4. Control Statements

2017 Spring

Yusung Kim
yskim525@skku.edu

Statements

- So far, we've used expression statements.

- Normally, statements in a program are executed one after the other. This is called **sequential** execution.

- There are control statements:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while` and `for`
  - ***Jump statements:*** `break` and `continue`.

Logical Expressions

- For the expression `i < j`;
  return a true value if `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
  `==` equal to
  `<=` less than or equal to
  `>=` greater than or equal to

- These operators produce 0 (false) or 1 (true) when used in expressions.

- The precedence of the relational operators is lower than that of the arithmetic operators.

  For example: `i + j < k - 1` means `(i + j) < (k - 1)`.

- The relational operators are left associative.

Logical Operators

!	logical negation
&&	logical *and*
||	logical *or*

- 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.

Combination of Relational and Logical Operators

- Is the following expression legal ?

```
i < j < k
```

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

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

`(i < j)` will yield 0 or 1.

- The correct expression is `i < j && j < k`.

The `if` statement

- The `if` statement allows a program to choose between two alternatives by testing an expression;

```
if ( expression ) statement
```

After ***expression*** is evaluated, if its value is nonzero, ***statement*** is executed.

```
if (line_num < 0)
{
        line_num = 0;
}
printf("line_num = %d\n", line_num);
```

The `else` statement

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

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

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

```
if (v1 > v2)
        max = v1;
else
        max = v2;
```

Nested `if` statement

- `if` statements can 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.

`else if` statement

- `else 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
```

Broker's Commission Program

- Purchasing items through a broker, the broker's commission often depends upon transaction size.
- Suppose that a broker charges the amounts shown in the following table (the minimum charge is **$39**):

Transaction size	*Commission rate*
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%

Broker's Commission Program

- 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
```

```
/* Calculates a broker's commission */

#include <stdio.h>

int main(void)
{
    float commission, value;

    printf("Enter value of trade: ");
    scanf("%f", &value);
```

```
    printf("Commission: $%.2f\n", commission);

    return 0;
}
```

The `switch` Statement

- A cascaded `if` statement is 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

- 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

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

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

- The word `switch` must be followed by an integer expression.
  - Characters are treated as integers in C

The `switch` Statement

`case` *constant-expression* **:**

- A ***constant expression*** can't contain variables or function calls.
  - `5 + 10` is a constant expression, but `n + 10` isn't a constant expression.
- The constant expression in a case label must evaluate to an integer (characters are acceptable).
- 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

- Duplicate case labels aren't allowed.
- 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 Role of the **`break`** Statement

- Executing a `break` statement causes the program to "break" out of the `switch` statement.
- Without `break` at the end of a case, control will flow into the next case.

```
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");
 }
```

When `grade` is 3, the message printed is

```
GoodAveragePoorFailingIllegal
```

Repetition Essentials

- A loop is a group of instructions that the computer executes repeatedly while its **loop condition** remains true.

- We have three means of repetition.
  - **definite repetition** : we know in advance exactly how many times the loop will be executed.
  - **indefinite repetition** : it's not known in advance how many times the loop will be executed.
  - **Infinite repetition:** don't stop if the loop condition is always true.

The `while` statement

- A `while` statement is the easiest way to set up a loop.

```
int i=0;
while (i < 10){   /* loop condition */
  i++;            /* loop body */
}
```

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

While Loop Examples

```
int i=0, n;

scanf("%d",&n);
while (i < n){
        i++;
}
```

```
int passwd;

scanf("%d",& passwd);
while (passwd != 1111 ){
        printf("wrong passwd!\n");
        scanf("%d",& passwd);
}
```

```
while (1){
        printf("sense temperature\n");
        sleep(1);
}
```

Program: Summing a Series of Numbers

- `sum.c` program sums a series of integers entered by the user:

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

- The program will need a loop that uses `scanf` to read a number and then adds the number to a running total.

sum.c

```
/* Sums a series of numbers */
 #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;
}
```

The `do-while` Statement

- General form of the do-while statement:

```
do { statements; } while(expression);
```

```
int i = 10;
do {
    i--;
    printf("Count down: %d\n", i);
} while (i > 0);
```

- The **loop body** is executed first, then the control expression is evaluated.
- The body of a statement is always executed **at least once**.

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.

numdigits.c

```
/* Calculates the number of digits in an integer */

#include <stdio.h>

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

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









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

  return 0;
}
```

The `for` statement

- The `for` statement has a “counting” variable.
- General form of the `for` statement:

  `for ( `*expr1* `; `*expr2* `; `*expr3* `) { `*statements;* `}`

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

The `for` statement

```
for (i = 0; i < 10; i++)
    printf("i = %d\n", i);
```

- *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 an operation to be performed at the **end of each loop iteration**.

Program: Printing a Table of Squares

```
/* Prints a table of squares using a for statement */
 #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);

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

  return 0;
}
```

Exiting from a Loop

- The normal exit point for a loop is at the **condition expression** in a `while` or `for` statement.
- Using the **`break`** statement, it's possible to provide an exit point in the middle or multiple exit points.

Break example

- Checking whether an input password `guess` is a correct password.
- We can use a `break` statement to terminate the loop as soon as the password is correct.

```
int guess;
int passwd = 1111;
while (1)
{
    scanf("%d",&guess);
    if (guess == passwd)
    break;
}
```

`break` in Nested Loops

- A `break` statement transfers control out of the innermost enclosing `while`, `do`, `for`, or `switch`.

  It means that when multiple clauses are nested, the `break` statement can escape only one level of nesting.

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

- `break` stops the `for` loop, but not the `while` loop.

The `continue` Statement

- A loop can use the `continue` statement:

```
int cnt=0, sum=0, num=0;
while (cnt < 10) {
  scanf("%d", &num);

  if (num == 0)
    continue;

  sum += num;
  cnt++;
  /* continue jumps to here */
}
```

The `continue` Statement

- The same loop written without using `continue`:

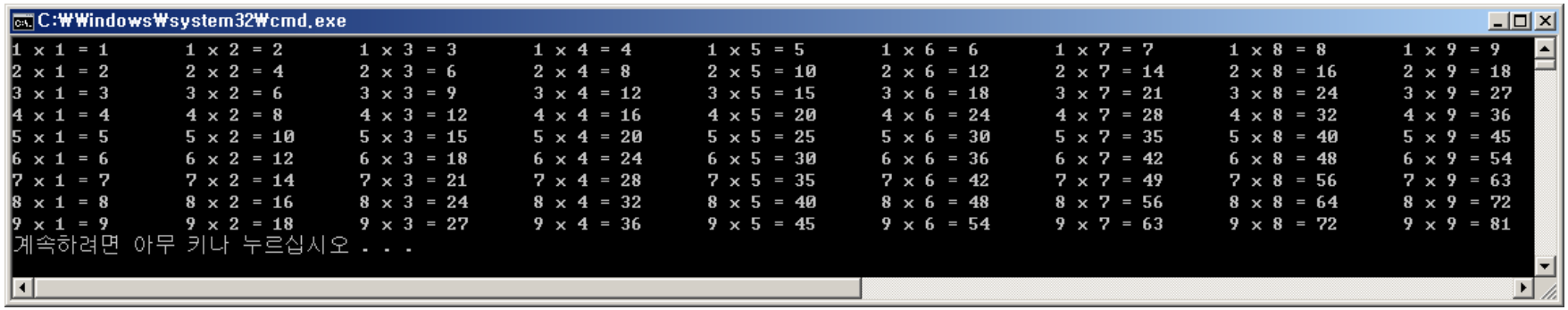
```
int cnt = 0, num = 0, sum = 0;
while (cnt < 10) {
  scanf("%d", &num);

  if (num != 0) {
    sum += num;
    cnt++;
  }
}
```

Nested Loop Statements

- You may use a loop statement inside another loop statement.

```
for (int i=1; i<10; i++){
        for (int j=1; j<10; j++){
                printf("%d x %d = %d\t", i,j,i*j);
        }
        printf("\n");
}
```



Summary

- Many problems require repeated processes.
- Computer can repeat more fast and exact than humans
- According to the problem characteristics;
  - The loop conditions are different
  - The number of nested loops is different
  - Selection statements can be used inside loops