Computing and Statistical Data Analysis
Comp 3

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

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.

Writing to and reading from files

Here is a simple program that opens an existing file in order to read data from it:

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(){
  // create an ifstream object (name arbitrary)...
  ifstream myInput;
  // Now open an existing file...
  myInput.open("myDataFile.txt");
  // check that operation worked...
  if ( myInput.fail() ) {
    cout << "Sorry, couldn't open file" << endl;
    exit(1);      // from cstdlib
  }
  ...
```

Reading from an input stream

The input file stream object is analogous to **`cin`**, but instead of getting data from the keyboard it reads from a file. Note use of "dot" to call the ifstream's "member functions", **`open`**, **`fail`**, etc.

Suppose the file contains columns of numbers like

```
    1.0    7.38   0.43
    2.0    8.59   0.52
    3.0    9.01   0.55
    ...
```

We can read in these numbers from the file:

```
double x, y, z;
for(int i=1; i<=numLines; i++){
  myInput >> x >> y >> z;
  cout << "Read " << x << " " << y << " " << z << endl;
}
```

This loop requires that we know the number of lines in the file.

Reading to the end of the file

Often we don't know the number of lines in a file ahead of time. We can use the "end of file" (**`eof`**) function:

```
double x, y, z;
int line = 0;
while ( !myInput.eof() ){
  myInput >> x >> y >> z;
  if ( !myInput.eof() ) {
    line++;
    cout << x << " " << y << " " << z << endl;
  }
}
cout << lines << " lines read from file" << endl;
...
myInput.close();     // close when finished
```

Note some gymnastics needed to avoid getting last line twice.

Writing data to a file

We can write to a file with an `ofstream` object:

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(){
  // create an ofstream object (name arbitrary)...
  ofstream myOutput;
  // Now open a new file...
  myOutput.open("myDataFile.txt");
  // check that operation worked...
  if ( myOutput.fail() ) {
    cout << "Sorry, couldn't open file" << endl;
    exit(1);      // from cstdlib
  }
  ...
```

Writing data to a file, cont.

Now the **`ofstream`** object behaves like **`cout`**:

```
for (int i=1; i<=n; i++){
  myOutput << i << "\t" << i*i << endl;
}
```

Note use of tab character **`\t`** for formatting (could also use e.g. `"    "` or)

Alternatively use the functions **`setf`**, **`precision`**, **`width`**, etc. These work the same way with an **`ofstream`** object as they do with **`cout`**, e.g.,

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

File access modes

The previous program would overwrite an existing file.
To append an existing file, we can specify:

```
myOutput.open("myDataFile.txt", ios::app);
```

This is an example of a file access mode. Another useful one is:

```
myOutput.open("myDataFile.txt", ios::bin);
```

The data is then written as binary, not formatted. This is much more compact, but we can't check the values with an editor.

For more than one option, separate with vertical bar:

```
myOutput.open("myDataFile.txt", ios::bin | ios::app);
```

Many options, also for **`ifstream`**. Google for details.

Putting it together

Now let’s put together some of what we’ve just seen. The program reads from a file a series of exam scores, computes the average and writes it to another file. In file **examAve.cc** we have

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "aveScore.h"
using namespace std;
int main(){
  // open input file
  ifstream inFile;
  inFile.open("studentScores.txt");
  if ( inFile.fail() ) {
    cerr << "Couldn’t open input file" << endl;
    exit(1);
  }
              ...
```

examAve, continued

```
// open the output file
ofstream outFile;
outFile.open("averageScores.txt");
if ( outFile.fail() ) {
  cerr << "Couldn't open output file" << endl;
  exit(1);
}

while ( !inFile.eof() ){
  int studentNum;
  double test1, test2, test3;
  inFile >> studentNum >> test1 >> test2 >> test3;
  if( !inFile.eof() ){
    double ave = aveScore (test1, test2, test3);
    outFile << studentNum << "\t" << ave << endl;
  }
}
```

More examAve

```
  // close up
  inFile.close();
  outFile.close();
  return 0;
}
```

Now the file aveScore.cc contains

```
double aveScore(double a, double b, double c){
  double ave = (a + b + c)/3.0;
  return ave;
}
```

More examAve and aveScore

The header file `aveScore.h` contains

```
#ifndef AVE_SCORE_H
#define AVE_SCORE_H
double aveScore(double, double, double);
#endif AVE_SCORE_H
```

We compile and link the program with

```
g++ -o examAve examAve.cc aveScore.cc
```

The input data file `studentScores.txt` might contain

```
1     73    65    68
2     52    45    44
3     83    85    91
```

etc. The example is trivial but we can generalize this to very complex programs.

Arrays

An array is a fixed-length list containing variables of the same type.

Declaring an array: ***`data-type variableName`***`[`***`numElements`***`];`

```
int score[10];
double energy[50], momentum[50];
const int MaxParticles = 100;
double ionizationRate[MaxParticles];
```

The number in brackets `[]` gives the total number of elements,e.g. the array `score` above has 10 elements, numbered 0 through 9. The individual elements are referred to as

```
score[0], score[1], score[2], ..., score[9]
```

The index of an array can be any integer expression with a value from zero up to the number of elements minus 1. If you try to access `score[10]` this is an error!

Arrays, continued

Array elements can be initialized with assignment statements and otherwise manipulated in expressions like normal variables:

```
const int NumYears = 50;
int year[NumYears];
for(int i=0; i<NumYears; i++){
  year[i] = i + 1960;
}
```

Note that C++ arrays always begin with zero, and the last element has an index equal to the number of elements minus one.

This makes it awkward to implement, e.g., n-dimensional vectors that are naturally numbered $\boldsymbol{x} = (x_1, \ldots, x_n)$.

In the C++ 98 standard, the size of the array must be known at compile time. In C99 (implemented by gcc), array length can be variable (set at run time). See also “dynamic” arrays (later).

Multidimensional arrays

An array can also have two or more indices. A two-dimensional array is often used to store the values of a matrix:

```
const int numRows = 2;
const int numColumns = 3;
double matrix[numRows][numColumns];
```

Again, notice that the array size is 2 by 3, but the row index runs from 0 to 1 and the column index from 0 to 2.

The elements are stored in memory in the order:

```
matrix[i][j], matrix[i][j+1], etc.
```

Usually we don’t need to know how the data are stored internally. (Ancient history: in FORTRAN, the left-most index gave adjacent elements in memory.)

Initializing arrays

We can initialize an array together with the declaration:

```
int myArray[5] = {2, 4, 6, 8, 10};
```

Similar for multi-dimensional arrays:

```
double matrix[numRows][numColumns] =
                    { {3, 7, 2}, {2, 5, 4} };
```

In practice we will usually initialize arrays with assignment statements.

Example: multiplication of matrix and vector

```
// Initialize vector x and matrix A
const int n = 5;
double x[n];
double A[n][n];
for (int i=0; i<n; i++){
  x[i] = someFunction(i);
  for (int j=0; j<n; j++){
    A[i][j] = anotherFunction(i, j);
  }
}

// Now find y = Ax
double y[n];
for (int i=0; i<n; i++){
  y[i] = 0.0;
  for (int j=0; j<n; j++){
    y[i] += A[i][j] * x[j];
  }
}
```

Passing arrays to functions

Suppose we want to use an array `a` of length `len` as an argument of a function. In the function's declaration we say, e.g.,

```
double sumElements(double a[], int len);
```

We don't need to specify the number of elements in the prototype, but we often pass the length into the function as an `int` variable.

Then in the function definition we have, e.g.,

```
double sumElements(double a[], int len){
  double sum = 0.0;
  for (int i=0; i<len; i++){
    sum += a[i];
  }
  return sum;
}
```

Passing arrays to functions, cont.

Then to call the function we say, e.g.,

```
double s = sumElements(myMatrix, itsLength);
```

Note there are no brackets for **`myMatrix`** when we pass it to the function.

You could, however, pass **`myMatrix[i]`**, not as a matrix but as a **`double`**, i.e., the *i*th element of **`myMatrix`**. For example,

```
double x = sqrt(myMatrix[i]);
```

Passing arrays to functions

When we pass an array to a function, it works as if passed by reference, even though we do not use the `&` notation as with non-array variables. (The array name is a "pointer" to the first array element. More on pointers later.)

This means that the array elements could wind up getting their values changed:

```
void changeArray (double a[], int len){
  for(int i=0; i<len; i++){
    a[i] *= 2.0;
  }
}

int main(){
  ...
  changeArray(a, len);    // elements of a doubled
```

Passing multidimensional arrays to functions

When passing a multidimensional array to a function, we need to specify in the prototype and function definition the number of elements for all but the left-most index:

```
  void processImage(int image[][numColumns],
                    int numRows, int numColumns){
     ...
```

(But we still probably need to pass the number of elements for both indices since their values are needed inside the function.)

Pointers

A pointer variable contains a memory address. It ‘points’ to a location in memory. To declare a pointer, use a star, e.g.,

```
int* iPtr;
double * xPtr;
char *c;
float *x, *y;
```

Note some freedom in where to put the star. I prefer the first notation as it emphasizes that `iPtr` is of type “pointer to `int`”.

(But in `int* iPtr, jPtr;` only `iPtr` is a pointer--need 2 stars.)

Name of pointer variable can be any valid identifier, but often useful to choose name to show it’s a pointer (suffix `Ptr`, etc.).

Pointers: the `&` operator

Suppose we have a variable `i` of type `int`:

```
int i = 3;
```

We can define a pointer variable to point to the memory location that contains `i`:

```
int* iPtr = &i;
```

Here `&` means "address of". Don't confuse it with the `&` used when passing arguments by reference.

Initializing pointers

A statement like

```
    int* iPtr;
```

declares a pointer variable, but does not initialize it. It will be pointing to some “random” location in memory. We need to set its value so that it points to a location we’re interested in, e.g., where we have stored a variable:

```
    iPtr = &i;
```

(just as ordinary variables must be initialized before use).

Dereferencing pointers: the `*` operator

Similarly we can use a pointer to access the value of the variable stored at that memory location. E.g. suppose **`iPtr = &i;`** then

```
int iCopy = *iPtr;    // now iCopy equals i
```

This is called 'dereferencing' the pointer. The * operator means "value stored in memory location being pointed to".

If we set a pointer equal to zero (or **`NULL`**) it points to nothing. (The address zero is reserved for null pointers.)

If we try to dereference a null pointer we get an error.

Why different kinds of pointers?

Suppose we declare

```
int* iPtr;        //  type "pointer to int"
float* fPtr;      //  type "pointer to float"
double* dPtr;     //  type "pointer to double"
```

We need different types of pointers because in general, the different data types (**`int, float, double`**) take up different amounts of memory. If declare another pointer and set

```
int* jPtr = iPtr + 1;
```

then the **`+1`** means "plus one unit of memory address for **`int`**", i.e., if we had **`int`** variables stored contiguously, **`jPtr`** would point to the one just after **`iPtr`**.

But the types **`float`**, **`double`**, etc., take up different amounts of memory, so the actual memory address increment is different.

Passing pointers as arguments

When a pointer is passed as an argument, it divulges an address to the called function, so the function can change the value stored at that address:

```
void passPointer(int* iPtr){
  *iPtr += 2;          // note *iPtr on left!
}

...
int i = 3;
int* iPtr = &i;
passPointer(iPtr);
cout << "i = " << i << endl;    // prints i = 5
passPointer(&i);              // equivalent to above
cout << "i = " << i << endl;    // prints i = 7
```

End result same as pass-by-reference, syntax different. (Usually pass by reference is the preferred technique.)

Pointers vs. reference variables

A reference variable behaves like an alias for a regular variable. To declare, place `&` after the type:

```
int i = 3;
int& j = i;          // j is a reference variable
j = 7;
cout << "i = " << i << endl;  // prints i = 7
```

Passing a reference variable to a function is the same as passing a normal variable by reference.

```
void passReference(int& i){
  i += 2;
}

passReference(j);
cout << "i = " << i << endl;  // prints i = 9
```

What to do with pointers

You can do lots of things with pointers in C++, many of which result in confusing code and hard-to-find bugs.

One of the main differences between Java and C++: Java doesn't have pointer variables (generally seen as a Good Thing).

One interesting use of pointers is that the name of an array is a pointer to the zeroth element in the array, e.g.,

```
double a[3] = {5, 7, 9};
double zerothVal = *a;      // has value of a[0]
```

The main usefulness of pointers for us is that they will allow us to allocate memory (create variables) dynamically, i.e., at run time, rather than at compile time.

Strings (the old way)

A string is a sequence of characters. In C and in earlier versions of C++, this was implemented with an array of variables of type **`char`**, ending with the character **`\0`** (counts as a single 'null' character):

```
char aString[] = "hello";  // inserts \0 at end
```

The **`cstring`** library (**`#include <cstring>`**) provides functions to copy strings, concatenate them, find substrings, etc. E.g.

```
char* strcpy(char* target, const char* source);
```

takes as input a string **`source`** and sets the value of a string **`target`**, equal to it. Note **`source`** is passed as **`const`** -- it can't be changed.

You will see plenty of code with old "C-style" strings, but there is now a better way: the **`string`** class (more on this later).

Example with **`strcpy`**

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
  char string1[] = "hello";
  char string2[50];
  strcpy(string2, string1);
  cout << "string2:  " << string2 << endl;
  return 0;
}
```

No need to count elements when initializing string with `"   "`.

Also `\0` is automatically inserted as last character.

Program will print: `string2 = hello`