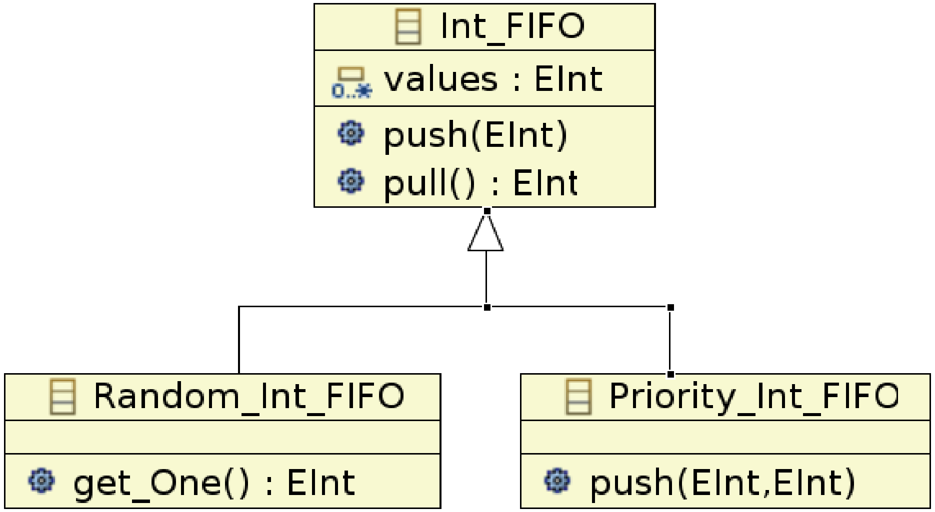
10 – *Object-oriented programming 3*

Julien Deantoni

Outline

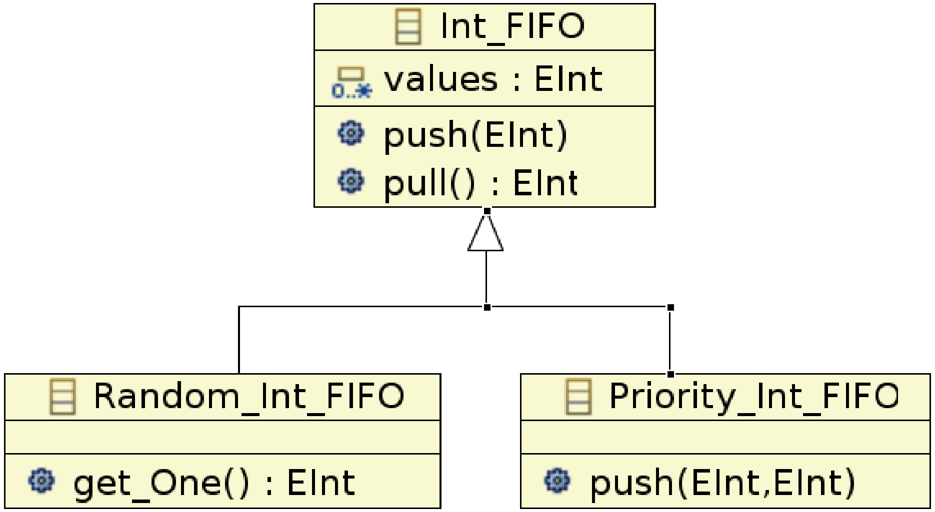
- Derivation public / private
- Derivation and templates
- Dynamic typing and virtual functions:
    - Another example: the **`Expression`** class

Derivation public / private



- Different kinds of FIFO, which contain some integers.
- Different access policies (pull(), get_One())
- Different storage policies

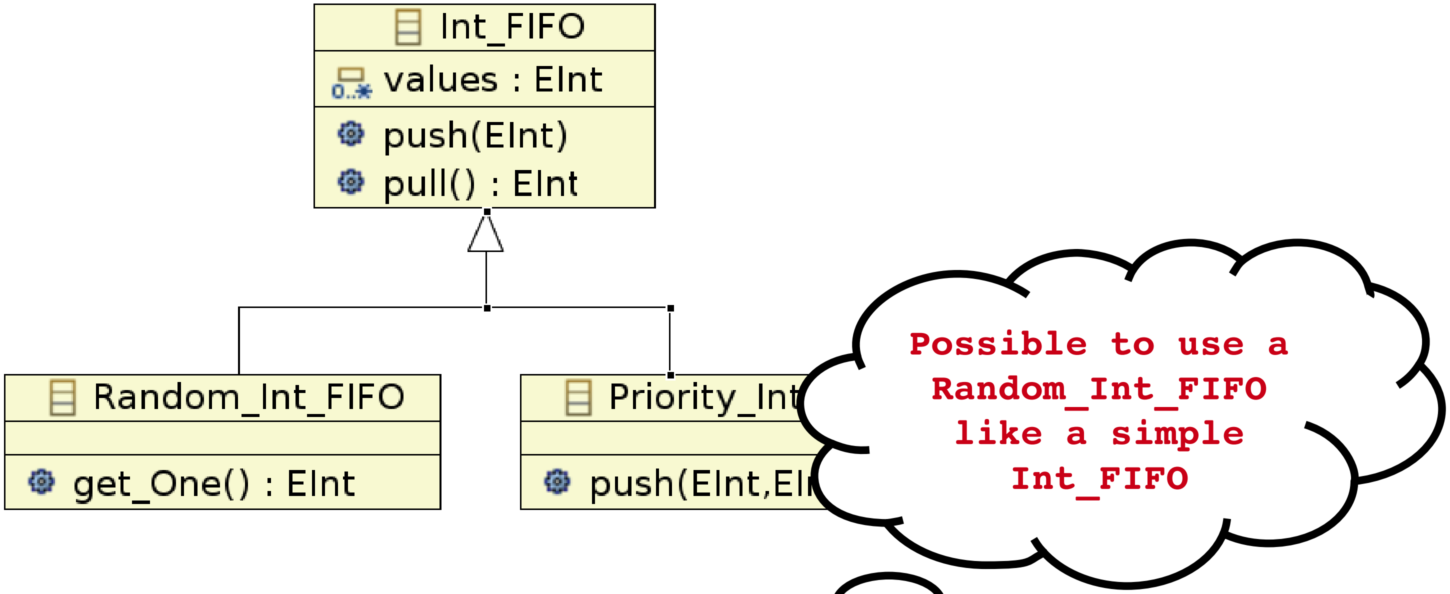
Derivation public / private



- What if we declare Random_Int_FIFO like that ?

```
class Random_Int_FIFO : public Int_FIFO
{
 public:
   int get_One();
}
```

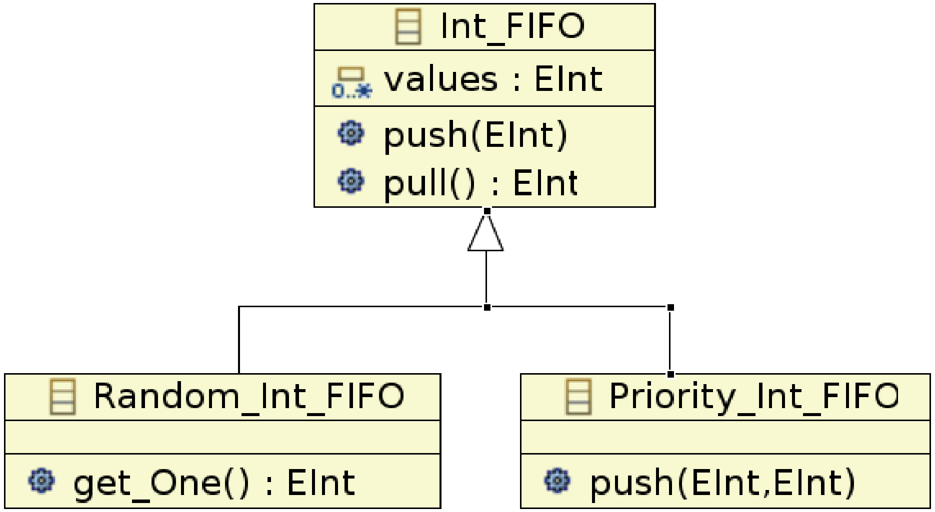
Derivation public / private



- What if we declare Random_Int_FIFO like that ?

```
class Random_Int_FIFO : public Int_FIFO
{
 public:
  int get_One();
}
```

Derivation public / private



- What if we declare Random_Int_FIFO like that ?

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
  int get_One();
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
  int get_One();
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...

No more possible to *push()* integers in the FIFO

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
  int get_One();
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...
  - But some parts of the interface can be set public again

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
  int get_One();
  using Int_FIFO::push;
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...
  - But some parts of the interface can be set public again

All member-function(s) named push are now public

```
class Random_Int_F : private Int_FIFO
{
 public:
  int get_One();
  using Int_FIFO::push;
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...
  - But some parts of the interface can be set public again

→ **private derivation is not a “is a” relation anymore !**

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
  int get_One();
  using Int_FIFO::push;
}
```

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...
  - But some parts of the interface can be set public again

→ **private derivation is not a “is a” relation anymore !**

→ **private derivation is closer to a “has a” relation.**

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
  int get_One();
  using Int_FIFO::push;
}
```

KAIROS

Derivation public / private

- The private derivation
  - All members of derived class become private
  - The “interface” of the derived class is lost...
  - But some parts of the interface can be set public again

→ **private derivation is not a “is a” relation anymore !**

→ **private derivation is closer to a “has a” relation.**

→ **Private inheritance means “is implemented in terms of”. It's usually inferior to composition** *[Effective Modern C++. Scott Meyers]*

```
class Random_Int_FIFO : private Int_FIFO
{
 public:
  int get_One();
  using Int_FIFO::push;
}
```

Derivation public / private

- The private derivation

```
class Person {};
class Student:private Person {};      // private
void eat(const Person& p){}          // anyone can eat
void study(const Student& s){}       // only students study

int main()
{
      Person p;      // p is a Person
      Student s;     // s is a Student
      eat(p);        // fine, p is a Person
      eat(s);        // error! s isn't a Person
      return 0;
}
```

Derivation public / private

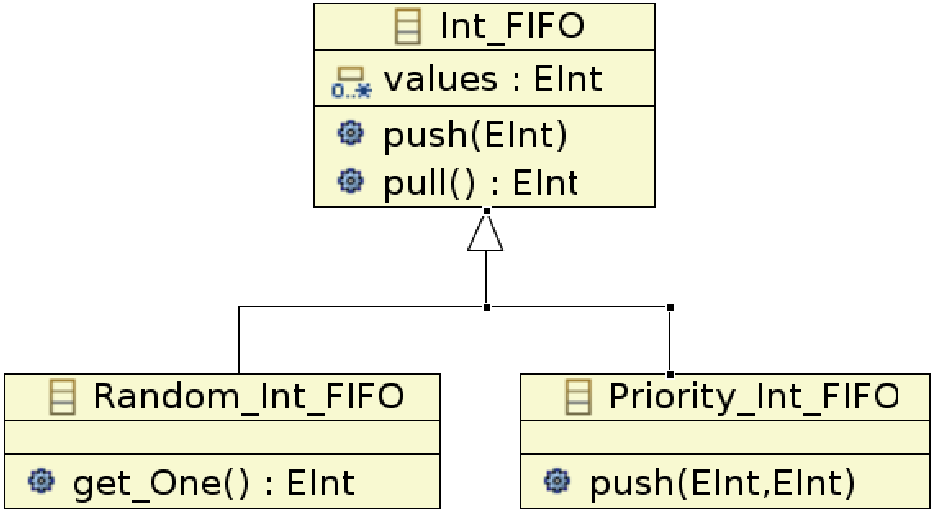
- The private derivation

```
class Person {};
class Student:private Person {};      // private
void eat(const Person& p){}          // anyone can eat
void study(const Student& s){}        // only students study

int main()
{
      Person p;      // p is a Person
      Student s;      // s is a Student
      eat(p);       // fine, p is a Person
      eat(s);       // error! s isn't a Person
      return 0;
}
```

→ in contrast to public inheritance, compilers will generally not convert a derived class object (Student) into a base class object (Person) if the inheritance relationship between the classes is private

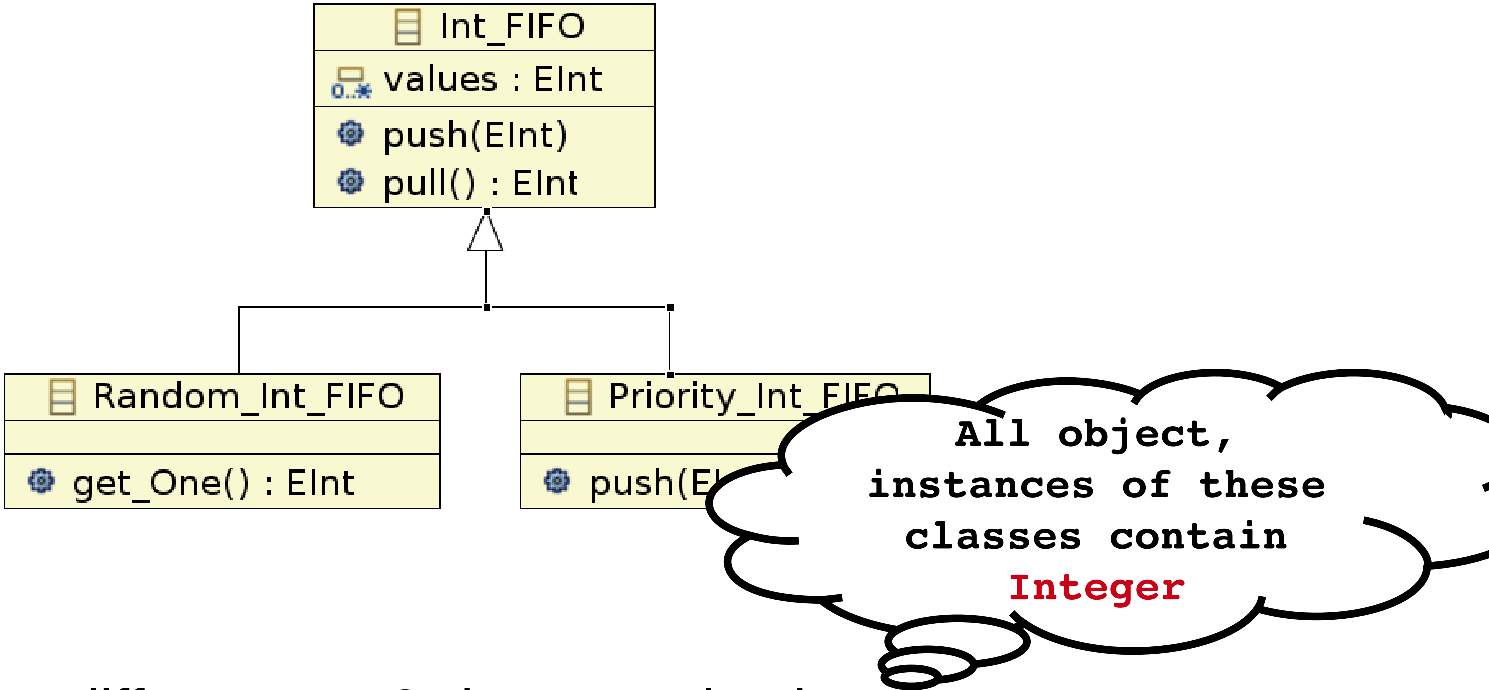
```
main.cpp: In function ‘int main()’:
main.cpp:11:14: error: ‘Person’ is an inaccessible base of ‘Student’
   11 |         eat(s);         // error! s isn't a Person
      |              ^
make: *** [Makefile:40: main.o] Error 1
```

Derivation and Template



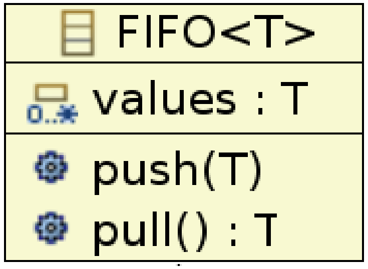
- Derivation
  - We have different FIFO that contains integers
  - Access policies are different
  - Different FIFO still share the internal representation (member attributes) and some members functions

Derivation and Template



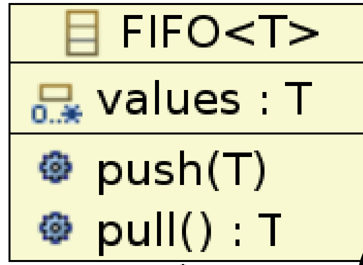
- Derivation
  - We have different FIFO that contains integers
  - Access policies are different
  - Different FIFO still share the internal representation (member attributes) and some members functions

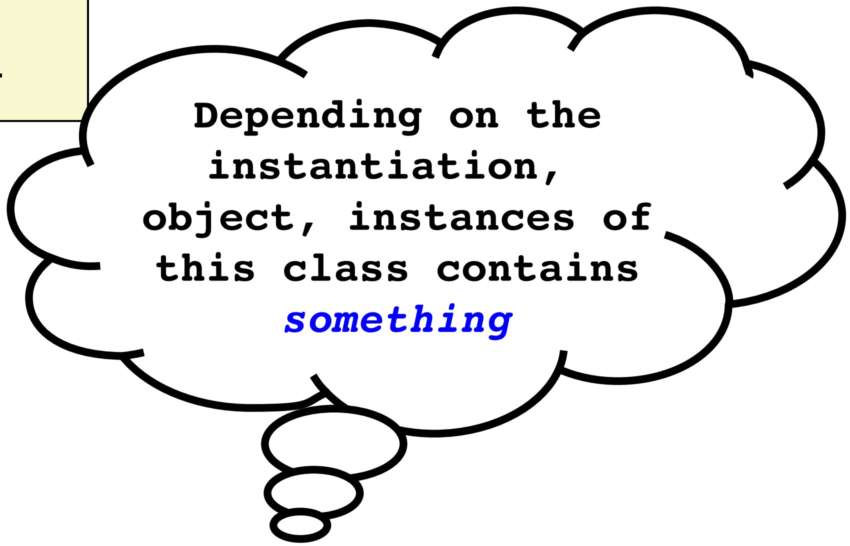
Derivation and Template



- Templates
  - We have one FIFO that contains a non predefined type
  - ~~Access policies are different~~
  - *Different FIFO still share the internal representation (member attributes) and **all** members functions*

Derivation and Template

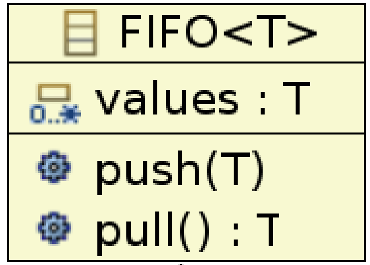




- Templates
  - We have one FIFO that contains a non predefined type
  - ~~Access policies are different~~
  - *Different FIFO still share the internal representation (member attributes) and **all** members functions*

Derivation and Template

```
int main()
{
FIFO<int> fint;

FIFO<char> fchar;

FIFO<FIFO<string> > fcomplex;
}
```

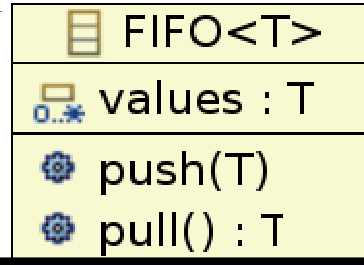


Depending on the instantiation, object, instances of this class contains *something*

- Templates
  - We have one FIFO that contains a non predefined type
  - ~~Access policies are different~~
  - *Different FIFO still share the internal representation (member attributes) and **all** members functions*

Derivation and Template

```
int main()
{
FIFO<int> fint;

FIFO<char> fchar;

FIFO<FIFO<string> > fcomplex;
}
```



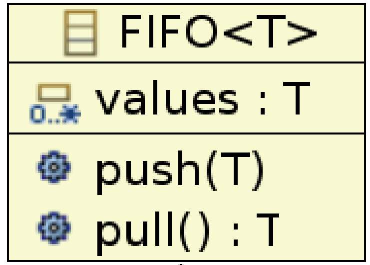
Before c++11 ,
Needs a space to differentiate
from the operator>> symbol

Depending on the
antiation,
instances of
lass contains
something

- Templates
  - We have one FIFO that contains a non predefined type
  - ~~Access policies are different~~
  - *Different FIFO still share the internal representation (member attributes) and **all** members functions*

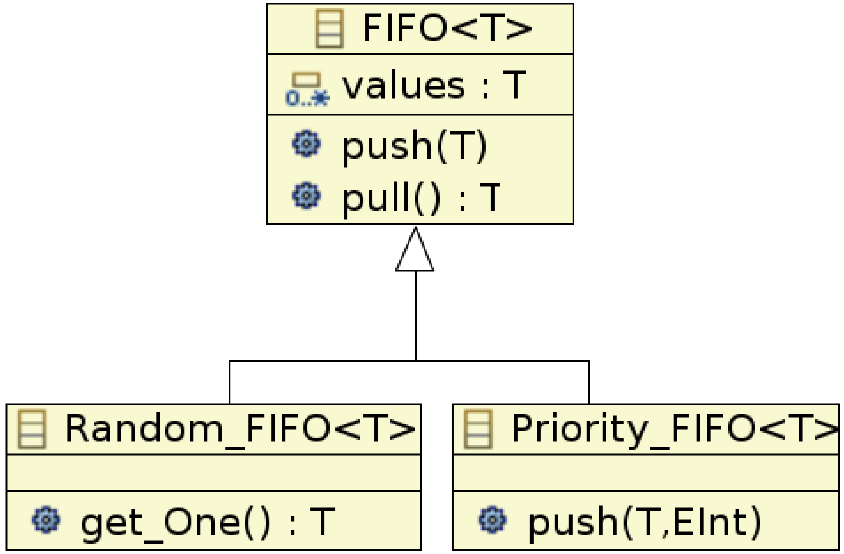
Derivation and Template

```
int main()
{
FIFO<int> fint;

FIFO<char> fchar;

FIFO<FIFO<string> > fcomplex;
}
```



- Templates
  - We have one FIFO that contains a non still predefined type
  - ~~**Access policies are different**~~
    - what if we want different policies ?
  - Different FIFO still share the internal representation (member attributes) and some members functions

Derivation and Template




```
int main()
{
FIFO<int> fint;

Random_FIFO<char> random_fchar;

Priority_FIFO<FIFO<string> > priority_fcomplex;
}
```

Derivation and class templates

- Two compatible mechanisms, with many combinations
  - Both base and derived classes are templates

```
template <typename T> class A {...};
template <typename T> class B : public A<T> {...};
```

  - A specialized version for the previous case

```
class B<int> : public A<int> {...};
```

  - Only the base class is template

```
template <typename T> class A {...};
class B : public A<int> {...};
```

  - Only the derived class is template

```
class A {...};
template <typename T> class B : public A {...};
```

KAIROS

Copy of derived classes

```
class A {...};
class B : public A {...};
B b1, b2 = b1; // initialization (construction)
b1 = b2;     // assignment
```

- Memberwise copy construction
  - If a derived class has a copy constructor, this constructor is entirely responsible for the initialization
  - If a derived class has a copy assignment operator, this operator is entirely responsible for the assignment
- When a class does not define a needed copy operation... C++ uses default copy (see next slide)

KAIROS

Default copy of derived classes (1)

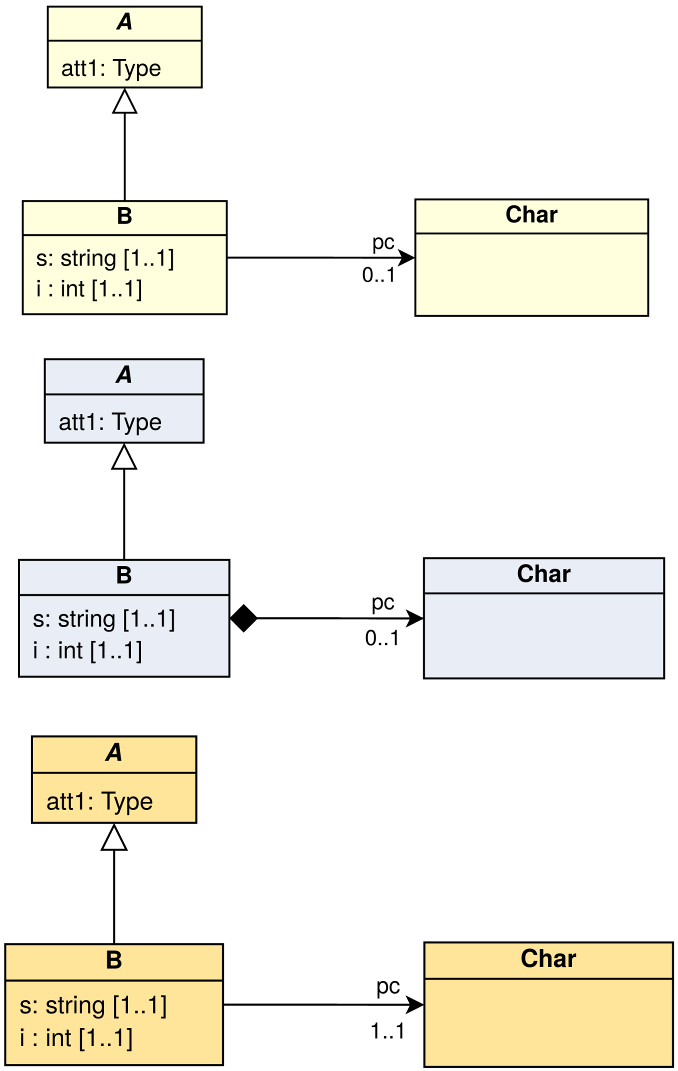
- If a class lacks copy operation(s)
    - The C++ compiler synthesizes the needed copy operation(s) (*default copy constructor*, *default copy assignment operator*)
        - Each member is copied according to its own copy semantics
        - Base class(es) are considered as members during the copy operation
        - The memberwise procedure is applied recursively
        - Built-in types are copied bitwise
    - The synthesis process may fail…

copy of derived classes

```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

copy of derived classes

```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```



Default copy of derived classes

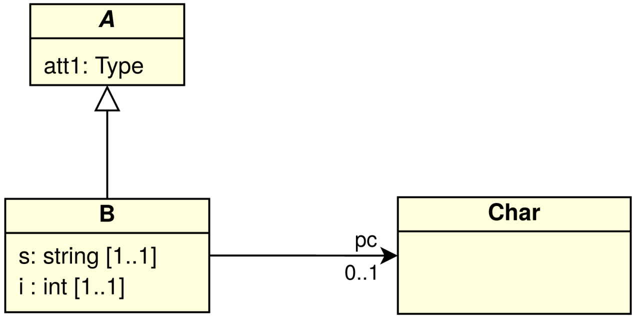
```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

```
B::B(const B& b)
  : A((A&)b),
    i(b.i), pc(b.pc), s(b.s)
{}
```

```
B& B::operator=(const B& b) {
  A::operator=(b); // !!
  i = b.i;
  pc = b.pc;
  s = b.s;
  return *this;
}
```

- Note that `i` *and* `pc` are bitwise copied

<u>**Default**</u> copy of derived classes



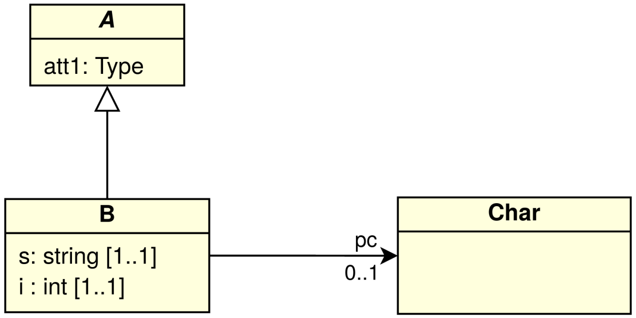

```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

```
B::B(const B& b)
  : A((A&)b),
    i(b.i), pc(b.pc), s(b.s)
{}
```

```
B& B::operator=(const B& b) {
  A::operator=(b); // !!
  i = b.i;
  pc = b.pc;
  s = b.s;
  return *this;
}
```

- Note that `i` *and* `pc` are bitwise copied

Default copy of derived classes



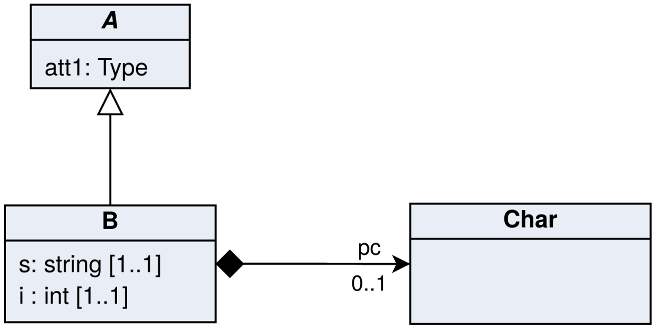

```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

```
B::B(const B& b)
  : A((A&)b),
    i(b.i), pc(b.pc), s(b.s)
{}
```

```
B& B::operator=(const B& b) {
  *(A*)this = (A&)b; // !!
  i = b.i;
  pc = b.pc;
  s = b.s;
  return *this;
}
```

- Note that `i` *and* `pc` are bitwise copied

Redefined copy of derived classes

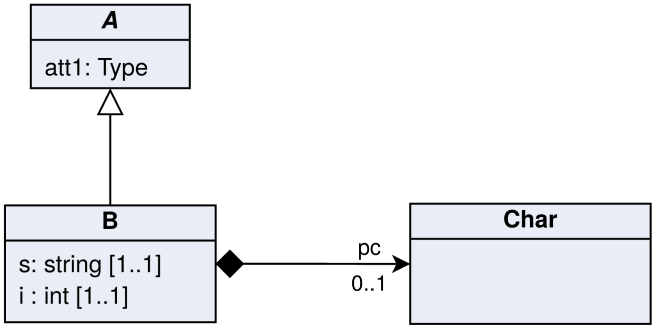



```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

```
B::B(const B& b)
  : A((A&)b),
    i(b.i),
         ??
    s(b.s)
{}
```

```
B& B::operator=(const B& b) {
  *(A*)this = (A&)b; // !!
  i = b.i;
         ??
  s = b.s;
  return *this;
}
```

Redefined copy of derived classes




```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

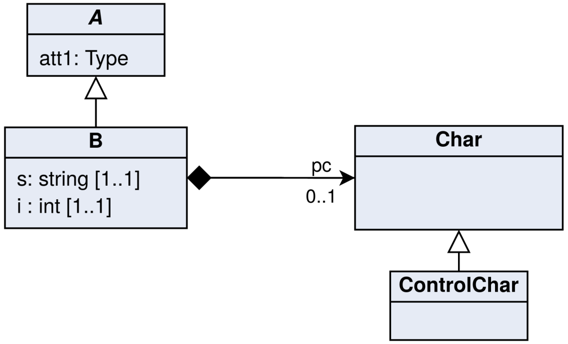
+ destructeur !! et attention aux setter

```
B::B(const B& b)
  : A((A&)b),
    i(b.i),
    pc(new Char(*b.pc)),
    s(b.s)
{}
```

+ test à nullptr

```
B& B::operator=(const B& b) {
  *(A*)this = (A&)b; // !!
  i = b.i;
  delete this->pc ;
  pc = new Char(*b.pc);
  s = b.s;
  return *this;
}
```

<u>**Redefined**</u> copy of derived classes

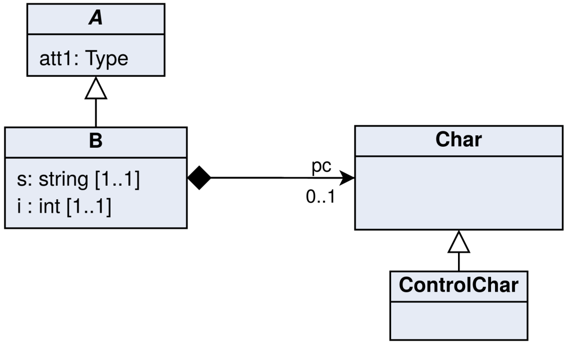



```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

```
B::B(const B& b)
   : A((A&)b),
     i(b.i),
              ??
     s(b.s)
{}
```

```
B& B::operator=(const B& b) {
   *(A*)this = (A&)b; // !!
   i = b.i;
              ??
   s = b.s;
   return *this;
}
```

Redefined copy of derived classes




```
class B : public A {
   int i;
   char* pc;
   string s;
   // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

```
B::B(const B& b)
  : A((A&)b),
    i(b.i),
    pc(b.pc->clone()),
    s(b.s)
{}
```

```
B& B::operator=(const B& b) {
  *(A*)this = (A&)b; // !!
  i = b.i;
  delete this->pc;
  pc = b.pc->clone();
  s = b.s;
  return *this;
}
```

+ destructeur !! et attention aux setter

copy of derived classes : failure cases

- Synthesis failure of default copy operations

```
class A {
private:
 const string _s; // const member
 B& _rb;          // reference data member


};
```

  - The **`const`** member or the reference data member prevent the synthesis of the default copy assignment (but *not* of the default copy constructor)

KAIROS

copy of derived classes : failure cases

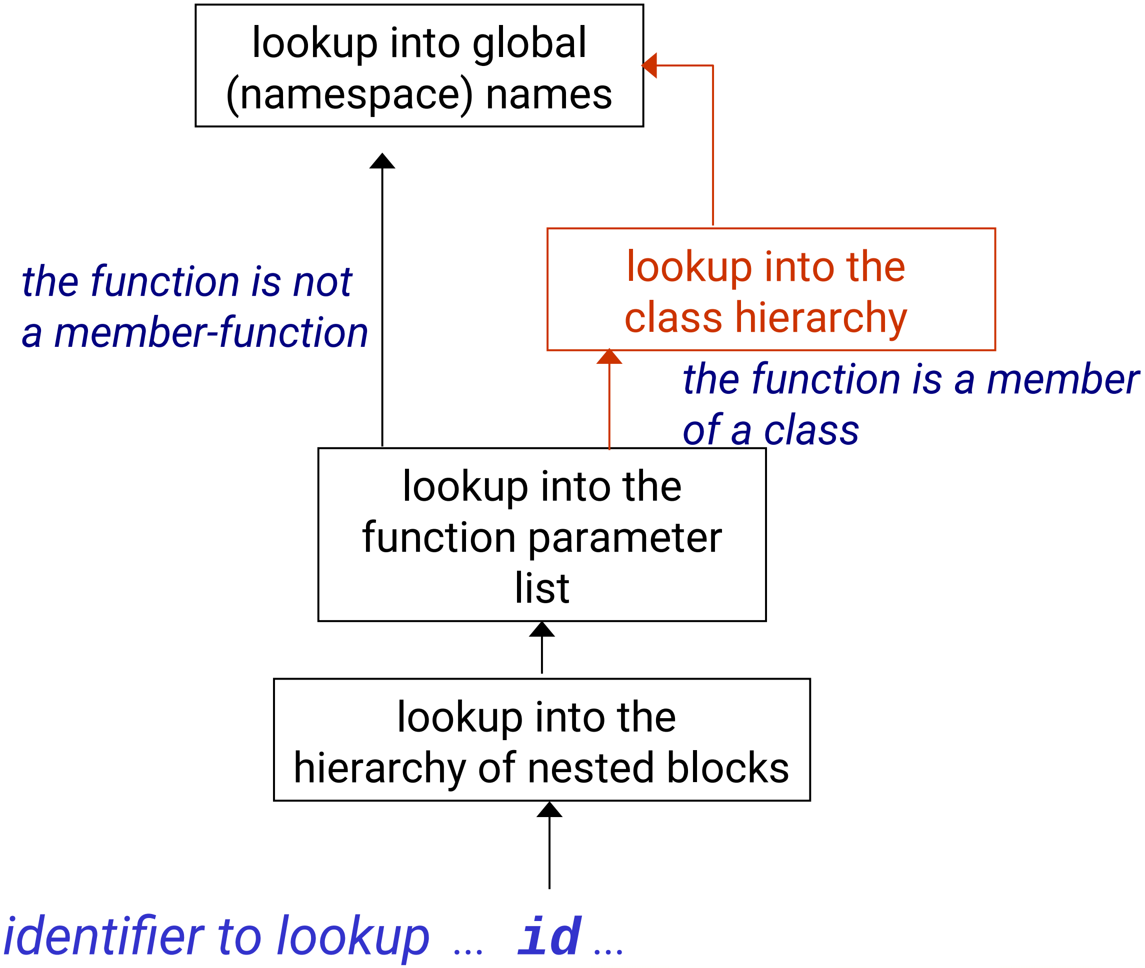
- Synthesis failure of default copy operations

```
class A {
private:
 const string _s; // const member
 B& _rb;          // reference data member
 A(const A&);     // private copy constructor
 A* operator=(const A&) = delete; //remove synthesized operator
};
```

  - The **const** member or the reference data member <u>prevent</u> the synthesis of the default copy assignment (but *not* of the default copy constructor)
  - The private copy constructor prevents the synthesis of the copy constructor for a class that contains an **A** by value
  - Since C++11, one can decide to remove any synthesized function

Name lookup and derived classes

- Name lookup
  - Searching for the right declaration of an identifier
  - Apply scope rules



Name lookup and derived classes (2)

```
class A {
public:
   int i; int j; int n;
};

class B : public A {
private:
   int j;
};

class C : public B {
private:
   int k;
public:
   void f(double);
};
```

```
int i; // global variable


void C::f(double n) {
   k = 0;  // this-> k, C::k

   n = 3.14;// function parameter

   j = 2; // B::j, but not
                // accessible here

   i = 3; // A::i

   i = ::i;  // ::i is global i
}
```

Name lookup and derived classes (2)

```
class A {
public:
   int i; int j; int n;
};

class B : public A {
private:
   int j;
};

class C : public B {
private:
   int k;
public:
   void f(double);
};
```

```
int i; // global variable


void C::f(double k) {
   k = k;  // != this->k=k

   n = 3.14;// function parameter

   j = 2; // B::j, but not
                 // accessible here

   i = 3; // A::i

   i = ::i;  // ::i is global i
}
```

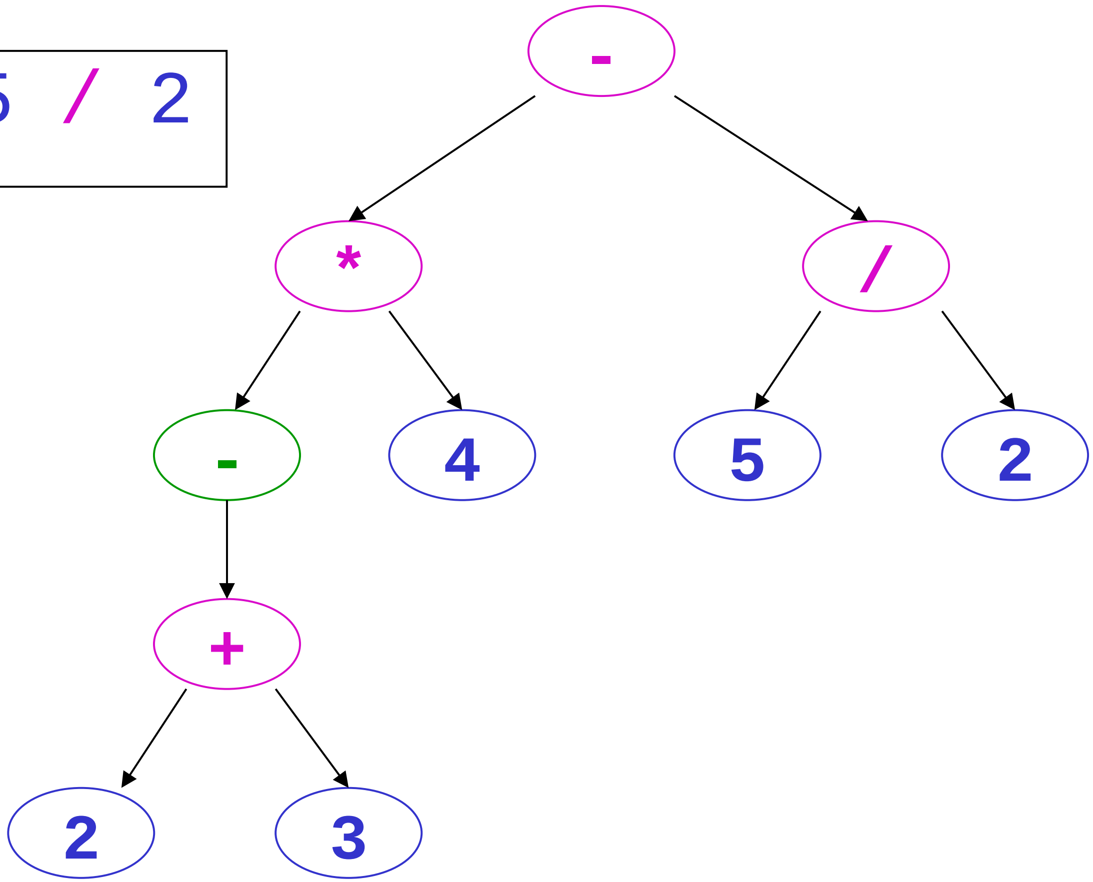
Name lookup and derived classes (2)

```
class A {
public:
   int i; int j; int n;
};

class B : public A {
private:
   int j;
};

class C : public B {
private:
   int k;
public:
   void f(double);
};
```

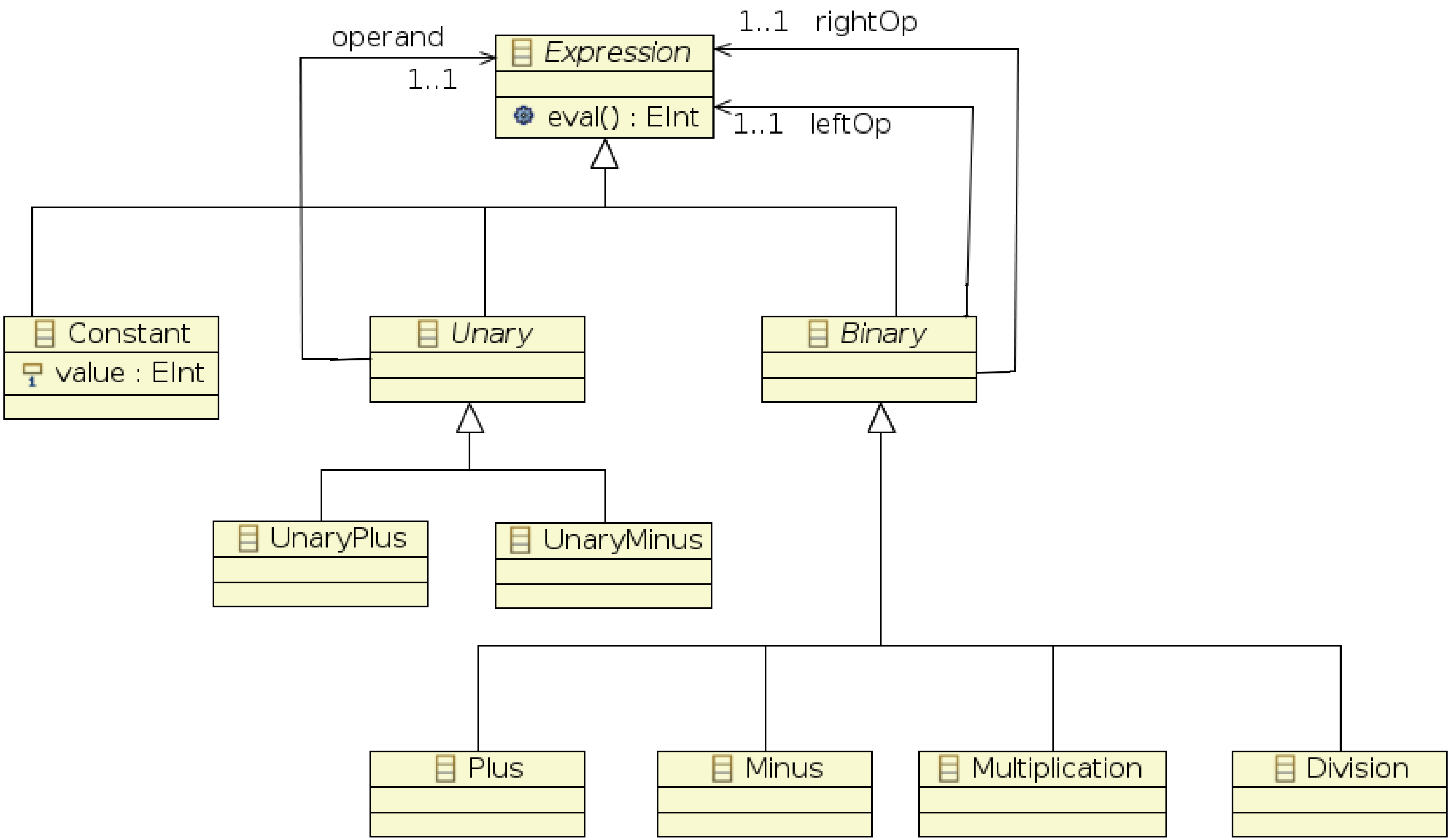
```
int i; // global variable


void C::f(double k) {
   k = k;  // != this->k=k

   n = 3.14;// function parameter

   A::j = 2; //this->A::j

   i = 3; // A::i

   i = ::i;  // ::i is global i
}
```

Class Expr
Arithmetic expressions as trees

```
-(2 + 3) * 4 - 5 / 2
```

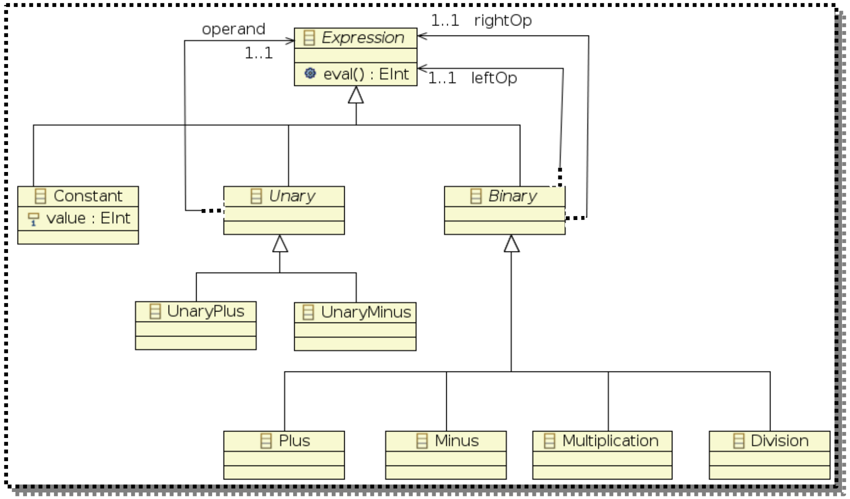


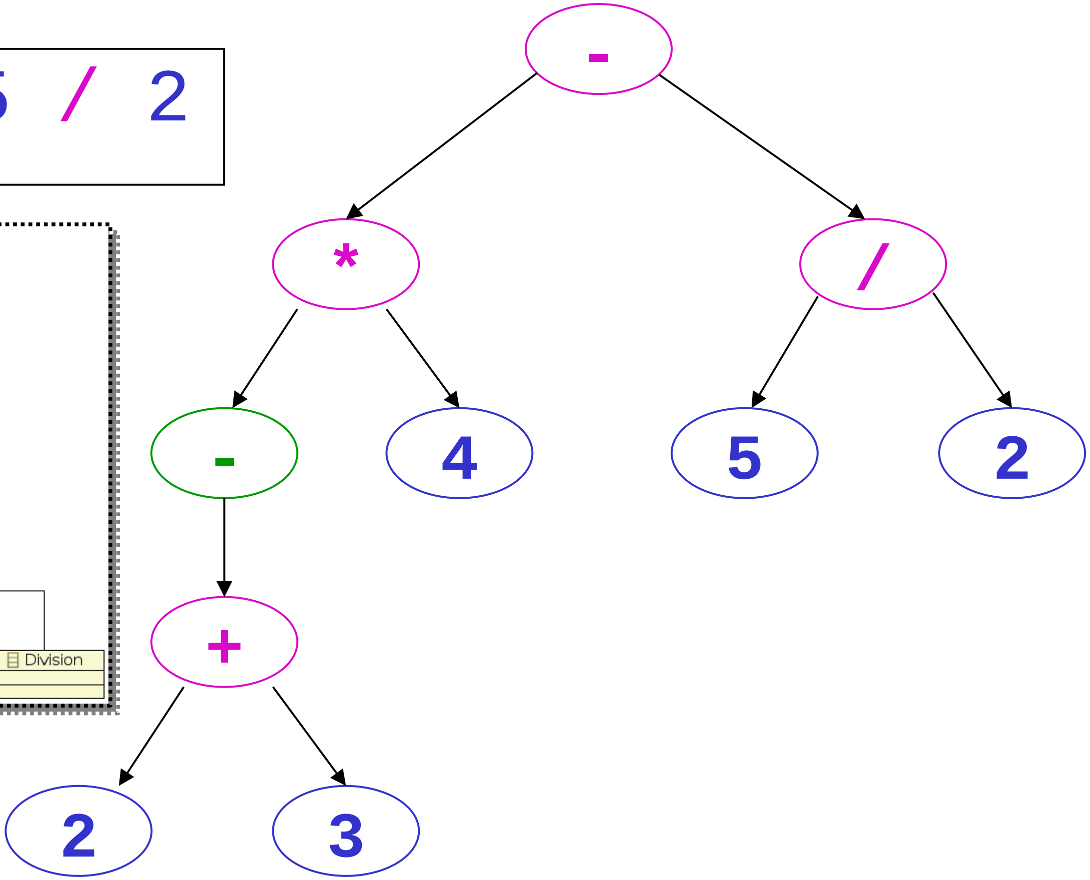
Class `Expression`



Arithmetic expressions as trees

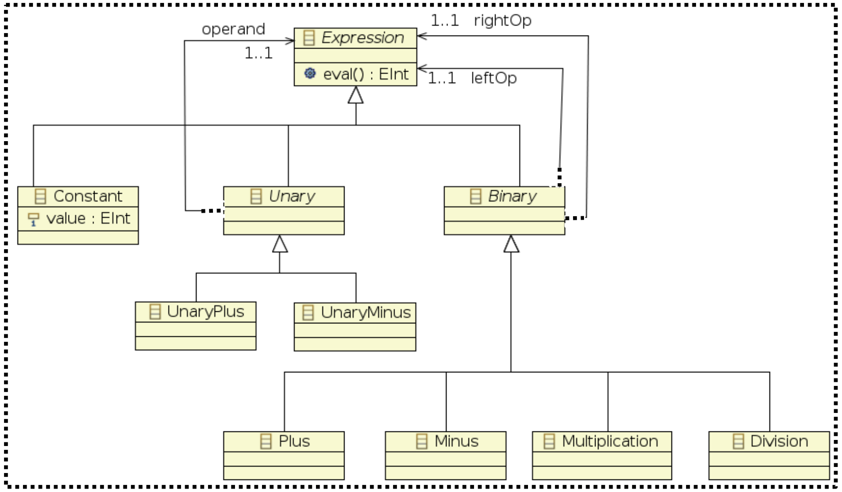
- (2 + 3) * 4 - 5 / 2





Class Expr
Abstract classes

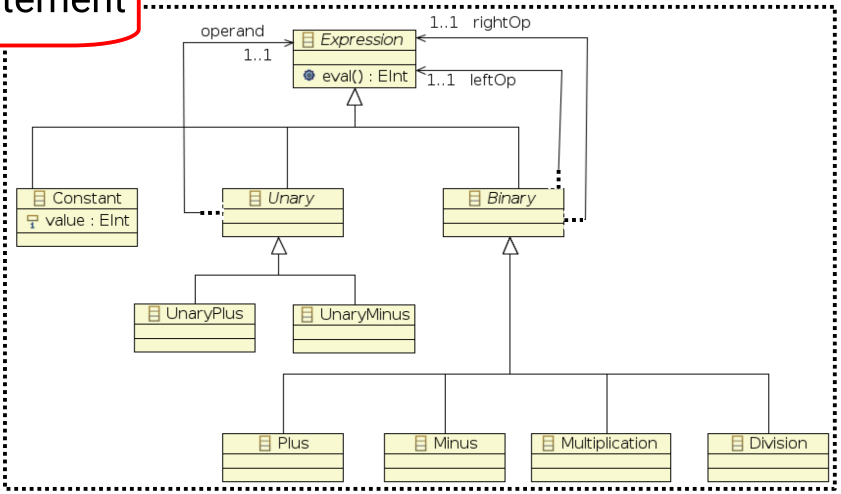
```
class Expr {
public:
    virtual int eval() const = 0; //fonction membre virtuelle pure
                                  --> rend Expr abstraite
};

class Unary : public Expr {
protected:
    Expr &op;
public:
    Unary(Expr& e) : op(e) {}
};

class Binary : public Expr {
protected:
   Expr &left_op, &right_op;
public:
   Binary(Expr& e1, Expr& e2) : left_op(e1), right_op(e2) {}
};
```



Class Expr
Abstract classes

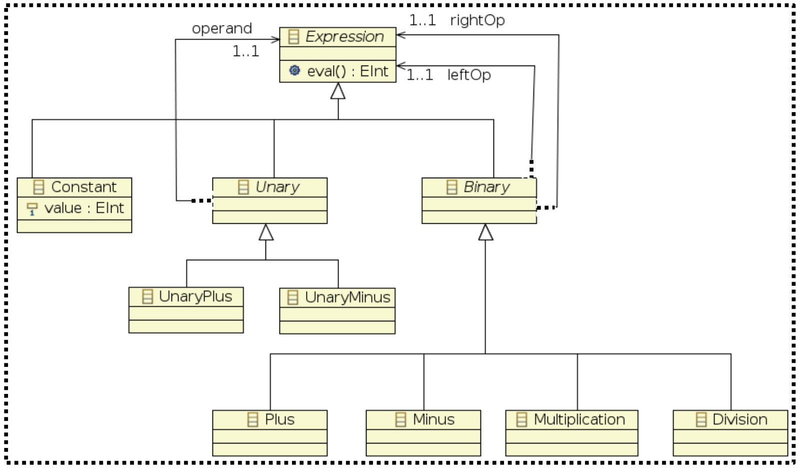
```
class Expr {
public:
    virtual int eval() const = 0; //fonction membre virtuelle pure
                                      --> rend Expr abstraite
};

class Unary : public Expr {
protected:
    Expr &op;
public:
    Unary(Expr& e) :Expr(), op(e) {}
};

class Binary : public Expr {
protected:
   Expr &left_op, &right_op;
public:
   Binary(Expr& e1, Expr& e2) :Expr(),left_op(e1), right_op(e2) {}
};
```

Appelé implicitement



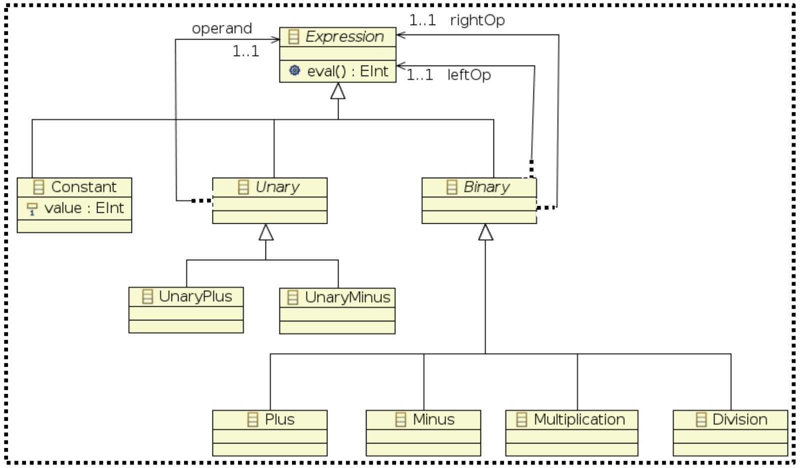
Appelé implicitement

Class Expr
Concrete classes




```
class Constant : public Expr {
private:
    int val;
public:
    Constant(int v) : val(v) {}
    int eval() const {return val;}
};

class UnaryMinus : public Unary {
public:
    UnaryMinus(Expr& e) : Unary(e) {}
    int eval() const {return -op->eval();}
};

class Multiplication : public Binary {
public:
    Multiplication(Expr& e1, Expr& e2) : Binary(e1, e2) {}
    int eval() const {
        return left_op->eval() * right_op->eval();
    }
};
```

Class Expr
Concrete classes




```
class Constant : public Expr {
private:
    int val;
public:
    Constant(int v) : Expr(), val(v) {}
    int eval() const {return val;}
};

class UnaryMinus : public Unary {
public:
    UnaryMinus(Expr& e) : Unary(e) {}
    int eval() const {return -op->eval();}
};

class Multiplication : public Binary {
public:
    Multiplication(Expr& e1, Expr& e2) : Binary(e1, e2) {}
    int eval() const {
        return left_op->eval() * right_op->eval();
    }
};
```

Class Expr
Using virtual functions

```
main()
{

// c1 = 3
Constant c1(3);
// c2 = 5
Constant c2(5);

// umin = -c1 == -3
UnaryMinus umin(c1);
// mult1 = c1*umin== -9
Multiplication mult1(c1, umin);
// min1 = c2 – (c1*umin) = 14
Minus min1(c2,mult1);
```

```
  cout << "c1 = "  << c1.eval()
       << endl;
  cout << "umin = " << umin.eval()
       << endl;
  cout << "mult1 = "  << mult1.eval()
       << endl;
  cout << "min1 = "  << min1.eval()
       << endl;
}
```

```
jdeanton@ziva$./doIt
c1 = 3
umin = -3
mult1 = -9
min1 = 14
```

Class Expr
Using virtual functions

```
main()
{

// c1 = 3
Constant c1(3);
// c2 = 5
Constant c2(5);

// umin = -c1 == -3
Uniminus umin(c1);
// mult1 = c1*umin== -9
Mult mult1(c1, umin);
// min1 = c2 – (c1*umin) = 14
Minus min1(c2,mult1);

Expr anExpr1= mult1;
Expr* anExpr2= &mult1;
Expr& anExpr3= mult1;
```

```
cout << "anExpr1 = "<< anExpr1.eval()<< endl;
cout << "anExpr2 = "<< anExpr2->eval()<< endl;
cout << "anExpr3 = "<< anExpr3.eval()<< endl;
}
```

```
jdeanton@ziva$./doIt
AnExpr1 =
anExpr2 =
anExpr3 =
```

Class Expr
Using virtual functions

```
main()
{

// c1 = 3
Constant c1(3);
// c2 = 5
Constant c2(5);

// umin = -c1 == -3
Uniminus umin(c1);
// mult1 = c1*umin== -9
Mult mult1(c1, umin);
// min1 = c2 – (c1*umin) = 14
Minus min1(c2,mult1);

Expr anExpr1= mult1;
Expr* anExpr2= &mult1;
Expr& anExpr3= mult1;
```

```
cout << "anExpr1 = "<< anExpr1.eval()<< endl;
cout << "anExpr2 = "<< anExpr2->eval()<< endl;
cout << "anExpr3 = "<< anExpr3.eval()<< endl;
}
```

```
jdeanton@ziva$./doIt
AnExpr1 = Ne compile même pas !!
anExpr2 = -9
anExpr1 = -9
```

Class Expr
Virtual function resolution (1)

- Static (compile-time) resolution is used instead of dynamic typing when
  - the virtual function is invoked through an instance

```
Uniminus u(e);
n = u.eval(); // Uniminus::eval
```

  - the version needed is explicited using the scope operator

```
class A {
public:
   virtual void f() {...}
};
class B : public A {
public:
   virtual void f() {
      A::f();
   }
};
```

  - the virtual function is invoked within a base class constructor or destructor…

Virtual function resolution (2)

- Calling a virtual function from a constructor or destructor

```
class A {
public:
    virtual void f() {
            // ...
    }
    A() {
        f(); // calls A::f
    }
};
```

```
class B : public A {
    int* _p;
public:
    virtual void f() {
        *_p = 10;
    }

    B() : A(), _p(new int(0)) {}
};
```

If `B::f` were called from A constructor, the program would crash since the pointer _p has not yet been initialized

Function that can be virtual

- Only member-functions (or member-operators) can be virtual; friends cannot
- **There is nothing such as virtual constructors**
- The destructor may be virtual
  (and generally is for abstract classes)

```
class Expr {
  virtual int eval() const = 0;
  virtual ~Expr() {};
};
class Unary : public Expr {
  ~Unary() {}
};
class Binary : public Expr {
  ~Binary() {}
};
```

You may have a look here: https://stackoverflow.com/questions/2198379/are-virtual-destructors-inherited

Function that can be virtual

- Only member-functions (or member-operators) can be virtual; friends cannot
- **There is nothing such as virtual constructors**
- The destructor may be virtual

  (and generally is for abstract classes)

```
class Expr {
    virtual int eval() const = 0;
    virtual ~Expr() = default;
  };
  class Unary : public Expr {
    virtual ~Unary() = default;
  };
  class Binary : public Expr {
    virtual ~Binary() = default;
};
```

You may have a look here: https://stackoverflow.com/questions/2198379/are-virtual-destructors-inherited

Questions ?