Object-Oriented Programming in C++

Pre-Lecture 4

Prof Niels Walet (Niels.Walet@manchester.ac.uk)

Room 7.07, Schuster Building

Version: February 15, 2020

Prelecture 4

Outline

- From structures to classes
- Basic features of the C++ class
  - public and private data
  - access functions
  - constructors and destructors
  - member functions outside class
  - function return types
- Vectors and objects

Classes

What is an object?

- Object-Oriented Programming is based on the concept of objects
- Think of real objects (e.g. apple, pencil, car):
  - they are defined by their properties (nouns)
  - they are also defined by their functionality (verbs)
- We can extend this concept to how we store and manipulate data
- 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;
};
```

Classes

Structures

We can then declare a structure for every particle and also define their associated data, using "dot" notation to access internal data members, e.g.

```
// Create 2 particles
particle electron;
electron.type="electron";
electron.mass=5.11e5;
electron.momentum=1.e6;
electron.energy=sqrt(electron.mass*electron.mass+electron.
  momentum*electron.momentum);
particle proton;
proton.type="proton";
proton.mass=0.938e9;
proton.momentum=3.e9;
proton.energy=sqrt(proton.mass*proton.mass+proton.momentum*
  proton.momentum);
```

Classes

Structures

Now we probably want to do something with our data so we can write some functions, e.g. 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 to calculate the Lorentz factor

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

Classes: Structures full code

```
// PL4/struct.cpp
// An example using a struct as a class
// Niels Walet, last updated 04/12/2019
#include<iostream>
#include<string>
#include<cmath>
struct particle
{
  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;
}
int main()
{
  // Create 2 particles
  particle electron;
  electron.type="electron";
  electron.mass=5.11e5;
  electron.momentum=1.e6;
  electron.energy=sqrt(electron.mass*electron.mass+electron.momentum*electron.momentum);
  particle proton;
  proton.type="proton";
  proton.mass=0.938e9;
  proton.momentum=3.e9;
  proton.energy=sqrt(proton.mass*proton.mass+proton.momentum*proton.momentum);
  // Print out details
  print_data(electron);
  print_data(proton);
  // Calculate Lorentz factors
  std::cout.precision(2);
  std::cout<<"Particle 1 has Lorentz factor gamma="
    <<gamma(electron)<<std::endl;
  std::cout<<"Particle 2 has Lorentz factor gamma="
    <<gamma(proton)<<std::endl;
  return 0;
}
```

Listing 1 : selection of PL4/struct.cpp

Classes

Structures: why not?

- Program outputs

```
Particle: [type,m,p,E] = [electron,5.11e+05,1e+06,1.12e+06]
Particle: [type,m,p,E] = [proton,9.38e+08,3e+09,3.14e+09]
Particle 1 has Lorentz factor gamma=2.2
Particle 2 has Lorentz factor gamma=3.4
```

- Some disadvantages of this method:
- Data for each structure must be defined outside of the structure declaration itself. Makes it easy to forget to set a particular value
  - Data is open to being altered or corrupted
  - Functions acting on the data are separate and require the data to be passed as a parameter
- What if we could combine the data and their functions in one structure?
- That is the key of object-oriented programming
- This is exactly what C++ offers in the form of a `class`

Classes

The C++ Class

- Look at a very basic class based on our particle structure

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

Listing 2 : selection of PL4/class1.cpp

- This code works identically.
- C++ definition: A `struct` is just a special `class` where all members are public
- In our `main` function, two objects (`p1` and `p2`) are created of class `particle` (known as instances of the class)
- Notice the `public:` keyword. This instructs the compiler that everything declared below it ***can*** be accessed from outside the class
- Anything not declared public will be invisible outside the class (is `private:`)
- There is a third option `protected:` will be discussed in later weeks.

Classes

The C++ Class: public and private data

- Now let us modify the class to

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

Listing 3 : selection of PL4/class2a.cpp

- Data is now `private:` to the class
- If we try to compile the code, we will now get a large number of errors as we are accessing private members outside the class!
- This is not a problem, but actually an advantage: it allows us to keep data secure
- But we still need a way to access private data!

Classes

The C++ Class: access functions

- Apart from the public nature of the data, the main distinction between a `class` and a `struct` is that a `class` can (*and almost always will*) include functions to manipulate its data
- Let us define one to set our type and one to print its value

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

Listing 4 : selection of PL4/class3.cpp

Classes

The C++ Class: access functions

- 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

```
string type("electron"); particle p1; p1.settype(type); p1.printtype();
```

- We only allow access to the data through access functions. We can protect our data from any undesirable consequences in designing these functions.

Classes

The C++ Class: constructors and destructors

- Setting all variables like this is rather clumsy - there is a better way!
- Two special functions called the constructor and destructor can be defined
- A constructor is a function that is automatically called when a new object is created
- Even more powerful when combines with default values.
- Can be overloaded, to give different actions dependent on arguments
- It has the same name as the class itself and has no return type
- Its main use is to set values for the object's member data
- Similarly, a destructor function is used to destroy an object's member data

Classes

The C++ Class: constructors and destructors

Our class with constructors and a destructor

```
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();
};
```

Listing 5 : selection of PL4/class4.cpp

Classes

The C++ Class: constructors and destructors

- Let's look at the first constructor

```
particle() = default ;
```

- This is our default constructor and will be called when we declare an object with no parameters, e.g. `particle p1; // calls our default constructor`
- In this case, the particle type is Ghost and its other data are set to zero.
- The destructor is called when a function exits (including `main`, i.e. end of program)

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

- Really only useful when dynamically allocating memory (if we use `new` in the constructor, `delete` would go here); but good practice to include one!

Classes

The C++ Class: constructors and destructors

- Note that we actually define two different constructors (making use of overloading!)
- The second is a 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 constructor allows us to pass values for our data when creating our object (and also computes the energy)

Classes

The C++ Class: member functions outside 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...)
- 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

```
void print_data();
```

Classes

The C++ Class: member functions outside class

- Now outside the class we define the function itself

```
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;
```

- The names of functions defined outside are preceded with the class name and the scope resolution operator ::
- This tells the compiler which class the function actually belongs to
- Without it the compiler would assume the function to be an ordinary function (not a member function of `particle`), and then it cannot act on private members....

Classes

The C++ Class: function return types

- Our member functions have not returned anything but this is possible as it is with normal functions
- As a counter example: define a function that returns Lorentz factor γ
- In the class we would write

```
double gamma() {return energy/mass;}
```

- Then in the main program we can use

```
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;
```

Listing 6 : selection of PL4/class4.cpp

Classes

Final refinement: using vectors

Look at the following piece of code

```
{
  std::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();
    std::cout<<"has Lorentz factor gamma="<<particle_it->gamma()<<std::endl;
  }
  return 0;
}
```

Listing 7 : selection of PL4/class5.cpp

Classes

Final refinement: using vectors

- This uses a few refinements. If we have a large number of particles, it is much easier to use a vector to contain all of them (or a list?)
- 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()` is the same as `(*particle_it).print_data()`, but easier to read.
- Remember, the iterator `particle_it` is like a pointer!

Classes

Buzzword Summary

- A class is the set of rules used to define our 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 its *own* set of data.
- 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 initialize 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.

The END