Lecture 3: Flow of Control

Curtin FIRST Robotics Club (FRC) Pre-season Training

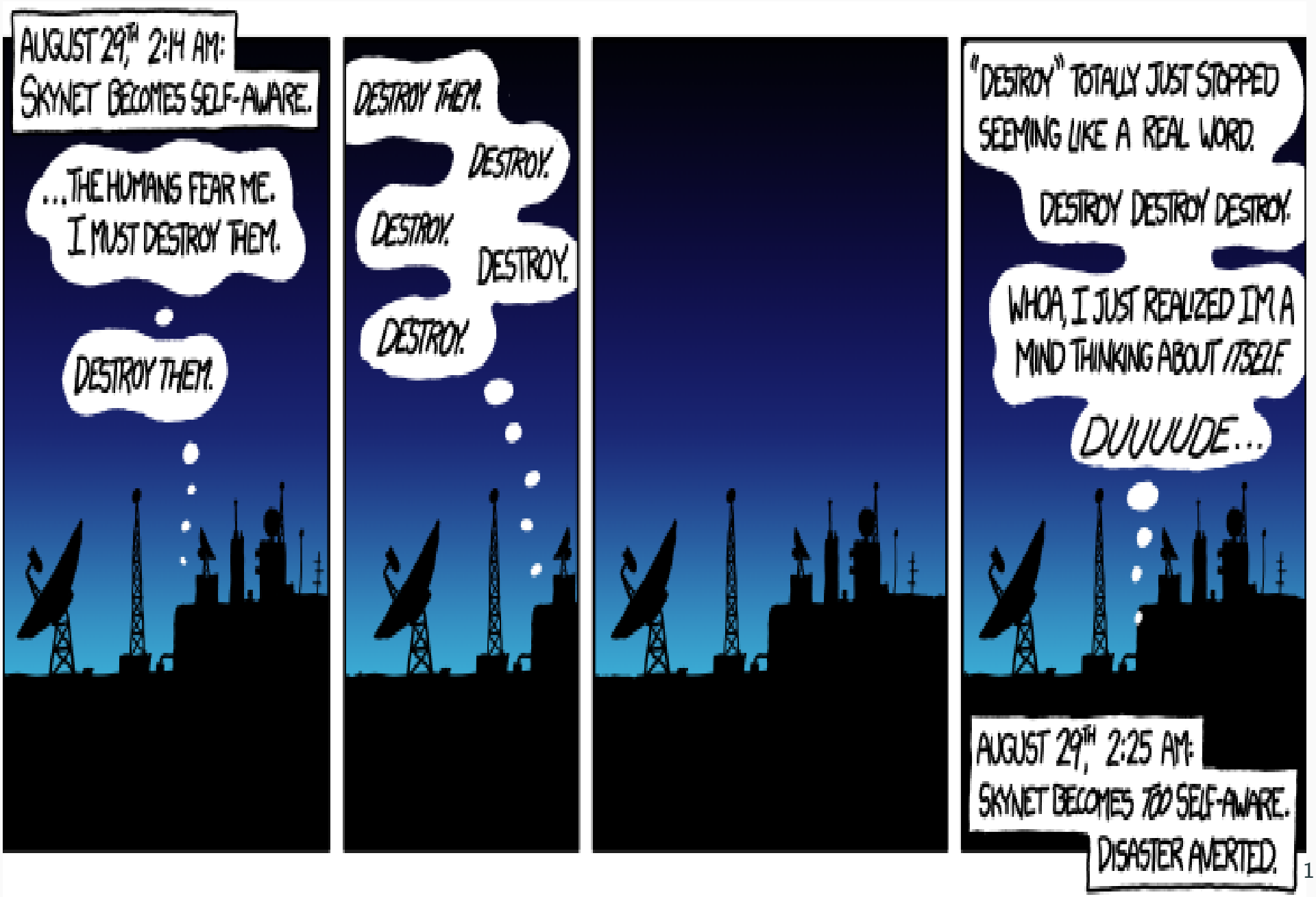
Scott Day

265815F@curtin.edu.au

October 29, 2016

Curtin University

Insert Mandatory Programming Joke



Control Structures

`Control structures` are portions of program code that contain statements within them and, depending on the circumstances, execute these statements in a certain way. There are typically two kinds: `conditionals` and `loops`.

Table of contents

Conditionals

Conditionals

In order for a program to change its behavior depending on the input, there must be a way to test that input. Conditionals allow the program to check the values of variables and to execute (or not execute) certain statements.

C++ has `if` and `switch-case` conditional structures.

Operators

Conditionals use two kinds of special operators: `relational` and `logical`.
These are used to determine whether some condtion is true or false.

Relational Operators

Relational operators are used to test a relation between two expressions:

Operator	Shorthand	Meaning
`>`		Greater than
`>=`	$\geq$	Greater than or equal to
`<`		Less than
`<=`	$\leq$	Less than or equal to
`==`		Equal to
`!=`	$\neq$	Not equal to

They work the same as the arithmetic operators (e.g. `a > b`) but return a Boolean value of either `true` or `false`, indicating whether the relation tested for holds (Boolean expression).

For example, if the variables `x` and `y` have been set to 6 and 2, respectively, then `x > y` returns `true`. Similarly, `x < 5` returns `false`.

Logical Operators

The logical operators are often used to combine relational expressions into more complicated Boolean expressions.

Operator	Meaning
&&	and
||	or
!	not

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

```
!(x > 2)
```

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

`!(x > 2)` false

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

```
!(x > 2)                    false
(x > y) && (y > 0)
```

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

`!(x > 2)`	false
`(x > y) && (y > 0)`	true

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

`!(x > 2)`	false
`(x > y) && (y > 0)`	true
`(x < y) && (y > 0)`	

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

`!(x > 2)`	false
`(x > y) && (y > 0)`	true
`(x < y) && (y > 0)`	false

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

```
    !(x > 2)          false
(x > y) && (y > 0)     true
(x < y) && (y > 0)    false
(x < y) || (y > 0)
```

Truth Tables

Operators return `true` or `false`, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

a	!a
true	false
false	true

Example questions using logical operators (assume `x = 6` and `y = 2`):

`!(x > 2)`	false
`(x > y) && (y > 0)`	true
`(x < y) && (y > 0)`	false
`(x < y) || (y > 0)`	true

C++ Boolean

Boolean variables can be used directly in these expressions, since they hold `true` and `false` values.

Funny enough, any kind of value can be used in a Boolean expression due to a quirk C++ has:

> `false` is represented by a value of 0 and anything that is not 0 is `true`.

So, "Hello, world!" is `true`, 2 is `true`, and any int variable holding a non-zero value is `true`. This means `!x` returns `false` and `x && y` returns `true`!

If

The `if` condition has the form:

```
1  if(condition)
2  {
3      statement1
4      statement2
5      ...
6  }
```

The condition is some expression whose value is being tested. If the condition resolves to a value of `true`, then the statements are executed before the program continues on. Otherwise, the statements are ignored.

If there is only one statement, the curly braces may be omitted, giving the form:

```
1  if(condition)
2      statement
```

If-Else

The `if-else` form is used to decide between two sequences of statements referred to as `blocks`:

```
1  if(condition)
2  {
3      statementA1
4      statementA2
5      ...
6  }
7  else
8  {
9      statementB1
10     statementB2
11     ...
12 }
```

If the condition is met, the block corresponding to the `if` is executed. Otherwise, the block corresponding to the `else` is executed.

If there is only one statement for any of the blocks, the curly bracers for that black may be omitted:

```
1  if(condition)
2      statementA1
3  else
4      statementB1
```

Else-If

The `else if` is used to decide between two or more blocks based on `multiple` conditions:

```
if(condition1)
{
    statement1
    statement2
    ...
}
else if(condition2)
{
    statementB1
    statementB2
    ...
}
```

If `condition1` is met, the block corresponding to the `if` is executed. If not, then only if `condition2` is met is the block corresponding to the `else if` executed.

If Example

Here is an example using these control structures:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int x = 6;
7      int y = 2;
8
9      if(x > y)
10         cout << "x is greater than y" << endl;
11     else if(y > x)
12         cout << "y is greater than x" << endl;
13     else
14         cout << "x and y are equal" << endl;
15
16     return 0;
17 }
```

The output of this program is `x is greater than y`. If we set the lines 6 and 7 to `int x = 2;` and `int y = 6;` respectively, then the output is `y is greater than x`.

If we replace the lines with `int x = 2` and `int y = 2;` then the output is `x and y are equal`.

Switch-Case

The `switch-case` is another conditional structure that may or may not execute certain statements.

```
1  switch(expression)
2  {
3      case constant1:
4              statementA1
5              ...
6              break;
7      case constant2:
8              statementB1
9              ...
10             break;
11     ...
12     default:
13             statementZ1
14             ...
15 }
```

The `switch` evaluates `expression` and, if `expression` is equal to `constant1`, then the statements beneath `case constant 1:` are executed until a `break` is encountered. If `expression` is not equal to `constant1`, then it is compared to `constant2`. If these are equal, then the statements beneath `case constant 2:` are executed until a break is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath `default:` are executed.

Switch-Case

Due to the peculiar behavior of switch-cases, curly braces are not necessary for cases where there is more than one statement (but they are necessary to enclose the entire switch-case).

Switch-cases generally have if-else equivalents but can often be a cleaner way of expressing the same behavior.

Switch-Case Example

Here is an example using `switch-case`:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int x = 6;
7
8      switch(x)
9      {
10         case 1:
11             cout << "x is 1" << endl;
12             break;
13         case 2:
14         case 3:
15             cout << " x is 2 or 3" << endl;
16             break;
17         default:
18             cout << "x is not 1, 2, or 3" << endl;
19     }
20
21     return 0;
22 }
```

This program will print `x is not 1, 2, or 3`. If we replace line 6 with

```
int x = 2;
```

Loops

Loops

Conditionals execute certain statements if certa`while` in conditions are met; loops execute certain statements while certain conditions are met. C++ has three kinds of loops: `while`, `do-while`, and `for`.

While

The `while` loop has a form similar to the if conditional:

```
while(condition)
{
    statement1
    statement2
    ...
}
```

As long as the condition holds, the block of statements will be repeatedly executed.

If there is only one statement, the curly braces may be omitted.

While Example

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int x = 0;
7
8      while(x < 10)
9          x = x + 1;
10
11     cout << "x is " << x << endl;
12
13     return 0;
14 }
```

This program will print `x is 10`.

Do-While

The `do-while` loop is a variation that guarantees the block of statements will be executed at least once:

```
1  do
2  {
3      statement1
4      statement2
5      ...
6  }
7  while(condition);
```

The block of statements is eecuted and then, if the condition holds, the program returns to the top of the block.

Curly braces are always required.

Also not note the semicolon after the `while` condition.

For

The `for` loop works like the while loop but with some change in syntax:

```
for(initialization; condition; incrementation)
{
    statement1
    statement2
    ...
}
```

The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop.

Curly braces may be omitted if there is only one statement.

For Example

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for(int x = 0; x < 10; x = x + 1) // or x++
7          cout << x << endl;
8
9      return 0;
10 }
```

This program will print out the values 0 through 9.

If the counter variable is already defined, there is no need to define a new one in the initialization porrtion of the for loop,

Nested Control Structures

Nested Control Structures

It is possible to place ifs inside of ifs and loops inside of loops by simply placing these structures inside the statement blocks.

This allows for more complicated program behaviour.

Nested If Conditionals

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int x = 6;
7      int y = 0;
8
9      if(x > y)
10     {
11         cout << "x is greater than y" << endl;
12
13         if(x == 6)
14             cout << "x is equal to 6" << endl;
15         else
16             cout << "x is not equal to 6" << endl;
17     }
18     else
19         cout << "x is not greater than y" << endl;
20
21     return 0;
22 }
```

This program will print `x is greater than y` on one line and then `x is equal to 6` on the next line.

Nested Loops

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for(int x = 0; x < 4; x = x + 1)
7      {
8          for(int y - 0; y < 4; y = y + 1)
9              cout << y;
10         cout << endl;
11     }
12
13     return 0;
14 }
```

This program will print four lines of 0123.

References I