Computing and Statistical Data Analysis
Comp 2

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

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.

Type casting

Often we need to interpret the value of a variable of one type as being of a different type, e.g., we may want to carry out floating-point division using variables of type `int`.

Suppose we have: `int n, m; n = 5; m = 3;` and we want to know the real-valued ratio of `n/m` (i.e. not truncated). We need to "type cast" `n` and `m` from `int` to `double` (or `float`):

```
double x = static_cast<double>(n) /
           static_cast<double>(m);
```

will give `x = 1.666666...`

Will also work here with `static_cast<double>(n)/m;`
but `static_cast<double>(n/m);` gives `1.0`.

Similarly we can use `static_cast<int>(x)` to turn a `float` or `double` into an `int`, etc.

Digression #1: `bool` vs. `int`

C and earlier versions of C++ did not have the type **`bool`**. Instead, an **`int`** value of zero was interpreted as false, and any other value as true. This still works in C++:

```
int num = 1;
if ( num ) {
  ...              // condition true if num != 0
```

It is best to avoid this. If you want true or false, use **`bool`**. If you want to check whether a number is zero, then use the corresponding boolean expression:

```
if ( num != 0 ) {
  ...              // condition true if num != 0
```

Digression #2: value of an assignment and == vs. =

Recall = is the assignment operator, e.g., `x = 3;`

== is used in boolean expressions, e.g., `if ( x == 3 ) { ...`

In C++, an assignment statement has an associated value, equal to the value assigned to the left-hand side. We may see:

```
int x, y;
x = y = 0;
```

This says first assign `0` to `y`, then assign its value (`0`) to `x`.
This can lead to very confusing code. Or worse:

```
if ( x = 0 ) { ... // condition always false!
```

Here what the author probably meant was

```
if ( x == 0 ) { ...
```

Standard mathematical functions

Simple mathematical functions are available through the standard C library **`cmath`** (previously **`math.h`**), including:

```
abs   acos  asin  atan  atan2 cos   cosh  exp
fabs  fmod  log   log10 pow   sin   sinh  sqrt
tan   tanh
```

Most of these can be used with **`float`** or **`double`** arguments; return value is then of same type.

Raising to a power, $z = x^y$, with **`z = pow(x,y)`** involves log and exponentiation operations; not very efficient for **`z = 2, 3`**, etc. Some advocate e.g. **`double xSquared = x*x;`**

To use these functions we need: **`#include <cmath>`**

Google for C++ cmath or see **`www.cplusplus.com`** for more info.

A simple example

Create file `testMath.cc` containing:

```
// Simple program to illustrate cmath library
#include <iostream>
#include <cmath>
using namespace std;
int main() {

  for (int i=1; i<=10; i++){
    double x = static_cast<double>(i);
    double y = sqrt(x);
    double z = pow(x, 1./3.);    // note decimal pts
    cout << x << "  " << y << "  " << z << endl;
  }

}
```

Note indentation and use of blank lines for clarity.

Running testMath

Compile and link: `g++ -o testMath testMath.cc`

Run the program: `./testMath`

```
1   1   1
2   1.41421   1.25992
3   1.73205   1.44225
4   2   1.5874
...
```

The numbers don't line up in neat columns -- more later.

Often it is useful to save output directly to a file. Unix allows us to redirect the output:

```
./testMath > outputFile.txt
```

Similarly, use `>>` to append file, `>!` to insist on overwriting.

These tricks work with any Unix commands, e.g., `ls, grep`, ...

Improved i/o: formatting tricks

Often it's convenient to control the formatting of numbers.

```
cout.setf(ios::fixed);
cout.precision(4);
```

will result in 4 places always to the right of the decimal point.

```
cout.setf(ios::scientific);
```

will give scientific notation, e.g., `3.4516e+05`. To undo this, use `cout.unsetf(ios::scientific);`

`cout.width(15)` will cause next item sent to `cout` to occupy 15 spaces, e.g.,

```
cout.width(5); cout << x;
cout.width(10); cout << y;
cout.width(10); cout << z << endl;
```

To use `cout.width` need `#include <iomanip>` .

More formatting: `printf` and `scanf`

Much of this can be done more easily with the C function **`printf`**:

```
printf ("formatting info" [, arguments]);
```

For example, for **`float`** or **`double x`** and **`int i`**:

```
printf("%f %d \n", x, i);
```

will give a decimal notation for **`x`** and integer for **`i`**.
`\n` does (almost) same as **`endl;`**

Suppose we want 8 spaces for **`x`**, 3 to the right of the decimal point, and 10 spaces for **`i`**:

```
printf("%8.3f %10d \n", x, i);
```

For more info google for printf examples, etc.

Also **`scanf`**, analogue of **`cin`**.

To use **`printf`** need **`#include <cstdlib>`** .

Scope basics

The scope of a variable is that region of the program in which it can be used.

If a block of code is enclosed in braces `{ }`, then this delimits the scope for variables declared inside the braces. This includes braces used for loops and if structures:

```
int x = 5;
for (int i=0; i<n; i++){
  int y = i + 3;
  x = x + y;
}
cout << "x = " << x << endl;  // OK
cout << "y = " << y << endl;  // BUG -- y out of scope
cout << "i = " << i << endl;  // BUG -- i out of scope
```

Variables declared outside any function, including `main`, have 'global scope'. They can be used anywhere in the program.

More scope

The meaning of a variable can be redefined in a limited 'local scope':

```
int x = 5;
{
  double x = 3.7;
  cout << "x = " << x << endl;  // will print x = 3.7
}
cout << "x = " << x << endl;    // will print x = 5
```

(This is bad style; example is only to illustrate local scope.)

In general try to keep the scope of variables as local as possible. This minimizes the chance of clashes with other variables to which you might try to assign the same name.

Namespaces

A namespace is a unique set of names (identifiers of variables, functions, objects) and defines the context in which they are used.

E.g., variables declared outside of any function are in the global namespace (they have global scope); and can be used anywhere.

A namespace can be defined with the **`namespace`** keyword:

```
namespace aNameSpace {
  double x = 1.0;
}
```

To refer to this **`x`** in some other part of the program (outside of its local namespace), we can use

```
aNameSpace::x
```

`::` is the scope resolution operator.

The `std` namespace

C++ provides automatically a namespace called **`std`**.

It contains all identifiers used in the standard C++ library (lots!), including, e.g., **`cin`**, **`cout`**, **`endl`**, ...

To use, e.g., **`cout`**, **`endl`**, we can say:

```
using std::cout;
using std::endl;
int main(){
  cout << "Hello" << endl;
  ...
```

or we can omit **`using`** and say

```
int main(){
  std::cout << "Hello" << std::endl;
  ...
```

using namespace std;

Or we can simply say

```
using namespace std;
int main(){
  cout << "Hello" << endl;
  ...
```

Although I do this in the lecture notes to keep them compact, it is not a good idea in real code. The namespace **std** contains thousands of identifiers and you run the risk of a name clash.

This construction can also be used with user-defined namespaces:

```
using namespace aNameSpace;
int main(){
  cout << x << endl;          // uses aNameSpace::x
  ...
```

Functions

Up to now we have seen the function **main**, as well as mathematical functions such as **sqrt** and **cos**. We can also define other functions, e.g.,

```
const double PI = 3.14159265;      // global constant
double ellipseArea(double, double);  // prototype
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}

double ellipseArea(double a, double b){
  return PI*a*b;
}
```

The usefulness of functions

Now we can 'call' **`ellipseArea`** whenever we need the area of an ellipse; this is modular programming.

The user doesn't need to know about the internal workings of the function, only that it returns the right result.

'Procedural abstraction' means that the implementation details of a function are hidden in its definition, and needn't concern the user of the function.

A well written function can be re-used in other parts of the program and in other programs.

Functions allow large programs to be developed by teams (as is true for classes, which we will see soon).

Declaring functions

Before we can use a function, we need to declare it at the top of the file (before `int main()`).

```
double ellipseArea(double, double);
```

This is called the 'prototype' of the function. It begins with the function's 'return type'. The function can be used in an expression like a variable of this type.

The prototype must also specify the types of the arguments, in the correct sequence. Variable names are optional in the prototype.

The specification of the types and order of the arguments is called the function's signature.

Defining functions

The function must then be defined, i.e., we must say what it does with its arguments and what it returns.

```
double ellipseArea(double a, double b){
  return PI*a*b;
}
```

The first word defines the type of value returned, here **`double`**.

Then comes a list of parameters, each preceded by its type.

Note the scope of **`a`** and **`b`** is local to the function **`ellipseArea`**. We could have given them names different from the **`a`** and **`b`** in the main program (and we often do).

Then the body of the function does the necessary computation and finally we have the **`return`** statement followed by the corresponding value of the function.

Return type of a function

The prototype must also indicate the return type of the function, e.g., **`int, float, double, char, bool`**.

```
double ellipseArea(double, double);
```

The function's return statement must return a value of this type.

```
double ellipseArea(double a, double b){
  return PI*a*b;
}
```

When calling the function, it must be used in the same manner as an expression of the corresponding return type, e.g.,

```
double volume = ellipseArea(a, b) * height;
```

Return type `void`

The return type may be 'void', in which case there is no return statement in the function (like a FORTRAN subroutine):

```
void showProduct(double a, double b){
  cout << "a*b = " << a*b << endl;
}
```

To call a function with return type void, we simply write its name with any arguments followed by a semicolon:

```
showProduct(3, 7);
```

Putting functions in separate files

Often we put functions in a separate files. The declaration of a function goes in a 'header file' called, e.g., **ellipseArea.h**, which contains the prototype:

```
#ifndef ELLIPSE_AREA_H
#define ELLIPSE_AREA_H

// function to compute area of an ellipse

double ellipseArea(double, double);

#endif
```

The directives **#ifndef** (if not defined), etc., serve to ensure that the prototype is not included multiple times. If **ELLIPSE_AREA_H** is already defined, the declaration is skipped.

Putting functions in separate files, continued

Then the header file is included (note use of `" "` rather than `< >`) in all files where the function is called:

```
#include <iostream>
#include "ellipseArea.h"
using namespace std;
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}
```

(`ellipseArea.h` does not have to be included in the file `ellipseArea.cc` where the function is defined.)

Passing arguments by value

Consider a function that tries to change the value of an argument:

```
void tryToChangeArg(int x){
  x = 2*x;
}
```

It won't work:

```
int x = 1;
tryToChangeArg(x);
cout << "now x = " << x << endl;  // x still = 1
```

This is because the argument is passed 'by value'. Only a copy of the value of **x** is passed to the function.

In general this is a Good Thing. We don't want arguments of functions to have their values changed unexpectedly.

Sometimes, however, we want to return modified values of the arguments. But a function can only return a single value.

Passing arguments by reference

We can change the argument's value passing it 'by reference'.
To do this we include an `&` after the argument type in the function's prototype and in its definition (but no `&` in the function call):

```
void tryToChangeArg(int&);          // prototype

void tryToChangeArg(int& x){        // definition
  x = 2*x;
}

int main(){
  int x = 1;
  tryToChangeArg(x);
  cout << "now x = " << x << endl;  // now x = 2
}
```

Argument passed by reference must be a variable, e.g.,
`tryToChangeArg(7);` will not compile.

Variable scope inside functions

Recall that the definition of a function is enclosed in braces. Therefore all variables defined inside it are local to that function.

```
double pow(double x, int n){
  double y = static_cast<double>(n) * log(x);
  return exp(y);
}


...
double y = pow(3,2);  // this is a different y
```

The variable **y** in the definition of **pow** is local. We can use the same variable name outside this function with no effect on or from the variable **y** inside **pow**.

Inline functions

For very short functions, we can include the keyword **inline** in their definition (must be in same file, before calling program):

```
inline double pow(double x, int n){
  double y = static_cast<double>(n) * log(x);
  return exp(y);
}
```

The compiler will (maybe) replace all instances of the function by the code specified in the definition. This will run faster than ordinary functions but results in a larger program.

Only use make very short functions inline and then only when speed is a concern, and then only when you've determined that the function is using a significant amount of time.

Default arguments

Sometimes it is convenient to specify default arguments for functions in their declaration:

```
double line(double x, double slope=1, double offset=0);
```

The function is then defined as usual:

```
double line(double x, double slope, double offset){
  return x*slope  +  offset;
}
```

We can then call the function with or without the defaults:

```
y = line (x, 3.7, 5.2);  // here slope=3.7, offset=5.2
y = line (x, 3.7);       // uses offset=0;
y = line (x);            // uses slope=1, offset=0
```

Function overloading

We can define versions of a function with different numbers or types of arguments (signatures). This is called function overloading:

```
double cube(double);
double cube (double x){
  return x*x*x;
}

double cube(float);
double cube (float x){
  double xd = static_cast<double>(x);
  return xd*xd*xd;
}
```

Return type can be same or different; argument list must differ in number of arguments or in their types.

Function overloading, cont.

When we call the function, the compiler looks at the signature of the arguments passed and figures out which version to use:

```
float x;
double y;
double z = cube(x);  // calls cube(float) version
double z = cube(y);  // calls cube(double) version
```

This is done e.g. in the standard math library **`cmath`**. There is a version of **`sqrt`** that takes a **`float`** (and returns **`float`**), and another that takes a **`double`** (and returns **`double`**).

Note it is not sufficient if functions differ only by return type -- they must differ in their argument list to be overloaded.

Operators (**`+`**, **`-`**, etc.) can also be overloaded. More later.