C++ Programming for Engineers

Petar Jervic
Francisco Luro
Andrew Moss (course-resp)

May 2, 2012

Overview

So far we have seen programs that are a series of steps (statements).

- Input
- Output
- Assignment
  - changing values in variables by evaluating expressions

But we need decisions to write more complex programs

- Ability to change the steps that we take according to data
- Allows different responses to different input
- More complex behaviour in a program
- Multiple possible sequences of actions...

If a condition is true then do something...

Example

Something that seems simple mathematically: $x = \frac{y}{z}$

- What happens when $z = 0$?
- The equation has an undefined value

Consider the same problem in C++

```
int x,y,z;
cin >> y >> z;
z = y/z;
cout << z;
```

Assume the user inputs 0 as the second value

- How can the computer indicate the problem?

Example (cont)

Several ways for the computer to indicate the error

- A magic value (predefined special constant)
- Set a flag that something has gone wrong
- Abort the program

In this case the program aborts (crashes)

- There is no possible magic value (all valid)
- We haven't used the mechanism for flags
- Simplest solution is to abort
- Programmer is responsible for preventing this...

Example (repairing)

A new kind of statement

- Tests a condition, chooses what to do

```
int x,y,z;
cin >> y >> z;
if(z!=0)
   z = y/z;
else
   cout << "Something bad happened";
cout << z;
```

The program no longer crashes

- But it doesn't really work properly either...

Choices

Programs make many different kinds of choices

- Was the input entered valid?
  - Did it contain the right kinds of characters?
  - Did it contain the right number of characters?
- What kind of symbol was received from the network?
  - Do we start a new connection?
  - Should we send a webpage?
- Did the phone rotate?
  - Should we redraw the screen?
- Did the player strike the ball?
  - Did the camera detect motion?
  - Does it look like the player?
  - Can we estimate their position?

Boolean choices

All of the previous examples were phrased in a specific way

- Did X happen (or not)?
- This is a binary choice
- Compare to What happened?
  - Much harder to express the result...

Binary values are the basis of Boolean logic

- Instead of 1 and 0 we use `true` and `false`
- We have operators that we can apply to boolean values
  - AND, e.g. `x && y`
  - OR, e.g. `x || y`
  - NOT, e.g. `!x`
- Notice the difference to the binary operators (`&`, `|`, `!`)

Choices about numbers

The boolean operators only operate on true/false values.

- Numbers need something else...
- A way to convert choices about numbers into boolean values
- Comparisons...

Is $x < 4$?

- This statement is either true or false
- We call this a *comparator* operator
- It compares numbers, and converts the *result* into a `bool` value

Numerical comparitors

Name	C++	Maths
Less than	`x<y`	$x < y$
Greater than	`x>y`	$x > y$
Less than or equal	`x<=y`	$x \leq y$
Greater than or equal	`x>=y`	$x \geq y$
Equal	`x==y`	$x = y$
Not equal	`x!=y`	$x \neq y$

- Once value have been compared we have truth values
- Boolean operators work on truth values
- So we can combine things...
  - `3<x && x<10`

Syntax

The condition is written inside brackets

```
if(x<7) cout << "It was small";
```

- The condition must evaluate to a boolean value
- Can use comparitors and boolean operators
- Previous weeks have focused on numerical / binary expressions
- This is a boolean expression
- A normal statement follows after the `if(condition)`
  - This can be any kind of statement you have seen so far
- Sometimes we may want to store boolean values as *flags*

```
bool c = x<7;
if(c) cout << "It was small";
```

More Syntax

The statement written immediately after the `if` condition...

- is said to be *inside* the if-statement
- sometimes we say it is *controlled* by the if-statement
- can be any kind of statement

So we can do this

```
if(x<7) if(y<10) cout << "Both quite small";
```

Consider how this would relate to

```
if(x<7 && y<10) cout << "Both quite small";
```

Break

Psuedo code

So far we have had specific examples

- But we want to discuss how you can write these statements in general
- So we use *psuedo* code
- This is NOT code — it will not compile
- It is way for one human to explain to another human what structure in a program looks like

The common factor in each example so far is this

```
if(condition) statement ;
```

```
if(condition)
  statement ;
```

Whitespace doesn't matter (the compiler uses `;`)

- Second form is more common, considered easier to read

Using scopes as blocks of code

Sometime we want to do more than one thing if a condition is true

```
if(x>10) cout << "Value was too large" ;
if(x>10) cin >> x;
```

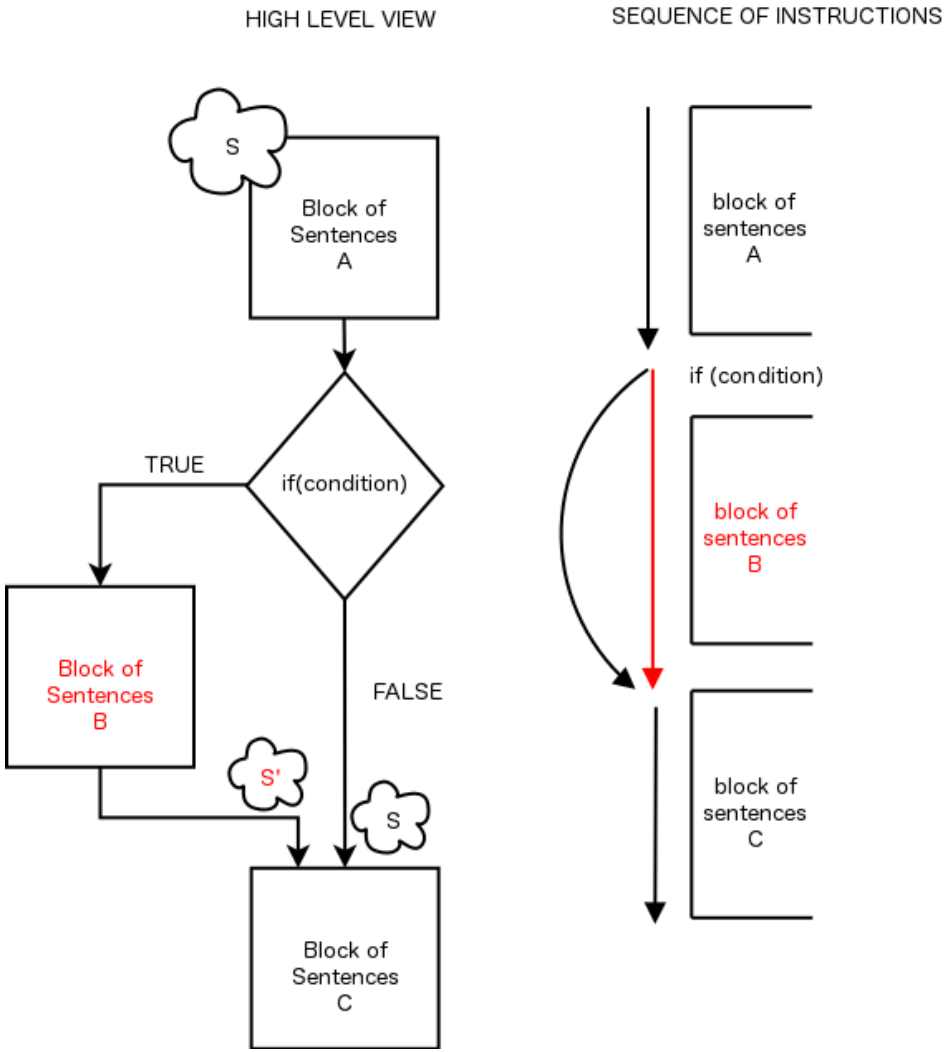
We can test the same condition multiple times

- But there is an easier (and more efficient) way

```
if(x>10) {
  cout << "Value was too large" ;
  cin >> x;
}
```

How if statements work

- Compiler writes assembly instructions as the program binary
- Instructions are a single sequence
- One instruction is a conditional jump
- Allows some instructions to be skipped
- Two possible sequences (paths) through the instructions



Mutually exclusive conditions

Consider the two conditions $x < 10$ and $x \geq 10$

- Cannot both be true for a single value of x
- One of them is always for any value of x

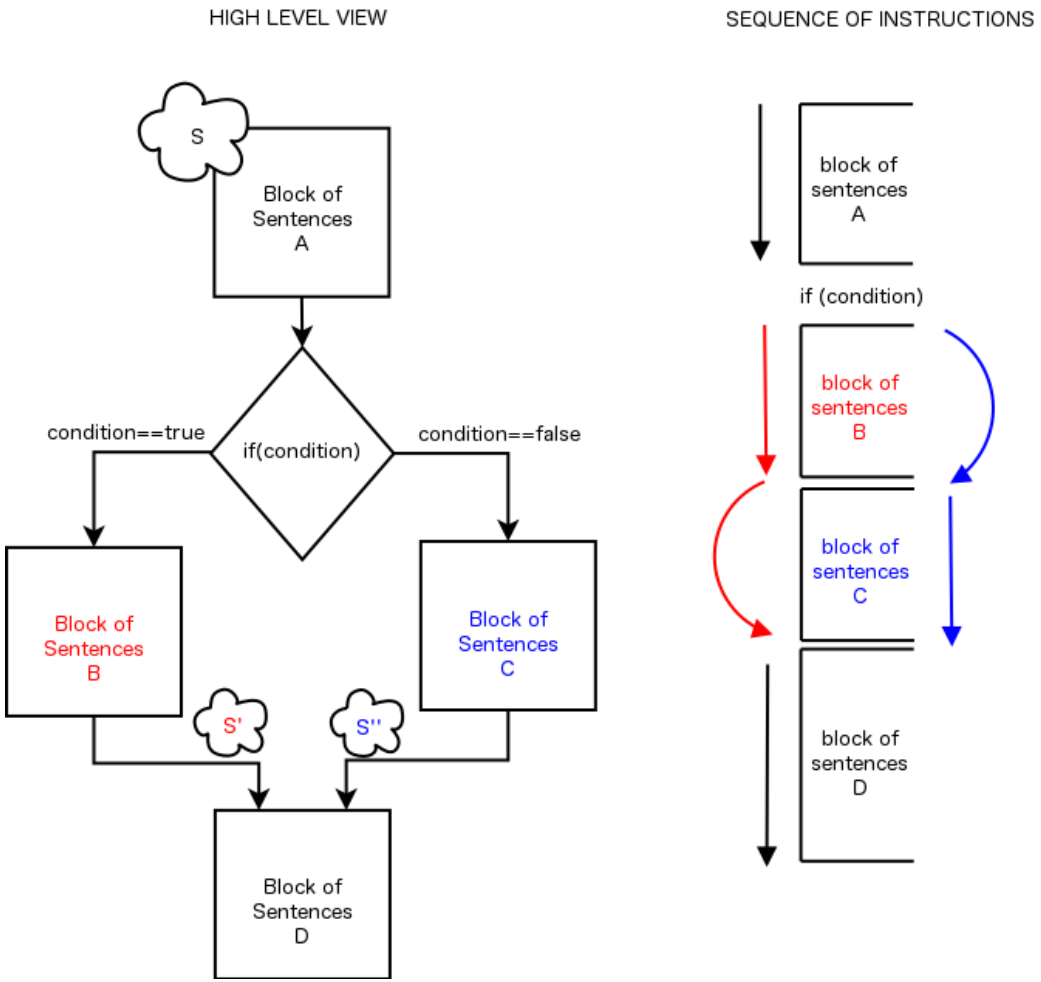
```
if(x<10)
  cout << "Small case" ;
if(x>=10)
  cout << "Large case" ;
```

It would be simpler to say the second case is when the first case is not true

```
if(x<10)
  cout << "Small case" ;
else
  cout << "Large case" ;
```

How the if-else is compiled

```
...
if(expression) {
     statements;
}
else {
     statements;
}
...
```



Nesting

We have seen that if-statements are binary choices

- We may need to make decisions with more cases
- Consider converting a month into a season
- Twelve possible values, four possible results

```
if(month<6) {
  if(month<3)
    season = "Winter";
  else
    season = "Spring";
}
else {
  if(month<9)
    season = "Summer";
  else
    season = "Autumn";
}
```

Unbalanced nests

The code in the two parts is unrelated

- Do not need the same structure in both parts
- Can build arbitrary combinations

```
if(temperature>0) {
  if(temperature>50)
    cout << "Hot";
  else
    cout << "Warm";
}
else {
  cout << "Cold";
}
```

Common pitfalls / source of bugs

The symbol = is an assignment

- The symbol == is a comparison
- `if (a=0) ...`

Indenting code with spaces is for your benefit

- The compiler ignores it
- Meaning of the code comes from the brackets `{ ...}` and semicolons `;`

```
if (a != 0)
  cout << "Not zero";
  cout << "Still not zero";
```