Introduction to Programming and Computing for Scientists

Lecture 6
Object oriented programming in C++: classes, inheritance, polymorphism

Katja Mankinen

thanks to Vytautas Vislavicius, Oleksandr Viazlo, Anders Floderus

Lund University

3 October 2018

Today's goals

You will learn

- to group data into classes
- to declare, define and access class members
- how to decrease your workload by using inheritance and polymorphism
- about a good coding style

From procedural oriented programming...

What should the program do next?

1. split your problem into a set of tasks and subtasks
2. make functions for the tasks
3. tell the computer to perform the tasks in sequence

... to Object Oriented Programming (OOP)

Object = data + functions

One way to think about a car: it consists of wheels, motor, doors, seats, windows, ... Another way: it can move, speed up, slow down, park....

- Encapsulation: binding together related data and functions, and keeping both safe from outside interference
- Inheritance: allowing to define a class in terms of another class, and being able to reuse the code
- Polymorphism: allowing objects to act differently in different situations with the same syntax

Benefits:

- Modularity, easier software design, maintenance and troubleshooting: the car object broke down $\rightarrow$ problem must be in that class
- Reusability through inheritance
- Flexibility through polymorphism

C++ class

- A class is a container for **data** and **functions**.
- Member variables (data members): variables in a class
- Member functions (methods): functions in a class.
- An instances of the class are called an ***objects***. You can create many instances of a class

	Student	Dog	Circle
class name (identifier)			
data members	name grade	name breed color	radius color
member functions	getName() getGrade()	getName() getColor()	setRadius() getArea()

Instances of the Dog class

	Dog RinTinTin	Dog Laika	Dog Snoopy
class name			
data members	name="Rin Tin Tin" breed="German Shepherd"	name="Laika" breed="mixed"	name="Snoopy" breed="beagle" color="white/black"
member functions	getName() getBreed()	getName() getBreed()	getName() getBreed() getColor()

Syntax

```
class Dog { //class name

public: // Access: the following members (data or functions) are accesible and available to all in the
      system
    Dog (int initialAge); // constructor with default values for data members
    ~Dog(); // destructor (does nothing in this case)

    //member functions:
    int GetAge() {return itsAge;} //inline*! "Getter": to allow others to read the value of a private data
      member
    void SetAge(int age) {itsAge = age;} //inline! "Setter": to allow classes to modify the value of a
      private data member
    void Woof() {cout << "Woof!\n"; }

private: // Access: the following members are accessible and available within this class only
    int itsAge; // data members (variables)

};

/// inline functions: the compiler places a copy of the code of the function at each point where the
      function is called at compile time -> any change to an inline function requires recompilation
// more complex methods: define the functions outside the class (in .cpp!) using the scope operator (::)
```

Dog class declaration and implementation

- Declaration (.h files): list of functions and variables
- Definition (.cpp files): implementation of functions

dog.h:

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

class Dog { // class name

public:
    Dog (int initialAge);
    ~Dog();
    int GetAge() {return itsAge;} // member functions (methods)
    void SetAge(int age) {itsAge = age;}
    void Woof() {cout << "Woof!\n"; }
private:
    int itsAge; // data member (variable)

};
```

Dog class declaration and implementation

dog.cpp:

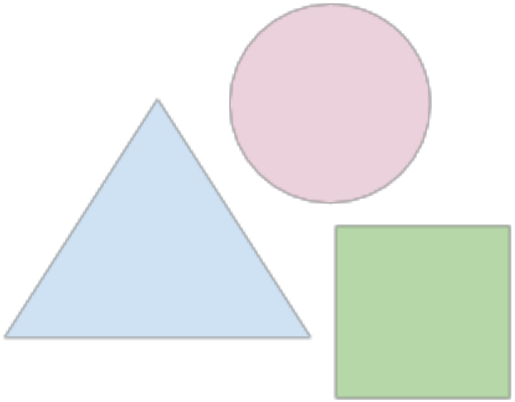
```
#include "dog.h" //be sure to include the header file!

Dog::Dog(int initialAge) //constructor. Scope operator (::) to define a member of a class outside the
      class. A constructor is called every time a new object is created.
{
   itsAge = initialAge;
}

Dog::~Dog() //destructor, takes no action
{
}

//Create a dog, set its age, have it "woof", tell us its age, have it "woof" again, and change its age.

int main()
{
    // Declare and construct an instance Snoopy of the class Dog, and set initialAge to 5
    Dog Snoopy(5);
    // or: Dog Snoopy = Dog(5);
    // Invoke a data member or member function: instanceName.memberName
    Snoopy.Woof();
    cout << "We created a dog called Snoopy and it is " << Snoopy.GetAge() << " years old!\n";
    Snoopy.Woof();
    Snoopy.SetAge(7);
    cout << "Now it is " << Snoopy.GetAge() << " years old.\n";
    //Snoopy.itsAge = 5; //error: int Dog::itsAge is private
    return 0;
}
```

C++ class: real life physics example

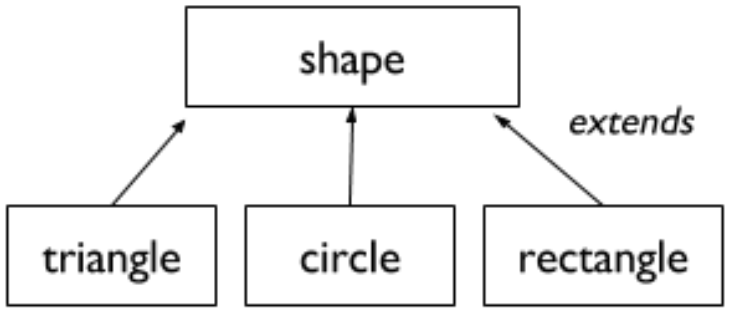
```
class particle { //Here I declare a class of type "particle"
  public:

  particle(int id, double pt); //Constructor. Call to create an instance of the class.
  ~particle(); //Destructor. Gets called when an instance of the class is destroyed.

  double pt();
  double m();
  double e();

  private: //This class has no private members
};

particle::particle(int id, double pt) { //Simply store the user supplied values
  id = 11; // 11 is electron by convention of Particle Data Group
  pt = 20; // in GeV
}

particle::~particle() {
  //There are no special tasks to perform when destroying a set of particles
}
```

Non-simplified ROOT version: TParticle.h

Inheritance: IS-A



triangle	circle	rectangle
base height	base height	base height
area() getBase() getHeight()	area() getRadius()	area() getBase() getHeight()



- What do triangle, circle and rectangle have in common? What is special in triangle (or circle or rectangle)?
- Inheritance is a way to share characteristics among similar types.
- A subtype inherits characteristics of its base type.

Inheritance

```
class triangle { //Let's take this chance to observe some good programming conventions
  public:
  triangle(double base = 0., double height = 0.);
  ~triangle();
  double area();
  double getBase(); //Getters and setters are used to handle private info
  double getHeight();
  void setBase(double base); //This function should make sure that the base is positive
  void setHeight(double height);

  private: //All member varibles should in general be private to facilitate encapsulation
  double base_; //Name private members with an underscore to avoid shadowing (= a variable declared within
      a certain scope has the same name as a variable declared in an outer scope)
  double height_;
};
```

```
class rectangle {
  public:
  rectangle(double base = 0., double height = 0.);
  ~rectangle();
  double area();
  double getBase();
  double getHeight();
  void setBase(double base);
  void setHeight(double height);

  private:
  double base_;
  double height_;
};
```

Inheritance

- We'd have to repeat a lot of code to write the `triangle` and `rectangle`.
- Inheritance simplifies this immensely. Both `triangle` and `rectangle` are really special cases of something more general. Let's call it a `shape`.

```
#include <cmath> //For fabs

class shape { //This class has the common characteristics of triangles and rectangles
  public:
  shape(double base = 0., double height = 0.);
  ~shape();
  double getBase() { return base_; } //Simple functions can be defined here in the header
  double getHeight() { return height_; }
  void setBase(double base) { base_ = fabs(base); }
  void setHeight(double height) { height_ = fabs(height); }

  protected: //Protected members can be accessed by this and whatever inherits from this
  double base_;
  double height_;
};
```

```
#include "shape.h"

shape::shape(double base, double height) {
  setBase(base); //Let's call these functions rather than duplicate the code
  setHeight(height);
}

shape::~shape() { }
```

Inheritance

- Now we'll make `triangle` inherit from `shape`. The only new code we have to write is whatever is specific to `triangle` (in this case the `area` function).

```
#include "shape.h" //Include the class that we want to inherit from

// class derivedClass : access_mode baseClass. Note only one colon (:) !
class triangle : public shape { //Triangle inherits from shape, access levels unchanged
  public:
  triangle(double base = 0., double height = 0.); //A ctor/dtor must still be provided
  ~triangle();
  double area() { return base_*height_/2.; } //This function is specific to triangle
};
```

```
#include "triangle.h"

// note 2 colons (::)!
triangle::triangle(double base, double height) : shape(base, height) {
  //The first (and in this case only) thing to do is initialize the parent object
}

triangle::~triangle() {
  //At the end of this destructor, the parent destructor is called automatically
}
```

- `shape` is the 'base' or 'parent' class, while `triangle` is the 'derived' class.
- When an object is created, the base part should always be constructed first. Destruction follows the opposite order - The base should be destroyed last.

Polymorphism

- Let's pretend the area of a shape is so crucial, we have functions to check it.

```
bool isBigEnough(rectangle& obj) {
  return obj.area() > 10.;
}

bool isBigEnough(triangle& obj) {
  return obj.area() > 10.;
}
```

- This is clunky because every new kind of shape needs its own function.
- Ideally, we'd like to have a single function that works with any kind of shape.

```
bool isBigEnough(shape& obj) {
  return obj.area() > 10.;
}
```

- This function will happily accept `triangle` and `reclangle` objects as arguments. They inherit from `shape`, so they *are* shapes.
- The problem is that `shape` does not declare an `area` function, so the compiler complains. We can try adding one to the class definition.

```
double area() { return 0.: } //Let's add this to shape.h
```

Polymorphism

- This small test program doesn't print the answer we want. The problem is of course that `isBigEnough` calls the `area` function in `shape`, which returns 0.

```
#include <iostream> //For cout
#include "triangle.h" //Both triangle.h and rectangle.h include shape.h
#include "rectangle.h" //I added #ifndef macros in shape.h, so it isn't doubly defined
using namespace std;

bool isBigEnough(shape& obj) { //A shape is big enough if its area is greater than 10
  return obj.area() > 10.;
}

int main() { //Create some shapes, print their area and see if they're big enough
  triangle tri(10., 10.);
  cout << "Triangle with area " << tri.area();
  if(isBigEnough(tri)) cout << " is big enough!" << endl;
  else cout << " is NOT big enough!" << endl;

  rectangle rec(5., 5.);
  cout << "Rectangle with area " << rec.area();
  if(isBigEnough(rec)) cout << " is big enough!" << endl;
  else cout << " is NOT big enough!" << endl;

  return 0;
}
```

```
Triangle with area 50 is NOT big enough!
Rectangle with area 25 is NOT big enough!
```

Polymorphism

- We can get the desired behavior by making the `area` function `virtual`.

```
virtual double area() { return 0.: } //Change the area function in shape.h to this
```

- When a function is `virtual`, derived classes are allowed to override it. If `isBigEnough` receives a `triangle`, the `triangle` version of `area` is called.
- The call to `area` thus behaves differently depending on whether the `shape` is a `triangle` or `rectangle`. Such a function is said to be *polymorphic*.
- After `area` is declared `virtual`, the test program gives the expected output.

```
Triangle with area 50 is big enough!
Rectangle with area 25 is big enough!
```

- You seldom need to tell a function what kind of `shape` it is dealing with at compile time. The proper behavior is achieved through polymorphism.
- Good use of inheritance and polymorphism will make code much easier to read, maintain and extend. One of the great strengths of OOP!

Polymorphism

- Virtual functions work the same way with pointers as with references.

```
bool isBigEnough(shape* obj) {
  return obj->area() > 10.; //Works as expected - area is called in the derived class
}
```

- Pointers mesh well with inheritance, because a pointer of type base class can point to a derived class. Remember, triangles and rectangles *are* shapes.

```
shape* shapePtr = new shape(10, 10); //Nothing new - A shape pointer pointing to a shape
isBigEnough(shapePtr); //The area function in shape is called, so this returns false

shape* triPtr = new triangle(10, 10); //Perfectly OK - Any triangle is also a shape
isBigEnough(triPtr); //The area function in triangle is called, so this returns true

triangle* illegalPtr = new shape(10, 10); //Not OK - A shape isn't necessarily a triangle
```

- If a class is handled polymorphically, it should have a `virtual` destructor.

```
virtual ~shape(); //Make the destructor virtual in shape.h, or you're in for trouble
```

```
shape* triPtr = new triangle(10, 10); //A shape pointer that points to a triangle
delete triPtr; //Calls ~shape(). Make it virtual so the triangle part is destroyed too!
```

Polymorphism

- We made `shape` return an area of zero, but in reality it is undefined.
- This is a valid concern. In fact, it doesn't make sense to instantiate a `shape` in the first place. Only `triangle` and `rectangle` are meaningful objects.
- To avoid this logical inconsistency, make the `area` function pure `virtual` in `shape`. A class that has a pure `virtual` function can not be instantiated.
- A class that contains at least one pure `virtual` function is called *abstract*.

```
virtual double area() = 0; //Put this in shape.h to make area a pure virtual function
```

- Any class that inherits from `shape` must now either implement `area` or be abstract itself. This ensures that no one can misuse our `shape` class.

```
triangle t(10, 10); //No problem - A triangle is a meaningful object
shape s(10, 10); //This will not compile. Shape is abstract and can not be instantiated!
```

The const keyword

- Declare a variable as const when you want to be certain that it is never modified. Trying to do so then results in a compile time error.

```
const int var = 10; //Remember to initialize at declaration time. Const variables can't be modified later
var = 20; //Nope! Because var is const, this results in a compile time error
```

- The const keyword acts on whatever word or symbol is to its immediate left. If there is nothing to its left, it acts on whatever is to its right instead.

```
int const var = 10; //These two lines are completely equivalent
const int var = 10; //Pick one usage and be consistent
```

- A const pointer (`int* const p`) must point to the same variable forever.
- A pointer to const (`int const* p`) can't be used to assign to a variable.

```
int foo = 10;
int bar = 20;

int* const p1 = &foo; //p1 is a constant pointer to int (so the pointer is const but not foo)
*p1 = 30; //No problem
p1 = &bar; //Error! p1 must forever point to foo

int const* p2 = &foo; //p2 points to a constant int (so foo is const but not the pointer itself)
*p2 = 30; //Error! foo can't be assigned to via p2. Assigning via e.g. p1 is still fine, though.
p2 = &bar; //No problem
```

The const keyword

- const variables and objects are picky about how they are used. They will only work with functions that have promised in advance not to change them.
- A function can promise not to change an argument by declaring it as const.
- This is relevant only when passing arguments by reference or pointer. When an argument is passed by value, any modifications are local to the function.

```
#include <iostream>

using namespace std;

void passByValue(int foo) { cout << foo << endl; } //None of these functions actually modify foo
void passByPtr(int* foo) { cout << *foo << endl; }
void passByConstPtr(int const* foo) { cout << *foo << endl; } //But this one explicitly promises not to!

int main() {
  const int foo = 10;
  passByValue(foo); //No problem! The function can only modify a local copy of foo
  passByPtr(&foo); //Compile time error! Function could in principle modify foo
  passByConstPtr(&foo); //OK! The function has promised not to modify foo
}
```

The const keyword

- Member functions of an object can be declared as const to promise that they won't try to modify any of the object's member variables.
- This promise must be made before a const object will use the functions.

```
class date { //This class represents a day, month and year
  public:
  date(int day, int month, int year); //You get the idea, so let's skip everything but the month part
  int getMonth() const; //This function is const - It promises not to change any of the member variables
  void setMonth(int month);

  private:
  int month_;
};
```

```
const date myBirthday(23, 8, 1986); //Changing my birthday makes no sense at all. I'll make it const!
int month = myBirthday.getMonth(); //No problem, getMonth has promised not to change anything
myBirthday.setMonth(8); //Results in a compiler error because setMonth is not const. It might make changes!
```

- Code that works as intended with const variables is called "const correct".
- If you want your code to be const correct, *do it right from the start!* It is extremely difficult to take a program that is not const correct and fix it.

Operator overloading

- When an operator does more than one thing, it is said to be overloaded.
- For example, the addition operator + is overloaded. It adds when acting on integers, but concatenates when acting on strings.

```
int aVal = 10;
int bVal = 20;
int cVal = aVal + bVal; //The addition operator adds two numbers and returns the sum, so cVal = 30

string aStr = "Hello";
string bStr = ", world!";
string cStr = aStr + bStr; //Now the same operator concatenates two strings, so cStr = "Hello, world!"
```

- Operators are really just convenient shorthands for function calls. The `operator` functions have silly names, but they are ordinary functions.

```
c = a.operator!(); //Equivalent to c = !a
c = a.operator+(b); //Equivalent to c = a + b
c = operator+(a,b); //Also equivalent to c = a + b. The operator doesn't have to be a member of a
```

Function overloading

- The overloaded functions must have different input parameters either by data types or their number

```
#include <iostream>

// volume of a cube
int volume(int s)
{
    return s*s*s;
}

// volume of a cylinder
double volume(double r, double h) //also works: int volume(double r, double h)
{
    return 3.1415926*r*r*h;
}

int main()
{
    std::cout << volume(10) << std::endl; // prints 1000
    std::cout << volume(2.5, 8) << std::endl; // prints 157.08
    return 0;
}
```

What is coding style?

Style == Readability

- How and when to use comments,
- Tabs or spaces for indentation (and how many spaces),
- Appropriate use of white space,
- Proper naming of variables and functions,
- Code grouping an organization,
- Patterns to be used/avoided.

Why Coding Style Matters?

- Make Errors Obvious.
- Easy to understand logic of your own old codes.
- Style is mandatory in any SW developer team.

Clean Code: **Meaningful names**

- Use meaningful (intention-revealing) names

```
const int size;
int nCycles;
double time;
```

Better

```
const int sizeOfAlloccationInBytes;
int numberOfAllocations;
double timeToSleepBetweenAllocations;
```

Clean Code: **Functions**

- Keep functions **small** (no more than 20 lines)

```
public void renderWebPage() {
  StringBuilder content = getContentBuilder();
  content.append("<html>");
  content.append("<head>");
  for (HeaderElement he : getHeaderElements()) {
    String headerEntry = he.getStartTag() + he.getContent() +
      he.getEndTag();
    content.append(headerEntry);
  }
  content.append("</head>");
  content.append("<body>");
  for (BodyElement be : getBodyElements()) {
    String bodyEntry = /* .. */
      content.append(bodyEntry);
  }
  content.append("</body>");
  content.append("</html>");
  OutputStream output = new OutputStream(response);
  output.write(content.toString().getBytes());
  output.close();
}
```

Clean Code: **Functions**

- Keep functions **small** (no more than 20 lines)

```
public void renderWebPage() {
  startPage();
  includeHeaderContent();
  includeBodyContent();
  endPage();
  writePageToResponse();
}

private void startPage() { /* ... */ }

private void includeHeaderContent() { /* ... */ }

private void includeBodyContent() { /* ... */ }

private void endPage() { /* ... */ }

private void writePageToResponse() { /* ... */ }
```

Clean Code: **Functions**

Function

- should do **one thing**,
- should do it **well**,
- should do it **only**,
- with less arguments is easier to use.

```
calendar.SetDate(2014,2,3);
```

```
calendar.SetDate(todaysDate);
```

Clean Code: **Comments**

- Intuitively understandable code is better than complex code with a lot comments.

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) != 0 && (employee.age > 65))
```

```
if (employee.isEligibleForFullBenefits())
```

- Describe why you do something - not how!

```
// We need to remove duplicates from the names because
// a person cannot have the same name more than once.
Set<String> uniqueNames = new HashSet<String>(names);
```

- Don't comment obvious things

```
// Check if members have been initialized. If not, do it!
if (members == null) {
  members = new ArrayList<Member>();
}
```

Final words

- Aim for simplicity, whenever possible.
- Stick to one coding style. Importance of code readability usually are underestimated.
- Use Coding Tools.
- Use Google and Stack Overflow.

- Procedural programming versus OOP $\rightarrow$ objects and data "versus" actions and logic
- The aim of this course is not to make you a wizard in C++, but to make you a better scientist!
- Take-home message: you will not learn only by reading or listening. Be active! Try and error! Create a small project, use a coding language of your own choice and become better!

Homework

Homework instructions are at Live@Lund.
Remember: homework is mandatory! But also remember: if you get stuck, you can submit incomplete homework and always ask for help from me.