Iteration / Loops

CSC100 / Introduction to programming in C/C++

README

- This script introduces C looping structures.
- Practice workbooks and input files in GDrive (`loops`).
- A PDF version of this file and of the completed practice workbooks is available in GitHub in `pdf/`.
- This section is based on chapter 4 in Davenport/Vine (2015) and chapter 6 in King (2008).

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`

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
- `[X]` What does the code in 1 do?

- We can 1 what happens.

  ```c
  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 ?
```

- [X]

  What would the pseudocode look like?

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

- [X]

  What would a BPMN model look like?

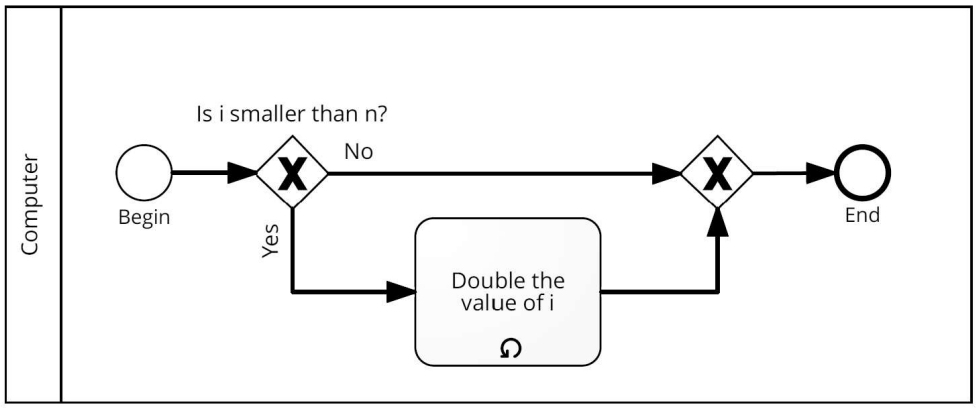

Figure 1: Simple while example

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

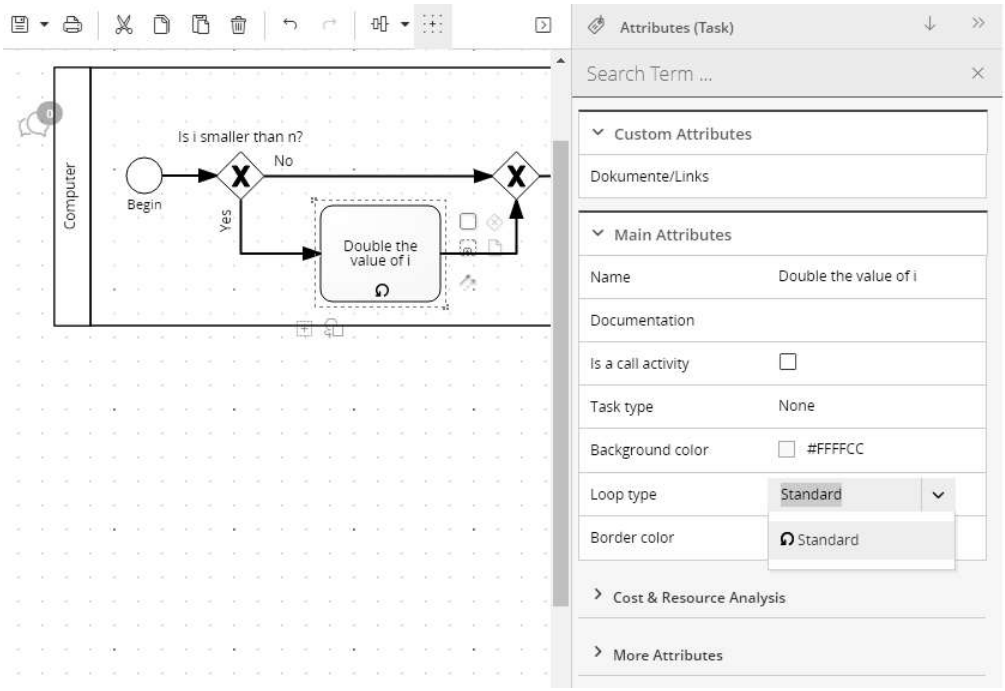

Figure 2: Simple while example

Countdown example

- [X]

  What does the following statement do?

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

- Generate test input file:

```
echo 10 > data/square_input
```

Solution

- [ ]

  Your turn! The input file `square_input` is already there.

```
int i, n;

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

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

```
Enter number of rows:
         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 CMD line
- Compute the sum of the integers
- Sample output:

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

- Generate test input file

```
echo 8 23 71 5 0 > data/sum_input
```

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
```

- Code:

```
int n, sum = 0;

printf("Enter integers (0 to terminate).\n");
scanf("%d", &n);
while ( n != 0 ) {
  sum += n;          // sum = sum + n
  scanf("%d", &n);
 }

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

```
Enter integers (0 to terminate).
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.

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.

Countdown example

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

- Here is the pseudocode:

```
do {
 print i
 decrement i by 1
 } while i is greater than 0
```

- Solution:

  This is the concise version with the decrement operator inside the function call.

```
int i = 10;

do {
  printf("T minus %d and counting\n", i--);
 } while (i > 0);
```

```
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 main difference to the `while` statement is that the loop body is executed at least once.
- Always use braces `{...}` around *all* `do` statements, because otherwise it can be mistaken for a `while` statement.

Summing numbers

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

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

printf("Enter integers (0 to terminate).\n");

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

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

```
Enter integers (0 to terminate).
The sum is 107
```

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
```

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

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

- Studying the equivalent `while` loop can yield important insights. For example

`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.

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 = 10;

for ( ; 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
```

- 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:

```
int i;

for ( i = 10 ; 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 **first** and **third** expressions are omitted, the resulting loop is nothing but a `while` statement in disguise:

```
int i = 10;

for ( ; 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
```

- 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);
```

TODO The comma operator

TODO Exiting from a loop

TODO The Null statement

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`.

[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: 2022-04-06 Wed 18:55