가능

```
if (i > j)
  if (i > k)
    max = i;
  else
    max = k;
else
  if (j > k)
    max = j;
  else
    max = k;
```

```
if (i > j) {
  if (i > k)
    max = i;
  else
    max = k;
} else {
  if (j > k)
    max = j;
  else
    max = k;
}
```

```
if (i > j) {
  if (i > k) {
    max = i;
  } else {
    max = k;
  }
} else {
  if (j > k) {
    max = j;
  } else {
    max = k;
  }
}
```

- Aligning each `else` with the matching `if` makes the nesting easier to see.

Cascaded `if` Statements

- A “cascaded” `if` statement is often the best way to test a series of conditions, stopping as soon as one of them is true.
- Example:

```
if (n < 0)
  printf("n is less than 0\n");
else
  if (n == 0)
    printf("n is equal to 0\n");
  else
    printf("n is greater than 0\n");
```

Cascaded `if` Statements

- Although the second `if` statement is nested inside the first, C programmers don’t usually indent it.
- Instead, they align each `else` with the original `if`:

```
if (n < 0)
  printf("n is less than 0\n");
else if (n == 0)
  printf("n is equal to 0\n");
else
  printf("n is greater than 0\n");
```

Cascaded `if` Statements

- This layout avoids the problem of excessive indentation when the number of tests is large:

```
if ( expression )
  statement
else if ( expression )
  statement
…
else if ( expression )
  statement
else
  statement
```

The “Dangling `else`” Problem

- When if statements are nested, the “dangling `else`” problem may occur:

```
if (y != 0)
  if (x != 0)
    result = x / y;
else
  printf("Error: y is equal to 0\n");
```

- The indentation suggests that the `else` clause belongs to the outer `if` statement.
- However, C follows the rule that an `else` clause belongs to the nearest `if` statement that hasn’t already been paired with an `else`.

Conditional Expressions

- C's ***conditional operator*** allows an expression to produce one of two values depending on the value of a condition.
- The conditional operator consists of two symbols (`?` and `:`), which must be used together:

  expr1* ? *expr2* : *expr3

- The operands can be of any type.
- The resulting expression is said to be a ***conditional expression.***
- it is often referred to as a ***ternary*** operator.
- The conditional expression *expr1* `?` *expr2* `:` *expr3* should be read "if *expr1* then *expr2* else *expr3*."

Conditional Expressions

- Example:

```
int i, j, k;

i = 1;
j = 2;
k = i > j ? i : j;              /* k is now 2 */
k = (i >= 0 ? i : 0) + j;       /* k is now 3 */
```

짧지만 난해함

- The parentheses are necessary, because the precedence of the conditional operator is less than that of the other operators discussed so far, with the exception of the assignment operators.
- Conditional expressions are often used in `return` statements:

```
return i > j ? i : j;
```

Conditional Expressions

- Calls of `printf` can sometimes benefit from condition expressions. Instead of

  ```
  if (i > j)
    printf("%d\n", i);
  else
    printf("%d\n", j);
  ```

  we could simply write

  ```
  printf("%d\n", i > j ? i : j);
  ```

- Conditional expressions are also common in certain kinds of macro definitions.

Boolean Values: 1 or 0; true or false

Old way

```
int flag;

flag = 0;
…
flag = 1;
```

Not good
for readability

C89

```
#define TRUE  1
#define FALSE 0

flag = FALSE;
…
flag = TRUE;
```

Natural to
understand

```
if (flag == TRUE)
  …
if (flag == FALSE)
  …
```

C89 Better Usage

```
#define BOOL int
BOOL flag;
```

Clearly represents
boolean condition

```
if (flag)
  …
if (!flag)
  …
```

Boolean Values in C99

- C99 provides the `_Bool` type.
- A Boolean variable can be declared by writing

```
_Bool flag; //special integer type only with 0 or 1
flag = 5;   /* flag is assigned 1 */
```

- C99's `<stdbool.h>` header defines a macro, `bool`, that stands for `_Bool`.

```
#include <stdbool.h>
 bool flag;   /* same as _Bool flag; */
```

- also supplies `true` and `false` macros which stand for 1 and 0

```
flag = false;
flag = true;
```

The `switch` Statement

A cascaded `if` statement

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```

`switch` statement

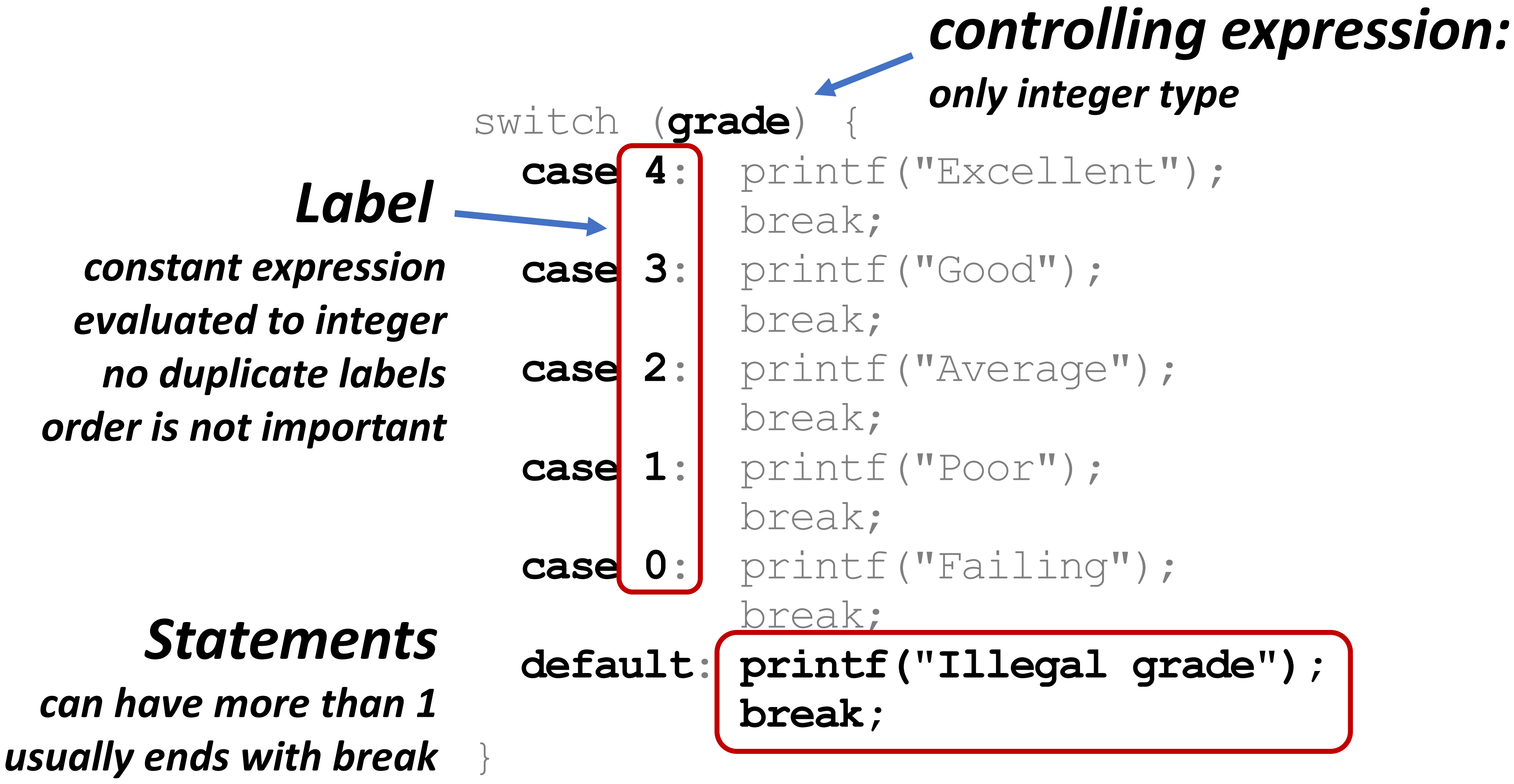
```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

The `switch` Statement

- A `switch` statement may be easier to read than a cascaded `if` statement.
- `switch` statements are often faster than `if` statements.
- Most common form of the `switch` statement:

```
switch ( expression ) {
  case constant-expression : statements
  …
  case constant-expression : statements
  default : statements
}
```

The `switch` Statement



controlling expression:

only integer type

```
switch (grade) {
  case 4: printf("Excellent");
          break;
  case 3: printf("Good");
          break;
  case 2: printf("Average");
          break;
  case 1: printf("Poor");
          break;
  case 0: printf("Failing");
          break;
  default: printf("Illegal grade");
           break;
}
```

Label

constant expression
evaluated to integer
no duplicate labels
order is not important

Statements

can have more than 1
usually ends with break

The `switch` Statement

- Several case labels may precede a group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

- If the `default` case is missing and the controlling expression's value doesn't match any case label, control passes to the next statement after the `switch`.