Computer Programming: Skills & Concepts (CP)

Loops

Cristina Alexandru

Tuesday 3 October 2017
Summary of Lecture 5

- if statements
- boolean conditions
- nested if
- refinements of quadratic.c
This Lecture

- Precedence of operators.
- The while statement.
- The for statement.
- fibonacci.c
A note about operator precedence

In everyday mathematics, when we write $4 + 5 \times 3$, we expect it to mean $4 + (5 \times 3)$, not $(4 + 5) \times 3$.

C does the same: every operator has a precedence, and brackets are automatically understood around higher precedence expressions: $\times$ has higher precedence than $+$, so $4 + 5 \times 3$ means what you think.

Higher precedence means “gets done first”.

We suggest that you only rely on the following:

- $\times$, $/$ and $\%$ have higher precedence than $+$ and $-$
- arithmetic operators have higher precedence than relational operators

and everywhere else, use brackets to make clear what you mean.
while

We have already seen our primary *programming construct* for branching (doing different things based on the result of a test). This is the if...else statement.

In programming, we also need need to repeat some action many times until we’ve reached a suitable stopping point. The while-statement allows us to specify this behaviour.

```java
while (condition) {
    statement-sequence
}
```

while means “repeat until failure” (of condition).

*statement-sequence* will usually alter some variables involved in *condition*. Why?
Early computers were used for printing mathematical tables. Consider printing a table of squares from 1 to 20:

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int n=1;
    while (n <= 20) {
        printf("The square of %4d is %4d.\n", n, n*n);
        n = n+1;
    }
    return EXIT_SUCCESS;
}
```

The `%4d` in the `printf` means ‘print as an integer and pad on the left with spaces to fill up 4 columns’. We’ll see other fancy stuff with `printf` later.
Fibonacci Numbers

0    1

0 + 1 = 1

1 + 1 = 2

1 + 2 = 3

2 + 3 = 5

3 + 5 = 8

5 + 8 = 13

8 + 13 = 21
Solving Fibonacci with **while**

- We need to keep adding the two previous Fibonacci numbers `while` we are \( \leq \) than \( n \).
- We will need a variable (call it `count`) to keep track of our `current Fibonacci`.
- Our **condition** for the **while**-statement will compare `count` with \( n \). Need to stop after we have reached the Fibonacci number for \( n \).
- The starting values are 0 (0th Fibonacci number) and 1 (1st Fibonacci number).
int main(void) {
    int n, next, count;
    int previous = 0;        /* Fibonacci 0 */
    int current = 1;         /* Fibonacci 1 */

    /* before here, n has been set to the bound */
    count = 2;
    while (count <= n) {
        next = previous + current;  // eg. 2nd fib is = 0 + 1
        previous = current;
        current = next;        // current is reset:
        count++;
    }
    printf("Fibonacci %d is %d\n", n, current);
    return EXIT_SUCCESS;
}
running fibonacci.c

:  ./a.out
Calculate which Fibonacci number?  1
Fibonacci 1 is 1

:  ./a.out
Calculate which Fibonacci number?  2
Fibonacci 2 is 1

:  ./a.out
Calculate which Fibonacci number?  7
Fibonacci 7 is 13
while-statement: Repeat n-times

\[\text{initialise-iterator} \]
\[\text{while ( not-iterator-endpoint ) } \{ \]
\[\quad \text{work-on-this-value} \]
\[\quad \text{next-iterator-value} \]
\[\}

It is very common to use while to perform some statements depending on \(i\) for all values of \(i\) up to some integer limit (as we did for fibonacci.c).
while-statement

Counting-up:

    count = 0;
    while (count < n) {
        statement-sequence;
        count++;
    }

Counting-down:

    count = n;
    while (count > 0 ) {
        statement-sequence;
        count--;
    }

Careful about ‘fencepost errors’: counting up by initializing iterator to 0 and looping while $< n$ does loop $n$ times with values $0, 1, \ldots, n-1$. 
The for-loop

Counting up with a for-loop:

    for (count = 0; count < n; count++) {
        statement-sequence
    }

The general form is:

    for (init-expression ; condition ; update-expression ) {
        statement-sequence
    }

which is the same as (apart from one small detail)

    init-expression ;
    while ( condition ) {
        statement-sequence
        update-expression ;
    }

We’ve told the same little lie about general forms as we told with the if-statement.
Fibonacci using for

```c
int n, next, count;
...
// set n to the required Fibonacci number
int previous = 0;       /* Fibonacci 0 */
int current = 1;        /* Fibonacci 1 */
for (count = 2; count <= n; count++) {
    next = previous + current;
    previous = current;
    current = next;
    // current now the count-th Fibonacci
}
// on leaving loop current is now n-th Fibonacci
```

What is the value of `count` after finishing the loop?
Prime Numbers

**Definition:** A prime number is any natural number greater than 1 which has no factors except itself and 1.

Prime: 3, 7, 11

Not Prime: 9 \((3 \cdot 3)\), 10 \((2 \cdot 5)\)

Simple test for primes:

\[ n \text{ is prime if } n > 1 \text{ and there is no integer } k \text{ between 2 and } \sqrt{n} \text{ such that } n \% k = 0. \]

The **while** and **for** statements are good candidates for writing a prime-testing program `prime.c`
Reading

For *precedence of operators*, read Section 2.9 of “A Book on C”.

Sections 4.8 (*while*) and 4.9 (*for*) of “A Book on C”.

There will be some loop-based programing exercises in labsheet 3.