# SUPA Graduate C++ Course
# Lecture 4

Sarah Boutle
University of Glasgow

Many thanks to Adrian Buzatu, S. Allwood-Spiers

# News

- **Assignment 2:**
  - Due: 7th November before the 3rd lab session

- **Assignment 3:**
  - Due: 14th November before the last lab session

- **Labs:**
  - Monday afternoons, 2-5pm
  - Room 220a, Kelvin Building, University of Glasgow
  - Please indicate if you intend to come in person **PLEASE UPDATE**
    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

- **Assignment 4 will be available on Monday**
  - Watch out for the SUPA mail
  - Due 28th November

# Last Week's Lecture

- More classes
  - Constructing objects
  - Memory allocation
  - delete
  - Operator overloading
  - Static members
  - Destructors
  - Copy Constructors

# This Week's Lecture

- More classes
  - Class templates
  - Inheritance
  - Polymorphism
- Makefiles

# Class example recap

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 templates

We defined TwoVector using doubles, what if we wanted to use float..

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {

public:
 TwoVector();
 TwoVector(double x, double y);
 double x();
 TwoVector(float x, float y);
 float x();
 .
 .
 .
```

We could cut and paste to create additional TwoVector class based on the new type

Very bad idea in terms of code maintenance..

Better solution: class templates

# Templates recap

Templates allow code re-use where the same functionality is needed to operate on many different classes or types. Can write class and function templates

Recall, from assignment 1:

```
template<typename T>
T calculate_absolute_value(T x, T y){
  return sqrt(x*x+y*y);
}

template<typename T>
T calculate_absolute_value(T x, T y, T z){
  return sqrt(x*x+y*y+z*z);
}

int main(int argc, char* argv[]){
...
   T absolute_value_2D = calculate_absolute_value<T>(x,y);
   T absolute_value_3D = calculate_absolute_value<T>(x,y,z);
```

# Class templates

Class definition....

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {

public:
  TwoVector();
  TwoVector(double x, double y);
  double x();
  double y();
  void setX(double x);
  void setY(double y);

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

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

template <class T>
class TwoVector {

public:
  TwoVector();
  TwoVector(T, T);
  T x();
  T y();
  void setX(T);
  void setY(T);

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

# Class templates

With templates, class declaration must be in same file as function definitions (put everything in TwoVector.h).

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

template <class T>
class TwoVector {

public:
  TwoVector();
  TwoVector(T, T);
  T x();
  T y();
  void setX(T);
  void setY(T);

private:
  T m_x;
  T m_y;
};
```

```
template <class T>
TwoVector<T>::TwoVector(T x, T y){
  m_x = x;
  m_y = y;
  m_counter++;
}

template <class T>
T TwoVector<T>::x(){ return m_x; }
template <class T>
T TwoVector<T>::y(){ return m_y; }

template <class T>
void TwoVector<T>::setX(T x){
  m_x = x;
}
template <class T>
void TwoVector<T>::setY(T y){
  m_y = y;
}
#endif
```

# Using class templates

To use a class template, insert the desired argument:

```
TwoVector<double> dVec; // creates double version
TwoVector<float> fVec;  // creates float version
```

**`TwoVector`** is no longer a class, it's only a template for classes.

**`TwoVector<double>`** and **`TwoVector<float>`** are classes (sometimes called "template classes", since they were made from class templates).

Class templates are particularly useful for container classes, such as vectors, remember Standard Template Library (STL) from last week.

# Inheritance

Often we define a class with is similar to an existing one

For example, imagine we already had an **`Animal`** class

```
class Animal {

public:
  double weight();
  double age();
  ...

private:
  double m_weight;
  double m_age;
  ...
};
```

# Inheritance

Often we define a class with is similar to an existing one

For example, imagine we already had an **`Animal`** class

And we have some objects which are dogs. The have some/many of the features of **`Animal`** and some extra ones....

```
class Animal {

public:
  double weight();
  double age();
  ...

private:
  double m_weight;
  double m_age;
  ...
};
```

```
class Dog {

public:
  double weight();
  double age();
  bool hasFleas();
  void bark();

private:
  double m_weight;
  double m_age;
  bool m_hasFleas;
};
```

# Inheritance

Rather than define a separate **`Dog`** class like this, we can derive it from **`Animal`**

**`Animal.h`**

```
class Animal {

public:
  double weight();
  double age();
  ...

private:
  double m_weight;
  double m_age;
  ...
};
```

**`Dog.h`**

```
#include "Animal.h"
class Dog : public Animal {

public:
  bool hasFleas();
  void bark();

private:
  bool m_hasFleas;
};
```

Animal is the "base class", Dog is the "derived class"

Dog inherits all of the members of Animal....

# Inheritance

Rather than define a separate **`Dog`** class like this, we can derive it from **`Animal`**

**Animal.h**

```
class Animal {

public:
 double weight();
 double age();
 ...

private:
 double m_weight;
 double m_age;
 ...
};
```

**Dog.h**

```
#include "Animal.h"
class Dog : public Animal {

public:
 bool hasFleas();
 void bark();

private:
 bool m_hasFleas;
};
```

A **`Dog`** *is an* **`Animal`**, **`Dog`** *inherits* from **`Animal`** and also has **`hasFleas`**

# Inheritance

However, the private members are inaccessible to to member functions in `Dog`

Fix this using new protection label: `protected`

This gives the derived classes access to those data members but keeps them inaccessible to the uses of the class.

`Animal.h`

```
class Animal {

public:
  double weight();
  double age();
  ...

protected:
  double m_weight;
  double m_age;
  ...
};
```

`Dog.h`

```
#include "Animal.h"
class Dog : public Animal {

public:
  bool hasFleas();
  void bark();

private:
  bool m_hasFleas;
};
```

# Polymorphism

We've already seen one type of polymorphism: function overloading

We might want to redefine a function of the base class to do or mean something different in the derived class

This is called overriding

For example, we might want **`age()`** to return normal years for **`Animal`** but human-equivalent years for **`Dog`**

Then the function takes on different forms, depending on the type of object calling it

- this is an example of polymorphism

Takes advantage of a key feature of class inheritance:

- a pointer to a derived class is type-compatible with a pointer to its base class

# Polymorphism: example

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }


};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;





}
```

Declare two pointers to `Polygon` and assign them the addresses of a `Rectangle` and `Triangle` object respectively

Valid since `Rectangle` and `Triangle` are classes derived from `Polygon`

# Polymorphism: example

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }


};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  rect.set_values (4,5);


}
```

Through the pointer to **`Polygon`**, one can access members inherited from **`Polygon`**

But they can also be accessed through the object itself

# Polymorphism: example

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }


};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  rect.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
}
```

But the members of the derived class cannot be accessed like that

Only access through the objects of the derived class themselves

# Polymorphism: example

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }
     int area(){
      return 0;}
};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);

  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl
}
```

Can make **`area()`** a member of the base class

Bu this won't give the desired behaviour:

the call of the function **`area()`** is being set once by the compiler as the version defined in the base class

# Polymorphism: example

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }
     virtual int area(){
      return 0;}
};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);

  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl
}
```

Simple modification: virtual

This time, the compiler looks at the contents of the pointer instead of it's type.

So the versions of **`area()`** given in **`Rectangle`** and **`Triangle`** are called instead

# Polymorphism: example

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }
     virtual int area(){
      return 0;}
};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);

  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;

  Polygon poly;
  Polygon * ppoly3 = &poly;
  cout << ppoly3->area() << endl;
}
```

Note: here `Polygon` is still a normal class..

we instantiate an object and access it's own definition of it's member function area (returns 0).

# Polymorphism: example

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }
    virtual int area() = 0;

};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

Base classes are allowed to have virtual member functions without definition

Syntax is to replace their definition by =0 (an equal sign and a zero)

This type of function is called a pure virtual function

Classes that contain at least one pure virtual function are called abstract base classes

Abstract base classes cannot be used to instantiate objects.

```
Polygon mypolygon;
Polygon *ppoly1
```

Not OK
OK

# Polymorphism: example

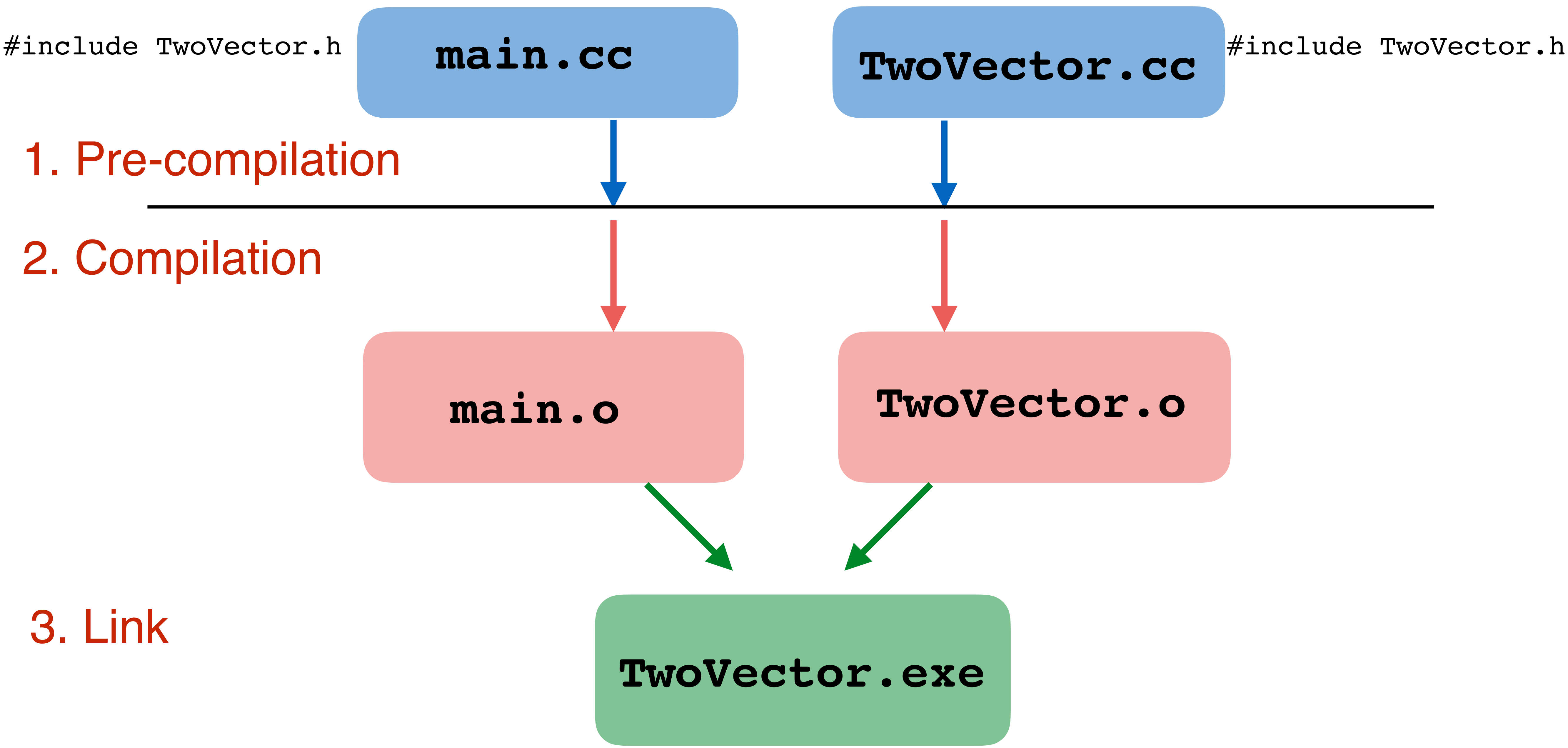
```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
      width=a; height=b; }
     virtual int area() = 0;
     void printarea()
      cout << this->area() << endl;}
};

class Rectangle: public Polygon {
  public:
    int area(){
      return width*height; }
};

class Triangle: public Polygon {
  public:
    int area(){
      return width*height/2; }
};
```

```
int main () {

  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;

  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);

  ppoly1->printarea();
  ppoly2->printarea();
}
```

It is now possible for a member of the abstract base class `Polygon` to use the `this pointer` to access the proper virtual members, even though `Polygon` itself has no implementation for this function

# Building an executable

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




```
g++ -c main.cc
g++ -c TwoVector.cc
g++ main.o TwoVector.o -o TwoVector.exe
```

# Makefiles

Now suppose we modify **`TwoVector.cc`**

We only need to recompile this file, not **`main.cc`**

But this is hard to keep track of, especially if we change a header file

**`Makefile`**

```
# S. Boutle
# A Makefile to build TwoVector.exe

TwoVector.exe : main.o TwoVector.o
  g++ -o TwoVector.exe main.o TwoVector.o

main.o : main.cc TwoVector.h
  g++ -c main.cc

TwoVector.o : TwoVector.cc TwoVector.h
  g++ -c TwoVector.cc
```

```
make
```

builds the target files

# Compiling with make

```
# S. Boutle
# A Makefile to build TwoVector.exe

CC=g++
TARGET=TwoVector
OBJECTS=main.o TwoVector.o

$(TARGET).exe: $(OBJECTS)
  @echo "**"
  @echo "** Linking Executable"
  @echo "**"
  $(CC) $(OBJECTS) -o $(TARGET).exe

clean:
  @rm -f *.o *~

veryclean: clean
  @rm -f $(TARGET).exe

main.o : main.cc TwoVector.h
  $(CC) -c main.cc

TwoVector.o : TwoVector.cc TwoVector.h
  $(CC) -c TwoVector.cc
```

makefiles can become extremely complicated and long

Often they are themselves not written by “humans” but rather constructed by an equally obscure shell script

Most often, software packages are distributed with a makefile that you may or may not need to edit

# This Week's Lecture

- More classes
  - Class templates
  - Inheritance
  - Polymorphism
- Makefiles

# News

- **Assignment 2:**
  - Due: 7th November before the 3rd lab session

- **Assignment 3:**
  - Due: 14th November before the last lab session

- **Labs:**
  - Monday afternoons, 2-5pm
  - Room 220a, Kelvin Building, University of Glasgow
  - Please indicate if you intend to come in person **PLEASE UPDATE**
    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

- **Assignment 4 will be available on Monday**
  - Watch out for the SUPA mail
  - Due 28th November

# 3rd Assignment

Due Monday 14th November before final lab session

In this task, you will solve the same task as in assignment 2, using the same input file as from Lab2, with planets by name, mass and distance to the Sun. But now you should use a class to solve this problem.

1. Create a class Planet that has the data members of name (type string), mass and distance (type double). Create this in it's own separate files (.cc and .h) and define member setter and getter functions for each of the data members.

While reading the file line by line, meaning planet by planet, for each line you will build a new object of type planet in your main program. Store all of them in a vector of planets, namely std::vector<Planet>. That is the power of creating our own types, as we can use them just as we would have used the regular types already created by C++, such as std::string and double. Test your setter and getter functions to ensure they work as expected.

2. You are familiar now with the std::sort of a std::vector. The default is by comparing the two objects. But what does it mean that a planet is larger in value than another one? You can overload the operators > and < to have multiple definitions, be it (a) that the name comes first alphabetical order, or (b) that the mass is larger, or (c) that the distance to the sun is larger.

To change easily from one sorting to the other, note that std::sort can take user-defined functions. Define a function for each three cases above, and repeat the tasks for assignment 2.

# 2nd Assignment

Due Monday 7th November before 3rd lab session

The file input.txt contains a list of planets (name, mass in kg in scientific notation, average distance to the Sun in astronomic units, where 1.0 is the distance from the Earth to the sun).

1. Print on each line the name of the planets, in alphabetical order.
2. Print on each line the name of the planet followed by its mass, in order of the mass. Same for distance to the sun.
3. Print the planet with the smallest and largest mass. Same for distance to the sun.
4. Compute the weighted average distance to the Sun of all the planets, defined as sum over the planets of mass times distance, and all divided by the sum of the masses of all planets

Tips:

1. scientific notation, useful for large numbers:
http://www.cplusplus.com/forum/general/9616/

2. std::map<std::string, double> to store pairing between a word and a number, when iterating over a map, the items come ranked, and if they are words, that means alphabetical order.

3. Use algorithms on the STL containers, like getting max and min from an std::vector<double>
http://stackoverflow.com/questions/10158756/using-stdmax-element-on-a-vectordouble