Computing and Statistical Data Analysis (PH4515, UofL PG Lectures)

Glen Cowan
Physics Department
Royal Holloway, University of London
Egham, Surrey   TW20 0EX

01784 443452
g.cowan@rhul.ac.uk
www.pp.rhul.ac.uk/~cowan/stat_course.html

Computing and Statistical Data Analysis: C++ Outline

1 Introduction to C++ and UNIX environment
2 Variables, types, expressions, loops
3 Type casting, functions
4 Files and streams
5 Arrays, strings, pointers
6 Classes, intro to Object Oriented Programming
7 Memory allocation, operator overloading, templates
8 Inheritance, STL, `gmake, ddd`

Some resources (computing part)

There are many web based resources, e.g.,

`www.doc.ic.ac.uk/~wjk/C++Intro` (Rob Miller, IC course)

`www.cplusplus.com` (online reference)

`www.icce.rug.nl/documents/cplusplus` (F. Brokken)

See links on course site or google for "C++ tutorial", etc.

There are thousands of books – see e.g.

W. Savitch, *Problem Solving with C++*, 4th edition
(lots of detail – very thick).
B. Stroustrup, *The C++ Programming Language*
(the classic – even thicker).
Lippman, Lajoie (& Moo), *C++ Primer*, A-W, 1998.

Introduction to UNIX/Linux

We will learn C++ using the Linux operating system

Open source, quasi-free version of UNIX

UNIX and C developed ~1970 at Bell Labs

Short, cryptic commands: `cd`, `ls`, `grep`, …

Other operating systems in 1970s, 80s ‘better’, (e.g. VMS) but, fast ‘RISC processors’ in early 1990s needed a cheap solution → we got UNIX

In 1991, Linus Torvalds writes a free, open source version of UNIX called Linux.

We currently use the distribution from CERN

Basic UNIX

UNIX tasks divide neatly into:

interaction between operating system and computer (the kernel),
interaction between operating system and user (the shell).

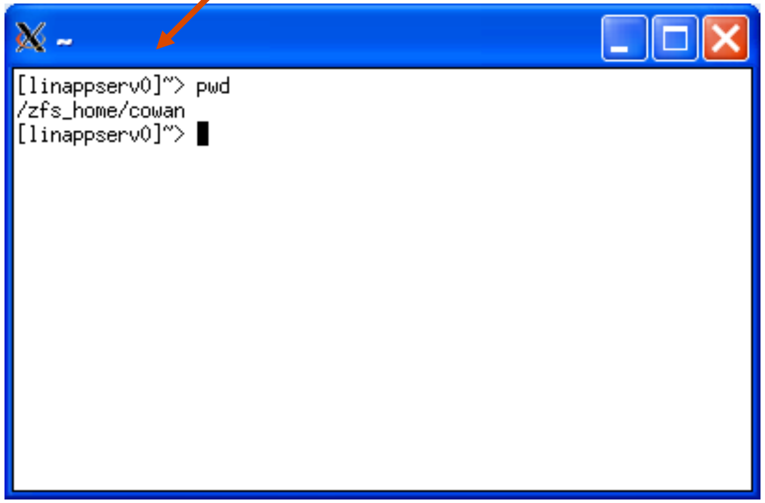
Several shells (i.e. command sets) available: sh, csh, tcsh, bash, …

Shell commands typed at a prompt, here **`[linappserv0]~>`** often set to indicate name of computer:

Command **`pwd`** to "print working directory", i.e., show the directory (folder) you're sitting in.

Commands are case sensitive.

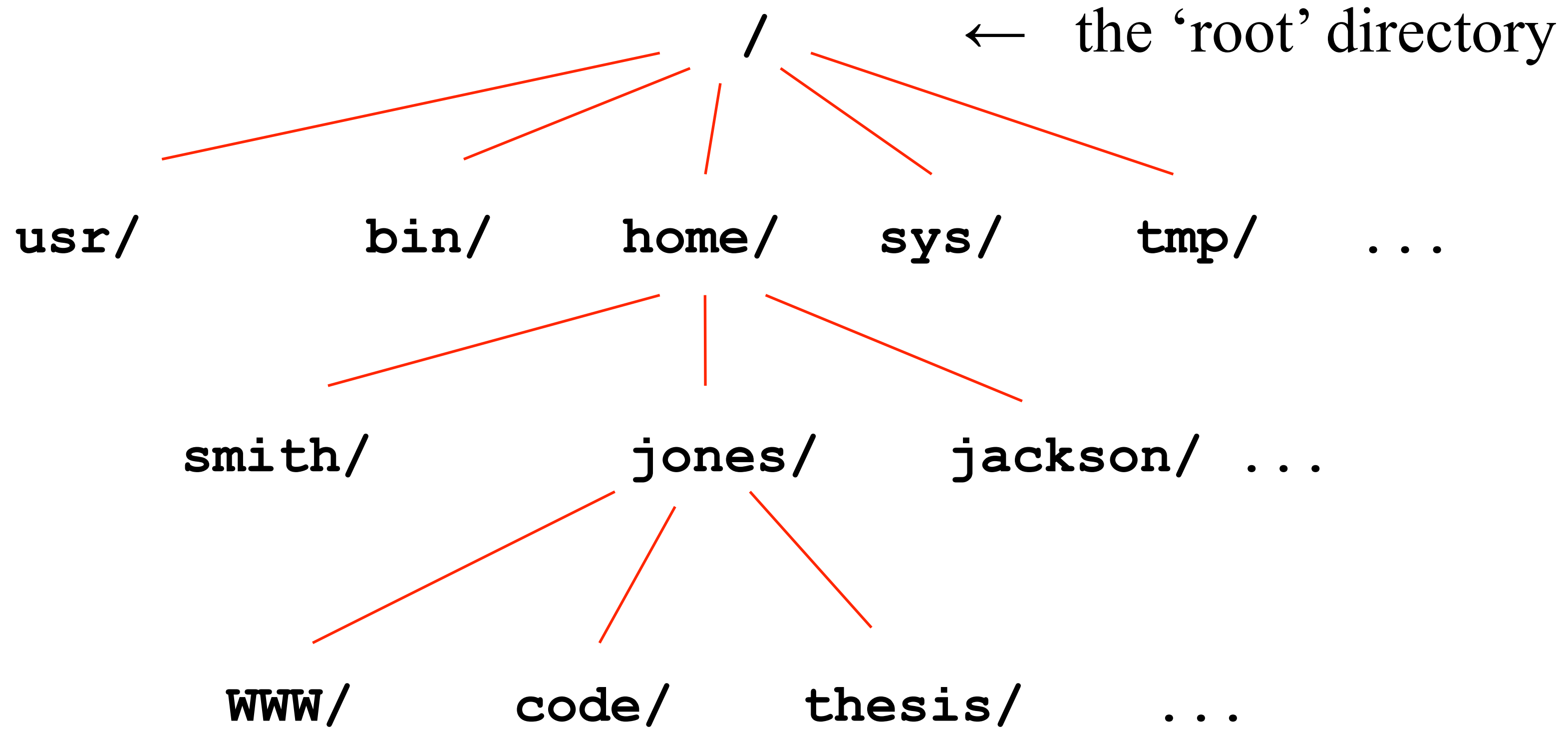
`PWD` will not work .



```
[linappserv0]~> pwd
/zfs_home/cowan
[linappserv0]~> █
```

UNIX file structure

Tree-like structure for files and directories (like folders):



File/directory names are case sensitive: **`thesis`** $\neq$ **`Thesis`**

Simple UNIX file tricks

A complete file name specifies the entire ‘path’

```
/home/jones/thesis/chapter1.tex
```

A tilde points to the home directory:

`~/thesis/chapter1.tex` ← the logged in user (e.g. jones)

`~smith/analysis/result.dat` ← a different user

Single dot points to current directory, two dots for the one above:

`/home/jones/thesis` ← current directory

`../code` ← same as /home/jones/code

A few UNIX commands (case sensitive!)

`pwd`	Show present working directory
`ls`	List files in present working directory
`ls -la`	List files of present working directory with details
`man ls`	Show manual page for `ls`. Works for all commands.
`man -k` ***`keyword`***	Searches man pages for info on “keyword”.
`cd`	Change present working directory to home directory.
`mkdir` ***`foo`***	Create subdirectory foo
`cd` ***`foo`***	Change to subdirectory foo (go down in tree)
`cd ..`	Go up one directory in tree
`rmdir` ***`foo`***	Remove subdirectory foo (must be empty)
`emacs` ***`foo`*** `&`	Edit file foo with emacs (& to run in background)
`more` ***`foo`***	Display file foo (space for next page)
`less` ***`foo`***	Similar to `more` ***`foo`***, but able to back up (`q` to quit)
`rm` ***`foo`***	Delete file foo

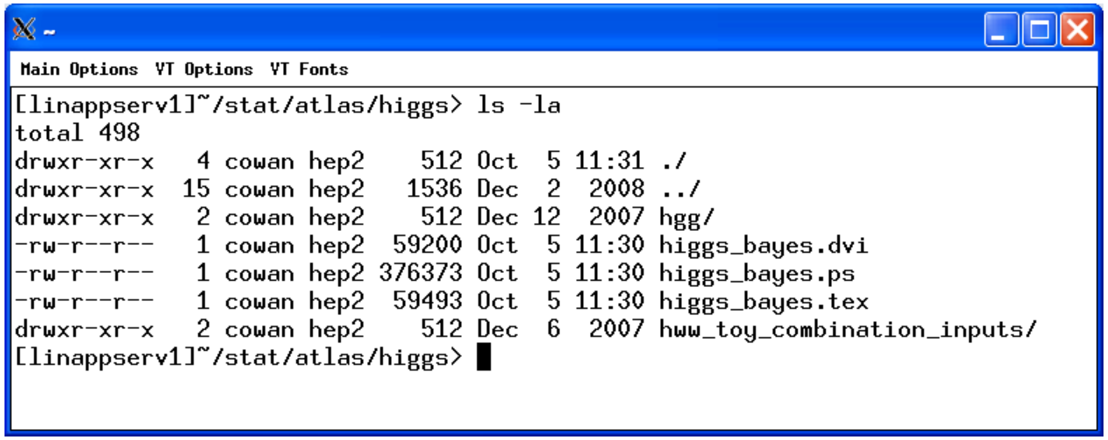
A few more UNIX commands

`cp foo bar`	Copy file foo to file bar, e.g., `cp ~smith/foo ./` copies Smith's file foo to my current directory
`mv foo bar`	Rename file foo to bar
`lpr foo`	Print file foo. Use `-P` to specify print queue, e.g., `lpr -Plj1 foo` (site dependent).
`ps`	Show existing processes
`kill 345`	Kill process 345 (`kill -9` as last resort)
`./foo`	Run executable program foo in current directory
`ctrl-c`	Terminate currently executing program
`chmod ug+x foo`	Change access mode so user and group have privilege to execute foo (Check with `ls -la`)

Better to read a book or online tutorial and use `man` pages

UNIX file access

If you type **ls -la**, you will see that each file and directory is characterized by a set of file access rights:



```
[linappserv1]~/stat/atlas/higgs> ls -la
total 498
drwxr-xr-x   4 cowan hep2    512 Oct  5 11:31 ./
drwxr-xr-x  15 cowan hep2   1536 Dec  2  2008 ../
drwxr-xr-x   2 cowan hep2    512 Dec 12  2007 hgg/
-rw-r--r--   1 cowan hep2  59200 Oct  5 11:30 higgs_bayes.dvi
-rw-r--r--   1 cowan hep2 376373 Oct  5 11:30 higgs_bayes.ps
-rw-r--r--   1 cowan hep2  59493 Oct  5 11:30 higgs_bayes.tex
drwxr-xr-x   2 cowan hep2    512 Dec  6  2007 hww_toy_combination_inputs/
[linappserv1]~/stat/atlas/higgs> 
```

Three groups of letters refer to: user (u), group (g) and other (o). The possible permissions are read (r), write (w), execute (x).

By default, everyone in your group will have read access to all of your files. To change this, use **chmod**, e.g.

```
chmod go-rwx hgg
```

prevents group and other from seeing the directory **hgg**.

Introduction to C++

Language C developed (from B) ~ 1970 at Bell Labs

Used to create parts of UNIX

C++ derived from C in early 1980s by Bjarne Stroustrup

"C with classes", i.e., user-defined data types that allow "Object Oriented Programming".

Java syntax based largely on C++ (head start if you know java)

C++ is case sensitive (**`a`** not same as **`A`**).

Currently most widely used programming language in High Energy Physics and many other science/engineering fields.

Recent switch after four decades of FORTRAN.

Compiling and running a simple C++ program

Using,e.g., emacs, create a file **HelloWorld.cc** containing:

```
// My first C++ program
#include <iostream>
using namespace std;
int main(){
  cout << "Hello World!" << endl;
  return 0;
}
```

We now need to compile the file (creates machine-readable code):

```
g++ -o HelloWorld HelloWorld.cc
```

$\uparrow$ Invokes compiler (gcc) $\nwarrow$ name of output file $\nwarrow$ source code

Run the program:

```
./HelloWorld
Hello World!
```

← you type this
← computer shows this

Notes on compiling/linking

```
g++ -o HelloWorld HelloWorld.cc
```

is an abbreviated way of saying first

```
g++ -c HelloWorld.cc
```

Compiler (`-c`) produces **`HelloWorld.o.`** (‘object files’)
Then ‘link’ the object file(s) with

```
g++ -o HelloWorld HelloWorld.o
```

If the program contains more than one source file, list with spaces; use \ to continue to a new line:

```
g++ -o HelloWorld HelloWorld.cc Bonjour.cc \
GruessGott.cc YoDude.cc
```

Writing programs in the Real World

Usually create a new directory for each new program.

For trivial programs, type compile commands by hand.

For less trivial but still small projects, create a file (a ‘script’) to contain the commands needed to build the program:

```
#!/bin/sh
# File build.sh to build HelloWorld
g++ -o HelloWorld HelloWorld.cc Bonjour.cc \
GruessGott.cc YoDude.cc
```

To use, must first have ‘execute access’ for the file:

```
chmod ug+x build.sh
./build.sh
```

$\leftarrow$ do this only once

$\leftarrow$ executes the script

A closer look at HelloWorld.cc

`// My first C++ program` is a comment (preferred style)

The older 'C style' comments are also allowed (cannot be nested):

```
/*
   These lines
   here are comments
*/

/* and so are these */
```

You should include enough comments in your code to make it understandable by someone else (or by yourself, later).

Each file should start with comments indicating author's name, main purpose of the code, required input, etc.

More HelloWorld.cc – include statements

`#include <iostream>` is a compiler directive.

Compiler directives start with `#`. These statements are not executed at run time but rather provide information to the compiler.

`#include <iostream>` tells the compiler that the code will use library routines whose definitions can be found in a file called `iostream`, usually located somewhere under `/usr/include`

Old style was `#include <iostream.h>`

`iostream` contains functions that perform i/o operations to communicate with keyboard and monitor.

In this case, we are using the iostream object `cout` to send text to the monitor. We will include it in almost all programs.

More HelloWorld.cc

`using namespace std;`   More later.   For now, just do it.

A C++ program is made up of functions. Every program contains exactly one function called main:

```
int main(){
  // body of program goes here

  return 0;
}
```

Functions "return" a value of a given type; `main` returns `int` (integer).

The `()` are for arguments. Here `main` takes no arguments.

The body of a function is enclosed in curly braces: `{   }`

`return 0;`   means `main` returns a value of 0 .

Finishing up HelloWorld.cc

The ‘meat’ of HelloWorld is contained in the line

```
cout << "Hello World!" << endl;
```

Like all statements, it ends with a semi-colon.

`cout` is an “output stream object”.

You send strings (sequences of characters) to `cout` with `<<`

We will see it also works for numerical quantities (automatic conversion to strings), e.g., `cout << "x = " << x << endl;`

Sending `endl` to `cout` indicates a new line. (Try omitting this.)

Old style was `"Hello World!\n"`

C++ building blocks

All of the words in a C++ program are either:

Reserved words: cannot be changed, e.g.,

`if, else, int, double, for, while, class`, ...

Library identifiers: default meanings usually not changed, e.g., **`cout, sqrt`** (square root), ...

Programmer-supplied identifiers:

e.g. variables created by the programmer,

`x, y, probeTemperature, photonEnergy`, ...

Valid identifier must begin with a letter or underscore ("_") , and can consist of letters, digits, and underscores.

Try to use meaningful variable names; suggest lowerCamelCase.

Data types

Data values can be stored in variables of several types.

Think of the variable as a small blackboard, and we have different types of blackboards for integers, reals, etc.
The variable name is a label for the blackboard.

Basic integer type: `int` (also `short`, `unsigned`, `long int`, ...)

Number of bits used depends on compiler; typically 32 bits.

Basic floating point types (i.e., for real numbers):

`float`	usually 32 bits	
`double`	usually 64 bits	← best for our purposes

Boolean: `bool` (equal to `true` or `false`)

Character: `char` (single ASCII character only, can be blank),
no native ‘string’ type; more on C++ strings later.

Declaring variables

All variables must be declared before use.

Usually declare just before 1st use.

Examples

```
int main(){
  int numPhotons;         // Use int to count things
  double photonEnergy;    // Use double for reals
  bool goodEvent;         // Use bool for true or false
  int minNum, maxNum;     // More than one on line
  int n = 17;             // Can initialize value
  double x = 37.2;        // when variable declared.
  char yesOrNo = ‘y’;     // Value of char in ‘ ‘
      ...
}
```

Assignment of values to variables

Declaring a variable establishes its name; value is undefined (unless done together with declaration).

Value is assigned using = (the assignment operator):

```
int main(){
  bool aOK = true;  // true, false predefined constants
  double x, y, z;
  x = 3.7;
  y = 5.2;
  z = x + y;
  cout << "z = " << z << endl;
  z = z + 2.8;        // N.B. not like usual equation
  cout << "now z = " << z << endl;
      ...
}
```

Constants

Sometimes we want to ensure the value of a variable doesn't change.

Useful to keep parameters of a problem in an easy to find place, where they are easy to modify.

Use keyword **`const`** in declaration:

```
const int numChannels = 12;
const double PI = 3.14159265;

// Attempted redefinition by Indiana State Legislature
PI = 3.2;               // ERROR will not compile
```

Old C style retained for compatibility (avoid this):

```
#define PI 3.14159265
```

Enumerations

Sometimes we want to assign numerical values to words, e.g.,

January = 1, February = 2, etc.

Use an 'enumeration' with keyword `enum`

```
enum { RED, GREEN, BLUE };
```

is shorthand for

```
const int RED = 0;
const int GREEN = 1;
const int BLUE = 2;
```

Enumeration starts by default with zero; can override:

```
enum { RED = 1, GREEN = 3, BLUE = 7 }
```

(If not assigned explicitly, value is one greater than previous.)

Expressions

C++ has obvious(?) notation for mathematical expressions:

operation	symbol
addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`
modulus	`%`

Note division of `int` values is truncated:

```
int n, m;   n = 5;   m = 3;
int ratio = n/m;          // ratio has value of 1
```

Modulus gives remainder of integer division:

```
int nModM = n%m;               // nModM has value 2
```

Operator precedence

`*` and `/` have precedence over `+` and `-`, i.e.,

`x*y +  u/v` means `(x*y) + (u/v)`

`*` and `/` have same precedence, carry out left to right:

`x/y/u*v` means `((x/y) / u) * v`

Similar for `+` and `-`

`x - y + z` means `(x - y) + z`

Many more rules (google for C++ operator precedence).

Easy to forget the details, so use parentheses unless it's obvious.

Boolean expressions and operators

Boolean expressions are either true or false, e.g.,

```
int n, m; n = 5; m = 3;
bool b = n < m;             // value of b is false
```

C++ notation for boolean expressions:

`greater than`	`>`
`greater than or equals`	`>=`
`less than`	`<`
`less than or equals`	`<=`
`equals`	`==`
`not equals`	`!=`

not =

Can be combined with `&&` (“and”), `||` (“or”) and `!` (“not”), e.g.,

```
(n < m) && (n != 0)              (false)
(n%m >= 5) || !(n == m)          (true)
```

Precedence of operations not obvious; if in doubt use parentheses.

Shorthand assignment statements

full statement	shorthand equivalent
`n = n + m`	`n += m`
`n = n - m`	`n -= m`
`n = n * m`	`n *= m`
`n = n / m`	`n /= m`
`n = n % m`	`n %= m`

Special case of increment or decrement by one:

full statement	shorthand equivalent
`n = n + 1`	`n++` (or `++n`)
`n = n - 1`	`n--` (or `--n`)

`++` or `--` before variable means first increment (or decrement), then carry out other operations in the statement (more later).

Getting input from the keyboard

Sometimes we want to type in a value from the keyboard and assign this value to a variable. For this use the iostream object **`cin`**:

```
int age;
cout << "Enter your age" << endl;
cin >> age;
cout << "Your age is " << age << endl;
```

When you run the program you see

```
Enter your age
23                    ← you type this, then “Enter”
Your age is 23
```

(Why is there no “**`jin`**” in java? What were they thinking???)

if and else

Simple flow control is done with `if` and `else`:

```
if ( boolean test expression ){
   Statements executed if test expression true
}
```

or

```
if (expression1 ){
   Statements executed if expression1 true
}
else if ( expression2 ) {
   Statements executed if expression1 false
   and expression2 true
}
else {
   Statements executed if both expression1 and
   expression2 false
}
```

more on if and else

Note indentation and placement of curly braces:

```
if ( x > y ){
   x = 0.5*x;
}
```

Some people prefer

```
if ( x > y )
{
   x = 0.5*x;
}
```

If only a single statement is to be executed, you can omit the curly braces -- this is usually a bad idea:

```
if  ( x > y )  x = 0.5*x;
```

Putting it together -- `checkArea.cc`

```
#include <iostream>
using namespace std;
int main() {
  const double maxArea = 20.0;
  double width, height;
  cout << "Enter width" << endl;
  cin >> width;
  cout << "Enter height" << endl;
  cin >> height;
  double area = width*height;
  if ( area > maxArea ){
    cout << "Area too large" << endl;
  else {
    cout << "Dimensions are OK" << endl;
  }
  return 0;
}
```

“while” loops

A **`while`** loop allows a set of statements to be repeated as long as a particular condition is true:

```
while( boolean expression ){
  // statements to be executed as long as
  // boolean expression is true

}
```

For this to be useful, the boolean expression must be updated upon each pass through the loop:

```
while (x < xMax){
  x += y;
  ...
}
```

Possible that statements never executed, or that loop is infinite.

"do-while" loops

A `do-while` loop is similar to a `while` loop, but always executes at least once, then continues as long as the specified condition is true.

```
do {
  // statements to be executed first time
  // through loop and then as long as
  // boolean expression is true

} while ( boolean expression )
```

Can be useful if first pass needed to initialize the boolean expression.

“for” loops

A **`for`** loop allows a set of statements to be repeated a fixed number of times. The general form is:

```
 for ( initialization action ;
       boolean expression ; update action ){
   // statements to be executed

 }
```

Often this will take on the form:

```
for (int i=0; i<n; i++){
  // statements to be executed n times

}
```

Note that here **`i`** is defined only inside the **`{ }`**.

Examples of loops

A for loop:

```
int sum = 0;
for (int i = 1; i<=n; i++){
  sum += i;
}
cout << "sum of integers from 1 to " << n <<
     " is " << sum << endl;
```

A do-while loop:

```
int n;
bool gotValidInput = false;
do {
  cout << "Enter a positive integer" << endl;
  cin >> n;
  gotValidInput = n > 0;
} while ( !gotValidInput );
```

Nested loops

Loops (as well as if-else structures, etc.) can be nested, i.e., you can put one inside another:

```
// loop over pixels in an image

for (int row=1; row<=nRows; row++){
  for (int column=1; column<=nColumns; column++){
    int b = imageBrightness(row, column);
    ...

  }      // loop over columns ends here
}        // loop over rows ends here
```

We can put any kind of loop into any other kind, e.g., `while` loops inside `for` loops, vice versa, etc.

More control of loops

continue causes a single iteration of loop to be skipped (jumps back to start of loop).

break causes exit from entire loop (only innermost one if inside nested loops).

```
while ( processEvent ) {

  if ( eventSize > maxSize ) { continue; }

  if ( numEventsDone > maxEventsDone ) {
    break;
  }

//  rest of statements in loop ...

}
```

Usually best to avoid **continue** or **break** by use of **if** statements.