C++: Classes

ens-lal

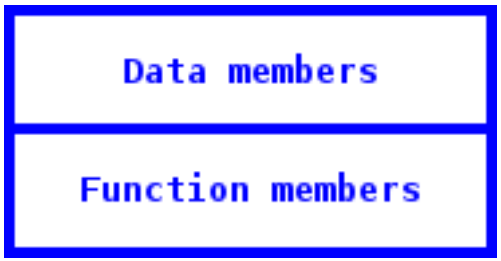
2013

Outline

- Introduction to Object Oriented Languages
  - what is an object
  - properties of an object
  - notions of inheritance and polymorphism
- Classes in C++
  - interface
  - implementation
  - constructors, destructor
  - inheritance, polymorphism

Object Oriented Language: what's an object ?

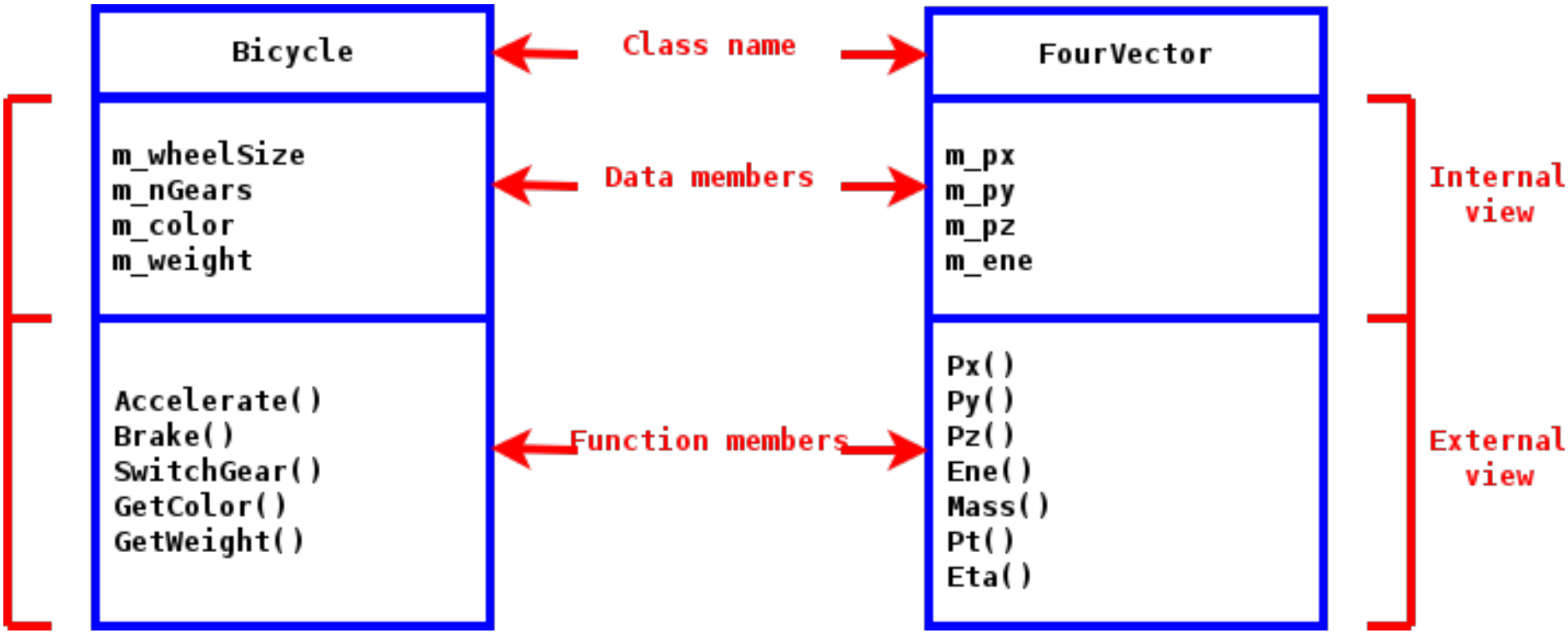
- an object is a computer entity holding:
  - data members (fields, attributes, instance's variables)
  - member functions (methods, subroutines)



Grouping variables and functions within the same entity is called encapsulation

- access to data and methods can be regulated:

Object: examples (using the UML notation)



Properties of an object

- an object has a state
  - corresponds to the value of its attributes at a given time t
  - an object's state can evolve with time
- an object is described by a class
  - a class is a prototype defining all the attributes and methods common to all objects of a given type
  - a class is a blueprint to create new objects with common traits
- an object has an identity
  - objects can be distinguished apart even if all of their attributes have the same value.

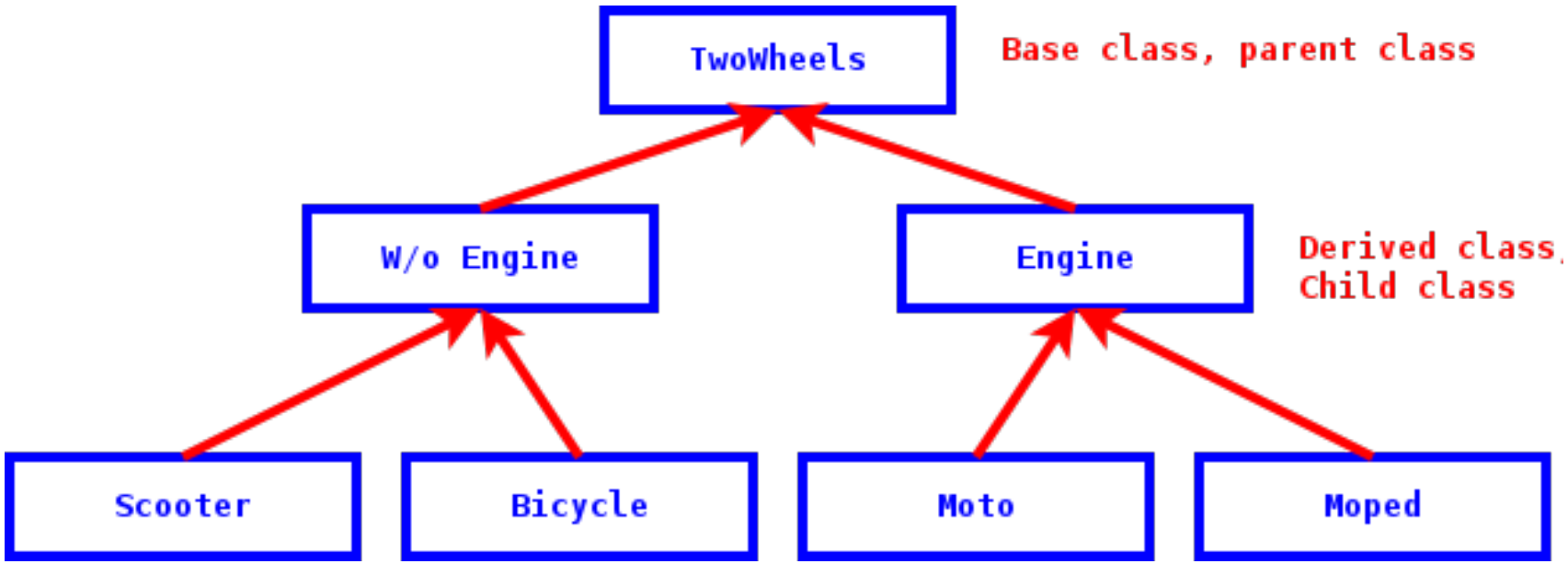
Do not confuse an instance (of a class) with a class (of objects)

- an instance refers to a particular object
- a class refers to a group or category of similar things.
  - the bicycle of my neighboor and mine are 2 instances of the same class "bicycle" even if their are strictly identical.

Notion of inheritance

Inheritance is one of the pillars of the Object Oriented Programming (OOP): it allows to create a new class from an already existing one.
The new class, called derived class, holds the attributes and methods from the parent class, plus the new attributes and new methods of that new class.
Inheritance allows to create a hierarchy of classes:

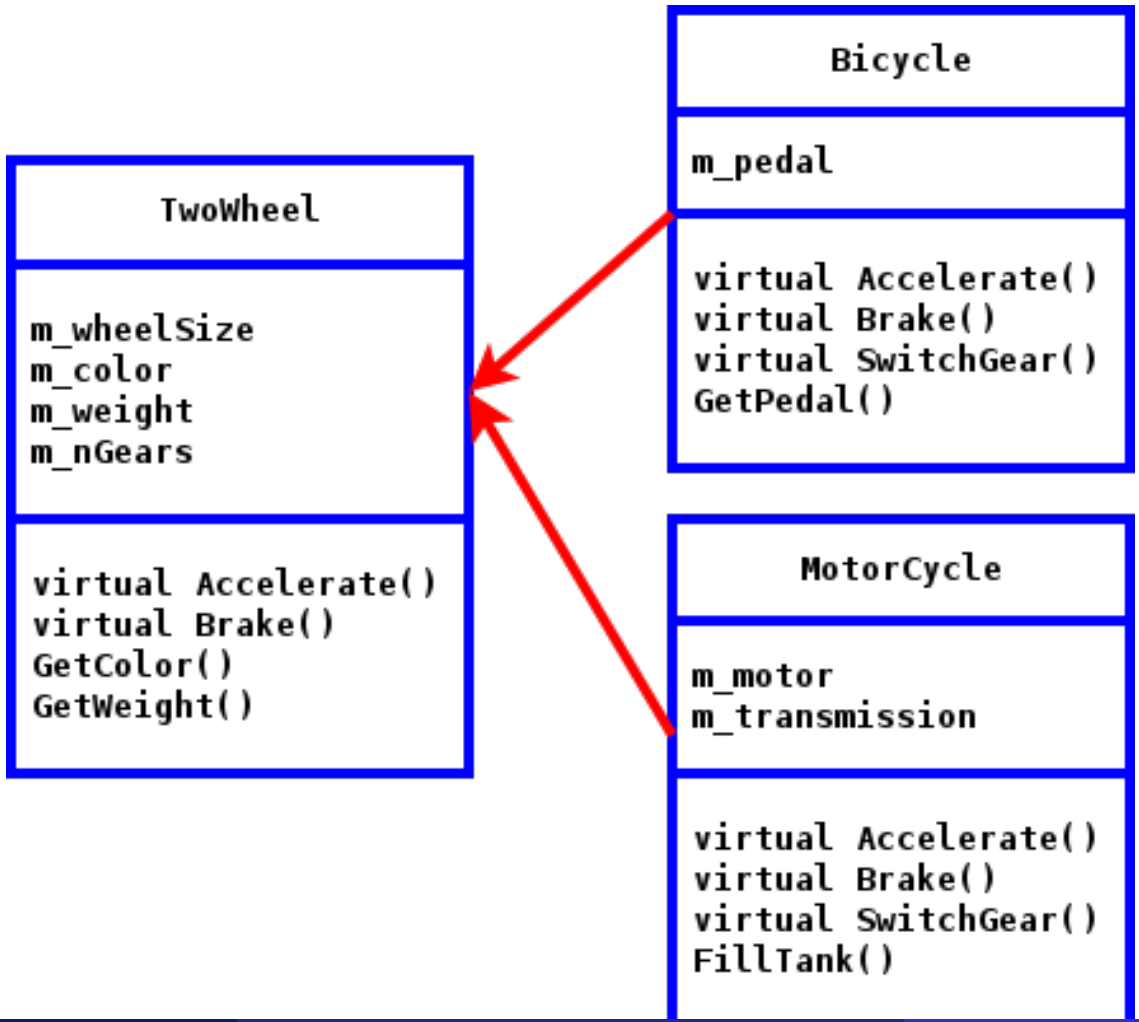
- the base class is the most generic class of that tree
- the derived classes are more and more specialized



Notion of polymorphism

- a derived class may provide a new definition for a method inherited from a parent class
  - *e.g.* if it needs to react differently when that method is called
  - this new definition will be override the parent one: it is called specialization
- polymorphism: the same operation or method does something different on different classes of the same hierarchy tree.
  - one can call this method w/o having to worry about the intrinsic type of that object.
  - one abstracts away the details of the more specialized classes of a family of objects by hiding those details behind a common interface (usually the base class)

- the method `Accelerate()` isn't implemented the same way for a `bicyle` and a `motocycle`.
- the definition provided for this method by each of these subclasses triggers a different behaviour whether the underlying TwoWheel object is Bicyle or a MotorCycle



OOP *vs* Procedural programming

- Pros
  - programs are easier to maintain
    - ⋆ in a procedural program, if one wants to modify a data structure, almost all the code needs to be rewritten
  - programs are clearer
    - ⋆ all the functions are attached to a data type
  - increased modularity
    - ⋆ possibility/easier to reuse code
- Cons
  - programs are less efficient
    - ⋆ memory-wise and speed-wise
    - ⋆ b/c of abstractions

Class: interface

Description/declaration of the internal structure of a class

Members' visibility:

- `public`: members accessible to everyone
- `private`: members accessible only from within the class
- `protected`: members accessible from within that class and its derived classes

```
class Ellipsis {
// -------> internal view
protected:
 float m_cx, m_cy;
 float m_a, m_b;
// <------- internal view

// -------> external view
public:
 void move(float dx,
           float dy);
 void zoom(float z);
 float surface();
// <------- external view
};
```

Class: implementation

Definition of its associated functions.

```
class Ellipsis {
protected:
 float m_cx, m_cy;
 float m_a, m_b;
public:
 void move(float dx,
           float dy);
 void zoom(float z);
 float surface();
};
```

```
void
Ellipsis::move(float dx, float dy)
{ m_cx += dx; m_cy += dy; }

void
Ellipsis::zoom(float z)
{ m_a *= z; m_b *= z; }

#include <math.h>
float
Ellipsis::surface()
{
  return 0.25 * M_PI * m_a * m_b;
}
```

Classes: instantiation

```
class Ellipsis {
protected:
 float m_cx, m_cy;
 float m_a, m_b;
public:
 void move(float dx,
           float dy);
 void zoom(float z);
 float surface();
};
```

```
int main(int argc, char **argv) {
 // allocation on the stack
 Ellipsis e;

 // access to members with '.'
 e.move(50., 0.);
 float s = e.surface();
 e.zoom(1.5);

 e.m_cx = 30.; // NOT allowed !!!
 e.m_a = 2.;   // NOT allowed !!!
 return 0;
}
```

Files layout

- by convention, files containing C++ code have `.cpp`, `.c++`, `.cc`, `.cxx` or `.C` for extension
- the file holding the declarations is called header file and has `.hh`, `.hxx` or `.h` as an extension
- by convention, one creates one `.cxx` and one `.hxx` file per class
  - each of these files are named after the class name, in lower case.
- by convention:
  - class names start with an upper case
  - data members names start with `m_` or just `_`
  - member functions names are lower case.

ellipsis.h

```
class Ellipsis {
protected:
 float m_cx, m_cy;
 float m_a, m_b;
public:
 void move(float dx,
           float dy);
 void zoom(float z);
 float surface();
};
```

ellipsis.cxx

```
#include <math.h>
#include "ellipsis.h"
void
Ellipsis::move(float dx,
               float dy)
{ m_cx += dx; m_cy += dy; }

void
Ellipsis::zoom(float z)
{ m_a *= z; m_b *= z; }

float
Ellipsis::surface()
{
  return 0.25 * M_PI * m_a * m_b;
}
```

main.cxx

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

int main(int argc, char **argv)
{
  Ellipsis e;

  e.move(50., 0.);

  float s = e.surface();
  std::cout << "surface= " << s << std::endl;

  e.zoom(1.5);

  return 0;
}
```

Constructors

- the constructor is member function responsible for allocating and initializing the data members of a new class instance
  - systematically called when an object is instantiated
  - has no return type
  - is named after the class' name
- a class can have multiple constructors
- special constructors:
  - default constructor
    - ★ no argument
    - ★ automatically generated by the compiler if the user does not provide one
  - copy constructor
    - ★ takes one argument of type "the object's type"
    - ★ creates "clones" of objects
    - ★ automatically generated by the compiler if none provided

ellipsis.h

```
class Ellipsis {
public:
 // default c-tor
 Ellipsis();
 // c-tor with parameters
 Ellipsis(float cx, float cy, float a, float b);
 // copy c-tor
 Ellipsis(const Ellipsis &e);

protected:
 float m_cx, m_cy;
 float m_a, m_b;
public:
 void move(float dx, float dy);
 void zoom(float z);
 float surface();
};
```

ellipsis.cxx

```
#include "ellipsis.h"
Ellipsis::Ellipsis()
{
  m_cx = m_cy = 0.;
  m_a = m_b = 1.;
}

Ellipsis::Ellipsis(float cx, float cy,
                   float a,  float b) :
  m_cx(cx), m_cy(cy),
  m_a ( a), m_b ( b)
{}

Ellipsis::Ellipsis(const Ellipsis& e) :
  m_cx(e.m_cx), m_cy(e.m_cy),
  m_a (e.m_a ), m_b (e.m_b )
{}
```

main.cxx

```
#include "ellipsis.h"

int main(int argc, char **argv)
{
  Ellipsis e1;
  Ellipsis e2(2.5, 6.5, 12., 15.);

  // e3 is a clone of e1
  Ellipsis e3(e1);

  // e4 is another clone of e1
  Ellipsis e4 = e1;

  return 0;
}
```

Destructor

- member function systematically called just before the destruction of an object
- named after the class' name, with ~ in front
- no return type
- no argument
- only one per class
- release resources (memory, network connection, file handles, ...) to the operating system

ellipsis.h

```
class Ellipsis {
public:
 // default c-tor
 Ellipsis();
 // c-tor with parameters
 Ellipsis(float cx, float cy, float a, float b);
 // copy c-tor
 Ellipsis(const Ellipsis &e);

 // d-tor
 ~Ellipsis();

protected:
 float m_cx, m_cy;
 float m_a, m_b;
 // etc... as before.
};
```

ellipsis.cxx

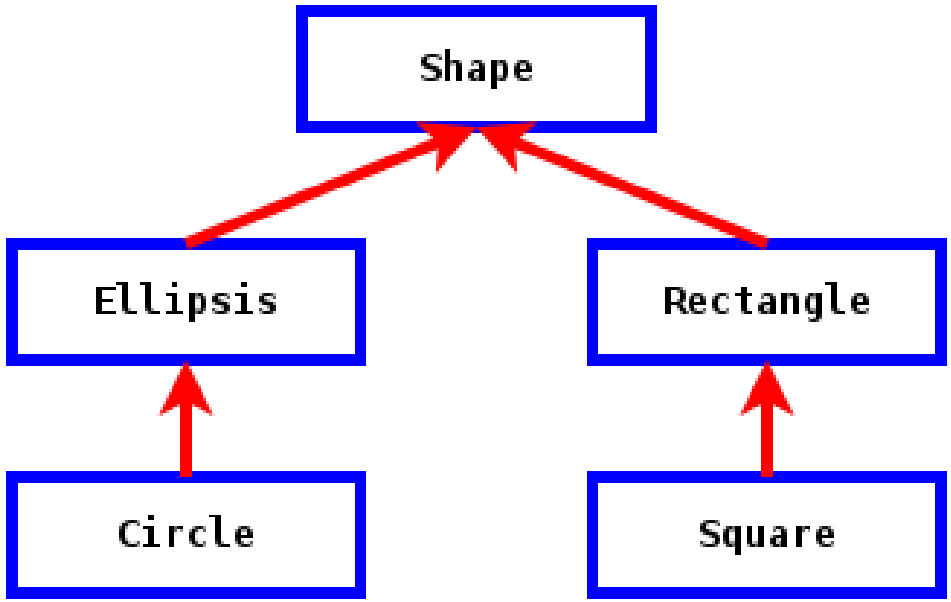
```
#include "ellipsis.h"
Ellipsis::~Ellipsis()
{
 // release resources...
 // we don't have anything to do here, for Ellipsis.
}

Ellipsis::Ellipsis(float cx, float cy,
                   float a,  float b) :
  m_cx(cx), m_cy(cy),
  m_a ( a), m_b ( b)
{}

// as before...
```

main.cxx

```
#include "ellipsis.h"
int main(int argc, char **argv)
{
  // allocate an ellipsis on the stack
  // => automatic memory
  Ellipsis e1;

  // allocate an ellipsis on the heap
  // => dynamic memory. *user* has to manage it.
  Ellipsis * e2 = new Ellipsis(2.5, 6.5, 12., 15.);

  // manually delete 'e2'
  // implicitely call d-tor of Ellipsis on e2
  delete e2;

  return 0;
} // <-- e1 "goes out of scope". d-tor is called.
```

Inheritance

- Inheritance allows de specialize a class by defining a Is A kind of relationship (`IsA` in the litterature)
- a circle may be modeled as a specialization of an ellipsis
  - it has the same properties plus some more which are specific to a circle
  - one then makes the `Circle` class derive from the `Ellipsis` class



ellipsis

```
// ellipsis.h --------------------------
class Ellipsis {
public:
 Ellipsis();
 //...
 virtual void display();
};
// eof ---------------------------------
```

```
// ellipsis.cxx ------------------------
#include <iostream>
#include "ellipsis.h"
void Ellipsis::display() {
  std::cout << "Ellipsis{a=" << m_a << ", b=" << m_b << "}"
            << std::endl;
}
// eof ---------------------------------
```

circle

```
// circle.h
#include "ellipsis.h"
class Circle : public Ellipsis {
public:
 Circle();
 Circle(float x, float y, float r);
 ~Circle();

 virtual void display();
};
// eof
```

circle

```
// circle.cxx
#include <iostream>
#include "circle.h"
Circle::Circle() : Ellipsis()
{}

Circle::Circle(float x, float y, float r) :
 Ellipsis(x, y, 2.*r, 2.*r)
{}

void Circle::display()
{
  std::cout
     << "Circle{radius=" << m_a*0.5 << "}"
     << std::endl;
}
```

main

```
#include "circle.h"

int main(int argc, char **argv)
{
  Circle c(5., 5., 15.);
  c.display();
  return 0;
}
```

```
$ ./test-circle
Circle{radius=15}
```

Polymorphism: An object inheriting a method from a parent class, can react or behave differently than the parent class when a call to that method takes place.

main

```
#include "circle.h"
int main(int argc, char **argv) {
 Ellipsis e(0., 0.5, 8.5, 10.2);
 e.move(-1, 1); e.display();

 Circle c(-2.5, 2.5, 7.4);
 // Ellipsis::move isn't redefined in Circle
 // => calls Ellipsis::move
 c.move(0.5, 1.5);
 // Ellipsis::display was redefined in Circle
 // => calls Circle::display
 c.display();
 return 0;
}
```

main

```
#include "circle.h"
int main(int argc, char **argv) {
  Ellipsis *e1 = new Ellipsis;
  // call the Ellipsis::display method
  e1->display();
  delete e1; e1 = NULL;

  Ellipsis *e2 = new Circle;
  // given that:
  //  - Ellipsis::display is a virtual method
  //  - Ellipsis::display is redefined in Circle::display
  // => call the ::display method of the underlying type
  //    (ie: Circle)
  // => this is called inheritance polymorphism
  e2->display();
  delete e2; e2 = NULL;
  return 0;
```

Vocabulary

- variable: associate a name (a symbol) with a value, whose value may evolve thru time. A variable has a type, defined once and for all by the program
- encapsulation: grouping variables and functions together inside an entity, called class.
- class: prototype (blueprint) defining the attributes and methods common to all the instances (objects) of a given type.
- class interface: description/declaration of the internal structure of a class, including the list of the data members and the declarations of member functions, in header file (usually a `.h`)
- class implementation: code definition of the functions declared in the class interface, in an implementation file (usually `.cxx`.)

Vocabulary (cont'd)

- Inheritance: allows to define a hierarchy tree of classes, each child class inheriting the methods and attributes of its parent(s)
- Polymorphism: 2 objects inheriting a method from the same parent class, can react differently to a call placed for this method (by redefining this method.) It is then possible to call this method w/o worrying about its underlying concrete type.

Questions ?

Questions ?