

Unit 5: Introduction to object oriented programming

Programming 2

Degree in Computer Engineering
University of Alicante
2025-2026



1. Introduction
2. Core concepts
3. OOP in C++
4. Objects and memory management
5. Relationships
6. Compilation
7. Exercises

Introduction

Definition

- *Object oriented programming* (OOP) is a programming paradigm that uses objects and their interactions to design software
- The whole application is treated as a set of cooperating objects and their relationships
- C++ is an object oriented language, but it also allows imperative (procedural) programming
- The approach when designing programs changes...
- ... but everything you have learned so far is still useful!

Classes and objects (1/4)

- We have already used classes and objects in Programming 2:

```
int i; // Declare a variable i of int type
string s; // Declare an object s of string class
```

- A *class* (or compound type) is a model for creating objects of that type
- An *object* of a certain class is called an *instance* of that class
- In the example above, `s` is an instance/object of the class `string`
- Classes are similar to simple types, although they allow many more functionalities

Classes and objects (2/4)

- A `struct` is a simple type
- It can be considered as a “light” class that only stores visible data:

```
struct Date{  
    int day;  
    int month;  
    int year;  
};
```

Classes and objects (3/4)

- A class contains data and a set of functions that manipulate this data, called *member functions* or *methods*
- You can control which data/methods are visible (`public`) and which are hidden (`private`)
- Member functions can access public and private data of their class
- Example of “basic” class equivalent to `struct Date`:*

```
class Date{  
    public: // Public data  
        int day;  
        int month;  
        int year;  
};
```

*We say “basic” because it does not offer any advantages with respect to `struct Date`

Classes and objects (4/4)

- Direct access to elements of the object, as in a struct:

```
Date d;  
d.day=12;
```

- In a good object oriented design, data is usually not accessed directly: methods are used to modify the data
- In the example above, `d.day=100` would not cause an error
- Methods can be used to control what values are given to the data:

```
class Date{  
    private: // Only accessible from class methods  
        int day;  
        int month;  
        int year;  
    public:  
        bool setDate(int d,int m,int y){...};  
};
```


Core concepts

- Principles on which object oriented design is based:
 - Abstraction
 - Encapsulation
 - Modularity
 - Inheritance
 - Polymorphism

Abstraction

- *Abstraction* denotes the essential characteristics of an object and its behaviour
- Each object can perform tasks, inform and change its state, communicating with other objects in the system without revealing how these features are implemented
- The abstraction process allows selecting the relevant features within a set, identifying common behaviours to create new classes
- The abstraction process takes place in the design phase

Encapsulation

- *Encapsulation* implies grouping together all the elements that can be considered as belonging to the same entity at the same level of abstraction
- The *interface* is the part of the object that is visible (public) to the other objects: a set of methods and data available to communicate with an object
- Each object hides its implementation (*how* it is done) and exposes an interface (*what* it does)
- Encapsulation protects the properties of an object against modification: only the object's own methods can access its state

Modularity (1/2)

- *Modularity* is the property that allows a program to be divided into smaller parts (called *modules*) as independent as possible
- These modules can be compiled separately, although they have connections to other modules
- Generally, each class is implemented in a separate module, but classes with similar functionalities can share a module

Modularity (2/2)

- A class `myClass` would be implemented using two source files:
 - `myClass.h`: contains constants used in this file, declaration of the class and its methods
 - `myClass.cc`: contains constants used in this file, implementation of the methods and maybe internal types used by the class
- The `main` function uses these classes as a client. It is included in a separate file (for example, `prog.cc`)
- To get the executable, you have to compile all the modules together. For example, to compile a program with two classes (`myClass1` and `myClass2`) and a main function (`prog.cc`):

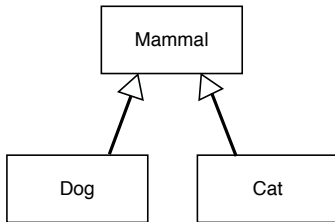
Terminal

```
$ g++ myClass1.cc myClass2.cc prog.cc -o prog
```

- This compilation method is adequate only if there are few classes
- At the end of this unit we will see how to properly compile programs with multiple classes using the *make* tool

Inheritance (1/2)

- Not studied in Programming 2
- Classes can be related to each other by forming a classification hierarchy
- *Inheritance* allows defining a new class from another class
- This applies when there are sufficient similarities and most of the features of the existing class are suitable for the new class
- In the following example, the *subclasses* `Dog` and `Cat` inherit the methods and attributes specified by the *superclass* `Mammal`:



Inheritance (2/2)

- Inheritance allows adopting features already implemented by other classes
- It facilitates the organisation of information at different levels of abstraction
- Objects inherit the properties and behaviour of all the classes to which they belong
- Derived objects can share (and extend) their behaviour without having to implement it again
- *Multiple inheritance* occurs when an object inherits from more than one class

Polymorphism

- **Not studied in Programming 2**
- *Polymorphism* is the property according to which the same expression refers to different actions
- For example, a method `move` may refer to different actions if it is applied to a plane or a car
- Different behaviours, associated with different objects, can share the same name
- References and object collections can contain objects of different types:

```
Mammal *a=new Dog;  
Mammal *b=new Cat;  
Mammal *c=new Seagull;
```

OOP in C++

Declaration and implementation (1/2)

- In this example, the class `SpaceShip` is implemented as a module using two files: `SpaceShip.h` and `SpaceShip.cc`

```
// SpaceShip.h (class declaration)  
class SpaceShip{  
    private:  
        int maxSpeed;  
        string name;  
    public:  
        SpaceShip(int ms,string nm); // Constructor  
        ~SpaceShip(); // Destructor  
        int trip(int distance);  
        string getName() const;  
};
```

Declaration and implementation (2/2)

```
// SpaceShip.cc (implementation of the methods)
#include "SpaceShip.h"

SpaceShip::SpaceShip(int ms,string nm){ // Constructor
    maxSpeed=ms;
    name=nm;
}

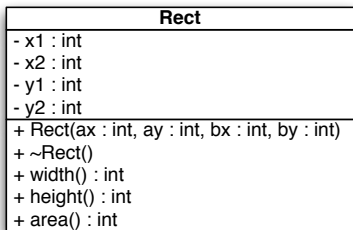
SpaceShip::~SpaceShip(){} // Destructor

int SpaceShip::trip(int distance){
    return distance/maxSpeed;
}

string SpaceShip::getName() const{
    return name;
}
```

UML diagram (1/3)

- A *UML* diagram allows describing the classes and relationships between classes in an object oriented design:



- The `-` in front of an attribute or method indicates that it is private
- The `+` indicates that it is a public attribute or method
- A horizontal line separates attributes (upper part) from methods (lower part)

UML diagram (2/3)

- Translation into code of the previous UML diagram:

```
// Rect.h (class declaration)
class Rect{
    private:
        int x1,y1,x2,y2;
    public:
        Rect(int ax,int ay,int bx,int by); // Constructor
        ~Rect(); // Destructor
        int width();
        int height();
        int area();
};
```

UML diagram (3/3)

```
// Rect.cc (implementation of the methods)
Rect::Rect(int ax,int ay,int bx,int by){
    x1=ax;
    y1=ay;
    x2=bx;
    y2=by;
}
Rect::~~Rect(){}
int Rect::width(){ return (x2-x1); }
int Rect::height(){ return (y2-y1); }
int Rect::area(){ return width()*height(); }
```

```
// main.cc (main program)
int main(){
    Rect r(10,20,40,50);
    cout << r.area() << endl;
}
```

Accessors

- It is not convenient to directly access the member attributes of a class (encapsulation principle)
- Usually attributes are defined as `private` and accessed by implementing *setter/getter/is* methods (called *accessors*):

Date
- day : int - month : int - year : int
+ getDay () : int + getMonth () : int + getYear () : int + setDay (d : int) : void + setMonth (m : int) : void + setYear (a : int) : void + isLeap () : bool

- `Setter` accessors allow controlling that attribute values are correct

- All classes must implement these four methods:
 - Constructor
 - Destructor
 - Copy constructor
 - Assignment operator
- The compiler creates these operations by default if they have not been defined by the programmer

Constructor (1/7)

- The *constructor* is automatically called when a new object of the class is created
- Classes must have at least one constructor method
- If a constructor is not defined, the compiler creates one by default without parameters (the member attributes of the objects thus created will be uninitialised)
- A class can have several constructors with different parameters (the constructor can be *overloaded*)
- Overloading is a type of polymorphism

Constructor (2/7)

- Examples of constructors:

```
Date::Date() { // Without parameters  
    day=1;  
    month=1;  
    year=1900;  
}  
  
Date::Date(int d,int m,int y){ // With three parameters  
    day=d;  
    month=m;  
    year=y;  
}
```

- Calls to the constructors:

```
Date d;  
Date d(10,2,2010);  
Date d(); // Compilation error!
```

Constructor (3/7)

- Constructors (as other functions) can have default parameters
- These default values are only declared in the header file (.h):

```
// Date.h  
class Date{  
    ...  
    Date(int d=1,int m=1,int y=1900);  
    ...  
}
```

- With this constructor we could create objects in several ways:

```
Date d; // day = 1, month = 1, year = 1900  
Date d(10,2,2010); // day = 10, month = 2, year = 2010  
Date d(10); // day = 10, month = 1, year = 1900  
Date d(18,5); // day = 18, month = 5, year = 1900
```

- The default parameters in the above example would be displayed as follows in a UML diagram:

Date
- day: int - month: int - year: int
+ Date (day: int=1, month: int=1, year: int=1900) ...

Constructor (5/7)

- If the parameters passed to the constructor are incorrect, the object should not be created
- This can be controlled through the use of *exceptions*:
 - An exception can be thrown with `throw` to indicate that an error occurred
 - An exception can be captured with `try/catch` to react to the error
- If an exception occurs and it is not captured, the program terminates immediately
- Exceptions should only be used when there is no other option (e.g. in constructors)

Constructor (6/7)

- Example of exception usage:

```
int root(int n){
    if(n<0)
        throw exception(); // Throw exception and finishes
    return sqrt(n);
}

int main(){
    try{ // Try to run these instructions
        int result=root(-1); // Causes an exception
        cout << result << endl; // This line is not executed
    }
    catch(...){ // Exceptions are captured here
        cout << "Negative number" << endl;
    }
}
```

Constructor (7/7)

- Example of constructor with exceptions:

```
Coordinate::Coordinate(int cx,int cy){  
    if(cx>=0 && cy>=0){  
        x=cx;  
        y=cy;  
    }  
    else  
        throw exception();  
}
```

```
int main(){  
    try{  
        Coordinate c(-2,4); // This object is not created  
    }  
    catch(...){  
        cout << "Wrong coordinate" << endl;  
    }  
}
```


Destructor (1/2)

- The *destructor* of the class must free the resources (usually dynamic memory) that the object is using
- A class only has one destructor function that has no arguments and returns no value
- It is a method with the same name as the class and preceded by the character ~:

```
// Declaration
~Date();

// Implementation
Date::~Date() {
    // Free allocated memory (if necessary)
}
```

Destructor (2/2)

- All classes need a destructor and, if not specified, the compiler creates one by default
- The compiler automatically calls the object destructor when object's scope ends
- It is also invoked by doing `delete`
- The destructor of an object implicitly invokes the destructors of all its attributes

Copy constructor (1/2)

- A *copy constructor* creates a new object from an existing one:

```
// Declaration
Date(const Date &d);

// Implementation
Date::Date(const Date &d) {
    day=d.day;
    month=d.month;
    year=d.year;
}
```

Copy constructor (2/2)

- The copy constructor is automatically called when:
 - A function returns an object
 - An object is initialised when it is declared:

```
Date d2(d1); // Copy constructor  
Date d2=d1; // Copy constructor  
d2=d1; // Constructor is not called but = operator
```

- An object is passed by value to a function:

```
void function(Date d);  
function(d);
```

- If no copy constructor is specified, the compiler creates one by default that makes a copy, attribute by attribute, of the object

Assignment operator

- Not studied in Programming 2
- The *assignment operator* (=) allows a direct assignment of two objects:

```
Date d1(10,2,2011); // Constructor  
Date d2; // Constructor  
d2=d1; // Assignment operator
```

- By default, the compiler creates an assignment operator that copies attribute by attribute
- It can be redefined if necessary

Inline declarations (1/2)

- Methods with little code can be implemented directly in the class (*inline* declaration):

```
// Rect.h
class Rect{
    private:
        int x1,y1,x2,y2;
    public:
        Rect(int ax,int ay,int bx,int by);
        ~Rect(){}; // Inline
        int width(){ return (x2-x1); }; // Inline
        int height(){ return (y2-y1); }; // Inline
        int area();
};
```

Inline declarations (2/2)

- It is more efficient to declare inline functions
- When compiling, the generated code for the inline functions is inserted at the point where the function is called (instead of putting the code somewhere else and making a call)
- Inline functions can also be implemented outside the class declaration, in the `.cc` file, using the reserved word `inline`:

```
inline int Rect::width() {  
    return (x2-x1);  
}
```

Constant methods (1/2)

- Methods that do not modify object attributes can be declared as *constant methods*:

```
int Date::getDay() const{ // Constant method
    return day;
}
```

- Only constant methods can be called upon a constant object:

```
int Date::getDay(){ // Not declared as const
    return day;
}
int main(){
    const Date d(10,10,2011);
    cout << d.getDay() << endl; // Compilation error
}
```

- `get` methods must be declared constant, as they simply return values and never modify the object

Constant methods (2/2)

- They are represented by putting `<<const>>` in front of the method name in UML diagrams:
- In this example there are four constant methods: `getSubtotal`, `getAmount`, `getPrice` and `getDescription`:

Line
- amount: int - price: float - description: string
+ Line() + <<const>> getSubtotal(): float + <<const>> getAmount(): int + <<const>> getPrice(): float + <<const>> getDescription(): string + setAmount(amount: int): void + setPrice(price: float): void + setDescription(description: string): void

Friend functions

- A *friend function* does not belong to the class but it can access class' private part
- The reserved word `friend` is used in the declaration:

```
class MyClass{  
    friend void aFriendFunction(int,MyClass &);  
    public:  
        ...  
    private:  
        int privateData;  
};
```

```
void aFriendFunction(int x,MyClass &c){  
    c.privateData=x; // Correct, because it is friend  
}
```

Input/output overload (1/4)

- Input/output operators of any class can be overloaded:

```
Date d;  
cin >> d;  
cout << d;
```

- The problem is that these functions cannot be member functions of a class, because the first operand (`cin/cout`) is not an object of that class (it is a `stream`)
- Operators are overloaded using friend functions:

```
friend ostream& operator<<(ostream &o, const Date &d);  
friend istream& operator>>(istream &o, Date &d);
```

- Declaration:

```
class Date{
    friend ostream& operator<<(ostream &os,const Date &d);
    friend istream& operator>>(istream &is,Date &d);
public:
    Date(int day=1,int month=1,int year=1900);
    ...
private:
    int day,month,year;
};
```

- Implementation:

```
ostream& operator<<(ostream &os, const Date &d) {  
    os << d.day << "/" << d.month << "/" << d.year;  
    return os;  
}
```

```
istream& operator>>(istream &is, Date &d) {  
    char dummy;  
    is >> d.day >> dummy >> d.month >> dummy >> d.year;  
    return is;  
}
```

Input/output overload (4/4)

- In a UML diagram, the word `<<friend>>` is put in front of the operator (as it is a friend function)
- In this example, the output operator (`operator<<`) is overloaded:

Invoice
<u>- nextId: int = 1</u> <u>+ VAT: const int = 21</u> - date: string - id: int
+ Invoice(c: Client*, date: string) + addLine(num: int, desc: string, price: float): void <u>- getNextId(): int</u> + <<friend>> operator<<: ostream &

Class attributes and methods (1/4)

- *Class attributes* have the same value for all the objects in the class (they are like global variables for the class)
- *Class methods* produce the same output for all the objects in the class and can only access class attributes
- They are also called *static* attributes and methods
- They are declared using the reserved word `static` when the class is defined:

```
class Date{  
    public:  
        static const int weeksPerYear=52;  
        static const int daysPerWeek=7;  
        static const int daysPerYear=365;  
        static string getFormat();  
        static bool setFormat(string);  
    private:  
        static string formatString;  
};
```

Class attributes and methods (2/4)

- In the file where the methods are implemented (.cc) the word `static` should not be included in the definition
- For the class `Date` of the previous example, we would have the following code:

```
// Date.cc  
string Date::getFormat(){ // Don't write static  
    ...  
}  
  
bool Date::setFormat(string s){ // Don't write static  
    ...  
}
```


Class attributes and methods (3/4)

- They are represented underlined in UML diagrams
- In this example there are two static attributes (`VAT` and `nextId`) and a static method (`getNextId`):

Invoice
<ul style="list-style-type: none">- <u>nextId: int = 1</u>+ <u>VAT: const int = 21</u>- date: string- id: int
<ul style="list-style-type: none">+ Invoice(c: Client*, date: string)+ addLine(num: int, desc: string, price: float): void- <u>getNextId(): int</u>+ <<friend>> operator<<: ostream &

Class attributes and methods (4/4)

- If the static attribute is not a simple type or is not constant, it must be declared in the class but gets its value outside of it:

```
// Date.h
class Date{
    ...
    static const string endWorld;
    ...
};
```

```
// Date.cc
const string Date::endWorld="2020"; // Don't write static
```

- It is possible to access static attributes or methods from outside of the class:

```
cout << Date::daysPerYear << endl; // Static attribute
cout << Date::getFormat() << endl; // Static method
```

The `this` pointer

- The `this` pointer is a pseudovariable that is not declared and cannot be modified
- Is an implicit argument received by all the methods (excluding the static ones), which points to the object receiving the message
- It is necessary when we want to disambiguate the name of a parameter, or when we want to pass the object as an argument to a nested function:

```
void Date::setDay(int day){  
    // day=day; Warning: day is assigned to itself  
    this->day=day;  
    cout << this->day << endl;  
}
```

Objects and memory management

Automatic objects and dynamic objects (1/5)

- Regarding their lifetime in memory, objects can be *automatic* or *dynamic*
- Dynamic objects are created at runtime using the operator `new`
- They stay in memory (usually in the *heap*) until they are explicitly deleted using the operator `delete`*
- Automatic objects are (automatically) allocated in memory (usually in the *stack*) at runtime when their scope is accessed
- They are (automatically) destroyed when outside their scope

*All these concepts were described in Unit 4

Automatic objects and dynamic objects (2/5)

- Creation of one automatic object and one dynamic object:

```
void func() {  
    Date d1; // Automatic object  
    Date *d2=NULL;  
    d2=new Date; // Dynamic object  
}  
  
int main() {  
    func();  
    ...  
}
```

- Right after calling `func`, `d1` is not in memory but `d2` is, although is no longer accessible
- Problem: the memory address pointed by `d2` is never deleted and continues occupied until the end of the program

Automatic objects and dynamic objects (3/5)

- In the previous example, we can return the pointer so that the object originally pointed by d2 can be accessed:

```
Date* func(){ // func returns a pointer to Date
    Date d1; // Automatic object
    d1.setDay(10);
    Date *d2=NULL;
    d2=new Date; // Dynamic object
    d2->setDay(20);
    return d2;
}
int main(){
    Date *d=NULL;
    d=func();
    cout << d->getDay(); // Prints "20"
    d->setMonth(1);
    delete d; // Dynamic object removed from memory
}
```

Automatic objects and dynamic objects (4/5)

- This is the previous example but returning an automatic object instead of a dynamic one:

```
Date func() {  
    Date d1; // Automatic object  
    d1.setDay(10);  
    return d1; // Copy constructor called  
}  
  
int main(){  
    Date d=func(); // Automatic object  
    cout << d.getDay(); // Prints "10"  
    d.setDay(1);  
}
```

- The copy constructor will be called in this case to initialise the object `d` in the function `main`

Automatic objects and dynamic objects (5/5)

- Assigning automatic and dynamic objects:

```
Date d1,d2,*d3=new Date,*d4=new Date;
d2.setDay(15);
d3->setDay(10);
d4->setDay(20);
d1=d2; // The assignment operator is called
Date d5=d1; // Copy constructor is called
d3=d4; // The object originally assigned to d3 cannot
      // be deleted
      // d3 and d4 refer now to the same object
cout << d3->getDay(); // Prints "20"
delete d3;
d4->setDay(5); // As wrong as d3->setDay(5);
              // d3 and d4 point to not valid memory
```

- The last instruction is incorrect, but it could work sometimes
- Deleting an object marks its memory address as ready to be reallocated (the memory is not immediately set to zero or assigned to new data)

Deep copy and shallow copy

- Consider an object of a class A which has a field which is a dynamic object:

```
class A{B *b; ... }
```

- A copy constructor of A is said to perform a *deep copy* of the object if it allocates new memory positions for the dynamic object:

```
A::A(const A &a) {  
    ...  
    b=new B(a.b);  
}
```

- A copy constructor of A is said to perform a *shallow copy* of the object if it simply makes the new field to point to the old one:






```
A::A(const A &a) {  
    ...  
    b=a.b;  
}
```

- Both options may be useful under different circumstances.

Relationships

Object relationships

- Main relationships between objects and classes:

Between objects	Association	
	Aggregation	
	Composition	
	Use	
Between classes	Generalization	

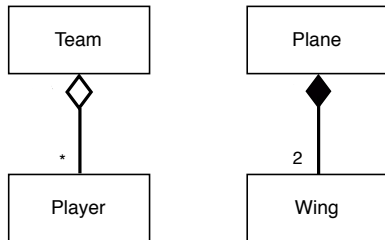
- Most relationships have cardinality:
 - One or more: $1..*$ ($1..n$)
 - Zero or more: $*$
 - Fixed value: m
- In Programming 2 we will study only aggregation and composition

Aggregation and composition (1/6)

- *Aggregation* and *composition* are whole-part relationships where an object is part of another
- They are asymmetric relationships
- The difference between aggregation and composition is the strength of the relationship: aggregation is a weaker relationship than composition

Aggregation and composition (2/6)

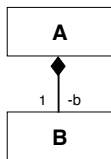
- In composition, when the container object is destroyed, the objects it contains are also destroyed
 - E.g.: the wing is part of the plane and makes no sense outside it (if the plane is sold, the wings are included)
- Regarding aggregation, this is not the case
 - E.g.: a team can be sold, but the players can go to another club (they do not disappear with the team)



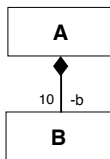
- Some relationships can be considered as aggregations or compositions depending on the context in which they are used
 - E.g.: the relation between a bicycle and its wheels
- Some authors consider that the only difference between the two concepts lies in their implementation: a composition is an “aggregation by value”

Aggregation and composition (4/6)

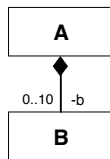
- Composition implementation:



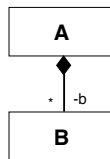
```
class A {
  private:
    B b;
  ...
};
```



```
class A {
  private:
    B b[10];
  ...
};
```



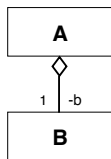
```
class A {
  private:
    vector<B> b;
    static const int N=10;
  ...
};
```



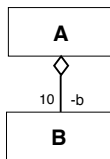
```
class A {
  private:
    vector<B> b;
  ...
};
```


Aggregation and composition (5/6)

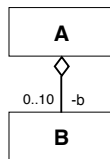
- Example of aggregation implementation:



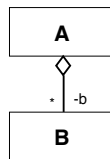
```
class A {
private:
    B *b;
...
};
```



```
class A {
private:
    B *b[10];
...
};
```



```
class A {
private:
    vector<B*> b;
    static const int N=10;
...
};
```



```
class A {
private:
    vector<B*> b;
...
};
```

Aggregation and composition (6/6)

- Aggregation implementation:

```
class A{  
    private:  
        B *b;  
    public:  
        A(B *b){ this->b=b; } // Constructor  
};
```

```
int main(){ // Two ways to implement aggregation  
    // 1- Using a pointer  
    B *b=new B;  
    A a(b); // Call to constructor  
    // 2- Using an object  
    B b;  
    A a(&b); // Call to constructor  
}
```

Compilation

The compilation process

- The translation of a source code into an executable program is done in two steps:
 - *Compilation*: the compiler translates the source code into object code (non-executable)
 - *Link*: the linker combines the object code with the language libraries (C/C++) and generates the executable file
- In C++ these two steps are carried out using the following instruction:

Terminal

```
$ g++ program.cc -o program
```

- Using the option `-c` the code is only compiled, generating object code (`.o`) but without linking it:

Terminal

```
$ g++ program.cc -c
```

Separate compilation (1/2)

- When a program consists of several source files (`.cc`), the process to obtain the executable is:
 1. Compile each source file separately, obtaining several object code files (`.o`):

Terminal

```
$ g++ -c C1.cc  
$ g++ -c C2.cc  
$ g++ -c prog.cc -c
```

2. Link the object code files with the language libraries and generate an executable:

Terminal

```
$ g++ C1.o C2.o prog.o -o prog
```

- If the program consists of few source files, the process can be done all at once:

Terminal

```
$ g++ C1.cc C2.cc prog.cc -o prog
```

Separate compilation (2/2)

- Problem: there is a header file `.h` that is used in several source files `.cc`
- What must be done if something is changed in the `.h` file?
 - Option 1: everything is recompiled again (“brute force”)
 - Option 2: search “by hand” where the header is used and recompile only these files
 - Option 3: automatically search where the header is used and only recompile these classes
- The best is “Option 3” and there is a tool called *make* that helps us do it

The *make* tool (1/6)

- The *make* tool helps in compiling large programs
- Allows setting *dependencies* between files
- Compiles a file when one of the files it depends on changes
- The text file *makefile* specifies the dependencies between files and what to do when something changes

The *make* tool (2/6)

- The *make* tool searches by default for a file called *makefile*
- This file describes a main *objective* (usually the executable program) and a series of secondary objectives
- The format of each objective in the *makefile* is:

```
<objective> : <dependencies>  
[tab]<instruction>
```

- The algorithm of *make* is very simple: *“If the date of any dependency is more recent than the date of the objective, then execute instruction”*

The *make* tool (3/6)

- Imagine that you have the following files:

```
// C1.cc  
#include "C1.h"  
...
```

```
// C2.cc  
#include "C2.h"  
#include "C1.h"  
...
```

```
// prog.cc  
#include "C1.h"  
#include "C2.h"  
...  
int main() {  
...  
}
```

The *make* tool (4/6)

- The makefile content would be:*

```
prog : C1.o C2.o prog.o
    g++ -Wall -g C1.o C2.o prog.o -o prog
C1.o : C1.cc C1.h
    g++ -Wall -g -c C1.cc
C2.o : C2.cc C2.h C1.h
    g++ -Wall -g -c C2.cc
prog.o : prog.cc C1.h C2.h
    g++ -Wall -g -c prog.cc
```

*The `-Wall` option shows all the *warnings* whereas `-g` adds debugging information

The *make* tool (5/6)

- In the previous example, if `C2.c` is modified and *make* is run:

Terminal

```
$ make
g++ -Wall -g -c C2.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

- And if `C2.h` is modified and *make* is run:

Terminal

```
$ make
g++ -Wall -g -c C2.cc
g++ -Wall -g -c prog.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

The *make* tool (6/6)

- Previous example using constants (more “professional”):*

```
CC = g++
CFLAGS = -Wall -g
OBJS = C1.o C2.o prog.o

prog : $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o prog
C1.o : C1.cc C1.h
    $(CC) $(CFLAGS) -c C1.cc
C2.o : C2.cc C2.h C1.h
    $(CC) $(CFLAGS) -c C2.cc
prog.o : prog.cc C1.h C2.h
    $(CC) $(CFLAGS) -c prog.cc
clean:
    rm -rf $(OBJS)
```

*More information at: [https://en.wikipedia.org/wiki/Make_\(software\)](https://en.wikipedia.org/wiki/Make_(software))

Header guards

- Compilation errors can occur when a header file is included in multiple files in our code
- The compiler thinks that the class in that header file is being declared multiple times
- It is necessary to use the instructions `#ifndef`, `#define` and `#endif` in our header files to avoid it

```
// C1.h
#ifndef _C1_H_
#define _C1_H_
...
class C1{
    ...
};
#endif
```

Exercises

Exercises (1/3)

Exercise 1

Implement the class in the following diagram:

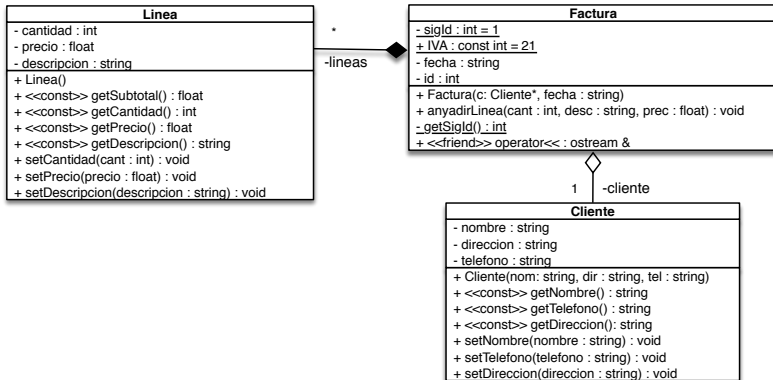
Coordenada
- x : float - y : float
+ Coordenada (cx: float=0, cy: float = 0) + Coordenada (const Coordenada &) + ~Coordenada() + <<const>> getX() : float + <<const>> getY() : float + setX (cx:float) : void + setY (cy:float) : void + <<friend>> operator << : ostream &

Create the files `Coordinate.cc` and `Coordinate.h`, and a *makefile* to compile them with a program `principal.cc`. The main function has to ask the user for two numbers and create with them a coordinate to print it with the output operator in the format `(x, y)`. Write the code necessary for each method to be used at least once.

Exercises (2/3)

Exercise 2

Implement the code corresponding to the following UML diagram:



Exercises (3/3)

Exercise 2 (continues)

Make a program that creates a new invoice, adds a product and prints it. From the constructor of `Factura` the method `getSigId` is called, which returns the value of `sigId` and increases it. Output example when printing an invoice:

```
Invoice No.: 12345
Date: 18/4/2011

Customer data
-----
Name: Agapito Piedralisa
Address: c/ Río Seco, 2
Phone: 123456789

Invoice details
-----
Line;Product;Quantity;Price per unit;Total price
--
1;USB mouse;1;8.43;8.43
2;RAM 2GB;2;21.15;42.3
3;Speakers;1;12.66;12.66

Subtotal: 63.39 €
VAT (21%): 13.3119 €
TOTAL: 76.7019 €
```