# Object-Oriented Programming in C++

## Pre-Lecture 7: Inheritance

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

Room 7.07, Schuster Building

February 29, 2020

# Prelecture 7

## Outline

We will continue our study of core elements of Object Oriented Programming; this time we will look at **Inheritance**

- More on OOP
- A simple example
- Access rules for inherited members
- Constructors in derived classes
- Overriding functions
- Multiple inheritance

# More on OOP

# Object Oriented Programming

- It is hopefully becoming clearer what OOP is about
- Traditional (procedural) programming focuses on what you do with data:
  - A program is primarily a procedure to process data
  - Data are usually input at beginning and output at end
- OOP turns this around - think instead: how do I want to organise my data and what operations do I allow to be performed on them?
- Data are now the primary concern
- A natural consequence: OOP is very modular. Classes are like mini-programs and (non-member) functions outside the class are used to implement these processes (through objects)

# Object Oriented Programming

Object oriented programming is based on 3 important concepts

1. Encapsulation: data and the methods that act on the data are defined together within a class. These data and methods are used via objects and the programmer has full control over how data and functions are accessed (security).
2. Inheritance: there is a way to define relationships between classes. A derived class can inherit the members (data and methods) of a parent, base class. The typical buzzword is “specialisation”
3. Polymorphism: the ability to create multiple derived classes with the same functions but with different actions. *One interface, multiple methods*

- We have been using the first so far - essentially defining isolated classes and using them through their instances (objects)
- In larger programs, we might want to create class hierarchies - this is where we use inheritance
- To use polymorphism we need to first understand how to implement inheritance
- Will discuss polymorphism next week!

# This week's example

# Inheritance

**Simple example**

Here is a code similar to the `galaxy` class from Assignment 4

```
4  #include<iostream>
5  #include<string>
6  class celestial_object
7  {
8  private:
9    std::string name;
10   double mass, distance, luminosity; // for compactness
11 public:
12   celestial_object() :
13     name{"no-name"}, mass{}, distance{}, luminosity{} {}
14   celestial_object(const std::string nm, const double l, const double m, const double d) :
15     name{nm}, mass{m}, distance{d}, luminosity{l} {}
16   ~celestial_object(){}
17   std::string celestial_object_name() {return name;}
18   friend std::ostream& operator<<(std::ostream&, const celestial_object& );
19 };
20 std::ostream& operator<<(std::ostream& o, const celestial_object& co)
21 {o <<" object "<<co.name <<": "<<std::endl
22    <<" mass "<<co.mass <<" Msun,"
23    <<" luminosity "<<co.luminosity <<" ,"
24    <<" distance (z) "<<co.distance <<std::endl;
25     return o;}
26 int main() {
27   celestial_object LMC{"Large Magellanic Cloud",-1,2e10,0.000875};
28   std::cout <<LMC;
29   return 0;
30 }
```

**Listing 1 :** selection of PL7/celestialobject.cpp

# Inheritance

## Simple example

- which produces

```
object Large Magellanic Cloud:
mass 2e+10 Msun, luminosity -1 , distance (z) 0.000875
```

- Now let's say we want to create a class for galaxies
- Inheritance allows us to do this: we can create a new class (`galaxy`) that inherits the members of the `celestialobject` class as well as having members of its own
- In C++, the `celestial_object` class is known as the base class and the `galaxy` class the derived class
- Let's see how this works by example

# Inheritance

## The galaxy class

```
20 class galaxy: public celestial_object {
21 private:
22   std::string hubble_type;
23 public:
24   galaxy() :
25     celestial_object{}, hubble_type{"Sc"} {}
26   galaxy(const std::string nm, const double l, const double m, const double d,
27       const std::string ht) :
28     celestial_object{nm,l,m,d}, hubble_type{ht} {}
29   ~galaxy(){}
30   friend std::ostream& operator<<(std::ostream& , const galaxy& );
31 };
32 std::ostream& operator<<(std::ostream& o, const galaxy& gx)
33 {o <<" galaxy "<<gx.getname() <<": "<<std::endl
34    <<" Hubble type "<<gx.hubble_type<<std::endl;
35     return o;
36 }
37 std::ostream& operator<<(std::ostream& o, const celestial_object& co)
38 {o <<" mass "<<co.mass <<" Msun,"
39    <<" luminosity "<<co.luminosity <<" ,"
40    <<" distance (z) "<<co.distance <<std::endl;
41     return o;
42 }
```

**Listing 2 :** selection of PL7/galaxy.cpp

# Inheritance

## The galaxy class: output

This produces the output

```
galaxy Large Magellanic Cloud:
Hubble type SBm
```

# Inheritance

## a simple example

- The most significant difference to note is

```
class galaxy: public celestial_object {
```

- This specifies that `galaxy` is a derived class of `celestial_object`
- The `public` qualifier sets the access level for inherited members of the base class. More about this later.
- Our `galaxy` class has its own members but can also access members of `celestial_object` (as it inherits these as a derived class)
- This is demonstrated in the operator `<<` where we call `getname()`, a member of `celestial_object`

# Inheritance and Access

# Inheritance

## access rules for inherited members

- In the previous example, why did we have to make the members of `celestial_object` `public`?

```
7  class celestial_object
8  {
9  private:
10   std::string name;
```

Listing 3 : selection of PL7/galaxywrong.cpp

- If we try this, we would get an error!
- `name` is a private member of `celestial_object`
- Private **really** does mean private - even derived classes cannot access private data from the base class!
- We can use the `getname()` function, which is `public` so is accessible anywhere
- We could instead make the base class data public as before but would lose protection from outside class
- C++ provides a better way to deal with this problem...

# Inheritance

## access rules for inherited members

- There is a 3rd access specifier for class members: protected
- Remember: private means access is granted only to other class members and friends
- For protected members this access is extended to derived classes
- Let's modify the base class as follows

```
6  class celestial_object
7  {
8  protected:
9    std::string name;
10   double mass, distance, luminosity;
```

Listing 4 : selection of PL7/galaxy2.cpp

- We can now access the information in galaxy directly

# Inheritance

## access rules for inherited members

- In the previous example we modified the access level of data members in our base class
- What access level is given to the inherited members in the derived class?
- This is set using the access specifier in the derived class definition

```
class galaxy: public celestial_object
{
private:
```

- When using `public`, the access levels for our inherited (non-private) members are the same as they were in the base class
- So, e.g. protected members of the base class are inherited as protected members of the derived class

# Inheritance

## access rules for inherited members

- Best to summarize rules in a table of inheritance in table below

| | | **Derived class specifier** | | |
|---|---|---|---|---|
| | | Private | Protected | Public |
| **Base class specifier** | Private | Not inherited | Not inherited | Not Inherited |
| | Protected | Private | Protected | Protected |
| | Public | Private | Protected | Public |

This shows the access to inherited members, based on base class specifier and the inheritance specification.

- Derived class access specifier is a request to set an access level
- Base class access specifier sets maximum access level
- Private members are the exception: never accessible outside a class

# Constructors in derived classes

# Inheritance

## constructors in derived classes

- When we instantiated the LMC as a `galaxy` object by writing

```
galaxy LMC{"Large Magellanic Cloud",-1,2e10,0.000875,"SBm↩
 "};
```

we invoked its parameterized constructor

```
galaxy(const std::string nm, const double l, const double↩
  m, const double d,
   const std::string ht) :
  celestial_object{nm,l,m,d}, hubble_type{ht} {}
```

- But `galaxy` is a derived class of `celestial_object`.
- If we don't specify anything, the default constructor for `celestial_object` is invoked whenever we create a `galaxy`.
- But it is possible to directly invoke `celestial_object` 's parameterized constructor as shown.

# Inheritance

## constructors in derived classes

- We have used both base class and derived class constructors to create our object.
- What order do the constructors (and destructors) get called in?
- You can add print statements to the constructors and destructors to reveal the answer...
- The base class constructor is called first, followed by the derived class constructor.
- The reverse must happen for the destructor: derived then base.
- This makes sense: we need to invoke base class constructor first so that there is data to be inherited by the instance of derived class!

# Overriding functions

# Inheritance

## overriding functions

- There is another feature of inheritance we can exploit to make our code look simpler.
- C++ allows us to define functions in base and derived class with the same name - this is called function overriding.
- Note: function overriding only makes sense when done in a separate (derived) class.
- We are re-writing the operations of a base class function to suit the specific needs of the derived class function.
- This is different from overloading, which allows more than one function with the same name to be defined within the same class.
- Overriding requires both functions to have *exactly the same* parameter list - overloading functions requires *different* parameter lists.
- Summary:
  - You overload functions within the same class, so that they can take different arguments but perform a similar task
  - You override functions of a base class to re-define their functionality in the derived class, using identical parameter lists

# Multiple inheritance

# Inheritance

## multiple inheritance

- C++ does not just restrict us to single inheritance: we can inherit from multiple sources and have multiple levels of inheritance to create a whole class hierarchy if we like.
- Each class in the hierarchy inherits members and extends their functionality to suit its own needs.
- Sometimes we will want to inherit members from more than one class - syntax is `class C : public A, public B`.
- Here, class `C` inherits all members of classes `A` and `B`.
- For `C`'s constructor we would write

```
C(const double Axin, const double Bxin, const double Cxin↩
 ) :
  A{Axin}, B{Bxin}, Cx{Cxin} {}
```

which calls the constructors for `A` and `B`.

# Inheritance

**multiple inheritance (code multiple.cpp)**

```
5  class A {
6  protected:
7    double Ax;
8  public:
9    A(const double Axin) : Ax{Axin} {}
10   void show(){std::cout<<"Ax="<<Ax<<std::endl;}
11 };
12 class B {
13 protected:
14   double Bx;
15 public:
16   B(const double Bxin) : Bx{Bxin} {}
17   void show(){std::cout<<"Bx="<<Bx<<std::endl;}
18 };
19 class C : public A, public B {  // Single derived class
20   double Cx;
21 public:
22   C(const double Axin, const double Bxin, const double Cxin) :
23     A{Axin}, B{Bxin}, Cx{Cxin} {}
24   void show(){std::cout<<"Ax,Bx,Cx = "<<Ax<<" "<<Bx<<" "<<Cx<<std::endl;}
25 };
26 int main()
27 {
28   C my_object{0.1,0.2,0.3};
29   my_object.show();
30   return 0;
31 }
```

Listing 5 : selection of PL7/multiple.cpp

# Inheritance

**multiple inheritance**

- We might want to create more than one level of inheritance (e.g. a linear chain)
- Example: create a `neutron_star`, derived from `star` (derived from `celestial_object`)

```
23 {
24 protected:
25   std::string spectral_class {"None"};
26 public:
27   star() :
28     celestial_object{},spectral_class{"None"} {}
29   star(const std::string nm, const double l, const double m, const double d,
30      const std::string sc) :
31     celestial_object{nm,l,m,d}, spectral_class{sc}{}
32   ~star(){}
33   friend std::ostream& operator<<(std::ostream&, const star& );
34 };
35 class neutron_star: public star
36 {
37 protected:
38   double radius ; // radius in km
39 public:
40   neutron_star() :
41     star{}, radius{} {spectral_class="pulsar";}
42   neutron_star(const std::string nm, const double l, const double m, const double d,
43            const double r) :
44     star{nm, l, m, d, "pulsar"}, radius{r} {}
45   ~neutron_star(){}
46   friend std::ostream& operator<<(std::ostream& , const neutron_star& );
47 };
48 std::ostream& operator<<(std::ostream& o, const neutron_star& st)
```

Listing 6 : selection of PL7/neutronstar.cpp

# Inheritance

## multiple inheritance

- Then in main we can write

```
70 {
71   neutron_star crab ("crab",  0, 1.4, parsectoz(2200),1.437815e-5) ;
72   std::cout << crab;
73   return 0;
74 }
```

which produces

```
neutron star crab:
radius 1.43782e-05 Rsun
spectral class pulsar
mass 1.4 Msun, luminosity 0 , distance (z) 5.214e-07
```

- Our new `neutron_star` object inherits members from both `star` and `celestial_object`!

# Specialisation: When to inherit

# Inheritance

## specialisation

- Inheritance is often used incorrectly (was in previous versions of this course!)
- In the correct way, it represents specialisation (is_a), i.e., a a derived class is_a base class
- To be contrasted with containment (has_a) i.e., a composite class that has a member that is an object of another class
- Similar relation is “belongs_to”, which is normally lumped together with “has_a”!
- Example could be `galaxy` contains `solar_system`; a `solar_system` contains `stars` and `planets`, all of which would be at the same level of inheritance from celestial_object; they **should not be inherited!**

# Summary

# Inheritance

## Summary and buzzwords

- Inheritance in OOP allows members of a class to be incorporated within another class.
- It extends the functionality of the existing class (known as the base class) by adding new members in the derived class.
- Not all members need to be inherited: this is controlled by the access specifiers.
- A new access specifier, protected, allows base class members to be accessed in the derived class, but not outside the class hierarchy.
- Derived class constructors need to call the base class constructors - the latter are called first and the related destructor last.
- Function overriding allows base class member function to be re-defined in derived class (otherwise they are inherited).
- Classes can inherit multiple base classes and one can create more than one level of inheritance.