Composite objects: `struct`s and `class`es

P. Ronchese
Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

Composite objects

C/C++ allow the definition of “composite objects”, i.e. objects containing several variables and/or other objects

- Useful to group together related variables/object
- Two types of composite objects: `struct` and `class` :
  - `struct` comes from plain-C
  - `class` is C++-specific

```
struct Point {
  float x;
  float y;
}; // a semicolon is required
```

Basic properties

Basic properties of composite objects

- their pointer or reference can be taken and passed or returned by a function
- they can contain native variables and/or their pointers
- they can contain other composite objects and/or their pointers
- they can contain pointers to themselves (directly or indirectly)
- they cannot contain instances of themselves (neither directly nor indirectly)
- their “members” can be accessed
- other properties (C++ specific, not available in plain C) will be shown later

Declaration and definition of `struct`s

In C/C++ all variables must be "declared" before being used; `struct`s need also being "defined"

Declaration: a name is simply stated as identifying a `struct`. It can be repeated.

```
// Point declaration
struct Point;
Point* pp;
```

Definition: all the members of the `struct` must be specified. One and only one definition must exist in each translation unit using it.
A definition is also a declaration.

```
// Point definition
struct Point {
  float x;
  float y;
};
Point p;
```

- To create a `struct` the definition is necessary
- To create a pointer the declaration is enough

Initialization

Initializer lists can be used to set member content when creating the `struct`.

```
Point p={23.5,43.1};
```

C++11 only

The “=” can be removed (uniform with other initializers)

```
Point p{23.5,43.1};
```

Access to members

`struct`s can be initialized with a list.
When a `struct` has been created,
its member are accessed with their names.

```
Point p={2.35,6.71};
std::cout << p.x << " " << p.y << std::endl;
```

The members of a `struct` can be accessed
starting from a pointer, too

```
Point p;
Point* pp=&p;
(*pp).x=4.59; // parentheses are needed
pp->y=-12.86; // equivalent to(*pp).y=-12.86
```

Nested `struct`s

A `struct` can be defined inside the definition of another one

```
struct Outer {
  ...
  struct Inner {
    float x;
    ...
  };
  Inner i;
  ...
};
  ...
  Outer o;
  o.i.x=13.6;
```

Arrays of pointers

When dealing with composite objects, array of pointers could be used; they can be created dynamically as other arrays. They appear as pointers to pointers.

```
Point** pp=new Point*[n];
for(i=0;i<n;++i) {
  Point* p=new Point;
  p->x=...;
  p->y=...;
  pp[i]=p;
}
...
for(i=0;i<n;++i) delete pp[i];
delete[] pp;
```

BEWARE! Creating an array of pointers does NOT create the objects!

Memory sharing: `union`s

In a `struct` the members are stored in memory sequentially; in an `union` the members share the same memory locations.

All the objects are stored starting from the same memory location

```
union Misc {
  float x;
  int i;
  char* p;
};
Misc m;
```

- The size of the `union` is the size of the largest object
- Only one object can be stored at once
- Undefined results are obtained when writing one object (e.g. `m.x`) and reading another one (e.g. `m.i`)

`struct`s and `class`es

`class`: the main improvement of C++ versus plain C

A `class` is essentially an evolution of a `struct`

Plain-C `struct`s contain only variables or other objects,
C++ `class`es provide several new functionalities:

- constructor(s) and destructor,
- functions handling data members,
- access specifiers to control access to data.

The standard extended to `struct`s the properties of `class`es

`class` "interface"

The definition of a `class`, with all its functions, is also called "interface"

```
class Point {
 public:  // accessible by all functions
  Point(float x, float y); // constructor
  ~Point();                //  destructor
  float getX() const; // member functions
  float getY() const;
  float dist(const Point& p) const;
 private: // accessible only by the class
  float xp; // member data
  float yp;
};
```

- `struct` : members are `public` by default
- `class` : members are `private` by default

Declaration, definition and implementation of `class`es

- A `struct/class` declaration can appear any number of times
- A `struct/class` definition (also called “interface”) must appear once and only once in each translation unit using it
- A `struct/class` implementation (functions code) must appear once and only once in the whole program, functions can be implemented in the definition being implicitly `inline`

> `class` definitions are usually coded in “header files”, with “include guards” to prevent multiple inclusions

```
#ifndef Point_h
#define Point_h
class Point {
  ...
};
#endif
```

Different naming conventions

Different meaning are sometimes given
to “declaration” and/or “definition”:

Instruction	Used here	Used somewhere	Unambiguous name
`class Pippo;`	declaration	declaration	forward declaration
`class Pippo { ... };`	definition	declaration	interface
`Pippo::...(...) { ... }`	implementation	definition	

Constructor and destructor

The "constructor" and "destructor" of a class are executed when an object is created or destroyed

```
Point::Point(float x, float y):
 xp(x),
 yp(y) {
}
Point::~Point() {
}
```

- Data members are initialized in the order they're declared in the class definition, not as they're listed in the constructor: important when a member is initialized using another one.
- Destructor is often empty; typical operations are:
  - `delete` dynamic objects used by the class
  - close files opened and used by the object
  - free other resources allocated by the object

Object destruction

An object can be destructed in different circumstances

- An automatic object is destroyed when it goes out of scope,
- A `static` object is destroyed at the execution end,
- A dynamic object is destroyed when a `delete` instruction is executed.

If an object is created with `new` and it is not destroyed with `delete` the destructor is NOT run. Any resource allocated by the object is simply released by the operating system at the execution end.

Function members

Function members (sometimes called "methods") have direct access to member data of the object

```
float Point::getX() const {
  return xp;
}
float Point::getY() const {
  return yp;
}
float Point::dist(const Point& p) const {
  return sqrt(pow(xp-p.xp,2)+pow(yp-p.yp,2));
}
```

Functions are declared `const` when
they do not modify any member of the object;
only `const` function can be called for `const` objects.

`const` and `mutable` members

Members declared `const`

- they cannot be modified during the object life
- they must be initialized in the contructor

Members declared `mutable`

- they can be modified also by functions declared `const` or through pointers to `const`
- they're to be used for members not affecting the "externally visible state" of an object.

Memeber initialization at declaration

C++11 only

Class members can be initialized in the definition.

```
class Pippo {
 public:
  Pippo();
  Pippo(int i);
  ~Pippo();
 ...
 private:
  int x = 1;
};
Pippo() { // x=1
}
Pippo(int i): x(i) { // x=i
}
```

Constructor types

Default constructor

The default constructor allows the creation of an object with no arguments:

- declared with no arguments
- declared with arguments all having a default value

Copy constructor

The copy constructor takes one single argument, of the same class. It's used any time an object is copied, e.g.:

- when an object is passed to a function by value
- when an object is returned by a function

User-defined constructor:
any constructor declared by the user
(including default and copy)

Defaulted constructor and destructor

Implicit declaration

If the definitions of a class does not contain any constructor and/or destructor, “defaulted” (i.e. implicitly declared) ones are automatically provided by the compiler itself.

- Implicitly-declared default constructor:
  the default constructor for each member is called
- Implicitly-declared destructor:
  the destructor for each member is called
- Implicitly-declared copy and assignment:
  each member is simply copied

If any user-defined constructor is declared,
with or without parameters,
there’s no default constructor implicit declaration.

Defaulted constructor and destructor recovery or drop

C++11 only

Defaulted constructor and destructor can be explicitly declared or removed.

```
class Point {
 public:
  Point() = default; // allow d.c.
  Point(float x, float y);
 ...
};
class Line {
 public:
  Line() = delete; // remove d.c.
  Line(const Point& p1,const Point& p2);
 ...
};
```

Implicitly declared members summary

A default constructor is implicitly declared if:

- no user-defined (generic) constructor is declared
- OR it's declared with the specification `= default`.

A default constructor is NOT implicitly declared if:

- at least one user-defined constructor is declared
- OR it's declared with the specification `= delete`.

A copy constructor is implicitly declared if:

- no user-defined copy constructor is declared
- OR it's declared with the specification `= default`.

A copy constructor is NOT implicitly declared if:

- an user-defined one is declared and implemented
- OR it's declared with the specification `= delete`.

A copy constructor (user-defined or implicitly declared) is always present, unless explicitly `delete`d.

Delegated constructor

C++11 only

A constructor can delegate to another one.

```
class Point {
 public:
  Point(float d);
  Point(float x, float y);
 ...
};
Point::Point(float d):
 Point(d,d) { // delegate to other constructor
}
```

Shared members declaration

Each “instance” of a `class` contains its own members, e.g. each `Point` contains its `x` and `y`

A member shared by all the instances of a class can be declared by using the keyword `static`

```
class Line { // ax+by+c=0
 public:
  Line(const Point& p1,const Point& p2);
  ~Line();
  Point intersect(const Line& l) const;
 private:
  static float tolerance;
  float a;
  float b;
  float c;
};
```

Shared members initialization

Shared (`static`) data members are not bound to any specific instance of a class

- They are created at the execution start (or when loading dynamic libraries), even if no instance is created
- They must be initialized, only once, outside any function

```
float Line::tolerance=1.0e-05;
Point Line::intersect(const Line& l) const {
  float det=(a*l.b)-(b*l.a);
  float chk=pow(  a,2)+pow(  b,2)+
            pow(l.a,2)+pow(l.b,2);
  if(fabs(det/chk)<tolerance)
     return Point(FLT_MAX,FLT_MAX);
  return Point(((b*l.c)-(c*l.b))/det,
               ((c*l.a)-(a*l.c))/det);
}
```

Shared members functions

Shared (`static`) member functions, not bound to any specific instance of a class, can be declared as well

- They can be called using the “scope resolution” (`::`) operator
- They can access only `static` data members
- They cannot access non-`static` data members, but for specific objects they may have access to (e.g. passed as parameters, or global objects)

```
class Pippo {
 public:
  static int count();
  ...
};
  ...
  int j=Pippo::count();
```

Cross references among `class`es

Two (or more) `class`es may exist, each one using the other as argument of its own functions: both must know about the other

```
class Line;
class Point {
 ...
  float dist(const Line& l) const;
 ...
};
```

```
class Point;
class Line {
 ...
  float dist(const Point& p) const;
 ...
};
```

Self reference

Each instance can obtain the pointer to itself from `this`

- It can be used as parameter when calling functions
- It can be returned by member functions
- It can be dereferenced to obtain the object instance

```
float Line::dist(const Point& p) const {
  return fabs((a*p.getX())+(b*p.getY())+c)/
         sqrt((a*a       )+(b*b       )  );
};
```

```
float Point::dist(const Line& l) const {
  return l.dist(*this);
};
```

friend functions and classes

A class can declare friend functions and classes, allowed to access its private members (use sparingly!).

```
class Point {
  friend class Line;
  // all functions of "Line" can access
  // private members of "Point"
  ...
};
class Line {
  friend
  float Point::dist(const Point& p) const;
  // only the function "dist" of "Point"
  // can access private members of "Line"
  ...
};
```

Operator members

Not only functions but also operators can be defined for classes

```
class Vector2D {
 public:
  Vector2D(float x, float y);
  ~Vector2D();
  float getX() const;
  float getY() const;
  Vector2D  operator+(const Vector2D& v);
  Vector2D& operator*=(float f);
 private:
  float xv;
  float yv;
};
```

- Operators are defined as other functions.
- Assignment operators return a “`*this`”.

Operators definition

Operator members are to be defined as member functions

```
Vector2D  Vector2D::operator+(const
                                    Vector2D& v) {
  return Vector2D(xv+v.xv,yv+v.yv);
}
Vector2D& Vector2D::operator*=(float f) {
  xv*=f;
  yv*=f;
  return *this;
};
```

Class operators can be used as the built-in ones, or through explicit function calls

```
Vector2D u( 2.3,4.5);
Vector2D v(-1.6,6.9);
Vector2D s=u+v;
u*=3; // equivalent to u.operator*=(3)
```

Operator functions

Operators can be defined also as global functions, where at least an argument must be a `class`

```
Vector2D operator+(const Vector2D& vl,
                   const Vector2D& vr) {
  return Vector2D(vl.getX()+vr.getX(),
                  vl.getY()+vr.getY());
}
Vector2D& operator*=(Vector2D& v,float f) {
  v = Vector2D(v.getX()*f,v.getY()*f);
  return v;
}
```

- Both implementations can be present
- The compiler flags as an error any ambiguous call
  - `u.operator+(v)` calls the operator member
  - `operator+(u,v)` calls the operator function

Functors

Objects usable as functions are called “functors”.

```
class Func {
 public:
  Func(int n):f(n) {};
  float operator()(float x) {return f*x;}
 private:
  int f;
};
int main() {
  // create a Funct setting it at 3
  Func m(3);
  // call the Funct with 5.3
  cout << m(5.3) << endl;
  return 0;
}
```

I/O Operators

Operator functions can be defined to write/read objects

```
std::ostream& operator<<(std::ostream& os,
                         const Vector2D& v) {
  os << v.getX() << " " << v.getY();
  return os;
};
std::istream& operator>>(std::istream& is,
                         Vector2D& v) {
  float x,y;
  is >> x >> y;
  v=Vector2D(x,y);
  return is;
};
```

I/O operator functions take a `std::istream&` or `std::ostream&` as argument, and return the same at the end

Operators limitations

Operators must be defined and implemented
as global function when:

- The left operand has built-in type (e.g. `int`, `float`, ...)
- The left operand type is a non modifiable `class` (e.g. `istream`)

Only existing operators (e.g. $+$, $-$, $*$, $/$, $=$, ...)
can be redefined `class`es, no new ones can be created
(e.g. $**$ for exponentiation)

Nested `class`es

A `class` can be defined inside the definition of another one (being visible outside or not if it's `public` or `private` respectively)

```
class Outer {
 public:
  ...
  class InnerPub {
    ...
  };
 private:
  class InnerPri {
    ...
  };
  ...
};
```

A `public` nested `class` can be accessed by using the scope resolution operator `::`
`Outer::InnerPub`.
Examples will be shown in the following.

Having several `class`es nested inside the same enclosing one emphasizes the relations among them.

Type conversions by constructor

Implicit type conversions

A constructor taking a single argument defines an implicit type conversion, unless an `explicit` keyword is added.

```
class A {
 public:
  A(int i);
 ...
};
void f(A a) {
   ...
}
   ...
   f(5);
   ...
```

```
class B {
 public:
  explicit B(int i);
 ...
};
void g(B b) {
   ...
}
   ...
   g(B(5));
   ...
```

Type conversions by operator

Implicit type conversions

Conversion to other types can be defined with operators.

```
class Pippo {
 public:
  operator int();
};
void f(int i);
```

```
...
Pippo p;
f(p);
...
```

C++11 only

An `explicit operator` prevents implicit type conversions.

```
class Pluto {
 public:
  explicit
  operator int();
};
```

```
...
Pluto p;
f(static_cast<int>(p));
...
```

Name conflicts

Names of `class`es must be unique throughout the whole program (libraries included): conflicts could arise.

Functions and `class`es can be declared and defined inside "namespaces"

```
namespace Geom {
  class Line;
  class Point {
    ...
  };
};
```

Classes defined inside namespaces can be accessed by mean of the "scope" operator `::`

```
...
int main() {
  Geom::Point p(1.2,7.4);
  ...
  return 0;
};
```

Default namespaces

Adding namespace to a `class` name produces a long name...

- A `typedef` can be used
- An `using` declaration or directive can be added

```
typedef Geom::Point point;
// define "point" as a short name
```

```
using Geom::Point;
// declaration: makes "Point"
// visible without namespace resolution
```

```
using namespace Geom;
// directive:   makes all names in "Geom"
// visible without namespace resolution
```

An `using` declaration or directive affects
all the following code in the same translation unit:
avoid including “`using`” directives in header files

Error handling

A lot of situations may occur where an operation cannot be performed:

- a division by zero is required,
- the square root of a negative number is required,
- an unvalid pointer is to be dereferenced,
- ...

An error flag is to be set, propagated back and properly handled (unless there's some reason to prefer an execution crash)

Exceptions are objects that:

- are "thrown" where the error condition occur,
- are "catched" anywhere in the function calling sequence,
- contain informations about the error.

Exception objects

Any object can be (in principle) used as exception

```
class MathException {
 public:
  enum errorType {divByZero,sqrNeg};
  MathException(errorType e) {error=e;}
  ~MathException() {}
  errorType get() const {return error;}
 private:
  errorType error;
};
```

```
float x;
int i;
...
if(i==0)throw
   MathException(MathException::divByZero);
x/=i;
```

Exception catching

Exceptions are handled by mean of "`try`" and "`catch`" blocks

```
try {
   ...   // any code that could possibly
   ...   // throw a "MathException"
}
catch (MathException& e) {
   if(e.get()==MathException::divByZero)
         cout << "division by zero" << endl;
}
```

When an exception is thrown
all the calling functions are immediately terminated
going back until a "`catch`" clause is found.

Several "`catch`" blocks can exist,
handling exceptions of different type.

Exception blocking

C++11 only

A function can be declared "`noexcept`" to prevent it transmitting any unhandled exception.

```
void f(int i, float x) noexcept;
```