Control Flow

adopted from KNK C Programming : A Modern Approach

요약

for loop

- 구성 요소
  - 초기값
  - 제어구문
  - 증가값

카운팅할 때 주로 사용되지만
용도가 많아 다양하게 활용됨

```
for (int cnt = 0, n = 5; cnt < n; ++cnt)
    odd += cnt;
```

while

- 구성 요소
  - 초기값
  - 제어구문
  - 증가값

1. 조건이 참인지 검사하고
2. 참인 경우에만 동작

```
int cnt = 0, n = 5;
while (cnt < n){
    odd += cnt
    cnt++;
}
```

do.. while

- 구성 요소
  - 초기값
  - 제어구문
  - 증가값

1. 조건 없이 1회 수행한 후
2. 조건이 참인지 검사하고
3. 참인 경우에만 동작

```
do{
    printf("hello\n");
} while (1);   // 1: 참, 0: 거짓
```

결과:
hello
hello
… (무한반복)

break; continue

- break;
  - 루프 밖으로 이동
  - 한 루프 밖으로만 이동
- continue;
  - 루프 마지막으로 이동
  - 동일한 루프에서 다시 시작

```
do{
    printf("hello\n");
    break;
    printf("world\n");
} while (1);
```

결과:
hello

```
do{
    printf("hello\n");
    continue;
    printf("world");
} while (1);
```

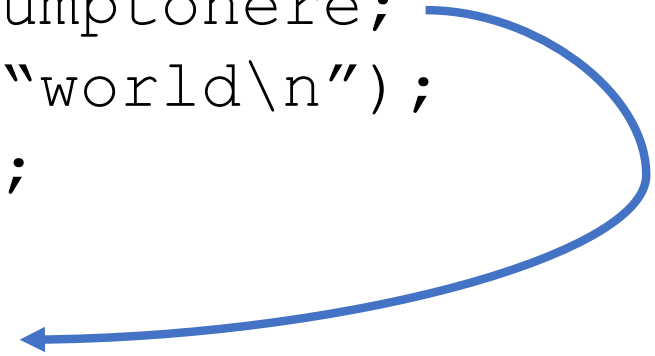
결과:
hello
hello
… (무한반복)

goto

- 지정된 레이블로 점프

```
do{
    printf("hello\n");
    goto  jumptohere;
    printf("world\n");
} while (1);

jumptohere:
    printf("aha!");
```



결과:
hello
aha!

Workshop: for, while, do while conversion

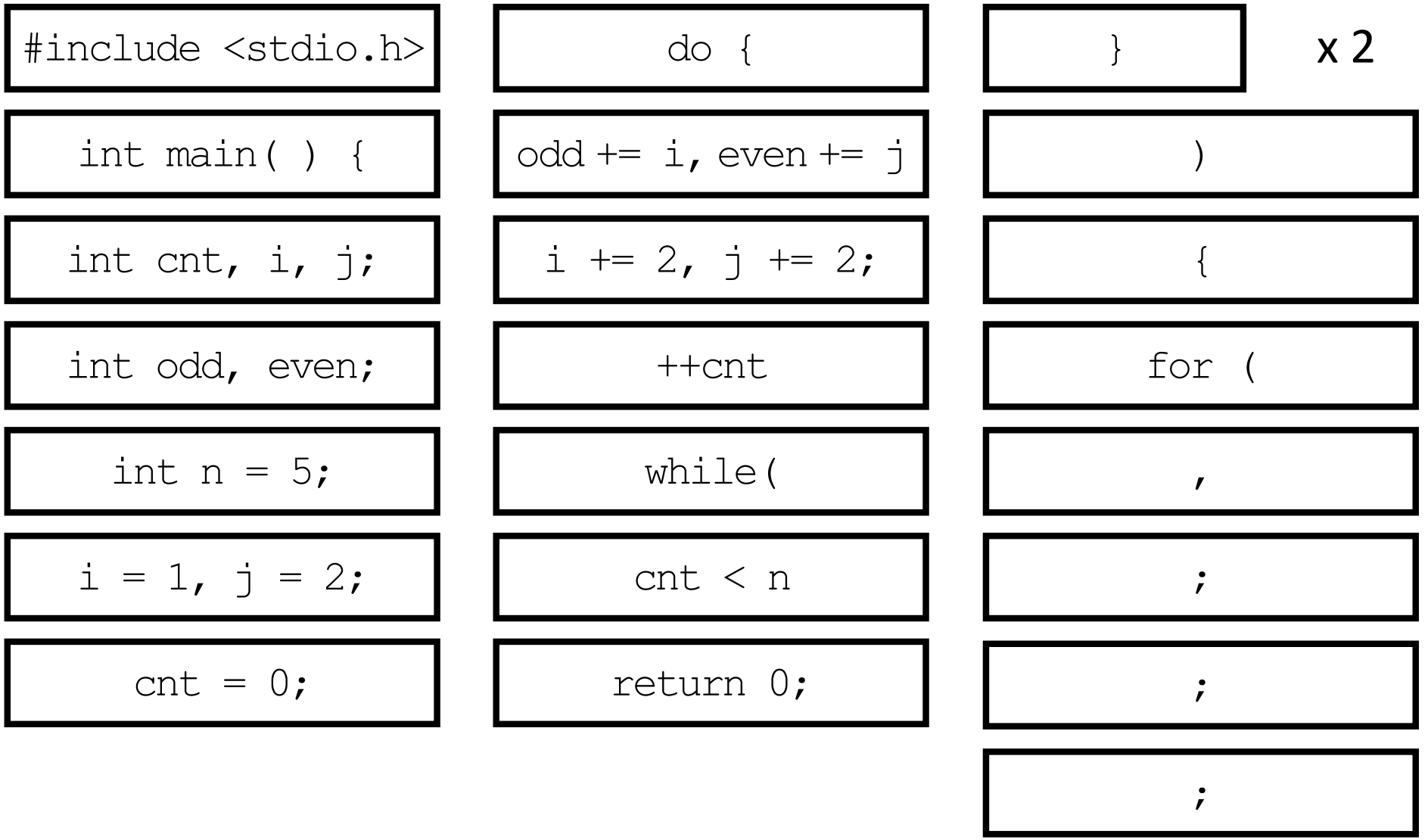
- 3 명이 한 팀이 되어 문제를 해결한다.
- 다음과 같은 코드가 있다

```
for (cnt = 0, i = 1, j =2; cnt < n; ++cnt, i += 2, j += 2)
    odd += i, even += j;
```

- 코드를 이해하고
  1. 쉼표를 사용하지 않는 for 문으로 변경
  2. while문으로 변경
  3. do while 문으로 변경

워크샵

사용할 네임카드



Workshop: break and goto

- 100 명의 성적의 평균을 구하는 프로그램의 일부이다

```
for (cnt = 0; cnt < n; ++cnt) {
    scanf("%d", &i);
    if (i >= 0){
        j += i;
    }
}
```

- 성적을 모두 입력한 후 -1을 입력하면 그 명 수만큼의 평균을 구하고 싶다. 예: 30, 50, 70, -1 입력시 3명의 평균을 구한다.
  - break 로 이 기능을 구현하시오
  - goto 로 이 기능을 구현하시오
  - continue 로 이 기능이 구현 가능한지 논하시오.

사용할 네임카드, 색 있는 것이 추가 된 것

#include <stdio.h>	for (	}	x3
int main( ) {	scanf(	) {	x2
int cnt, i, j;	i	)	
"%d",	++cnt	for (	out :
i >= 0	if (	j += i;	goto out;
&	cnt < n;	;	} else {
cnt = 0;	return j/cnt;	i < 0	break;

Workshop: null statement

- 다음 문장을 널 문장을 사용하여 루프바디가 없도록 만들기

```
for (n = 0; m > 0; n++)
    m /= 2;
```

- 다음 문장에서 잘못된 부분 찾기

```
if (n % 2 == 0);
    printf("n is even\n");
```

워크샵 용 세부 설명

Iteration Statements (반복 문장)

- C's iteration statements are used to set up loops.
- A ***loop*** (루프) is a statement whose job is to repeatedly execute some other statement (the ***loop body*** *루프 바디에 반복할 문장 작성*)
- In C, every loop has a ***controlling expression***. 루프에는 제어구문 필요
- Each time the loop body is executed (an ***iteration*** of the loop), the controlling expression is evaluated. 한 번 반복할 때마다 제어구문 재평가
  - If the expression is true (has a value that's not zero) the loop continues to execute. 참일 때만 반복

Iteration Statements

- C provides three iteration statements:
  - The `while` statement is used for loops whose controlling expression is tested *before* the loop body is executed.
    제어구문 평가 후 참이면 루프바디 실행
  - The `do` statement is used if the expression is tested *after* the loop body is executed.
    루프바디 1회 실행후 제어구문 평가, 참이면 다시 실행
  - The `for` statement is convenient for loops that increment or decrement a counting variable.
    변수의 증감이 쉬운 반복문

The `while` Statement

- Using a `while` statement is the easiest way to set up a loop.
  가장 쉽게 만들 수 있는 반복문
- The `while` statement has the form (문법)

`while` (*expression*) *statement*

- *expression (수식, 바디 실행여부 결정)* is the controlling expression; *statement* is the loop body.

The `while` Statement

- Example of a `while` statement:

```
while (i < n)  /* controlling expression */
  i = i * 2;   /* loop body */
```

- When a `while` statement is executed, the controlling expression is evaluated first. 제일 먼저 제어 구문을 평가
- If its value is nonzero (true), the loop body is executed and the expression is tested again. 양수/참이면 루프바디 실행, 다시 제어구문 평가
- The process continues until the controlling expression eventually has the value zero. 영 또는 거짓이면 종료

The `while` Statement

- A `while` statement that computes the smallest power of 2 that is greater than or equal to a number `n`:
  n과 같거나 큰 수 중 가장 작은 2의 배수

```
i = 1;
while (i < n)
  i = i * 2;
```

- A trace of the loop when `n` has the value 10:

`i = 1;`	`i` is now 1.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 2.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 4.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 8.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 16.
Is `i < n`?	No; exit from loop.

The `while` Statement

- If multiple statements are needed, use braces to create a single compound statement:
  루프바디에 여러 문장을 넣으려면 중괄호를 사용

```
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

- Some programmers always use braces, even when they're not strictly necessary: 한 문장만 쓰더라도 중괄호 사용가능

```
while (i < n) {
  i = i * 2;
}
```

The `while` Statement

- The following statements display a series of "countdown" messages:

```
i = 10;                          //i가 0 이하일 때만 i>0이 거짓
while (i > 0) {                  //0이하이면 루프바디가 실행안됨
  printf("T minus %d and counting\n", i);
  i--;
}

                                 // 다음처럼 한 줄로 표현 가능
                                 // printf("T minus %d and counting\n", i--);
```

- The final message printed is `T minus 1 and counting`.

Infinite Loops

- A `while` statement won't terminate if the controlling expression always has a nonzero value.
  평가 결과가가 0이외의 값이면 종료 안함
- C programmers sometimes deliberately create an ***infinite loop*** by using a nonzero constant as the controlling expression: 무한루프를 일부러 만들기도 함

  ```
  while (1) …
  ```

- A `while` statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (`break`, `goto`, `return`) or calls a function that causes the program to terminate. 무한루프에서는 별도 종료/분기 호출이 필요함

The do Statement

- General form of the do statement: 문법

```
do statement while ( expression ) ;
```

- When a do statement is executed, the loop body is executed first, then the controlling expression is evaluated.
  루프바디를 먼저 실행 후 제어구문 평가
- If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more.
  "제어구문이 0이 아니면 루프바디 재 실행하고 다시 제어구문을 평가"의 반복

The do Statement

- The countdown example rewritten as a `do` statement:

```
i = 10;
do {                    // do, while에서는 항상 중괄호 사용
  printf("T minus %d and counting\n", --i);
} while (i > 0);
```

- The `do` statement is often indistinguishable from the `while` statement. `do`와 `while`은 차이가 없음
- The only difference is that the body of a `do` statement is always executed at least once.
  유일한 차이는 최소 1번 루프바디 실행된다는 것

The `for` Statement

- The `for` statement is ideal for loops that have a "counting" variable, but it's versatile enough to be used for other kinds of loops as well. 다양한 방식으로 루프를 구현할 수 있음
- General form of the `for` statement:

초기 값 묶음 제어구문 묶음 증가구문 묶음

`for` (*expr1* ; *expr2* ; *expr3*) *statement*

expr1, *expr2*, and *expr3* are expressions.

- Example:

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

The `for` Statement

- Except in a few rare cases, a `for` loop can always be replaced by an equivalent `while` loop:
  거의 모든 경우 for는 while문으로 변경 가능

```
expr1;
while ( expr2 ) {
    statement
    expr3;
}
```

The `for` Statement

- *expr1* is an initialization step that's performed only once, before the loop begins to execute. 초기화만 반복문 밖으로 보냄
- *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero).
  expr2가 종료를 제어함, 0이 아니면 계속 실행, 심지어 없어도 실행
- *expr3* is an operation to be performed at the end of each loop iteration. 루프바디를 실행한 후 expr3 수행

Conversion

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

The `for` Statement

- Studying the equivalent `while` statement can help clarify the fine points of a `for` statement.
- For example, what if `i--` is replaced by `--i`?

```
for (i = 10; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

- The equivalent `while` loop shows that the change has no effect on the behavior of the loop:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  --i;
}
```

for-to-while.c

```
/* case 1 */
    for (int i = 10; i > 0; --i)
        printf("T minus %d and counting\n", i);

/* case 2 */
    printf("\n\ncase 2\n");
    for (int i = 10; i > 0; i--)
        printf("T minus %d and counting\n", i);

/* case 3 */
    printf("\n\ncase 3\n");
    int i = 10;
    while (i > 0) {
        printf("T minus %d and counting\n", --i);
     // printf("T minus %d and counting\n", i--);
```

The `for` Statement

- Since the first and third expressions in a `for` statement are executed as statements, their values are irrelevant—they're useful only for their side effects.
  첫 수식과 마지막 수식의 값은 중요하지 않고 그 문장으로 인한 영향만 의미 있음
- Consequently, these two expressions are usually assignments or increment/decrement expressions.
  결과적으로, 첫 수식과 마지막 수식은 할당이나 증감 관련 수식만 쓰임

`for` Statement Idioms

- The `for` statement is usually the best choice for loops that "count up" (increment a variable) or "count down" (decrement a variable). 변수의 값을 반복적으로 더하거나 또는 빼는 루프에서 유용
- A `for` statement that counts up or down a total of `n` times will usually have one of the following forms: 많이 활용되는 패턴

Counting up from* 0 *to* `n`–1*: `for (i = 0; i < n; i++) …`

Counting up from* 1 *to* `n`*: `for (i = 1; i <= n; i++) …`

Counting down from* `n`–1 *to* 0*: `for (i = n - 1; i >= 0; i--) …`

Counting down from* `n` *to* 1*: `for (i = n; i > 0; i--) …`

`for` Statement Idioms

- Common `for` statement errors:
  - Using < instead of > (or vice versa) in the controlling expression. “Counting up” loops should use the < or <= operator. “Counting down” loops should use > or >=.
  - Using == in the controlling expression instead of <, <=, >, or >=.
  - “Off-by-one” errors such as writing the controlling expression as `i <= n` instead of `i < n`.

Omitting Expressions in a `for` Statement

- C allows any or all of the expressions that control a `for` statement to be omitted. 수식 중 일부를 안써도 됨
- If the *first* expression is omitted, no initialization is performed before the loop is executed:
  첫수식이 없으면 초기화 안됨; for문 전에 초기화 할 필요 있음

```
i = 10;
for (; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

Omitting Expressions in a `for` Statement

- If the *third* expression is omitted, the loop body is responsible for ensuring that the value of the second expression eventually becomes false:
  세번째 수식이 없으면 종료값을 만족 못 시킬 수있음. 루프바디에서 처리

```
for (i = 10; i > 0;  )
  printf("T minus %d and counting\n", i--);
```

Omitting Expressions in a `for` Statement

- When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a while statement in disguise:
  첫수식과 세번째 수식이 없다면 while문과 같음

```
for (; i > 0;)
  printf("T minus %d and counting\n", i--);
```

  is the same as

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

- The `while` version is clearer and therefore preferable.
  while을 쓰는 것이 더 읽기 좋음;

Omitting Expressions in a `for` Statement

- If the *second* expression is missing, it defaults to a true value, so the `for` statement doesn't terminate (unless stopped in some other fashion).
  두번째 수식이 없으면 항상 참으로 판단함, 종료하지 않음
- For example, some programmers use the following `for` statement to establish an infinite loop:
  무한 루프의 표현 방법 중 하나

```
for (;;) …
```

`for` Statements in C99

- In C99, the first expression in a `for` statement can be replaced by a declaration. c99에서는 새 변수를 선언하면서 초기화 할 수 있음
- This feature allows the programmer to declare a variable for use by the loop:

```
for (int i = 0; i < n; i++)
  …
```

- The variable `i` need not have been declared prior to this statement. 변수 i는 이전에 선언된 적 없음

`for` Statements in C99

- A variable declared by a `for` statement can't be accessed outside the body of the loop (we say that it's not ***visible*** outside the loop):
  단, for문 밖에서는 변수 i가 보이지 않기 때문에 쓸 수 없음

```
for (int i = 0; i < n; i++) {
  …
  printf("%d", i);
   /* legal; i is visible inside loop */
  …
}
printf("%d", i);    /*** WRONG ***/
```

`for` Statements in C99

- Having a `for` statement declare its own control variable is usually a good idea: it's convenient and it can make programs easier to understand. for문 전용 임시 카운트 변수를 선언하는 것은 유용함
- However, if the program needs to access the variable after loop termination, it's necessary to use the older form of the `for` statement. 단, 선언한 변수를 for문 종료후에는 활용하려면 for문 밖에서 선언
- A `for` statement may declare more than one variable, provided that all variables have the same type: 여러 변수를 한번에 선언가능

```
for (int i = 0, j = 0; i < n; i++)
  …
```

The Comma Operator

- On occasion, a `for` statement may need to have two (or more) initialization expressions or one that increments several variables each time through the loop.
  경우에 따라 하나의 이상의 변수를 초기화 할 필요가 있음
- This effect can be accomplished by using a ***comma expression*** as the first or third expression in the `for` statement.
  첫 수식과 마지막 수식은 쉼표로 다른 변수들을 추가 가능함
- A comma expression has the form

  expr1 , expr2

  where *expr1* and *expr2* are any two expressions.

The Comma Operator

- Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.
  쉼표로 연결된 두 식 중 첫 수식이 side effect(예: i = 0)가 없으면 의미 없음

- When the comma expression `++i, i + j` is evaluated, `i` is first incremented, then `i + j` is evaluated.
  ++i를 먼저 계산, i+j 계산시 증가한 i를 활용
  - If `i` and `j` have the values 1 and 5, respectively, the value of the expression will be 7, and `i` will be incremented to 2.

The Comma Operator

- The comma operator is left associative, so the compiler interprets

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

as

```
((i = 1), (j = 2)), (k = (i + j))
```

왼쪽에서 오른쪽으로 식이 결합됨

The Comma Operator

- The comma operator makes it possible to “glue” two expressions together to form a single expression.
  식은 두개 이지만, 하나의 수식으로 볼 수 있음
- Example:

```
for (sum = 0, i = 1; i <= N; i++)
  sum += i;
```

- With additional commas, the `for` statement could initialize more than two variables.쉼표로 하나 이상의 변수 선언 가능

Exiting from a Loop

- The normal exit point for a loop is at the beginning (as in a `while` or `for` statement) or at the end (the `do` statement).
  일반적으로 루프의 종료는 제어구문에서 판단됨

- Using the `break` statement, it's possible to write a loop with an exit point in the middle or a loop with more than one exit point.
  break를 사용하여 루프바디 중간 또는 임의의 지점에서 종료할 수 있음

The `break` Statement

- The `break` statement can transfer control out of a switch statement, but it can also be used to jump out of a `while`, `do`, or `for` loop. switch문외에 while, do, for 문에서도 break 사용 가능
- A loop that checks whether a number `n` is prime can use a `break` statement to terminate the loop as soon as a divisor is found:
  n이 소수인지 판단하는 루프에서 찾으면 break로 즉시 종료시킬 수 있음

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

The `break` Statement

- After the loop has terminated, an `if` statement can be use to determine whether termination was premature (hence `n` isn't prime) or normal (`n` is prime):
  break로 종료 후에는 if 문으로 검증해야 함

```
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

The `break` Statement

- The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end. break는 루프 수행 중에 종료할 때 유용
- Loops that read user input, terminating when a particular value is entered, often fall into this category:
  예시: 사용자로부터 숫자를 입력 받는 구문

```
for (;;) {
  printf("Enter a number (enter 0 to stop): ");
  scanf("%d", &n);
  if (n == 0)
    break;
  printf("%d cubed is %d\n", n, n * n * n);
}
```

The `break` Statement

- A `break` statement transfers control out of the innermost enclosing `while`, `do`, `for`, or `switch`. 중괄호 묶음 하나 밖으로 이동
- When these statements are nested, the `break` statement can escape only one level of nesting. 중첩시 하나의 묶음만 종료됨
- Example:

```
while (…) {
  switch (…) {
    …
    break;
    …
  }
}
```

- `break` transfers control out of the `switch` statement, but not out of the `while` loop. switch에서만 종료하고 while문은 계속 수행

The `continue` Statement

- The `continue` statement is similar to `break`: 종료는 유사
  - `break` transfers control just past the end of a loop.
    루프 밖으로 이동
  - `continue` transfers control to a point just before the end of the loop body. 루프바디 내에서 마지막으로 이동
- With `break`, control leaves the loop; with `continue`, control remains inside the loop.
  break는 루프 밖의 문장이 제어권 획득, continue는 루프가 제어권 유지
- There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.
  break는 다용도, continue는 루프에만 쓰임

The `continue` Statement

- A loop that uses the `continue` statement:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
  /* continue jumps to here */
}
```

The `continue` Statement

- The same loop written without using `continue`:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i != 0) {
    sum += i;
    n++;
  }
}
```

The `goto` Statement

- The `goto` statement is capable of jumping to any statement in a function, provided that the statement has a ***label.***
  label/라벨이 있는 위치로 이동
- A label is just an identifier placed at the beginning of a statement:

  라벨 실행될 문장

  identifier **:** *statement*

- The `goto` statement itself has the form

  라벨

  `goto` *identifier* **;**

- Executing the statement `goto` *L*`;` transfers control to the statement that follows the label *L*, which must be in the same function as the `goto` statement itself.

The `goto` Statement

- If C didn't have a `break` statement, a `goto` statement could be used to exit from a loop:

  goto를 쓸 수 없다면 break로 루프 밖으로 나갈 수 있음

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    goto done;

done:
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

The goto Statement

- goto는 break, continue, return, exit등으로 대부분 대체될 수 있기에 거의 안쓰임
- Consider the problem of exiting a loop from within a `switch` statement. 단 중첩 블럭에서 빠져나올 땐 활용됨
- The `break` statement doesn't have the desired effect: it exits from the `switch`, but not from the loop.
  break로는 switch와 while 블럭 밖으로 한 번에 나오지 못함
- A `goto` statement solves the problem:

```
while (…) {
  switch (…) {
    …
    goto loop_done;    /* break won't work here */
    …
  }
}
loop_done: …
```

goto 문장은 중첩 루프 밖으로 나오는 데 유용함

- The `goto` statement is also useful for exiting from nested loops.

The Null Statement

- A statement can be ***null***—devoid of symbols except for the semicolon at the end. 루프바디의 실행될 문장이 null일 수 있음, null은 비어 있다는 뜻
- The following line contains three statements:

```
i = 0; ; j = 1;
```

  세미콜론의 수를 보면 문장이 3개 있는 것을 알 수 있음. 단, 2번은 어떤 일도 안함

- The null statement is primarily good for one thing: writing loops whose bodies are empty.
  null문장을 쓰는 이유는 루프 내에서 할 일이 없을 때를 표현하기 위함

The Null Statement

- Consider the following prime-finding loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

앞에서 본 소수를 찾는 문장에 if 절의 수식(n%d == 0)을 for 의 종료 조건에 넣으면 루프 내에서 할 일이 없음

- If the `n % d == 0` condition is moved into the loop's controlling expression, the body of the loop becomes empty:

```
for (d = 2; d < n && n % d != 0; d++)

  /* empty loop body */ ;
```

- To avoid confusion, C programmers customarily put the null statement on a line by itself. 반복문을 독립 문장으로 쓸 때 혼란을 줄이기 위해 널 문장을 한 줄에 독립적으로 기록함.

The Null Statement

- Accidentally putting a semicolon after the parentheses in an `if`, `while`, or `for` statement creates a null statement.
  주의, 쉼표를 while, for, if 문장 뒤에 쓰면 널문장을 삽입한 것
- Example 1:

```
if (d == 0);                                    /*** WRONG ***/
  printf("Error: Division by zero\n"); //if와 상관없이 출력
```

  The call of `printf` isn't inside the `if` statement, so it's performed regardless of whether `d` is equal to 0.

- Example 2:

```
i = 10;
while (i > 0);                                  /*** WRONG ***/
{
  printf("T minus %d and counting\n", i); //while과 상관없이 출력
  --i; //while과 상관없이 출력
}
```

  The extra semicolon creates an infinite loop. 이 경우 무한 루프가 됨

The Null Statement

- Example 3: while은 루프바디 없이 1회 실행하고 printf문장이 실행됨, 들여 쓰기에 속으면 안됨

```
i = 11;
while (--i > 0);                        /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

The loop body is executed only once; the message printed is:

```
T minus 0 and counting
```

- Example 4: for 문으로 작성 된 것 외에는 위의 예제와 동일

```
for (i = 10; i > 0; i--);               /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

Again, the loop body is executed only once, and the same message is printed as in Example 3.