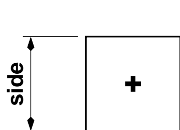
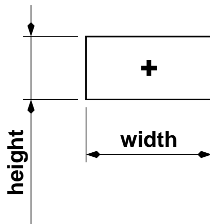
Cours de C++

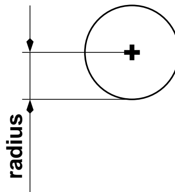
Héritage

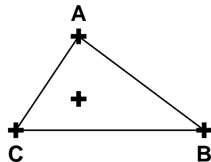
Cécile Braunstein
cecile.braunstein@lip6.fr

Basic cases

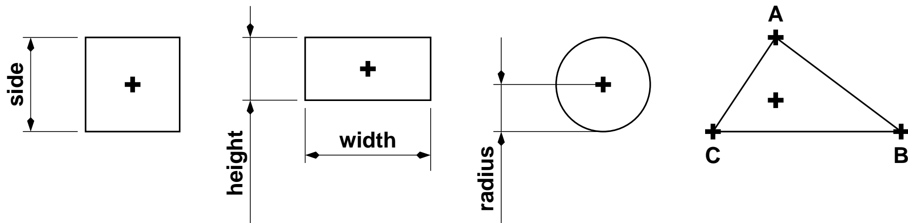








Basic cases



Square
_center
_side
GetCenter()
Draw()
Erase()

Rectangle
_center
_width
_height
GetCenter()
Draw()
Erase()

Circle
_center
_radius
GetCenter()
Draw()
Erase()

Triangle
_center
_pointA
_pointB
_pointC
GetCenter()
Draw()
Erase()

Look more closer

Square
_center
_side
GetCenter()
Draw()
Erase()

Rectangle
_center
_width
_height
GetCenter()
Draw()
Erase()

Circle
_center
_radius
GetCenter()
Draw()
Erase()

Triangle
_center
_pointA
_pointB
_pointC
GetCenter()
Draw()
Erase()

Look more closer

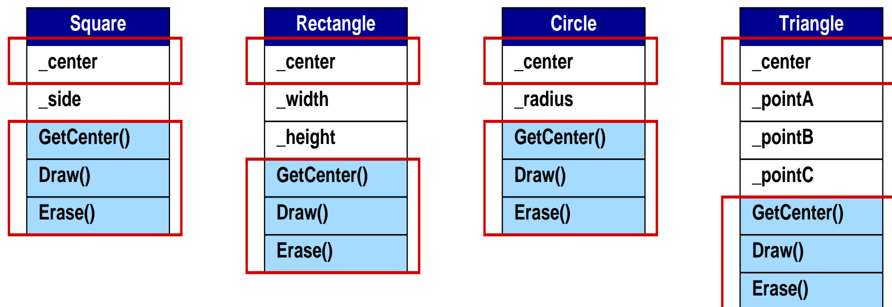
Square
_center
_side
GetCenter()
Draw()
Erase()

Rectangle
_center
_width
_height
GetCenter()
Draw()
Erase()

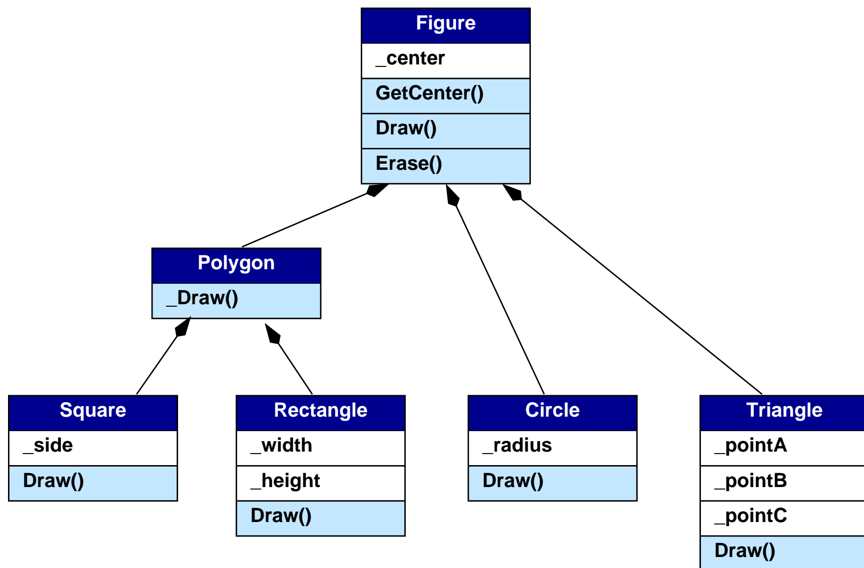
Circle
_center
_radius
GetCenter()
Draw()
Erase()

Triangle
_center
_pointA
_pointB
_pointC
GetCenter()
Draw()
Erase()

Look more closer



Class hierarchy



Protection revisited

```
class Figure {
 private :
 Point _center;

 public :
 Figure(Point& center);

 Point& GetCenter();
 void Draw();
 void Erase();

};
```

Protection revisited

```
class Figure {
 protected :
 Point _center;

 public :
 Figure(Point& center);

 Point& GetCenter();
 void Draw();
 void Erase();

};
```

More about protection

Let $\mathcal{B}$ and $\mathcal{C}$ be two classes such that $\mathcal{C}$ derived from $\mathcal{B}$ *publicaly*.

`private` and `public`

- `private` members of $\mathcal{B}$: Only class $\mathcal{B}$ may access to these members
- `public` members of $\mathcal{B}$: Everyone may access to these members

`protected`

- $\mathcal{B}$ and $\mathcal{C}$ have access to these members
- They are still part of the interface
- Users of class $\mathcal{C}$ can not have direct access to these members

Composition of protection

3 types of inheritance

- `public` : Like the definition of a sub-type.
- `private` or `protected` : Hide details of the implementation

Change access to the class members

		Members of the base class		
		public	protected	private
Derived class	public	public	protected	no access
	protected	protected	protected	no access
	private	private	private	no access

Constructor

Run of the constructor for derived object

1. Allocating memory space for the entire object (base-class + derived-class members)
2. Calling the base-class constructor to initialize the base-class part of the object
3. Initializing the members of the derived class as directed by the constructor initializer
4. Executing the body of the constructor, if any

Constructors of the base-class are always called.

Destructor

Run of the destructor for derived object

1. Executing the body of the destructor, if any
2. Destroying the members of the derived class as directed by the destructor in the opposite order
3. Calling the base-class destructor
4. Deallocating memory space for the entire object (base-class + derived-class members)

Destructors of the base-class are always called.

Example - base class

```
class student{
protected:
double final, partiel;
std::vector<double> homework;
public:
student();
student(std::istream&);
std::string name() const;
std::istream& read(std::istream&);
double grade();
};

bool compare(const student& s1, const student& s2);
bool compare_grade(const student& s1, const student& s2)
    ;
```

Example - derived class

```
class stagiaire: public student{
private:
  double stage;
public:
  stagiaire();
  stagiaire(std::istream&);
  double grade() const;
    std::istream& read(std::istream&);

};
```

Example - constructor

student constructor

```
student::student():final(0):midterm(0){}

student::student(std::istream& in){
  read(in);
}
```

stagiaire constructor

```
stagiaire::stagiaire():stage(0){}

stagiaire::stagiaire(std::istream in){
  read(in);
}
```

Example - constructor

student constructor

```
student::student():final(0):midterm(0){}

student::student(std::istream& in){
  read(in);
}
```

stagiaire constructor

```
stagiaire::stagiaire():stage(0){}

stagiaire::stagiaire(std::istream in){
  read(in);
}
```

Example - constructor

student constructor

```
student::student():final(0):midterm(0){}

student::student(std::istream& in){
  read(in);
}
```

stagiaire constructor

```
stagiaire::stagiaire():stage(0){}

stagiaire::stagiaire(std::istream in){
  read(in);
}
```

Example - use

Constructor

```
stagiare g1(cin);
stagiare g2(cin);

student e1(cin);
student e2(cin);
```

Function call

```
bool compare(const student& s1, const student& s2){
 return s1.name() < s2.name();
}

compare(g1,g2);
compare(e1,e2);
compare(g,e);
```

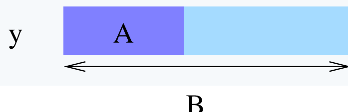
Static cast

Type known at compile time

```
class A {...} ;
class B : public A {...};
```

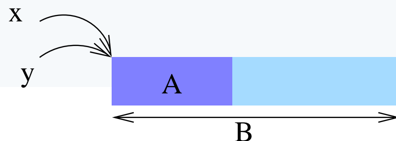
Object

```
B y;
A x = y;
```



Pointer and reference

```
B* y;
A* x = y;
```



Dynamic cast 1

Type known at run time

```
class A {...} ;
class B : public A {...};
```

Only for references pointers

```
B x;
A y = x;
A *ptry = &x;
A &refy = x;
```

- the static type of `*ptry` and `refx` is `A`
- the dynamic type of `*ptry` and `refx` is `B`

Dynamic cast 2

Syntax

```
dynamic_cast<T*>(p)
```

- `p` is a pointer
- Transform the type of `p` in `T`
- If it's not possible returns `NULL`

```
dynamic_cast<T&>(p)
```

- `p` is a reference
- Transform the type of `p` in `T`
- If it's not possible raise an exception

An other example

Comparing grade

Sometimes, we really want to know the real type at run time.

```
bool compare_grade(const student& s1, const student& s2)
{
 return s1.grade() < s2.grade();
}

student e1,e2;
stagiaire s1,s2;
compare_grade(e1,e2);
compare_grade(s1,s2);
```

How to be sure that the right function `grade` is used ?

Polymorphism

Definition

Polymorphism defines the notion that a the behavior of an object doesn't have to be known at compile time. The real object type may be known at run time.

What for ?

- Build container with not exactly same type element inside
- For the destructor
- Re-use the code for an other application

virtual function

```
class student{
public :
 virtual double grade() const;
 //...};
```

Virtual function

When ?

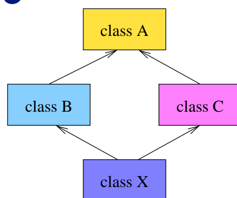
- Calling a function that depends on the actual of object
- Making this decision at run time

How ?

- Keyword `virtual` used only inside the class definition
- Do not need to repeat this keyword in the implementation
- When it's inherited, no need to repeat this keyword

➛ A destructor has to be virtual

Multiple inheritance



Derived from many classes

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class X : public B, public C { /* ... */ };
```

- The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.
- A derived class can inherit an indirect base-class more than once

Leads to ambiguities

Resolving ambiguities

Members with same names from different classes

- C++ compilers resolves some ambiguities by choosing the minimal path to a member
- Use the scope operator `A::function`

Two same members from different class

- Sometimes it's the correct behavior
- Virtual inheritance

```
class A { /* ... */ };
class B : public virtual A{ /* ... */ };
class C : public virtual A{ /* ... */ };
class X : public B, public C { /* ... */ };
```