Chapter 6

Loops

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 is not zero) the loop continues to execute

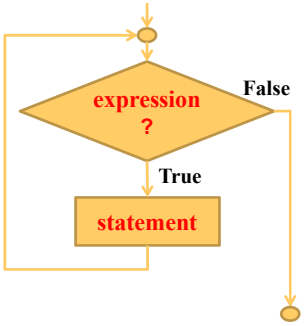
Iteration Statements

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

The `while` Statement

- Using a `while` statement is the easiest way to set up a loop
- The `while` statement has the form

  `while (`*expression*`)` *statement*
- *expression* is the **controlling expression**
- *statement* is the loop body



The `while` Statement

- Example of a `while` statement

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

- When a `while` statement is executed, the controlling expression is evaluated first
- 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 (false)

The `while` Statement

- Example: A `while` statement that computes the smallest 2^m that is greater than or equal to a number `n`:

```
i = 1; n = 10; /* i here means 2 to the power m */
while (i < n)
  i = i * 2; /* You can also write it as i *= 2 */
```

- Some programmers always use braces (*to be in the safe side*)

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

- A trace of the loop:

```
i = 1;
```

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

The `while` Statement

- If multiple statements are needed, use braces to create a single compound statement
- The following statements display a series of "countdown" messages:

```
i = 4;
while (i > 0)
{ printf("Counting down %d\n", i);
  i--;
}
```

- This program prints:

```
Counting down 4
Counting down 3
Counting down 2
Counting down 1
```


The `while` Statement

- Observations about the `while` statement:
  - The *controlling expression* is false when a `while` loop terminates;
    For example, 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

Infinite Loops

- A `while` statement will not 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, e.g., `exit`

Program: Printing a Table of Squares

- The `square.c` program uses a `while` statement to print a table of squares
- The user specifies the number of entries in the table:

```
This program prints a table of squares.
Enter number of entries in table: 5
          1          1
          2          4
          3          9
          4         16
          5         25
```

square.c

```
/* Prints a table of squares using a while 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);

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

  return 0;
}
```


Program: Summing a Series of Numbers

- The `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

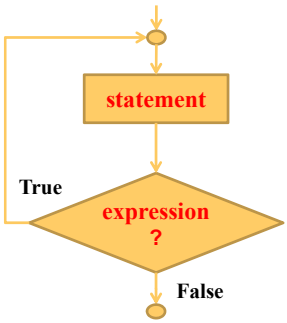
sum.c

```
/* 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 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 zero (false), the loop will break, otherwise, the loop will continue



The do Statement

- The main loop of the `sum.c` example can be rewritten as a `do` statement as follow:

```
do
{ scanf("%d", &n);
  sum += n;
} while (n != 0);
```

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

- 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

- It is a good idea to use braces around the body of the `do`, even if it is just one statement, because a `do` statement without braces can easily be mistaken for a `while` statement:

```
do
printf("Counting down %d\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

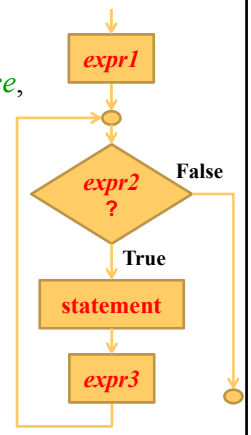
numdigits.c

```
/* 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);

  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, *expr2*, and *expr3* are expressions
- *expr1*: an initialization step that is *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*: an operation to be performed *at the end of each loop iteration*
- Example:

```
for (i = 4; i > 0; i--)
  printf("Counting down %d\n", i);
```

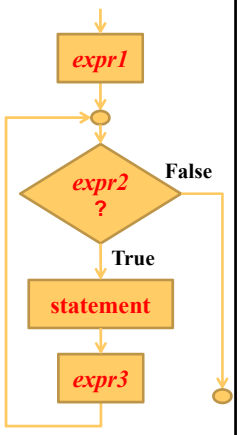


The `for` Statement

- A `for` loop can always be replaced by an equivalent `while` loop:
  `for (`*expr1*`;` *expr2*`;` *expr3*`)` *statement*

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

- *expr1*: an initialization step that is 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*: an operation to be performed at the end of each loop iteration



The `for` Statement

- Example

```
for (i = 4; i > 0; i--)
  printf("Counting down %d\n", i);
```

is equivalent to:

```
i = 4;
while (i > 0)
{ printf("Counting down %d\n", i);
  i--;
}
```


The `for` Statement

- The `for` statement is usually the best choice for loops that:
  - "count up" (increment a variable) or
  - "count down" (decrement a variable)
- Examples:

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--) …`

The `for` Statement

- Common `for` statement errors:
  - 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`
  - Using `>` instead of `<` (or vice versa) in the *controlling expression*
    - "Counting up" loops should use the `<` or `<=` operator
    - "Counting down" loops should use `>` or `>=`

Omitting Expressions in a `for` Statement

- `C` allows *any* or *all* of the expressions that control a `for` statement to be omitted ***(this is not the case with*** `while`***)***
- If the *first* expression is omitted, no initialization is performed at the beginning of the loop:

```
i = 4;
for (; i > 0; --i)
  printf("Counting down %d\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 = 4; i > 0;)
  printf("Counting down %d\n", i--);
```

Omitting Expressions in a `for` Statement

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

```
for (; i > 0;)
  printf("Counting down %d\n", i--);
```

is the same as

```
while (i > 0)
  printf("Counting down %d\n", i--);
```


Omitting Expressions in a `for` Statement

- If the *second* expression is missing,
  - it means always true
  - the `for` statement does not terminate; 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, e.g., `exit`
- Some programmers use the following `for` statement to establish an infinite loop:

```
for (;;) …
```

`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 and initialize a variable for use by the loop:

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

- The variable `i` need not to have been declared prior to this statement

`for` Statements in C99

- A variable declared by a `for` statement ***can not*** be accessed outside the body of the loop (we say that it is ***invisible*** outside the loop):

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

`for` Statements in C99

- Having a `for` statement declares its own control variable is:
  - usually a good idea
  - convenient and it can make programs easier to understand
- However, if the program needs to access the variable after loop termination, it is 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
  - 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

  expr-a , *expr-b*

  where *expr-a* and *expr-b* are any two expressions

The Comma Operator

- A comma expression is evaluated in two steps:
  - First, *expr-a* is evaluated and its value is discarded
  - Second, *expr-b* is evaluated; its value is the value of the entire expression
- Evaluating *expr-a* should always have a side effect
  - if it does not, then *expr-a* 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

- 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)))
```

The Comma Operator

- 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 (Section 6.1) can be improved by converting its `while` loop to a `for` loop

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

can be converted to

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

square2.c

```
/* 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("%10d%10d\n", i, i * i);

  return 0;
}
```


Program: Printing a Table of Squares (Revisited)

- 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 is 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`)

Program: Printing a Table of Squares (Revisited)

- **Odd Value: 1 3 5 7 ... 2×n-1**
- **Term: 1 2 3 4 ... n**
- **Odd Value: 2×1-1 2×2-1 2×3-1 2×4-1 2×n-1**

- The sum of an arithmetic series is:
  `0.5×(first + last) × number_of_terms`
  `0.5×(1 + 2×n-1) × n`
  `0.5×(2 × n) × n`
  `n×n`
- The flexibility of the `for` statement can sometimes be useful, but in this case the original program was clearer

square3.c

```
/* Prints a table of squares using an odd method */

#include <stdio.h>

int main(void)
{
  int term, n, odd, square;

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

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

  return 0;
}
```

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 (as in a `do` statement)
- Using the `break` statement, it is possible to write a loop
  - with an exit point in the middle of a loop
  - with more than one exit point
- The `break` statement can be used to transfer control and jump:
  - out of a `switch` statement
  - 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 use to determine whether the termination was
  - premature (hence `n` is not prime) or
  - normal (`n` is prime):

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

The `break` Statement

- The `break` statement is particularly useful for writing loops that read user input, terminating when a particular value is entered:

```
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

- When these statements are nested, the `break` statement can *escape only one level* of nesting (the innermost enclosing `while`, `do`, `for`, or `switch`)
- Example:

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

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

The `continue` Statement

- The `continue` statement forces the next iteration of a loop to take place, skipping any code between itself and the conditional expression that controls the loop
- Example on a loop that uses the `continue` statement:

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


The `continue` Statement

- The same loop written without using `continue`:

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

`break` Statement vs `continue` Statement

- A comparison between `break` and `continue` statements:
  - `break` transfers control just *after* the end of a loop
  - With `break`, control leaves the loop
  - `break` can be used in `switch` statements and loops (`while`, `do`, and `for`)
  - `continue` transfers control just *before* the end of the loop body
  - with `continue`, control remains inside the loop
  - `continue` is *limited to* only loops (`while`, `do`, and `for`)

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

- the goto statement can be helpful once in a while
- Consider the problem of *exiting a loop* from within a switch statement
- The break statement does not 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 will not 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
- The heart of such a program will be a loop:

```
for (;;)
{  prompt user to enter command;
   read command;
   execute command;
}
```

Program: Balancing a Checkbook

- Executing the command will require a `switch` statement (or cascaded `if` statement):

```
for (;;)
{ prompt user to enter command;
  read command;
  switch (command)
  { case command1: perform operation1; break;
    case command2: perform operation2; break;
      .
      .
      .
    case commandn: perform operationn; break;
    default: print error message; break;
  }
}
```


Program: Balancing a Checkbook

- The `checking.c` program, which maintains a checkbook balance, uses a loop of this type
- The user is allowed to clear the account balance, credit money to the account, debit money from the account, display the current balance, and exit the program

Program: Balancing a Checkbook

```
*** checkbook-balancing program ***
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit

Enter command: 0
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
```


checking.c

```
/* Balances a checkbook */

#include <stdio.h>

int main(void)
{
  int cmd;
  float balance = 0.0f, credit, debit;

  printf("*** 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:
        balance = 0.0f;
        break;
```

```
      case 1:
        printf("Enter amount of credit: ");
        scanf("%f", &credit);
        balance += credit;
        break;
      case 2:
        printf("Enter amount of debit: ");
        scanf("%f", &debit);
        balance -= debit;
        break;
      case 3:
        printf("Current balance: $%.2f\n", balance);
        break;
      case 4:
        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

- 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

- 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` is not inside the `if` statement, so it is performed regardless of whether `d` is equal to `0`

- **Example 2:**

```
i = 10;
while (i > 0);                        /*** WRONG ***/
{
  printf("Counting down %d\n", i);
  --i;
}
```

The extra semicolon creates an infinite loop

The Null Statement

- **Example 3:**

```
i = 4;
while (i-- > 0);                /*** WRONG ***/
  printf("Counting down %d\n", i);
```

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

```
Counting down 0
```

- **Example 4:**

```
for (i = 4; i > 0; i--);        /*** WRONG ***/
  printf("Counting down %d\n", i);
```

Again, the supposedly loop body is executed only once, the message printed is:

```
Counting down 0
```