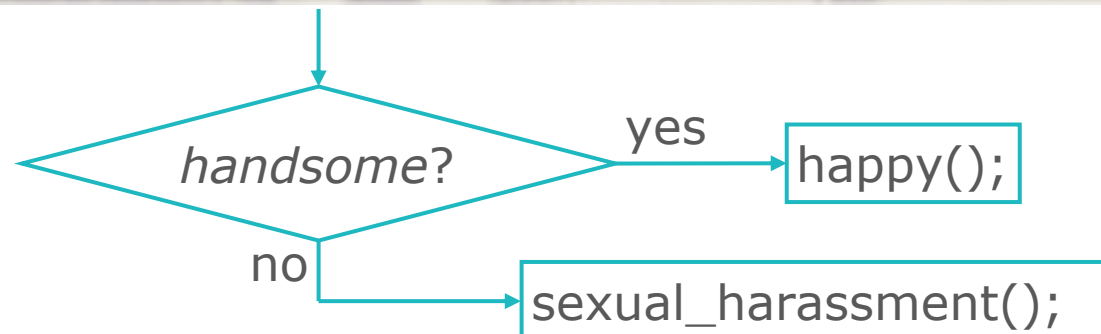


Lecture 3 - Control Structures

Meng-Hsun Tsai
CSIE, NCKU

```
if (handsome)
    happy();
else
    sexual_harassment();
```



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

3.1 Logical Expression



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`.
- In many programming languages, an expression such as `i < j` would have a special **"Boolean" or "logical" type**.
- **In C, 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

- These operators produce 0 (false) or 1 (true) when used in expressions.
- The relational operators can be used to compare integers and floating-point numbers, with operands of mixed types allowed.
- The precedence of the relational operators is lower than that of the arithmetic operators.
- For example, $i + j < k - 1$ means $(i + j) < (k - 1)$.

Relational Operators (cont.)

- The relational operators are **left associative**.

- The expression

$i < j < k$

is legal, but does not test whether j lies between i and k .

- Since the $<$ operator is left associative, this expression is equivalent to

$(i < j) < k$

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

- The **correct expression** is $i < j \ \&\& \ j < k$.

Equality Operators

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

`==` equal to

`!=` not equal to

- The equality operators are **left associative** and **produce** either **0 (false)** or **1 (true)** as their result.
- The equality operators have **lower precedence than** the **relational operators**, so the expression

`i < j == j < k`

is equivalent to

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

Logical Operators

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

! logical negation

& & logical *and*

| | logical *or*

- The ! operator is **unary**, while & & and | | are **binary**.
- The logical operators **produce 0 or 1** as their result.
- The logical operators treat **any nonzero operand** as a **true** value and **any zero operand** as a **false** value.

Logical Operators (cont.)

- Behavior of the logical operators:

!expr has the value **1** if *expr* has the value **0**.

expr1 && expr2 has the value **1** if the values of *expr1* and *expr2* are both nonzero.

expr1 || expr2 has the value **1** if either *expr1* or *expr2* (or both) has a nonzero value.

- In all other cases, these operators produce the value 0.

Logical Operators (cont.)

- Both `&&` and `||` perform “**short-circuit**” **evaluation**: they first evaluate the left operand, then the right one.
- If the **value** of the expression **can be deduced from the left operand alone**, the **right operand isn't evaluated**.
- Example: `(i != 0) && (j / i > 0)`
`(i != 0)` is evaluated first.
 - If `i` **isn't equal to 0**, then `(j / i > 0)` **is evaluated**.
 - If `i` **is 0**, the entire expression must be false, so **there's no need to evaluate** `(j / i > 0)`. Without short-circuit evaluation, **division by zero** would have occurred.

Logical Operators (cont.)

- Thanks to the short-circuit nature of the `&&` and `||` operators, side effects in logical expressions may not always occur.
- Example:
 - `i > 0 && ++j > 0`
 - If `i > 0` is false, then `++j > 0` is not evaluated, so `j` isn't incremented.
 - The problem can be fixed by changing the condition to `++j > 0 && i > 0` or, even better, by incrementing `j` separately.

Logical Operators (cont.)

- 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**.

3.2 The `if` Statement



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 (cont.)

- **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 (cont.)

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

```
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 (cont.)

- Example:

```
{ 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++;  
}
```

- Each **inner statement** still ends with a **semicolon**, but the **compound statement itself** does not.
- Compound statements **are also common in loops** and other places where the syntax of C requires a single statement.

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**.

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

- Inner statements are usually indented, but if they're **short** they can be **put on the same line** as the `if` and `else`:

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

The `else` Clause (cont.)

- It's not unusual for `if` statements to be **nested** inside other `if` statements:

```
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.

The `else` Clause (cont.)

- To avoid confusion, don't hesitate to add braces:

```
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;  
    }  
}
```

The `else` Clause (cont.)

- Advantages of using braces even when they're not required:
 - Makes programs easier to modify, because more statements can easily be added to any `if` or `else` clause.
 - Helps avoid errors that can result from forgetting to use braces when adding statements to an `if` or `else` clause.

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 (cont.)

- Aligning each `else` with the original `if` 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
```

```
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");
```


Program: Calculating a Broker's Commission

- When stocks are sold or purchased through a broker, the broker's commission often depends upon the value of the stocks traded.
- Suppose that a broker charges the amounts shown in the table:

- The minimum charge is \$39.
- The `broker.c` program asks the user to enter the amount of the trade, then displays the amount of the commission:

Enter value of trade: 30000

Commission: \$166.00

<i>Transaction size</i>	<i>Commission rate</i>
Under \$2,500	\$30 + 1.7%
\$2,500–\$6,250	\$56 + 0.66%
\$6,250–\$20,000	\$76 + 0.34%
\$20,000–\$50,000	\$100 + 0.22%
\$50,000–\$500,000	\$155 + 0.11%
Over \$500,000	\$255 + 0.09%

- The **heart** of the program is a **cascaded if statement** that **determines which range** the trade falls into.

Program: Calculating a Broker's Commission (cont.)

broker.c

```
1 #include <stdio.h>
2 int main(void)
3 {
4     float commission, value;
5     printf("Enter value of trade: ");
6     scanf("%f", &value);
7
8     if (value < 2500.00f)
9         commission = 30.00f + .017f * value;
10    else if (value < 6250.00f)
11        commission = 56.00f + .0066f * value;
12    else if (value < 20000.00f)
13        commission = 76.00f + .0034f * value;
14    else if (value < 50000.00f)
15        commission = 100.00f + .0022f * value;
16    else if (value < 500000.00f)
17        commission = 155.00f + .0011f * value;
18    else
19        commission = 255.00f + .0009f * value;
20    if (commission < 39.00f)
21        commission = 39.00f;
22
23    printf("Commission: $%.2f\n",
24        commission);
25    return 0;
26 }
```

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`.

The “Dangling `else`” Problem (cont.)

- To make the `else` clause part of the outer `if` statement, we can **enclose the inner `if` statement in braces**:

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

- **Using braces** in the original `if` statement would have **avoided the problem** in the first place.

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 expression is evaluated in stages: ***expr1*** is **evaluated first**;
 - if its value **isn't zero**, then ***expr2*** is evaluated, and its value is the value of the entire conditional expression.
 - if the value **is zero**, then the value of ***expr3*** is the value of the conditional.
- The conditional operator **requires three operands**, so it is often referred to as a **ternary operator**.

Conditional Expressions (cont.)

- 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 */
```

- 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 tend to make programs shorter but harder to understand, so it's probably best to use them sparingly.
- Conditional expressions are often used in return statements:

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

Conditional Expressions (cont.)

- 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

- **C** provides the `_Bool` type.
- A Boolean variable can be declared by writing
`_Bool flag;`
- `_Bool` is an **integer** type, so a `_Bool` variable is really just an integer variable in disguise.
- Unlike an ordinary integer variable, however, a `_Bool` variable **can only be assigned 0 or 1**.
- **Attempting to store a nonzero value** into a `_Bool` variable will cause the variable to **be assigned 1**:

```
flag = 5;    /* flag is assigned 1 */
```


Boolean Values (cont.)

- It's **legal** (although not advisable) to **perform arithmetic** on `_Bool` variables.
- It's also **legal** to **print** a `_Bool` variable (either 0 or 1 will be displayed).
- And, of course, a `_Bool` variable can be **tested** in an `if` statement:

```
if (flag)    /* tests whether flag is 1 */
```

```
...
```

Boolean Values (cont.)

- C's `<stdbool.h>` header makes it easier to work with Boolean values.
- It **defines a macro**, `bool`, that stands for `_Bool`.
- If `<stdbool.h>` is included, we can write

```
bool flag;    /* same as _Bool flag; */
```

- `<stdbool.h>` also supplies macros named `true` and `false`, which stand for **1** and **0**, respectively, making it possible to write

```
flag = false;
```

```
...
```

```
flag = true;
```

3.3 The `switch` Statement



The `switch` Statement

- A **cascaded if statement** can be used to **compare an expression against a series of values**:

```
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");
```

The `switch` Statement (cont.)

- The `switch statement` is an **alternative**:

```
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 (cont.)

- 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 (cont.)

- The word `switch` must be followed by an **integer expression** — the ***controlling expression*** — in parentheses.
- **Characters** are treated as **integers** in C and thus **can be tested** in `switch` statements.
- **Floating-point numbers** and **strings** **don't qualify**, however.

The `switch` Statement (cont.)

- Each case begins with a label of the form
`case constant-expression :`
- A ***constant expression*** is much like an ordinary expression except that it **can't contain variables or function calls**.
 - `5` **is** a constant expression, and `5 + 10` **is** a constant expression, but `n + 10` **isn't** a constant expression (**unless `n` is a macro** that represents a **constant**).
- The constant expression in a case label must evaluate to an **integer** (**characters** are acceptable).

The `switch` Statement (cont.)

- After each `case` label comes any number of statements.
- No braces are required around the statements.
- The last statement in each group is normally `break`.

The `switch` Statement (cont.)

- Duplicate case labels aren't allowed.
- The **order** of the cases **doesn't matter**, and the `default` case **doesn't need** to come **last**.
- 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;  
}
```

The `switch` Statement (cont.)

- To save space, several case labels can be put on the same line:

```
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**.

The Role of the `break` Statement

- Executing a `break` statement causes the program to “break” out of the `switch` statement; execution continues at the next statement after the `switch`.
- The `switch` statement is really a form of “computed jump.”
- When the controlling expression is evaluated, control jumps to the case label matching the value of the `switch` expression.
- A case label is nothing more than a marker indicating a position within the `switch`.

The Role of the **break** Statement (cont.)

- **Without `break`** (or some other jump statement) at the end of a case, control will **flow into the next case**.

- **Example:**

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

- **If the value of `grade` is 3**, the message printed is

GoodAveragePoorFailingIllegal grade

The Role of the `break` Statement (cont.)

- **Omitting `break`** is **sometimes** done **intentionally**, but it's **usually** just an **oversight**.
- It's a **good idea** to **point out deliberate omissions** of `break`:

```
switch (grade) {  
    case 4: case 3: case 2: case 1:  
        num_passing++;  
        /* FALL THROUGH */  
    case 0: total_grades++;  
        break;  
}
```

- Although the **last case** never needs a **`break` statement**, including one **makes it easy to add cases** in the future.

Program: Printing a Date in Legal Form

- Contracts and other legal documents are often dated in the following way:

Dated this _____ day of _____ , 20__ .

- The `date.c` program will display a date in this form after the user enters the date in month/day/year form:

Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.

- The program uses `switch` statements to add “th” (or “st” or “nd” or “rd”) to the day, and to print the month as a word instead of a number.

Program: Printing a Date in Legal Form (cont.)

date.c

```
#include <stdio.h>
int main(void)
{
    int month, day, year;

    printf("Enter date (mm/dd/yy): ");
    scanf("%d /%d /%d", &month,
        &day, &year);

    printf("Dated this %d", day);
    switch (day) {
        case 1: case 21: case 31:
            printf("st"); break;
        case 2: case 22:
            printf("nd"); break;
        case 3: case 23:
            printf("rd"); break;
        default: printf("th"); break;
    }
    printf(" day of ");

    switch (month) {
        case 1: printf("January"); break;
        case 2: printf("February"); break;
        case 3: printf("March"); break;
        case 4: printf("April"); break;
        case 5: printf("May"); break;
        case 6: printf("June"); break;
        case 7: printf("July"); break;
        case 8: printf("August"); break;
        case 9: printf("September"); break;
        case 10: printf("October"); break;
        case 11: printf("November"); break;
        case 12: printf("December"); break;
    }

    printf(", 20%.2d.\n", year);
    return 0;
}
```



```
Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.
```


3.4 `while` and `do` Statement



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.

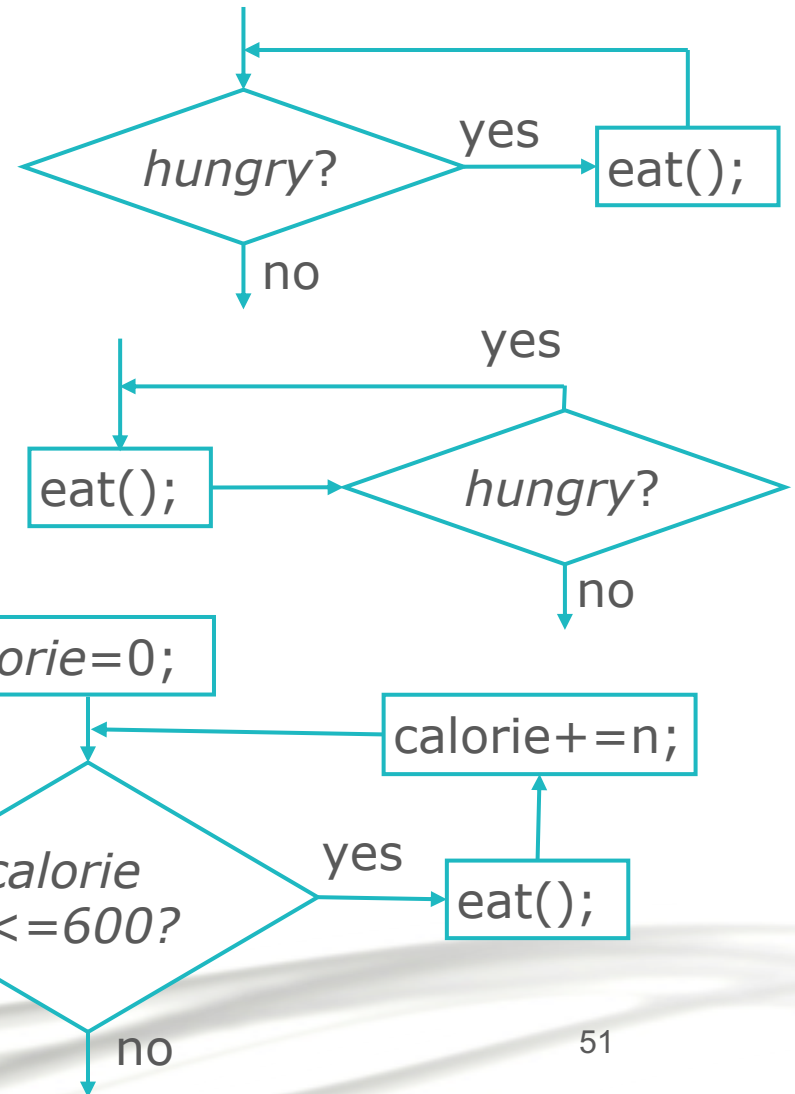
```
while (hungry)  
{  
    eat();  
}
```

→ **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**.



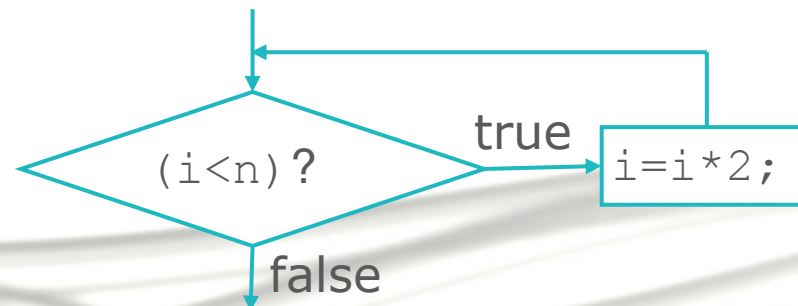
The `while` Statement

- 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 < 10</code> (true)	<code>i = 1 * 2</code> (= 2)
2	<code>2 < 10</code> (true)	<code>i = 2 * 2</code> (= 4)
3	<code>4 < 10</code> (true)	<code>i = 4 * 2</code> (= 8)
4	<code>8 < 10</code> (true)	<code>i = 8 * 2</code> (= 16)
5	<code>16 < 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

```
#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;
}
```

square.c

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;
```

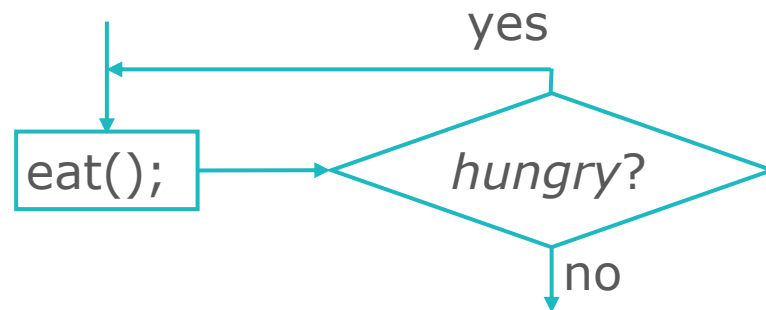
This program sums a series of integers.
Enter integers (0 to terminate): 8 23 71 5 0
The sum is: 107

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;
}
```


3.5 The `for` Statement

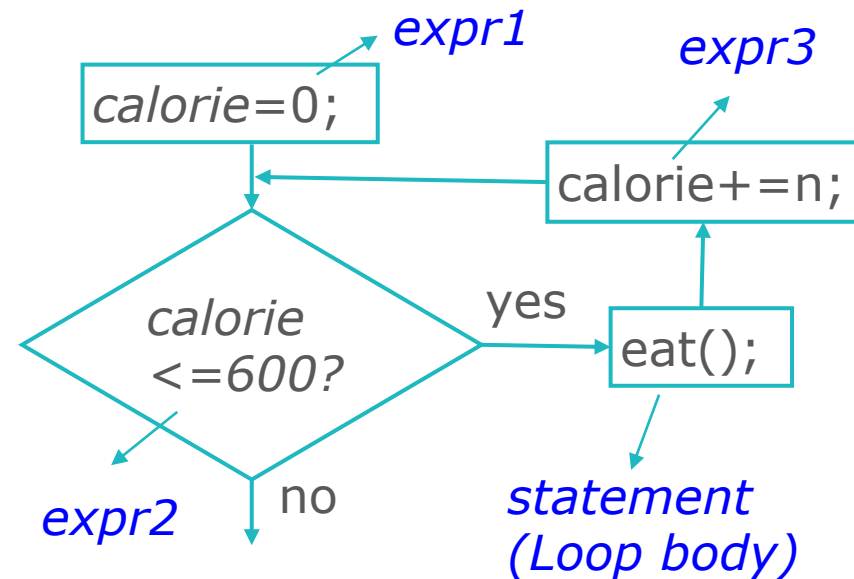


The `for` Statement

- 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 $<=$ 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**.
- 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).

<code>for (; ;) ...</code>		<code>while (1) ...</code>
------------------------------	---------------------------------------------------------------------------------------	----------------------------

Declare Variable in the First Expression

- 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 ***/
```

Declare Variable in the First Expression (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

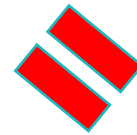
```
#include <stdio.h>
int main(void)
{
    int i, n, odd, square;

    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;
```

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



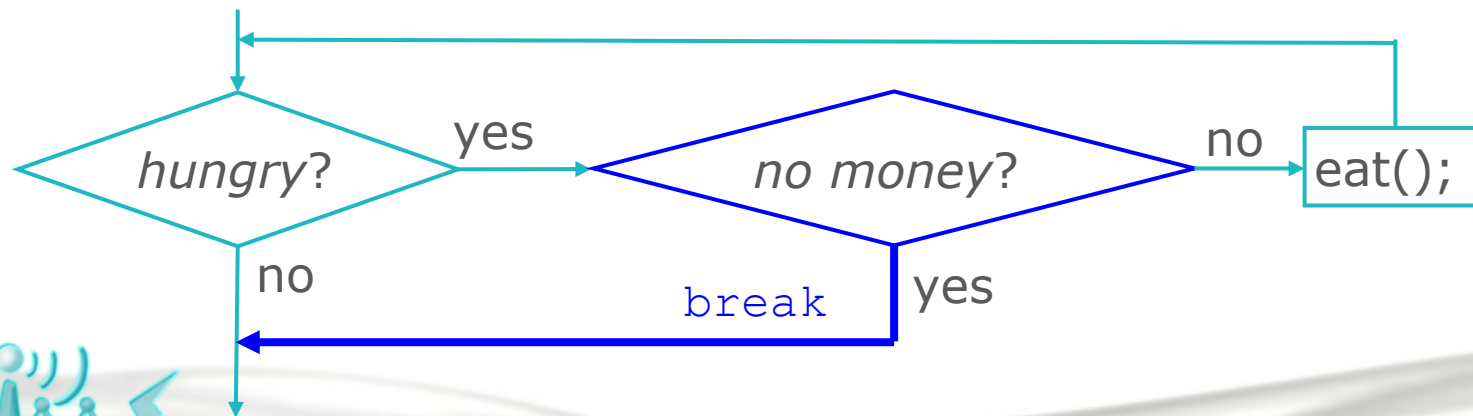
```
for (i = 1, odd = 3, square = 1; i <= n;
     ++i, square += odd, odd += 2)
    printf("%10d%10d\n", i, square);
```

3.6 Exiting from a Loop



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:

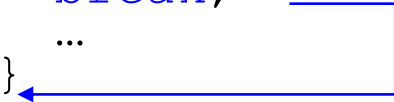
```
for (;;) {  
    printf("Enter a number (enter 0 to stop): ");  
    scanf("%d", &n);  
    if (n == 0)  
        break;  
    printf("%d cubed is %d\n", n, n * n * n);  
}
```

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;  
        ...  
    }  
}
```

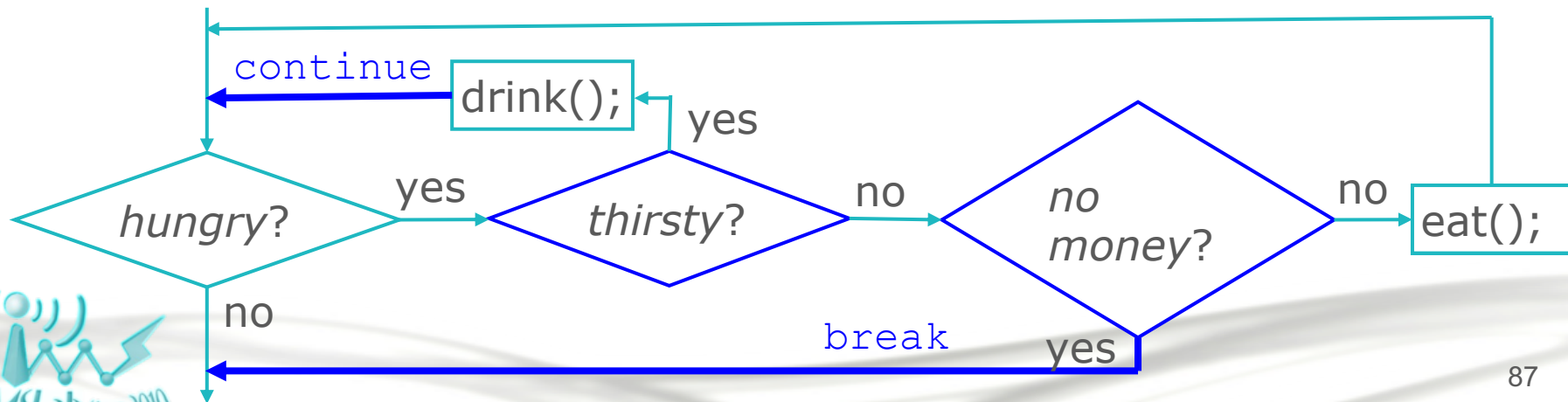


A blue line with an arrowhead at the end starts from the `break;` statement, moves right, then down, then left, ending with an arrowhead pointing to the closing brace of the `while` loop, illustrating that the `break` statement exits the `while` loop.

- `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

- `if` statements
 - `if (expression) statement`
`if (expression) { statements }`
 - `if (expression) statement else statement`
`if (expression) { statements } else { statements }`
 - Logical Expression
 - yields an integer: either 0 (false) or 1 (true).
 - Relational Operators: `<` `>` `<=` `>=`
 - Equality Operators: `==` `!=`
 - Logical Operators: `!` `&&` `||`
- left associative
- right associative and unary

A Quick Review to This Lecture (cont.)

- To test whether j lies between i and k , use
 $i < j \ \&\& \ j < k$ instead of $i < j < k$ ❌
- Logical operators treat **any nonzero operand** as a **true** value and **any zero operand** as a **false** value.
- Both $\&\&$ and $||$ perform “**short-circuit**” evaluation:
 $(i \neq 0) \ \&\& \ (j / i > 0)$
If the **value** of the expression **can be deduced** from the **left operand alone**, the **right operand isn't evaluated**.
- **Confusing $==$ with $=$** is perhaps the **most common error**.
 $\text{if } (i == 0) \dots$ tests whether i is **equal** to 0.
 $\text{if } (i = 0) \dots$ **assigns** 0 to i , then **tests** i (zero/nonzero)

A Quick Review to This Lecture (cont.)

Nested if statement

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

Cascaded if statement

```
if (n < 0)  
    printf("n is less than 0\n");  
else if (n == 0)  
    printf("n is equal to 0\n");  
else if (n <= 1)  
    printf("n is between 0 and 1\n");  
else  
    printf("n is greater than 1\n");
```

A Quick Review to This Lecture (cont.)

- the “dangling else” problem (using braces to avoid it)

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

- Conditional Expressions

```
k = (i > j) ? i : j;
```


A Quick Review to This Lecture (cont.)

- Boolean Values
 - Using **macro**

```
#define BOOL int
#define TRUE 1
#define FALSE 0

BOOL flag;
flag = TRUE;
if (flag == FALSE) ... /* or if(!flag) */
```

- Using **_Bool**

```
#define TRUE 1
#define FALSE 0

_Bool flag;
flag = TRUE;
if (flag == FALSE) ... /* or if(!flag) */
```

A Quick Review to This Lecture (cont.)

- Include `<stdbool.h>`

```
#include <stdbool.h>
bool flag;
flag = true;
if (flag == false) ... /* or if(!flag) */
```

- `switch` Statement (**faster** and **easier to read** than **cascaded if**)

No variables **or** function calls integer (or character) expression

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

A Quick Review to This Lecture (cont.)

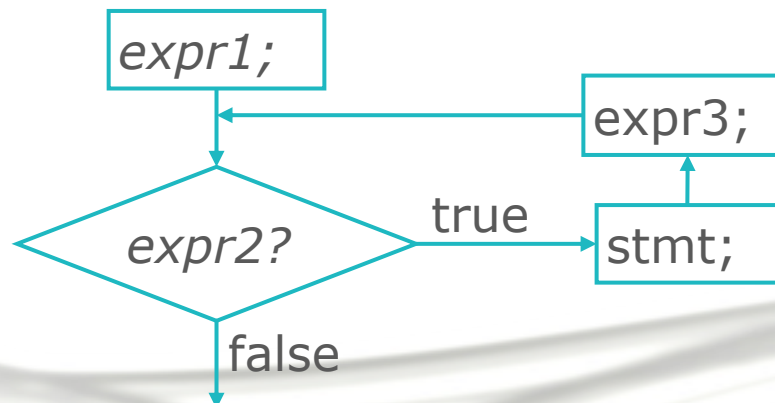
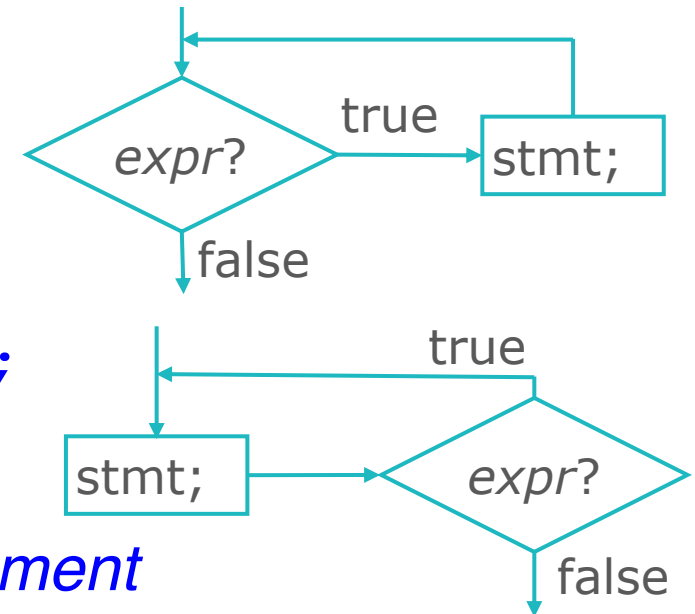
- **Without break** (or some other jump statement) at the end of a case, control will **flow into the next case**.

```
switch (grade) {  
    case 4: case 3: case 2: case 1:  
        num_passing++;  
        /* FALL THROUGH */  
    case 0: total_grades++;  
        break;  
}
```

- A **case label** is **nothing more than a marker** indicating a position within the switch.
- 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**.

A Quick Review to This Lecture (cont.)

- 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

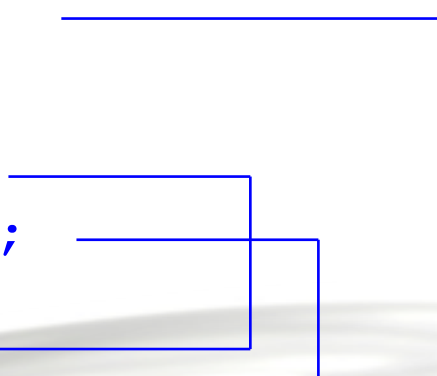
should have side effect

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);`