Object Oriented Programming USTH, Master ICT, year 1

Aveneau Lilian

lilian.aveneau@usth.edu.vn
XLIM/ASALI, XLIM/SRI
CNRS, Computer Science Department
University of Poitiers

2019/2020

Lecture 3 – Classes and Objects

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Structures and member functions

Before the classes, let us see C++ structures... rarely used, they are particular case of classes (except data encapsulation)

In C:

```
struct Point {    /* defines "structure" Point                */
    int m_x;      /* defines "fields" x and y, aka members data */
    int m_y;      /*     or instance variables for classes     */
};
```

Structures and member functions

Before the classes, let us see C++ structures... rarely used, they are particular case of classes (except data encapsulation)

In C:

```
struct Point {   /* defines "structure" Point                    */
    int m_x;     /* defines "fields" x and y, aka members data */
    int m_y;     /*     or instance variables for classes       */
};
```

In C++, we can add “member” functions

```
struct Point {
    int m_x, m_y;    // classical field declaration
    // member functions declaration
    void initialize ( const int, const int );
    void move ( const int, const int );
    void print ();
};
```

Structures and member functions

Before the classes, let us see C++ structures... rarely used, they are particular case of classes (except data encapsulation)

In C:

```
struct Point {   /* defines "structure" Point                  */
    int m_x;     /* defines "fields" x and y, aka members data */
    int m_y;     /*     or instance variables for classes      */
};
```

In C++, we can add “member” functions

```
struct Point {
    int m_x, m_y;     // classical field declaration
    // member functions declaration
    void initialize ( const int, const int );
    void move ( const int, const int );
    void print ();
};
```

Remarks

- Definition of functions (implementation) is done elsewhere
- No argument for the structure itself (silent) ...

Member functions definition

Usage of "resolution" operator is mandatory

- Avoid name clashes
- Else: like classical functions
- Grant transparent access to members

```
#include <iostream>
#include "Point.h"      /// contains structure Point declaration
using namespace std;
void Point::initialize ( const int abs, const int ord ) { // "Point::" solves range problem
    m_x = abs;    m_y = ord; /// access to members "m_x" and "m_y" ...
}
void Point::move ( const int dx, const int dy ) {
    m_x += dx;    m_y += dy;
}
void Point::print () {
    cout << "Point at (" << m_x << "," << m_y << ")" << endl;
}
```

Member functions definition

Usage of "resolution" operator is mandatory

- Avoid name clashes
- Else: like classical functions
- Grant transparent access to members

```
#include <iostream>
#include "Point.h"      /// contains structure Point declaration
using namespace std;
void Point::initialize ( const int abs, const int ord ) { // "Point::" solves range problem
    m_x = abs;    m_y = ord; /// access to members "m_x" and "m_y" ...
}
void Point::move ( const int dx, const int dy ) {
    m_x += dx;    m_y += dy;
}
void Point::print () {
    cout << "Point at (" << m_x << "," << m_y << ")" << endl;
}
```

Usage: always in regard to given structure Point

```
    Point p; /// NB: ''struct'' keyword useless ...
    p.initialize ( 0, 0 ); // idem "p.m_x  = 0 ; p.m_y  = 0;"
    p.move ( 2, 1 );       // idem "p.m_x += 2 ; p.m_y += 1;"
    p.print ();            // ...
```

Member functions definition

Usage of "resolution" operator is mandatory

- Avoid name clashes
- Else: like classical functions
- Grant transparent access to members

```
#include <iostream>
#include "Point.h"      /// contains structure Point declaration
using namespace std;
void Point::initialize ( const int abs, const int ord ) { // "Point::" solves range problem
    m_x = abs;   m_y = ord; /// access to members "m_x" and "m_y" ...
}
void Point::move ( const int dx, const int dy ) {
    m_x += dx;   m_y += dy;
}
void Point::print () {
    cout << "Point at (" << m_x << "," << m_y << ")" << endl;
}
```

Usage: always in regard to given structure `Point`

```
Point p; /// NB: ''struct'' keyword useless ...
p.initialize ( 0, 0 ); // idem "p.m_x  = 0 ; p.m_y  = 0;"
p.move ( 2, 1 );       // idem "p.m_x += 2 ; p.m_y += 1;"
p.print ();            // ...
```

Member functions are always relative to a single variable: not ambiguous!

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Class is more general than structure

Differences/usage

- Use `class` instead of `struct`
- Member access control: keywords `public` and `private`
- Similar usage

Class is more general than structure

Differences/usage

- Use `class` instead of `struct`
- Member access control: keywords `public` and `private`
- Similar usage

```
#include <iostream>
using namespace std;
class Point {  /// defines class "Point"
  private:     // useless, it is the default case
    int m_x;   /// private members
    int m_y;   /// ...
  public:      /// now, public members
    void initialize ( const int, const int ); /// ...
    void move ( const int, const int );       /// methods
    void print ();                            /// ...
}; // do not forget the ";"

void Point::initialize ( const int abs, const int ord ) { m_x = abs; m_y = ord; }
void Point::move ( const int dx, const int dy ) { m_x += dx; m_y += dy; }
void Point::print () { cout << "Point("<<m_x<<","<<m_y<<")"<<endl; }
int main () { /// example of class "Point" usages
    Point a, b;
    a.initialize (5, 2); a.print (); /// display Point(5,2)
    a.move (-2, 4);      a.print (); /// display Point(3,6)
    b.initialize (1,-1); b.print (); /// display Point(1,-1)
    return 0;
}
```

Some remarks about this example

- `a` and `b` are called instances of class `Point`, or objects of type `Point`

Some remarks about this example

- `a` and `b` are called instances of class `Point`, or objects of type `Point`
- Respect pure OOP (data encapsulation)
  Write `a.m_x` ⇒ compilation error

Some remarks about this example

- a and b are called instances of class Point, or objects of type Point
- Respect pure OOP (data encapsulation)
  Write `a.m_x` ⇒ compilation error
- Possible to have private members functions

Some remarks about this example

- a and b are called instances of class Point, or objects of type Point
- Respect pure OOP (data encapsulation)
  Write a.m_x ⇒ compilation error
- Possible to have private members functions
- Keywords private (default) and public may be used as many time as you need

Some remarks about this example

- `a` and `b` are called instances of class `Point`,
  or objects of type `Point`
- Respect pure OOP (data encapsulation)
  Write `a.m_x` ⇒ compilation error
- Possible to have private members functions
- Keywords `private` (default) and `public` may be used
  as many time as you need
- Declare all to public ∼ a structure

Some remarks about this example

- `a` and `b` are called instances of class `Point`,
  or objects of type `Point`
- Respect pure OOP (data encapsulation)
  Write `a.m_x` $\Rightarrow$ compilation error
- Possible to have private members functions
- Keywords `private` (default) and `public` may be used
  as many time as you need
- Declare all to public $\sim$ a structure
- We will use "class" for all the usages ...

Some remarks about this example

- a and b are called instances of class Point, or objects of type Point
- Respect pure OOP (data encapsulation)
  Write a.m_x $\Rightarrow$ compilation error
- Possible to have private members functions
- Keywords private (default) and public may be used as many time as you need
- Declare all to public $\sim$ a structure
- We will use "class" for all the usages ...
- Third keyword exists, protected useful with inheritance

Some remarks about this example

- a and b are called instances of class `Point`,
  or objects of type `Point`
- Respect pure OOP (data encapsulation)
  Write `a.m_x` ⇒ compilation error
- Possible to have private members functions
- Keywords `private` (default) and `public` may be used
  as many time as you need
- Declare all to public ∼ a structure
- We will use "class" for all the usages ...
- Third keyword exists, `protected` useful with inheritance
- Notion of anonymous class exists (no name)

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Object assignment

With C, structured variable assignment is possible:

```
struct Point a, b; /// same previous structure
a = b; /// equivalent to: a.m_x = b.m_x; a.m_y = b.m_y;
```

Object assignment

With C, structured variable assignment is possible:

```
struct Point a, b; /// same previous structure
a = b; /// equivalent to: a.m_x = b.m_x; a.m_y = b.m_y;
```

Extension to C++, for general objects.

Corresponds to by value copy of data members, public AND private

```
class Point {
    int m_x; // private per default
  public:
    int m_y;
    ...
};
Point a;
Point b;
a.m_x = b.m_x; /// impossible since "m_x" is private
a.m_y = b.m_y; /// here it is ok
a = b;         /// corresponds to a.m_y = b.m_y AND a.m_x = b.m_x
```

- Always legal, do not violate encapsulation principle

Object assignment

With C, structured variable assignment is possible:

```
struct Point a, b; /// same previous structure
a = b; /// equivalent to: a.m_x = b.m_x; a.m_y = b.m_y;
```

Extension to C++, for general objects.

Corresponds to by value copy of data members, public AND private

```
class Point {
    int m_x; // private per default
  public:
    int m_y;
    ...
};
Point a;
Point b;
a.m_x = b.m_x; /// impossible since "m_x" is private
a.m_y = b.m_y; /// here it is ok
a = b;         /// corresponds to a.m_y = b.m_y AND a.m_x = b.m_x
```

- Always legal, do not violate encapsulation principle
- Often needs to overload operator "=" ...

Object assignment

With C, structured variable assignment is possible:

```
struct Point a, b; /// same previous structure
a = b; /// equivalent to: a.m_x = b.m_x; a.m_y = b.m_y;
```

Extension to C++, for general objects.

Corresponds to by value copy of data members, public AND private

```
class Point {
    int m_x; // private per default
  public:
    int m_y;
    ...
};
Point a;
Point b;
a.m_x = b.m_x; /// impossible since "m_x" is private
a.m_y = b.m_y; /// here it is ok
a = b;         /// corresponds to a.m_y = b.m_y AND a.m_x = b.m_x
```

- Always legal, do not violate encapsulation principle
- Often needs to overload operator "=" ...
- Java differs: copy the reference, not the members

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Introduction

Rule: only `static` objects are set to zero

- Necessary to call member function to initialize other cases
  - Imply programmer, thus high error risk
  - Quid if needed operations (dynamic allocation, Database access, ...)

Introduction

Rule: only `static` objects are set to zero

- Necessary to call member function to initialize other cases
  - Imply programmer, thus high error risk
  - Quid if needed operations (dynamic allocation, Database access, ...)

- "Automatic" solution: constructor
  - Member function, automatically called at each object creation
  - For all allocation patterns : static, automatic, dynamic
  - Todays's lecture: limited to the first two (not with `new`)

Introduction

Rule: only `static` objects are set to zero

- Necessary to call member function to initialize other cases
  - Imply programmer, thus high error risk
  - Quid if needed operations (dynamic allocation, Database access, ...)
- "Automatic" solution: constructor
  - Member function, automatically called at each object creation
  - For all allocation patterns : static, automatic, dynamic
  - Todays's lecture: limited to the first two (not with `new`)
- Object also may have destructor
  - Method automatically called at object releasing
  - Automatic class: at block exit or method exit

Introduction

Rule: only `static` objects are set to zero

- Necessary to call member function to initialize other cases
  - Imply programmer, thus high error risk
  - Quid if needed operations (dynamic allocation, Database access, ...)

- "Automatic" solution: constructor
  - Member function, automatically called at each object creation
  - For all allocation patterns : static, automatic, dynamic
  - Todays's lecture: limited to the first two (not with `new`)

- Object also may have destructor
  - Method automatically called at object releasing
  - Automatic class: at block exit or method exit

- Naming convention
  - Constructor: class name
  - Destructor: idem, preceded by tilde ($\sim$)

Example with one constructor

"`Point`" class definition: exit `initialize()` !

```
class Point {
    int m_x, m_y; /// private members
  public:
    Point( const int, const int ); // This is a constructor
    void move ( const int, const int );
    void print ();
};
```

Example with one constructor

"Point" class definition: exit `initialize()` !

```
class Point {
    int m_x, m_y; /// private members
  public:
    Point( const int, const int ); // This is a constructor
    void move ( const int, const int );
    void print ();
};
```

Usage

Classical version: `Point a;` BAD

Safeguard: required to use one existing constructor ...

Example with one constructor

"Point" class definition: exit initialize() !

```
class Point {
    int m_x, m_y; /// private members
  public:
    Point( const int, const int ); // This is a constructor
    void move ( const int, const int );
    void print ();
};
```

Usage

Classical version: `Point a;` BAD

Safeguard: required to use one existing constructor ...

```
#include <iostream>
using namespace std;
class Point {
  ... /// idem before
};
// constructor: notice the syntax
Point::Point(const int abs, const int ord) {
    m_x = abs;     m_y = ord;
}
void Point::move(const int dx,const int dy){
    m_x += dx;     m_y += dy;
}
```

```
void Point::print () {
  cout<< "Point("<<m_x <<","<<m_y <<")"<<endl;
}
int main () {
    Point a(5,2); // call "Point(const int, co
    a.print ();
    a.move (-2, 4);
    a.print ();
    Point b(1,-1); // call "Point(const int,co
    b.print ();
    return 0;
}
```

Remarks and second example

1. Without argument constructor $\Rightarrow$ "`Point a;`"
   but not "`Point a();`"
2. Possible to overload constructor!
3. Default constructor exists, and is without argument

Remarks and second example

1. Without argument constructor ⇒ "Point a;"
   but not "Point a();"
2. Possible to overload constructor!
3. Default constructor exists, and is without argument

Example with destructor

```
#include <iostream>
using namespace std;
class Test {
 public:
  int m_num;
  Test (const int); /// constructor
  ~Test ();         /// destructor
};
Test::Test (const int n) { /// constructor
  m_num = n;
  cout<<"++Constructor call - num="
      <<m_num<<endl;
}
Test::~Test() { /// destructor
  cout<<"--Destructor call - num="
      <<m_num<<endl;
}
```

```
void fct (int p) {
  Test x(2*p);
}
int main () {
  Test a(1);
  for (int i=1; i<=2; ++i)
    fct(i);
  return 0;
}
```

Remarks and second example

1. Without argument constructor ⇒ "Point a;" but not "Point a();"
2. Possible to overload constructor!
3. Default constructor exists, and is without argument

Example with destructor

```
#include <iostream>
using namespace std;
class Test {
 public:
  int m_num;
  Test (const int); /// constructor
  ~Test ();         /// destructor
};
Test::Test (const int n) { /// constructor
  m_num = n;
  cout<<"++Constructor call - num="
      <<m_num<<endl;
}
Test::~Test() { /// destructor
  cout<<"--Destructor call - num="
      <<m_num<<endl;
}
```

```
void fct (int p) {
  Test x(2*p);
}
int main () {
  Test a(1);
  for (int i=1; i<=2; ++i)
    fct(i);
  return 0;
}
```

```
// ++Constructor call - num=1
// ++constructeur call -num=2
// --Destructor call -num=2
// ++Constructor call -num=4
// --Destructor call -num=4
// --Destructor call -num=1
```

Role of constructors and destructors

```
#include <iostream>
#include <cstdlib>   // defines ''rand()''
using namespace std;
class Chance {
  int  m_nb;
  int* m_val;
public:
  Chance (const int nb=10, const int max=100); /// value number, and max value
  ~Chance();
  void print();
};
Chance::Chance(const int nb, const int max) {
  m_nb = nb;            /// now, we will prefix members with "m_", to avoid
  m_val = new int[nb]; /// conflicts, and to read them easily
  for (int i=0; i<nb; ++i)
    m_val[i] = max*double(rand())/RAND_MAX;
}
Chance::~Chance () { /// mandatory, else how to clean
  delete m_val;       /// correctly allocated memory in heap?
}
void Chance::print () {
  for (int i=0; i<_nb; ++i)  cout<<m_val[i]<<" ";
  cout<<endl;
}
int main() {
  Chance series1 (10, 5);
  series1.print();     // 0 0 3 2 2 1 0 3 3 4
  Chance series2 (6);
  series2.print();     // 38 51 83 3 5 52
  series2 = series1;   /// What??? Error, double "delete" leading to: "ABORT TRAP" !!
  return 0;            /// Hence, generally needs to overload the operator ''=''
}
```

Some rules

- Constructor: may have any number of arguments, even zero; do not return a value
- Destructor: no argument, do not return a value

Some rules

- Constructor: may have any number of arguments, even zero; do not return a value
- Destructor: no argument, do not return a value
- May be public or private (generally public)
  - Destructor, not important: no more be callable ...
  - Constructor become unusable, except by object's methods ...

Some rules

- Constructor: may have any number of arguments, even zero; do not return a value
- Destructor: no argument, do not return a value
- May be public or private (generally public)
  - Destructor, not important: no more be callable ...
  - Constructor become unusable, except by object's methods ...
- Case needing private constructor:
  1. Abstract class (for inheritance)

Some rules

- Constructor: may have any number of arguments, even zero; do not return a value
- Destructor: no argument, do not return a value
- May be public or private (generally public)
  - Destructor, not important: no more be callable ...
  - Constructor become unusable, except by object's methods ...
- Case needing private constructor:
  1. Abstract class (for inheritance)
  2. Class have others constructors with at least a public one

Some rules

- Constructor: may have any number of arguments, even zero; do not return a value
- Destructor: no argument, do not return a value
- May be public or private (generally public)
  - Destructor, not important: no more be callable ...
  - Constructor become unusable, except by object's methods ...
- Case needing private constructor:
  1. Abstract class (for inheritance)
  2. Class have others constructors with at least a public one
  3. *Design patterns singleton & factory*: only 1 possible instance; a *static* method responsible for return it, + create it at 1st call, via private constructor

Some rules

- Constructor: may have any number of arguments, even zero; do not return a value
- Destructor: no argument, do not return a value
- May be public or private (generally public)
  - Destructor, not important: no more be callable ...
  - Constructor become unusable, except by object's methods ...
- Case needing private constructor:
  1. Abstract class (for inheritance)
  2. Class have others constructors with at least a public one
  3. *Design patterns singleton & factory*: only 1 possible instance; a *static* method responsible for return it, + create it at 1st call, via private constructor
- Comparison to Java: default and explicit initializations
  - Do not exist in C++
  - Thus almost always necessary to define some constructors

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Qualifier static for data members

Instances of same class have their own data members:

```
class Example1 {
  int m_n;
  float m_x;
  ...
};
Example1 a, b;
```

Here: a and b do not share their memory

Qualifier static for data members

Instances of same class have their own data members:

```
class Example1 {
  int m_n;
  float m_x;
  ...
};
Example1 a, b;
```

Here: a and b do not share their memory

- m_n and m_x are called instance variables

Qualifier static for data members

Instances of same class have their own data members:

```
class Example1 {
  int m_n;
  float m_x;
  ...
};
Example1 a, b;
```

Here: a and b do not share their memory

- m_n and m_x are called instance variables

Definition of class member

Member having infinite lifetime

```
class Example2 {
  static int m_n; // m_n is a class member, not an instance one
  float m_x;
  ...
};
Example2 a, b;
```

Now, a and b have their “own” m_x, but they share m_n

Qualifier `static` for data members

Instances of same class have their own data members:

```
class Example1 {
  int m_n;
  float m_x;
  ...
};
Example1 a, b;
```

Here: `a` and `b` do not share their memory

- `m_n` and `m_x` are called instance variables

Definition of class member

Member having infinite lifetime

```
class Example2 {
  static int m_n; // m_n is a class member, not an instance one
  float m_x;
  ...
};
Example2 a, b;
```

Now, `a` and `b` have their “own” `m_x`, but they share `m_n`

So `static` have one more meaning: “common to all instances”

Example

```cpp
/// Singleton \& Factory example
#include <iostream>
#include <cstdlib> // rand()
using namespace std;
class Chance {
  int  m_nb;
  int* m_val;
  // instance that realizes SINGLETON
  static Chance *m_instance;
  // constructor and destructor are private
  Chance (const int=10, const int=100);
  ~Chance();
  Chance operator=(Chance&); // hidden
public:
  void print();
  static Chance &instance(); // FACTORY
};
/// constructor ...
Chance::Chance(const int nb, const int max){
  m_nb = nb;
  m_val = new int[nb];
  for (int i=0; i<nb; ++i)
    m_val[i] = double(rand())/RAND_MAX*max;
}
/// Destructor is never called! (private)
Chance::~Chance () {
  cout<<"singleton destruction"<<endl;
  delete m_val;/// for the beauty ...
}
```

```cpp
void Chance::print () {
  for (int i=0; i<m_nb; ++i)
    cout<<m_val[i]<<" ";
  cout<<endl;
}
/// Book memory for class data
/// ... but only once
Chance *Chance::m_instance = NULL;
Chance& Chance::instance() {
  if (m_instance == NULL)
    m_instance = new Chance();
  return *m_instance;
}
void f() {
  Chance &suite = Chance::instance();
  suite.print();
}
int main() {
  Chance &suite1 = Chance::instance();
  suite1.print();
  for (int i=0; i<2; i++) f();
  return 0;
}

/// output:
///0 13 75 45 53 21 4 67 67 93
///0 13 75 45 53 21 4 67 67 93
///0 13 75 45 53 21 4 67 67 93
```

Example

```cpp
/// Singleton \& Factory example
#include <iostream>
#include <cstdlib> // rand()
using namespace std;
class Chance {
  int  m_nb;
  int* m_val;
  // instance that realizes SINGLETON
  static Chance *m_instance;
  // constructor and destructor are private
  Chance (const int=10, const int=100);
  ~Chance();
  Chance operator=(Chance&); // hidden
public:
  void print();
  static Chance &instance(); // FACTORY
};
/// constructor ...
Chance::Chance(const int nb, const int max){
  m_nb = nb;
  m_val = new int[nb];
  for (int i=0; i<nb; ++i)
    m_val[i] = double(rand())/RAND_MAX*max;
}
/// Destructor is never called! (private)
Chance::~Chance () {
  cout<<"singleton destruction"<<endl;
  delete m_val;/// for the beauty ...
}
```

```cpp
void Chance::print () {
  for (int i=0; i<m_nb; ++i)
    cout<<m_val[i]<<" ";
  cout<<endl;
}
/// Book memory for class data
/// ... but only once
Chance *Chance::m_instance = NULL;
Chance& Chance::instance() {
  if (m_instance == NULL)
    m_instance = new Chance();
  return *m_instance;
}
void f() {
  Chance &suite = Chance::instance();
  suite.print();
}
int main() {
  Chance &suite1 = Chance::instance();
  suite1.print();
  for (int i=0; i<2; i++) f();
  return 0;
}

/// output:
///0 13 75 45 53 21 4 67 67 93
///0 13 75 45 53 21 4 67 67 93
///0 13 75 45 53 21 4 67 67 93
```

Warning, difference with Java (that allows inside block initialization)

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- **Overloading methods and default arguments**
- Inline methods
- Objects and methods
- Static and constant methods

Overloading:

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

class Point {
  int m_x, m_y;
public:
  Point();                          //    I
  Point(const int);                 //   II
  Point(const int, const int);      //  III
  void print();                     //    I
  void print( const char* );        //   II
};

Point::Point() { /// I: set to zero
  m_x = 0; m_y = 0;
}
Point::Point( const int a ) { /// II
  m_x = m_y = a;   /// idem III(a,a)
}
```

```cpp
Point::Point(const int a,const int o){// III
  m_x = a; m_y = o;
}
void Point::print() {/// classical
  cout<<"Point("<<m_x<<","<<m_y<<")"<<endl;
}
void Point::print( const char* msg ) {
  cout << msg ; /// display first  message
  print ();     /// then classical version
}
int main () {
  Point a;             // cons I
  a.print();           // I
  Point b(5);          // cons II
  b.print("b - ");     // II
  Point c(3,12);       // cons III
  c.print("c - ");     // II
  return 0;            // destructor ;-)
}
```

Overloading:

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

class Point {
  int m_x, m_y;
public:
  Point();                          //   I
  Point(const int);                 //  II
  Point(const int, const int);      // III
  void print();                     //   I
  void print( const char* );        //  II
};

Point::Point() { /// I: set to zero
  m_x = 0; m_y = 0;
}
Point::Point( const int a ) { /// II
  m_x = m_y = a;    /// idem III(a,a)
}
```

```
Point::Point(const int a,const int o){// III
  m_x = a; m_y = o;
}
void Point::print() {/// classical
  cout<<"Point("<<m_x<<","<<m_y<<")"<<endl;
}
void Point::print( const char* msg ) {
  cout << msg ; /// display first message
  print ();     /// then classical version
}
int main () {
  Point a;            // cons I
  a.print();          // I
  Point b(5);         // cons II
  b.print("b - ");    // II
  Point c(3,12);      // cons III
  c.print("c - ");    // II
  return 0;           // destructor ;-)
}
```

Private or public status of method

```
class A {
private: void f(char c) { c=0; };
public:  void f(int i) { i=0; };
};
int main () {
  A a; char c;
  a.f(c); /// ???
  return 0;
}
```

- Compilation produces message:
  ```
  //member.cpp: In function 'int main()':
  //member.cpp:2: error: 'void A::f(char)' is private
  //member.cpp:9: error: within this context
  ```
- Problem: identifier resolution

⇒Search without "private" or "public" status

Default arguments

Apply to methods, when possible

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

class Point {
  int m_x, m_y;
public:
  Point();
  Point( const int );
  Point( const int, const int );
  void print( const char*msg="" );
};

Point::Point() { /// I
  m_x = 0; m_y = 0;
}
Point::Point( const int a ) { /// II
  m_x = m_y = a;
}
```

```
Point::Point(const int a,const int o){// III
  m_x = a; m_y = o;
}
void Point::print( const char* msg ) {
  cout << msg ;
  cout<<"Point("<<m_x<<","<<m_y<<")"<<endl;
}
int main () {
  Point a;              // cons I
  a.print();            // I
  Point b(5);           // cons II
  b.print("b - ");      // I
  Point c(3,12);        // cons III
  c.print("c - ");      // I
  return 0;             // destructor ;-)
}
```

Default arguments

Apply to methods, when possible

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

class Point {
  int m_x, m_y;
public:
  Point();
  Point( const int );
  Point( const int, const int );
  void print( const char*msg="" );
};

Point::Point() { /// I
  m_x = 0; m_y = 0;
}
Point::Point( const int a ) { /// II
  m_x = m_y = a;
}
```

```
Point::Point(const int a,const int o){// III
  m_x = a; m_y = o;
}
void Point::print( const char* msg ) {
  cout << msg ;
  cout<<"Point("<<m_x<<","<<m_y<<")"<<endl;
}
int main () {
  Point a;            // cons I
  a.print();          // I
  Point b(5);         // cons II
  b.print("b␣-␣");    // I
  Point c(3,12);      // cons III
  c.print("c␣-␣");    // I
  return 0;           // destructor ;-)
}
```

Here no overloading for our `Point` class constructors

- Only because of our 1-argument version ...
- Other version: `Point::Point(int a=0, int b=0) { m_x=a; m_y=b; }`

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

How it works

- Either by supplying definition into declaration

```cpp
#ifndef _POINT_H
#define _POINT_H
class Point
{
    int m_x, m_y;
  public:
    /// constructor I
    Point () { m_x=0, m_y=0; }
    /// constructor II
    Point (const int a) { m_x = m_y = a; }
    /// Constructor III
    Point (const int a, const int o) {
      m_x=a; m_y=o;
    }
    // print is not inline!
    void print (const char*msg="");
}
#endif
/// NB :
///    in this case, not necessary
///    to use "inline" keyword!
```

- Or like ordinary functions, with `inline`

```cpp
#ifndef _POINT_H
#define _POINT_H
#include <iostream>
class Point {
    int m_x, m_y;
  public:
    /// 2nd variant, only one constructor
    inline Point (const int=0, const int=0);
    inline void print (const char*="");
}
inline
Point::Point (const int abs, const int ord){
  m_x=abs; m_y=ord;
}
inline
void Point::print (const char*msg){
  std::cout << msg
            << " Point(" << m_x << ","
            << m_y << ")" << std::endl;
}
#endif
```

How it works

- Either by supplying definition into declaration

```
#ifndef _POINT_H
#define _POINT_H
class Point
{
    int m_x, m_y;
  public:
    /// constructor I
    Point () { m_x=0, m_y=0; }
    /// constructor II
    Point (const int a) { m_x = m_y = a; }
    /// Constructor III
    Point (const int a, const int o) {
      m_x=a; m_y=o;
    }
    // print is not inline!
    void print (const char*msg="");
}
#endif
/// NB :
///   in this case, not necessary
///   to use "inline" keyword!
```

- Or like ordinary functions, with `inline`

```
#ifndef _POINT_H
#define _POINT_H
#include <iostream>
class Point {
    int m_x, m_y;
  public:
    /// 2nd variant, only one constructor
    inline Point (const int=0, const int=0);
    inline void print (const char*="");
}
inline
Point::Point (const int abs, const int ord){
  m_x=abs; m_y=ord;
}
inline
void Point::print (const char*msg){
  std::cout << msg
            << " Point(" << m_x << ","
            << m_y << ")" << std::endl;
}
#endif
```

NB: definition are located (correctly) into a header file, like it always should be the case

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Transmitting an object as argument

By default, method receives “pointer to current instance” plus some arguments. These last may be objects of same type:

```cpp
#include <iostream>
using namespace std;
class Point {
  int m_x, m_y;
public:
  Point(const int a=0, const int o=0)
  { m_x=a; m_y=o; }
  bool is_equal ( Point p ) {
    return p.m_x == m_x && p.m_y == m_y;
  }
};
```

```cpp
int main ()
{
  Point a;
  Point b(1);
  Point c(1,0);
  cout<<"a==b ? "<<a.is_equal(b) << endl;
  cout<<"b==c ? "<<b.is_equal(c) << endl;
  cout<<"c==a ? "<<c.is_equal(a) << endl;
  return 0;
}
```

In C++ the encapsulation unit is the class and not the instance

Transmitting an object as argument

By default, method receives "pointer to current instance" plus some arguments. These last may be objects of same type:

```
#include <iostream>
using namespace std;
class Point {
  int m_x, m_y;
public:
  Point(const int a=0, const int o=0)
  { m_x=a; m_y=o; }
  bool is_equal ( Point p ) {
    return p.m_x == m_x && p.m_y == m_y;
  }
};
```

```
int main ()
{
  Point a;
  Point b(1);
  Point c(1,0);
  cout<<"a==b ? "<<a.is_equal(b) << endl;
  cout<<"b==c ? "<<b.is_equal(c) << endl;
  cout<<"c==a ? "<<c.is_equal(a) << endl;
  return 0;
}
```

In C++ the encapsulation unit is the class and not the instance

Transmission mode

- By address, if possible constant

```
bool is_equal (const Point*const p) { return m_x==p->m_x && m_y==p->m_y; }
```

Transmitting an object as argument

By default, method receives "pointer to current instance" plus some arguments. These last may be objects of same type:

```
#include <iostream>
using namespace std;
class Point {
  int m_x, m_y;
public:
  Point(const int a=0, const int o=0)
  { m_x=a; m_y=o; }
  bool is_equal ( Point p ) {
    return p.m_x == m_x && p.m_y == m_y;
  }
};
```

```
int main ()
{
  Point a;
  Point b(1);
  Point c(1,0);
  cout<<"a==b ? "<<a.is_equal(b) << endl;
  cout<<"b==c ? "<<b.is_equal(c) << endl;
  cout<<"c==a ? "<<c.is_equal(a) << endl;
  return 0;
}
```

In C++ the encapsulation unit is the class and not the instance

Transmission mode

- By address, if possible constant

```
bool is_equal (const Point*const p) { return m_x==p->m_x && m_y==p->m_y; }
```

- By reference if possible constant

```
bool is_equal (const Point& p) { return m_x==p.m_x && m_y==p.m_y; }
```

Returning an object from method

Same principle as for arguments, with transmission:

- By value, with simple copy (take care to pointers!)
- By address or reference: take care to not return local object

Returning an object from method

Same principle as for arguments, with transmission:

- By value, with simple copy (take care to pointers!)
- By address or reference: take care to not return local object

The returned object can be:

- Of same type, in which case method has access to private members
- Or not, in which case method has not access to private members

Examples

```cpp
Point Point::symmetrical() {
  Point res;
  res.m_x = -m_x; res.m_y = -m_y;
  return res;
} // CORRECT
```

```cpp
Point& Point::symmetrical() {
  Point res;
  res.m_x = -m_x; res.m_y = -m_y;
  return res;
} // INCORRECT!!
```

Returning an object from method

Same principle as for arguments, with transmission:

- By value, with simple copy (take care to pointers!)
- By address or reference: take care to not return local object

The returned object can be:

- Of same type, in which case method has access to private members
- Or not, in which case method has not access to private members

Examples

```
Point Point::symmetrical() {
  Point res;
  res.m_x = -m_x; res.m_y = -m_y;
  return res;
} // CORRECT
```

```
Point& Point::symmetrical() {
  Point res;
  res.m_x = -m_x; res.m_y = -m_y;
  return res;
} // INCORRECT!!
```

Autoreference: keyword `this`

Inside all methods

- By definition: pointer to object's instance that calls the method
- Vital for some classes ... or friend functions

Overview

- Structures in C++
- Notion of class
- Objects assignment
- Constructors & destructors
- Static data members
- Overloading methods and default arguments
- Inline methods
- Objects and methods
- Static and constant methods

Static methods

Already seen with class "Chance"

```
#include <iostream>
using namespace std;
class Test
{ /// realize counter of created objects
  static int m_cpt;
 public:
  Test()  { m_cpt++; };
  ~Test() { m_cpt--; };
  static int counter() {
    return m_cpt;
  }
};
int Test::m_cpt = 0; // ''allocation''
```

```
void f() {
  Test u, v;                           // +2
  cout<<Test::counter()<<endl;
}                                      // -2
int main () {
  cout<<Test::counter()<<endl;  // => 0
  f();                          // => 2
  cout<<Test::counter()<<endl;  // => 0
  Test t;                       // +1
  f();                          // => 3
  cout<<Test::counter()<<endl;  // => 1
  return 0;                     // -1
}
```

Main usage: Factory (e.g. class Complex)

```
class Complex {
  double m_r, m_i; /// Cartesian internal representation
  Complex (const double r, const double i); /// Cartesian specific
 public:
  ~Complex(); // must be public (!= Singleton)
  // 2 Factories
  static Complex fromCartesian (const double r, const double i);
  static Complex fromPolar (const double m, const double a );
};
```

Constant methods

Case of constant object

- Public instance's variables: not modifiable, of course
- Can methods modify instance variable?
- How to authorize it or forbid it?

Constant methods

Case of constant object

- Public instance's variables: not modifiable, of course
- Can methods modify instance variable?
- How to authorize it or forbid it?

Explicitly declare usable methods over constant instances

```cpp
class Point {
  int m_x, m_y;
 public:
  Point (const int a=0,const int b=0) {
    m_x=a; m_y=b;
  }
  void print() const;
  void move (const int a, const int b) {
    m_x+=a; m_y+=b;
  }
};
void Point::print() const { /// repeat keyword!
  ++m_x; /// compilation error!
}
```

```cpp
int main ()
{
  Point a;
  const Point b;
  a.print();     // OK
  a.move(2,2);   // OK
  b.print();     // OK
  b.move(2,2);   // compil. error
  return 0;
}
```

Works with function overloading

The mutable members

Constant method cannot modify non static instance variable ...

But normalization allows it, via the qualifier `mutable`

```
class Thing {
 public:
  int m_p;
  mutable int m_n;
  void f1() { m_p=5; ++m_n; }
  void f2() const {
    m_p=5; // compilation error
    ++m_n; // OK!
  }
};
...
const Thing t; // f1 unusable, f2 ok
t.m_n = 5;     // OK!
t.m_p = 4;     // compilation error
...
```