Computing Fundamentals
Flow Control

Salvatore Filippone

salvatore.filippone@uniroma2.it

2014–2015

Flow control

It is often the case that the program we are writing may execute different instructions given different inputs:

- The code may have to take different paths according to the data;
- The code may have to repeat multiple times some instructions.

Therefore we have the two categories:

1. Conditional execution statements;
2. Iteration statements.

Conditional execution

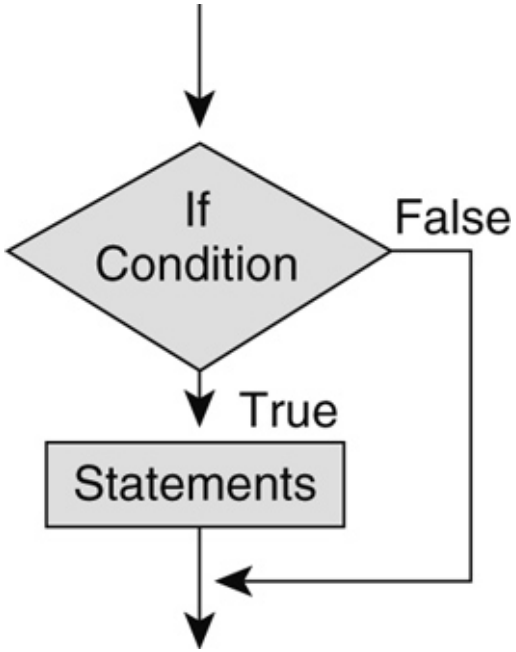
Conditional execution:

> *Executing different instructions according to different input conditions*

In practice this takes two forms:

1. `IF` statements
2. `SWITCH` statement

Conditional execution



Main points:

- The control is through a boolean-valued expression;
- The statements can be nested to an arbitrary depth;

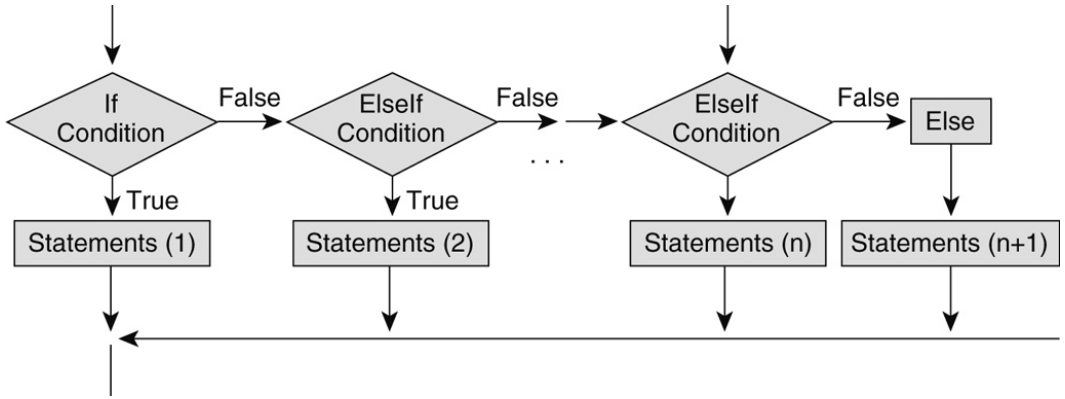
Conditional execution

A template:

```
if < expression 1 >
   <code block >
else
    if <expression 2>
       <code block 2>
    else
       <code block 3 >
    end
end
```

Conditional execution

If the various statements are all at the same "level":




```
if < expression 1 >
   <code block >
elseif <expression 2>
   <code block 2>
else
   <code block 3 >
end
```

Conditional execution

`IF` statements can be controlled by:

- A boolean constant `true` `false`
- A variable containing a boolean value;
- A generic Boolean expression: see boolean operators `~` `&&` `||`
- A boolean function on arrays: `all` and `any`

Note the *short-circuit* semantics.

Conditional execution

SWITCH statements are useful when control is through the value of a scalar variable:

```
switch number
case{0,1,2}
   disp("Number between 0 and 2")
case{3}
   disp("The number 3")
otherwise
   disp("Something else")
end
```

Very convenient, but:

- Beware of floating-point cases;
- The first matching case is executed;
- Good practice: use `otherwise`

SWITCH can have string labels, which are *not* possibile with IF.

Conditional execution

Example:

Find the roots of a quadratic polynomial

Iterations

General concept:

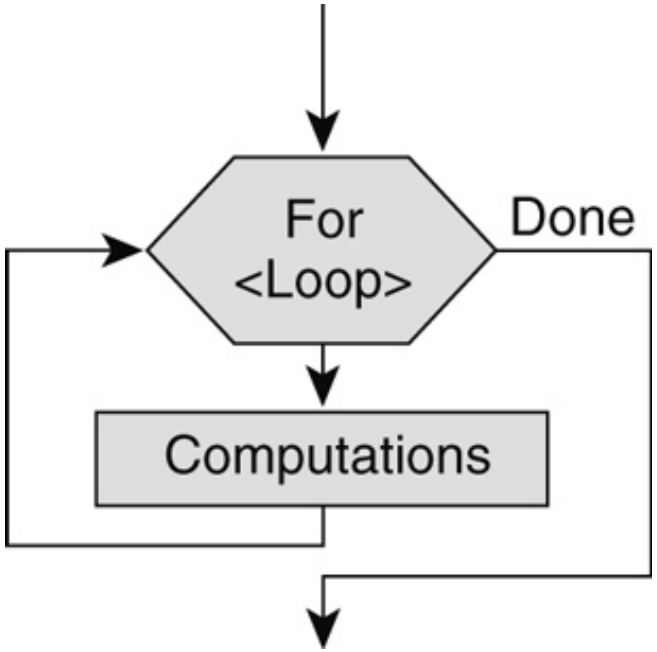
A block of code has to be executed multiple times

Two variants:

1. `for` loop;
2. `while` loop.

Iterations: `for`

Restricted applicability but very powerful: used when number of iterations can be known in advance:



Iterations: `for`

Template:

```
for <variable> = <vector>
    <code block>
end
```

Note: the number of iterations is equal to the number of entires of `<vector>` at the time the loop is entered.

Iterations: `for`

Example:

```
sm=0;
for i = 1:length(v)
    sm = sm + v(i);
end
disp(sm)
```

Iterations: `for`

The previous example could have been written as:

```
sm=0;
for x = v
    sm = sm + x;
end
disp(sm)
```

because `x` takes at each iteration the value of the corresponding entry in `v`.

Iterations: continue and break

`continue`: skip the rest of the loop body, but stay within the loop

```
for <variable> = <vector>
    <code block>
    if (<condition>) continue
    <block 2>
end
```

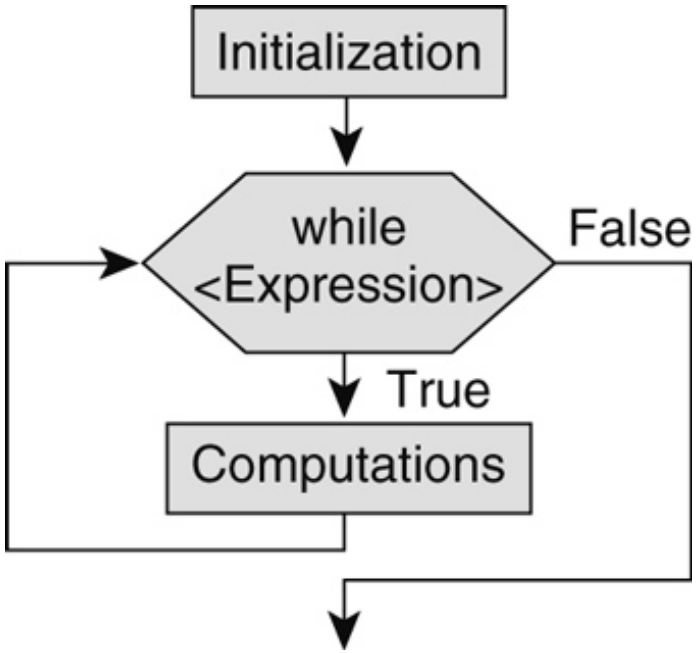
```
for <variable> = <vector>
    <code block>
    if (~<condition>)
        <block 2>
    end
end
```

`break`: exit from the loop

```
for <variable> = <vector>
    <code block>
    if (<codition>) break
    <block 2>
end
<block 3>
```

Iterations: `while`

Most general loop construct available: any kind of cycle can be recast into a `while` loop



Iterations: `while`

Template:

```
<initialization>
while (<condition>)
    <code block>
    <update condition>
end
```

Note: the number of iterations is not known a priori.

Iterations: `while`

Most common errors in using the `while` loop:

1. Wrong loop condition;
2. Variable in loop condition not initialized before loop;
3. Variable in loop condition not updated in loop body.

Example:

Find the GCD of two integers with the Euclid algorithm