Formation C++ Ubisoft - Module 1

Romain Arcila[1,2]
Charles de Rousiers[1]

15 mars 2009

[1] Inria Grenoble
[2] Liris -CNRS Lyon

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

But de la formation

- Concept de base du C++
- Gestion de la mémoire
- Design Pattern
- Template

$\Rightarrow$ Utilisation correcte du C++.

Deroulement - Planning

1. Concept de base C++ : rappels, objet, operateur
2. La mémoire en C++
3. Design Patern
4. Template (2 séances)
5. Optimization
6. Test Unitaire

Deroulement - Fonctionnement

Mélange :

- Cours
- TD
- TP

Mini-Projet : Raytracer

http ://eva-
sion.inrialpes.fr/Membres/Romain.Arcila/files/seance1.zip

Plan de la séance

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Allocation mémoire I

Nouvelles primitives d'allocation mémoire

- new / delete
  allocation/désallocation d'un élément
- new[] / delete []
  allocation/désallocation d'un tableau élément

Code : Allocation

```
int *a = new int(5);
delete a;
int *b = new int[5];
delete [] b;
```

Allocation mémoire II

- Remplace `free`/`malloc`.
- `malloc` et `free` : intéraction avec du code C

Warning : Allocation

Ne pas mélanger les primitives.

Plus de détails dans le module 2.

Classe de stockage

- Plusieurs types de classes de stockage :
  - visibilité : *static*, *extern*, *register*
  - qualificatif : *auto*, *const*, *volatile*,
- Visibilité et qualificatif combinable.
- Signification dépendante du contexte.

Classe de stockage : variable globale

- Visibilité :
  - static : variable utilisable uniquement dans le fichier en cours. (idem pour les fonctions)
  - extern : variable déclarée dans un autre fichier.
  - register : la variable est dans un registre (& impossible)
- Attribut :
  - const : constante, le contenu est fixé lors de la déclaration
  - volatile : la variable peut être modifiée par un processus externe
  - auto : classe de stockage standard

Classe de Stockage : variable globale

Code : Classe de Stockage

```
int i; // same as auto int i;
const int j=0;
static int k=0;
extern int l;
register int m;
static extern n; // Error
```

Classe de stockage : variable dans les fonctions

- Visibilité :
  - static : initialisée lors du premier appel.
- Attribut :
  - const : constante, le contenu est fixé lors de la déclaration
  - volatile : la variable peut être modifié par un processus externe
  - auto : classe de stockage standard

Classe de Stockage III

Présentation uniquement sémantique

$\Rightarrow$ conséquence sur la vitesse dans le module optimisation

Reference

Principe : définir un alias sur une autre variable.

Code : Référence

```cpp
int a=5;
int &b=a;
b=6; // a==6
```

Reference II

Warning : Reference

- Doit être initialisée

```
int &ref; // Interdit
```

- Ne peut être réaffectée

```
int a=5;
int b;
int &ref=a:
ref=b; // a==b, et non ref sur b;
```

- Ne peut etre initialisée avec un temporaire si la référence n'est pas constante

```
int &ref=Constructeur(); // interdit
const int& ref2= Constructeur() // ok, permet les arguments par
// defaut dans les fonctions : void f(const int& a=2);
```

Principale utilisation : Passage de paramètre dans les fonctions

Fonctions

en C : Passage par valeur ou par pointeur

Code : Paramètres en C

```
void f(int a); // a est une copie
void f(int* a);
```

Problème de la copie.

Fonctions II

en C++ : Passage par valeur, par pointeur ET par référence les plus utilisés.

Code : Paramètres en C++

```
void f(int a); // a est une copie
void f(int* a); // a est un pointeur
void f(const int* a); // contenu constant : \lstinline$ *a=?$ interdit
void f(int* const a); // ne peut etre deplace : \lstinline$a ++$ interdit
void f(const int * const a); // contenu et adresse constant
void f(int& a); // alias sur a
void f(const int& a); // reference constante sur a
```

Fonctions III

Code : pointeur/reference

```
void f(int* a);
void g(int& a);
```

Code : Exemple

```
int main(int argc, char** argv) {
  int a;
  int b;

  f(&a);
  g(b);

  return 0;
}
```

Ceci est une question de préférence.

Surcharge

Principe : Plusieurs fonctions peuvent porter le même nom.

Code : Surcharge

En C :

```
int addInt(int a, int b);
Complex addComplex(Complex a, Complex b);
```

En C++ :

```
int add(int a, int b);
Complex add(Complex a, Complex b);
```

Un prototype == une signature.

Surcharge : Résolution

Signature :

- nombre d'arguments
- type des arguments
- Type de retour : non pris en compte

Résolution

1. Signature (quasi-)exacte : T, T en T&, "constification", tableau en pointeur
2. Promotion et conversion utilisateur
3. Cast avec perte
4. Fonction avec paramètres variables

Résolution exemple

Code : Résolution

```
void print(int);  void print(const char*);
void print(double); void print(long);
void print(char); void print (...);

short s;
print('a');
print(49);
print("a");
print(s);
print(1.f);
```

Résolution exemple 2

Code : Résolution

```
void f(double, int);
void f(int,double);
int a;
int b;
f(a,b); // a caste en double ou b caste en double
//////////////
void g(double);
int g(double); // erreur
```

Résolution 3

Surchage pointeur/int

```
void f(char * s);
void f(int a);
f(NULL); // ??
```

Appel de `f(int)` car en C++, `NULL = (void*) 0`

Inline

But : Supprimer un appel de fonction

Code : Inline

```
inline void f(int a);
```

- Disponibilite.
- Non obligatoire : le compilateur peut refuser d'inliner la fonction.

L'implémentation de la fonction doit être disponible lors de l'appel de la fonction.

Argument par défaut

Code : Argument par défaut

```
void f(int i=0, int j=2);

f(); // appel f(0,2);
f(2); // appel f(2,2);
f(2,3); // appel f(2,3);
```

Warning

```
void g(int i=0, int j); // interdit
```

Plan

1. Présentation de la formation
2. C++ vs C
3. **Programmation Orientée Objet**
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Encapsulation

Objet == Encapsulation

Un objet :

- variables = attributs
- fonctions opérant sur ces attributs = méthodes

Syntaxe

```
class Name {
  void f (...);
  int g (..);
  int a;
}; // <= important
```

```
struct Name {
  void f (...);
  int g (..);
  int a;
}; // <= important
```

Déclaration - Défintion

Déclaration

- C.h :

```
class C {
  int f(int);
  int g(double a) {...} // g est inline
  int h();
};

inline int C::h() {...} // h est inline
```

- C.cc :

```
int C::f(int) {}
```

Cycle de Vie

Cycle de vie

```
class A{};
{
  A a; //Constructeur
  A b(a); // Constructeur de copie
  A c=a; // Constructeur de copie
  c=b; // Operator=
} // << Destructeur de A
```

Constructeur I

Principe :

Initialiser une instance :

- initialiser les variables.
- allouer de la mémoire.
- autre ... (Think RAII)

Syntaxe :

Constructeur

```
class A {
  A() {a=0; b="aa";}
  A (... arg ...) {a=arg1;b=arg2;}
  int a; string b;
};
```

Constructeur : Liste d'initialisation

Syntaxe

Code : Liste d'initialisation

```
class A {
  A(int _a, string _b): a(_a), b(a) {}
    int a;
    int b;
  };
```

- Plus rapide
- Souvent obligatoire
- Une règle : meme ordre dans la liste que celui des déclarations

Constructeur : Liste d'initialisation

Code : Ordre des arguments

```
class A {
  A(int _a): a(_a),b(a) {}
  int b;
  int a;
}
// "b" est  initialise  en premier, a ce moment la "a" a
// n'importe quelle  valeur.
```

Destructeur

Principe

Désallouer les ressources allouées par le constructeur

Syntaxe

Code : Destructeur

```
class A {
  ~A() {}
};
```

- 1 seul destructeur par classe
- toute allocation dans le constructeur == désallocation dans le destructeur

Constructeur de Copie

Principe

Instancier un objet à partir d'un autre

Syntaxe

Code : Constructeur de copie

```
A(const A& _a): a(_a.a) {} // & et liste  initialisation
A a;
A b(a) // b est  initialise  en fonction de A;
```

Opérateur d'affectation

Principe

Affecter un autre objet à une autre instance

Syntaxe

Code : Affectation

```
A& operator=(const A& other) {
  if (this != &other) {
    a=other.a;
    b=other.b;
  }
  return *this;
}
A a;
A b;
b=a; // b egale a A
```

Classe Canonique

Le C++ fournit par défaut

- Un constructeur par defaut
- Un destructeur
- Un constructeur de copie
- Operateur d'affectation

Si on definit un constructeur, le constructeur fourni par le C++disparait. Il faut le réimplénter

Constructeur

Code : Constructeur

```
class A {
  int a;
  public:
    A(const int& _a): a(_a) {}
};
A a; // error
// correction
class A {
  int a;
  public A(const int& _a=0): a(_a) {}
};
A a; // ok
```

Copie bit a bit

Par defaut copie bit à bit : copie des pointeur (adresses) et non du contenue ⇒ contenu des pointeurs partagée entre toutes les copies.

Allocation ⇒ Constructeur de copie, Destructeur, Operator=

Visibilité I

Objet == Encapsulation
⇒ Besoin de visibilité

Visibilité II

Objet == Encapsulation

- Public : interface = Eléments accessibles de l'extérieur
- Private : implémentation = Eléments non accessibles de l'extérieur
- Protected : implémentation et interface

Visibilité III

class == (struct%visibilité)

Visibilité IV

classes et fonctions amies : confère un accès priviligié= acces aux champs privés

- non symétrique : classe A amie de classe B n'implique pas B amie de A
- non transitif : classe A amie de classe B et classe B amie de classe C n'implique pas A amie de C
- non hérité : classe A amie de B n'implique pas fille(A) amie de B

Visibilité IV

syntaxe

Code : Friend

```
class A {
  friend void f(const A& a); // amie de la fonction f
  friend class C; // amie de la classe C;
  friend class D::g // amie de la methode g de D
};
```

Remarques :

- L'abus de friend est mauvais
- Une classe interne n'a pas d'accès priviligié
- friend ⇒ classes fortement liées (ex : cell et list)

Classe de stockage

- attribut
  - auto : défaut
  - const : ne peut être modifié
  - volatile : modifié par un processus externe
  - static : partagé par toutes les instances d'une classe
  - mutable : peut être modifié dans une méthode const
- méthode
  - const : méthode ne modifiant pas l'état de l'objet
  - volatile : pouvant être modifiée par un autre processus
  - static : méthode pour toute les instances

Mutable

Code : Mutable

```
class Mut {
  int a;
  mutable int b;
  void f() const {
    a++; // error
    b++; // ok
  }
};
```

Classe de stockage II

- attribut const : initialisé obligatoirement dans la liste
- attribut static :
  - déclaré dans un fichier source : $intA :: i = 5$ (sauf type intégral constant)
  - problème initialisation : ordre non fixé.

Code : static

```
struct A { static int a; static int b};
int A::a=0;
int A::b=A::a; // WARNING
```

Classe de stockage III

- méthode const
  - fait partie de la signature : surcharge méthode const/non const
  - syntaxe : `class A {void f() const; void g();};`
  - spécifie un contrat
  - seule méthode appelable sur un objet constant

    `const A a; a.g(); //error`
- static :
  - ne peut accéder qu'aux attributs statiques
  - Appel `class A { static void g(); }; A::g()`
- méthode volatile
  - fait partie de la signature : surcharge méthode volatile/non volatile
  - syntaxe : `class A {void f() volatile ;};`
  - seule méthode appelable sur un objet volatile

RAII

Ressources aquisition is initialization :

Création d'un objet : initialisation d'une ressource et aquisition

Destruction : relâchement de la ressource

$\Rightarrow$ Assure que les ressources sont gérées correctement

Exemple :

- Fichier :
  - constructeur : ouverture du fichier
  - destructeur : fermeture du fichier
- Shared Pointeur : module 2

RAII 2 : Scoped Handle

Code : Scoped Handle Non template

```
struct Deleter { virtual void operator()( int *i) const =0 ; };

struct DeleterDel: public Deleter { void operator()( int *i) const {delete i;} };

struct DeleterMalloc: public Deleter { void operator()( int *i) const {free(i);} };

class ScopedHandle {
  int* value;
  const Deleter& meth;
public:
   ScopedHandle(int* arg, const Deleter& del): value(arg), meth(del) {}
  ~ScopedHandle() { meth(value); }
};

{
  int *a= new int;
  int *b=(int*) malloc(sizeof( int ));
  ScopedHandle sc(a,DeleterDel());
  ScopedHandle sc2(b,DeleterMalloc());
  // play with a and b
}
```

RAII 2 : Scoped Handle

Code : Scoped Handle Generalisation Template

```
template<class T, class T2>
class ScopedHandle {
public:
   ScopedHandle(T arg, T2 _close): value(arg), meth(_close) {}
  ~ScopedHandle() { meth(value); }
private:
  T value;
  T2 meth;
};

struct Deleter {
  void operator()(int *i) {delete i;}
};

{
  int *a= new int;
  int *b=(int*) malloc(sizeof(int));
  ScopedHandle<int*,Deleter> sc(a,Deleter());
  ScopedHandle<int*, void (*)(void*)> sc2(b,&free);
}
```

Divers

- constructeur explicit :
  - syntaxe `class A { explicit A(int); };`
  - oblige à appeler explicitement le constructeur ⇒ interdit les conversions implicites.

Code : explicit

```
struct A { A(int a) {}};
struct B { explicit B(int a) {}};

void f(A a) {}
void g(B a) {}

int a;
f(a);
g(a); // failed
g(B(a)); // ok
```

- Utiliser le plus possible

Concept

- Utilise la programmation par objet.
- Utilise l'héritage (sous classe) pour spécifier le comportement du programme.

Syntaxe

Code : Syntaxe

```
class B: {public|protected|private} A {
  B(...): A(...) {}
};
```

Construction

Constructeur de la classe mère : list initialisation

Ordre de construction : mère vers fille.

Destruction

Destructeur de la classe mère automatiquement appelé :

Code : Destructeur

```
class B: public A{
  ~B() {} // destructeur de A appeler
}
```

Ordre de destruction : fille vers mère.

Copie

Appelé dans le constructeur de copie

Code : Constructeur

```
class B: public A{
  B(const B& b): A(b) {}
}
```

Affectation

Code : Affectation

```
class B: public A {
  B& operator=(const B& b) {
    A::operator=(B); // ou  static_cast ...
    ...
  }
};
```

Type Héritage

- public : méthode hérité garde la visibilité, B et vu comme un A
- protected :méthode public $\Rightarrow$ protected
- private : public,protected $\Rightarrow$ private, B != A

Type Héritage

- public : Relation "est un"
- private :"implémenter en"

Heritage

Code : Heritage

```
class A {
  void f() {cout << "a" << endl; }
};
class B: public A {
};

a.f()
b.f()
```

affiche "a a"

Heritage II

Code : Heritage

```
class A {
  void f() {cout << "a" << endl; }
};
class B: public A {
  void f() {cout << "b" << endl; }
};

a.f()
b.f()

A *a1=new A;
A *a2=new B;
a1->f();
a2->f();
```

affiche "a b a a"

Heritage II

Code : Heritage

```
class A {
  virtual void f() {cout << "a" << endl; }
};
class B: public A {
  void f() {cout << "b" << endl; }
};

a.f()
b.f()

A *a1=new A;
A *a2=new B;
a1->f();
a2->f();
```

affiche "a b a b"

Heritage III

Code : Heritage

```
class A {
  virtual void f() {cout << "a" << endl; }
};
class B: public A {
  void f() {cout << "b" << endl; }
};

a.f()
b.f()

A a1;
A a2=B();
a1.f();
a2.f();
```

affiche "a b a a"

Heritage Synthese

- Redefinition de méthode : virtual
- Polymorphisme : pointeur et reference

Pure

Code : Heritage

```
class A {
  virtual void f()=0;
};
A a();
```

Des remarques ?

Pure 2

Méthode virtuelle pure : Définie à un niveau de la hiérarchie où on ne peut résoudre le problème.
Toute classe héritant doit implémenter ces méthodes

classe Abstraite == Interface.

Virtual++

Une méthode virtuelle pure peut avoir une implémentation.

- non appelable directement
- fournit une implémentation par défaut

Code : Virtual implémentation

```
struct A {
  virtual void m()=0;

};
void A::m(){cout << "defaut"<< endl;}

struct B: public 1 {
  virtual void m(){A::m()}
}
```

Une méthode virtuelle ne peut être inline.

classe Abstraite

- Par défaut, une méthode est non virtual
- une méthode virtual sera toujours virtual

Heritage

Code : Heritage

```
class A {
};

class B: public A {
  B(): i(new int) {}
  ~B(){delete i;}
  int *i;
};

{
  A* a=new B;
}
```

Tout est ok.

Heritage

Non : le destructeur de A n'est pas virtuel

Code : Heritage corrige

```
class A {
  virtual ~A(){}
};
class B: public A {
  B(): i(new int) {}
  virtual ~B(){delete i;}
  int *i;
};

{
  A* a=new B;
}
```

Destructeur dans A non virtual ⇔ destructeur de A

Destructeur virtuel

Conclusion :

Toute classe destinée à être héritée doit avoir un destructeur.

Corollaire :

Ne pas hériter d'une classe au destructeur non virtuel

Corollaire :

Empecher héritage : destructeur non virtual.

Polymorphisme

Utilisation de l'héritage :

Code : Polymorphisme

```
class Figure {
  void rotate(float angle) = 0;
  void draw() = 0;
};

class Square : Public Figure {//implemente square
};

class Circle : Public Figure {//implemente square
};

void f() {
  std::vector<Figure*> figs;
  // add Square et Circle to figs
  for (int i=0; i< figs.size(); ++i)
    figs[i]->draw();
}
```

Heritage multiple

heritage de plusieurs classes.

Code : Heritage multiple

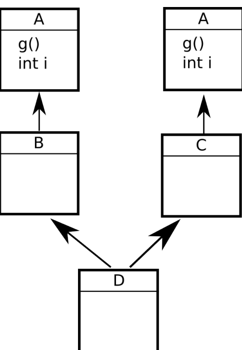
```
class A { void f };
class B { void g };
class C : public A, private C {};
```

Heritage multiple

Code : Heritage Multiple

```
struct A{};
struct B: public A{};
struct C: public A{};
struct D: public B, public C {};
```

Heritage multiple



Heritage multiple

Code : Heritage multiple

```
struct A {
  virtual void g() {cout << "A::g()"<< endl;}
  int i;
};

struct B : public A {
  virtual void g() {cout << "B::g()"<< endl;}
};

struct C: public A {
  virtual void g() {cout << "C::g()"<< endl;}
};

struct D: public B, public C {
  virtual void g() {cout << "D::g()"<< endl;}
};
```

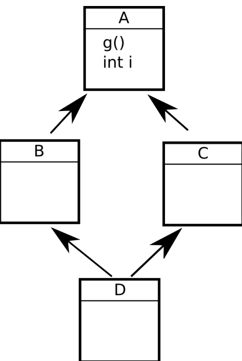
```
void f( A* a) { a->g();}
void h( B* a) { a->g();}
void i( C* a) { a->g();}
void j( B* a) { a->g();}
void j( C* a) { a->g();}

D *d= new D();
d->g();// ambiguous if not overidden by D
d->C::g(); d->B::g();
d->C::i=2; d->B::i=3;

cout << d->C::i << endl;
cout << d->B::i << endl;

f(dynamic_cast<B*>(d)); f(dynamic_cast<C*>(d));

h(d); i(d);

j(dynamic_cast<B*>(d));
j(dynamic_cast<C*>(d));
```

Heritage multiple

Code : Virtual Heritage Multiple

```
struct A{};
struct B: virtual public A{};
struct C: virtual public A{};
struct D: public B, public C {};
```

Heritage multiple



Heritage multiple

Code : Héritage virtuel

```
struct A {
  virtual void g() {cout << "A::g()"<< endl;}
  int i;
};

struct B : virtual public A {
// virtual void g() {cout << "B::g()"<< endl;}
};

struct C: virtual public A {
// virtual void g() {cout << "C::g()"<< endl;}
};

struct D: virtual public B, public C {
// virtual void g() {cout << "D::g()"<< endl;}
};
```

```
void f( A* a) { a->g(); }
void h( B* a) { a->g(); }
void i( C* a) { a->g(); }
void j( B* a) { a->g(); }
void j( C* a) { a->g(); }

D *d= new D();
d->g(); // ambiguous if overidden by B and C,
// if overidden B or C call B/C->g
d->g(); d->g();
d->C::i=2; // Same i
d->B::i=3; // Same i

cout << d->C::i << endl;
cout << d->B::i << endl;

f(d);  f(d);
h(d); i(d);
j(dynamic_cast<B*>(d));
j(dynamic_cast<C*>(d));
```

Heritage : Tricks

- pas de méthode virtuelle dans constructeur et destructeur.
- constructeur de copie virtuel.

Clone

```
class Object {
  virtual clone() const=0;
};
// implement type of Object
void f(const Object* obj) const {
  Object* obj_copy=obj->clone() // Quelque soit obj.
}
```

Heritage : Tricks II

- `vector<A*>` != `vector<B*>` $\Rightarrow$ ne pas convertir un `vector<A*>` en `vector<B*>`
- Ne pas redefinir une méthode non virtuelle. (comportement différent suivant que l'on passe par un pointeur ou non)
- Ne pas changer la valeur d'un parametre par defaut. (idem au dessus)
- Ne pas cacher les méthodes héritées.

Methode heritee cache

```
struct A {
  void f(int a);
};
struct B: public A { void f();}
B b;
b.f(5) // error
```

Heritage : Tricks II

- `vector<A*>` != `vector<B*>` ⇒ ne pas convertir un `vector<A*>` en `vector<B*>`
- Ne pas redefinir une méthode non virtuelle. (comportement différent suivant que l'on passe par un pointeur ou non)
- Ne pas changer la valeur d'un parametre par defaut. (idem au dessus)
- Ne pas cacher les méthodes héritées.

Methode heritee cache

```
struct A {
  void f(int a);
};
struct B: public A { void f();}
B b;
b.f(5) // error

struct B: public A { void f(); using A::f;}
B b;
b.f(5) // ok
```

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Template

- Généricité
- Rappel Rapide

Template Syntaxe

- Principe Simple
- Nombreux piège

Template

```
template<class T>
class Vec3G {
  T[3] vec;
  public :
    typedef T type;
    Vec3G(const T& t1=T(), const T& t2=T(),
          const T& t3=T()): vec(t1), vec(t2), vec(t3) {}
           ....
};
Vec3G<float> v; // tous les T sont remplaces par float .
```

CRTP

CRTP : idiom du C++ : Une clasee hérite d'elle même
Utilisation principale polymorphisme statique Détailler dans le module Template.

CRTP

```
template<class Derived>
class Base {
  void print() const {
    static_cast <Derived&>(this).print();
  }
};
```

CRTP

CRTP

```
class D1: public Base<D1> {
  void print() const {
    cout << "d1" << endl;
  }
};
D1 d;
d.print();
```

CRTP

CRTP : template deplie avec D1

```
TO BE WRITTEN
```

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Durée de vie

variable sur le tas : jusqu'a la désallocation

- Variable automatique et constante :
  - global : durée du programme
  - fonction et bloc ({}) :durée du bloc
  - classe : durée de la classe
- variable static :
  - global : durée du programme
  - fonction : 1ere execution de la fonction $\longrightarrow$ fin du programme
  - classe : durée du programme

visibilité

Code : Conflit

```
// socket.h
int open(int port); // open a socket and return a file descriptor
char* read(int desc); // read for a socket

// file.h
int open(string name); // open a file and return a file descriptor
char *read(int fd); // read for a file

// main.cc
#include "socket.h"
#include "file.h"

// conflit redeclaration de read
```

Namespace

Namespace : permet de regrouper des entités (classe, fonction...)

Code : Namespace

```
// socket.h
namespace socket{
  int open(int port); // open a socket and return a file descriptor
  char* read(int desc); // read for a socket
}
// file.h
namespace file {
  int open(string name); // open a file and return a file descriptor
  char *read(int fd); // read for a file
}
```

→ résout les conflits

Namespace II

- peut étre réouvert
- peut s'étendre dans plusieurs fichiers
- peut contenir des namespaces

Namespace III

using namespace :importe tout l'espace de nom dans l'espace courant

Code : using

```
// main.cc
#include "socket.h"
#include "file.h"

using namespace socket;
int sock=open(0);
```

Namespace IV

`using namespace::item` :importe item dans l'espace courant

using : :

```
// main.cc
#include "socket.h"
#include "file.h"

using namespace socket::open;
int sock=open(0);
```

Utilisation

Code : Erreur

```
// main.cc
#include "socket.h"
#include "file.h"

using namespace socket;
using namespace file;
int sock=open(0); // erreur
```

Opérateur de résolution de portée " : :" : spécifie où trouver l'élément.

Code : Operateur de Portée

```
// main.cc
#include "socket.h"
#include "file.h"

using namespace socket;
using namespace file;
int sock=socket::open(0);
```

Namespace alias

Namespace long ou imbriqué :

Code : Namespace

```
namespace un_nom_tres_long {
  int i;
}
namespace A {
  namespace B {
    namespace C {
      int i;
} } }

un_nom_tres_long::i=5;
A::B::C::i=2;
```

Solution : alias de namespace.

Code : Namespace Alias

```
namespace nom= un_nom_tres_long;
namespace abc=A::B::C;
nom::i=5;
abc::i=2;
```

Anonyme

Namespace anonyme :

Namespace Anonyme

```
namespace { int i;}
```

`i` a une portée de fichier, à préférer aux variables statiques.

Regle

Ne pas donner le même nom à un namespace et une classe.

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Principe

Surcharge :

Surcharge de fonctions

```
void f(int a, int b);
void f(double a, double b);
```

et les opérateurs ?

Surcharge des operateurs ?

```
int a,b;
int c=a+b;
Vecteur d,e;
Vecteur f=d+e; //???
```

$\Rightarrow$ Définir des opérations sur des types utilisateurs

Principe II

Définition opérateur :

- interne : `class Point {Type operatorOP(...){}};`
- externe : `Type operatorOP(..., ...){}`

Opérateur interne ou externe.

Principe

```
Point p1,p2,p3;
p1=p2+p3;
```

Réécrit :

```
Point p1,p2,p3;
p1=p2.operateur(p3); //1er essai
p1=operator+(p2,p3); //2eme essai
// ceci est effectue a la compilation
```

Règles de base

Des règles ... logiques !

- Garder le sens des opérateurs
- Tenir compte des priorités.
- Garder la sémantique

Sémantique : Exemple

```
const Point operator+(const Point& p1, const Point& p2) {
  Point p(p1);
  p.x+=p2.x;
  p.y+=p2.y;
  return p;
}
```

Que se passe-t-il si : 1) les arguments ne sont pas constants ? 2) on renvoie une reference ? 3) le resultat n'est pas constant ?

Reponse

1. l'operateur peut modifier ces arguments
2. reference sur un temporaire
3. (p1+p2)=...;

Opérateur Surchargeable

- Opération Arithmétique

  `+ += ++ (pre et post)`
  `- -= -- (pre et post)`
  `* *=`
  `/ /=`

- Operateur bit a bit

  `>> >>= << <<=`
  `^ ^= | |=`
  `& &= ~`

- Operation Booleene

  `! || &&`

- Comparaison

  `== != < <= > >=`

- Autre

  `* -> , [] () new delete`

Regles II

- operator+() : prefix : renvoie une reference
- operator+(int) : postfix : renvoie une copie
- operatorOP=, operator[], operator(), renvoient des références : interne
- operator<< et operator>> : externe
- Pour + − == ... : symétrique : externe, sinon interne
- Eviter de surcharger operator,
- Ne PAS surcharger operator||, operator&&...

Operator= (1)

Operator= est particulier !

Affectation

```
Point a;  Point b;
a=b; //a.operator=(b)
```

Prototype : `Type& operator=(const Point& b);`

Il faut éviter l'auto-affectation : `a=a`

Affectation Correcte

```
Type& operator(const Point& b){
  if (this != &b) {....}
  return *this;
}
```

Operator= (2)

Affection

```
struct A {
  int x;
  A& operator=(const A& a) {
    x=a.x;
    return *this;
  }
};

struct B: public A{
  int y;
  B& operator=(const B& a) {
    y=a.y;
    return *this;
  }
};
```

Est ce suffisant ?

Operator= (3)

Réponse : **NON**

Affection Corrigé

```
struct A {
  int x;
  A& operator=(const A& a) {
    x=a.x;
    return *this;
  }
};

struct B: public A{
  int y;
  B& operator=(const B& a) {
    A::operator=(a);
    y=a.y;
    return *this;
  }
};
```

Operateur virgule : Non template

Code : Operator,

```
vector<int>&
operator,(vector<int>& v, int i) {
  v.push_back(i);
  return v;
}

struct Sequence{
  vector<int> v;
};

vector<int>& operator+=(vector<int>& v, Sequence& s) {
  v.insert(v.end(),s.v.begin(), s.v.end());
  return v;
}
```

```
Sequence&
operator,(Sequence& seq,int j) {
  seq.v.push_back(j);
  return seq;
}

vector<int> v;
v=v,1,2; // v contient 1 2
Sequence s;
v+=(s,2,3,5); // v contient  1 2 2 3 5
```

A éviter : Regle de priorite de "," tres confuse

Operateur virgule

Code : Operator,

```
template<class T>
vector<T>&
operator,(vector<T>& v, T i) {
  v.push_back(i);
  return v;
}

template<class T>
struct Sequence{
  vector<T> v;
};

template<class T>
vector<T>& operator+=(vector<T>& v, Sequence<T>& s) {
  v.insert(v.end(),s.v.begin(), s.v.end());
  return v;
}
```

```
template<class T>
Sequence<T>&
operator,(Sequence<T>& seq,T j) {
  seq.v.push_back(j);
  return seq;
}

vector<int> v;
v=v,1,2;
Sequence<int> s;
v+=(s,2,3,5);
```

A éviter : Regle de priorite de "," tres confuse

Operator&&

Problème : toutes les closes sont évaluées :

Code : Operator&&

```
struct A {
  bool operator&&(...);
  void methode();
};

A* a=0;
if (a && a->methode()) {} // a->methode appele meme si a=0;
```

Operateur de flux

Operateur toujours externe

Code : Operateur de flux

```
isteam& operator>>(istream &is, Complex& c) {
  is >> c.re;
  is >> c.im;
  return is;
}
osteam& operator<<(ostream &os, const Complex& c) {
  os << c.re;
  os << c.im;
  return os;
}
```

Typeid

Permet de connaitre le type d'un objet et de comparer des classes.

Code : typeid

```
#include <typeinfo>

class A {};
class B {};
class C: public A {};
class D: public A {};

if (typeid(A) == typeid(A))
  cout << "Yep Yep" << endl;
if (typeid(A) != typeid(B))
  cout << "Yep Yep" << endl;

A* c=new C(); A* d=new D();

if (typeid(c) == typeid(d))
  cout << "Yep Yep" << endl;
}
```

Operator Tricks

- préférer `++i` à `i++`
- Réduire les opérateur amis ⇒ Exprimer OP en fonction de OP= :

Code : OperatorOP=

```
operator+(const A&a,const A&b){ A tmp(a); return (tmp+=b);}
```

- Virtual `operator<<` :

Code : virtual operator<<

```
struct A {
  virtual ostream& print(ostream& stream) {
    stream << "Mother"; return stream;}
};
struct B: public a {
  ostream& print(ostream& stream) {stream << "Child"; return stream;}
};
ostream& operator<<(ostream& os, const A& a) {return a.print(os);}
```

Marche avec les fonctions friends.

Operator Tricks II

Barton-Nackman trick : Repose sur le CRTP (Module 4 !)

Code : Bartin-Nackman

```
template<class T> class Arithm {
  friend T operator+(const T& a, const T& b) { T tmp(a); return (tmp+=b);}
  friend T operator-(const T& a, const T& b) { T tmp(a); return (tmp-=b);}
  friend T operator/(const T& a, const T& b) { T tmp(a); return (tmp/=b);}
  friend T operator*(const T& a, const T& b) { T tmp(a); return (tmp*=b);}
}; //NOTE : friend obligatoire

class Point: private Arithm<Point> {
  // Fournit +=, -=, *=, /= : +,-,/,* fournit automatiquement
};
```

Egalement faisable pour les comparaisons ...

Operator Tricks III

Barton-Nackman trick (2nd version) : Repose sur le CRTP (Module 4 !)

Code : Bartin-Nackman

```
template<class T> class Arithm {
  T operator+(const T& b) { T tmp( static_cast <T&>(*this)); return (tmp+=b);}
   ...
};

class Point: public InternalArithm<Point> {
  // Fournit +=, -=, *=, /= : +,-,/,* fournit automatiquement
};
```

Egalement faisable pour les comparaisons ...

Operator Trick III : Conversion Utilisateur

Conversion type utilisateur :

Code : Conversion Syntaxe

```
operator T() const {...} // pas de type de retour
```

Code : Conversion Syntaxe

```
struct A {
  operator int() const {return i;}
  int i;
};
A a;
int j=a; // Appel A.operator int()
```

Ce sont des méthodes (internes).

Operator Trick IV : Cast

Chaque cast a une fonction et une seule.

- reinterpret_cast <T>() : Conversion avec aucune vérification
- static_cast <T>() : conversion avec un minimum de vérification
- dynamic_cast<T>() : conversion haut/bas/transversale dans une hierarchie de classes
- const_cast <T>() : enlève l'attribut const sur une variable

Ne plus utiliser les conversionde type C : `(type)`

Operator Trick IV : const_cast

- permet de retirer l'attribut const ou volatile
- la variable doit être non const à l'origine.

Operator Trick IV : static_cast

- cast avec vérification à la compilation : rapide.
- peu de vérification

Operator Trick IV : dynamic_cast

- cast avec vérification à l'utilisation : plus lent.
- Vérification sur la compatibilité

Operator Trick IV : reinterpret_cast

- cast sans vérification à l'utilisation : rapide.
- Aucune vérification.
- Souvent non portable.

Operator Trick V : Functor

- Remplace les pointeurs de fonctions : très utilisé dans la STL et Boost.
- Utilise `operator()` pour simuler des appels de fonctions.

Code : Functor

En C :

```
typedef int (*operator)(int, int);
void f(int *tab, int size, operator op) {
  for (int i=0; i < size; ++i)
    tab[i]=operator(tab[i],5);
}
```

En C++ :

```
template<class T>
void f(int *tab, int size, T op) {
  for (int i=0; i < size; ++i)
    tab[i]=op(tab[i],5);
}
```

Operator Trick V : Functor II

Functor II

```
struct test { virtual bool operator()( int e) const =0; };

struct positive: public test { bool operator()( int e) const {return e>0;} };

struct negative: public test { bool operator()( int e) const {return e<0;} };

class vector_constraint {
  vector<int> v;
  const test& _validate;
public:
  vector_constraint(const test& t=positive()): _validate(t) {}
  void push_back(int e) {
    assert(_validate(e));
    v.push_back(e);
  }
};

  positive p;
  vector_constraint v(p);
  v.push_back(1); // ok
  v.push_back(-2); // assertion
```

Operator Trick V : Functor II

Functor II

```
template<class T>
class vector_constraint{
public:
  vector_constraint(): _validate(T()) {}
  void push_back(int e) {
    assert(_validate(e));
    v.push_back(e);
  }
private:
  vector<int> v;
  T _validate;
};

struct positive {
  bool operator()( int e) {return e>0;}
};

vector_constraint<positive> v;
v.push_back(1); // ok
v.push_back(0); // assertion
```

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Design ? ?

Pourquoi ? On sait écrire des classes ? ?

Ce n'est pas suffisant !

3 étapes

- Design d'une class
- Ecriture
- Maintenance et Amélioration

Design

Visibilité des elements : définir précisément

Règle générale : attributs privés

Si les attribut sont privés : ne pas renvoyer de référence :

- supprime les règles de visibilités.
- Que se passe-t-il si :

Design

```
class A {
  int a;
  public:
  int &getA() {return A;}
};
A* a=new A;
int& i=a.getA(); // Reference
delete a;
i=5; // ?
```

Même Probleme avec const& et pointeur

Design II

- minimaliste
- complète
- simple
- facilement mémorisable
- lisible

Design Exemple

Mauvais design : Constructeur avec de nombreux arguments :

Code : Bad !

```
class File { File(const char* filename, bool read, bool write,
  bool append, bool binary, size_t block) {...}};
File f("tmp", true,true, false , false ,1024);
```

- illisible
- Seulement binary ?

Arguments par defaut ne sont pas une solution

Solution 1

Classe (interne) pour gerer les options.

Solution 1

```
class File {
  public:
  class Option {
    Option(): read(true), write(true), append(true),
    binary(true),block(1024) {}
    Option& setReadable(bool r) {read=r; return *this}
    Option& setWritable(bool r) {write=r; return *this}
    Option& setBlockSize(int size) {block=size; return *this}
     ...
  };
  File(char* filename, Option& options) {...}
};
```

Solution 1

Classe (interne) pour gerer les options.

Solution 1

```
class File {
  public:
  class Option {
    Option(): read(true), write(true), append(true),
    binary(true),block(1024) {}
    Option& setReadable(bool r) {read=r; return *this}
    Option& setWritable(bool r) {write=r; return *this}
    Option& setBlockSize(int size) {block=size; return *this}
     ...
  };
  File(char* filename, Option& options) {...}
};

File::Option options;
options.setReadable(true).setBlockSize(1024).setBinary(true);
File file("tmp",options);
```

Solution 2

Méthode permettant de définir les options :

Solution 2

```
class File {
  public:
  File& setReadable(bool r) {read=r; return *this}
  File& setWritable(bool r) {write=r; return *this}
  File& setBlockSize(int size) {block=size; return *this}
  File(char* filename) {...}
};
```

Solution 2

Méthode permettant de définir les options :

Solution 2

```
class File {
  public:
  File& setReadable(bool r) {read=r; return *this}
  File& setWritable(bool r) {write=r; return *this}
  File& setBlockSize(int size) {block=size; return *this}
  File(char* filename) {...}
};

File file("tmp");
file.setReadable(true).setBlockSize(1024).setBinary(true);
```

Paramètres Booléens

Un paramètre booléen n'est pas toujours suffisant :

Booléen

```
find("test", 'e', true);
```

Paramètres Booléens

Un paramètre booléen n'est pas toujours suffisant :

Booléen

```
find("test", 'e', true);
```

true : casse ? backward search

Paramêtres Booléens

Un paramètre booléen n'est pas toujours suffisant :

Booléen

```
find("test",'e',true);
```

true : casse ? backward search

Une énumération peut être plus explicite ! !

Plus explicite

```
find("test",'e',FIND::CaseInsentive);
```

Autre

- passage d'argument consistent
- getter/setter
- nom de méthode
- définir les variables le plus tard possible
- ...

Api ttt des erreurs

Les erreurs possibles et leurs traitements font partie du design.

Api ttt des erreurs

Les erreurs possibles et leurs traitement font partie de l'API.

$\Rightarrow$ Analyse des objets : fonctions et méthodes

Analyse

- Domaine de fonction : valeur interdite
- Erreur pouvant survenir : mémoire épuisé, droit d'accés refusé.

Règle de base :après chaque appel système, tester les erreurs.

Traitement des erreurs

On peut définir plusieurs niveaux et comportements

Niveau	comportement
Grave	assertion
Gérable	traitement
Récupérable localement	récupération

Assertion

Erreur ne pouvant être contournée, empêchant la suite du programme.

assertion

Assertion :

- Permet de s'arréter au moment de l'erreur et d'obtenir le fichier la fonction et la ligne de l'erreur.
- Peut génère un core dump : état de la pile.

Assertion II

- macro : éviter les tests avec effet de bord.
- retour pour le programmeur : n'aide pas l'utilisateur.

ASSERT

```
#ifndef NDEBUG
#define ASSERT(test, message) \
if (!(test)) { \
  cerr << message << endl; \
  assert(test);     \
}
#endif
```

- Stop le programme ⇒ Erreur à résoudre en premier

Récupérable

Erreur pouvant survenir dont la correction est triviale et locale.

Erreur minime

```
bool f(int v) {
  if (v < 0)
    v=0;
  int i=sqrt(v);
  // do something with i;
}
```

Conclusion

Ce sont les deux types d'erreurs faciles à gérér.

Pour les autres :

- code d'erreur
  - style C
  - léger
  - Dur a gérer
- exception
  - C++
  - plus lourd
  - plus simple

Code erreur

Deux maniéres

- code en retour.

```
int f (...) {
  if (error1)
    return 1;
  else if (error2)
    return 2;
  return 0;
}
```

- code en paramètre.

```
void f (..., int &error) {
  if (error1)
    error=1;
  else if (error2)
    error=2;
  error=0;
}
```

Code Erreur II

- contraignant.
- destructeur à appeler manuellement.

Exception

Syntaxe

Code : Exception

```
//lancer  une exception
throw type

//recuperer une exception
try {

} catch (type ) {

}
```

Exception II

- n'importe quel type peut être lancé
- catch peut être plusieurs niveaux au dessus
- plusieurs catch possibles
- throw relance l'exception courante
- catch(...) attrape toutes les exceptions
- lorsque qu'une exception est levée, les destructeurs sont appelés automatiquement

Exception et Héritage

Les exceptions peuvent être héritées

Heritage

```
struct Exception {
  public:
    virtual void print() const { cerr << "Erreur de base" <<endl;
};

struct Exception1: public Exception {
  void print() const { cerr << "Ex1" << endl;}
};

struct Exception2: public Exception {
  void print() const { cerr << "Ex2" <<endl;}
};

struct Exception3: public Exception {
  void print() const { cerr << "Ex3" <<endl;}
};
```

Exception et Héritage

Catch et exceptions héritées :

Catch Exception : heritage

```
try {
   ...
} catch (Exception& ex) {
   ...
} catch (Exception1& ex) {

} catch (...) {
}
```

Exception et Héritage

Pour attraper les exceptions héritées, l'ordre importe :

Catch Exception : heritage Corrige

```
try {
   ...
} catch (Exception1& ex) {
   ...
} catch (Exception2& ex) {
   ...
} catch (Exception& ex) {
} catch (...) {
}
```

Regle sur les exception

- catch par référence
- l'ordre des catch importe
- Exception dans les destructeurs et les constructeurs : cas particuliers.

Spécification des exceptions

On peut spécifier les exceptions lancer par une fonction (non obligatoire et non restrictif :

Code : Spécification exceptions

```
void f (...) throw(); // pas d'exception
void g (...) throw(out_of_bound...); //
```

Ceci est à éviter.

Log

- Element relativement facile a écrire
- Element Critique : permet de détailler les étapes (penser aux assertions)
- Beaucoup de rigueur

Ecriture

- test unitaire (plus de détails dans le module 5)
- vérification mémoire (module 2)
- écriture de la documentation publique (doxygen) : description de l'API
- écriture de la documentation privée.

Maintenance

- Phase de design
- Phase d'écriture
- Contraintes

Contraintes

- Compatibilite API
- Compatibilite ABI

Compatibilite API

- modification des méthodes existantes : garder le comportement d'origine ⇔ test unitaire
- Ajout de nouvelle méthodes, classes. . . ⇔ respecter les conventions de l'API
- Ne pas supprimer les anciennes méthodes (marquer as deprecated)

Compatibilite ABI

Il est possible de :

- Ajouter de nouvelle fonctions non virtuelle
- Ajouter une énumération a une classe
- Ajouter des éléménts à une énumération
- Ajouter d'ajouter de nouvelles fonction statique
- Ajouter des classes

Compatibilite ABI II

Il est interdit de :

- Ajouter de nouvelles fonctions virtuelles
- changer l'ordre des fonctions virtuelles
- changer la signature d'une fonction
- changer la visibilité
- Ajouter ou changer l'ordre des attributs

Workaround ABI

- utiliser le pimpl idiom
- autres.

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Idioms ?

- Blocs de code propre à un langage
- Certains idioms sont déjà vus dans les parties précédentes.
  - *virtual print* (plus généralement *virtual friend*)
  - operateur hérité
- L'idiom du smart pointer sera vu dans le module 2.

Non Copyable mixin

Empêcher la copie et l'égalite.

Non Copyable Mixin

```
template <class T>
class NonCopyable {
  protected:
  NonCopyable () {}
  ~NonCopyable () {}
  private:
  NonCopyable (const NonCopyable &);
  T & operator = (const T &);
};
class Incopiable: private NonCopyable<Incopiable> {};
```

Pimpl

Découpler l'interface et l'implémentation.

Code : Pimpl idiom (Basique)

```
.h:
class PimplImpl;
class Pimpl {
  PimplImpl* impl;
  public:
    Pimpl();
    virtual ~Pimpl();
    void print();
    Pimpl();
};
```

```
.cc:
class PimplImpl{
  public:
    PimplImpl() {}
    virtual ~PimplImpl() {}
    void print() {cout << "impl" << endl;}
};

Pimpl::Pimpl(): impl(new PimplImpl) {}
void Pimpl::print() {impl->print();}
```

Avantages : compilation, ABI et d'API
Inconvénients : indirections.
Héritage possible.

Execute Around

Effectuer une action à chaque appel d'une méthode sur un objet.

Execute Around

Code : Execute around Non Template

```
typedef vector<int> vint;

struct Op {
   virtual void operator()(vint* t) const =0;
};

struct PrintSize: public Op {
  void operator()(vint* t) const {
    cout << "Size: " << t->size() << endl;
  }
};

struct ReinsertLast: public Op {
  void operator()(vint* t) const {
    t->push_back(t->back());
  }
};

class AroundElement {
 public:
  class proxy {
    vint * element;
    const Op& before;
    const Op& after;
 public:
 proxy (vint *t, const Op& _before,
        const Op& _after) :
   element (t), before(_before),after(_after) {
     before(element);
 }

 vint* operator -> () { return element; }
 ~proxy () {after(element);}
 };

 AroundElement(vint *t,
   const Op& before, const Op& after):
 element(t), _before(before), _after(after) {}

 proxy operator -> () {
   return proxy(element,_before,_after);
 }
 private:
   vint* element;
   const Op& _before;
   const Op& _after;
 };
```

Execute around pointer : Utilisation

Code : Utilisation non template

```
AroundElement vect (new vector<int>, PrintSize(), ReinsertLast());
vect->push_back (10); // Note use of -> operator instead of . operator
vect->push_back (20);
vect->push_back (40);
```

Execute Around

Effectuer une action à chaque appel d'une méthode sur un objet.

Code : Execute around

```
template<class T>
struct PrintSize {
  void operator()(T* t) {
    cout << "Size: " << t->size() << endl;
  }
};

template<class T>
struct ReinsertLast {
  void operator()(T* t) {
    t->push_back(t->back());
  }
};

template<class T, template<class U> class Before,
 template<class U> class After>
class AroundElement {
 public:
    class proxy {
    public:
    proxy (T *t) : element (t) {
      Before<T>()(element);
    }
    T* operator ->() {
      return element;
    }
    ~proxy () {After<T>()(element);}
    private:
    T * element;
  };

  AroundElement(T *t) : element(t) {}
  proxy operator ->() {
     return proxy (element);
  }
 private:
  T* element;
};
```

Execute around pointer : Utilisation

Code : utilisation

```
AroundElement<vector<int>,PrintSize, ReinsertLast> vect (new vector<int>);
vect->push_back (10); // Note use of -> operator instead of . operator
vect->push_back (20);
vect->push_back (40);
```

Non Throwing swap

Etre sur que l'échange de deux variables ne lève pas d'exception : utilise le Pimpl Idiom.

Non Throwing Swap

```
class A {
  AImpl *impl // <-- note :implementation
  public :
    void swap(A& other) { std::swap(impl,a.impl);}
};
namespace std { template<> void swap(A& a1, A& a2) {a1.swap(a2);}}
```

Swap and Copy

Garantie que l'affectation a lieu de manière atomique et sans fuite mémoire : utilise le Non-throwing swap.

Swap and Copy

```
struct A {
  A& operator=(const A& other) {
    A tmp(other);
    tmp.swap(*this);
    return *this;
  }
};
```

Plan

1. Présentation de la formation
2. C++ vs C
3. Programmation Orientée Objet
4. Template
5. Visibilité - Durée de vie
6. Surcharges Opérateur
7. Design
8. Idioms
9. Divers

Divers I

Macro : usage réduit en C++.

test

utilisation	C	C++
constante	#define MAX 125	const int MAX=125;

fonction inline	#define max(a,b)\
	 (a)<(b)?(b):(a)
		inline int
		 max(int a, int b){
		 return a<b?b:a;
		 }

genericite	#define max(a,b) \
	 (a)<(b)?(b):(a)
		template<class T>
		 T max(T a, T b){
		 return a<b?b:a;
		 }

En C++, macros == automatisation de code

Macro

Code : Macro

```
#define max(a,b) a<b?b:a
max(a++,b)
///////
a++<b?b:a++
```

Divers II

- Cast : utiliser les `*_cast<type>()` au lieu de `(type)`
- Header : header C en C++ : `<cNom>` : `#include <cmath>` au lieu de `#include <math.h>`
- Utilisation de `std::vector` pour les tableaux
- Surcharge : décoration de nom pou l'édition des liens :
  void f(int,int) ⇒ fii(int, int) ;
  Appel de fonction C et export C : `extern "C"`