# Cours de C++ 1

## Les Classes et les Objets

Cécile Braunstein cecile.braunstein@lip6.fr

Automne 2018

# Généralité

## Notes

| | |
|---|---|
| TP, Interros cours | 30% |
| Contrôles TP $\times 2$ | 35% |
| Mini-projet | 35% |

## References

- B. STROUSTRUP, Programming: principles and practice using C++, Pearson Education,2010
- A. KOENIG, B. MOO, Accelerated C++, Addison Wesley 2006
- S. MEYER, Effective C++, Addison Wesley 2005

- `http://www.cplusplus.com`
- `http://www.cppreference.com`
- `http://www.qwant.com`

- @meetingcpp
- @Scott_Meyer
- @c_plus_plus
- @CppCast

# Why C++ ?

## C++

- Middle-level language
- Developed by Bjarne Stroustrup in 1979 at Bell Labs
- First view as an extension of C (*C with classes*)

## Philosophy

- C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
- C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
- C++ is object language it implies :
  - Re-use
  - Modularity
  - Maintainability

# When to use it ?

- Want to be fast and need of abstraction
- System porgraming
- low-level programing

But it can be hard

- Complex code
- Handling the memory
- Segmentation fault
- Mix C/C++
- . . .

Part I

# The very first example

# Hello world !

helloworld.cpp

```
// The first programm

#include <iostream>

int main()
{
  std::cout << "Hello, world !" << std::endl ;
  return 0;
}
```

# Call the compiler

```
g++ -Wall -g helloworld.cpp -o hello
```

## Compile Options

g++ accepts most options as gcc

- Wall : all warnings
- g : include debug code
- o : specify the output file name (`a.out` by default)

# Program details

## #include

Many fundamentals facilities are part of standard library rather than core language

```
#include <iostream>
```

**#include** directive + angle brackets refers to standard header

## main function

- Every C++ program must contain a function named `main`. When we run the program, the implementation call this function.
- The result of this function is an integer to tell the implementation if the program ran successfully
  Convention :

$$0 : \textit{success} \quad | \quad \neq 0 : \textit{fail}$$

# Using the standard library for output

```cpp
std::cout << "Hello, world !" << std::endl;
```

- `<<` : output stream operator
- `std::` : namespace `std`
- `std::cout` : standard output stream
- `std::endl` : stream manipulator

# Namespace

## Purpose

- Collection of identifier (variable name, type name . . . )
- Avoid name conflict :

```
int cout = 2;
std::cout << cout << std::endl;
```

## Declaration

```
namespace myNamespace
{
  int a, b;
}
```

*Usage:*

```
myNamespace::a
myNamespace::b
```

# Scope example

```
// namespaces
#include <iostream>
using namespace std;

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << endl;
  cout << bar::value() << endl;
  cout << bar::pi << endl;
}
```

# Using namespace

**Tell the compiler which name you are using.**

(These lines should be included in the general part of your program)

- Refer to the a specific name of the standard library

```
using std::cout;
```

- Refer to all the names of a namespace

```
using namespace std;
```

# Using example

```
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}
namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}
int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

# Expressions in C++

## Type

- A variable is an object that has a name.
- An object is a part of the memory that has a type.
- Every object, expression and function has a type.
- Types specify properties of data and operations on that data.

**Primitive types**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Type | bool | char | int | float | double | void | wchar_t |
| Modifier | signed | | unsigned | | short | long | long long |

To improve the code re-use it is important to use the right type at the right place !

# Variable definition

## Local variable

- Variable can be define anywhere in the program.
- local variable are destroyed when an end of block is reached.

```
{
  std::string name; // var creation
  std::cin >> name // var life
  std::cout << "Hello " << name << std::endl;
} // var death
```

- Variable has a type and an interface

## How to define a variable

| | |
|---|---|
| *type-name name;* | (definition) |
| *type-name name = value;* | (definition + initialization) |
| *type-name name(args);* | (definition + initialization) |

# Declaration vs. Definition

## Declaration

Tells the compiler about the name and the type of something

```
extern int x;                    // object declaration
size_t numDigit(int number); // function declaration;
class Widget;                    // Class declaration
```

## Definition

Provides the compiler with details :

- set size of memory,
- code body,
- initialization,
- ...

# Variable default-initialization

```
#include <iostream>
#include <string>
using namespace std;
int main(){
  int a = 2;
  int b(4);
  int c;
  cout << a << "  " << b << "  "
       << c << endl;

  string name;
  string surname("Max");
  cout << name << "  " <<
    surname << endl;
  return 0;
}
```

## Rules

- Class type
  - Always initialized
  - Implicit initialization → call default constructor
  - string : implicitly empty ("\ 0")
- Primitive type
  - No implicit initialization
  - Variable may be undefined

# Constant

```
const unsigned int SIZE_MAX = 15 ;
```

## Purpose

Keyword `const` :

- Part of a variable's definition
- The variable must be initialized as part of its definition

Use :

- Promise that the value of the variable is unchanged during its lifetime
- Make program easier to understand : A name give more information than a value
- May be used as global parameters

# Pointers

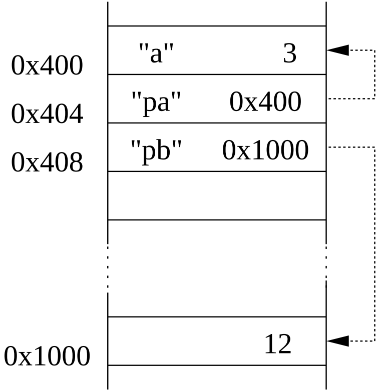
## Definition

A pointer is a value that represents the address of an object.

Every distinct object has a unique address. It's the the part of the computer's memory that contains the object.

```
int main()
{
    int a = 3;
    int *pa ;
    int *pb ;

    pa = &a;
    pb = (int*)malloc(sizeof(int)) ;
    *pb = 12
    return 0;
}
```



# Operators on pointers

| | |
|---|---|
| `&x` | : address operator |
| `*px` | : dereference operator |
| `T* p` | : declaration of a pointer to `T` (`*p` has a type `T`) |
| `NULL` | : constant value, differs from every pointer to any object |

# Operations on pointers

## Exercice

What is the output of this program. We assume that `&x = 0xbf84e7b8`

```
#include<iostream>
using namespace std;
int main(){
    int x = 5;
    int* p = &x;
    cout << "x = " << x << endl;
    cout << "p = " << p << " ; *p = " << *p << endl;

    *p=6;
    p = p + 1;
    cout <<"x = " << x << endl;
    cout << "p = " << p << " ; *p = " << *p << endl;
    return 0;
}
```

# Constant and pointers

```
char greeting[] = "Hello";
char * p = greeting          // non-const pointer,
                             // non-const data

const char * p = greeting    // non-const pointer,
                             // const data

char * const p = greeting    // const pointer,
                             // non-const data

const char * const p = greeting // const pointer,
                                // const data
```

# Arrays

## Definition

- Part of the core language
- Sequence of one ore more objects of the same size
- The number of elements must be known at compile time

An array is not a class type

## Good use

```cpp
double coords[3];
```

```cpp
const size_t NDim = 3;
double coords[NDim];
```

- The constant is known at compile time.
- Better for documentation purpose.

# Array initialization

```
const int DIM = 3;
double tab[DIM] = {1,2,3};

double number[] = {1,2,3,4,5,6};

const in month_length[] = {
31, 28, 31, 30, 31, 30,
31, 31, 30, 31, 30, 31
};
```

## Number of elements

- The size may be implicit :

```
size_t  n= sizeof(number)/sizeof(*number);
```

But always known at compile time

# Memory management

## Three kinds

1. Automatic management: system's job
2. Static allocation: once and only once
3. Dynamic allocation: with respect to our needs

# Automatic memory management

## Local variables

- The program allocates memory when it encounters the definition of the variable
- The program deallocates that memory at the end of the block containing the definition.
  - ➢ Any pointers to this variable become invalid

```
int* invalid_pointer ()
{
    int x;
    return &x; // never !
}
```

# Static allocation

```cpp
int y;
int * pointer_to_static ()
{
    static int x;
    return &x;
}
```

## Global/static variables

- x,y are allocated once and only once before the function call.
- x,y are deallocated only at the end of the run.
- x,y are initialized only once: the first time the program run encounters the definition.
- The function always return the same address.

# Life time example

```cpp
int a = 1;
void f()
{
  int b = 1;
  static int c = a;
  cout << " a = " << a++
       << " b = " << b++
       << " c = " << c << endl;
  c = c + 2;
}

int main()
{
  while( a < 4) f() ;
  return 0;
}
```

# Dynamic allocation

## Allocate **new**

```
int* p = new int(42);
```

- Allocate new objects of type **int**
- Initialize the object to 42
- Cause `p` to point to that object

The object stays around until it is deleted or the program ends.

## Deallocate **delete**

```
delete p;
```

- Frees space memory used by `*p`
- Invalids `p`
- **delete** only object created by **new**

Deleting a zero pointer has no effect.

# Dynamic allocation example

```
class mine
{
  int m;
public:
  mine(int x):m(x){cout << "m(" << m << ") created" << endl;};
  ~mine(){cout << "m(" << m << ") destroyed" << endl;};
};
void f()
{
  mine m(42);
  mine * p = new mine(24);
  cout<< "END OF F" <<endl;

}
int main()
{
   f();
   cout<< "AFTER RETURN OF F" <<endl;
   cout<< "END OF MAIN" <<endl;
  return 0;
}
```

# Allocating and deallocating an array

```
T* p = new T[n]
delete[] p
```

### Allocation

- Allocates and default-initializes an array of n places
- Returns a pointer to the first element in the array

### Deallocation

- Destroys the objects in the array
- Frees the memory used to hold the array
- Invalids the pointer p;

# Multidimensional arrays

## Allocate

```cpp
int n = 4;
int (*M)[3]=new int[n][3];
// n lines of 3 columns
```

```cpp
int n2 = 4;
type** M = new type*[n] ;
for(int i=0 ; i< n ; ++ i)
{
    M[i] = new type[n2] ;
}
// n lines of n2 columns
```

## Deallocate

```cpp
delete[] M;
```

```cpp
for(int i=0 ; i< n ; ++ i)
{
    delete[] M[i] ;
}
delete[] M;
```

# References

```
int i;
int &r = i;

int j = 2;
r = j;
```

## New in C++

- A reference is a pointer self-dereferenced
- It acts as a synonym for the refered variable
- It's an address but after initialization all operation affect the pointed variable

## Useful ? Yes !

- Give a specific name to no-name element (table element)
- Use in a parameter list of function.

# References vs. Pointers

```cpp
#include<iostream>
using namespace std;
void increment(int& v)
{
    v++;
}
int main(){
 int a = 3 ;  int* pa;
 int & ra = a;
 pa = &a ;  ra = 4;
 increment(a);

 cout << "a  = " << a <<" &a = " << &a<< endl;
 cout << "*pa = " << *pa << " pa = " << pa << endl;
 cout << "ra = " << ra <<" &ra = "<< &ra << endl;
 return 0;
}
```

# References - other examples

Explain the following lines :

```cpp
double d;
const double d_const = 4.0;
double &a = d;
const double &b = d;
double &c = d_const;
const double &c_const = d_const;
```

# References - other examples

Explain the following lines :

```cpp
double d;
const double d_const = 4.0;
double &a = d;
const double &b = d;
double &c = d_const;
const double &c_const = d_const;
```

## Example

```cpp
double d; // declare a double
const double d_const = 4.0; // declare a const double
double &a = d; // a is a synomym for d
const double &b = d; // b is a read-only synonym for d
double &c = d_const; // This is not possible
const double &c_const = d_const; // c_const is a synonym for d_const
```

# Parameters

## Example

Computing student's grade

```
double grade(double midterm, double final, double homework)
{
  return 0.2 * midterm + 0.4 * final  + 0.4 * homework;
}
```

## Parameters list

Behaves like local variables to the function :

- Calling the function : create the variables
- Returning from the function : destroy the variables

# Call by value

```
std::cout << "Your final grade is : " << setprecision(3)
          << grade(midterm,final,sum/count)
          << setprecision(prec) << std::endl;
```

## Arguments

- Arguments can be a variable or an expression.
- Each argument is used to initialize the corresponding parameters
- The parameters take a copy of the value of the argument

# Call by reference

We want to have a function that returns two values at once.

```
int function_f(int a,int& b)
{
    r = a + b
    b = b + 1;

    return r;
}
```

## Reference

- Fast : only the address is in `b`.
- No copy
- The function will modify `b`
- The compiler manages the operators "*" and "&"

# Call by `const` reference

```cpp
int  function_f(int  a,const int& b)
{
   r  = a + b

   return r;
}
```

`const`

- Direct access to the argument
- No copy of the argument
- Promise we will not change the value

# Resume

| Call by | const ref | value | ref |
|---|---|---|---|
| | `void f(const string &a)` | `void f(string a)` | `void f(string &a)` |
| modification of `a` | No | local | with side effect |
| accepted values | All | All | non-temporary |
| advantages | security<br>no copy | simple | more general<br>no copy |

# Part II

# Object Oriented Programming

# Organizing programs and data

## Thinking big

To keep larger programs manageable, we need break it into independents named parts.

Fundamental ways of organizing program :

- Functions
- Data structure
- Class : combine Functions and data structure

## And then ...

- Divide program into files
- Compile separately
- Write Makefile

# Programmation oriented object (POO)

## Advantages

- Re-use
- Modularity
- Maintainability

## Oriented object language

Before :

- Data more or less well organized
- Functions and computation applied on these data
- A program is a following of affectation and computation

POO :

- Modules (*classes*) representing data and functions
- A program is a set of *objects* interacting by calling their own functions(*methods*)

# Concepts

## Objects

An object is a recognizable element characterized by its structure (*attributes*) and its behavior (*methods*)

➛ Object = Class instance

## Class

Groups and creates objects with the same properties (method and attributes).

Class members :

- Attributes : define the domain of value
- Methods : define behavior ; set of function modifying the state of an object

A class has got at least two methods (create and delete) - *may be implicit*

# Information hiding

## Purpose

Restrict access to a class by its interface

- Put constraints for the use and the interaction between objects.
- Programmer see only a part of the object corresponding to its behavior
- Help updates and changes for a class.

## Class has two parts

- An interface : access for external users,
- Internal data and internal implementation.

# Defining new types in C++

```
class Rectangle{
    double _h;
    double _w;
public:
    std :: istream& read(std::istream&);
    double area() const;
};
```

An object Rectangle is made of memory composed by :

- 2 double numbers
- 2 functions
- default constructor and destructor.

Usually written in a header file.

## Create interface

Our Goal :

- Hiding implementation details
- Users can access only through functions

# New style

```cpp
#include <iostream>

class Rectangle{
  double h;
  double w;
public:
  std::istream& read(std::istream &in){  in  >> h >> w; return in ;}
  double area() const {return h * w;}
};

int  main(){
  Rectangle my_rect;
  my_rect.read(std::cin) ;
  std::cout << "Area:  " << my_rect.area() << std::endl;
  return 0;
}
```

# Protection - Data Encapsulation

```
class Rectangle{
public:
  // interface
    void set_rectangle(double,double);
    bool is_higher(const Rectangle& r) const {return h > r.h;}
    double area() const;
    std :: istream& read(std::istream&);

private:
  // implementation
    double _h;
    double _w;
};
Rectangle p,q;
```

# Protection label

Each protection label defines the accessibility of all members that follow the label.

### labels

They can appear in any order

- `private` : Inaccessible members from outside
- `public` : accessible members from outside

### `struct` or `class` ?

There is no difference except :

- default protection : private for a class ; public for struct.
- by convention : struct for simple data structure

# Member functions - Definition

**read**

```
istream& Rectangle::read(istream& in)
{
    in >> _h >> _w;
    return in;
}
```

Usually implemented in the source files

## Particularities

- The name of the function `Rectangle::read`
- No object `Rectangle` in parameters list
- Direct access to data elements of our object

# Member functions

## `area`

```
double Rectangle::area() const
{
   return _h*_w;
}
```

**What's new ?**

- `area` is a member of `Rectangle` : implicit reference to the object
- and `const` ?

# Const member function

```cpp
double Rectangle::area() const {...}  //new
double area(const Rectangle&) {...} //old
```

## Const

- In the old version we ensure that the grade function do not change the parameter
- In the new version, the function is qualified as `const`
- `area` can be applied to a `const` or no`const` object
- `read` cannot be call by a `const` object

# Member functions

**`is_higher`**

```cpp
bool is_higher(const Rectangle& r) const {return _h > r._h;}
```

## Inline function

- To avoid function call overhead, we can *inline* funciton
- Ask the compiler to replace the call by the code if it's possible.

# Life cycle of an object

## Run of the constructor for derived object

1. Allocating memory space for the entire object (base-class + class members)
2. Calling the base-class constructor to initialize the base-class part of the object
3. Initializing the members following the declaration order.
4. Executing the body of the constructor, if any

Constructors of the base-class are always called.

# Constructor

## Definition

- Special member functions that defines how object are initialized.
- If no constructor defined the compiler will synthesized one for us.
- They have the same name as the name of the class itself.
- They have no return type and no return instruction.

```
class Rectangle{
    Rectangle();  //construct an empty object
    Rectangle(std::istream&);  //construct by reading a stream as before
    Rectangle(double h, double w); //construct with given initial value
};
```

# Call the constructor

When an object is created, a call to the constructor is always performed.

## How to call it ?

```
//Basic form
Rectangle a(2,3);
Rectangle b = Rectangle(5,7);

// Constructor with only one parameter
Rectangle c = cin; //     Rectangle c = Rectangle(cin)

// Dynamic allocation initialisation is not mandatory
Rectangle* d = new Rectangle(1,5);

// For anonymous object
cout << Rectangle(3,4).is_higher(Rectangle(2,7)) << endl;

//Default constructor
Rectangle e; // Rectangle e = Rectangle()
Rectangle f[10] // 10 calls of Rectangle()
```

# The default constructor

## Implementation

The one without argument.

```
Rectangle::Rectangle():_h(0),_w(0) {}
```

### Constructor initializer

When we create a new class object :

1. The implementation allocate memory to hold the object
2. It initializes the object using initial values as specified in an initializer list
3. It executes the constructor body

# Calling the constructor initializer

A not so good version :

```
Segment::Segment(int x1, int y1,
                 int  x2,  int  y2,
                 int  w){
    start  = Point(x1,y1);
   end = Point(x2,y2);
   width  = w;
   }
```

A better one:

```
Segment::Segment(int x1, int y1,
                 int  x2,  int  y2,
                 int  w):start (x1,y1),
         end(x2,y2),  width(w){}
```

## When should I use constructor initializer ?

- Members object don’t have default constructor.
- Constant members.
- Reference members.

# Copy Constructor

```
Rectangle a(1,2);
Rectangle b = a.scale(2);
Rectangle c = b;
cout << b.is_higher(c)  << endl;
```

Explicit or implicit copies are controlled by the copy constructor.

## Copy constructor

- Exists to initialize a new object of the same type
- Define what a copy means (including function member)
- Does not change the initial object

```
Rectangle(const Rectangle& r);
```

# How a copy constructor works ?

## The compilator may synthesises one for us

- Each members are just copied out
- If there is object member their copy constructor is called
- Otherwise it is a simple "bit to bit" memory copy.

## Our own copy constructor

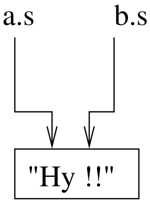
Completely useless for `rectangle` !
So let's take an other example.

# How a copy constructor works ?

```
class OurString {
    char * s;
public:
    OurString(char * s_new);
};

OurString::OurString(char * s_new)
{
        s = new char[strlen(s_new)+1];
        strcopy(s,s_new);
}
```

```
OurString a("Hy !!");
OurString b = a;
```



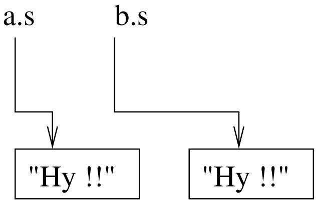
# How a copy constructor works ?

```
class OurString {
    char * s;
public:
    OurString(char * s_new);
    OurString(const OurString&);

};

OurString::OurString(const OurString& str)
{
        s = new char[strlen(str.s)+1];
        strcopy(s, str.s);
}
```

```
OurString a("Hy !!");
OurString b = a;
```



# Destructor

```
class Rectangle{
 ~Rectangle();
};
```

## Definition

- Free the allocated memory
- Only one in a class
- Can be synthesized if it doesn't exist

# rectangle.h

```cpp
#include <iostream>
class Rectangle{
public:
    Rectangle();
    Rectangle(std::istream&);
    Rectangle(const Rectangle&r);
    Rectangle(double, double);
    Rectangle scale(double);
    void set_rectangle(double,double);
    bool is_higher(const Rectangle& r) const {return _h > r._h;};
    std :: istream& read(std::istream&);
    double area() const;
    friend std :: ostream& operator<< (std::ostream& out,const Rectangle& r);
private:
    double _h;
    double _w;
};
```

# rectangle.cpp

```cpp
#include "rectangle.h"

using namespace std;

Rectangle::Rectangle():_h(0),_w(0){}

Rectangle::Rectangle(double x, double y):_h(x),_w(y){}

Rectangle::Rectangle(std::istream& in)
{
    read(in);
}

std::istream& Rectangle::read(std::istream& in){
    in >> _h >> _w;
    return in;
}

double Rectangle::area() const
{
    return _h*_w;
}
```

```cpp
Rectangle Rectangle::scale(double x)
{
    return Rectangle(x*_h,x*_w);
}

void Rectangle::set_rectangle(double x,double y)
{
    _h = x;
    _w = y;
}
std::ostream& operator<< (std::ostream& out,const Rectangle& r)
{
  return (out << "Rectangle(" << r._h << "," << r._w << ")");
}

int main()
{
    Rectangle r = cin;
    //r.read(cin);
    //Rectangle l = r.scale(2);
    cout << r.area()<< endl;
    //cout << l << endl;
    return 0;
}
```

# Synthesized Constructor

- If you don’t write any constructor ; C++ might automatically synthesize a default constructor for you
  - the default constructor is one that takes no arguments and that initializes all member variables to 0-equivalents (0, NULL, false, ..)
  - C++ does this iff your class has no const or reference data members
- If you don’t define your own copy constructor, C++ will synthesize one for you
  - it will do a shallow copy of all of the fields (i.e., member variables) of your class
  - sometimes the right thing, sometimes the wrong thing

# Overloaded functions

**Same name but different**

- Two functions/methods may have the same name
- But their signature have to be different
- The compiler resolves the choice
- If the compiler fails an error diagnostic is produced

# Overloaded functions - example

```
#include <iostream>
#include <string>
double grade(double mid, double final, double hw){
  return 0.2 * mid + 0.4 * final + 0.4 * hw;
}
double grade(double mid, double final, double hw1, double hw2){
  return 0.2 * mid + 0.4 * final + 0.4 * ((hw1+hw2)/2);
}
int main(){
    double x;
    x = grade(10,15,14);
    std::cout << x << std::endl;
    x = grade(10,15,14,20);
    std::cout << x << std::endl;
    return 0;
}
```

# Overloaded operators

The effect of an operator depends on the type of its operands.

## Example

```
#include<iostream>
#include<string>
int main()
{    // Example 1
     int a = 2;
     int b = 3;
     std::cout << a + b << std::endl;

     // Example 2
     std::string s = "Hello,";
     std::cout << s + "World !" << std::endl;

     return 0;
}
```

# Our own overloaded operators

As a member

```
class Point{
  int x;
  int y;
public:
  Point(int a,int b){x=a;y=b;};

  Point operator+(const Point& a){
    return Point(x + a.x , y + a.y);
  };
```

OR

As a non-member

```
Point operator+(const Point& b,
                const Point& a){
  return Point(b.getx() + a.getx() ,
               b.gety() + a.gety());
}
```

```
int main()
{
    Point p1(3,4);
    Point p2(7,6);
    p1 = p1 + p2;

    cout << "(" << p1.x <<"," ;
    cout << p1.y << ")"<< endl;
    return 0;
}
```

Result ?

# Write its own operators

## Generalities

An operator is used in expressions.
An expression returns a result and may have some side effects.
➢ It can be defined as a function.

## Structure

```
return_type operator@ (argument_list){
   Opertor body
}
```

## Restrictions

- The operators **::** (scope resolution), **.** (member access), **.*** (member access through pointer to member), and **?:** (ternary conditional) cannot be overloaded.
- New operators cannot be created

# Copy assignment Operator operator=

Should be a member of the class

## Assignment behavior

```
int x, y , z;
x = y = z = 15;
```

- The assignment is right-associative and returns a reference to its left-hand argument.
- All members should be copied

```
Point& operator=(const Point& p){
  copy(p);
  return *this;
}
```

# Synthesized assignment operator

- If you don't overload the assignment operator, C++ will synthesize one for you
  - it will do a shallow copy of all of the fields (i.e., member variables) of your class
  - sometimes the right thing, sometimes the wrong thing

# Friends

## No member but quite close

- A friend function, operator or class of class Point has the same access rights as a member.
- Access private and public members.
- The class chooses its friends.

```
class Point{
    double x, y;
public:
    ...
    friend std::ostream& operator<<(std::ostream& , const Point &);
};
std::ostream& operator<<(std::ostream& out, const Point& p){
        return out << "(" << p.x<<"," << p.y<<")"<<endl;}
```

# Static members

```
class Rectangle{
public:
  static  int  _nb_rectangle;
  Rectangle(){_nb_rectangle++;};
  Rectangle(double a,double b):_h(a),_w(b){_nb_rectangle++;};
private:
  double _h;
  double _w;
};
```

## How it works ?

- Share by all objects of the same type.
- There exists one and only one memory part for a given class.
- Initializing : in the global part of a program (for public and private members)
- Calling : `p.nbRectangle` or `Rectangle::nbRectangle`

# Static member functions

## Differ from ordinary functions

- Do not operate on a object of the class type.
- Associate to the class, not to a particular object.
- Access only static members.

```
class Rectangle{
 double h;
 double w;
 static Rectangle * _first_rect ;
public:
 static int nbRectangle;
 Rectangle(){nbRectangle++;};
 Rectangle(double a, double b):h(a),w(b){nbRectangle++;};
 static void show_first() { cout << "(" <<
     _first_rect ->h<<"," << _first_rect->w<<")"<<endl;}
};
```

# Static use

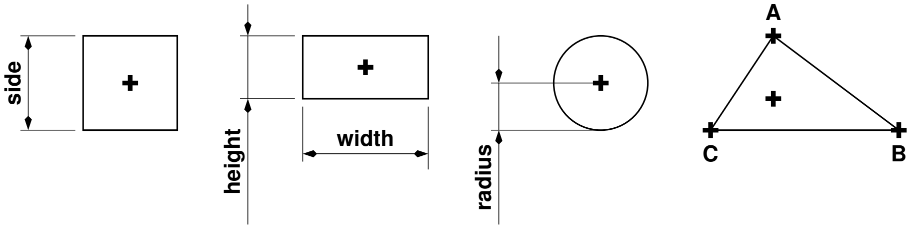
```cpp
#include <iostream>
using namespace std;
int Rectangle::nb_rectangle = 0;
Rectangle * Rectangle::first_rect = new Rectangle(2,2);

int main(){
    Rectangle p(3,4) ;
  cout << Rectangle::nb_rectangle << endl;
    Rectangle::show_first() ;
    p.show_first() ;
    return 0;
}
```

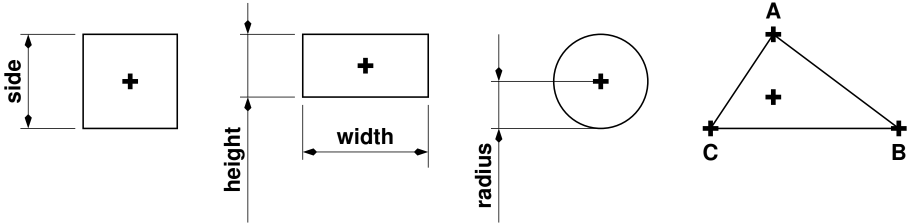
# Part III

# Inheritance

# Basic cases



# Basic cases




| Square |
|---|
| _center |
| _side |
| get_center() |
| draw() |
| erase() |
| get_sides() |

| Rectangle |
|---|
| _center |
| _width |
| _height |
| get_center() |
| draw() |
| erase() |
| get_sides() |

| Circle |
|---|
| _center |
| _radius |
| get_center() |
| draw() |
| erase() |

| Triangle |
|---|
| _center |
| _pointA |
| _pointB |
| _pointC |
| get_center() |
| draw() |
| erase() |
| get_sides() |

# Look more closer

| Square |
| --- |
| _center |
| _side |
| get_center() |
| draw() |
| erase() |
| get_sides() |

| Rectangle |
| --- |
| _center |
| _width |
| _height |
| get_center() |
| draw() |
| erase() |
| get_sides() |

| Circle |
| --- |
| _center |
| _radius |
| get_center() |
| draw() |
| erase() |

| Triangle |
| --- |
| _center |
| _pointA |
| _pointB |
| _pointC |
| get_center() |
| draw() |
| erase() |
| get_sides() |

# Look more closer

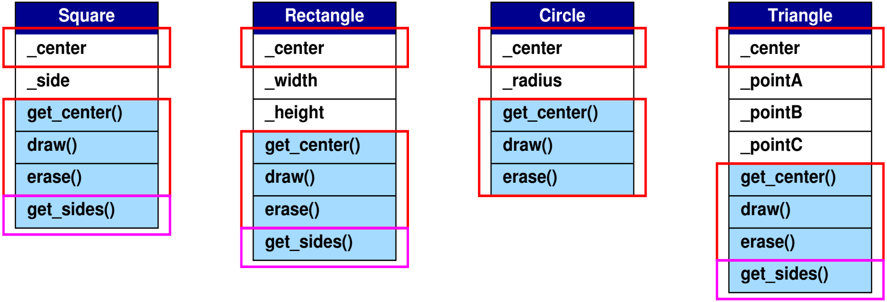
| Square |
| --- |
| _center |
| _side |
| get_center() |
| draw() |
| erase() |
| get_sides() |

| Rectangle |
| --- |
| _center |
| _width |
| _height |
| get_center() |
| draw() |
| erase() |
| get_sides() |

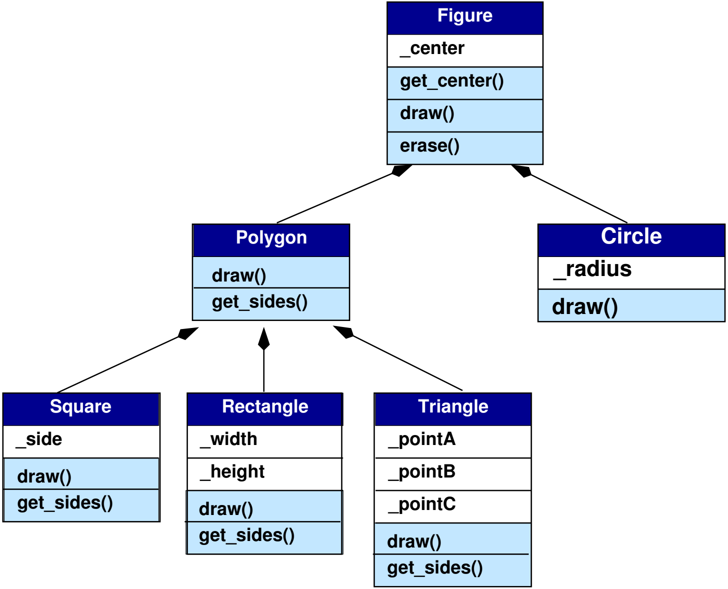
| Circle |
| --- |
| _center |
| _radius |
| get_center() |
| draw() |
| erase() |

| Triangle |
| --- |
| _center |
| _pointA |
| _pointB |
| _pointC |
| get_center() |
| draw() |
| erase() |
| get_sides() |

# Look more closer



# Class hierarchy



# Defining class hierarchy

```cpp
class Figure {
 private :
 Point _center;

 public :
 Figure(Point& center);

 Point& get_center();
 void draw() const;
 void erase();

};
```

```cpp
#include "figure.h"

class Circle : public Figure{
	private:
		double _radius;

	public:
		Circle () ;
		void draw() const;
};
```

## Inheritance limit

The constructor, the destructor, assignment operator of Figure are not members of the derived class.

# Public inheritance

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

## What the compiler will say about that ?

```
void Circle::Draw(){
    std::cout << "center : ";
    std::cout  << " center :" << _center << " radius : " << _radius << std::endl;
}
```

```
figure.h: In member function 'void Circle::Draw()':
figure.h:5: erreur: 'Point Figure::_center' is private
circle.cc:8: erreur: à l'intérieur du contexte
```

# Protection revisited

```cpp
class Figure {
 protected :
 Point _center;

 public :
 Figure(Point& center);

 Point& get_center();
 void draw();
 void erase();

};
```

**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.

# Constructors

## base-class

```
Figure::Figure(){std::cout<<"Default Figure" << std::endl;}

Figure::Figure(Point& center):_center(center){
std::cout<<"Figure with center" << std::endl;
}
```

## derived class

```
Circle::Circle():_radius(0){std::cout<<"Default Circle" << std::endl;}

Circle::Circle(Point c,double r):Figure(c),_radius(r){
std::cout<<"Circle init" << std::endl;
}
```

# Example - use

## Constructor

```
Figure f1(p);
Figure f2(p1);

Circle  c1(p,3);
Circle  g2(p,4);
```

## Function call

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

compare(f1,f2);
compare(c1,c2);
compare(c,f);
```

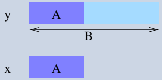
# Static cast

Type known at compile time

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

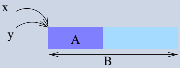
## Object

```
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 `refy` is `A`
- the dynamic type of `*ptry` and `refy` 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.

```
void draw_picture(const Figure& s1, const Figure& s2)
{
  s1.draw();
  s2.draw();
}

Figure e1,e2;
Circle  s1,s2;
draw_picture(e1,e2);
draw_picture(s1,s2);
```

How to be sure that the right method `draw()` is used ?

# Polymorphism

For references and pointers, sometimes we want to know at which class the object really belongs ?

## Definition

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

## What for ?

- Build container with heterogeneous types inside
- For the destructor
- Re-use the code for an other application

# `virtual` function

```
class Figure{
public :
  virtual  void draw() const;
  //  ...};
```

## Virtual function

We can declare function that can be redefine in derived class.
As before, so what ?

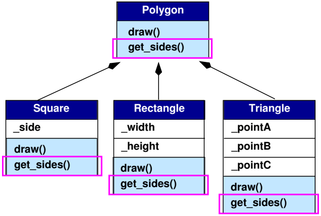
- Calling a function that depends on the actual type of an object
- Making this decision at run time

How ?

- Keyword `virtual` used only inside the class definition
- When it's inherited, no need to repeat this keyword

➢ A destructor has to be virtual

# Abstract class




## Abstract concept

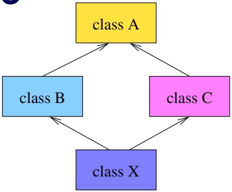
- Define as a base-class
- Can not be implemented

## Pure virtual

```
class Polygon : public Figure{
  public:
  virtual  double get_sides() = 0;
};
```

- If one pure virtual function ⇒ Abstract class
- If function not defined in the derived class ⇒ Abstract class too.

# Multiple inheritance




## Derived from many classes

```cpp
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 { /* ... */ };
```

# Part IV

# The Stream Library

# I/O stream

## Read and write

### Stream library

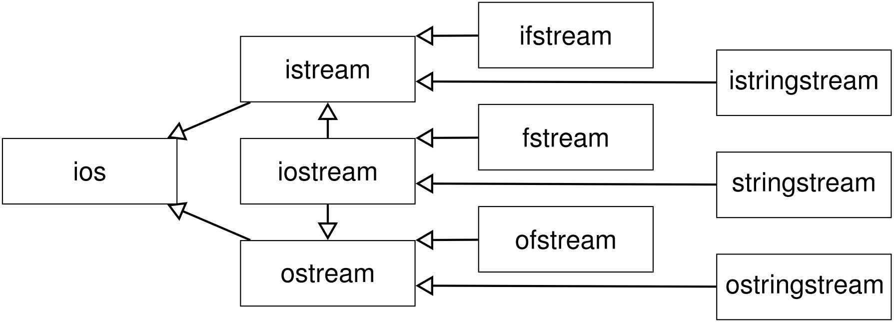
The iostream library is an object-oriented library that provides input and output functionality using streams.

- Input/output is implemented entirely in the library
- No language features supports I/O

### Stream definition

- Represent a device on which input and output operations are performed.
- Can be represented as a source or destination of characters of indefinite length
- Associated generally to a physical source or destination of characters(disk file,keyboard,console)

# Class hierarchy




## Using inheritance

- Basic functions are defined only once
- Same operators/functions used for all kind of stream
- Your own classes can be derived easily that look and behave like the standard ones.

# The class stream

## What's inside ?

- Formatting in formations (format flags, field with, precision ...)
- State information (error state flags)
- Types (flags types, stream size ...)
- Operations (!)
- Members functions (set/get flags, floating-point precision ...)

# The class stream - example

```
#include<iostream>
using namespace std;
int  main()
{
  double f = 3.14159;
  cout.precision(10);
  cout << f << endl;
  cout.setf(ios::fixed);    // floatfield set to fixed
  cout << f << endl;
  cout.flags( ios::right | ios::hex | ios::showbase );
  cout.width (10);
  cout << 100 << endl;
  cout.unsetf ( ios_base::showbase | ios::hex);
  cout.width (10);
  cout. fill ('>');
  cout << 100 << endl;
  return 0;
}
```

# The class input and output stream

**<iostream>**

### <ostream> / write

- << : insert data with format operator
- put/write : put character/write block of data
- tellp/seekp : get/set position of the put pointer

➢ cout, cerr, clog are instantiations of this class.

### <istream> / read

- >> : extract data with format operator
- get/getline : get data from stream
- tellg/seekg : get/set position of the get pointer

➢ cin is an instantiation of this class.

fstream only adds open and close file member function.

# Manipulators

Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators.

## Some examples

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main () {
  cout << showbase << hex;
  cout << uppercase << 77 << "\t" << nouppercase << 77 << endl;

  double f = 3.14159;
  cout << setprecision(5) << f << "\t" << setprecision(7) << f << endl;
  return 0;
}
```