Exemples de décoration de nom

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

`int Something::Inside::Deeper::deeperMethod(void)`
`_ZN9Something6Inside6Deeper10deepMethodEv`

`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

```
3  struct Vehicle {};
4  struct Car : Vehicle {};
5
6  int main(){
7    std::vector<Vehicle *> vehicles;
8    std::vector<Car *> cars;
9
10   vehicles = cars; // ERROR
11 }
```

Surcharge, redéfinition et variance

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

```
3  class Top {};
4  class Middle : public Top {};
5  class Bottom : public Middle {};
6
7  struct Up {
8     virtual std::string cov(Top* t)       {
9        return "Up";
10    }
11    virtual std::string inv(Middle* m)    {
12       return "Up";
13    }
14    virtual std::string contra(Bottom* m) {
15       return "Up";
16    }
17 };
18 struct Down : public Up  {
19    std::string cov(Middle* t)    {
20       return "Down";
21    }
22    std::string inv(Middle* m)    {
23       return "Down";
24    }
25    std::string contra(Middle* m) {
26       return "Down";
27    }
28 };
29 int main() {
30    Up* u = new Up();
31    Down* d = new Down();
32    Up* ud = new Down();
33    Top* top = new Top();
34    Middle* mid = new Middle();
35    Bottom* bot = new Bottom();
```

	u.?(?)	**d**.?(?)	**ud**.?(?)
?.**cov(top)**	Up	Compil error	Up
?.**cov(mid)**	Up	Down	Up
?.**cov(bot)**	Up	Down	Up
?.**inv(top)**	Compil error	Compil error	Compil error
?.**inv(mid)**	Up	Down	Down
?.**inv(bot)**	Up	Down	Down
?.**contra(top)**	Compil error	Compil error	Compil error
?.**contra(mid)**	Compil error	Down	Compil error
?.**contra(bot)**	Up	Down	Up

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;
} // never used delete, therefore leak

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
} // never used 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);
```

Règle de 3 (C++98)

```
class X {
  ~X();                       // Destructor
  X(X const& other);          // Copy constructor
  X& operator=(X const& o); // Assignment operator
};
```

Règle de 5 (C++11)

```
class X {
  ~X();                       // Destructor
  X(X const& other);          // Copy constructor
  X(X && other);              // Move constructor
  X& operator=(X const& o); // Copy Assignmt oper.
  X& operator=(X && o);// Move assignment operator
};
```