

*A program without a loop and a structured variable isn't worth writing.*

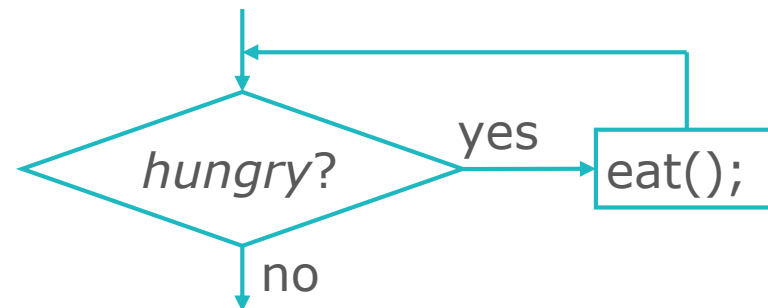
*-- Alan Perlis*



# Lecture 6 - Loops

Meng-Hsun Tsai  
CSIE, NCKU

```
while (hungry) {  
    eat();  
}
```



# Recall on Statements

- Most of C's remaining statements fall into three categories:

- **Selection statements:** `if` and `switch` → Lecture 5
- **Iteration statements:** `while`, `do`, and `for` → This lecture
- **Jump statements:** `break`, `continue`, `goto` and `return`. → This lecture

- Other C statements:

- **Compound statement :** `{ stmts }` → This lecture
- **Null statement :** `;` → This lecture
- **Expression statements :** `i++;` → This lecture

# Iteration Statements

- C's **iteration statements** are used to **set up loops**.
- A **loop** is a statement whose job is to **repeatedly execute some other statement** (the **loop body**).
- In C, every loop has a **controlling expression**.
- Each time the loop body is executed (an **iteration** of the loop), the controlling expression is evaluated.
  - If the expression is true (has a value that's not zero) the loop continues to execute.

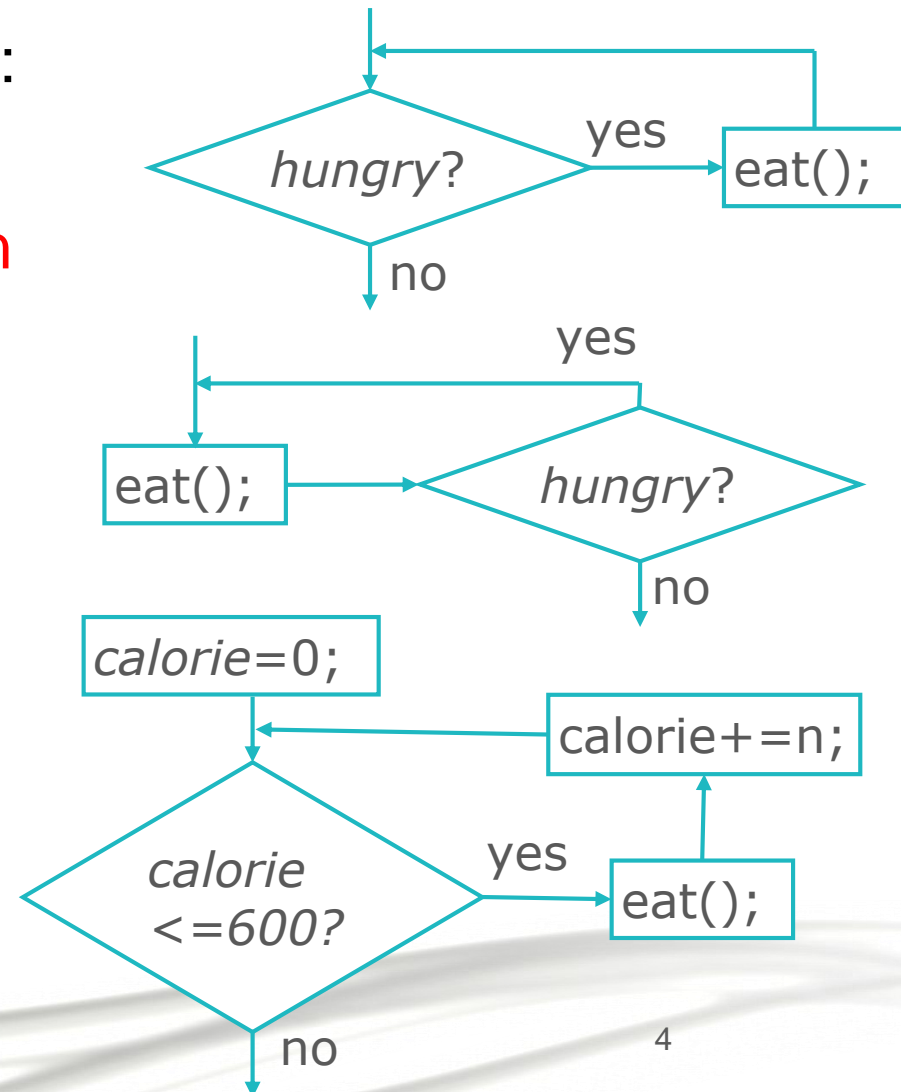
```
while (hungry)  
{  
    eat ();  
}
```

→ **controlling expression**

} **loop body**

# Iteration Statements (cont.)

- C provides **three iteration statements**:
  - The **while** statement is used for loops whose controlling **expression is tested before** the **loop body** is executed.
  - The **do** statement is used if the **expression is tested after** the **loop body** is executed.
  - The **for** statement is convenient for loops that **increment or decrement a counting variable**.



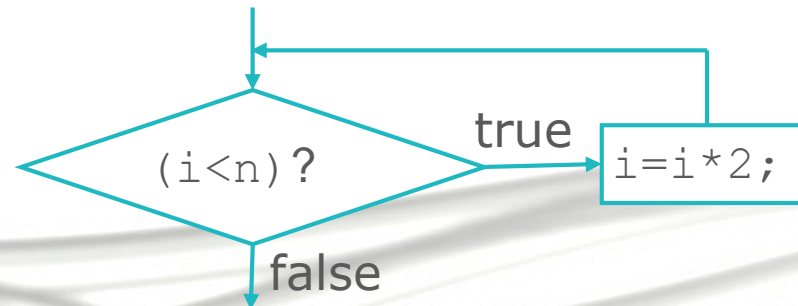
# The `while` Statement

- The `while` statement has the form

`while ( expression ) statement`

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

- expression* is the **controlling expression**; *statement* is the **loop body**.
- When a `while` statement is executed, the **controlling expression is evaluated first**.
- If its value is **nonzero (true)**, the **loop body is executed** and the **expression is tested again**.
- The process **continues until** the **controlling expression** eventually has the value **zero**.



# The `while` Statement (cont.)

- A `while` statement that computes the **smallest power of 2** that is **greater than or equal to** a number `n`:

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

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

Iteration	Controlling expression	Loop body
1	<code>1 &lt; 10</code> (true)	<code>i = 1 * 2</code> (= 2)
2	<code>2 &lt; 10</code> (true)	<code>i = 2 * 2</code> (= 4)
3	<code>4 &lt; 10</code> (true)	<code>i = 4 * 2</code> (= 8)
4	<code>8 &lt; 10</code> (true)	<code>i = 8 * 2</code> (= 16)
5	<code>16 &lt; 10</code> (false)	

`i` is **16** after the loop

# The `while` Statement (cont.)

- If multiple statements are needed, use braces to create a single compound statement:

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

- Some programmers always use braces, even when they're not strictly necessary:

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

# The `while` Statement (cont.)

- The following statements display a series of “countdown” messages:

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

```
T minus 10 and counting
T minus 9 and counting
T minus 8 and counting
T minus 7 and counting
T minus 6 and counting
T minus 5 and counting
T minus 4 and counting
T minus 3 and counting
T minus 2 and counting
T minus 1 and counting
```



# The `while` Statement (cont.)

- Observations about the `while` statement:
  - The controlling expression is false when a `while` loop terminates. Thus, **when a loop controlled by `i > 0` terminates, `i` must be less than or equal to 0.**
  - The **body** of a `while` loop **may not be executed at all**, because the **controlling expression** is **tested before** the **body** is executed.
  - A `while` statement can often be written in a variety of ways. A more concise version of the countdown loop:

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

# Infinite Loops

- A `while` statement **won't terminate** if the controlling expression **always** has a **nonzero** value.
- C programmers sometimes **deliberately** create an ***infinite loop*** by using a nonzero constant as the controlling expression:  

```
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**.

# Program: Printing a Table of Squares

**square.c**

```
#include <stdio.h>
int main(void)
{
    int i, n;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    while (i <= n) {
        printf("%10d%10d\n", i,
               i * i);
        i++;
    }
    return 0;
}
```

This program prints a table of squares.  
Enter number of entries in table: 5

1	1
2	4
3	9
4	16
5	25

# Program: Summing a Series of Numbers

**sum.c**

```
#include <stdio.h>

int main(void)
{
    int n, sum = 0;

    printf("This program sums a series of integers.\n");
    printf("Enter integers (0 to terminate): ");

    scanf("%d", &n);
    while (n != 0) {
        sum += n;
        scanf("%d", &n);
    }
    printf("The sum is: %d\n", sum);

    return 0;
```

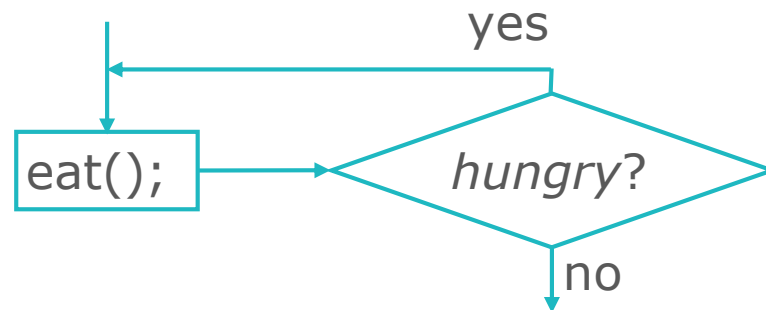
This program sums a series of integers.  
Enter integers (0 to terminate): 8 23 71 5 0  
The sum is: 107

# The do Statement

- General form of the do statement:

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

- When a do statement is executed, the **loop body** is executed **first**, **then** the **controlling expression is evaluated**.
- If the value of the **expression** is **nonzero**, the **loop body** is executed **again** and then the expression is evaluated once more.



# The `do` Statement (cont.)

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

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

- The `do` statement is **often indistinguishable** from the `while` statement.
- The **only difference** is that the body of a `do` statement is **always executed at least once**.

# The `do` Statement (cont.)

- It's a **good idea** to **use braces in *all* `do` statements**, whether or not they're needed, because a `do` statement without braces can easily be mistaken for a `while` statement:

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

- A careless reader might think that the word `while` was the beginning of a `while` statement.

# Program: Calculating the Number of Digits in an Integer

- The `numdigits.c` program calculates the **number of digits** in an **integer** entered by the user:

```
Enter a nonnegative integer: 60  
The number has 2 digit(s).
```

- The program will **divide** the user's input **by 10 repeatedly until it becomes 0**; the number of divisions performed is the number of digits.
- Writing this loop **as a `do` statement is better than using a `while` statement**, because every integer—even **0**—has **at least one digit**.



# Program: Calculating the Number of Digits in an Integer (cont.)

**numdigits.c**

```
#include <stdio.h>
int main(void)
{
    int digits = 0, n;

    printf("Enter a nonnegative integer: ");
    scanf("%d", &n);

    do {
        n /= 10;
        digits++;
    } while (n > 0);

    printf("The number has %d digit(s).\n", digits);

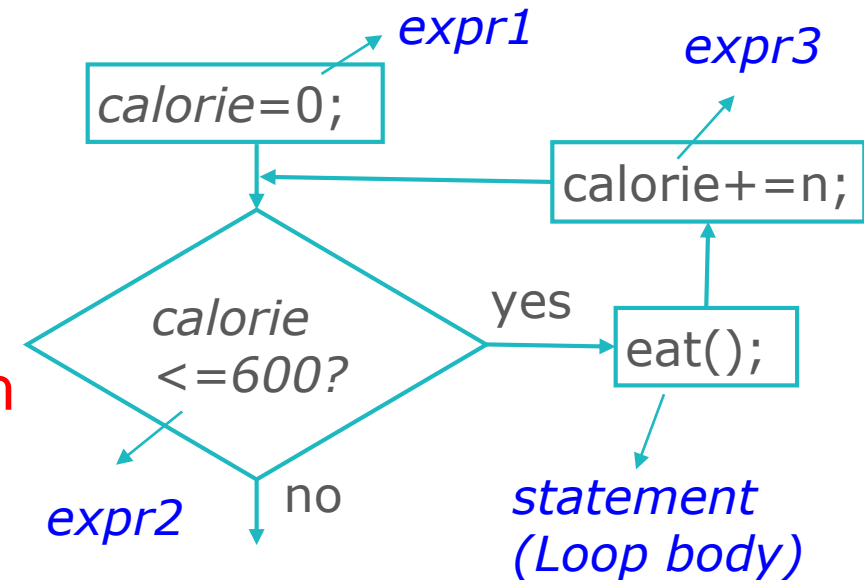
    return 0;
}
```

# The `for` Statement

- General form of the `for` statement:

`for ( expr1 ; expr2 ; expr3 ) statement`

- expr1* is an **initialization step** that's **performed only once**, before the loop begins to execute.
- expr2* **controls loop termination** (the loop continues executing as long as the value of *expr2* is nonzero).
- expr3* is performed **at the end of each loop iteration**.



# The `for` Statement (cont.)

- 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.
- Except in a few rare cases, a `for` loop can always be replaced by an equivalent `while` loop:

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

```
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 (cont.)

- 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.
- Consequently, these two expressions are usually **assignments** or **increment/decrement** expressions.

# 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).
- A `for` statement that counts up or down **a total of  $n$  times** will usually have one of the following forms:

**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 (cont.)

- Common for statement errors:
  - Using  $<$  instead of  $>$  (or vice versa) in the controlling expression. “Counting up” loops should use the  $<$  or  $\leq$  operator. “Counting down” loops should use  $>$  or  $\geq$ .
  - Using  $==$  in the controlling expression instead of  $<$ ,  $\leq$ ,  $>$ , or  $\geq$ .
  - “Off-by-one” errors such as writing the controlling expression as  $i \leq n$  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**.
- If the **first expression** is **omitted**, **no initialization** is performed before the loop is executed:

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

- If the **third expression** is **omitted**, the **loop body** is **responsible** for **ensuring** that the value of the **second expression** eventually **becomes false**:

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

# Omitting Expressions in a `for` Statement (cont.)

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

```
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.
- 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).

```
for ( ; ; ) ...
```



```
while (1) ...
```



# for Statements in C99

- In C99, the **first expression** in a `for` statement can be replaced by a **declaration**.
- This feature allows the programmer to declare a variable for use by the loop.
- 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):

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

# for Statements in C99 (cont.)

- 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**.
- However, if the program **needs to access the variable after loop termination**, it's **necessary to use the older form** of the `for` statement.
- A `for` statement **may declare more than one variable**, provided that all variables have the **same type**:

```
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.
- This effect can be accomplished by using a ***comma expression*** as the **first** or **third expression** in the `for` statement.
- A comma expression has the form

*expr1 , expr2*

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

# The Comma Operator (cont.)

- A comma expression is evaluated in two steps:
  - First, *expr1* is **evaluated** and its **value discarded**.
  - Second, *expr2* is **evaluated**; its **value** is the value **of the entire expression**.
- Evaluating *expr1* should **always have a side effect**; if it doesn't, then *expr1* serves no purpose.
- When the comma expression *++i, i + j* is evaluated, **i is first incremented, then i + j** is evaluated.
  - 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 (cont.)

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

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

as

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

- Since the left operand in a comma expression is evaluated before the right operand, the assignments  $i = 1$ ,  $j = 2$ , and  $k = i + j$  will be performed from left to right.

# The Comma Operator (cont.)

- The comma operator makes it possible to “glue” two expressions together to form a single expression.
- Certain macro definitions can benefit from the comma operator.
- The `for` statement is the only other place where the comma operator is likely to be found.

- Example:

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

- With additional commas, the `for` statement could initialize more than two variables.

# Program: Printing a Table of Squares (Revisited)

- The `square.c` program can be improved by converting its **while loop to a for loop**.

```
#include <stdio.h>                                square2.c
int main(void)
{
    int i, n;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++)
        printf("%10d%10d\n", i, i * i);

    return 0;
}
```

```
i = 1;
while (i <= n) {
    printf("%10d%10d\n", i,
              i * i);

    i++;
}
```

# Program: Printing a Table of Squares (Revisited) (cont.)

- C places **no restrictions on the three expressions** that control the behavior of a `for` statement.
- Although these expressions **usually initialize, test, and update** the same variable, there's **no requirement** that they be related in any way.
- The `square3.c` program is equivalent to `square2.c`, but contains a `for` statement that **initializes one variable (`square`)**, **tests another (`i`)**, and **increments a third (`odd`)**.
- The flexibility of the `for` statement can sometimes be useful, but in this case **the original program was clearer**.



# Program: Printing a Table of Squares (Revisited) (cont.)

## square3.c

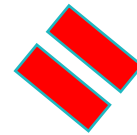
```
#include <stdio.h>
int main(void)
{
    int i, n, odd, square;

    printf("This program prints a table of squares.\n");
    printf("Enter number of entries in table: ");
    scanf("%d", &n);

    i = 1;
    odd = 3;
    for (square = 1; i <= n; odd += 2) {
        printf("%10d%10d\n", i, square);
        ++i;
        square += odd;
    }

    return 0;
```

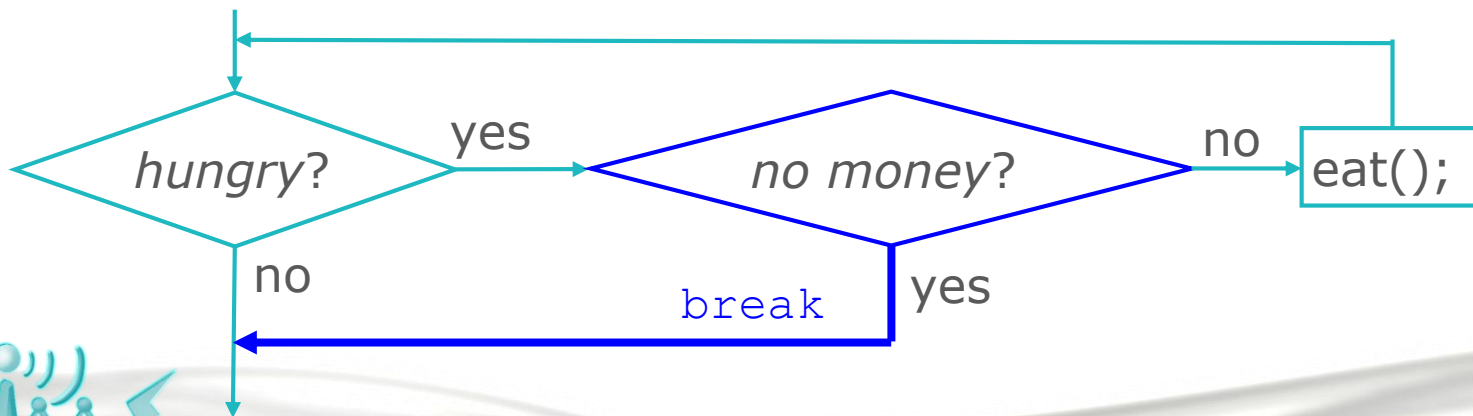
$$(x+1)^2 - x^2 = x^2 + 2x + 1 - x^2 = 2x + 1$$



```
for (i = 1, odd = 3, square = 1; i <= n;
     ++i, square += odd, odd += 2)
    printf("%10d%10d\n", i, square);
```

# 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).
- 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**.
- 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.



# The **break** Statement

- 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:

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

- After the loop has terminated, an **if** statement can be used to determine whether termination was premature (hence  $n$  isn't prime) or normal ( $n$  is prime):

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

# The **break** Statement (cont.)

- Loops that **read user input, terminating when a particular value is entered**, can use `break` to exit:


```
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 (cont.)

- A `break` statement transfers control out of the **innermost enclosing** `while`, `do`, `for`, or `switch`.
- When these statements are nested, the `break` statement can escape **only one level of nesting**.

- Example:

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

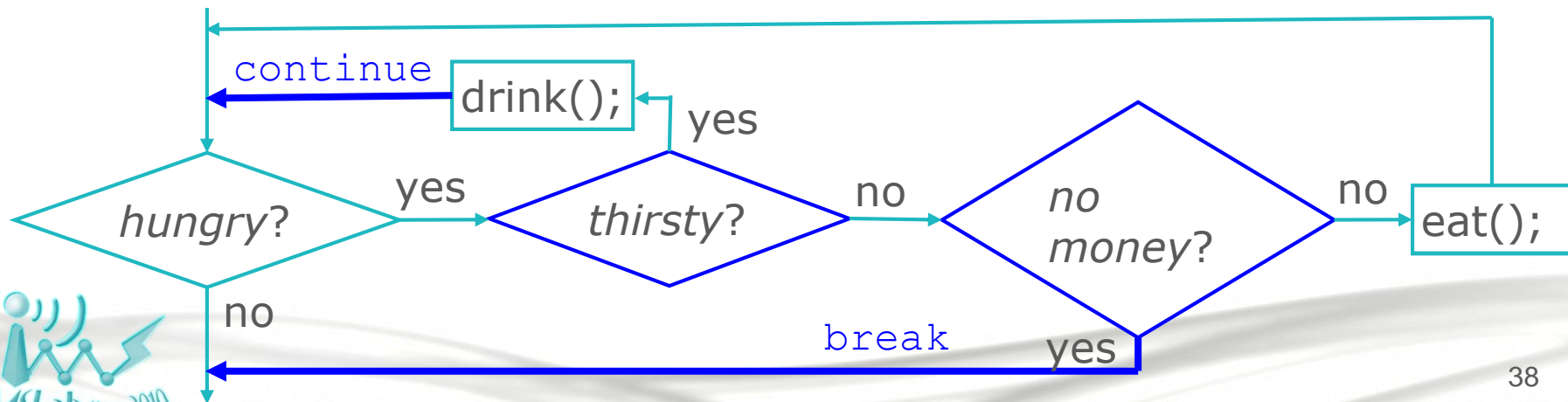


A blue line originates from the `break;` statement, moves horizontally to the right, then vertically down, and finally horizontally left to point at the closing curly brace of the `while` loop, illustrating that the `break` statement exits the `while` loop.

- `break` transfers control **out of the** `switch` statement, but **not out of the** `while` loop.

# The `continue` Statement

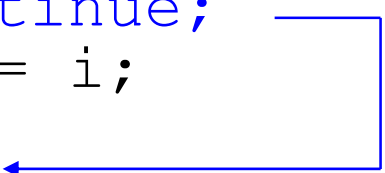
- The `continue` statement is **similar to `break`**:
  - `break` transfers control just **past the end** of a loop.
  - `continue` transfers control to a point just **before the end** of the loop body.
- With `break`, control **leaves the loop**; with `continue`, control **remains inside the loop**.



# The continue Statement (cont.)

- 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.**
- A loop that uses the `continue` statement:

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



without-continue version

```
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**.
- A **label** is **just an identifier** placed at the beginning of a statement:  
*identifier : statement*
- A statement may have **more than one label**.
- The `goto` statement itself has the form  
*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 (cont.)

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

```
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 (cont.)

- The `goto` statement is **rarely needed** in everyday C programming.
- The `break`, `continue`, and `return` statements—which **are essentially restricted goto statements**—and the `exit` function **are sufficient to handle most situations** that might require a `goto` in other languages.
- Nonetheless, the `goto` statement can be helpful once in a while.

# The goto Statement (cont.)

- Consider the problem of **exiting a loop from within a switch** statement.
- The `break` statement doesn't have the desired effect: it **exits from the switch**, but **not from the loop**.
- A `goto` statement solves the problem:

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

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

# Program: Balancing a Checkbook

- Many simple **interactive programs** present the user with **a list of commands** to choose from.
- Once a command is entered, the program **performs the desired action**, then **prompts the user for another command**.
- This **process continues until** the **user selects** an “exit” or “quit” command.

```
for (;;) {  
    prompt user to enter command;  
    read command;  
    switch (command) {  
        case command1: perform operation1; break;  
        ...  
        case commandexit: exit loop;  
        default: print error message; break;  
    }  
}
```

# Program: Balancing a Checkbook (cont.)

- The program allows the user to **clear the account balance**, **credit money** to the account, **debit money** from the account, **display the current balance**, and **exit** the program.

```
*** ACME checkbook-balancing program ***
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit
Enter command: 1
Enter amount of credit: 1042.56
Enter command: 2
Enter amount of debit: 133.79
Enter command: 1
Enter amount of credit: 1754.32
Enter command: 2
Enter amount of debit: 1400
Enter command: 2
Enter amount of debit: 68
Enter command: 2
Enter amount of debit: 50
Enter command: 3
Current balance: $1145.09
Enter command: 4
```

# Program: Balancing a Checkbook (cont.)

## checking.c

```
#include <stdio.h>
int main(void)
{
    int cmd;
    float balance = 0.0f, credit, debit;

    printf("*** ACME checkbook-balancing program ***\n");
    printf("Commands: 0=clear, 1=credit, 2=debit, ");
    printf("3=balance, 4=exit\n\n");
    for (;;) {
        printf("Enter command: ");
        scanf("%d", &cmd);
        switch (cmd) {
            case 0: /* clear */
                balance = 0.0f;
                break;
```

# Program: Balancing a Checkbook (cont.)

```
case 1: /* credit */
    printf("Enter amount of credit: ");
    scanf("%f", &credit);
    balance += credit;
    break;
case 2: /* debit */
    printf("Enter amount of debit: ");
    scanf("%f", &debit);
    balance -= debit;
    break;
case 3: /* display */
    printf("Current balance: $%.2f\n", balance);
    break;
case 4: /* exit */
    return 0;
default:
    printf("Commands: 0=clear, 1=credit, 2=debit, ");
    printf("3=balance, 4=exit\n\n");
    break;
```



# The Null Statement

- A statement can be **null**—devoid of symbols except for the semicolon at the end.
- The following line contains **three statements**:  

```
i = 0; ; j = 1;
```
- The null statement is primarily **good for one thing: writing loops whose bodies are empty**.



# The Null Statement (cont.)

- Consider the following prime-finding loop:

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

- 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.

# The Null Statement (cont.)

- Accidentally **putting a semicolon after** the parentheses in an `if`, `while`, or `for` statement creates a null statement.

- Example 1:

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

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);  
    --i;  
}
```

The extra semicolon creates an **infinite loop**.

# The Null Statement (cont.)

- Example 3:

```
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

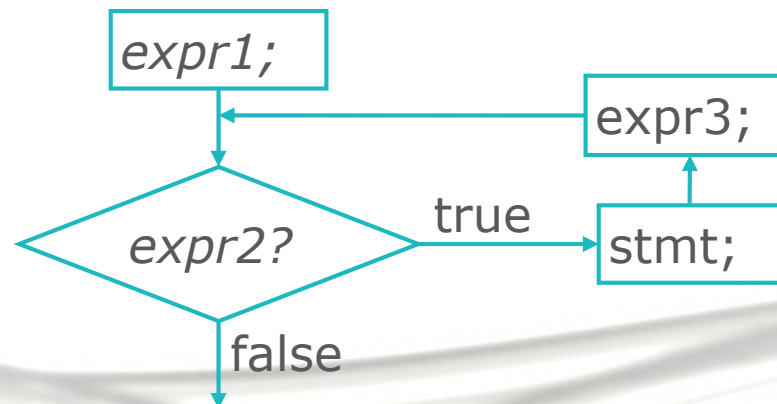
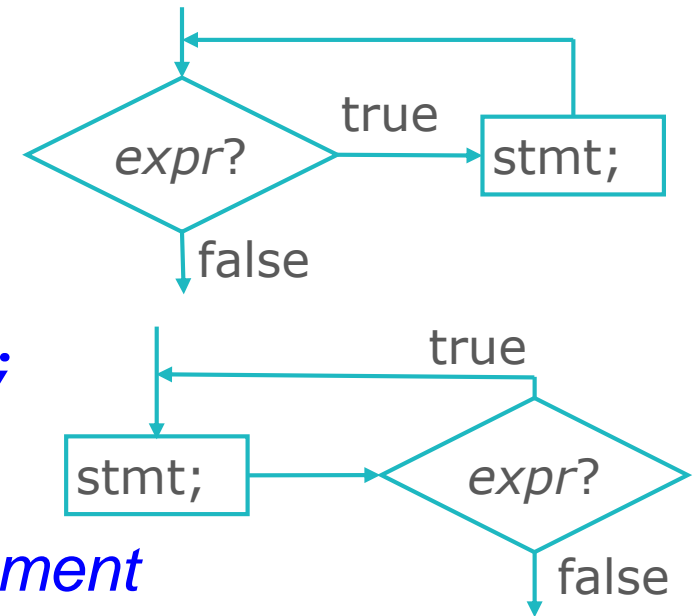
- Example 4:

```
for (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.

# A Quick Review to This Lecture

- while statement (the **most general**)  
`while ( expression ) statement`
- do statement (execute **at least once**)  
`do statement while ( expression ) ;`
- for statement (with **counting** variable)  
`for ( expr1 ; expr2 ; expr3 ) statement`



# A Quick Review to This Lecture (cont.)

- Infinite Loop (needs `break`, `goto`, `return` or `exit()` to leave)
  - `while (1) ...`
  - `do ... while (1);`    */\* rarely used \*/*
  - `for(;;) ...`
- `for` statement with **declaration** (variable **not visible outside**)
  - `for (int i = 0; i < n; i++) ...`
  - `for (int i = 0, j = 0; i < n; i++) ...`
- Comma operator (used in `macro` definition and `for` statement)

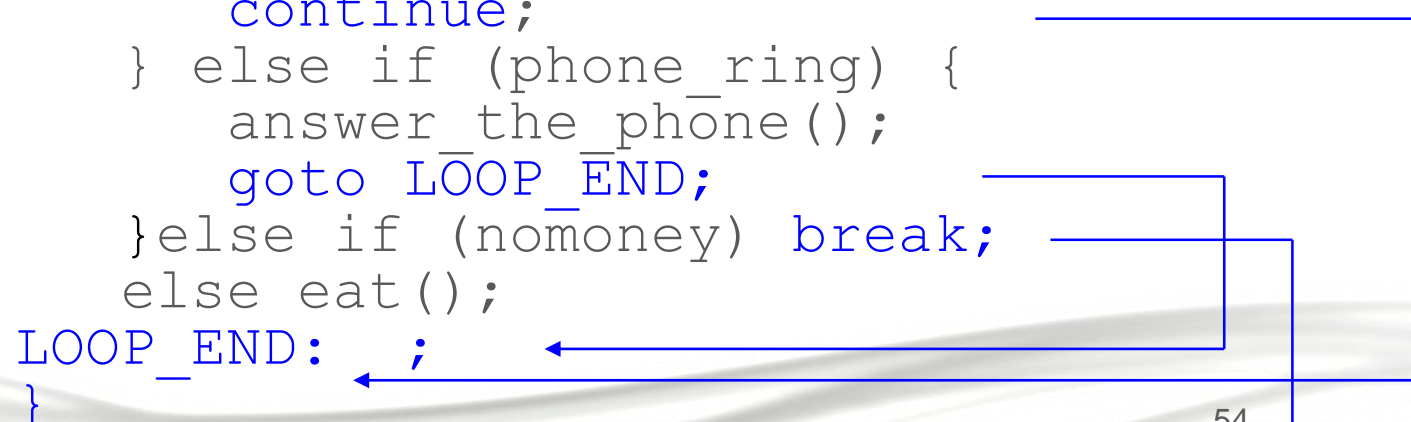
*expr1 , expr2*

← should have side effect      ← value of the entire expression

# A Quick Review to This Lecture (cont.)

- **break jumps out one level of** **switch,** while, do or for
- **(carefully used) continue jumps to the end (inside) of** while, do or for
- **(rarely used) goto jumps to any statement with specified label (inside function)**

```
while (hungry) {  
    if(thirsty) {  
        drink();  
        continue;  
    } else if (phone_ring) {  
        answer_the_phone();  
        goto LOOP_END;  
    } else if (nomoney) break;  
    else eat();  
LOOP_END: ;  
}
```



# A Quick Review to This Lecture (cont.)

- Null statement (useful for **loops with empty body**)

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

- Careless usage (body missing)

- `if (d == 0);`  
    `printf("always executed (once, of course)\n");`
- `while (i > 0);`  
    `printf("never executed (infinite loop) i=%d\n", i++);`
- `while (--i > 0);`  
    `printf("executed only once\n", i);`
- `for (i = 10; i > 0; i--);`  
    `printf(" executed only once\n", i);`