Iteration / Loops

CSC100 / Introduction to programming in C/C++ - Summer 2022

Table of Contents

1. README

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

2. Loops

- A **loop** is a statement whose job is to repeatedly execute over some other statement (the **loop body**).
- 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 non-zero), the loop continues to execute.
- C provides three iteration statements: `while`, `do`, and `for`

3. The `while` statement

General form

- The `while` statement has the general form

  ```
  while ( /expression/ ) statement
  ```

- The `statement` is executed as long as the `expression` is true.

Simple example

- A simple example.

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

- Parentheses `(...)` are mandatory
- Braces `{ }` are used for multi-line statements
- `[ ]` What does the code in 1 do?

- We can 1 what happens.

```
int i = 1, n = 10;
while ( i < n ) {
  i = i * 2;
  printf("%d < %d ?\n", i, n);
 }
```

```
2 < 10 ?
4 < 10 ?
8 < 10 ?
16 < 10 ?
```

- `[ ]`

  What would the pseudocode look like?

```
While i is smaller than n
  double the value of i
end loop
```

- `[ ]`

  What would a BPMN model look like?

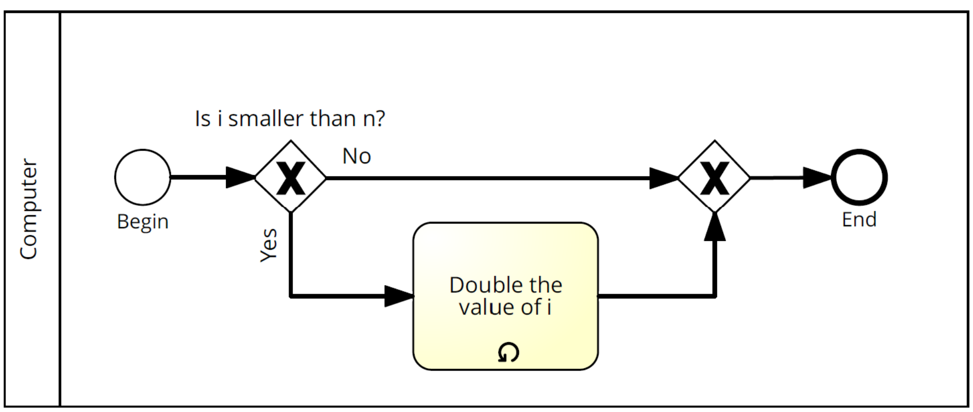

Figure 1: Simple while example

- The task (C statement) is overloaded with a **loop** attribute.

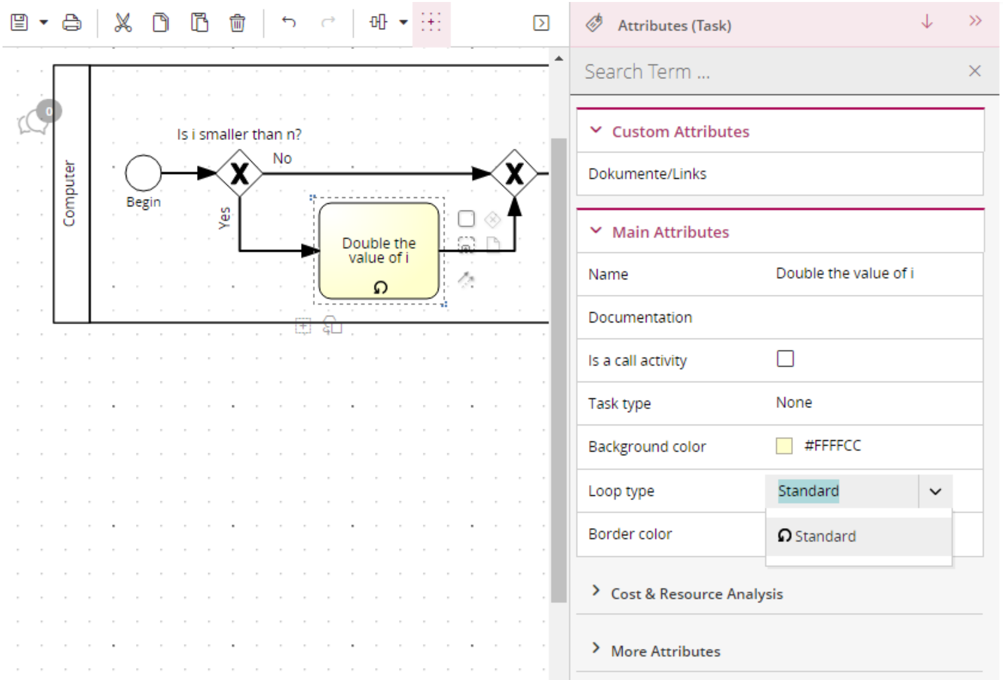

Figure 2: Simple while example

Countdown example

- [ ]

  What does the following statement do?

```
int i = 10;
while ( i > 0 ) {
  printf("T minus %d and counting\n", i);
  i--;  // same as i = i - 1; (executed from the right)
 }
printf("i = %d\n", 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
i = 0
```

- [X] Why are we using `i--` and not `--i` ?[1]
- [X] When would the `while` statements be bypassed completely?[2]
- [X]

1 could be made more concise - can you guess how?

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

- Note that in the concise version 1, it makes a difference if we use `i--` or `--i`. Try it!

Infinite loops

- If the controlling expression always has a non-zero value, the `while` statement will not terminate.

- The compiler does not check this. The program 1 has to be stopped manually (`C-g`).

```
//    while (1)
  //    puts("Still running...\n");
```

- `[X]` Tangle it, compile and run `inf.c` on the CMD line.
- `[X]` Why don't you see any output in Emacs? [3]
- To stop infinite loops from within, you need to provide `break`, `goto` or `return` statements.

Printing a table of squares

Problem

- Prompt the users to enter a number `n`
- Compute the squares of all integers from `1` to `n`.
- Print `n` and its square as a table of `n` rows

- Sample output:

```
Enter number of rows:
         1         1
         2         4
         3         9
         4        16
         5        25
         6        36
         7        49
         8        64
         9        81
        10       100
```

Solution

- Generate test input file:

```
echo 10 > ./src/square_input
cat ./src/square_input
```

```
int i, n;

printf("Enter number of rows: ");
scanf("%d", &n); printf("%d\n", n);

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

```
Enter number of rows: 10
         1         1
         2         4
         3         9
         4        16
         5        25
         6        36
         7        49
         8        64
         9        81
        10       100
```

Summing numbers

Problem

- Input a series of integers via the command line
- Compute the sum of the integers

- Sample output:

```
Enter integers (0 to terminate). 8 23 71 5 0
The sum is 107
```

Solution

- Scan numbers one after the other
- The program should exit when a `0` is scanned
- To sum, we can use the compound operator `+=`

- Pseudocode:

```
declare and initialize variables
scan first integer

while integer non-zero
  sum integer
  scan next integer

print the sum
```

- Generate test input file:

```
echo 8 23 71 5 0 > ./src/sum_input
cat ./src/sum_input
```

- Code:

```
int n, sum = 0;

printf("Enter integers (0 to terminate): ");
scanf("%d", &n);  printf("%d ", n);   // need non-zero number to start
while ( n != 0 ) {
  sum += n;         // sum = sum + n
  scanf("%d", &n); printf("%d ", n);
 }

printf("\nThe sum is %d\n", sum);
```

```
Enter integers (0 to terminate): 8 23 71 5 0
The sum is 107
```

  - There are two identical calls to `scanf`, because we need a non-zero number to enter the `while` loop in the first place.

4. The `do` statement

General form

- The `do` statement has the general form

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

- It's like a `while` statement whose controlling expression is tested *after* each execution of the loop body.
- 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 non-zero, the loop body is executed again and the expression is evaluated once more.
- Execution of the `do` statement terminates when the controlling *expression* has the value `0` (*FALSE*) **after** the loop body has been executed.
- Always use braces `{...}` around *all* `do` statements, because otherwise it can be mistaken for a `while` statement.

Calculating the number of digits in an integer

- `do` is handy for loops that must execute at least once.
- Let's write a program that calculates the number of digits in an integer entered by the user.

- Sample output:

```
Enter a nonnegative integer: 656
The number has 2 digits(s).
```

- Strategy: *digits* correspond to base 10 - if we divide the input by 10 repeatedly until it becomes 0 (via integer truncation), the number of divisions performed is the number of digits.

```
656 / 10 => 65 (remainder 6/10)
65  / 10 => 6  (remainder 5/10)
6   / 10 => 0  (remainder 6/10)
```

- Sample input: #+name in:dowhile

```
echo 656 > ./src/dowhile
cat ./src/dowhile
```

- Pseudocode:

```
do
  divide input n by 10
  add result to digits
while n is greater than 0
```

- Code:

```
int digits = 0; // number of digits
int n;  // input

printf("Enter a non-negative integer: ");
scanf("%d", &n); printf("%d\n", n);

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

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

```
Enter a non-negative integer: 2147483647
The number has 10 digit(s).
```

- `int` is actually a so-called *signed integer*, a 32-bit datum that encodes integers in the range `[-2147483647,2147483647]`. Any integer larger than this will not work - we have to use long integer types (and a different conversion specifier).

Counting down

Go to the practice workbook and rewrite 1 using a `do...while` statement.

Summing numbers

- Go to the practice workbook and rewrite the summing numbers program 1 using `do...while`.

5. The `for` statement

General form

- The `for` statement has the general form

  `for ( /expr1 ; expr2 ; expr3/ ) /statement/ ;`

- Here, `expr1`, `expr2` and `expr3` are expressions.

Simple example: countdown

- You recognize the familiar countdown program - except that the `for` loop includes initialization, condition and counting down all in one go.

```
int i;

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

Swapping `for` and `while`

- `for` loops can be replaced by `while` loops:

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

- Studying the equivalent `while` loop can yield important insights: you remember what happened when we swapped the postfix for a prefix operator in the `while` loop 1. Rewriting this program as a `for` loop, we get:

```
int i = 10;  /* expr1 */
while ( i > 0 /* expr2 */) {
  printf("T minus %d and counting\n", i-- /* expr3 */ );
 }
```

- `[X]`

  What should the argument of `for` look like?

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

- Since the first and third expressions in a `for` statement are executed as statements, their values are irrelevant.

`for` statement patterns

- `for` loops are best when counting up or down

PATTERN / IDIOM	CODE
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-- )`

  - Counting up loops rely on < and <=, while counting down loops rely on > and >= operators.
  - Note that the controlling expression does **not** use == but = instead - we're not looking for Boolean/truth values but for beginning numerical values.
  - The following is cool (but also dangerous): you can initialize the counting variable inside the first expression:

    ```
    // int i;
    for ( int i = 3 ; i > 0 ; i--)
      printf("T minus %d and counting\n", i);
    ```

    ```
    T minus 3 and counting
    T minus 2 and counting
    ```

```
T minus 1 and counting
```

Omitting expressions

- Some `for` loops may not need all 3 expressions, though the separators `;` must all three be present
- If the **first** expression is omitted, no initialization is performed before the loop is executed:

```
int i = 3;

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

```
T minus 3 and counting
T minus 2 and counting
T minus 1 and counting
```

- If the **third** expression is omitted, the loop body is responsible for ensuring that the value of the 2nd expression eventually becomes false so that the loop ends:

```
for (int i = 3 ; i > 0 ; )
  printf("T minus %d and counting\n", i--);
```

```
T minus 3 and counting
T minus 2 and counting
T minus 1 and counting
```

- If the **first** and **third** expressions are omitted, the resulting loop is nothing but a `while` statement in disguise:

```
int i = 3;

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

```
T minus 3 and counting
T minus 2 and counting
T minus 1 and counting
```

- The `while` version is clearer and to be preferred:

```
int i = 10;

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

- If the **second** expression is missing, it defaults to a true value so that the `for` loop will cause an infinite loop:

```
int i;

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

Printing a table of squares

- The program 1 can be improved by converting its `while` loop to a `for` loop:

```
int i, n;

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

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

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

- Inputfile

```
echo "5" > ./src/square1_input
cat ./src/square1_input
```

- In 1, all three expressions are controlled by the variable `i` for initialization, testing, and updating. However, **there is no requirement that they be related in any way**: the version 1 of the same program demonstrates this:

```
int i; // testing variable
int n; // upper bound constant
int odd; // incrementing variable
int square; // initialization variable

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

```
i   = 1;
odd = 3;
puts("         i    square       odd");
puts("------------------------------");

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

```
This program prints a table of squares.
Enter number of entries in table: 5
         i    square       odd
------------------------------
         1         1         3
         2         4         5
         3         9         7
         4        16         9
         5        25        11
```

- The `for` statement in 1 initializes one variable (`square`), tests another (`i`), and increments a third (`odd`).

  `i` is the number to be squared, `square` is the square of `i`, and `odd` is the odd number that must be added to the current square to get the next square (without having to multiply anything).

6. Exiting from a loop

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`.

The `break` statement

Overview

- 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 > ./src/break_input
cat ./src/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);
```

```
loop: n is 10
loop: n is 9
loop: n is 8
loop: n is 7
loop: n is 6
loop: n is 5
loop: n is 4
loop: n is 3
loop: n is 2
loop: n is 1
n is 0
```

- 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
```

  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

Overview

- 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 > ./src/continue
cat ./src/continue
```

Pseudo code:

```
while n smaller than 10
  get input i           // scanf
  if input is 0 go back // 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);
```

```
sum is 10
```

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 primt 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).

7. 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:

```
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;
    }
  }
}
```

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

Enter command:
```

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

8. References

- Davenport/Vine (2015) C Programming for the Absolute Beginner (3ed). Cengage Learning.
- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton.
- Orgmode.org (n.d.). 16 Working with Source Code [website]. URL: orgmode.org

Footnotes:

[1] `i--` is evaluated from the left, while `--i` is evaluated from the right. Both stand for `i = i - 1`, but `i--` assigns the current value of `i` and then subtracts `1`, while `--i` subtracts `1` and then assigns the result to `i`. In this case, the result is the same because we don't have any more statements that use `i` but if there were, it would make a difference.

[2] The loop body will not be entered if the expression tests out as false, i.e. if `i` is zero or negative.

[3] Because the program never reaches the end, it never gets to `return 0;`

Author: Marcus Birkenkrahe
Created: 2022-06-16 Thu 11:39