SUPA Graduate C++ Course
Lecture 2

Sarah Boutle
University of Glasgow

Many thanks to Adrian Buzatu, S. Allwood-Spiers

News

- **First assignment uploaded to course page**
  - Due: 31st October before the 2nd lab session
  - Please upload your source files to course page

- **Labs:** STARTING NEXT WEEK
  - Monday afternoons, 2-5pm starting 24th October
  - Room 220a, Kelvin Building, University of Glasgow
  - Can be used to get help for the assignments, but not compulsory
  - Please indicate if you intend to come in person :
    http://doodle.com/poll/8n84fuqkt2bvzqtu
  - Probably most useful to you to bring your own laptop, but there are (a limited number of) linux machines available

- **Next assignment will be available on Monday**
  - Watch out for the SUPA mail

Last Week's Lecture: Basic Elements

- Introduction
  - Foreword
  - Programming Methodology
- Basic C/C++ Syntax
  - Simple types and operators
  - Conditional Statements
  - Loops
  - Functions
  - Header files
  - File input

This Week's Lecture

- More basic elements
  - Writing to a file
  - Formatting numbers
  - Arrays
  - Pointers

- Introduction to Classes
  - Simple example
  - Objects
  - Members
  - Constructors

Writing to a file

We can write to a file with an ofstream object (remember ifstream from last week)

First open the file:

```
#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
   }
  ...
```

Write to output stream

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

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

`\t` is the tab character, could also use " "

Alternatively use the functions **`setf`**, **`precision`**, **`width`**, etc.

These work the same way with an **`ofstream`** object or **`cout`**,

see next slide....

Don't forget to close the file:

```
myOutput.close();
```

Formatting tricks

Often it is convenient to control the formatting of numbers

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

results in 4 places to the right of the decimal point.

To use scientific notation (e.g. `3.4516e+05`):

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

And to undo this: `cout.unsetf(ios::scientific);`

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

`cout.width(5);` causes next item sent to cout to occupy 5 space

To use this, need `#include <iomanip>`

Arrays

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

```
int score[10];
double energy[50], momentum[50];
const int MaxStudents = 100;
double examMarks[MaxStudents];
```

number of elements in array

Refer to the individual elements with:

```
score[0], score[1], score[2], score[3],......score[9]
```

index from zero to # of elements minus 1, **accessing score[10] gives an error!**

An array can also have two or more indices:

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

Arrays

Declaring an array causes memory to be assigned but does not zero elements.

Elements can be initialized with assign statements:

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

Elements can also be initialized with the declaration:

```
int myArray[5] = {2, 4, 6, 8, 10};
double matrix[numRows][numColumns] = { {3, 7, 2}, {2, 5, 4} };
```

Passing arrays to functions

Suppose we want to use an array `a` of length `len` as an argument of a function

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

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

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

Then to call the function:

```
double s = sumElements(myArray, arrayLength);
```

no brackets

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 in a moment)

So array elements can end 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
```

Pointers; the & operator

A pointer variable contains a memory address "pointing" to a location in memory

Declare a pointer with a star:

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

flexibility on where to put the star

here only iPtr is a pointer

Unlike a variable declaration pointer declarations do not cause memory to be assigned

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:

```
int i =3;
iPtr = &i;
```

here & means "address of", don't confuse with & used when passing by reference

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 the memory location being pointed to"

If we set a pointer 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 will get an error

Why different kinds of pointers?

```
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.)

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

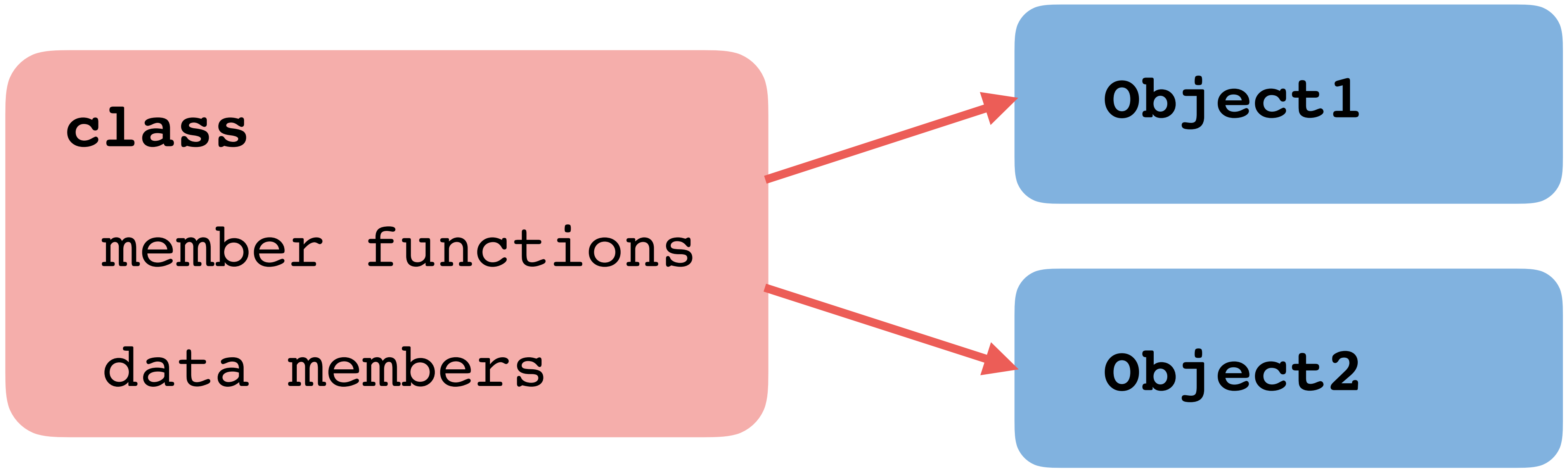
```
double a[3] = {5, 7, 9};
double zerothVal = *a; // has value of a[0]
```

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

Objects: an introduction

A class is the building block of Object Oriented programming

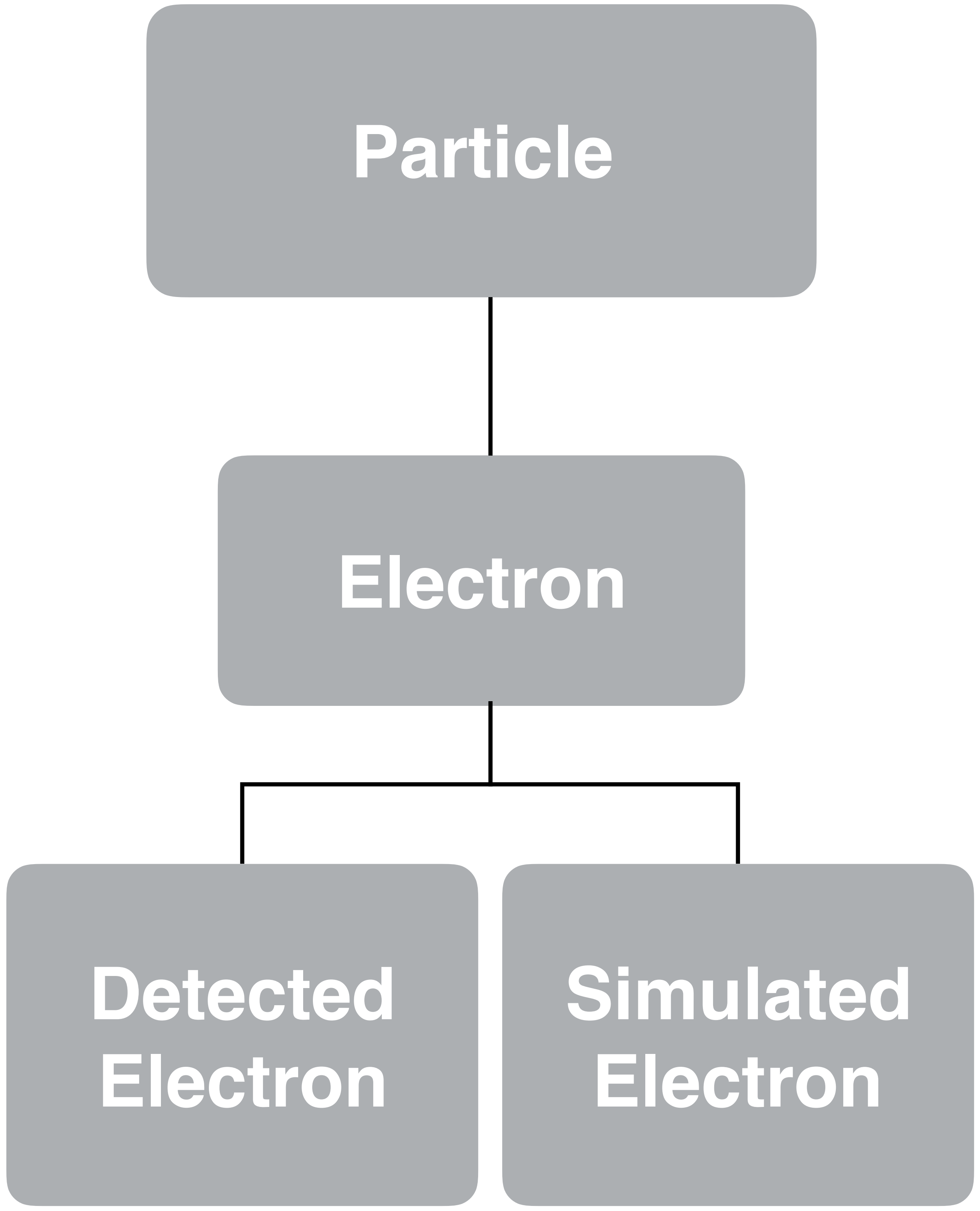
- a class is like a user-defined data type
- an **object** is an **instance** of a **class**



An example: from particle physics

Particle	Has momentum (px, py, pz), energy (E) Work in terms of 4-vectors From these we calculate: - mass - transverse momentum
Electron	Has all of the above + charge and an ID code
Detected Electron	Has all of the above + information about the signal in the detector
Simulated Electron	Has all of the above + information about the decay it originates from

Designing a program with Object Orientation



Build up complexity using class building blocks.

- Create more general base classes
- Use inheritance to build on existing functionality.
- Could use objects from any part of the inheritance tree within a program.

Class declaration: in header file

```
class TwoVector {

public:

  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:

  double m_x;
  double m_y;

};
```

Defines a class to represent a two-dimensional vector

Put this in `TwoVector.h`

Put definitions of functions in `TwoVector.cc`

Member functions

Data members

Class declaration: protection labels

```
class TwoVector {

public:

  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:

  double m_x;
  double m_y;

};
```

public methods/member functions

- generally accessible: can be accesssed through any object of a class
- constructors and accessors (to set or get values)

private methods and data members/ attributes

- accessible only from within the class
- useful for hiding code implementation from users

Class declaration: naming conventions

```
class TwoVector {

public:

  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:

  double m_x;
  double m_y;

};
```

Class names:

start with Capital letter

Member functions:

lowerCamelCase

private data members

start with m_

Class declaration: Constructors

```
class TwoVector {

public:

  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:

  double m_x;
  double m_y;

};
```

First two members of the class here, are the "constructors"

- Always has same member as the class
- This is the function that is called when an object is created
- Constructor has no return type
- There can be different constructors with different signatures (type and number of arguments)

Class declaration: Constructors

```
class TwoVector {

public:

  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:

  double m_x;
  double m_y;

};
```

So an object can be instantiated by

```
TwoVector myTwoVec;
```

or

```
TwoVector myTwoVec(myX, myY);
```

the appropriate constructor is called automatically

When the constructor is invoked:

- Memory is allocated for the object
- Members are initialized
- Body of constructor is executed

Defining the constructor

Define the constructors in `TwoVector.cc`:

```
TwoVector::TwoVector() {
 m_x = 0;
 m_y = 0;
}
TwoVector::TwoVector(double x, double y){
 m_x = x;
 m_y = y;
}
```

Precede each function name with the class name and ::

(the scope resolution operator)

the constructor serves to initialize the object, remember:

```
TwoVector myFirstTwoVec;
TwoVector mySecondTwoVec(myX, myY);
TwoVector myThirdTwoVec = myFirstTwoVec;
```

"copy constructor" automatically provided

Class declaration: in header file

```
class TwoVector {

public:

  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:

  double m_x;
  double m_y;

};
```

REMINDER

Member functions

Data members

Member functions and data hiding

We call an object's member functions like this:

```
TwoVector v(1.5, 3.7);          // creates an object v
double vX = v.x();
cout << "vX = " << vX << endl; // prints vX = 1.5
...
```

If the class had public data members, i.e. **`m_x`** was **`public`**, we could do:

```
double vX = x.m_x;
```

We usually keep data members private and allow the user to access the data through the public member functions. This is called "data hiding"

So if, for example, we changed the internal representation to polar coordinates, we would rewrite the function definition of x()... but the user would not see any change.

Defining member functions

Put the member function defintion in **`TwoVector.cc`**:

```
double TwoVector::x() { return m_x; }
double TwoVector::y() { return m_y; }

double TwoVector::r() {
 return sqrt(m_x*m_x + m_y*m_y);
}

double TwoVector::theta() {
 return atan2(m_y, m_x); // from cmath
}
...
```

These are called accessor or getter functions.

Sometimes useful to name them **`getTheta()`**......

They access the data of an object but do now change the internal state of the object.

Note here, the argument list is empty.

More member functions

Also in **TwoVector.cc** we have the following definitions:

```
void TwoVector::setX(double x) { m_x = x; }
void TwoVector::setY(double y) { m_y = y; }

void TwoVector::setR(double r) {
 double cosTheta = m_x / this->r();
 double sinTheta = m_y / this->r();
 m_x = r * cosTheta;
 m_y = r * sinTheta;
}
```

inside each object's member functions, C++ automatically provides a pointer called "this"

(use here is optional)

These are called setter functions.

They are allowed to manipulate the **private** data members as they belong to the class.

Use in the same way:

```
TwoVector v(1.5, 3.7);
v.setX(2.9);
```

changes v's value of m_x to 2.9

Class example so far

TwoVector.h

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {

public:
  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:
  double m_x;
  double m_y;
};
#endif
```

TwoVector.cc

```
#include "TwoVector.h"
#include <cmath>
#include <iostream>

TwoVector::TwoVector() {
  m_x = 0;
  m_y = 0;
}
TwoVector::TwoVector(double x,
double y){
  m_x = x;
  m_y = y;
}
```

Class example so far

TwoVector.h

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {

public:
  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  double r();
  double theta();
  void setX(double x);
  void setY(double y);
  void setR(double r);
  void setTheta(double theta);

private:
  double m_x;
  double m_y;
};
#endif
```

TwoVector.cc

```
double TwoVector::x() { return m_x;}
double TwoVector::y() { return m_y;}
double TwoVector::r() {
  return sqrt(m_x*m_x + m_y*m_y);
}
double TwoVector::theta() {
  return atan2(m_y, m_x);
}

void TwoVector::setX(double x) {
  m_x = x;
}
void TwoVector::setY(double y) {
  m_y = y;
}
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

Class example so far

main.cc

```
#include <iostream>
#include "TwoVector.h"

using namespace std;

int main() {

  TwoVector v(1.5, 3.7);
  TwoVector u(1.4, 8.8);
  v.setX(2.9);
  u.setR(1.3);

  cout << "Vector 1, R=" << v.r() << ", Theta=" << v.theta() << endl;
  cout << "Vector 2, R=" << u.r() << ", Theta=" << u.theta() << endl;

  return 0;
};
```

Final steps

Compile

```
g++ -o myProgram main.cc TwoVector.cc
```

Run:

```
./myProgram
```

Soon we'll look at Makefiles.....

Review

- More basic elements
  - Formatting numbers
  - Writing to a file
  - Arrays
  - Pointers

- Introduction to Classes
  - Simple example
  - Objects
  - Members
  - Constructors

News

- **First assignment uploaded to course page**
  - Due: 31st October before the 2nd lab session
  - Please upload your source files to course page

- **Labs:** STARTING NEXT WEEK
  - Monday afternoons, 2-5pm starting 24th October
  - Room 220a, Kelvin Building, University of Glasgow
  - Can be used to get help for the assignments, but not compulsory
  - Please indicate if you intend to come in person : http://doodle.com/poll/8n84fuqkt2bvzqtu
  - Probably most useful to you to bring your own laptop, but there are (a limited number of) linux machines available

- **Next assignment will be available on Monday**
  - Watch out for the SUPA mail