Institut Mines-Télécom
Didier GUERIOT (Dpt. ITI)
Johanne VINCENT (Dpt. INFO)
Reda BELLAFQIRA (Dpt. ITI)
Didier.Gueriot@imt-atlantique.fr
Johanne.Vincent@imt-atlantique.fr
Reda.Bellafqira@imt-atlantique.fr

Advanced C++ programming

C++03 C++11 C++14 C++17 C++20

- **Introduction to C++**
  - → C++: from C and beyond
  - → Classes, objects and lifetime (vs. JAVA)
  - Slot 3 → Oriented-Object Programming (inheritance, polymorphism)
- **Memory management & object manipulation**
  - → References, « copy » / « move » object construction
  - → Overloading operators
- **Template vs OO programming**
  - → Template functions and classes
- **The Standard Template Library**
  - → Containers, iterators and algorithms
  - → Using sequence & associative containers …
- **Smart pointers (STL & Boost)**

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Encapsulation (1/3)

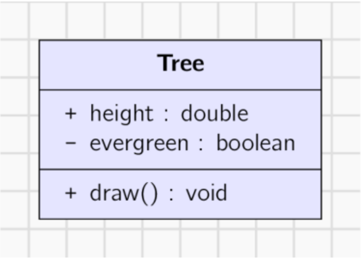
IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP 3

- **Tuning attributes & methods outside visibility**
  - Defines object data and actions that can be used from outside the class scope: **`public`** (access) / **`private`** (no access)
  - Class default visibilities: **`struct`** (full access: **`public`**), **`class`** (no access: **`private`**)

Tree.h

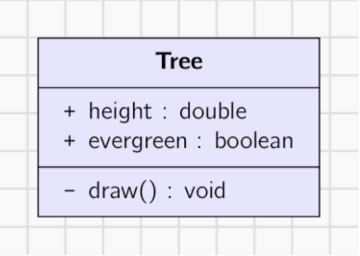
```
class Tree {
  bool   evergreen ;

public :
  double height ;

  void draw() ;
}
```




```
Tree a ;

a.draw() ;               // OK (method call)
a.height = 5.45 ;        // OK (write access)
cout << a.height         // OK (read access)
a.evergreen = true ;     // Error
cout << a.evergreen      // Error
```

Tree.h

```
struct Tree {
  double height ;
  bool   evergreen ;

private :
  void draw() ;
}
```

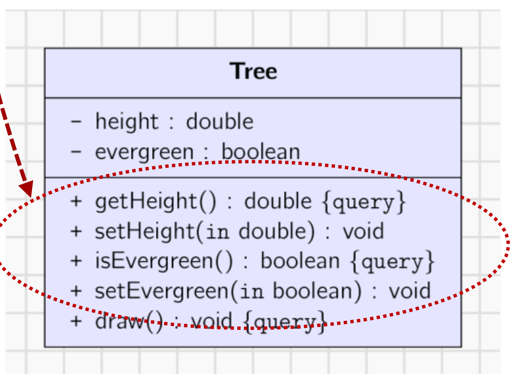


compilation

```
Tree a ;

a.draw() ;                // Error

cout << a.height          // OK
cout << a.evergreen       // OK
```

Encapsulation (2/3)

- **Access control to attributes**
  - Through **accessors** and **mutators**: controlling how data object are read or written from outside.
  - Allows data actual implementation to be hidden
    - ▸ only an interface to data is provided to outside user.



Tree.h

```
class Tree {
  bool   evergreen ;
  double height ;

public :
  void setHeight (double h) ;
  double getHeight () ;

  void setEvergreen (bool p) ;
  bool isEvergreen () ;

  void draw() ;
}
```

Tree.cpp

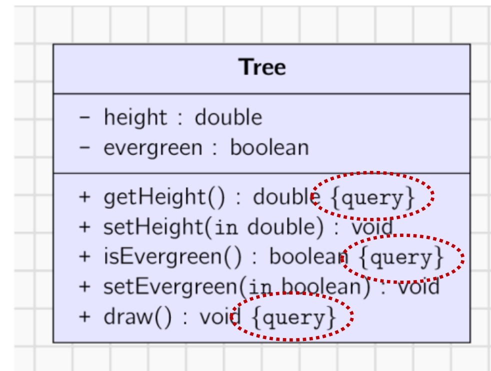
```
#include " Tree.h"

void Tree::setEvergreen (bool p) {
  evergreen = p ;
}

bool Tree::isEvergreen () {
  return evergreen ;
}

void Tree::setHeight (double h) {
  height = h ;
}

double Tree::getHeight () {
  return height ;
}
```

```
Tree a ;
a.height = 6.25 ;        // Error
cout << a.height ;       // Error
cout << a.getHeight()    // OK
a.setHeight (6.25) ;     // OK
```

compilation

Encapsulation (3/3)

- **Does a method change one object state ?**
  - The programmer can tell the compiler that a method should not change the object by contract (keyword `const`)
  - The compiler enforces that !



Tree.h

```
class Tree {
  bool   evergreen ;
  double height ;

public :
  void setHeight (double h) ;
  double getHeight () const ;

  void setEvergreen (bool p) ;
  bool isEvergreen () const ;

  void draw() ;
}
```

compilation

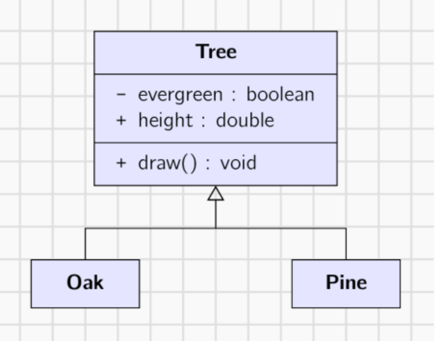
Tree.cpp

```
#include "Tree.h"

…

double Tree::getHeight const () {
  evergreen = false ;
  return height ;
}
```

Inheritance (1/5)

TP 8

- **Generalization : « is a » relationship between classes**
  - The **child** class (or derived class or subclass) inherits from the **parent** class (or super class or base class)
  - The child class acquires all properties (attributes) and functionalities (methods) of the parent
    - ▶ A pine « is a » tree but all trees are not pines !



Tree.h

```
class Tree {
  bool   evergreen ;

public :
  double height ;
  void draw() ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw() {
  return ;
}
```

Pine.h

```
#include "Tree.h"

class Pine : public Tree {
}
```

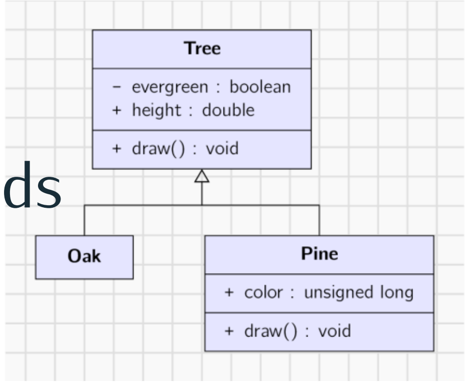
Pine.cpp

```
#include "Pine.h"
```

```
Tree a ;
Pine e ;

a.draw() ;
e.draw() ;   // OK

cout << a.height ;
cout << e.height ;    // OK
```

Inheritance (2/5)

- **Specialization of the derived classes**
  - Allows code redefinition of inherited methods
  - Allows adding new attributes and methods



Tree.h

```
class Tree {
  bool   evergreen ;

public :
  double height ;
  void draw() ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw() {
  cout << "Draw Tree" ;
  return ;
}
```

Pine.h

```
#include "Tree.h"

class Pine : public Tree {
public :
  unsigned long color ;
  void draw () ;
}
```

Pine.cpp

```
#include "Pine.h"

void Pine::draw() {
  cout << "Draw Pine" ;
  return ;
}
```

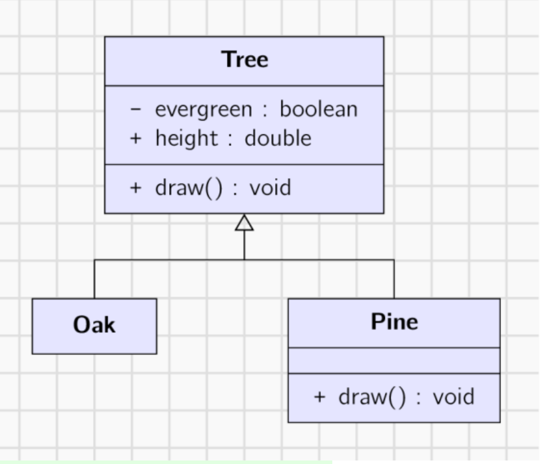
```
Tree a ;
Pine e ;

a.draw() ;
e.draw() ;    // OK (which one ?)

cout << a.height ;
cout << e.height ;     // OK

cout << e.color ;      // OK
cout << a.color ;      // Error
```

compilation

`color` is not an attribute of the `Tree` class

Inheritance (3/5)

- **Visibility of inherited attributes or methods**
  - From derived classes ?
  - From outside ?



Tree.h

```
class Tree {
  bool   evergreen ;

public :
  double height ;

  void draw() ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw() {
  cout << "Draw Tree"
    << evergreen << endl ;
  return ;
}
```

Pine.h

```
#include "Tree.h"

class Pine : public Tree {
public :
  void draw () ;
}
```

Pine.cpp

```
#include "Pine.h"

void Pine::draw() {
  cout << "Draw Pine"
    << evergreen << endl ;
  return ;
}
```

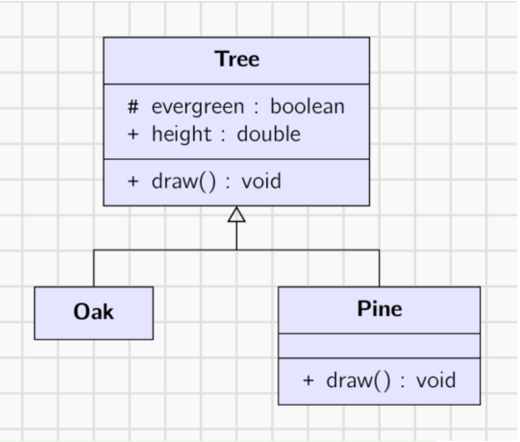
```
Tree a ;
Pine e ;

a.draw() ;
e.draw() ;          // Erreur

cout << a.height ;
cout << e.height ; // OK
```

compilation

evergreen is a private attribute of Tree class

Inheritance (4/5)

- **An extra keyword: « protected »**
  - Attributes & methods only visible by derived classes



Tree.h

```
class Tree {
protected :
  bool   evergreen ;

public :
  double height ;

  void draw() ;
}
```

Pine.h

```
#include "Tree.h"

class Pine : public Tree {
public :
  void draw () ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw() {
  cout << "Draw Tree"
    << evergreen << endl ;
  return ;
}
```

Pine.cpp

```
#include "Pine.h"

void Pine::draw() {
  cout << "Draw Pine"
    << evergreen << endl ;
  return ;
}
```

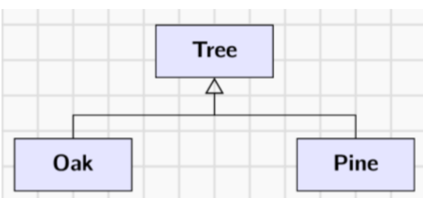
```
Tree a ;
Pine e ;

a.draw() ;
e.draw() ;              // OK

cout << a.height ;
cout << e.height ;      // OK

cout << e.evergreen ;   // Error
```

attribute `evergreen` is not visible from outside the hierarchy (`protected`) but derived classes from `Tree` can have unrestricted access.

compilation

Inheritance (5/5)

- **Constructors / Destructors call sequence**
  - When creating a `Pine` object, constructors from `Tree` to `Pine` are called: top-down sequence for initialization
  - When deleting a `Pine` object, destructors from `Pine` to `Tree` are called: bottom-up sequence for cleaning




```
class Tree {
public :
  bool   evergreen ;
  double height ;
}

class Pine : public Tree {
}
```

```
Pine e ;
```

```
class Tree {
protected :
  bool   evergreen ;
  double height ;
}

class Pine : public Tree {
  Pine (double h) :
    height(h), evergreen(true)
  {}
}
```

```
Pine e (100.0) ;
```

```
class Tree {
private :
  bool   evergreen ;
  double height ;
public :
  Tree (double h, bool e) :
    height(h), evergreen(e)
  {}
}

class Pine : public Tree {
public :
  Pine (double h) :
    Tree(h, true)
  {}
}
```

Inheritance vs. aggregation

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

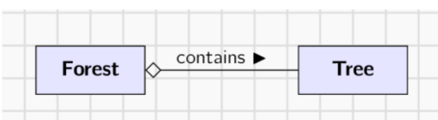
- **Aggregation: « has a » relationship (objects "inside" others)**
  - "Where" are the contained objects stored?
    - Actual inclusion or just references

```
struct Forest {
  Tree* trees ;
}
```

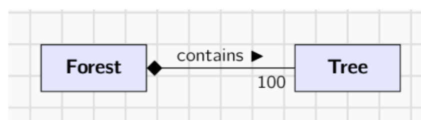
OR

```
struct Forest {
  Tree trees[100] ;
}
```

  - What's the lifetime of the "contained" objects?
    - Aggregation or Composition (can a tree live longer than the forest?)



OR




```
Forest::~Forest() {
  ???
}
```

- **Aggregation is not inheritance**

« A forest contains trees but is not a tree »

▶ **`Forest`** class does not inherit from **`Tree`** class

Function overloading (1/2)

- **Same function name with different signatures**
  - All should perform the same kind of task !
  - Returned value type is not part of the signature
  - **Warning** : calling the function should be unambiguous (the compiler is unable to choose for you)

Tree.h

```
class Tree {
public :
   void draw() ;
   void draw (int i , int j) ;
   void draw (double x , double y) ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw () { ... }
void Tree::draw (int i, int j) { ... }
void Tree::draw (double x , double y) { ... }
```

compilation

```
Tree a ;

a.draw () ;                  // OK
a.draw (10 , 20) ;           // OK
a.draw (15.55 , 34.67) ;     // OK

a.draw (10 , 34.5) ;         // Error
```

Ambiguous call ▶ two methods can be called :
1) `void draw (int i , int j)`
2) `void draw (double x , double y)`

Function overloading (2/2)

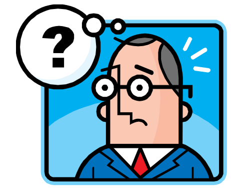
IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

- **NULL macro** (often `#define NULL 0`)
  - Replaced by preprocessor ▸ no type control!

```
void fn (int n) ;

void fn (char* s) ;
```

```
// Which 'fn' function
//    is called ?
fn (NULL) ;
```



- **A new "NULL" pointer value: `nullptr`**
  - Pointer value for an actual pointer pointing to nothing
    - ≠ NULL which is often integer 0
    - ≠ `void*` which is an untyped pointer (just one address memory)
  - `nullptr` has a type `std::nullptr_t`
    - Easy overloading for calling functions with pointer to nothing

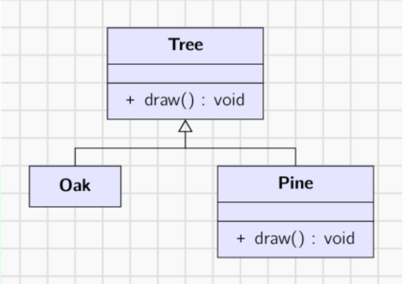
```
#include <cstddef>

void fn (std::nullptr_t p) ;
```

```
fn (nullptr) ;
```

Function overriding (1/2)

TP 9

- **Calling a method from a pointer (or a reference)**
  - Same method name with same signature (within a class hierarchy)
  - Static by **default** with C++ (≠ JAVA)
  - Compile-time decision ▸ only the pointer type is used to decide which method to call (from the hierarchy)



Tree.h

```
class Tree {
public :
  void draw() ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw() {
  cout << "Draw Tree" ;
}
```

Pine.h

```
#include "Tree.h"

class Pine : public Tree {
public :
  void draw () ;
}
```

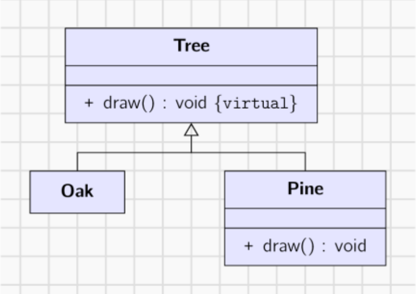
Pine.cpp

```
#include "Pine.h"

void Pine::draw() {
  cout << "Draw Pine" ;
}
```

```
Tree a ;
Pine e ;

// pa type is "pointer to a tree"
// pa points to a pine : no problem as
//   a pine is a tree
Tree* pa = &e ;

a.draw() ;   // Tree::draw()
e.draw() ;   // Pine::draw()

// Tree::draw() or Pine::draw()?
pa->draw() ;     // OK but which one?
(*pa).draw() ;   // OK but which one?
```

Function overriding (2/2)

- **Calling a method from a pointer (or a reference)**
  - Same method name with same signature (within a class hierarchy)
  - If polymorphism is required (JAVA standard behavior)
    - Runtime decision ▸ the type of the actual pointed object is used to decide which method to call from the hierarchy
    - "**virtual**" must be added to the method signature



Tree.h

```
class Tree {
public :
  virtual void draw() ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw() {
  cout << "Draw Tree" ;
}
```

Pine.h

```
#include "Tree.h"

class Pine : public Tree {
public :
  void draw () ;
}
```

Pine.cpp

```
#include "Pine.h"

void Pine::draw() {
  cout << "Draw Pine" ;
}
```

```
Tree a ;
Pine e ;

Tree* pa = &e ;

a.draw() ;
e.draw() ;

// Which one will
//     be called?
pa->draw() ;
(*pa).draw() ;
```

```
Draw Tree
Draw Pine
Draw Pine
Draw Pine
```

Add "virtual" to destructors too !!!

Pure virtual method & abstract class

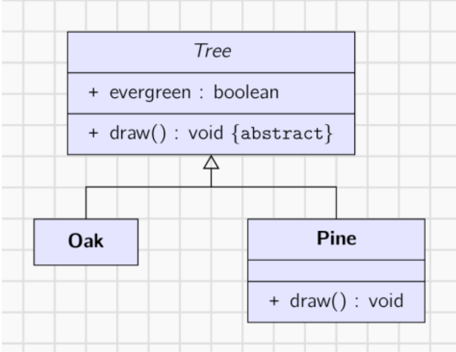
- **A pure virtual method has no implementation**
  - Its class becomes an abstract class ▸ no object can be created as the method implementation is missing
  - Other methods may be implemented including constructors

Tree.h

```
class Tree {
public :
  bool evergreen ;
  virtual void draw() = 0 ;
}
```

```
Tree a ;
a.draw() ;
```

compilation



Pine.h

```
#include "Pine.h"

class Pine : public Tree {
public :
  void draw () ;
}
```

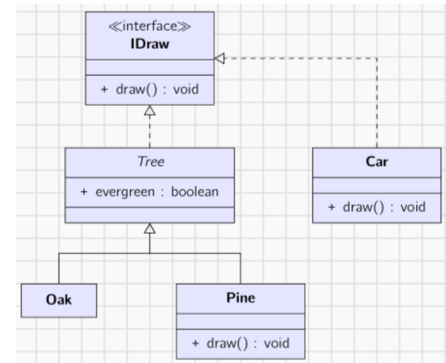
Pine.cpp

```
#include "Pine.h"

void Pine::draw() {
  cout << "Draw Pine" ;
  return ;
}
```

```
Pine e ;
Tree* pa = &e ;

pa->draw() ;
e.draw() ;
```

No instance of class `Tree` can be created as the `draw` method has no implementation

Interface & abstract class



- **How to define an interface (≡ JAVA)?**
  - Abstract class specifying a list of methods to be implemented by the derived classes
    - Derived classes must respect this **contract** to be used through it, bypassing their actual type
      - Here « to be drawable » ≡ implement a method called `draw()`
      - Derived classes (`Tree`, `Pine`, `Car`) may be abstract or not

IDraw.h
```
class IDraw {
public :
  virtual void draw() = 0 ;
}
```

Tree.h
```
#include "IDraw.h"
class Tree : public IDraw {
}
```

Tree.cpp
```
#include "Tree.h"
```

Pine.h
```
#include "Tree.h"
class Pine : public Tree {
public :
  virtual void draw() ;
}
```

Pine.cpp
```
#include "Pine.h"

void Pine::draw() {
  cout << "Draw Pine" ;
}
```

Car.h
```
#include "IDraw.h"
class Car : public IDraw {
public :
  virtual void draw() ;
}
```

Car.cpp
```
#include "Car.h"

void Car::draw() {
  cout << "Draw Car" ;
}
```

```
Tree a ;
a.draw() ;
```
compilation

```
Pine e ;
Tree* pa = &e ;
IDraw* pp = &e ;

e.draw() ;
pa->draw() ;
pp->draw() ;
```

```
Car c ;
IDraw* pc = &c ;

c.draw() ;
pc->draw() ;
```

JAVA / C++ mechanisms

- JAVA
  - Single inheritance (`inherits`)
  - Interface implementation (`implements`)
- C++
  - Multiple inheritance
  - Virtual methods (polymorphism)
  - Pure virtual methods (abstract class, interface)
  - No C++ syntaxis specificities between "interface" and abstract class ▸ programmer intentions make the difference

The compiler helps you!

C++11

- New keywords (to use)
  - **override**: to indicate a virtual method is redefined

```
struct A {
  virtual void foo() ;
  void bar() ;
} ;
struct B : A {
  void foo() const override ;
  void foo() override ;
  void bar() override ;
} ;
```

```
B::foo does not override A::foo (signature mismatch)
B::foo overrides A::foo
B::bar is not virtual
```

compilation

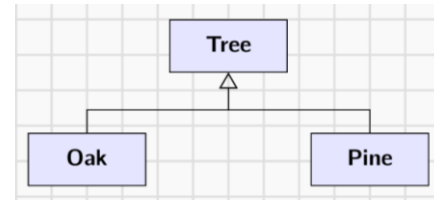
  - **final**: no more derived classes / virtual method redefinition

```
struct A {
  virtual void foo() final ;
  void bar() final ;
} ;
struct B final : A {
  void foo() ;
} ;
struct C : B {
} ;
```

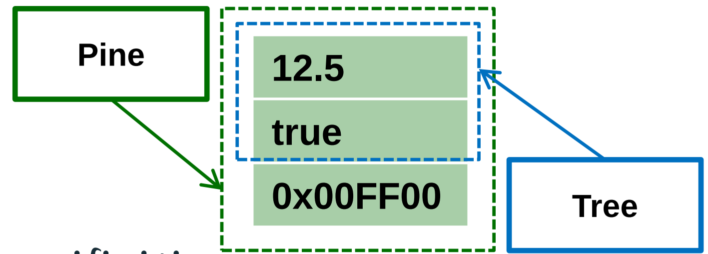
```
A::foo is final
non-virtual function A::bar cannot be final
struct B is final
foo cannot be overridden as it's final in A
B is final
```

compilation

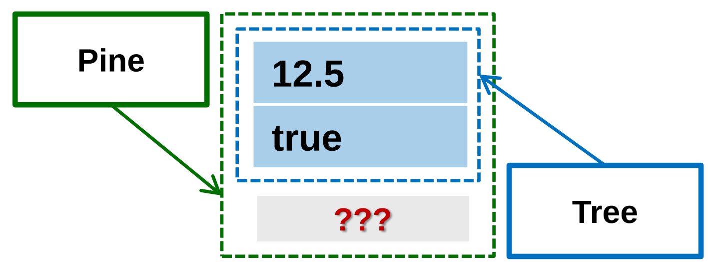
Type casting (1/3)



- **Upcast**
  - Consider an object of a derived class as an object of a base class: ***A pine is a Tree***
  - Always possible :
    - implicit type casting!
    - but no further access to derived class specificities

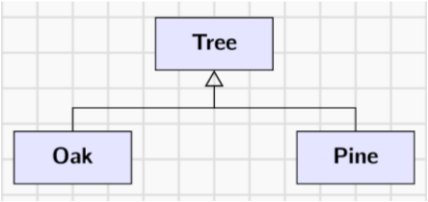


- **Downcast**
  - Consider an object of a base class as an object of a derived class: ***Is a Tree a Pine?***
  - It's your decision ▸ downcasting must be asked explicitly by the programmer



Type casting (2/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom



- Dynamic (**`dynamic_cast`**)
  - Check at **runtime** if the cast is valid
    - non usable on base types except **`void*`**
    - fit for polymorphism ▸ actually, **`pa`** points to a type **`Pine`** object even if **`pa`** is a pointer to **`Tree`** (base class): valid downcast

```
Tree* pa = new Pine ;
Pine* pe = dynamic_cast<Pine*>(pa) ;
```

    - if invalid cast: returns null pointer or raises **`bad_cast`** exception

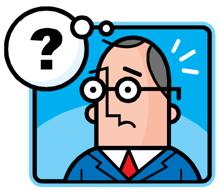
- Static (**`static_cast`**)
  - No "dynamic" type checking at runtime
    - Faster if the programmer **knows for sure** the actual object type

```
int k = static_cast<int>(12.5) ;
```

```
Tree* pa = new Pine ;
Pine* pe = static_cast<Pine*>(pa) ;
Oak*  ps = static_cast<Oak*>(pa) ;
```

Type casting (3/3)

- Bypassing read-only (`const_cast`)
  - Only removes the constness of a variable / pointer / reference (and it is **the only cast** that can do it)
  - Bypass original programmer intentions
    - do you really want to do it? design flaw on your side ?



```
Tree a ;
const Tree* pa1 = &a ;                   // OK : implicit cast
Tree* pa2 = const_cast<Tree*>(pa1) ;     // explicit cast required to compile
```

- Reinterpretation (`reinterpret_cast`)
  - Consider an object to be from another type (no check at all)
  - **« Everything » is possible**

```
Tree a ;
Car* pv = reinterpret_cast<Car*>(&a) ;
```

    - May be dangerous but sometimes useful (data mapping)
    - Undefined behavior in case of polymorphism

Practice

- **Define and use one class using POO concepts**
  - Encapsulation
  - Overloading
- **What about deriving classes?**
  - Inheritance
  - Polymorphism
- **Going further…**
  - Abstract classes & « interface »
  - Type casting