

Introduction to Programming using C++

Lecture Two: Further Syntax

Andrew Washbrook
`ajw@hep.ph.liv.ac.uk`

Factor calculation example


```
int myNumber, factor, nfactors = 0;
cout << "Give me a number\n";
cin >> myNumber;
for (int factor = 1; factor < myNumber; factor++) {
    if ((!(myNumber % factor)) && (factor != 1)) {
        cout << factor << " is a factor\n";
        ++nfactors;
    }
}
if (!nfactors) cout << myNumber << " is prime\n";
```

The for loop

```
for (initialisation; condition; action) {  
    statement 1;  
    :  
    statement n;  
}
```

- In the lifetime of the **for** statement:
 - the **initialisation** statement is called only once.
 - the **action** statement, along with all the statements within the block, are repeatedly executed (or looped) until..
 - the expression in the **condition** statement is false.

```
for (i = 1; i < 10; i = i+1)  
{  
    statement 1;  
    :  
    statement n;  
}
```

- ▶ Variable **i** is initialised to 1.
 - ▶ Process the statements in the block.
 - ▶ Increment **i** by 1.
 - ▶ If **i** is less than 10 continue.
- 

Using the for loop

- The **for** loop is much more than a simple iterator.
- The initialisation, the condition and the action parts of the **for** statement are all **optional**. Each of these statements can also include multiple expressions.

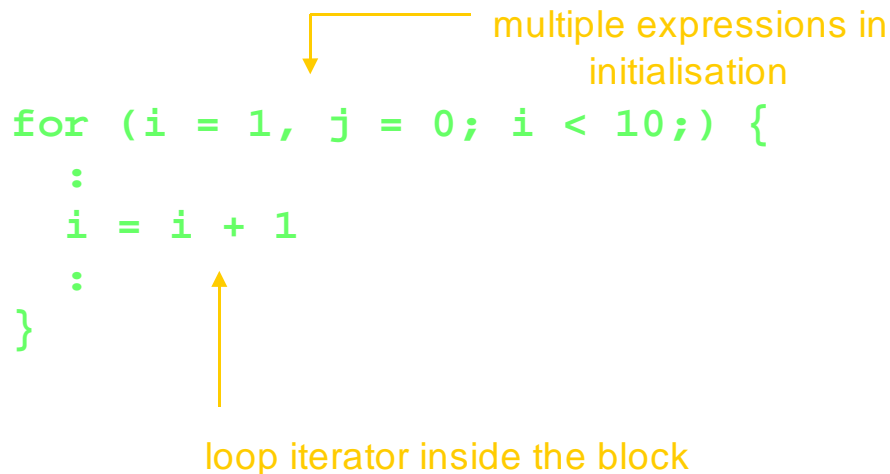


Diagram illustrating a **for** loop structure with annotations:

```
for (i = 1, j = 0; i < 10;) {  
    :  
    i = i + 1  
    :  
}
```

Annotations:

- multiple expressions in initialisation (points to `i = 1, j = 0`)
- loop iterator inside the block (points to `i = i + 1`)

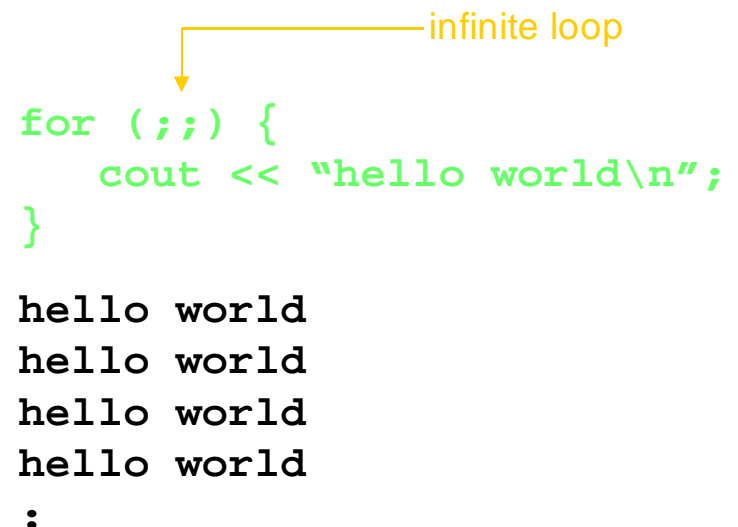


Diagram illustrating an infinite **for** loop structure with annotation:

```
for (;;) {  
    cout << "hello world\n";  
}
```

Annotation:

- infinite loop (points to `for (;;)`)

Output:

```
hello world  
hello world  
hello world  
hello world  
:
```

Increment Operators

factor++

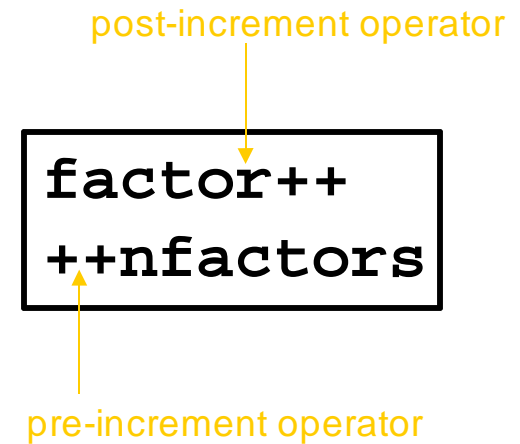
- The ++ operator increments the value of an integer variable by 1.
- This is the origin of the name “C++” !
- Similarly, the -- operator **decrements** the value of an integer variable by 1.

i++ is the same as i = i + 1

i-- is the same as i = i - 1

Pre and Post-fix Operators

- These both increment the value of the variable by 1 **but** they are not identical.
- Both operators will have specific uses in your code, know when to use them and use them properly.



value of `x` assigned to `y` and `x` is increased by 1

```
int x = 2, y = 0;  
y = x++; // y assigned the value 2  
y = ++x; // y assigned the value 4
```

value of `x` is increased by 1 and assigned to `y`

- The increment and decrement operators are **unary** operators. The mathematical operators are **binary** operators.

Logical Operators

```
if ((!(myNumber % factor)) && (factor != 1))
```

- Expressions are evaluated as false if their value is zero, true if non-zero.
- A conditional statement can therefore consist of multiple expressions with logical **and**, **or** and **not** operators.

!x	logical negation
x && y	logical and
x y	logical or

↑
precedence

```
if ((x == 2) && (y > 5))  
if ((x == 2) || (y > 5))
```

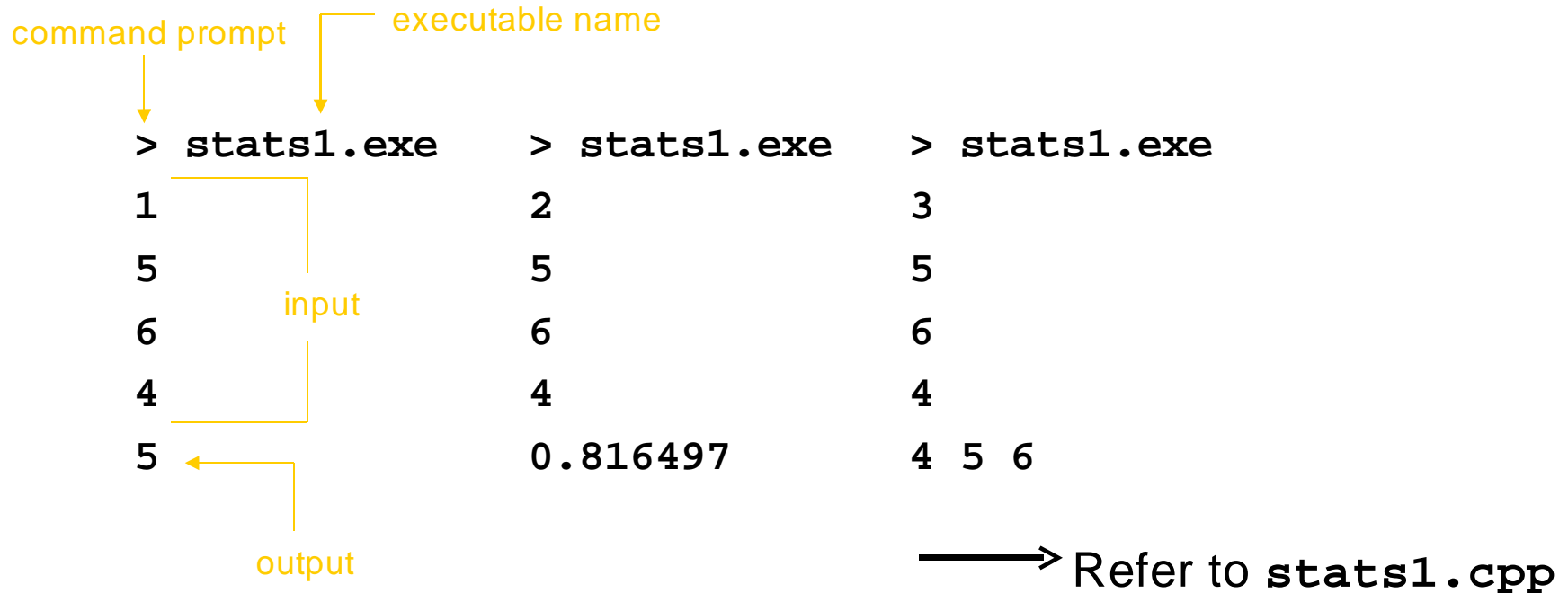
↑
ignored if x equals 2

```
int x = 1, y = 1, z = 2;  
if (x == 1 && y > 2 || z < 3) {  
    cout << "condition true\n";  
}
```

condition true

Simple statistics example

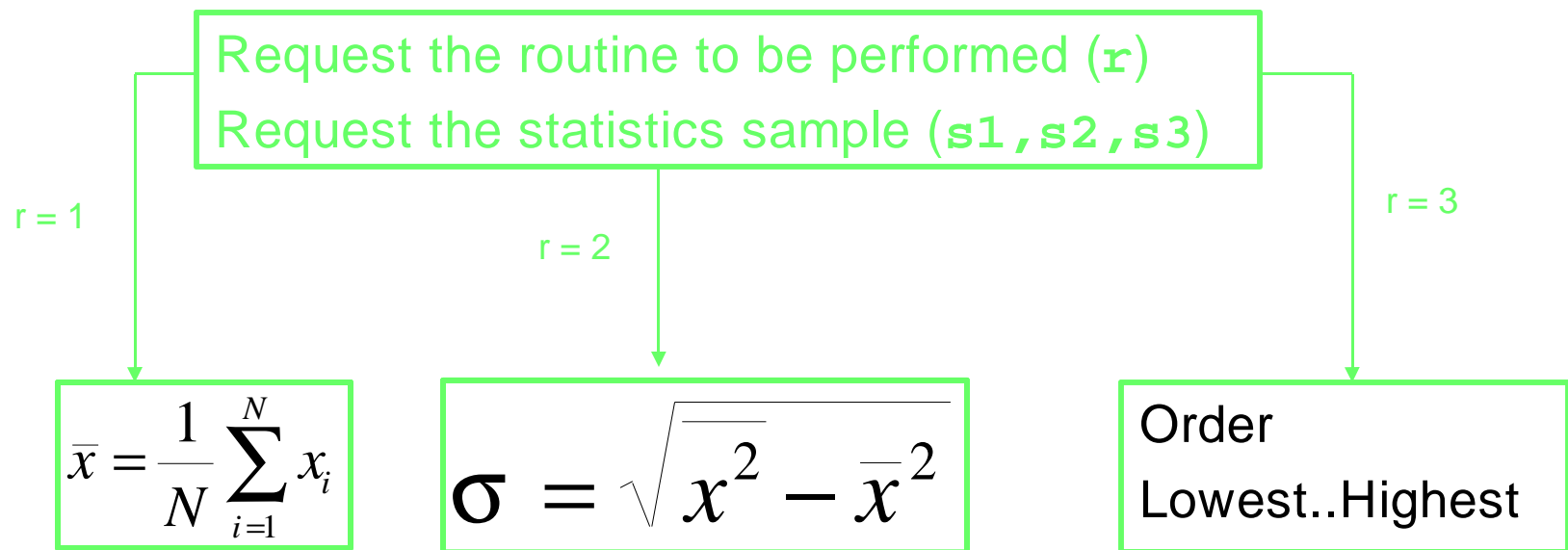
- We will now focus on the development of the small statistical calculation program.



What is the purpose of this program?

Purpose of the program

- The program performs one of the following routines:
calculate the **mean** of the input sample.
calculate the **standard deviation** of the input sample.
order the input sample from lowest to highest value.



Was this obvious from looking at the source code?

Purpose of the program

The function of the code is unclear to everyone apart from the author.

- Comments are required to explain the purpose of the code.
- Annotations are not just required within the source code. Explanatory text is required for the user of the program.

old input routine

`cin >> r;`

new input routine

```
// get a routine to run
cout << "\nWhat would you like to do?" << endl;
cout << "1 - Calculate the mean" << endl;
cout << "2 - Calculate the standard deviation" << endl;
cout << "3 - Order the sample lowest first" << endl;
cin >> r;
```

Adding whitespace

Source code needs spacing out to increase readability.

- The efficiency of the program does **not** decrease if **whitespace** is liberally used throughout the source code.
- Often useful to add line breaks to partition chunks of code.

a series of statements
associated with one task.

```
md=0;s1m=s1-m_2;s1ms=pow(s1m,2);  
s2m=s2-m_2;s2ms=pow(s2m,2);  
s3m=s3-m_2;s3ms=pow(s3m,2);md=s1ms+s2ms+s3ms;  
m_3=md/3;s=sqrt(m_3);
```

Comments
Whitespace

stats2.cpp

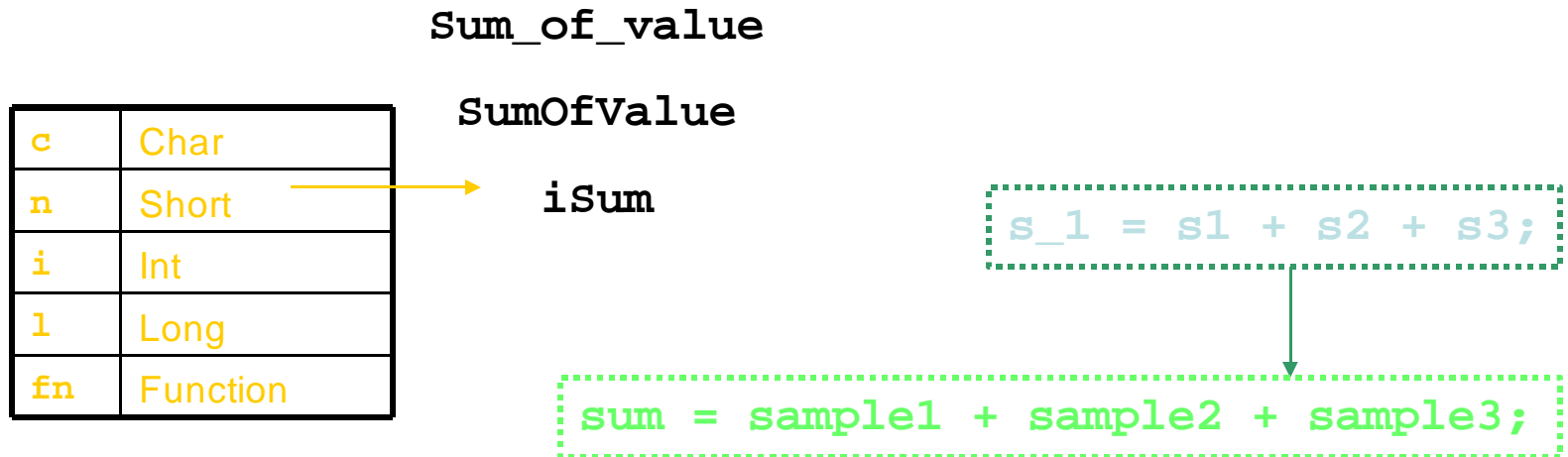
```
// calculate x_i - mean  
md = 0;  
s1m = s1 - m_2;  
s1ms = pow(s1m,2);  
s2m = s2 - m_2;  
s2ms = pow(s2m,2);  
s3m = s3 - m_2;  
s3ms = pow(s3m,2);  
md = s1ms + s2ms + s3ms;  
  
// calculate standard deviation  
m_3 = md / 3;  
s = sqrt(m_3);
```

Variable Naming

Variable names bear no relation to their role in the program.

- The purpose of the program can be interpreted much easier if the variables have names that indicate their role within the program.
- In other words, descriptive names for variables enable other developers to follow the "story" told by the code.

Some sample naming systems



Declaration Styles

All the variables are declared at the top of the program.

```
int main() {  
    int x = 1;  
    y = x; //  
    error  
    :  
}  
y assigned to  
x before y is  
declared
```

- In C++, a variable can be declared anywhere within the body of the program provided the declaration precedes the definition.
- All variables can be declared at the top of `main()`.

BUT..

- It is preferable to declare variables close to where they are first assigned a value.
- This seems too chaotic, why bother? This practice is introduced to implement an important feature of C++.

The Lifetime of Objects

The lifetime of an object associated with a variable:
starts with the declaration of the variable.
ends with the termination of the containing block.

```
int x = 4; int y = 2;  
if (y > 0) {  
    cout << "x = " << x << endl;  
}  
if (y > 1) {  
    int x = 7;  
    cout << "x = " << x << endl;  
    x++;  
}  
cout << "x = " << x << endl;
```

variable name is the same
but this is a different object

x = 4
x = 7
x = 4

refers to the object value
assigned outside of the
above block

- The same variable name can be repeated in different blocks.
- Outside of the block the variable runs out of scope.
- The variable has a scope local to the block in which it was declared in.

Useful variable names

Moved declarations away
from the top of the main
program

—————→ **stats3.cpp**

Extending the sample size

The program can only handle a data sample of a fixed size.

- We would now like the program to accept a sample of any size.
- The first step is to add an option to dictate the size of the sample.

```
cout << "How big is the sample?" << endl;  
int size;  
cin >> size;
```

- But this is not enough to extend the sample size. The code has been specifically written to process three values:

```
sum = sample1 + sample2 + sample3;  
float sumDivSize = (sum / 3);
```

Extending the sample size

- Can a larger sample size be accommodated by simply adding more variables to hold the new values?
- Yes, but the idea is simply not scalable:

```
float sampleSwap;  
if (sample1 > sample2) {  
    sampleSwap = sample1;  
    sample1 = sample2;  
    sample2 = sampleSwap;  
}  
if (sample1 > sample3) {  
    sampleSwap = sample1;  
    sample1 = sample3;  
    sample3 = sampleSwap;  
}  
if (sample2 > sample3) {  
    sampleSwap = sample2;  
    sample2 = sample3;  
    sample3 = sampleSwap;  
}
```

number of statements needed
to order the data sample.



Sample size	Statements
3	13
4	22
5	34
10	139

- The more pertinent question is, how does the program cope with holding the values of the sample if the sample size is only to be decided at runtime?

Arrays

type variable[int]

- An **array** is an **ordered** collection of objects of the same type.

```
int x[10];  
int y = x[4];  
float x[5] = {5.0,4.3,6.1,2.1,9.2};  
int x[2][3] = {{1,2,3},{4,5,6}};
```

example of an array element assigned
to non-array variable.

arrays can be declared and initialised
in the same statement.

arrays can be multi-dimensional
(matrices)

- An object stored in an array is referred to as an array **element**.
- In the first example the array **x** holds 10 objects of type **integer**.
The first element in the array **x** is **x[0]** and the last element is **x[9]**.

Arrays are an integral part of most other computing languages but are approached with suspicion in C++ (see Lecture 5).

Using for loops

- An array can be used to store a data sample of any reasonable size.
- However, this is still not the final solution. The routines are still limited to only accepting three values.
- Remove this dependency by using a for loop to iterate through all the data sample.

```
cout << "How big is the sample?" << endl;
int size;
cin >> size;

// get sample values
float sample[size];
.
```

```
// calculate sum of sample
float sum = 0;
sum = sample1 + sample2 + sample3;
```

```
// calculate sum of sample
int samplecount;
for (sampleCount = 1; sampleCount <= size; sampleCount = sampleCount + 1) {
    int sampleCountArray = sampleCount - 1;
    sum = sum + sample[sampleCountArray];
}
```

value of size is decided at runtime

Start filling array from 0

The Bubble Sort

- The for loop (and a nested for loop) will significantly improve the current ordering routine.

this **algorithm** is commonly known as the bubble sort.

```
float sampleSwap;  
if (sample1 > sample2) {  
    sampleSwap = sample1;  
    sample1 = sample2;  
    sample2 = sampleSwap;  
}  
if (sample1 > sample3) {  
    sampleSwap = sample1;  
    sample1 = sample3;  
    sample3 = sampleSwap;  
}  
if (sample2 > sample3) {  
    sampleSwap = sample2;  
    sample2 = sample3;  
    sample3 = sampleSwap;  
}
```

```
for (int i = 0; i < (size - 1); i = i + 1) {  
    for (int j = 0; j < (size - 1); j = j + 1) {  
        if (sample[j] > sample[j+1]) {  
            float swap = sample[j];  
            sample[j] = sample[j+1];  
            sample[j+1] = swap;  
        }  
    }  
}
```

Routines can process a data sample of an unrestricted size.

stats4.cpp

Error Catching

There is no way of handling incorrect input
by the user.

- If the input value for the routine is not in the range 1 to 3 the code will compile but the program will not indicate an error back to the user.
- Do not assume that everyone who uses your code will be competent!

incorrect option
entered by the user

```
> stats
What would you like to do?
1 - Calculate the mean
2 - Calculate the standard deviation
3 - Order the sample lowest first
4
How big is the sample?
3
Sample 1: 4
Sample 2: 5
Sample 3: 6
>_
```

program exited normally

Error Catching

- If the input routine value is not in the desired range then an error message will not be reported back to the user.
- The solution is to replace the separate `if` blocks with one `if - else if - else` block.

```
if (request == 1) {  
    :  
}  
if (request == 2) {  
    :  
}  
if (request == 3) {  
    :  
}
```

Program protected
against basic user errors.

↓
stats5.cpp

```
if (request == 1) {  
    :  
}  
else if (request == 2) {  
    :  
}  
else if (request == 3) {  
    :  
}  
else {  
    cerr << "Routine number does not exist\n";  
    return 1;  
}
```

return 1 rather than 0 to indicate to the
operating system that the program did
not run successfully.

cerr captures error messages, clog
captures log messages.

Unnecessary Code

Code is unnecessarily verbose.

- It is good coding practice to attain a balance between conciseness and clarity.

Move the declaration of this variable into the `for` loop.

Use the post-increment operator `++` instead.

```
// calculate sum of sample
int sampleCount;
for (sampleCount = 1; sampleCount <= size; sampleCount = sampleCount + 1) {
    int sampleCountArray = sampleCount - 1;
    sum = sum + sample[sampleCountArray];
}
```

The variable `sampleCountArray` is not needed.

- The less variables you have in an algorithm the less potential mistakes you will make! Well, almost..

Self Assigned Operators

```
x = x + 5; // add 5 to x  
x += 5; // add 5 to x  
x += n; // add n to x
```

- It is common coding practice to apply a mathematical operation to a variable and then assign the result back to the same variable.
- in C++, the same result can be achieved by applying self-assigned operators.

- There are similar self-assigned operators for the other mathematical operations.

+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division

The source code is more concise.



stats6.cpp

```
sum = sum + sample[sampleCount-1];
```



```
sum += sample[sampleCount-1];
```

Alternatives to the for loop

Simplify the iteration routines in the code

```
// calculate sum of sample
for (int sampleCount = 1; sampleCount <= size; sampleCount++) {
    sum += sample[sampleCount-1];
}
```

- The for loop is much more versatile than just iterating the value of a conditional variable by 1.
- There are other loops that provide the same result with less syntactic baggage..

The while loops

- There are two other types of loops you can use in C++:

condition check is done
before the block is
executed.

```
while (expression) {  
    statement;  
    :  
}
```

repeatedly executes the statements in the
block **while** the expression is true.

```
do {  
    statement;  
    :  
} while (expression)
```

condition is checked is done **after** the block
is executed.

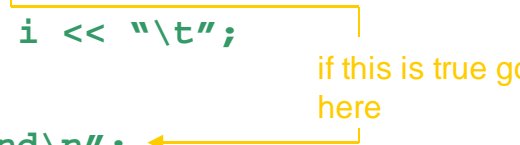
Which while loop should I use?

- Use the **while** loop if it is possible that the statement block may **never** be executed.
- Use the **do...while** loop if the statement block has to be executed **at least once**.

Breaking the loop

- It is possible to exit the **while**, **do...while** and **for** loops even if the condition is still true.

```
for (int i = 0; i < 5; i++) {  
    if (i == 3) break;  
    cout << i << "\t";  
}  
cout << "end\n";
```




```
int val;  
while (cin >> val) {  
    if (!val) break;  
}  
cout << "out of the loop\n";
```

1 2 3 end

1
0
out of the loop

```
int x = 0, y = 2;  
while (x < 3) {  
    x++;  
    if (x == y) continue;  
    cout << x << "\t";  
}  
cout << "end\n";
```



1 3 end

- The **break** statement can be used to exit the nearest enclosed loop.
- The **continue** statement can be used to skip remaining statements in the block.
- There is one other unspeakable way of getting out of a loop..the `goto` statement.

Iteration in a while loop

- How does the **while** loop become an iteration tool?

```
int x = 1;
while (x < 10) {
    :
    x++;
}
```

↑
iterator is found in body of the loop

```
int x = 0;
while (x++ < 10) {
    :
}
```

↑
increment operator in condition expression
also provides iteration.

```
for (int sampleCount = 1; sampleCount <= size; sampleCount++) {
    sum = sum + sample[sampleCount-1];
}
```

```
// calculate sum of sample
int sampleCount = 0;
while (++sampleCount <= size) sum += sample[sampleCount-1];
```

Used while loops for all
iterations.

↓
stats7.cpp

↑
Has this improved the readability of the code? That is up to you..