

# Introduction to Programming using C++

## Lecture Two: Further Syntax

Carl Gwilliam

[gwilliam@hep.ph.liv.ac.uk](mailto:gwilliam@hep.ph.liv.ac.uk)

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

# Factor calculation example

```
int myNumber, 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 i;  
    :  
    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 (int i=1; i<10; i++) {  
    statements;  
}
```

- Variable **i** initialised to **1**.
- Process statements in block.
- Increment **i** by **1**.
- If **i** 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.

```
// multiple expressions
// in initialisation
for (i = 1, j = 0; i < 10;) {
    :
    // iterator inside block
    i = i + 1
    :
}
```

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

```
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++      //the same as i = i + 1  
i--      //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.

```
factor++  
++nfactors
```

```
int x = 2, y = 0;  
y = x++;      // x incremented after assignment -> y = 2  
y = ++x;      // x incremented before assignment -> y = 4
```

- 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.

PRECEDENCE ↑

!x	Logical negation
x && y	Logical AND
x    y	Logical OR

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

```
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 [**stats1.cpp**]

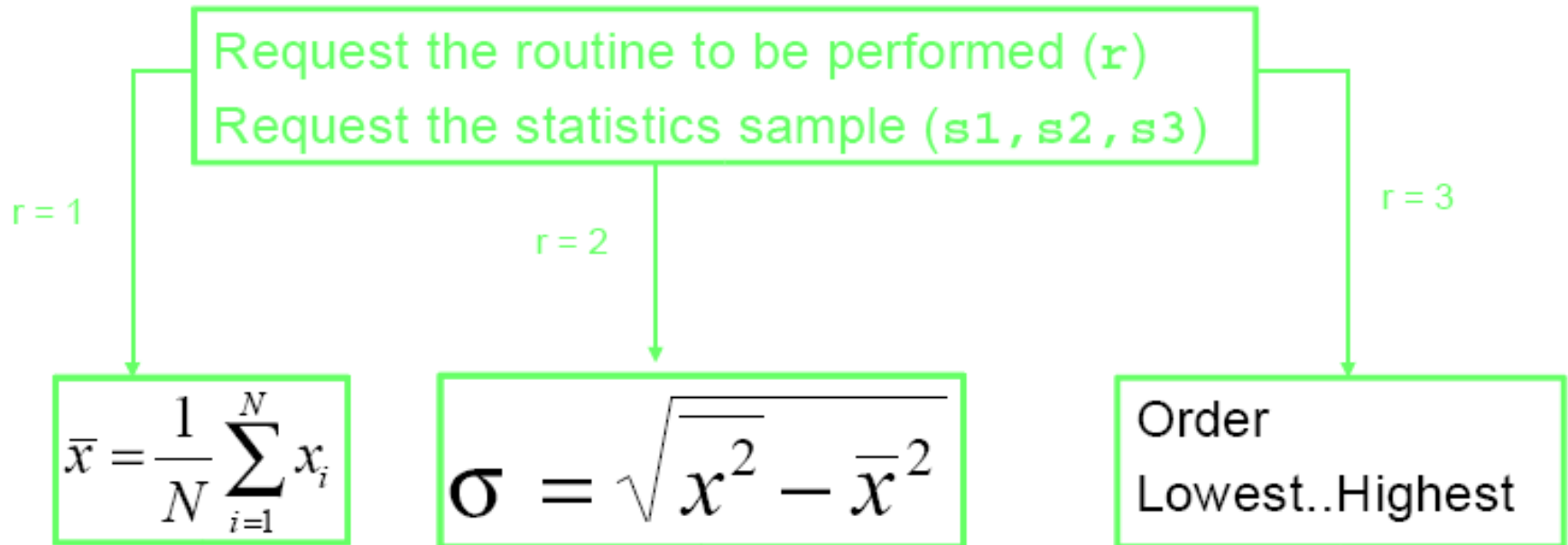
> stats1.exe	> stats1.exe	> stats1.exe
1	2	3
5	5	5
6	6	6
4	4	4
5	0.816397	4 5 6

- 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.
- 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 [**stats2.cpp**].

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

```
// calculate x_i - mean  
md = 0;  
sim = s1 - m_2;  
sims = pow(sim,2);  
s2m = s2 - m_2;  
s2ms = pow(s2m,2);  
s3m = s3 - m_2;  
s3ms = pow(s3m,2);  
md = sims + s2ms + s3ms;  
  
// calculate std  
// 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 conventions:

Sum\_of\_value

SumofValue

sumOfValue

iSum

// bad

s\_I = sI + s2 + s3;

// better

sum = sample1 + sample2 + sample3;

# Declaration Styles

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

```
int main() {  
    int x = 1;  
    // error  
    y = x;  
    :  
}
```

- In C++, a variable can be declared anywhere within the body of the program provided the declaration precedes the assignment.
- 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;    //diff object
    cout << "x = " << x << endl;
    x++;
}
// original object
cout << "x = " << x << endl;
```

- Same variable name can be repeated in different blocks.
- Outside of the block the variable runs out of scope.
- Variable has a scope local to the block in which it was declared
- See **[stats3.cpp]**.

x = 4

x = 7

x = 4

# 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?
- Yes, but this is not scalable:

Sample size	statements
3	13
4	22
5	34
10	139

- How does the program cope with holding the sample values if the sample size is only to be decided at runtime?

```
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;  
}
```



# Arrays

type variable [size]

- An **array** is an **ordered** collection of objs of **same** type.
- An object stored in an array is referred to as an array **element**.
- In the 1<sup>st</sup> example array **x** holds 10 objects of type **int**.
- The first element in the **x** is **x[0]** and the last is **x[9]**.

```
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}}; // 2D
```

- Arrays are an integral part of most other computing languages but are approached with suspicion in C++

# Using for loops

- An array can be used to store a data sample of any reasonable size (memory is limited).

```
cout << "How big is the sample?" << endl;  
int size;  
cin >> size;  
float sample[size]; //bad, but allowed  
                      //by some compilers  
float* sample = new float[size]; // better
```

- To dynamically assign memory (properly) you need to use the **new** syntax and **pointers**, we'll cover this later.
- However, this is not the final solution as the routines are still limited to only accepting three values ...

# Using for loops

- Remove this dependency by using a for loop to iterate through all the data sample.

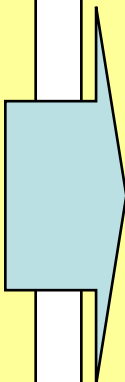
```
// calculate sum of sample
int sampleCount;

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

# 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;  
}
```

- Using another (nested) for loop can significantly improve the current ordering routine too:



```
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;  
        }  
    }  
}
```



**[stats4.cpp]**

# Error Catching

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

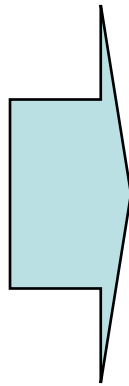
- If the input value for the routine is not in range 1 to 3 the code will compile but the program will not indicate an error back to the user.
- Do not assume that everyone who uses your code will be competent!

```
> 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
> _
```

# Error Catching

- 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) {  
    :  
}
```



```
if (request == 1) {  
    :  
} else if (request == 2) {  
    :  
} else if (request == 3) {  
    :  
} else {  
    cerr << "Routine doesn't exist\n";  
    return 1;  
}  
[stats5.cpp]
```

- Non-zero return code indicates to os that program did not terminate successfully
- cerr writes so stderr rather than stdout (also clog)

# Unnecessary Code

Code is unnecessarily verbose.

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

```
// calculate sum of sample
int sampleCount;    //declaration can be inside for loop
for (sampleCount = 1; sampleCount <= size;
    sampleCount = sampleCount + 1) {    //can use ++
    int sampleCountArray = sampleCount - 1;
    // above line is unneeded
    sum = sum + sample[sampleCountArray];
}
```

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

# Self Assigned Operators

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

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

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

- The code is then more concise  
**[stats6.cpp]**

```
sum = sum + sample[sampleCount-1];
// more concise
sum += sample[sampleCount-1];
```



# Alternatives to the for loop

Simplify the iteration routines in the code

```
// calculate sum of sample  
int sampleCount;  
for (sampleCount = 1; sampleCount <= size;  
     sampleCount = sampleCount + 1) {  
    sum = 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++:

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

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

- The statements within the block are repeatedly executed while expression is true.

## Which 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 **while**, **do..while** and **for** loops even if the condition is still true.
- 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's one other unspeakable way out of a loop ...

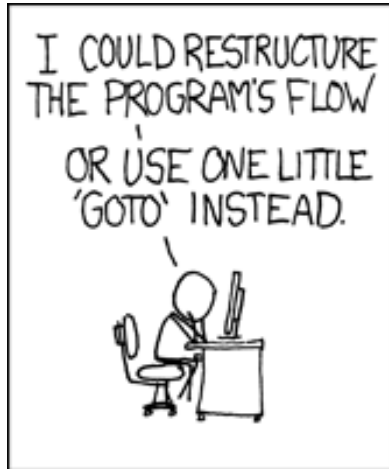
```
for (int i = 0; i < 5; i++) {  
    if (i == 3) break;  
    cout << i << "\t";  
}  
cout << "end\n";
```

0 1 2 end

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

1 3 end

... goto, but don't use it or:



# Iteration in a while loop

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

```
//increment in body  
int x = 1;  
while (x < 10) {  
    :  
    x++;  
}
```

```
//increment in  
//condition  
int x = 0;  
while (x++ < 10) {  
    :  
}
```

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

[stats7.cpp]

//replaced by ...

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