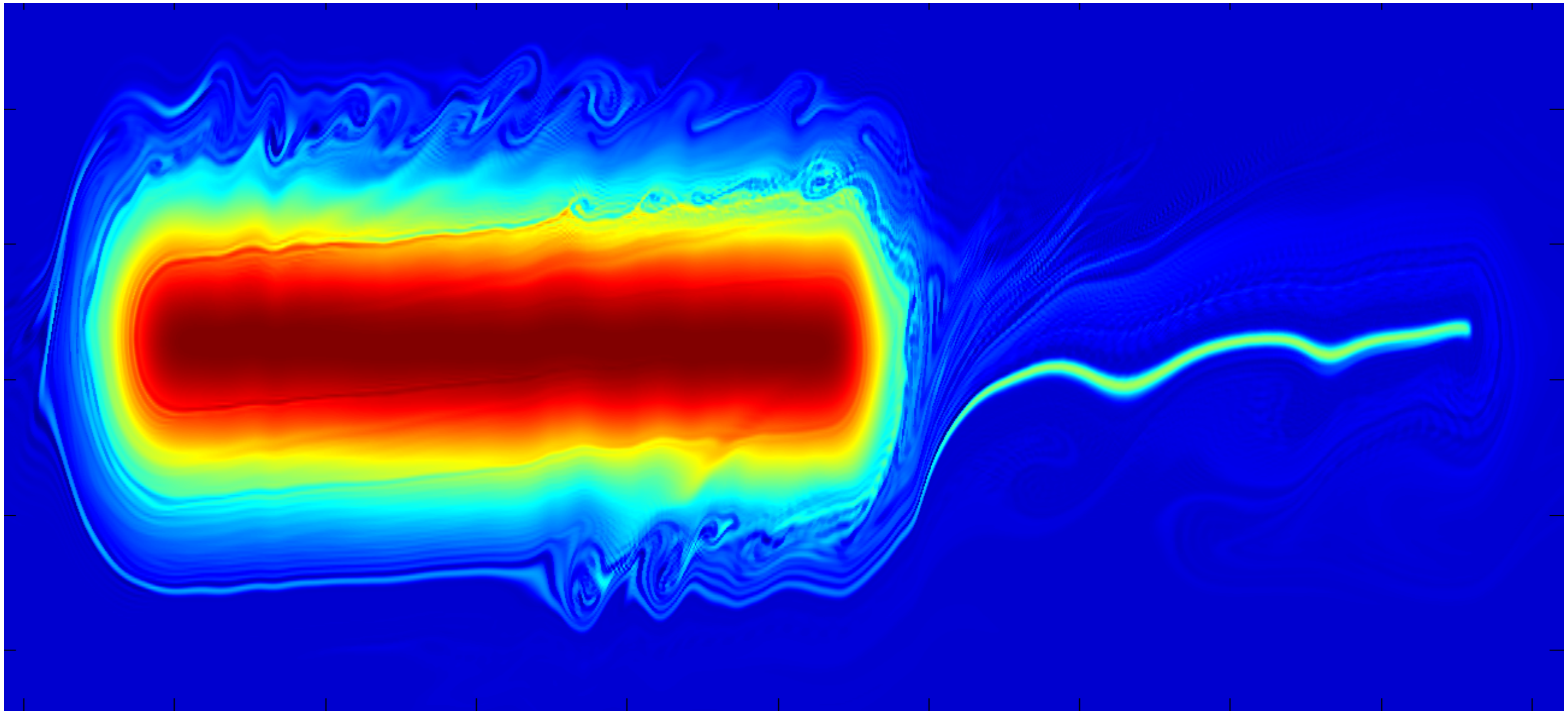


Numerical Simulations II - WS 2019

Chapter 3 - Structs

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Typedef

- Typedef allows us to introduce our own datatypes

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

typedef int* intp;
typedef complex<double> cmplx;

int main(void)
{
    int* nx,ny; //Only nx is a pointer,
                //ny is a regular int


    int *NX, *NY; //Both are pointers to int

    intp MX, MY; // Both are pointers to int

    cmplx c; // c is a double precision complex number

    cmplx* cp= new cmplx[*MX]; //cp is a pointer
                               //to an array of length
                               //*MX
}
```

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Structs

- Structs are data structures that are made up from several components. Each component may have another datatype.
- Example: When we study the motion of a particle due to some force, we will need the values v_x, v_y, v_z, x, y, z. Maybe we will need additional values like mass m, charge q, ...
  - Structs allow to summarize these under the same name
- Structs are defined as

```
struct{
        component1;
        component2;
};
```

  - Don't forget the finalizing semi-colon!
- Any datatype we know can become component of a struct

Structs

- Once a struct is defined it can be used like any other datatype

```
#include <cmath>
#include <iostream>

using namespace std;

//-------------------------------
struct particle{
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};

//-------------------------------
//-------------------------------
int main(){
    particle p1;

    p1.x = 0.0; p1.y = 0.5; p1.z=0;
    p1.vx = 1.0; p1.vz = 0.2;
    p1.m = 1836; p1.q = -1;


    return 0;
}
```

Accessing members via pointers to structs

- If we have a pointer to a struct, access to the struct members is provided only by the -> operator
- No explicit de-referencing is necessary

```
#include <iostream>
using namespace std;
//--------------------------------
struct particle{
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};
//--------------------------------
int main(){
    particle p1;
    particle* pp = &p1;

    p1.x = 0.0; p1.y = 0.5; p1.z=0;
  p1.vx = 1.0; p1.vz = 0.2;
  p1.m = 1836; p1.q = -1;

  // pp.x = 1; // Will not work since pp is a pointer
  pp->x = 1;
  pp->y = 1;

  cout << "x = " << p1.x << ",\t y =" << p1.y << endl;
  return 0;
}
```

Member-functions of structs

- So far structs are structures that help us to encapsulate various variables into a coherent structure
- Usually we do not only store data, but also process this data. We can make the functions processing the data stored in a struct a member of the struct itself. These functions are called *member-functions*.
- Example: In the particle struct we store all velocity components and all coordinates of the particle. We might want to calculate
  - the distance of the particle from a particular place (we need *x,y,z*)
  - the magnitude of the velocity (we need v_x, v_y, v_z)
- Making a function part of the struct and accessing the function is done very similar to variables:

```
struct{
   double vx, vy, vz;
   double v();
}
```

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Member-functions of structs

Without member-function

```
#include <cmath>
#include <iostream>
using namespace std;
//------------------------------
struct particle{
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};
//------------------------------
double v(const particle& p);
//------------------------------
int main(){
    particle p1;

    p1.vx = 1.0; p1.vz = 0.2;

    double velocity = v(p1);

    return 0;
}
//------------------------------
double v(const particle& p){
    return sqrt(p.vx*p.vx + p.vy*p.vy + p.vz*p.vz);
}
```

With member-function

```
#include <cmath>
#include <iostream>
using namespace std;
//------------------------------
struct particle{
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
    double v();
};
//------------------------------
int main(){
    particle p1;

    p1.vx = 1.0; p1.vz = 0.2;

    double velocity = p1.v();

    return 0;
}
//------------------------------
double particle::v(){
    return sqrt(vx*vx + vy*vy + vz*vz);
}
```

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Member-functions of structs

- Previously the function `v(particle& p)` was a general function, accepting any struct p.
- As a member function, the function belongs to a particular particle struct. Thus, v() has no longer any paramters.
- Since it belongs to a particular struct (here named `p1`) it can access the other struct members directly
  - We can write vx,vy,vz to access the variables in the function v()
- When we define the function v(), we now have to tell the compiler explicitly to which type of struct the function belongs. This is done via the *scope operator* ::
  - `double v(particle& p)` ➠ `double particle::v()`

```
#include <cmath>
#include <iostream>
using namespace std;
//-------------------------------
struct particle{
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
    double v();
};
//-------------------------------
int main(){
    particle p1;

    p1.vx = 1.0; p1.vz = 0.2;

    double velocity = p1.v();

    return 0;
}
//-------------------------------
double particle::v(){
    return sqrt(vx*vx + vy*vy + vz*vz);
}
```

Abstract datatypes and objects

- In C structs can only hold data, no functions, thus they are new datatypes (however, one of the variables could be a pointer to a function...)
- In C++ structs can contain data and member-functions which process this data. This is called *abstract datatype*.
- When we create a variable which holds a struct including member-functions we call this variable an *object*. The object is a so called *instance* of the struct.

public, private, protected

- We can restrict access to data and function members of a struct from the outside via the keywords `public`, `private` and `protected`
- When no explicit access rule is specified the default for struct members is `public`.
  Every function of our program can modify all public data and call all public member-functions of the object.
- Variables and methods marked as `private` can not be altered/called from the outside of the struct. Private variables can only be changed by a function of the object itself. Private functions may only be called by other member functions of the object.
- Protected ➠ Later, when we talk about classes

- Why hide data or functions via `private` from the outside world?
  - Reduce errors; avoid others (but mainly yourself) from changing data
  - Stronger differentiation between the new datatype and internal infrastructure
  - Implementation of private parts may change completely but the outside world will never notice

public, private, protected

To distinguish between parameter and member variable we use different names for both

```
#include <iostream>
#include <cmath>
using namespace std;
//-------------------------------
struct particle{
public:
    double get_mass(){return m;}
    void set_mass(const double M){m = M;}
    int get_number(){return n;}
    void set_number(const int N){n =N;}
    void set_pos(const double X, const double Y, const double Z);
    void initialize();
    double get_momentum();
private:
    double v();
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};
//-------------------------------
int main(){
    particle p1;

    p1.initialize();
    p1.set_number(2);
    p1.set_mass(1);

    cout << p1.get_number() << "\t" << p1.get_momentum() << endl;
}
```

functions can be declared and defined within the struct

```
double particle::v(){
    return sqrt(vx*vx + vy*vy + vz*vz);
}
//-------------------------------
void particle::set_pos(const double X,
                       const double Y,
                       const double Z){
 x = X; y = Y; z = Z;
}
//-------------------------------
void particle::initialize(){
    m = 0; q = 0;
    vx = 0; vz = 0; vy = 0;
    x = 0; y = 0; z = 0;
}
//-------------------------------
double particle::get_momentum(){
    return v()*m;
}
```

Member-function can access private data-members

Member-function can access private member-function

C++ - Structs

Passing structs as arguments

```
void work(particle p, particle& q, particle* r,
          const particle s, const particle& t){

    p.initialize();
    q.initialize();

    (*r).initialize();
    r->initialize();

    // t.initialize();
    // s.initialize(); //will not work, since s is const
                       //but initialize() does not explicitly
                       //guarantee that s will not be changed

}

//------------------------------
int main(){
    particle p1,p2,p3,p4, p5;

    work(p1, p2, &p3, p4, p5);

}
```

- Structs are passed like any other datatype.
- p is passed as a copy of p1 from main
  - might be problematic if particle struct contains pointer (see later)
  - requires extra memory for copy
- r is passed as pointer
  - needs explicit de-referecing or -> notation to access members
- q is passed as reference
  - most convinient
- passing as const particle s
  - same problems as for p
  - will cause trouble with current definition of member functions

Potential problems with structs

```
int main(){
    particle p1;
    particle p2;

    p1.initialize();
    p1.set_number(2);
    p1.set_mass(100);

    cout << p2.get_number() << "\t" << p2.get_mass() << endl;

    p2 = p1;

    cout << p1.get_number() << "\t" << p1.get_mass() << endl;
    cout << p2.get_number() << "\t" << p2.get_mass() << endl;

}
```

Output:

```
1606416896    0
2    100
2    100
```

- There is no way to force us to initialize any struct object before using it. This is dangerous, there is no pre-defined initial state on which we can rely. We have to make sure everything is initialized before we start working with an object.
- From where does C++ know what it should do when we write `p2 = p1` ?
  Obviously here it did what we expected...

Potential problems with structs

- The = operator from C++ creates a bit-wise copy of the right-hand side object. This is potentially dangerous if the struct contains a pointer...

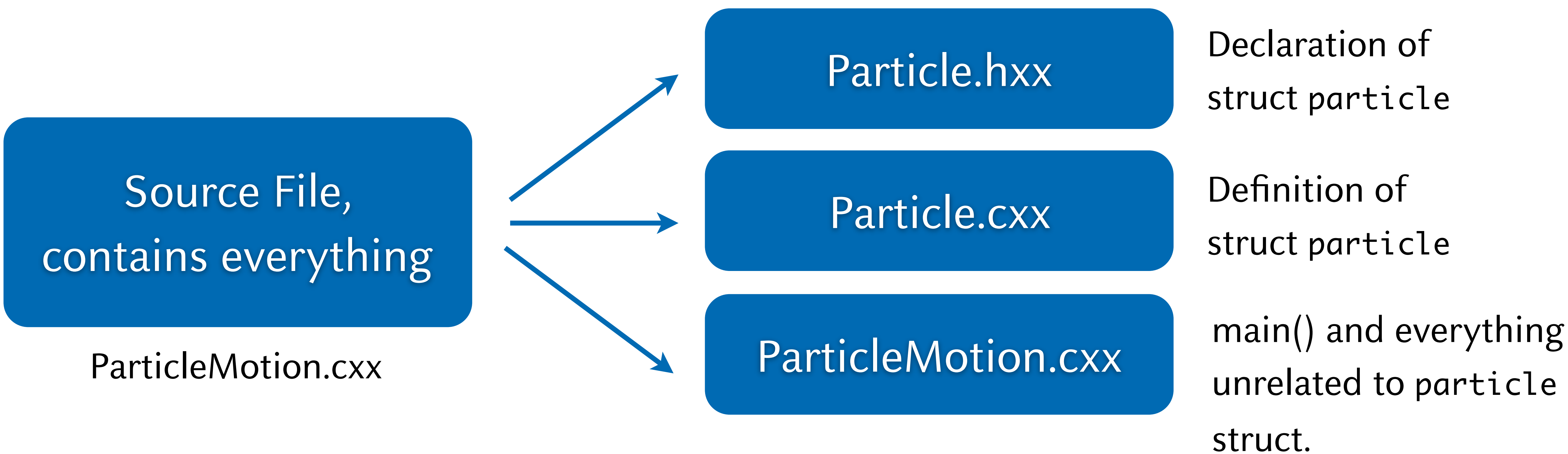
```
#include <iostream>

using namespace std;

struct frog{
    double* p;
};

int main(){
  frog f,g;
  double k = 2;

  f.p = &k;
  g = f;

  *(g.p) = 0.5;

  cout << *(g.p) << endl;
  cout << *(f.p) << endl;

}
```

Output:

```
0.5
0.5
```

Splitting code into separate files

- To keep source code structured, it is advisable to split source code into multiple files
- Typically we split struct declaration from definition of the member-function and from our main program



Splitting code into separate files

```
/*
 * Particle.hxx
 */

#ifndef PARTICLE_HXX_
#define PARTICLE_HXX_


struct particle{
public:
     void set_charge(const double Q){q = Q;}
     double get_mass(){return m;}
     void set_mass(const double M){m = M;}
     int get_number(){return n;}
     void set_number(const int N){n =N;}
     void set_pos(const double X, const double Y, const double Z);
     void get_pos(double& X, double& Y, double& Z);
     void set_v(const double VX, const double VY, const double VZ);
     void get_v(double& VX, double& VY, double& VZ);
     void initialize();
private:
     int n;
     double x,y,z;
     double vx,vy,vz;
     double q,m;
};

#endif /* PARTICLE_HXX_ */
```

- The header file `Particle.hxx` contains only the declaration of the struct `particle`
- The pre-processor commands (lines starting with # are *compile guards*, preventing including the same header more than once)
- A header file should never contain a line wich says `using namespace std;`

Splitting code into separate files

```
#include "Particle.hxx"
#include <iostream>
using namespace std;
//-------------------------------
//-------------------------------
// Set particle position
void particle::set_pos(const double X, const double Y,
                       const double Z){
 x = X; y = Y; z = Z;}
//-------------------------------
// Other get and set functions would appear here
// we skip them because of little space here...
//-------------------------------
// Put all member variables to 0
void particle::initialize(){
     m = 0; q = 0; n=0;
     vx = 0; vz = 0; vy = 0;
     x = 0; y = 0; z = 0;
}
```

- The file Particle.cxx contains the definitions of the particle member-functions (which have not been defined in Particle.hxx)
- To make sure the compiler can make sense of the code, we have to include the header file via `#include "Particle.hxx"`
- Use the double quotes " " when including a header file which is in the same directory, use < > when including files from system paths
- In the source file Particle.cxx it is fine to write `using namespace std;`

Splitting code into separate files

```
#include "Particle.hxx"
using namespace std;
//-------------------------------
//-------------------------------
int main(){
    particle p;

    p.initialize();
    p.set_pos(-10,-1,0);
    p.set_v(0.1,0.1,0);
    p.set_mass(1);
    p.set_charge(1);



}
```

- Our main source file `ParticleMotion.cxx` becomes much more readable now
- We have to inlcude the header file `Particle.hxx` now to make use of the particle struct.

Compiling multi-file projects

- Once we splitted the source-code, we have the files
  - `Particle.hxx`
  - `Particle.cxx`
  - `ParticleMotion.cxx`
- To obtain an executable binary file we need to create *binary objects* from the .cxx files:
  - `g++ -Wall -ansi -c Particle.cxx`
  - `g++ -Wall -ansi -c ParticleMotion.cxx`
- We obtain the two files `Particle.o` and `ParticleMotion.o`, from which we generate the final exectuable via
  - `g++ -o ParticleSimulation Particle.o ParticleMotion.o`
- This can be written short-hand
  - `g++ -o ParticleSimulation Particle.cxx ParticleMotion.cxx`

Compiling multi-file projects

- The benefit of first *compiling* separate source files into binary objects and then *linking* the objects into an executable binary file is that we can shorten time for re-compiling.
- When we change code which belongs only to one object, only this object has to be recompiled (all others remain unchanged) to link a new version of the executable.

- We can also spread binary objects to other users which can then link them to their programs to use our functions.
- The other users however needs both, the binary file and the header-file. Only the header file contains the information we need (description of functions headers, structs,...) for our C++ code to use whatever is in the object.
- This is the idea of a library: Provide readily compiled functions and their interfaced to a user.