C++ - Pre-lecture 4

Introduction to classes

Caterina Doglioni, PHYS30762 - OOP in C++ 2023
Credits: Niels Walet, License:

MANCHESTER
1824
The University of Manchester

Part 0.1: in this lecture

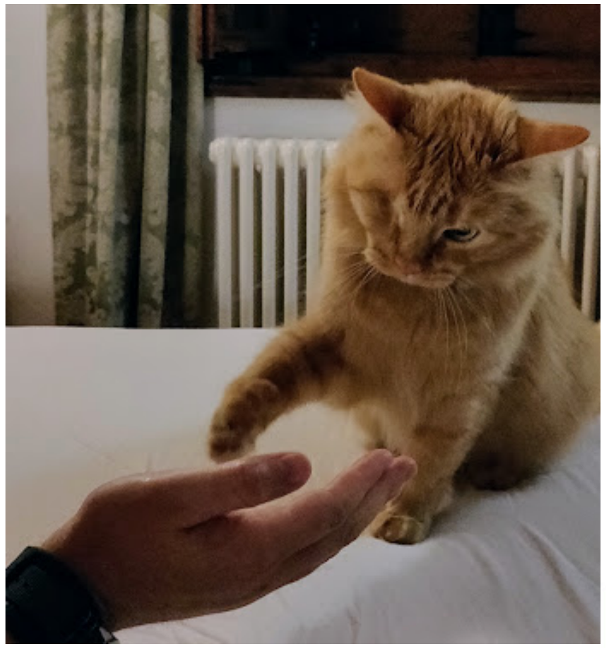
In this lecture

- We will take our first steps using *Objects* in Object-Oriented Programming
- Objects ⇔ Classes
    - Think of real objects:
        - defined by their **properties** (nouns)
        - defined by their **functionality** (verbs)
    - Extending this concept to storing and manipulating data:
        - properties = ***data members***
        - functionalities = ***member functions***

MANCHESTER 1824
The University of Manchester

A real-life example

- Class name: **cat**
    - properties = ***data members***
        - ***name**: Bob*
        - ***fur color**: ginger*
        - ***eye(s)**: yellow*
    - functionalities = ***member functions***
        - ***sleep()***
        - ***high_five_with_claws()***



MANCHESTER 1824
The University of Manchester

Our example object: a particle

- **Type**: electron
- **Electric charge**: -1
- **Mass**: 511 MeV
- **Momentum**: xxx
- **Energy**: yyy
- **β-factor**: zzz
  (for non-physics-students: this has to do with special relativity, if the particle is travelling at close to the speed of light then this factor is close to 1)

MANCHESTER 1824
The University of Manchester

Part 1: the struct

A struct

A C-inspired structure that holds properties

- A simple way to capture all the properties of an object (which actually originates in the C language) is the `struct`

- Example: consider a *particle object*.
  - We can define a struct to hold its properties (data)

```
struct particle {
  std::string type;
  double mass;
  double momentum;
  double energy;
}
```

  - Its properties are accessed using its name and the “dot” (.) notation

```
particle p; //make an object of type “particle”
p.type=“electron”;
p.mass=0.511;
```

MANCHESTER 1824
The University of Manchester

struct, functions and data

How to do something with the struct's properties

- Let's have a function to print out the data

```
void print_data(const struct particle &p) {
std::cout.precision(3); // 2 significant figures
std::cout<<"Particle: [type,m,p,E] = ["<<p.type<<","<< p.mass
<<“,"<<p.momentum<<","<<p.energy<<"]"<<std::endl; return;
}
```

- or calculate the Lorentz factor

```
double gamma(const struct particle &p) {
return p.energy/p.mass; }
```

- Those functions can't be stored in the struct (even though they logically belong to it), they can only use the struct as input
- Another disadvantage: the struct declaration can't set up or keep default values for its data members
- Enter **C++ classes**, combining data and functions

Part 2: the class

Our first class

The particle

```
#include<iostream>
#include<string>
#include<cmath>
class particle
{
public:
  std::string type;
  double mass;
  double momentum;
  double energy;
};
void print_data(const struct particle &p)
{
  std::cout.precision(3); // 2 significant figures
  std::cout<<"Particle: [type,m,p,E] = ["<<p.type<<","<< p.mass
      <<","<<p.momentum<<","<<p.energy<<"]"<<std::endl;
  return;
}
double gamma(const struct particle &p)
{
  return p.energy/p.mass;
}
```

Find this code at: Prelecture4/class1.cpp

- This code works exactly as the previous struct
- Advantages:
  - we can decide whether the data in the class is accessible and modifiable by the outside world (*public*) or not (*private*)
    - a struct is a class where all data members are public
  - we can also add functions to this class

MANCHESTER 1824 The University of Manchester

Public/private (for protected, see later lectures)

The principle of least privilege (or: C++'s GDPR)

- Public: can access all data members from "the outside"

```
class particle
{
public:

std::string type;
double mass;
double momentum;
double energy;

};
```

- Private: cannot access data members from "the outside"

```
class particle
{
private:

std::string type;
double mass;
double momentum;
double energy;

//How to access data members: 'accessor' functions
//These must be inside the class!

public:
// Function to set type of particle
void set_type(const string &ptype) {type=ptype;}
// Function to print type of particle
void print_type() {cout<<"Particle is of type "<<type<< endl;}

};
```

- **Principle of least privilege** - *elements of a class (data or functions) should be private unless proven to be needed to be public*
  - This also means that users should not need to rely on / look at the implementation of a class: use **interface** only
  - Advantage: you can change the internal behaviour of a class without affecting its users, as they only use the public interface and functions

Find code that does not compile at: Prelecture4/class2.cpp and modify it so that it does!

Access functions / accessors

How to set/get private member data

- Here, we added two public functions. This is because we wish to access these functions from outside the class.
  - When a new object is created, we use the functions to refer to that particular object.
  - We access these functions in a similar way to accessing the object's (public) data: `myObject.myFunction(myArgument)`; making clear that the function is associated with the object
  - Example for earlier code (this would go in the `main()` function):
    ```
    string type("electron"); particle p1;
    p1.set_type(type); p1.print_type();
    ```
- We only allow access to the data through access functions/accessors. We can protect our data from any undesirable consequences in designing these functions.

MANCHESTER 1824
The University of Manchester

Our particle class

Now with member functions

```
#include<iostream>
#include<string>
#include<cmath>
class particle
{
private:
  std::string type {"Ghost"};
  double mass {0.0};
  double momentum {0.0};
  double energy {0.0};
public:
// Default constructor
  particle() = default ;
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {}
  ~particle(){std::cout<<"Destroying "<<type<<std::endl;}  // Destructor
  double gamma() {return energy/mass;}
  void print_data();
};

void particle::print_data()
{
  std::cout.precision(3); // 2 significant figures
  std::cout<<"Particle: [type,m,p,E] = ["<<type<<","<< mass
      <<","<<momentum<<","<<energy<<"]"<<std::endl;
  return;
}
```

Find this code at: Prelecture4/class4.cpp

Constructors and destructors

Special member functions

- Constructors are required in C++ classes to create new objects
  - they must have the same name as the class
  - they also initialise data members
  - there is a default constructor (can "do nothing, leaving things as defaults)…

```
// Default constructor
  particle() = default ;
```

  - as well as constructors that can take arguments, same as any other function in C++
    - the syntax using : and { } assigns data members (including calculations!)

```
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {}
```

    - this is done using overloading - to be seen in later lectures
- Destructors 'destroy' the class object

```
~particle(){std::cout<<"Destroying "<<type<<std::endl;}  // Destructor
```

  - They are called when the object of that class goes out of scope, or they can be called by the user
  - Implementation particularly important if dynamical allocation of memory in the class, otherwise memory leak

Find this code at: Prelecture4/class4.cpp

MANCHESTER 1824 The University of Manchester

Functions in classes

And function prototypes inside the class

- So far, all functions were defined within the class itself (e.g. constructors), but we have not specified the details for `print_data`!
- Such a larger member function, included in full detail, can make the code look clumsy
- Solution: put implementation of such member functions outside of the class (or even in a separate file…see bonus content)
- Important note: member functions must be prototyped inside the class
- Example: define a function to print an object's data.
    - We first declare its existence inside class using function prototype
    - And define what it does outside the class, remembering the scope resolution operator ::
        - Common compiler error if that is forgotten: function cannot access data members

Find this code at: Prelecture4/class4.cpp

Refinement of classes

Return values for functions, using vectors

- If we have a large number of particle objects, we can store them into a vector
  - Then we can iterate on the vector using iterators

auto pointers:
we are keeping the best for last…

```
int main()
{
  vector<particle> particle_data;
  particle_data.push_back(particle("electron",5.11e5,1.e6));
  particle_data.push_back(particle("proton",0.938e9,3.e9));
  //vector<particle>::iterator particle_it;
  for(auto particle_it=particle_data.begin();
      particle_it<particle_data.end();
      ++particle_it){
    particle_it->print_data();
    cout<<"has Lorentz factor gamma="<<particle_it->gamma()<<endl;
  }
  return 0;
}
```

Find this code at: Prelecture4/class5.cpp

MANCHESTER 1824
The University of Manchester

Refinement of classes

Return values for functions, using vectors

- This code used a few refinements. If we have a large number of particles, it is much easier to use a vector to contain all of them
- We can then use iterators over the data to output all the information
- Here we use the arrow -> operator to get a class member of a dereferenced pointer `particle_it->print_data();`
  - `particle_it->print_data()` is the same as `(*particle_it).print_data()`, but easier to read.
- Remember, the iterator `particle_it` is like a pointer!

Find this code at: Prelecture4/class5.cpp

Summary of classes so far

“Buzzword summary”

- A **class** is the set of rules used to **define our C++ objects**. It specifies which types of **data** and **functions** are created and their **scope** (private or public)
- An **object** is an **instance of a class**. Each object will have specified its own set of data (values of the data members).
- A **member** refers to either **data** or a **function** belonging to a particular class, e.g. a constructor will be a member function. Member functions are sometimes called **methods**
- A **constructor** is a special function called when a class is instantiated, usually to **initialise** an object’s member data. If not user generated, generated by compiler.
- A **destructor** is the function called when an object is **destroyed** (usually automatically when exiting a function; we say “the object goes **out of scope**”– this happens when we can no longer access the object). If not user generated, generated by compiler.

MANCHESTER 1824
The University of Manchester

Bonus content

[not in assignments/projects, unless you feel brave]

In Visual Studio / Visual Studio Code we always deal with one .cpp file at a time, b
ut how do we deal with separate headers and implementations?

Proper answer: via Makefile / CMake…
Working answer: see next slide

Headers and implementation

A reminder

- C++ classes are usually split into two different parts, often in different files
  - *Header* (file extension: .h, .hpp) = where things are defined
    - Think of it as the index of a book
    - Usually headers contain *interfaces* that help you get an overview of what something (e.g a function) does, without the clutter of the full implementation
  - *Implementation* (file extension: .cxx, .cpp) = where things are implemented
    - Think of it as the book content
    - The implementation of the class's functions go here

MANCHESTER 1824 The University of Manchester

Our particle class, split in .cxx and .h

Compilation will have to change slightly…

```
#ifndef PARTICLE_H
#define PARTICLE_H

#include<iostream>
#include<string>
#include<cmath>
class particle
{
private:
  std::string type {"Ghost"};
  double mass {0.0};
  double momentum {0.0};
  double energy {0.0};
public:
// Default constructor
  particle() = default ;
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {};
  ~particle(){std::cout<<"Destroying "<<type<<std::endl;}  // Destructor (in-line)
  double gamma() {return energy/mass;} // One-line functions are OK in-line
  void print_data();
};

#endif
```

Find this code at: Prelecture4/particle.h

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

void particle::print_data()
{
  std::cout.precision(3); // 2 significant figures
  std::cout<<"Particle: [type,m,p,E] = ["<<type<<","<< mass
    <<","<<momentum<<","<<energy<<"]"<<std::endl;
  return;
}
```

Find this code at: Prelecture4/particle.cpp

```
#include<iostream>
#include<string>
#include<cmath>
#include "particle.h"

int main()
{
  // Set values for the two particles
  particle electron("electron",5.11e5,1.e6);
  particle proton("proton",0.938e9,3.e9);
  // Print out details
  electron.print_data();
  proton.print_data();
  // Calculate Lorentz factors
  std::cout.precision(2);
  std::cout<<"Particle 1 has Lorentz factor gamma="
    <<electron.gamma()<<std::endl;
  std::cout<<"Particle 2 has Lorentz factor gamma="
    <<proton.gamma()<<std::endl;
  return 0;
}
```

Find this code at: Prelecture4/main_class4.cpp

The University of Manchester

How do we compile this?

Make sure all .cpp are compiled,
the .h are included automatically

- [yourCompilerDir]/g++-11 -fdiagnostics-color=always -g [yourdir]/main_class4.cpp [yourdir]/particle.cpp -o [yourdir]/main_class4