# 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 5 – Inheritance

- Simple inheritance
- Inheritance and access control
- Canonical form of derived class
- Multiple inheritance

# Overview

- Simple inheritance
- Inheritance and access control
- Canonical form of derived class
- Multiple inheritance

# Notion of inheritance

## First example: a *colored* point, by inheritance from Point

```cpp
class Point { /// Class "mother"
  int m_x, m_y;
 public:
  void initialize( int, int );
  void move( int, int );
  void print();
};
```

```cpp
class ColoredPoint : public Point { /// Derive from
  short m_color;
 public:
  void ColoredPoint( short color ) {
    m_color = color;
  }
};
```

# Notion of inheritance

First example: a *colored* point, by inheritance from Point

```
class Point { /// Class "mother"
  int m_x, m_y;
 public:
  void initialize( int, int );
  void move( int, int );
  void print();
};
```

```
class ColoredPoint : public Point { /// Derive from
  short m_color;
 public:
  void ColoredPoint( short color ) {
    m_color = color;
  }
};
```

## Interest of this inheritance

- Mainly: code *reusing* from Point ...
- But also: respect the principle of encapsulation
- Inherited classes are always reachable (public members)

# Notion of inheritance

First example: a *colored* point, by inheritance from Point

```
class Point { /// Class "mother"
  int m_x, m_y;
 public:
  void initialize( int, int );
  void move( int, int );
  void print();
};
```

```
class ColoredPoint : public Point { /// Derive from
  short m_color;
 public:
  void ColoredPoint( short color ) {
    m_color = color;
  }
};
```

## Interest of this inheritance

- Mainly: code *reusing* from Point ...
- But also: respect the principle of encapsulation
- Inherited classes are always reachable (public members)

Example of use:

```
ColoredPoint p;          /// Call to by default constructor
p.initialize( 10, 20);   // Member function inherited from class Point
p.color( 5 );            /// Member function of ColoredPoint
p.print();               // Inherited member function
p.move( 2, 4 );          /// idem
p.print();               /// idem
```

# Access to ancestor members into inherited class

## Respecting OOP basic principle

Derived class cannot access to private members of its ancestors classes

So, cannot write the following member function:

```
void printc() {
  cout<<"ColoredPoint at (" << m_x << "," << m_y << ")" << endl;
  cout<<"\tof color " << m_color << endl;
}
```

# Access to ancestor members into inherited class

## Respecting OOP basic principle

Derived class cannot access to private members of its ancestors classes

So, cannot write the following member function:

```
void printc() {
  cout<<"ColoredPoint at (" << m_x << "," << m_y << ")" << endl;
  cout<<"\tof color " << m_color << endl;
}
```

But, we can write this:

```
void printc() {
  print(); // call to inherited member function, idem : (*this).print()
  cout<<"\tof color " << m_color << endl;
}
```

Likewise, a dedicated initialization function will be:

```
void initializec( int x, int y, short color ) {
  initialize( x, y ); // idem : this->initialize( x, y )
  m_color = color;
}
```

# Access to ancestor members into inherited class

## Respecting OOP basic principle

Derived class cannot access to private members of its ancestors classes

So, cannot write the following member function:

```
void printc() {
  cout<<"ColoredPoint at (" << m_x << "," << m_y << ")" << endl;
  cout<<"\tof color " << m_color << endl;
}
```

But, we can write this:

```
void printc() {
  print(); // call to inherited member function, idem : (*this).print()
  cout<<"\tof color " << m_color << endl;
}
```

Likewise, a dedicated initialization function will be:

```
void initializec( int x, int y, short color ) {
  initialize( x, y ); // idem : this->initialize( x, y )
  m_color = color;
}
```

## Accessible members by inheritance

Derived class has access to PUBLIC members of its ancestor classes

# Redefining derived members

- Possible, but hide ancestor class ones
- Range resolution to access them, inside and outside derived class
- Works both for both member function and member data

# Redefining derived members

- Possible, but hide ancestor class ones
- Range resolution to access them, inside and outside derived class
- Works both for both member function and member data

```cpp
#include <iostream>
#include "point.h"

class ColoredPoint : public Point {
  short m_color;
public:
  void ColoredPoint( short color ) { m_color = color; }
  void print() ; // redefinition, hide inherited eponymous function
  void initialize( int, int, short ); // idem
};
void ColoredPoint::print() {
  Point::print();  // resolution, member function of class Point
  std::cout<<"\tand my color is "<<m_color<<std::endl;
}
void ColoredPoint::initialize( int x, int y, short couleur ) {
  Point::initialize( x, y ); // same thing
  m_color = color;
}

int main () {
  ColoredPoint p;
  p.initialize( 10, 20, 5 ); p.print(); // 2 functions from ColoredPoint
  p.Point::print();                    // Inherited function from Point
  p.move( 2, 4 );             p.print(); // Functions from ColoredPoint
  p.color( 2 );               p.print(); // Idem
  return 0;
}
```

# Redefinition and overloading

## Overloading: limits search to unique range

- Overloading completely hide inherited eponymous functions
- Possible to associate them to searching mechanism, with instruction `using A::f;` into class inheriting from A ...

# Redefinition and overloading

## Overloading: limits search to unique range

- Overloading completely hide inherited eponymous functions
- Possible to associate them to searching mechanism, with instruction `using A::f;` into class inheriting from A ...

## First example

```
class A {
 public:
  void f(int n)  { .... }
  void f(char c) { .... }
};

class B : public A {
 public:
  void f(float x) { .... }
};

int main () {
  int n; char c; A a; B b;
  a.f(n); // call A::f(int)
  a.f(c); // call A::f(char)
  b.f(n); // call B::f(float);
  b.f(c); // call B::f(float);
  return 0;
}
```

# Redefinition and overloading

## Overloading: limits search to unique range

- Overloading completely hide inherited eponymous functions
- Possible to associate them to searching mechanism, with instruction `using A::f;` into class inheriting from A ...

### First example

```
class A {
 public:
  void f(int n)  { .... }
  void f(char c) { .... }
};

class B : public A {
 public:
  void f(float x) { .... }
};

int main () {
  int n; char c; A a; B b;
  a.f(n); // call A::f(int)
  a.f(c); // call A::f(char)
  b.f(n); // call B::f(float);
  b.f(c); // call B::f(float);
  return 0;
}
```

### Example 2

```
1  class A {
2   public:
3    void f(int n)  { .... }
4    void f(char c) { .... }
5    void g(int n)  { .... }
6  };
7  class B : public A {
8   public:
9    void f(int n)  { .... }
10   void g(int a, int b) { .... }
11
12 };
13 int main () {
14   int n; char c; B b;
15   b.f(n);    // call B::f(int);
16   b.f(c);    // call B::f(int);
17   b.g(n);    // compilation error ...
18   return 0; // solution: 11— "using A::g;"
19 }
```

# Constructors

## Hierarchy of calls

Hierarchical calls:

```
class A {
  public:
    A(...);
    ~A();
    ...
};
```

```
class B : public A {
  public:
    B(...);
    ~B();
    ...
};
```

- Writing `B b(...);` $\Longrightarrow$ call constructor of A and then of B

# Constructors

## Hierarchy of calls

Hierarchical calls:

```
class A {
  public:
    A(...);
    ~A();
    ...
};
```

```
class B : public A {
  public:
    B(...);
    ~B();
    ...
};
```

- Writing `B b(...);` $\Longrightarrow$ call constructor of A and then of B
- Inverse order for destructors: ∼B() and then ∼A()

# Constructors

## Hierarchy of calls

Hierarchical calls:

```
class A {
  public:
    A(...);
    ~A();
    ...
};
```

```
class B : public A {
  public:
    B(...);
    ~B();
    ...
};
```

- Writing `B b(...);` $\Longrightarrow$ call constructor of A and then of B
- Inverse order for destructors: ∼B() and then ∼A()

## Transmitting information between constructors

- Mechanism allowing that: initialization list

```
class Point {
  int m_x, m_y;
 public:
  Point(int x=0, int y=0) {
    m_x=x; m_y=y;
  }
  ~Point() {}
};
```

```
class ColoredPoint : public Point {
  short m_color;
 public:
  ColoredPoint(int x, int y, short color)
    : Point( x, y ) {
    m_color = color;
  } // call the "good" constructor!
};
```

# Constructors

## Hierarchy of calls

Hierarchical calls:

```
class A {
  public:
    A(...);
    ~A();
    ...
};
```

```
class B : public A {
  public:
    B(...);
    ~B();
    ...
};
```

- Writing `B b(...);` $\Longrightarrow$ call constructor of A and then of B
- Inverse order for destructors: ∼B() and then ∼A()

## Transmitting information between constructors

- Mechanism allowing that: initialization list

```
class Point {
  int m_x, m_y;
 public:
  Point(int x=0, int y=0) {
    m_x=x; m_y=y;
  }
  ~Point() {}
};
```

```
class ColoredPoint : public Point {
  short m_color;
 public:
  ColoredPoint(int x, int y, short color)
     : Point( x, y ) {
    m_color = color;
  } // call the "good" constructor!
};
```

- Take care about derived class that do not have constructor

  $\Longrightarrow$ How to send initialization values to ancestors?

# Overview

- Simple inheritance
- Inheritance and access control
- Canonical form of derived class
- Multiple inheritance

# Access control: protected members

New status, adding to `public` and `private`: `protected`

- Inaccessible for class users
- Accessible to heirs (or successors)

# Access control: protected members

New status, adding to `public` and `private`: `protected`

- Inaccessible for class users
- Accessible to heirs (or successors)

## Example

```
class Point {
 protected:
  int m_x, m_y;
 public:
  Point(int x, int y) {
    m_x = x; m_y = y;
  }
  void print() {
    cout<<"Point("<<m_x<<","<<m_y<<")\n";
  }
};
```

# Access control: protected members

New status, adding to `public` and `private`: `protected`

- Inaccessible for class users
- Accessible to heirs (or successors)

## Example

```cpp
class Point {
 protected:
  int m_x, m_y;
 public:
  Point(int x, int y) {
    m_x = x; m_y = y;
  }
  void print() {
    cout<<"Point("<<m_x<<","<<m_y<<")\n";
  }
};
```

```cpp
class ColoredPoint : public Point {
  short m_color;
 public:
  ColoredPoint(int x, int y, short color)
    : Point(x,y) {
    m_color = color;
  }
  void print() {// access to protected members
    cout<<"ColoredPoint("<<m_x<<","<<m_y
                        <<","<<m_color")\n";
  }
};
```

# Access control: protected members

New status, adding to `public` and `private`: `protected`

- Inaccessible for class users
- Accessible to heirs (or successors)

## Example

```cpp
class Point {
 protected:
  int m_x, m_y;
 public:
  Point(int x, int y) {
    m_x = x; m_y = y;
  }
  void print() {
    cout<<"Point("<<m_x<<","<<m_y<<")\n";
  }
};
```

```cpp
class ColoredPoint : public Point {
  short m_color;
 public:
  ColoredPoint(int x, int y, short color)
    : Point(x,y) {
    m_color = color;
  }
  void print() {// access to protected members
    cout<<"ColoredPoint("<<m_x<<","<<m_y
                        <<","<<m_color")\n";
  }
};
```

## Half public, half private

Example:

```cpp
1 int main () {
2   ColoredPoint cp( 10, 2, 1 );
3   cp.print(); // displays using ColoredPoint::print();
4   cout<<"Point at ("<<pc.m_x<<","<<pc.m_y<<")\n";
5   return 0;
6 }
```

$\Longrightarrow$ Compilation error line 4: `m_x` and `m_y` are encapsulated

# Public and private derivations

Three authorized derivation modes, same keywords than members ...

```
class ColorPoint
        : public Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

```
class ColorPoint
        : protected Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

```
class ColorPoint
        : private Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

# Public and private derivations

Three authorized derivation modes, same keywords than members ...

```
class ColorPoint
        : public Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

```
class ColorPoint
        : protected Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

```
class ColorPoint
        : private Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

$\Longrightarrow$ Modify access to public or protected members ...

- Except explicit mention (re-declaration, or `using` instruction)
- Example with member function `move()` ...

# Public and private derivations

Three authorized derivation modes, same keywords than members ...

```
class ColorPoint
        : public Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

```
class ColorPoint
        : protected Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

```
class ColorPoint
        : private Point
{  short m_color
 public:
  ColorPoint(int, int, short);
  void print();
  void color( int );
};
```

$\Longrightarrow$ Modify access to public or protected members ...

- Except explicit mention (re-declaration, or `using` instruction)
- Example with member function `move()` ...

## Summary

| Base class | | | public | | protected | | private | |
|---|---|---|---|---|---|---|---|---|
| Initial Status | Access FMF | User access | New status | User Access | New status | User Access | New status | User Access |
| `public` | Y | Y | `public` | Y | `protected` | N | `private` | N |
| `protected` | Y | N | `protected` | N | `protected` | N | `private` | N |
| `private` | Y | N | `private` | N | `private` | N | `private` | N |

NB: access FMF means Friend of Member Functions

# Compatibility between base and derived classes

Basic rules: derived class object may replace base class object

## *upcasting* example

```cpp
enum note { middleC, Csharp, Cflat }; /// Etc.

class Instrument {
public:
  void play(note) const {}
};

/// Wind objects are Instruments
/// because they have the same interface:
class Wind : public Instrument {};
```

```cpp
void tune(Instrument& i) {
  /// ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
  return 0;
}
```

# Compatibility between base and derived classes

Basic rules: derived class object may replace base class object

## *upcasting* example

```
enum note { middleC, Csharp, Cflat }; /// Etc.

class Instrument {
public:
  void play(note) const {}
};

/// Wind objects are Instruments
/// because they have the same interface:
class Wind : public Instrument {};
```

```
void tune(Instrument& i) {
  /// ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
  return 0;
}
```

## Pointer cast

```
class Point {
  int m_x, m_y;
 public:
  ...
  void print();
};
```

```
class ColorPoint : public Point {
  short m_color;
 public:
  ...
  void print(); // redefinition
};
```

Only (natural) cast bottom to “the top”:

```
Point      p (1,2) ,   *adp  = &p;
ColorPoint cp(0,0,1) , *adcp = &cp;
adp  = adcp; // OK, go up into tree view
adcp = adp ; // Compilation error (without cast operator)
```

# Limitations linked to static type of object

```
class Point { /// Again, same object definition
protected:
  int m_x, m_y;
public:
  Point( int a=0, int o=0 ) : m_x(a), m_y(o) {}
  void print() const { cout<< "Point(" << m_x << "," << m_y << ")" <<endl;  }
};
class ColoredPoint : public Point { /// Child
  short m_color;
public:
  ColoredPoint( int a=0, int o=0, short color=1 ) : Point( a, o ), m_color( color ) {}
  void print() const {
    cout<< "ColoredPoint(" << m_x << "," << m_y << "," << m_color << ")" <<endl;
  }
};
int main () {
  Point p( 3, 5 );            Point *adp = &p;
  ColoredPoint cp( 8, 6, 2 ); ColoredPoint *adcp = &cp; // IT WILL DISPLAYS
  adp->print();               adcp->print();            // Point(3,5)
  cout<< "--------------------------------" <<endl;     // ColoredPoint(8,6,2)
  adp = adcp;                                           // --------------------------------
  adp ->print();                                        // Point(8,6)
  adcp->print();                                        // ColoredPoint(8,6,2)
  return 0;
}
```

# Limitations linked to static type of object

```
class Point { /// Again, same object definition
protected:
  int m_x, m_y;
public:
  Point( int a=0, int o=0 ) : m_x(a), m_y(o) {}
  void print() const { cout<< "Point(" << m_x << "," << m_y << ")" <<endl;  }
};
class ColoredPoint : public Point { /// Child
  short m_color;
public:
  ColoredPoint( int a=0, int o=0, short color=1 ) : Point( a, o ), m_color( color ) {}
  void print() const {
    cout<< "ColoredPoint(" << m_x << "," << m_y << "," << m_color << ")" <<endl;
  }
};
int main () {
  Point p( 3, 5 );              Point *adp = &p;
  ColoredPoint cp( 8, 6, 2 ); ColoredPoint *adcp = &cp; // IT WILL DISPLAYS
  adp->print();                 adcp->print();            // Point(3,5)
  cout<< "---------------------------------" <<endl;     // ColoredPoint(8,6,2)
  adp = adcp;                                             // ---------------------------------
  adp ->print();                                          // Point(8,6)
  adcp->print();                                          // ColoredPoint(8,6,2)
  return 0;
}
```

- Method identification done by compiler, statically
  - Virtual function allows dynamic choice (next lecture)
- Risk to access to private member of base class ...

# Overview

- Simple inheritance
- Inheritance and access control
- Canonical form of derived class
- Multiple inheritance

# By-copy constructor (CC)

## First case: no CC into derived class

Then, usage of “default” constructor

1. Using CC base class when it exists
2. Else using default CC base class
3. Warning: if CC constructor is private in base class ...

# By-copy constructor (CC)

## First case: no CC into derived class

Then, usage of "default" constructor

1. Using CC base class when it exists
2. Else using default CC base class
3. Warning: if CC constructor is private in base class ...

## Second case: writing CC into derived class

C++ do not handle automatic call of base class CC

```
class Point {
protected:  int m_x, m_y;
public:     Point( int a=0, int o=0 ) : m_x(a), m_y(o) {}
            Point( Point& p ) : m_x(p.m_x), m_y(p.m_y) {}
};
class ColoredPoint : public Point { // We explicitly call the base class constructor
private: short m_c;                // (following hierarchical order)
public:  ColoredPoint(int a=0, int o=0, short c=1) : Point( a, o ) , m_c(c) { };
         ColoredPoint( ColoredPoint& cp ) : Point(cp), m_c(cp.m_c) {};
};
void fct ( ColoredPoint cp ) {  cout<< "fct()" << endl;  }
int main () {  ColoredPoint cp(2,4,6);
               fct( cp );              /// What the constructor call order is?
               return 0;               /// Same for destructors and functions?
}
```

# Assignment operator and inheritance

## First case: derived class does not overload =

Behavior is similar to by-copy constructor case ...

# Assignment operator and inheritance

## First case: derived class does not overload =

Behavior is similar to by-copy constructor case ...

## Second case: derived class does overload =

Base class assignment have to be considered ...

```
class Point {
  int m_x, m_y;
public:
  ...
  Point& operator=(Point& p) {
    m_x = p.m_x;
    m_y = p.m_y;
    return *this;
  }
};
```

```
class ColoredPoint : public Point {
  short m_color;
public:
  ...
  ColoredPoint& operator=(ColoredPoint& pc) {
    Point::operator=( pc ); // <- DO NOT FORGET
    m_color = pc.m_color;   //    TO COPY THE
    return *this;           //    INHERITED PARTS
  }
};
```

# Assignment operator and inheritance

## First case: derived class does not overload =

Behavior is similar to by-copy constructor case ...

## Second case: derived class does overload =

Base class assignment have to be considered ...

```cpp
class Point {
  int m_x, m_y;
public:
  ...
  Point& operator=(Point& p) {
    m_x = p.m_x;
    m_y = p.m_y;
    return *this;
  }
};
```

```cpp
class ColoredPoint : public Point {
  short m_color;
public:
  ...
  ColoredPoint& operator=(ColoredPoint& pc) {
    Point::operator=( pc ); // <- DO NOT FORGET
    m_color = pc.m_color;   //    TO COPY THE
    return *this;           //    INHERITED PARTS
  }
};
```

```cpp
int main () {
  ColoredPoint cp(2,4,6);
  ColoredPoint cp2 = cp; // by-copy constructor
  cp2 = cp;              // assignment operator
  return 0;
}
```

# Canonical form of derived class

Complete base class scheme seen in lecture 4:

```
/// Base class
class T
{
 public:
  T ( ... ) ;                       /// constructors, others than by-copy
  T ( const T& ) ;                  /// by-copy constructor (recommended form)
  ~T ();                            /// destructor
  T& T::operator=( const T& ); /// assignment operator (recommended form)
};

/// Derived class
class U : public T
{
 public:
  U (...) : T( ... ) , ...      {} // Recommended to use complete initialization list
  U ( const U& x) : T(x) , ... {} // (including all members, plus the base class)

  ~U ();

  U& U::operator=( const U& x )
  {
    T::operator=( x ) ;             // first, copy the base class
    ... // then do what you have to
  }
};
```

# The limits of inheritance

Let us consider following situation:

```
class A {
  ...
 public:
  T f(...);
  ...
};
```

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

# The limits of inheritance

Let us consider following situation:

```cpp
class A {
  ...
 public:
  T f(...);
  ...
};
```

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

## Type of return value of `f()`

- Do not worry if `T` is any
- If `T == A`, how to return `B` in `B::f`? Is it desirable?
  - Solution using virtual function (next lecture)

# The limits of inheritance

Let us consider following situation:

```
class A {
  ...
 public:
  T f(...);
  ...
};
```

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

## Type of return value of `f()`

- Do not worry if `T` is any
- If `T == A`, how to return `B` in `B::f`? Is it desirable?
  - Solution using virtual function (next lecture)

## Argument types of `f()`

Example with `T f(A);`

- Sending instance of A: ok
- Sending instance of B: ok, but after cast

# The limits of inheritance

Let us consider following situation:

```
class A {
  ...
 public:
  T f(...);
  ...
};
```

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

## Type of return value of `f()`

- Do not worry if `T` is any
- If `T == A`, how to return `B` in `B::f`? Is it desirable?
  - Solution using virtual function (next lecture)

## Argument types of `f()`

Example with `T f(A);`

- Sending instance of A: ok
- Sending instance of B: ok, but after cast

Example `limits.cpp`: addition and equals ...

# Example: limits.cpp

```cpp
#include <iostream>
using namespace std;
class Point {
protected:
  int m_x, m_y;
public:
  Point( int a=0, int o=0) : m_x(a), m_y(o) {};
  Point( const Point& p ) : m_x(p.m_x), m_y(p.m_y) {};
  Point &operator+( const Point& p ) { /// What meaning for ColoredPoint?
    m_x += p.m_x;
    m_y += p.m_y;
    return (*this);
  }
  friend int equals ( const Point&, const Point& ); /// Quid with ColoredPoint?
  void print() { cout<<"Point("<<m_x<<","<<m_y<<")"<<endl; }
};
int equals( const Point& a, const Point& b ) {  return a.m_x == b.m_x && a.m_y == b.m_y; }
class ColoredPoint : public Point {
  short m_color;
public:
  ColoredPoint( int a=0, int o=0, short c=1 ) : Point( a, o ), m_color(c) {};
  ColoredPoint( const ColoredPoint& p ) : Point( p ), m_color(p.m_color)  {};
};
int main () {
  ColoredPoint a(2,5,3), b(2,5,9);
  Point c;
  if ( equals( a, b ) ) cout<<"a equals to b"<<endl;
  else cout<<"a differs from b"<<endl;        // Yes, they are equals as points!
  if ( equals( a, c ) ) cout<<"a equals to c"<<endl;
  else cout<<"a differs from c"<<endl;        // No, different coordinates
  c = a+b;     c.print(); /// Compilation error if c was ColoredPoint
  return 0;
}
```

# Example of derived class

Let us restart with dynamic vector seen in lecture 4, slide 20:

```
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) { m_v = new int[ m_n = n ]; } /// lacks of by-copy constructor
  ~Vector () { delete m_v; }                   /// and assignment operator
  int & operator [] ( int idx ) { return m_v[ idx<0? 0 : idx>=n ? n-1 : idx ]; }
};
```

# Example of derived class

Let us restart with dynamic vector seen in lecture 4, slide 20:

```
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) { m_v = new int[ m_n = n ]; } /// lacks of by-copy constructor
  ~Vector () { delete m_v; }                   /// and assignment operator
  int & operator [] ( int idx ) { return m_v[ idx<0? 0 : idx>=n ? n-1 : idx ]; }
};
```

Let us build dynamic vector with variable limits:

```
VVector v( 15, 24 ) ; // vector with 10 elements, indexed from 15 to 24 ...
```

# Example of derived class

Let us restart with dynamic vector seen in lecture 4, slide 20:

```
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) { m_v = new int[ m_n = n ]; } /// lacks of by-copy constructor
  ~Vector () { delete m_v; }                   /// and assignment operator
  int & operator [] ( int idx ) { return m_v[ idx<0? 0 : idx>=n ? n-1 : idx ]; }
};
```

Let us build dynamic vector with variable limits:

```
VVector v( 15, 24 ) ; // vector with 10 elements, indexed from 15 to 24 ...
```

Constructor with two arguments, and two supplementary instance variables:

```
class VVector : public Vector {
  int m_start, m_end;
public:
  VVector( int start, int end ) : Vector(end-start+1), m_start(start), m_end(end) {}
```

# Example of derived class

Let us restart with dynamic vector seen in lecture 4, slide 20:

```
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) { m_v = new int[ m_n = n ]; } /// lacks of by-copy constructor
  ~Vector () { delete m_v; }                   /// and assignment operator
  int & operator [] ( int idx ) { return m_v[ idx<0? 0 : idx>=n ? n-1 : idx ]; }
};
```

Let us build dynamic vector with variable limits:

```
VVector v( 15, 24 ) ; // vector with 10 elements, indexed from 15 to 24 ...
```

Constructor with two arguments, and two supplementary instance variables:

```
class VVector : public Vector {
  int m_start, m_end;
public:
  VVector( int start, int end ) : Vector(end-start+1), m_start(start), m_end(end) {}
```

Do we need a destructor? No, it is useless ...

# Example of derived class

Let us restart with dynamic vector seen in lecture 4, slide 20:

```
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) { m_v = new int[ m_n = n ]; } /// lacks of by-copy constructor
  ~Vector () { delete m_v; }                   /// and assignment operator
  int & operator [] ( int idx ) { return m_v[ idx<0? 0 : idx>=n ? n-1 : idx ]; }
};
```

Let us build dynamic vector with variable limits:

```
VVector v( 15, 24 ) ; // vector with 10 elements, indexed from 15 to 24 ...
```

Constructor with two arguments, and two supplementary instance variables:

```
class VVector : public Vector {
  int m_start, m_end;
public:
  VVector( int start, int end ) : Vector(end-start+1), m_start(start), m_end(end) {}
```

Do we need a destructor? No, it is useless ...

Access to elements: mandatory to overload [] operator ...

```
  int &operator[] ( int i ) {
    return Vector::operator[] ( i-m_start );
  }
  int operator[] (int i) const { return Vector::operator[](i-m_start); }
};
```

# Example of derived class

Let us restart with dynamic vector seen in lecture 4, slide 20:

```
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) { m_v = new int[ m_n = n ]; } /// lacks of by-copy constructor
  ~Vector () { delete m_v; }                   /// and assignment operator
  int & operator [] ( int idx ) { return m_v[ idx<0? 0 : idx>=n ? n-1 : idx ]; }
};
```

Let us build dynamic vector with variable limits:

```
VVector v( 15, 24 ) ; // vector with 10 elements, indexed from 15 to 24 ...
```

Constructor with two arguments, and two supplementary instance variables:

```
class VVector : public Vector {
  int m_start, m_end;
public:
  VVector( int start, int end ) : Vector(end-start+1), m_start(start), m_end(end) {}
```

Do we need a destructor? No, it is useless ...

Access to elements: mandatory to overload [] operator ...

```
  int &operator[] ( int i ) {
    return Vector::operator[] ( i-m_start );
  }
  int operator[] (int i) const { return Vector::operator[](i-m_start); }
};
```

NB: It lacks by-copy constructor and assignment operator ;-)

# Overview

- Simple inheritance
- Inheritance and access control
- Canonical form of derived class
- Multiple inheritance

# Introduction to multiple inheritance

An class may naturally inherits from many others ...

- Unique constraint: oriented graph without circuit (no self inheritance)
- Almost all things seen for simple inheritance apply to multiple case ...
- Some problems

# Introduction to multiple inheritance

An class may naturally inherits from many others ...

- Unique constraint: oriented graph without circuit (no self inheritance)
- Almost all things seen for simple inheritance apply to multiple case ...
- Some problems

## Owed difficulties to multiple inheritance

- How to express multiple dependency?
- Which order for constructor calls / destructor calls?
- Conflict management: D inherits both from B and from C, what inherit from A

# Multiple inheritance application

Similar to multiple composition ... Let us take an example:

```
class Point {
  int m_x, m_y;
public:
  Point(int a=0, int o=0) : m_x(a), m_y(o) {}
  ~Point() {}
  print() { cout<< ... << endl; }
};
```

```
class Color {
  short m_color;
public:
  Color(short c=1) : m_color(c) {}
  ~Color() {}
  print() { cout<< ... << endl; }
};
```

```
class ColoredPoint : public Point, public Color {
public:                                          // Call both constructors ...
  ColoredPoint( int a=0, int o=0, short c=1 ) : Point( a, o ), Color( c ) {}
  ~ColoredPoint() { cout<<"---- ColoredPoint() releasing" << endl; }
  void print() {
    Point::print();   /// range resolution allows correct selection
    Color::print();   /// of inherited methods and data members
  }
};
```

# Multiple inheritance application

Similar to multiple composition ... Let us take an example:

```
class Point {
  int m_x, m_y;
public:
  Point(int a=0, int o=0) : m_x(a), m_y(o) {}
  ~Point() {}
  print() { cout<< ... << endl; }
};
```

```
class Color {
  short m_color;
public:
  Color(short c=1) : m_color(c) {}
  ~Color() {}
  print() { cout<< ... << endl; }
};
```

```
class ColoredPoint : public Point, public Color {
public:                                   // Call both constructors ...
  ColoredPoint( int a=0, int o=0, short c=1 ) : Point( a, o ), Color( c ) {}
  ~ColoredPoint() { cout<<"---- ColoredPoint() releasing" << endl; }
  void print() {
    Point::print();  /// range resolution allows correct selection
    Color::print();  /// of inherited methods and data members
  }
};
```

Order of constructors and destructors calls:

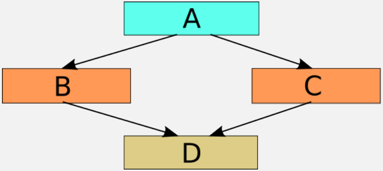
- Constructors: order of base classes declaration (eventually recursive), then constructor ...
- Destructors : the reverse ;-)

Example by instrumenting the previous 3 classes (`sources/multiple1`)...

# Multiples occurrences of a same base class

Let us consider following situation:

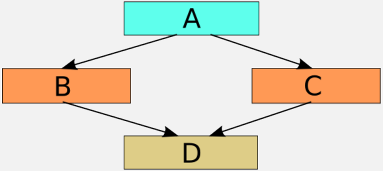
- B inherits from A
- C inherits from A
- D inherits from B and C



# Multiples occurrences of a same base class

Let us consider following situation:

- B inherits from A
- C inherits from A
- D inherits from B and C




It is like if any D's instance contains:

- 1 instance of B, 1 instance of C, and 2 different instances of A !
- Differentiation into D:

```
class A { int m_x; ... };
class B : public A { ... };
class C : public A { ... };
class D ; public B, public C {
  ...
  void f () {
    cout << "m_x = " << B::m_x << " and " << C::m_x << endl;
  }
};
```

Example with `sources/multiple2.cpp`

# Example

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

class A {
protected:
  int m_x;
public:
  A(int x=0) : m_x(x) { cout<< "++++ A("<< m_x << ")" << endl; }
  ~A()                { cout<< "---- A("<< m_x << ")" << endl; }
};

class B : protected A {
public:
  B(int x=0) : A(x + 10) { cout<< "++++ B("<< A::m_x << ")" << endl; }
  ~B()                   { cout<< "---- B("<< A::m_x << ")" << endl; }
};

class C : protected A {
public:
  C(int x=0) : A(x + 100) { cout<< "++++ C("<< A::m_x << ")" << endl; }
  ~C()                    { cout<< "---- C("<< A::m_x << ")" << endl; }
};

class D : protected B, protected C {
public:
  D(int x=0) : B(x), C(x) { cout<< "++++ D("<< B::m_x << "," << C::m_x << ")" << endl; }
  ~D()                    { cout<< "---- D("<< B::m_x << "," << C::m_x << ")" << endl; }
};

int main () {     D(3);
                  return 0;
}
```

# Notion of virtual class

It is possible to avoid this duplication with virtual classes

```
class B : public virtual A { ... }; // or: virtual public A
class C : public virtual A { ... }; // (order does not mind)
class D : public B, public C {
  ...
  void f () { /// no more double, so no more range resolution
    cout << " x = " << m_x << endl;
  }
};
```

Defining A as virtual into B's declaration means that A will be copied once into all heirs of B ...

# Notion of virtual class

It is possible to avoid this duplication with virtual classes

```
class B : public virtual A { ... }; // or: virtual public A
class C : public virtual A { ... }; // (order does not mind)
class D : public B, public C {
  ...
  void f () { /// no more double, so no more range resolution
    cout << " x = " << m_x << endl;
  }
};
```

Defining A as virtual into B's declaration means that A will be copied once into all heirs of B ...

For B, it changes nothing ...

# Notion of virtual class

It is possible to avoid this duplication with virtual classes

```
class B : public virtual A { ... }; // or: virtual public A
class C : public virtual A { ... }; // (order does not mind)
class D : public B, public C {
  ...
  void f () { /// no more double, so no more range resolution
    cout << " x = " << m_x << endl;
  }
};
```

Defining A as virtual into B's declaration means that A will be copied once into all heirs of B ...

For B, it changes nothing ...

**Order of constructors and destructors calls**

- First come constructor of virtual classes
- Then, the others

# Example: constructor calls order

```cpp
#include <iostream>
using namespace std;
////////////////////////////////////////////
class O {
public:
  O()  { cout << "++++ O()" << endl; }
  ~O() { cout << "---- O()" << endl; }
};
////////////////////////////////////////////
class A : public O {
protected:
  int m_x;
public:
  A(int x=0) : O(), m_x(x) {
    cout<< "++++ A("<< m_x << ")" << endl;
  }
  ~A() {
    cout<< "---- A("<< m_x << ")" << endl;
  }
};

////////////////////////////////////////////
class B : public virtual A {
public:
  B(int x=0) { // no direct constructor
    m_x = x + 10;
    cout<< "++++ B("<< A::m_x << ")" << endl;
  }
  ~B() {
    cout<< "---- B("<< A::m_x << ")" << endl;
  }
};
////////////////////////////////////////////
class C : virtual public A {
public:
  C(int x=0) {
    m_x = x+100;
    cout<< "++++ C("<< A::m_x << ")" << endl;
  }
  ~C() {
    cout<< "---- C("<< A::m_x << ")" << endl;
  }
};

////////////////////////////////////////////
class D : public B, public C {
public:
  D(int x=0) : B(x), C(x) {
    cout<< "++++ D("<< B::m_x << ","
        << C::m_x << ")" << endl;
  }
  ~D() {
    cout<< "---- D("<< B::m_x << ","
        << C::m_x << ")" << endl;
  }
};

////////////////////////////////////////////
int main () {
  D(3);
  return 0;
}
// test : ./sources/multiple3
```

# Example of multiple inheritance I

```cpp
#include <iostream>
using namespace std;
///
class Point {
protected:
  int m_x, m_y;
public:
  Point( int abs=0, int ord=0) : m_x(abs), m_y(ord) {
    cout << "++++++ Point(" << abs << "," << ord << ")" << endl;
  }
  void print() { cout << "Coordinates: " << m_x << " and " << m_y << endl; }
};
///
class Color {
  short m_color;
public:
  Color( short color=1 ) : m_color( color ) {
    cout << "++++ Color(" << m_color << ")" << endl;
  }
  void print() { cout << "Color : " << m_color << endl; }
};
///
class Mass {
  int m_mass;
public:
  Mass( int m=100 ) : m_mass( m ) {
    cout << "++++ Mass(" << m_mass << ")" << endl;
  }
  void print () { cout << "Mass: " << m_mass << endl; }
};
```

# Example of multiple inheritance II

```
class ColoredPoint : public virtual Point, public Color {
public:
  ColoredPoint( int abs, int ord, int cl ) : Point(abs, ord), Color( cl ) {
    cout << "++ ColoredPoint(" << m_x << "," << m_y << "," << cl << ")" << endl;
  }
  ColoredPoint( int cl ) : Color( cl ) { cout << "++ ColoredPoint(" << cl << ")" << endl; }
  void print() {    Point::print();    Color::print();   }
};
///
class MassPoint : public virtual Point, public Mass {
public:
  MassPoint( int abs, int ord, int ms ) : Point(abs, ord), Mass( ms ) {
    cout << "++ MassPoint(" << m_x << "," << m_y << "," << ms << ")" << endl;
  }
  void print() {    Point::print();    Mass::print();   }
};
///
class ColoredMassPoint : public ColoredPoint, public MassPoint {
public:
  ColoredMassPoint( int abs, int ord, short c, int m )
    : Point( abs, ord ), ColoredPoint( c ), MassPoint( abs, ord, m ) {
    cout << "ColoredMassPoint(" << abs << "," << ord << "," << c << "," << m << ")" << endl;
  }
  void print() {    Point::print();    Color::print();    Mass::print();   }
};
///
int main () {
  cout << "****************" << endl;  ColoredPoint     p( 3, 9, 2 );          p.print();
  cout << "****************" << endl;  MassPoint        mp( 12, 25, 100 );    mp.print();
  cout << "****************" << endl;  ColoredMassPoint cmp( 2, 5, 10, 20 ); cmp.print();
  return 0;
}
```