Object Oriented Programming
USTH, Master ICT, year 1

Aveneau Lilian

lilian.aveneau@univ-poitiers.fr
XLIM/ASALI/IG, CNRS, Computer Science Department
University of Poitiers

2017/2018

Lecture 4 – Friendship & Operator Overloading

- Friendship
- Friendship examples: vector & matrix
- Operator overloading
- Generalities about operator overloading
- Example of overloading with $=$
- Canonical form of class
- Overloading indexed access and parentheses
- Overloading `new` and `delete`
- Cast

Introduction

Pure OOP requires data encapsulation

- IV are “private”
- Protection unit is the class
- Forbids access to private members of another class

Painful constraint in some case

Example: matrix by vector product

- Make data public? Loose encapsulation benefits
- Create some getters? Good, but extra execution time

Solution

- Friend function: grant access to private data
- Advantage: allows access control at class level

Independent friend function

Let us restart from lecture 3 class "Point":

```
#include <iostream>
using namespace std;
class Point {
  int x, y;
public:
  Point (const int abs=0, const int ord=0) { x=abs; y=ord; }
  friend int is_equal (const Point&, const Point&); // May be anywhere inside class
};
// do not recall the friend relationship
inline int is_equal (const Point& p, const Point& q) { // Separate compilation is possible
  return p.x == q.x && p.y == q.y;                    // No more "this"
}
int main () {
  Point a(1,0), b(1), c;
  if ( is_equal (a,b) ) cout<< "a is equal to b" << endl;
  else                  cout<< "a different from b" << endl;
  if ( is_equal (a,c) ) cout<< "a is equal to c" << endl;
  else                  cout<< "a different from c" << endl;
  return 0;
}
```

Remarks

- Friend function of given class A generally receives or returns a value of type A
- When it returns: always by value

Member function is friend of other class

Particular case of the previous one, implies range resolution

```
class A {
  /// private part
  ....
  /// public part
  ...
  // f is a function from class B
  friend int B::f (char, A);
};
```

```
class B {
  ....
  int f (char, A);
  ....
};
int B::f( char c, A a ) {
  // f has access to A's private members
}
```

Remarks

- To compile A, an anterior declaration of B is needed
  But to compile B, class A must be known

```
class A; // A is a class

class B {
  ....
  int f (char, A);
  ....
};
```

```
class A { // we know B
  ....
  friend int B::f (char, A);
  ...
};
int B::f( char c, A a ) {
  ... // Knows A and B
}
```

- Cross declaration between functions from different classes:
  declare the one friend of the other

Friend function of many classes, friendship between classes

Friend of some classes

```
class B;

class A
{
  ....
  friend void f (A, B);
  ....
};
```

```
class B {
  ....
  friend void f (A, B);
  ...
};
void f( A a, B b ) {
  ... // knows A and B
}
```

One more time, take care to declarations

- Known classes into other classes
- Classes declared before friend functions

Friendship between classes

- Simplify writing: grant access to private members to all methods of granter class
- We add into granter class: `friend class B;`
  - Need to know that B is a class
  - Avoid to give header of concerned functions

Example: independent friend function

```cpp
#include <iostream>
using namespace std;
class Matrix; // to declare "product" into vector
class Vector {
  double m_v[3];
public:
  Vector (double x=0, double y=0, double z=0) { m_v[0]=x; m_v[1]=y; m_v[2]=z; }
  void print () const { cout<<"vector("<<m_v[0]<<","<<m_v[1]<<","<<m_v[2]<<")"<<endl; }
  friend Vector product (const Matrix&, const Vector&); // friend to two classes
};
class Matrix { // We may have start with matrix ...
  double m_m[3][3]; // line, row
public:
  Matrix(double t[][3]) {
    for (int i=0;i<3;i++)
      for (int j=0;j<3;j++) m_m[i][j] = t[i][j];
  }
  friend Vector product (const Matrix&, const Vector&);
};
Vector product ( const Matrix& m, const Vector& v ) { // friend function of 2 classes
  Vector res; /// all to zero
  for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)  res.m_v[i] += m.m_m[i][j]*v.m_v[j];
  return res;
}
int main () {
  Vector v(1,2,3);
  double t[3][3] = { {1,2,3}, {4,5,6}, {7,8,9} };
  Matrix m(t);
  const Vector p=product (m, v);     p.print(); // display "vector(14,32,50)"
  return 0;
}
```

Example: friend function member of class

```
#include <iostream>
using namespace std;
class Vector;  // To declare "product" into Matrix
class Matrix { // Must be declare before Vector ...
  double m_m[3][3]; // line, row
public:
  Matrix(double t[][3]) {
    for (int i=0;i<3;i++)
      for (int j=0;j<3;j++) m_m[i][j] = t[i][j];
  }
  Vector product ( const Vector& ) const; // It is a method, now!
};
class Vector { // Must be declare before Matrix::product
  double m_v[3];
public:
  Vector (double x=0, double y=0, double z=0) { m_v[0]=x; m_v[1]=y; m_v[2]=z; }
  void print () const { cout<<"Vector("<<m_v[0]<<","<<m_v[1]<<","<<m_v[2]<<")"<<endl; }
  friend Vector Matrix::product (const Vector&) const;
};
Vector Matrix::product ( const Vector& v ) const { // range resolution
  vecteur res; /// set to zero
  for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)  res.m_v[i] += m_m[i][j]*v.m_v[j];
  return res;
}
int main () {
  Vector v(1,2,3);
  double t[3][3] = { {1,2,3}, {4,5,6}, {7,8,9} };
  Matrix m(t);
  const Vector p=m.product (v);     p.print(); // display "Vector(14,32,50)"
  return 0;
}
```

Introduction

Function overloading: same name for many $\neq$ functions

- Good one chosen by compiler

C++ extends this behavior to class operator

- Already exists in C for fundamental types:
  - Example a+b: + works with integers, reals, doubles ...
  - Idem with $*$ for the product, or pointer indirection

C++ mechanism

Example with class Point:

```
class Point {
  int m_x, m_y;
public:
  Point(int=0, int=0);
  Point operator + (Point&) const;
};
```

- Operator "+" is a correct internal product
- Keyword `operator`, followed by overloaded operator
- Can be written with friend function!

Overloading with friend function

In such a case, symmetrical aspect appears for binary operators

```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; }
  void print() { cout<<"Point("<<m_x<<","<<m_y<<")"<<endl; }
  friend Point operator+ (const Point&, const Point&) ;
};

Point operator+(const Point&a, const Point&b) {
  return Point( a.m_x+b.m_x, a.m_y+b.m_y );
}

int main () {
  Point a(1,2); a.print();
  Point b(2,5); b.print();
  Point c;
  c = a+b;   c.print(); // equivalent to ``c = operator + (a, b);''
  c = a+b+c; c.print(); // equivalent to ``c = operator + (operator + (a, b) , c);''
  return 0;
}
// Point(1,2)
// Point(2,5)
// Point(3,7)
// Point(6,14)
```

Operator overloading with member function

In such a case, asymmetric aspect appears for binary operators

```
#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; }
  void print() { cout<<"Point("<<m_x<<","<<m_y<<")"<<endl; }
  Point operator+ (const Point&) const;
};
Point Point::operator+(const Point&p) const { // binary operator, but asymmetric writing
  return Point(m_x+p.m_x, m_y+p.m_y);
}
int main () {
  Point a(1,2); a.print();
  Point b(2,5); b.print();
  Point c;
  c = a+b;   c.print(); // equivalent to ``c = a.operator + (b);''
  c = a+b+c; c.print(); // equivalent to ``c = c.operator + (a.operator + (b));''
  return 0;
}
// Same result as with the previous example
```

- Asymmetry: want to use friend function? Warning, bad idea
- c=a+b+c: may creates temporary objects (depends on compiler)
- Reference: should be used for arguments (with `const`)
  generally forbidden for returning

Use only existing operators

- No new operator (except `->*`, `.*`, `new` ...)
- Respect the operator priority

Arity	Operators	Associativity
Binary	() [] ->	->
Unary	+ − ++ −− ! ∼ * & new new[] delete delete[] (cast)	<-
Binary	* %	->
Binary	->* .*	->
Binary	+ −	->
Binary	<< >>	->
Binary	< <= >= >	->
Binary	== !=	->
Binary	&	->
Binary	∧	->
Binary	$\vert\vert$	->
Binary	&&	->
Binary	$\vert$	->
Binary	= += -= *= /= %= &= ∧= $\vert$= <<= >>=	->
Binary	,	->

- All except ".", "::" and "?:"
- Operators $->*$ and $.*$ have restricted usage: applicable to pointers to members
- Post and pre incrementation/decrementation: pre is the unary version, post is the binary one with `int` argument

Being in class context

At least one argument must be of class type

It must be either:

- Member function: argument of type implicit class. For binary operator, second argument may be of any type ...
- Independent function: generally friend function, with necessarily an argument of class type

Guarantee impossibility to overload fundamental type operators ...
Exception: `new` and `delete`, globally definable

Specific limits

- [], (), $->$, `new` and `delete`
- Only a member function

Avoid hypothesis on operators role

Programmer write operators: show some common sense!

No link between operators

Example of integers with $+$, $=$ and $+=$

- For a class: need to write the 3 (no automatic generation)
- Moreover, programmer may furnish another meanings

Commutativity?

C++ do not know!

- Example with
  `friend Complex operator+(Complex, double);`
  $\Longrightarrow$ do not imply
  `friend Complex operator+(double, Complex);`
- Faster solution: overload cast operator!

Operators ++ and −−

Since C++ version 3.0, there exists distinction between post and pre ...

Convention: post version has an unused argument of type `int`

Example for index of matrix of dimension 2

```
#include <iostream>
using namespace std;
class Index {
  unsigned int m_i, m_j, m_nb_col, m_nb_lig;
  Index() {} // to forbid it ;-)
public:
  Index( unsigned int nb_col, unsigned int nb_lig ) {
    m_nb_col = nb_col; m_nb_lig = nb_lig; m_i = m_j = 0; }
  Index(Index& i) { m_nb_col=i.m_nb_col; m_nb_lig=i.m_nb_lig; m_i=i.m_i; m_j=i.m_j; }
  void print() const { cout<<m_i<<" "<<m_j<<endl; }
  Index& operator++() { // pre
    if ( ++m_j >= m_nb_col ) { m_j = 0; if ( ++m_i >= m_nb_lig ) m_i = 0; }
    return *this; // current value
  }
  Index operator++(int nothing __attribute__((unused))) { // post
    Index res(*this); // by copy constructor
    if ( ++m_j >= m_nb_col ) { m_j = 0; if ( ++m_i >= m_nb_lig ) m_i = 0; }
    return res; // returns the old value!
  }
};
int main () {
  Index idx(2,2);
  for (int i=0; i<2; i++) for (int j=0; j<2; j++) (++idx).print();
  for (int i=0; i<2; i++) for (int j=0; j<2; j++) (idx++).print();
  return 0;
}
```

Operators = and &

In general: any undefined operator cannot be used ...

$\Longrightarrow$ lead to compilation error

There exist 2 exceptions

- Operator &, that returns an object address by default
- Operator =, that exists by default
  - Recopies member by member like C-struct
  - Thus problem with dynamically allocated members
  - Analogy with by copy-constructor: if it does not exist, it also do a copy member-by-member
  - If members are objects, by default assignment operator may use the operator defined on the target class

Reminder

Dynamic vector case

```
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) ;
  ~Vector() { delete m_v; };
  ...
};
```

- Argument of function:

```
Vector a(5);
void f(Vector);
f(a);
```

destructor is called, `m_v` released ...

Assignment case

Have same problems: with `Vector a(5), b(3);`

- `b=a;` exhibits same double free risk ...
- Moreover, problem of memory leaks

Solution: copy operator

with good memory management, and checking equality `a=a`

Return value choice:

must return the object to manage multiple assignment

```cpp
#include <iostream>  /// Warning: it is an example, with missing important methods
using namespace std; /// for instance the copy constructor ...
class Vector {
  int  m_n;      // number of elements
  int *m_v;      // pointer to first element
public:
  Vector (int n) {
    m_v = new int[ m_n = n ];
    for (int i=0; i<m_n; i++) m_v[i] = 0;
    cout << "++ obj size " << m_n << " at " << this << " - v.dyn at " << m_v << endl;
  }
  ~Vector () {
    cout << "-- obj size " << m_n << " at " << this << " - v.dyn at " << m_v << endl;
    delete m_v;
  }
  Vector &operator=(const Vector& v) { // you can try to remove by reference return value
    cout << "== operator= call with address " << this << " and " << &v << endl;
    if ( this != &v ) {
      cout << "     clean dynamic vector at " << m_v << endl;
      delete m_v;
      m_v = new int[ m_n = v.m_n ];
      cout << "       new dynamic vector at " << m_v << endl;
      for (int i=0; i<m_n; i++) m_v[i] = v.m_v[i];
    }
    else cout << "        nothing to do .... " << endl;
    return *this;
  }
};
int main () {
  Vector a(5), b(3), c(4);
  cout << "** assignment a = b" << endl;          a = b;
  cout << "** assignment c = c" << endl;          c = c;
  cout << "** assignment a = b = c" << endl;      a = b = c;
  return 0;
}
```

Canonical form

If a class has pointer members to dynamic memory, it requires the following 4 member functions:

- constructor, with dynamic allocation
- destructor, with memory freeing
- copy constructor
- assignment operator

It leads to:

```
class T {
public:
  T(...);                       /// constructor other than by copy
  T(const T&);                  // copy constructor (recommended form)
                                ///     may be private to forbid it
  ~T();                         /// destructor
  T& operator=(const T&);       // assignment (recommended form),
                                ///     may be private to forbid it
};
```

- Use `const` whenever it is possible
- Return value for assignment operator
- Take care about by value argument, that hides a copy!

How to access `Vector`'s element?

- Specific methods `set()`, `get()`?
- No: overload the operator []
- Difficulty: make a `lvalue`, so return a reference ...
- C++ constraint: necessarily a member function
- Hence, we have the prototype `type& operator[](int)`

```
#include <iostream>
using namespace std;
class Vector {
  int  m_n;
  int *m_v;
public:
  Vector (int n) { m_v = new int[ m_n = n ]; }
  ~Vector () { delete m_v; }
  int & operator [] ( int idx ) { return m_v[ idx ]; } ///misses overflow control
};
int main () {
  Vector a(3), b(3), c(4);
  for (int i=0; i<3; i++) { a[i] = i; b[i] = 2*(1+i); } // 2 lvalues!
  for (int i=0; i<3; i++) c[i] = a[i] + b[i];            // lvalue and 2 rvalues
  for (int i=0; i<3; i++) cout << c[i] << " ";           // only rvalue
  cout << endl;
  return 0;
}
// display: 2 5 8
```

Some remarks

1. Operand transmission generally by reference
   (not here, while small size)
2. C++ forbids operator [] as a friend function
   - Not recommended for operator that may modify object
3. Only constant methods are usable with constant instances
   - Make constant the operator will forbid its usage as `lvalue`
   - Solution: second operator, constant ... with by value return

   ```
   int Vector::operator[] ( int idx ) const {
     return m_v[i]; // by value return
   }
   ```

4. Semantics of operator is chosen by programmer
   - We may have used (), or <, or comma ...
   - Common sense! Must stay usable by everyone

Parentheses overloading

Allows to define Functors: function objects ... usable as function

Utility

1. Allows prior initialization for some functions
   - *e.g.* a Gaussian: mean and standard deviation
2. Callback functions, *i.e.* sent as argument to another function

Example

```
class GaussianFunction {
  double m_mu, m_sigma;
public:
  GaussianFunction ( double mu, float sigma ) { ... };        // function parametrization
  double operator() (double x) { return exp(-0.5*pow((x-m_mu)/m_sigma,2.0))...; } //apply
};
```

Such an object can be instanciated:

```
GaussianFunction fct ( 2.5, 0.55 ); // creates ``function'' object
```

Application:

```
double result = fct( 3.5 ); // returns the result of its application
```

Principles

Preliminary remarks

- In all: 4 operators (with the 2 array versions)
- May redefine `new` and `delete` specifically for a given class (in as many classes as needed)
- But also at global level, for fundamental types and by default objects
- Define `new` and `delete` do not redefine array version

Overloading for a given class

Operator `new`: `void* operator new(size_t)`

- In member function (but not friend) with argument `size_t`, giving the required size (in bytes)
- Returns type `void*`, pointing to allocated memory

Operator `delete` `void operator delete(void*)`

- Receive as argument the previously allocated address by `new`

Example

```
#include <iostream>
#include <cstddef> // define size_t
using namespace std;
class Point {
  static int m_npt, m_npt_dyn; /// total number of instance, + those dynamically allocated
  int m_x, m_y;
public:
  Point (int a=0, int o=0) { m_x=a; m_y=o; ++m_npt;
                             cout<<"++ total number of points: "<<m_npt<<endl; }
  ~Point () { --m_npt; cout << "-- total number of points: " << m_npt << endl; }
  void * operator new (size_t sz) {
      ++ m_npt_dyn; cout << "++> there are " << m_npt_dyn << " dynamic points " << endl;
      return ::new char[sz]; }  // ACCESS TO GLOBAL OPERATOR
  void operator delete (void* dp) {
      -- m_npt_dyn; cout << "<-- there are " << m_npt_dyn << " dynamic points " << endl;
      ::delete (char*)dp;    }  // ACCESS TO GLOBAL OPERATOR
};
int Point::m_npt     = 0; /// static member initialization
int Point::m_npt_dyn = 0; /// static member initialization
int main () {  Point a(3, 5);                 // ++ total number of points: 1
               Point *ad1 = new Point(1, 3); // ++> there are 1 dynamic points
                                              // ++ total number of points: 2
               Point b;                       // ++ total number of points: 3
               Point *ad2 = new Point(2, 0); // ++> there are 2 dynamic points
                                              // ++ total number of points: 4
               delete ad1;                    // -- total number of points: 3
                                              // <-- there are 1 dynamic points
               Point c(2);                    // ++ total number of points: 4
               delete ad2;                    // -- total number of points: 3
                                              // <-- there are 0 dynamic points
               return 0;                      // -- total number of points: 2, then 1 then 0
}
```

Supplementary information

Remarks

1. Overloading acts only on dynamically allocated object
2. Constructor and destructor are logically called after/before
3. In our example: `Point*ad=new Point[10];` should not imply overloaded operator

Overloading new[] and delete[]

With good prototypes (warning: size is the array one)

```
void*operator new[] (size_t sz) {
  .....
  return ::new char[sz];
}
```

```
void operator delete[] (void *dp) {
  .....
  ::delete (char*)dp;
}
```

Overloading global operators

- Using independent functions, with same prototypes
- take care to not start an ∞ iterative process
  (at least, use `malloc()` and `free()`)

Principe

Reminder:

- Explicit or implicit cast
- Constructor allows cast to class, *e.g.* `Point(int abscissa);`
- Cast from one class to other: operator (*e.g.* `Point and Complex`)

Syntax of cast operator

Basic rules:

- Always must be defined as member function
- Do not write the type of the return value

Example:

```
class Point {
  int m_x, m_y;
public:
  operator int () { return m_x; } // no explicit return value!
  ....
};
```

Cast examples

```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;
    cout<<"++ creates Point("<<a<<","<<o<<")"<<endl;
  }
  Point (const Point &p) {       /// by-copy constructor
    m_x = p.m_x;   m_y = p.m_y;   //  no more used since cast
    cout<<":: by-copy constructor call"<<endl;
  }
  operator int() {
    cout <<"== call int() for Point("<<m_x<<","<<m_y<<")"<<endl;
    return m_x;
  }
};
void fct (int n) { /// any function
  cout << "** call fct("<<n<<")"<<endl;
}
int main () {
  Point a(3,4);
  fct (6);           // normal call
  fct (int(a));      // explicit cast call
  fct (a);           // implicit call, copy is not used
  int n = a;         // another implicit call
  float x = 2.*a;    // another implicit call, plus cast to double
  a = 12;            // implicit call, call to constructor Point(int a, 0)
  return 0;
}
```

Choosing between constructor or assignment operator

```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;
    cout << "++ creates Point " << m_x << " " << m_y << " at " << this << endl;
  }
  Point &operator=(const Point& p) {
    m_x = p.m_x;    m_y=p.m_y;
    cout << "== assignment Point->Point from " << &p << " at "<< this << endl;
    return *this;
  }
  Point &operator=(const int n) {
    m_x = n;    m_y = 0;
    cout << "== assignment int->Point from " << n << " at "<< this << endl;
    return *this;
  }
};
int main () {
  Point a(3, 4); // ++ creates Point 3 4 at 0x7fff5fbff420
  a = 12;        // == assignment int->Point from 12 at 0x7fff5fbff420
  return 0;      // RULE: no cast here, it is assignment
}
```

The UDC are used only when necessary

Enhance an operator meaning

Factorize many operators into one

```
#include <iostream>
using namespace std;
class Point {
  int m_x, m_y;
public:
  Point (const int a=0, const int o=0) { /// Constructor 0, 1 or 2 arguments
    m_x = a; m_y = o;   cout << "** creates Point " << m_x << " " << m_y << endl;
  }
  friend Point operator+(const Point&, const Point&);
  void print() const { cout << "Point("<<m_x<<","<<m_y<<")"; }
};
Point operator+(const Point& a, const Point& b) {
  cout << "++ adds ";        a.print();
  cout << " and    ";        b.print();
  cout << endl;
  return Point(a.m_x+b.m_x, a.m_y+b.m_y);
}
int main () {
  Point a;         // ** creates Point 0 0
  Point b(9,4);    // ** creates Point 9 4
  a = b+5;         // ** creates Point 5 0
                   // ++ adds Point(9,4) and Point(5,0)
                   // ** creates Point 14 4
  a = 2+b;         // ** creates Point 2 0
                   // ++ adds Point(2,0) and Point(9,4)
                   // ** creates Point 11 4
  a = 2+2;         // ** creates Point 4 0
  return 0;
}
```

Warning: does not work with degrading cast ... (e.g. `double` to `int`)

Forbids implicit cast at constructor level

Planned in C++ norm, with keyword `explicit`

Example:

```
#include <iostream>
using namespace std;
class Point {
  int m_x, m_y;
public:
  explicit Point (const int a=0, const int o=0) { // no authorized implicit cast
    m_x = a; m_y = o;
    cout << "**_creates_Point_" << m_x << "_" << m_y << endl;
  }
  friend Point operator+(const Point&, const Point&);
  void print() const { cout << "Point("<<m_x<<","<<m_y<<")"; }
};
Point operator+(const Point& a, const Point& b) {
  return Point(a.m_x+b.m_x, a.m_y+b.m_y);
}
int main () {
  Point a = 12;    // illegal, but "Point a;" will work
  Point b(9, 4);
  a = 2+b;         // illegal, implicit cast forbidden by "explicit" keyword
  a = Point(3);    // OK, explicit cast by constructor usage
  a = b+Point(5); // idem
  return 0;
}
```

Casting one class to another one

```
#include <iostream>
using namespace std;
class Vector2d; /// geometrical meaning ; needed declaration
class Point2d {  /// idem
  int m_x, m_y;
public:
  Point2d (const int a=0, const int o=0) { m_x = a; m_y = o; }
  operator Vector2d();
};
class Vector2d {
  double m_x, m_y;
public:
  Vector2d (const double x=0, const double y=0) { m_x = x; m_y = y; }
  friend Point2d::operator Vector2d ();
  void print() const { cout<<"Vector2d("<<m_x<<","<<m_y<<")"<<endl; }
};
Point2d::operator Vector2d() {
  Vector2d res( m_x, m_y );
  cout<<"cast Point2d("<<m_x<<","<<m_y<<") to Vector2d("<<res.m_x<<","<<res.m_y<<")"<<endl;
  return res;
}
int main () {
  Point2d a(2,5);
  Vector2d v = (Vector2d)a;  v.print(); // explicit cast: cast Point2d(2,5) to Vector2d(2,5)
  Point2d b(9, 12);
  v = b;                     v.print(); // implicit cast: cast Point2d(9,12) to Vector2d(9,1
  return 0;
}
```

By constructor cast

```
#include <iostream>
using namespace std;
class Point2d; /// to declare constructor "Vector2d(const Point2d&);"
class Vector2d {
  double m_x, m_y;
public:
  Vector2d (const double x=0, const double y=0) { m_x = x; m_y = y; }
  Vector2d (const Point2d &p);
  void print() const { cout<<"Vector2d("<<m_x<<","<<m_y<<")"<<endl; }
};
class Point2d {  /// idem
  int m_x, m_y;
public:
  Point2d (const int a=0, const int o=0) { m_x = a; m_y = o; }
  friend Vector2d::Vector2d (const Point2d&); // grant access to private members
};                                            /// needs class members knowledge
Vector2d::Vector2d(const Point2d &p) {
  m_x = p.m_x;      m_y = p.m_y;
}
int main () {
  Point2d a(3,5);
  Vector2d v(a);  // cast done by constructor
  v.print();    // Vector2d(3,5)
  return 0;
}
```

Limitation:

cannot create both constructor $A \to B$ and cast operator $A \to B$

Give meaning to another class operator

Example: cast from `Point2d` to `Vector2d` and then by operator+

```cpp
#include <iostream>
using namespace std;
class Vector2d;
class Point2d {
  int m_x, m_y;
public:
  Point2d (const int a=0, const int o=0) { m_x = a; m_y = o; }
  inline operator Vector2d () const; // cast Vector2d to Point2d ...
  void print() const { cout<<"Point2d("<<m_x<<","<<m_y<<")"<<endl; }
};
class Vector2d {
  double m_x, m_y;
public:
  Vector2d (const double x=0, const double y=0) { m_x = x; m_y = y; }
  void print() const { cout<<"Vector("<<m_x<<","<<m_y<<")"<<endl; }
  friend Point2d::operator Vector2d() const; // needed: it is a "Point2d" method
  friend Vector2d operator+(const Vector2d&, const Vector2d&);
};
Point2d::operator Vector2d() const { return Vector2d(m_x, m_y); }
Vector2d operator+(const Vector2d&a, const Vector2d&b){
  return Vector2d(a.m_x+b.m_x, a.m_y+b.m_y);
}
int main () {
  Point2d a(3,4), b(7,9), c;
  Vector2d x(3.5,2.8), y;
  y = x + a; y.print(); // should work if ''+'' was member function
  y = a + x; y.print(); // should not work if ''+'' was member function
  y = a + b; y.print(); // should not work if ''+'' was member function
  return 0;
}
```