Summary of Lecture 5

- if statements
- boolean conditions
- nested if
- refinements of quadratic.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: $*$ has higher precedence than $+$, so $4 + 5 * 3$ means what you think.

Higher precedence means “gets done first”.

We suggest that you only rely on the following:

- $*$, $/$ 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 to repeat some action many times until we’ve reached a suitable stopping point. The while-statement allows us to specify this behaviour.

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

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

statement-sequence will usually alter some variables involved in condition.

Why?

---

Printing a table

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;
}
```

---

Fibonacci Numbers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 + 1 = 1</td>
<td></td>
</tr>
<tr>
<td>1 + 1 = 2</td>
<td></td>
</tr>
<tr>
<td>1 + 2 = 3</td>
<td></td>
</tr>
<tr>
<td>2 + 3 = 5</td>
<td></td>
</tr>
<tr>
<td>3 + 5 = 8</td>
<td></td>
</tr>
<tr>
<td>5 + 8 = 13</td>
<td></td>
</tr>
<tr>
<td>8 + 13 = 21</td>
<td></td>
</tr>
</tbody>
</table>

---

Solving Fibonacci with while

- We need to keep adding the two previous Fibonacci numbers ‘while’ we are ≤ 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)
```c
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, current);
    return EXIT_SUCCESS;
}
```

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:**

```c
count = 0;
while (count < n) {
    next = previous + current; // eg. 2nd fib is = 0 + 1
    previous = current;
    current = next; // current is reset:
    count++;
}
```

**Counting-down:**

```c
count = n;
while (count > 0) {
    next = previous + current; // current is reset:
    count--;
}
```

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

Counting up with a for-loop:

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

The general form is:

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

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

```c
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.

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`

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?

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 programming exercises in labsheet 3.