Design and Computing 1

Introduction to Scientific Computing

Daniel Poole
Department of Aerospace Engineering
University of Bristol
`d.j.poole@bristol.ac.uk`

2017

LECTURE 3

Intro to Scientific Computing

- Have considered an intro to the history of scientific computing
- Have used programs to perform simple calculations

TODAY

- What about if we want to perform a calculation that depends on some event
- What about if we want to perform a calculation multiple times
- Using branching, conditionals and looping $\Rightarrow$ Control flow
- A more involved example that involves taking a problem and converting to a program

Control Flow

So far we have only considered the most basic of programs, where the code is executed linearly (i.e. every line after the one previously) and all lines are executed. However, almost all programs will not operate in this manner, and instead have an aspect of control flow to them.

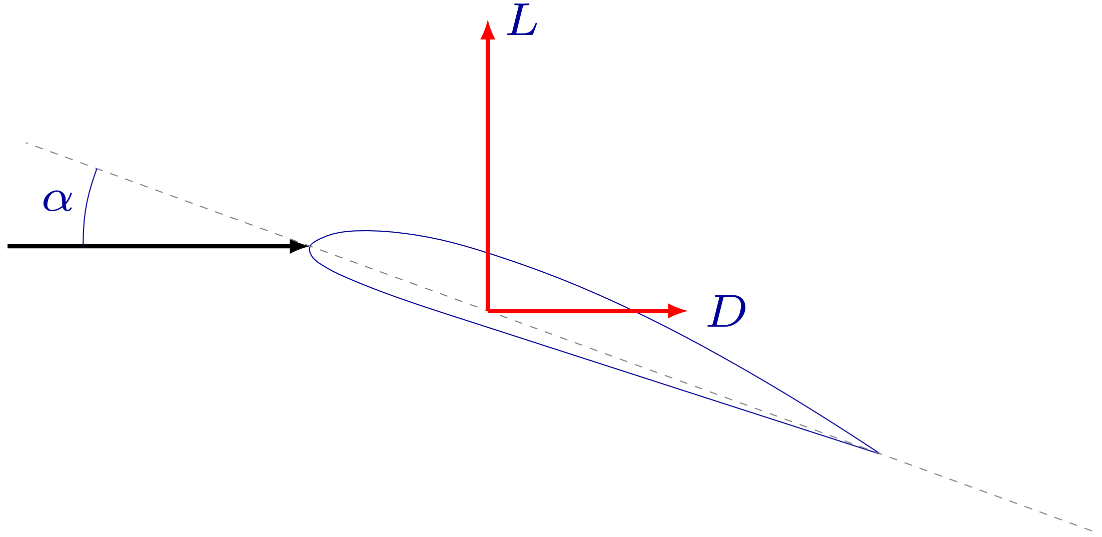
Control flow primarily includes branching and looping. Branching is when boolean tests are set and depending on the outcome, certain operations are executed over others. Looping is really where computers come into their own, as these sorts of statements execute a given block of code multiple times. Looping is arguably the most useful aspect of coding that you will learn to use as this type of logic forms the basis of many methods that solve the types of problems you will experience during the course and your careers.

We will firstly consider branching, followed by looping.

Branching

Branching

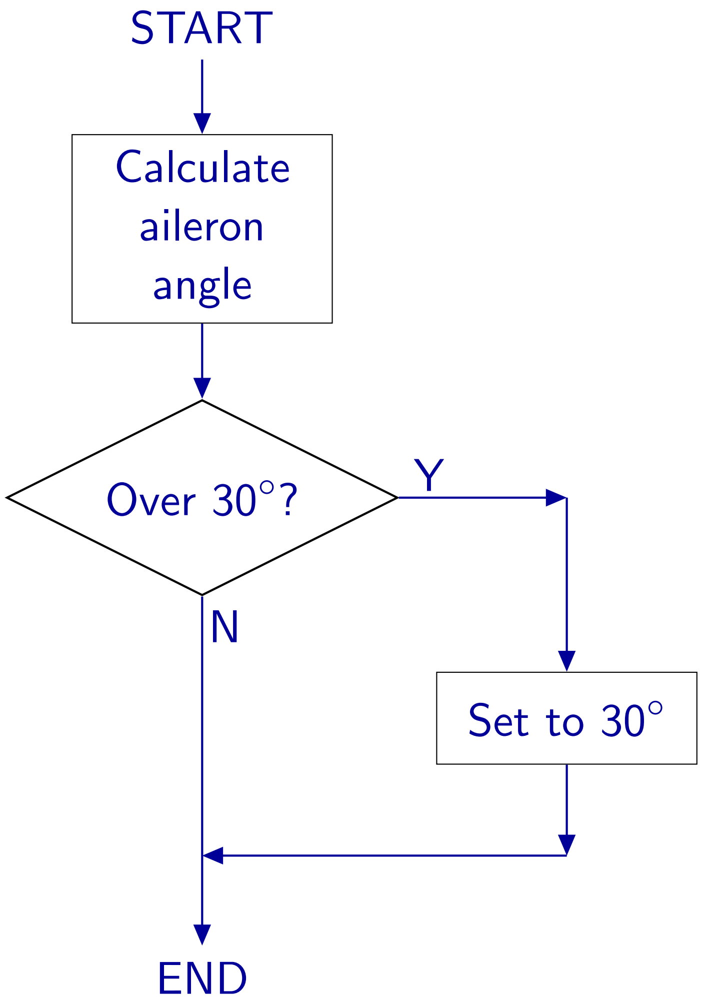
So far we have only considered the simplest of programs where operations are performed in a linear order. Example is the first lab session.



$$\begin{tikzcd}[row sep=large]
\text{START} \arrow[d] \\
\boxed{\text{Read in } v,\ \alpha} \arrow[d] \\
\boxed{\text{Calculate } q} \arrow[d] \\
\boxed{\text{Calculate } L} \arrow[d] \\
\boxed{\text{Print } L} \arrow[d] \\
\text{END}
\end{tikzcd}$$

Branching

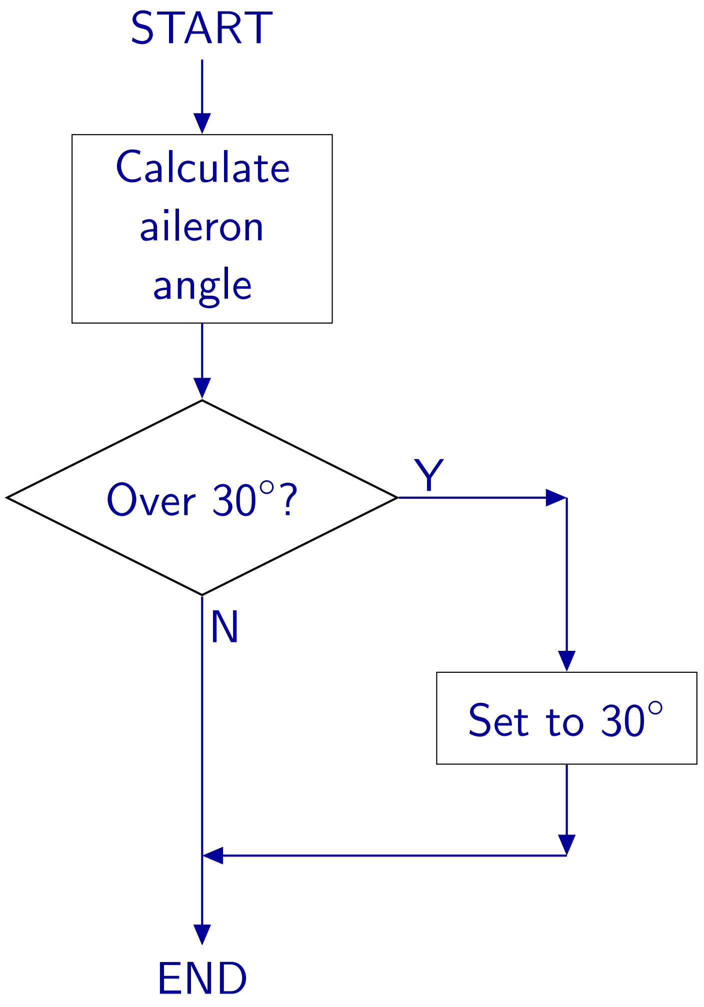
But, what if we need to do one of two calculations and the one we choose depends on some boolean test we can perform. Take, for example, setting an upper limit on an aileron deflection. For mechanical reasons, there will be a maximum deflection that the actuation system can give to the aileron.



Branching - `if`

This is where an `if` statement can be used. The `if` statement contains some boolean test where the answer can either be true or false. If the statement is true then the code within the `if` statement is executed, otherwise it is not and the program continues as normal.

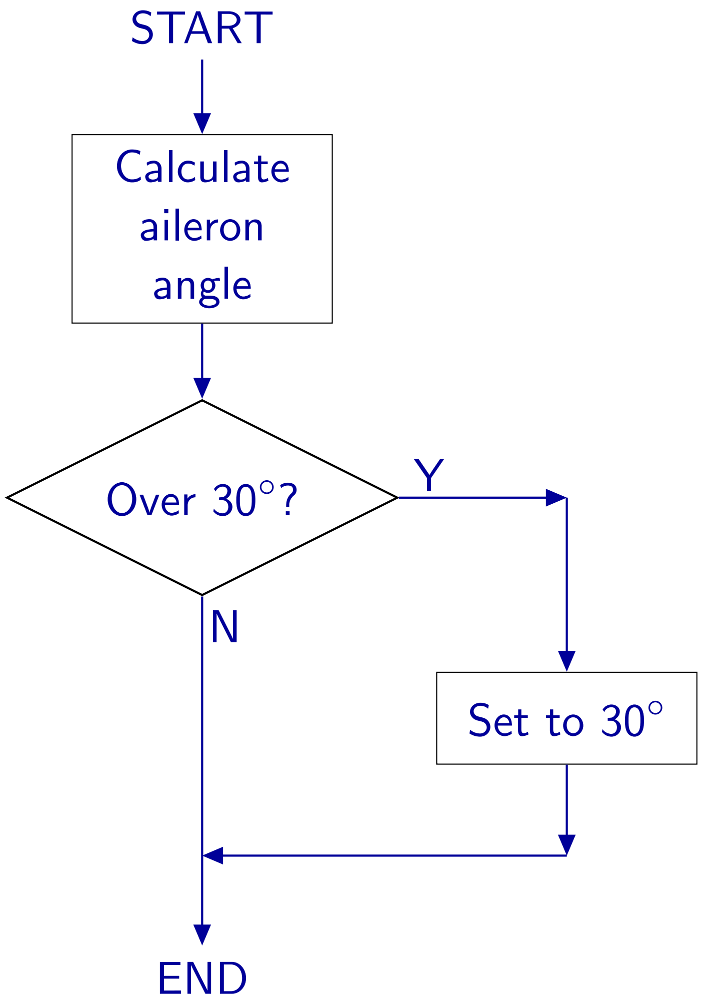
DEMO: if statement



Branching - `if`

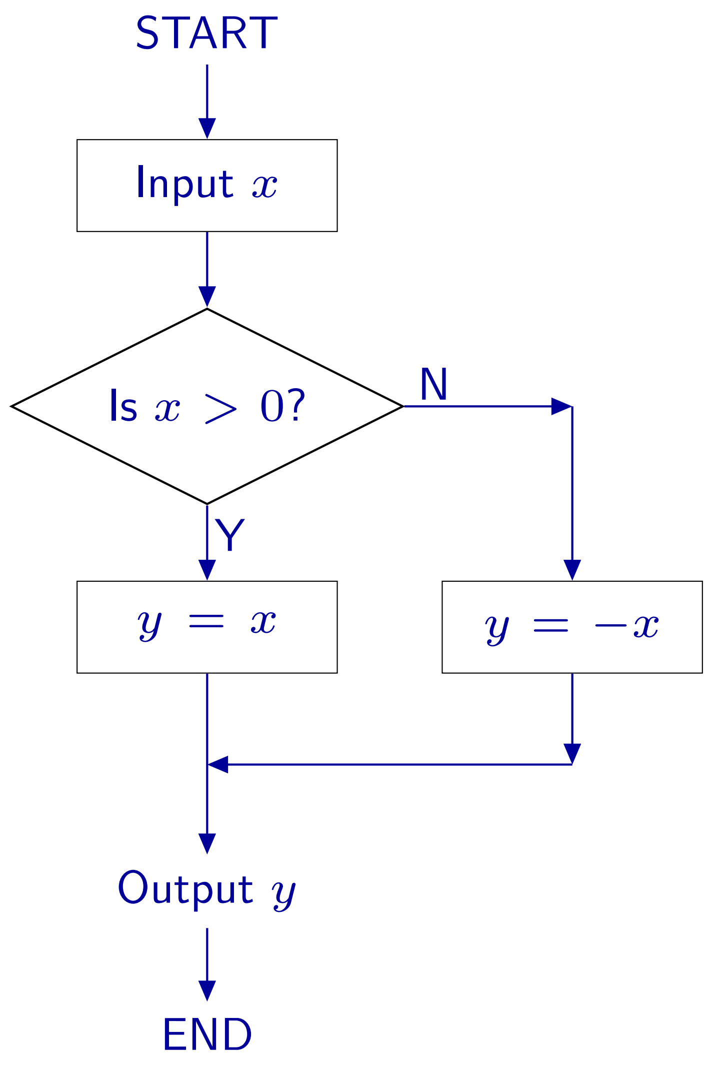
This is where an `if` statement can be used. The `if` statement contains some boolean test where the answer can either be true or false. If the statement is true then the code within the `if` statement is executed, otherwise it is not and the program continues as normal.

```
if (ailAngle>30.0) {

  ailAngle=30.0;

}
```



Branching - `if-else`

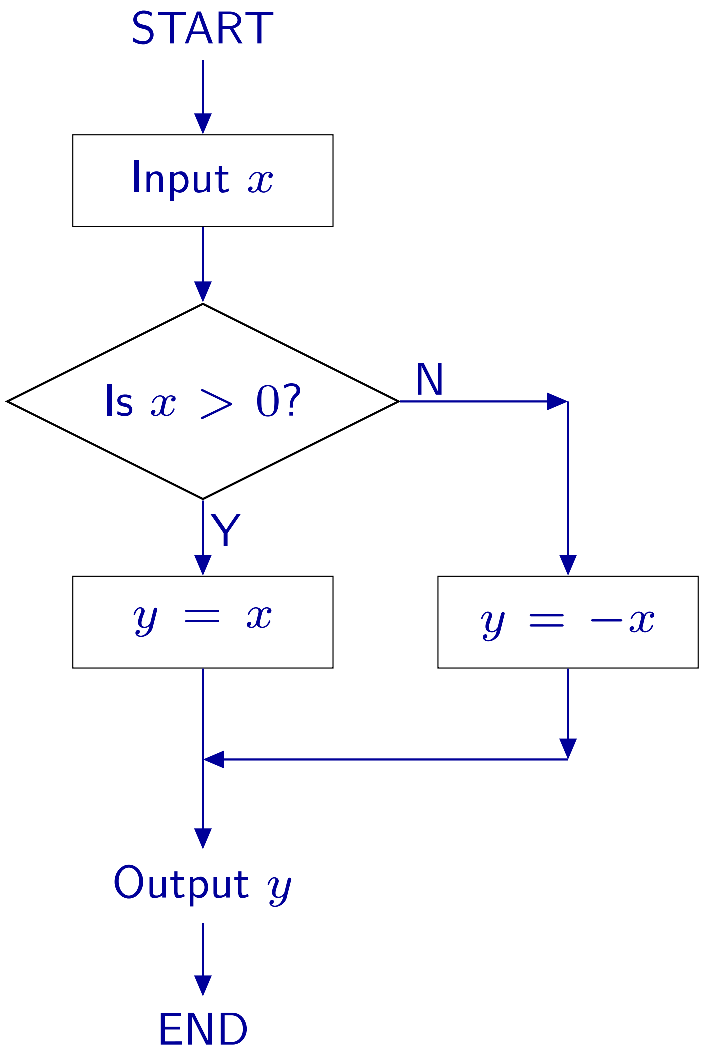
A situation may also arise where we need to execute another statement in the event that the first boolean test is not satisfied. For example, the absolute value flowchart that we saw in the first lecture.



Branching - `if-else`

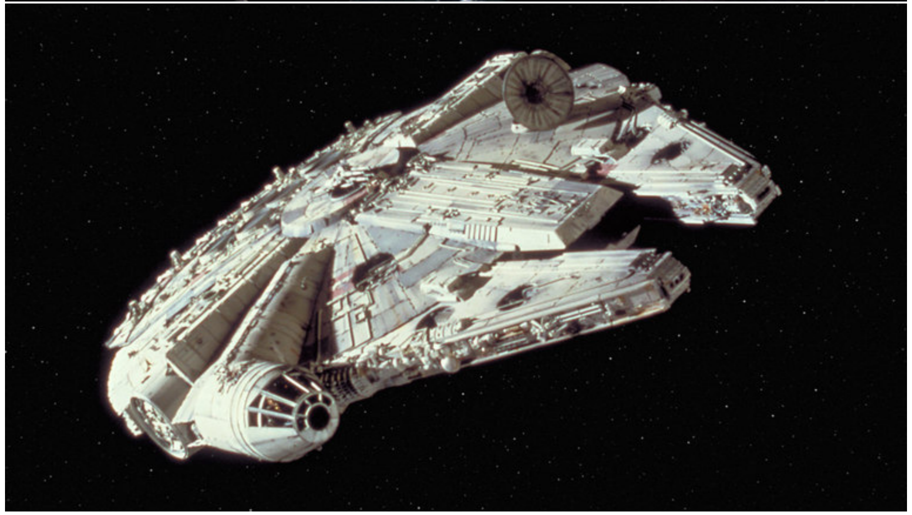
A situation may also arise where we need to execute another statement in the event that the first boolean test is not satisfied. For example, the absolute value flowchart that we saw in the first lecture.

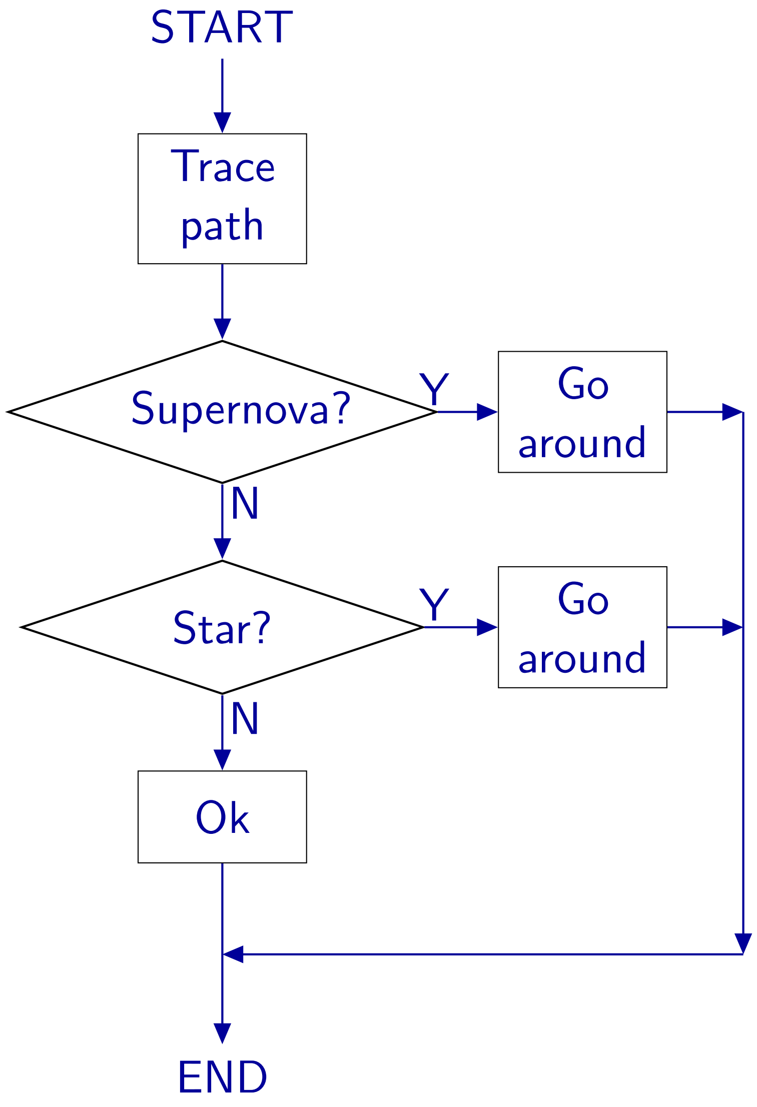
```
if (x>0.0) {

  y=x;

} else {

  y=-x;

}
```



Branching - `if-elseif-else`

We can also have multiple boolean tests after the first boolean test. Consider tracing a path through hyperspace:

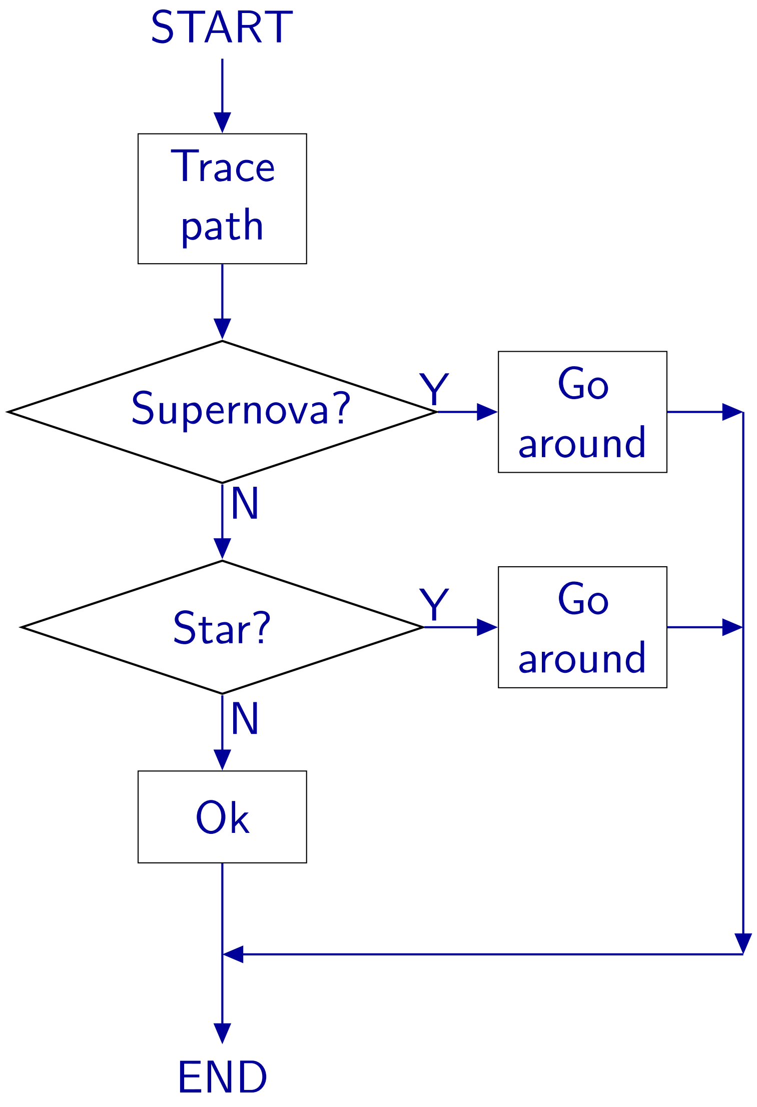




Branching - `if-elseif-else`

We can also have multiple boolean tests after the first boolean test. Consider tracing a path through hyperspace:

```
if (xpath == xsupernova) {
  xpath=xpath+1;
}
else if (xpath == xstar) {
  xpath=xpath-1;
}
else {
  xpath=xpath;
}
```



Branching

The syntax for the `if-elseif-else` statement (you can simplify to `if-else`, `if-elseif` or `if` as appropriate is:

```
if (<expression>) {<statements>}
else if (<expression>) {<statements>}
else {<statements>}
```

Common boolean operators used for testing are:

Expression	True iff
`a < b`	a is strictly less than b
`a > b`	a is strictly greater than b
`a <= b`	a is less than or equal to b
`a >= b`	a is greater than or equal to b
`a == b`	a is equal to b
`a != b`	a is not equal to b
`a && b`	both a and b are true
`a || b`	either a or b is true

Branching

We can have individual tests

```
if (a >= b) { ... }
```

Or we can have multiple tests:

```
if ((a<c) || (a>b)) { ... }
```

Branching - `switch`

In some cases of branching, we may require too many `if` statements than is practical. For example, we may have a number of options we wish to execute, and there maybe ten options. We could, of course, write ten `if` statements, however, this is an extremely inefficient way of doing things. For this scenario, the `switch` statement is perfect. The syntax is:

```
switch (<var>) {
 case <val1> : <statements>;
  ...
 case <valn> : <statements>;
 default : <statements>;
}
```

- `<var>` is the variable which the switch is based upon
- `<val1>` is the value of `<var>` that allows execution of the block of `<statements>`
- `default` is used to catch any remaining values not covered by the values of `<val>`

Branching - `switch`

An example would be to run one of two options (these could be different functions, which you will learn about in later lectures):

```c
#include<stdio.h>
int main() {
  int num;
  printf("Input a number:\n");
  scanf("%i",&num);
  switch (num) {
    case 1 : printf("Running option 1\n");
             break;
    case 2 : printf("Running option 2\n");
             break;
    default : printf("Running no options\n");
              break;
  }
  return(0);
}
```

Looping

Looping - `for`

We have now learnt how to branch. The next stage of basic control flow through a program is the idea of looping, or executing a block of code multiple times, either by a set number, or until a given condition is satisfied.

We will firstly consider the most basic type of loop, which is one where a given block of code is executed a set number of times.

A simple example would be if we wanted to calculate the product of all the numbers up to n (i.e. $n!$):

$$y = n! = \prod_{i=1}^{n} i \tag{1}$$

DEMO: factorial

Looping - `for`

When we know how many times we wish to perform the loop (this can also be something the user selects), a `for` loop is the structure we use. The syntax is:

```
for (<st1>; <exp>; <st2>) {<statements>}
```

- Start with the initialisation `<st1>`
- Perform `<statements>`
- Increment the loop `<st2>`
- Repeat until `<exp>` is satisfied

Looping - `for`

Practically, the syntax is normally written with an `int` counter variable, in this case `ii`:

```
for (ii=1; ii<=10; ii=ii+1) {<statements>}
```

- Start with the initialisation, `ii=1`
- Perform `<statements>`
- Increment the loop counter by 1 each time the loop is performed, `ii=ii+1`
- Repeat if `ii` less than or equal to ten

In this case, the `<statements>` block of code is performed ten times.

Looping - `for`

We can now write a code that will calculate $\prod^n i$ or $\sum^n i$

```c
#include <stdio.h>
int main() {
  // VARIABLE DECLARATIONS
  int ii, nfactorial=1, n=5;

  // MULTIPLY N TIMES
  for(ii=1; ii<=n; ii=ii+1){
     nfactorial=nfactorial*ii;
  }

  // PRINT RESULT
  printf("%i\n",nfactorial);
  return(0);
}
```

```c
#include <stdio.h>
int main() {
  // VARIABLE DECLARATIONS
  int ii, nsum=0, n=5;

  // ADD N TIMES
  for(ii=1; ii<=n; ii=ii+1){
     nsum=nsum+ii;
  }

  // PRINT RESULT
  printf("%i\n",nsum);
  return(0);
}
```

Note the difference in the initial value of the `nfactorial` and `nsum` values - if `nfactorial=0` initially, then the final answer will be 0 ☹

Looping - `break`

We may also wish to exit a loop early, if some other condition is met. In this case we use the `break` statement.

DEMO: break

```
for(ii=1; ii<=10; ii=ii+1){
   printf("Step %i\n",ii);
   printf("Do you wish to stop?\n");
   scanf("%1s",&c);
   if (c=='y') {
      break;
   }
}
```

The `break` statement will cause the code to exit the current loop. Note also the use of indenting (which shows the scope of the loops and conditionals) and that the `if` statement is nested within the loop.

Looping

We can also ‘nest’ loops within one another. In this case we will need separate counter variables for each loop.

```
nn=0;
for(ii=1; ii<=3; ii=ii+1){
  for(jj=1; jj<=2; jj=jj+1){
    nn=nn+1;
    printf("ii=%i jj=%i c=%i\n",ii,jj,nn);
  }
}
```

```
ii=1 jj=1 c=1
ii=1 jj=2 c=2
ii=2 jj=1 c=3
ii=2 jj=2 c=4
ii=3 jj=1 c=5
ii=3 jj=2 c=6
```

Notice that the inner loop is executed within the outer loop i.e. we perform the number of inner iterations, outer iteration times. We have set up a counter to show this.

WARNING: we can nest as many loops as we want but be careful as this can very quickly multiply up to a large number of iterations.

Looping - `while`

The `for` loop allows us to perform some statements (or block of statements) a set number of times. However, we may not know how many times we wish to perform a set of statements, but only know that we want to keep looping until we have fulfilled some criteria. This is where we will introduce the `while` (and `do-while`) statements.

The `while` statement will test an `<expression>` first and then execute the loop until that condition is no longer satisfied:

```
while (<expression>) {<statements>}
```

The `do while` statement is a similar construct, except the test is done at the end of the loop to ensure the loop is performed at least once:

```
do {<statements>} while (<expression>);
```

Looping - for

In fact, the for loop is just a specific example of the while loop so we can use them interchangeably e.g. calculating $n!$

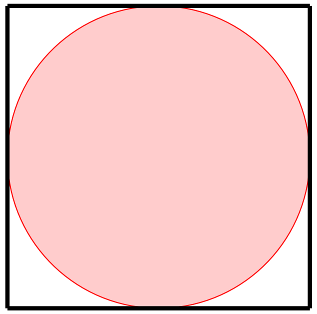
```c
#include <stdio.h>
int main() {
  // VARIABLE DECLARATIONS
  int ii=1, nfactorial=1, n=5;

  // MULTIPLY N TIMES
  do {
     nfactorial=nfactorial*ii;
     ii=ii+1;
  } while (ii<=n);

  // PRINT RESULT
  printf("%i\n",nfactorial);
  return(0);
}
```

```c
#include <stdio.h>
int main() {
  // VARIABLE DECLARATIONS
  int ii=1, nfactorial=1, n=5;

  // MULTIPLY N TIMES
  while (ii<=n) {
     nfactorial=nfactorial*ii;
     ii=ii+1;
  }

  // PRINT RESULT
  printf("%i\n",nfactorial);
  return(0);
}
```

Looping - `while`

As a demo, let's consider a program for estimating the value of π. The value of π is of course known to us ($\pi = 3.14159265\ldots$), but it can be estimated using a Monte-Carlo method. For this course, we won't go into the details of Monte-Carlo methods, but essentially they are a group of techniques that require performing many tests and doing statistical analysis on the results.

In this example, we can estimate pi by 'throwing some darts' onto a square. We can then count the number of darts inside a circle that is inscribed within the square.



$$\frac{A_{circ}}{A_{square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

$$\therefore \pi \approx 4\frac{N_{circ}}{N_{square}}$$

So lets find out how many darts we would have to throw until π is within 10%, 1%, 0.1% of the real value.

DEMO: estimating pi

Summary

- Have introduced branching of code based on conditionals:
  - `if`: test a few conditions to branch
  - `switch`: branch based on one variable with some options

- Have introduced looping based on conditionals and constants
  - `for`: loop over some code a set number of times
  - `while`: check condition not met then loop over code until condition is met
  - `do-while`: loop over code, check condition met, if not then repeat

LAB

- Effective use of control flow
- Numerical methods for solving equations