Iteration / exiting from loops

CSC100 / Introduction to programming in C/C++ - Spring 2025

Marcus Birkenkrahe

April 12, 2025

README

- This script introduces C looping structures.
- This section is based on chapter 4 in Davenport/Vine (2015) and chapter 6 in King (2008).

Overview

- Loops can have exit points before (`while`, `for`) or after (`do`) the loop body.
- You can exit a loop (or any other statement) in the middle, too using: `break`, `continue`, and `goto`, (and `return`).

The `break` statement

- Remember the use of `break` after a `switch` statement:

```
switch (...) {
  case 1:
    ...
    break;
  case 2:
  ...
}
```

- Likewise, `break` can be used to jump out of a `while`, `do` or `for` loop.

- Especially useful when breaking a loop as soon as a particular value is entered.

Example

- Let's create an input file. We want to break a loop as soon as the number `0` is reached.

```
echo 10 9 8 7 6 5 4 3 2 1 0 > ../data/break_input
cat ../data/break_input
```

- Here's some code: what does it do? What would happen without the `break` statement? Would you know how to test that?

```
int n;
for (;;) {
  scanf("%d", &n);
  if (n == 0) break;
  printf("loop: n is %d\n", n);
 }
printf("n is %d\n", n);
```

- A good way to check/record an algorithm: pseudo code!

  Here is the pseudo code for the program **with** `break`:

```
for ever
    scan an integer
    if integer is 0
       break for loop
    else
       print the integer
print the integer (0)
```

  Here is the pseudo code for the program **without** `break`:

```
for ever
    scan an integer
    if integer is 0
       print the integer
```

- Let's tangle the code and run it with/without the `break` on the command line.

Practice

- **Important:** the `break` statement only breaks out of the **innermost** loop statement. If statements are nested, it can only escape **one** level of nesting.
- Example: The `break` only gets you out of the `switch` but not the `while` statement.

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

- **Do-It-Yourself practice:**

  1. Open Emacs, create a file `break.org`, put in the appropriate header, and construct an example demonstrating this behavior of `break`.
  2. For the `while` loop, re-use the counting program, counting up to 3.
  3. For the `switch ... case` selection, label the cases 1,2,3 and print the label.

The `continue` statement

- The `continue` statement does not exit from a loop. It brings you to a point just before the end of the loop body.
- With `break`, control leaves the loop, with `continue`, control remains inside the loop.
- `continue` is limited to loops, it does not work with `switch`.

Example: summing up numbers.

The loop terminates when 10 non-zero numbers have been read. Whenever the number `0` is read, `continue` is executed, the rest of the loop body is skipped, but we're still inside the loop.

Input file:

```
echo 1 1 1 1 1 1 1 1 0 1 1 > continue
cat continue
```

Pseudo code:

```
while n smaller than 10
  get input i           // scanf
  if input is 0 go on   // continue
  else add input to sum // sum += i
  increment n           // n++
print sum               // printf
```

Code:

```
int n=0, sum = 0;
int i;

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

Practice: world without `continue`

What if there was no `continue` available?

Download the practice file `continue.org` and change the program accordingly, from: `tinyurl.com/475m5x4n`

The `goto` statement

- The `goto` statement can jump to *any* statement in a function provided the function has a *label*.
- A *label* is an identifier placed at the beginning of a statement (known to you from the `switch...case` selection statement):

  ```
  identifier :  statement
  ```

  A statement can have more than one label. The `goto` statement looks like this:

  ```
  goto identifier ;
  ```

- Here is an example using `goto` to exit prematurely from a loop.

  The program looks for prime numbers.

  ```
  int d, n = 3;
  for (d = 2; d < n; d++ )
    printf("%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, d);
  ```

  ```
  2
  3 is prime
  ```

- Once, the use of `goto` was very common, but programs with `goto` statements tend to be hard to debug.
- A good use for `goto` is during debugging, because you can jump ship when an exception occurs, and run a small test routine (designing a function to do this is an alternative).

Extended example: balancing a checkbook

- Let's develop a program that maintains a checkbook balance.
- The program will offer the user a menu of choices:
  1. clear the account balance
  2. credit money to the account
  3. debit money from the account
  4. display the current balance
  5. exit the program
- These choices are represented by integers 0,1,2,3,4 resp. which are implemented as `switch` `case` labels.
- Here is a sample program session with the compile program `checking`:

```
pi@raspberrypi:~$ ./checking
--- ACME checkbook-balancing program ---
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit

Enter command: 3
Current balance: $0.00
Enter command: 1
Enter amount of credit: 100.00
Enter command: 3
Current balance: $100.00
Enter command: 2
Enter amount of debit: 50.00
Enter command: 3
Current balance: $50.00
Enter command: 4
pi@raspberrypi:~$
```

When the user enters the command `4` (exit), the program needs to exit from the `switch` statement *and* the surrounding loop: the `break` statement won't help, and we prefer not to use a `goto` statement. Instead, the program executes a `return` statement, which will cause the `main` function to return to the operating system.

- Pseudo code:

```
for ever until exit (4)
    Get input cmd (0...4)
    cmd = 0:
      clear balance
    cmd = 1:
      get credit amount
      credit amount to balance
    cmd = 2:
      get debit amount
      subtract amount from balance
    cmd = 3:
      print current balance
    cmd = 4:
      end program
```

- Because the session interactivity is essential, we tangle the file `checking.c`, compile and run it on the command line.

```
/* Balances a checkbook */
#include <stdio.h>

int main(void)
{
  int cmd; // user choice 0...4
  float balance = 0.0f, credit, debit;

  // User instructions
  printf("*** ACME checkbook-balancing program ***\n");
  printf("Commands: 0=clear, 1=credit, 2=debit, ");
  printf("3=balance, 4=exit\n\n");

  for(;;) {  // do this forever until exit=4
    printf("Enter command: ");
    scanf("%d", &cmd);
    switch (cmd) {
    case 0:            // clear balance
      balance = 0.0f;
      break;
```

```
    case 1:             // credit amount
      printf("Enter amount of credit: ");
      scanf("%f", &credit);
      balance += credit;
      break;
    case 2:             // debit amount
      printf("Enter amount of debit: ");
      scanf("%f", &debit);
      balance -= debit;
      break;
    case 3:           // print balance
      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;
    }
  }
}
```

- Get the program: tinyurl.com/2p975xs4 - tangle, compile and run it.