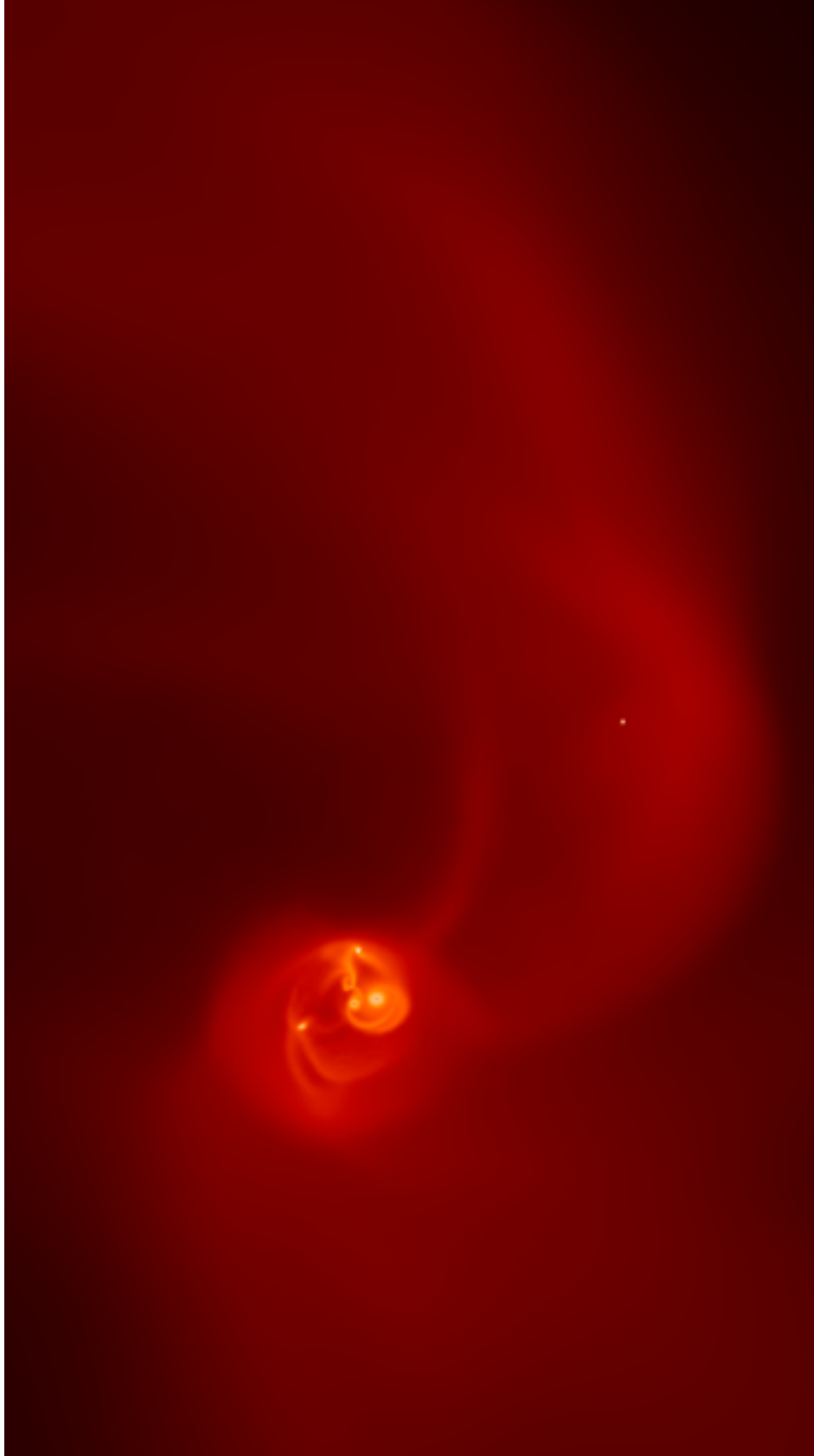
Welcome to C++

David Hubber

USM, LMU, München
Excellence Cluster Universe,
Garching be München

26th Sept 2015

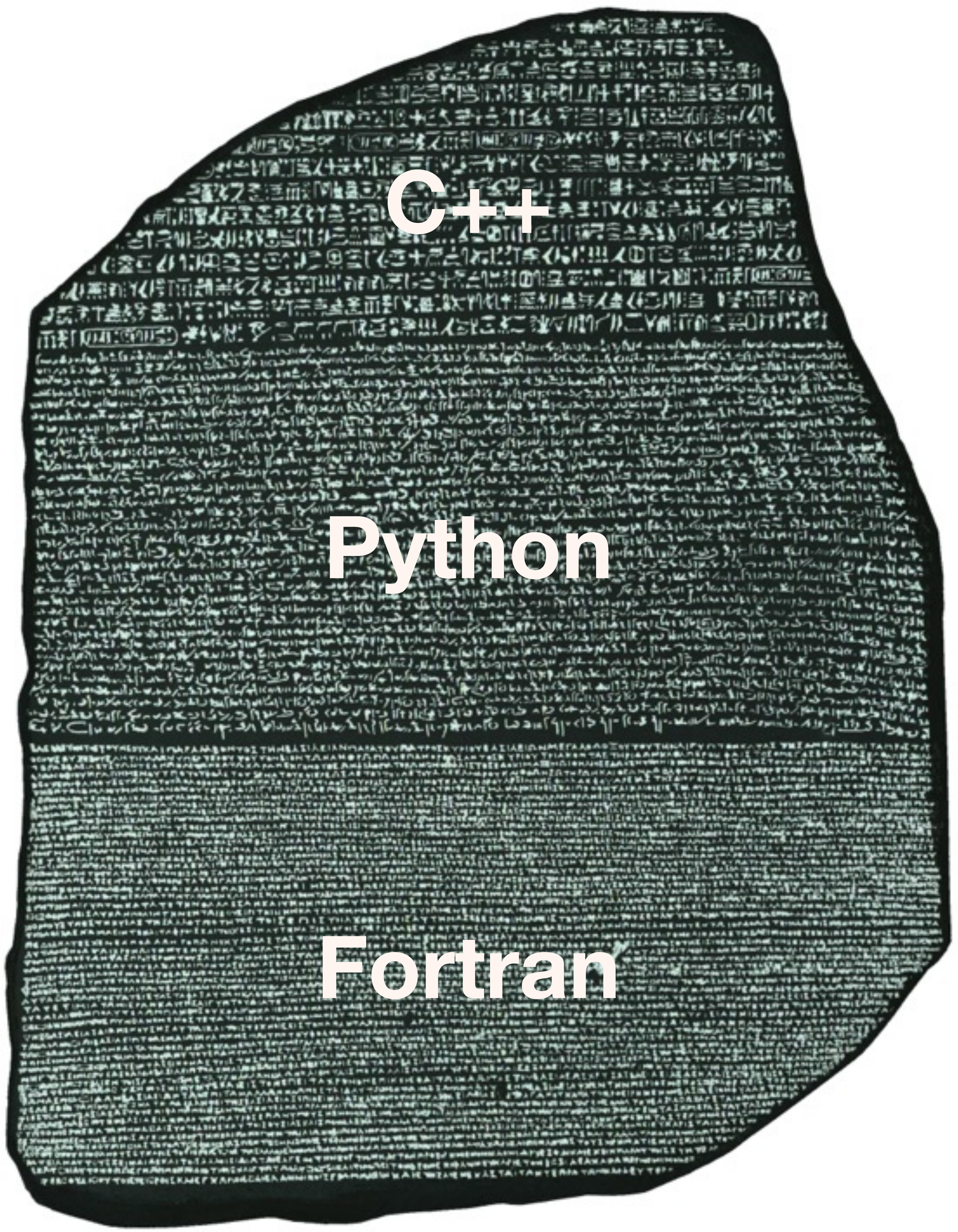


Why C++?
What was wrong with Fortran?

- C++ is an **object oriented language**, which has many advantages in big coding projects over traditional procedural languages (such as Fortran).
    - Object-oriented programming generally leads to more modular coding, which improves code reability
    - It also allows more natural code re-use (and in tandem reducing code duplication) through inheritance
    - Even if you don't want to use advanced C++ features, you can still use the code in a simple procedural way
- C++ supports **generic programming**, which allows generic functions to be constructed reducing code duplicaition
- Outside of science, C++ is one of most used languages in software engineering and is a useful addition to your CV.

Course Philosophy

- First of all, this small crash-course will **NOT** make you a C++ expert overnight!
- There is **no possible way** we can go through all of C++ in one day!
- Instead this course is designed to
    - Help people who **already know how to program** (i.e. hopefully **you guys**) to bridge over to C++
    - Introduce and practice some of the concepts in object-oriented programming that are then used in GANDALF
    - Hopefully convince you of the usefulness of object-oriented programming and to use it in the future


C++
Python
Fortran

Structure of this crash-course

- Basic usage of C++ will be covered by a short overview of the most important features of the language
- However, most of the ‘trivial’ questions will hopefully be covered by the C++ phrasebook **so we will not cover those directly** (so **please look there before asking any questions**).
- Next, we will discuss some of ‘newer’ features in C++, such as object-oriented programming, templates and inheritance, with small examples and practical exercises
- Then we have a cup of tea or coffee to recharge!
- Finally, we (i.e. you) will attempt to write a small object-oriented program in C++

Practical 1 : Hello World

Comments in C++ can be written with ‘*// comment here*’

```
// Hello World - my 1st C++ program
#include <iostream>

int main()
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

iostream is a C++ library containing various IO functions

main is the start point of any C++ code. Every C++ program must have one, and only one main function (similar to **program** in Fortran)

std::cout is an IO function for printing strings (and other variables) to screen

Functions should **return** a value of the same type as the function (i.e. in this case, int)

IMPORTANT! - Every regular C++ line must end with a semi-colon. Most compiler errors result from forgetting it (Bet you a pint of beer you forget one this week!!)

The **function code** is contained within the ‘squigly brackets’, i.e. { …. }

- Type this into the file ‘HelloWorld.cpp’

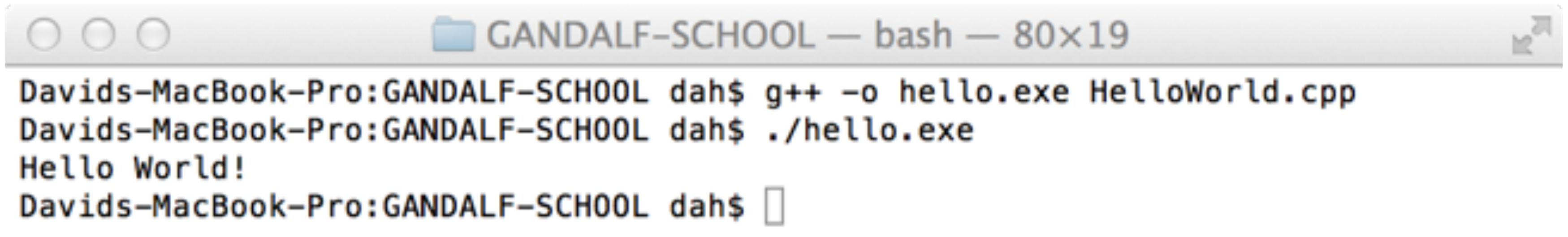
Compiling and running 'Hello World'

- To compile the program, type

```
c++ -o hello.exe HelloWorld.cpp
```

where 'c++' is the name of the C++ compiler (e.g. g++, clang++, icpc)

- And then running the program should give the output :



```
Davids-MacBook-Pro:GANDALF-SCHOOL dah$ g++ -o hello.exe HelloWorld.cpp
Davids-MacBook-Pro:GANDALF-SCHOOL dah$ ./hello.exe
Hello World!
Davids-MacBook-Pro:GANDALF-SCHOOL dah$ 
```

Basics of C++

- C++ programs usually comprise of two main file types :
    - **Header files** (*.h, *.hpp) - contain definitions, interfaces and variables
    - **Source files** (*.c, *.cpp, *.cxx) - contains the source code itself
- C++ programs **MUST** contain one and only one **main function** :
    - This is the entry and exit point of the program when run
- C++ is a **case-sensitive** language
- All regular lines of C++ code **MUST END** with a semi-colon character, i.e. a ';'
    - If anyone gets through this course **WITHOUT forgetting** a semi-colon when compiling, I'll buy them a beer!!

Data types in C++

C++ data types

```
int i;                  // 32-bit integer
long int j;             // 64-bit integer
unsigned int k;         // 32-bit integer, positive only
bool flag;              // Boolean/logical flag
float x;                // 32-bit floating point
double y;               // 64-bit float
char letter;            // Single character string
std::string name;       // 'std' variable length string
```

Equivalent Fortran types

```
integer :: i
integer(kind=8) :: j

logical :: flag
real :: x
real(kind=8) :: y
character :: letter
character(len=100) :: name
```

- Unlike Fortran, you can declare C++ variables **ANYWHERE** in a function (but **before** the variables is actuallyused)
- You can also **initialise** a variable in the declaration, e.g.

```
int i = 100;
std::string name = "Fred";
```

- To declare a variable as a constant, add the keyword ‘const’

```
const int N = 10;
```

User-defined data types : structs

- **structs** are user-defined data structures that can contain any combination of the basic C++ data types

```
struct Particle {
  int id;
  float m;
  float r[3];
  float v[3];
  float a[3];
};
```

- They are potentially very powerful in :
  - Easily creating data types to represent different complicated things with different data types (NOT possible with simple arrays)
  - All the data for a single instance of the struct (e.g a single particle in this simple example) is **contiguous** in memory, i.e. more cache efficient

Functions in C++

- **Functions** in C/C++ require
  - An argument list (even if it's nothing/void)
  - A return type (even if it's void)
  - All the function code to be in { ….. }

```
int AddNumbers(int a, int b) {
  int c = a + b;
  return c;
}
```

  - The function then may simply be called with

```
int total = AddNumbers(1, 3);
```

  - The function may also be called without recording the return value

```
AddNumbers(1, 3);
```

Practical 2 : Simple programs

- Write a function to compute and return N-factorial (i.e. N!). Print the final value out to the screen.
- Modify the previous program to allow the user to input N from the command line.
- Modify the program to repeatedly ask for and compute the factorial. Exit the program when a negative input is given.
- What is the maximum value of N you can use before the program breaks down? Why is this?

Arrays in C++

- Static arrays (i.e. created at compile time) can easily be created in C/C++ just like declaring a normal variable

```
int ids[10];
float r[3];
float v[10][3];
```

- If required, arrays have to be initialised

```
int ids[10] = {0};
for (j=0; j<10; j++) ids[j] = 0;
```

- **Important** : Note that **arrays are indexed from 0 to N-1**, so in the example above with ids[10], the first element is id[0] and the final element is id[9]

- Arrays can also be dynamically allocated in C++
- Before we can discuss allocation of arrays, we must learn about **pointers**!

Pointers!
With great power comes great responsibility

- **Pointers** are one of the most powerful and deadly forces known to C++ programmers
- In their most basic form, pointers simply **point at an address in memory**

Address	1	2	3	4	5	6
Variable			x		xptr	
Value			1.5		3	

```
float x;                        // Basic variable
x = 1.5f;                       // Set value of variable

float *xptr;                    // Pointer to a float variable
xptr = &x;                      // Set pointer to point at address of variable 'x'
```

- The contents of the address itself can be accessed with the * operator

```
std::cout << "xptr  : " << xptr  << std::endl;      // Outputs '3'
std::cout << "x     : " << x     << std::endl;      // Outputs '1.5'
std::cout << "*xptr : " << *xptr << std::endl;      // Also outputs '1.5'
```

Pointers!
What's the point?

- The big question to some people might be **'How can this possibly be useful??'**
- Pointers have a range of uses
  - They are used to **allocate memory**
  - They can be used to track variables or arrays directly **without copying** (especially useful if several places need to know the value of a variable)
  - This are used to pass arrays as arguments in functions (related to previous point)
- However, they also have some drawbacks
  - If the variable or array goes **out of scope** but the pointer is still active, then the pointer is therefore pointing to nothing at all; called a **dangling pointer**
  - If there are bugs involving pointers, they can lead to memory corruption which can be hard to track down

Allocating memory in C++

- Memory is allocated in C++ using the **new** operator

```
float *x;
x = new float[100];
```

- It can be used on either regular data types (e.g. int, float) or on dervied data types (i.e. structs)

```
struct Complex {
  float re;
  float im;
};
Complex *com;
com = new Complex[200];
```

- These can now be accessed as normal arrays

```
x[0] = 0.0f;

com[10].re = 1.0;
com[10].im = 0.0;
```

- The memory can be deallocated using the **delete** operator

```
delete[] com;
delete[] x;
```

- Note that forgetting to deallocate arrays when leaving a function can lead to a **memory leak**

Arrays and pointers

- When it comes to passing arrays as function arguments, arrays are represented a simple pointer (in fact, **many books will say that “Arrays ARE pointers”**)
- In fact, when an array is passed as an argument, **only the pointer itself** is ever passed (which points to the beginning of the array in memory)

```
float sum_array(int N, float *values) {
  int i;
  float sum = 0.0f;
  for (i=0; i<N; i++) sum += values[i];
  return sum;
}
```

- **However**, the function only knows the beginning of the array, and **not the size or the end.**
- Therefore, if you try to access beyond the end of an array (e.g. values[N], values[N+1], etc..), you will get **total garbage AND the compiler or program won’t complain!**

Practical 3 : Pointers

- Write a small program that creates a simple variable i and a pointer to it, iptr
- Experiment with changing the values of i and iptr. What happens if you change the value of i? What happens if you change the value of iptr?
- Write a program that calculates and stores the value of N! in an allocated array up to a given size (user input from the command line). Write all values to screen from the array.
- Print the values of the array pointer before allocation, after allocation and finally after deallocation

What exactly is an ‘object’?

- An object in a programming sense is very similiar to how we might describe an object in real-life. Take for example a car :
  - It has properties, such as its colour, size, top speed, etc.. (similar to **variables**)
  - It can do things, such as drive forward, reverse, beep the horn, etc.. (similar to **functions**)
- In Astrophysics, we could take a star for example :
  - Mass, luminosity, radius, position, velocity, etc..
  - Exerts/feels gravitational force, moves, explodes, etc..
- However, objects can also be more abstract things that have properties and functions, like **algorithms**, e.g. hydrodynamics algorithm
  - Fluid density, velocity, internal energy, etc …
  - Compute Euler Equations, move/transfer fluid, etc..

Anatomy of a class

Public variables
(anyone can see and modify these)

Constructor & destructor functions
(More about these soon)

Public functions
(anyone can call these)

Private variables
(only internal functions can see and modify these)

Private functions
(only internal functions can call these)

Car.h

```
class Car {

 public:
  bool automatic;
  int numDoors;
  int colour;
  string numberPlate;

  Car(int, string);
  ~Car();

  bool Accelerate(float);
  bool Brake();
  float TurnWheel(float);
  bool TurnKey(bool);


 private:
  int numCylinders;
  bool engineActive;
  float brakeTemperature;
  float horsePower;

  bool ApplyBrakeDiscs();
  bool InjectPetrolIntoEngine(float);

};
```

Functions in classes

- Functions in classes (usually defined in say, “Car.cpp”) must be written using the class name appended to the function name as follows

Car.cpp

```cpp
#include "Car.h"

// Constructor
Car::Car(int _colour, string _numberPlate) {
  automatic   = false;
  numDoors    = 4;
  colour      = _colour;
  numberPlate = _numberPlate;
}

// Destructor
Car::~Car() {
}

// Public functions belonging to class
bool Car::Accelerate(float accelRate) {
  return InjectPetrolIntoEngine(accelRate);
}

bool Car::Brake(void) {
  return ApplyBrakeDiscs();
}
```

What is a constructor exactly?

- Following our idea of an object being like some real-life entity, like a car, the car doesn't just build itself right? Your car has to be built by a mechanic in a workshop or a factory, and built to whatever specifications (e.g. colour, tyre choice) the buyer demands!
- So, a **constructor 'builds' the object for you, by allocating the memory you need for your object and setting values for the variables**
- An object can be constructed in the same way a normal variable is declared (except including the constructor parameters), i.e.

```
Car myCar(red, "GANDALF");
```

- Alternatively, if the variable is declared as a pointer, it can be constructed with the **new** command, i.e.

```
Car *myCar;
myCar = new Car(red, "GANDALF");
```

What is a constructor exactly?

- Another important reason to use constructors is for variable initialisation.
- How many in this room spent **minutes/hours/days/weeks/months** debugging some code only to find that **you forgot to initialise the variables**?
- OR perhaps more importantly, **you assumed the compiler would set the value to zero and it DIDN'T!**
- With a constructor, you have complete power to initialise the values to whatever you want and ensure it is always initialised whenever a new object is created

```
Car *myCar;
myCar = new Car(red, "GANDALF");
```

```
Car::Car(int _colour, string _numberPlate) {
  automatic   = false;
  numDoors    = 4;
  colour      = _colour;
  numberPlate = _numberPlate;
}
```

Practical 4 : A basic vector class

Vector.h

```
class Vector
{
public:
  float v[3];

  Vector();
  ~Vector();

  float Magnitude();
  Vector UnitVector();
};
```

- Complete the Vector class, placing the source code for the functions in a 'Vector.cpp' file (remember to include the Vector.h file)
- Create a main program which uses the Vector class. Check that it compiles and works correctly. Output the magnitude and unit vector of some vector.
- Create a function (external to the class) that takes two Vectors as arguments and returns the scalar product

Templates : What are they?

- Consider you are writing some function that you might need to use with many different data types (e.g. min/max), then how do you do it without having to do this?

```
int min(int a, int b) {
  if (a < b) return a;
  else return b;
}

long int min(long int a, long int b) {
  if (a < b) return a;
  else return b;
}

float min(float a, float b) {
  if (a < b) return a;
  else return b;
}

etc..
```

- Imagine there was some ‘magical’ way that you could write the function just once and it would work for all possible valid data types!

Templates : Simple example

- A **template** is a declaration that uses a generic data type (often called 'T') to create the function (at compile time) for whichever specific data type you need
- For our min function example, the templated function would be written as :

```cpp
template <typename T>
T min(T a, T b) {
  if (a < b) return a;
  else return b;
}
```

compare to

```cpp
float min(float a, float b) {
  if (a < b) return a;
  else return b;
}
```

- Whenever our 'min' function is called in the code, the compiler checks the types of the arguments, and if it matches, then it generates the function for that type automatically.

Practical 5 : Simple template example

- In a C++ program, this could be used in the following way

```cpp
#include <iostream>

template <typename T>
T min(T a, T b) {
  if (a < b) return a;
  else return b;
}

int main() {
  float a = 1.0f, b=2.0f;
  float minval = min(a, b);
  std::cout << "Minimum value : " << minval << std::endl;
}
```

- Write this function to a file, compile it and run it.
- Try other variable types (e.g. int, double) to show it still works
- What happens if the types of a and b don't match?

Template parameters

- Templates can also be used to generate functions using some specific value of a parameter
- Take our particle data structure that we defined earlier :

```
struct Particle {
  int id;
  float m;
  float r[3];
  float v[3];
  float a[3];
};
```

- What if we wanted to use this in 1 or 2 dimensions? We could write out a 1 or 2 dimensional version, but this would be a waste again!
- Instead **we can define a template parameter for the number of dimensions**

```
template <int ndim>
struct Particle {
  int id;
  float m;
  float r[ndim];
  float v[ndim];
  float a[ndim];
};
```

- We could then declare this as a variable in the code as any other variable but **with the template parameter value added in angular brackets**, < … >, e.g.

```
Particle<1> part1d;
Particle<2> part2d;
```

Templating a class

- This approach can also be used to **template an entire class**
- However **ALL** functions external to the class definition (e.g. the class functions in the cpp files) must have the template declaration

```
template <int ndim>
float Vector<ndim>::Magnitude()
{
  int k;
  float magsqd = 0.0f;
  for (k=0; k<ndim; k++) magsqd += v[k]*v[k];
  return sqrtf(magsqd);
}
```

- Also, each version of the class you wish to template for a particular parameter (e.g. ndim = 1, 2, etc..) MUST be declared in .cpp file with :

```
template class Vector<1>;
template class Vector<2>;
```

Practical 6 : Templating the Vector class

- Template our earlier Vector class so that it can be used for **any given dimensionality**.
- Also remember to template the dot product function
- Verify that it works in 1 and 2 dimensions
- What happens if you try and compute the dot product between Vectors of different dimensionality?

What is inheritance?

- Let's go back to our car class!
- Let's say the mechanic or factory plans on making several different designs that have different functionality
  - Some cars may be 2-wheel drive and some 4-wheel drive
  - Some cars might be automatics, some use manual gear shifts
  - Some things may be present in one model but not in the other(e.g. built-in GPS, auto-pilot when you feel sleepy)
  - But **a lot of things are exactly the same** (e.g. dimensions, chasis, lights, etc..)

- Since most of the basic design is the same, instead of starting with brand new plans, the mechanics and engineers can take the original basic design and modify it to result in the 'improved' car.
- The new design will **inherit** features from the original design

Creating a virtual base class

- One key conept with inheritance are **virtual functions**
- A virtual function is a function whose definition can be **replaced by an alternative version in a child class**
- This allows programmers to create **different implementations of the same algorithm**
- A virtual function is simply declared by adding the **virtual** keyword in front of the return type

```
virtual bool Accelerate(float);
```

- If the virtual function ends with ‘ = 0’, then the class has **no definition** and the child class is **required** to provide a defintion.

```
virtual bool Accelerate(float) = 0;
```

Creating a virtual base class

```
class Car {
 public:
  Car();
  virtual bool Accelerate(float) = 0;
};
```

```
class BatMobile : public Car {
 public:
  BatMobile() : Car() {};
  virtual bool Accelerate(float);

 private:
  bool ActivateFlameThrower();
};
```

```
class DeLoreanTimeMachine : public Car {
 public:
  DeLoreanTimeMachine() : Car() {};
  virtual bool Accelerate(float);

 private:
  bool ActivateFluxCapacitor();
};
```

- We can then take a pointer of type ‘Car’ and then create an object of any child class derived from the Car class
- We can then call any of the Car class functions (but not the individual private functions)

```
Car *car;
if (myCar == "BatMobile") {
  car = new BatMobile();
}
else if (myCar == "DeLorean") {
  car = new DeLoreanTimeMachine();
}
car->Accelerate();
```

Practical 7 : Inheritance

- We will create a super simple class for computing a basic mathematical function, such as computing the integer power of some given number
- First, create a base virtual class containing the basic interface

```
class PowerFunction {
 public:
  PowerFunction() {};
  ~PowerFunction() {};

  virtual float Power(float, int) = 0;
};
```

- Next, create **TWO child classes** that inherit from this simple empty class, one that uses the C++ implementation (i.e. powf), and one that uses your own simple implementation
- Create a main program that creates objects and uses both

Practical : Write a simple C++ N-body code

- For a nice simple Astrophysics-related example, we will write a basic N-body code in C++ to try and put together all of the ideas we've discussed.

- The equations of motion are simply Newton's law of gravitation, either the standard form (left) or using a softening length (right)

$$\mathbf{a}_i = -\sum_{j=1}^{N} \frac{G\,m_j}{|\mathbf{r}_{ij}^2|}\,\hat{r} \qquad \mathbf{a}_i = -\sum_{j=1}^{N} \frac{G\,m_j}{|\mathbf{r}_{ij}^2 + \epsilon^2|}\,\hat{r}$$

- For super-simple Euler integration, we simply compute the force at the beginning of the timestep, and the integrate the positions and velocities accordingly, i.e.

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\,\Delta t$$
$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \mathbf{a}_i(t)\,\Delta t$$

Practical : Write a simple C++ N-body code

- We will need THREE basic classes to create this N-body simulation
  - A **Star** class : simple data structure for a star (e.g. position, velocity, mass etc..)
  - A **Nbody** class : contains main N-body algorithm with related functions
  - A **NbodySimulation** class : container for whole simulation; reads in parameters file, controls main loop, terminates simulation, etc..
- We will create a super-simple parameter file

```
euler       // N-body algorithm
ic          // Initial conditions
8           // No. of stars
0.1         // epsilon (softening length)
1.0         // Maximum velocity given to stars
0.01        // (Constant) timestep
1.0         // Total simulation time
```

Practical : Write a simple C++ N-body code

- The plan is :
  - Look at the basic (but incomplete) framework provided
  - Write the basic N-body class using an **Euler integrator** and finish the missing parts of the code to get it working
  - Next write an inherited N-body class using a different integrator, e.g. **Leapfrog integrator**

$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\,\Delta t + \frac{1}{2}\,\mathbf{a}_i(t)\,\Delta t^2$$

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \frac{1}{2}\left(\mathbf{a}_i(t) + \mathbf{a}_i(t+\Delta t)\right)\,\Delta t$$

Practical : Write a simple C++ N-body code

- Extra challenges (if everything goes well) :
  - Can you generate regular periodic output from the simulation and plot this to show the motion of the stars?
  - Calculate the energy error at the end of the simulation
  - Can you template the various classes so that the code can be run in 2 or 3 dimensions chosen at run-time? (1-dimension doesn't really make sense for N-body)
  - Use STL vectors instead of C++ arrays (new/delete)