Implémentation du C++

Héritage trivial en mémoire

```
class Mere {
   int u, v;
public:
  void toto() { std::cout<<"MereToto\n"; }
  void tutu() { std::cout<<"MereTutu\n"; }
};
class Fille: public Mere {
   int x, y;
public:
  void toto() { std::cout<<"FilleToto\n"; }
  void tutu() { std::cout<<"FilleTutu\n"; }
};
int main() {
   Mere a;
   Fille b;
   Mere a2 = Fille();
 //Fille b2 = static_cast<Fille>(a2); // interdit
   Mere* a3 = new Fille();

   printf("sizeof: a=%d; b=%d; a2=%d; a3=%d\n",
          sizeof(a), sizeof(b),  sizeof(a2), sizeof(a3));

   a.toto();
   b.toto();
   a2.toto();
   a3->toto();
   return 0;
}
```

Héritage polymorphique en mémoire

```
class Mere {
   int u, v;
 public:
   virtual void toto() { std::cout<<"MereToto\n"; }
   virtual void tutu() { std::cout<<"MereTutu\n"; }
};
class Fille: public Mere {
   int x, y;
 public:
   void toto() override { std::cout<<"FilleToto\n"; }
   void tutu() override { std::cout<<"FilleTutu\n"; }
};
int main() {
   Mere a;
   Fille b;
   Mere a2 = Fille();
   Mere* a3 = new Fille();

   a.toto();
   b.toto();
   a2.toto();
   a3->toto();

   std::cout << sizeof(a) << "\n";
   std::cout << sizeof(b) << "\n";
   std::cout << sizeof(a2)<< "\n";
   std::cout << sizeof(a3)<< "\n";
   return 0;
}
```

Exemples de décoration de nom

```
void h(int)        _Z1hi
void h(int, char)  _Z1hic
void h(void)       _Z1hv
```

```
int Something::Inside::Deeper::deeperMethod(void) _ZN9Something6Inside6Deeper10deeperMethodEv
```

web.mit.edu/tibbetts/Public/inside-c/www/mangling.html

Redéfinition et variance

```
struct A {
  virtual void boom()  {std::cout << "A::boom\n";}
};
struct B : public A {
  void boom() override {std::cout << "B::boom\n";}
};

struct Up {
   virtual A bidule()  { return A();     }
   virtual A* machin() { return new A(); }
   virtual void truc(A a)   { a.boom();  }
   virtual void chose(A* a) { a->boom(); }
   virtual void chouette(A a) { a.boom();  }
   virtual void muche(A* a)   { a->boom(); }
};
struct Down : public Up {
// CERTAINES DES LIGNES SUIVANTES SONT INVALIDES
   B bidule() override  { return B(); }

   B* machin() override { return new B(); }

   void truc(B b) override { b.boom();  }

   void chose(B* b) override { b->boom(); }

   void chouette(B b) { b.boom();  }

   void muche(B* b) { b->boom(); }

};
int main()
{  Up up; Down down; A a;  B b;

   up  .bidule().boom();
   down.bidule().boom();
   up  .machin()->boom();
   down.machin()->boom();

   up.truc(a);
   up.truc(b);
   up.chose(&a);
   up.chose(&b);

   down.truc(a);
   down.truc(b);
   down.chose(&a);
   down.chose(&b);
   down.chouette(a);
   down.chouette(b);
   down.muche(&a);
   down.muche(&b);
   return 0;
}
```

Templates et variance

```
struct Vehicle {};
struct Car : Vehicle {};

int main(){
  std::vector<Vehicle *> vehicles;
  std::vector<Car *> cars;

  vehicles = cars; // ERROR
}
```

Surcharge, redéfinition et variance

`https://www.imt-atlantique.fr/fr/personne/antoine-beugnard` (onglet LOO)

```
class Top {};
class Middle : public Top {};
class Bottom : public Middle {};

struct Up {
   virtual std::string cov(Top* t)       {
      return "Up";
   }
   virtual std::string inv(Middle* m)    {
      return "Up";
   }
   virtual std::string contra(Bottom* m) {
      return "Up";
   }
};
struct Down : public Up  {
   std::string cov(Middle* t)    {
      return "Down";
   }
   std::string inv(Middle* m)    {
      return "Down";
   }
   std::string contra(Middle* m) {
      return "Down";
   }
};
int main() {
   Up* u = new Up();
   Down* d = new Down();
   Up* ud = new Down();
   Top* top = new Top();
   Middle* mid = new Middle();
   Bottom* bot = new Bottom();
```

	u.?(?)	**d**.?(?)	**ud**.?(?)
?.cov(top)			
?.cov(mid)			
?.cov(bot)			
?.inv(top)			
?.inv(mid)			
?.inv(bot)			
?.contra(top)			
?.contra(mid)			
?.contra(bot)			

Gestion mémoire automatique

Code sans RAII

```
void foo(int n) {
  char * p = new char[n];   // <----
  do_stuff(p);
  do_some_more_stuff();
  /* etc */
  delete[] p;               // <----
}
```

Code avec RAII

```
class char_buffer {
private:
  char * p_;
public:
  char_buffer(int n) : p_(new char[n]) {}
  ~char_buffer() { delete[] p_; }

  operator char* () const { return p_; }
  const char* operator* () const { return p_; }
  char& operator[](int i) const { return p_[i]; }
};

void foo(int n){
  char_buffer p(n);
  do_stuff(p);       // use p like a char*
  do_some_more_stuff();
  p[0] = 'a';        // use like an array
  std::strcpy(s, p); // use like regular pointer
  std::cout << p << std::endl;
  /* etc */
} // memory deallocated, no matter how we leave
```

std::unique_ptr

```
#include <iostream> // for std::cout, std::endl
#include <memory>   // for std::unique_ptr

void Leaky() {
  int *x = new int(5); // heap allocated
  (*x)++;
  std::cout << *x << std::endl;
} // no delete, thus leaking

void NotLeaky() {
  auto x = std::make_unique<int>(5);
  // std::unique_ptr<int> x(new int(5));
  (*x)++;
  std::cout << *x << std::endl;
  std::unique_ptr<int> y = x;       // ERROR
} // no explicit delete, but x is not leaked

int main(int argc, char **argv) {
  Leaky();
  NotLeaky();
  return 0;
}
```

Déplacement de pointeur unique

```
auto x = std::make_unique<int>(5);

std::unique_ptr<int> y = x;            // ERROR
std::unique_ptr<int> y = x.release();  // OK
std::unique_ptr<int> y = std::move(x); // OK
```

Héritage et smart pointers

```
class Base {};
class Derived : public Base {};
...
Derived * dp1 = new Derived;
Base * bp1 = dp1;
Base * bp2(dp1);
Base * bp3 = new Derived;

...
using BasePtr = std::shared_ptr<Base>;
using DerivedPtr = std::shared_ptr<Derived>;

auto de1= std::make_shared<Derived>();
BasePtr ba1 = dp1;
BasePtr ba2(dp1);
BasePtr ba3(new Derived);

DerivedPtr de3 = static_pointer_cast<Derived>(ba3);
```