Un petit tour du C++

(la mise à jour de ce cours s'est arrêtée en 1998)

Didier Donsez

IMA – IMAG/LSR/ADELE

Didier.Donsez@imag.fr, Didier.Donsez@ieee.org

http://www-adele.imag.fr/~donsez/cours

Combinaison de fonctionnalités

langage C

-> Performances

programmation système

langage ADA

Concept de Généricité et de Modularité

Surcharge

langage OO

Héritage Statique (simple et multiple)

Spécialisation - Généralisation - Abstraction

Smalltalk, ObjLisp, Eiffel, Objective-C, Java, C# … et bien sûr C++

la syntaxe de base : le langage C

syntaxte classique

type de base

(`char, int, long, float, double, unsigned` ....)

type constructeur

(`array [], struct, union`)

opérateurs, expressions, blocs, structures de contrôle

(`if, switch, while, for`)

fonctions

(bibliothéque d'entrée/sortie)

Extension du langage C

référence &

nommage multiple d'un objet

```
int a;
int& abis = a;
```

simplification d'écriture pour paramêtres par variable

```
int inc_ptr(int* i) { return ++(*i); }
int b,c;
c = inc_ptr(&b);
int inc_ref(int& i) { return ++i; }
c = inc_ref(b) ;
```

surcharge de fonctions

```
void print( int ) { ... };
void print( double ) { ... };
void print( char* ) { ... };
```

fonction inline

```
inline int increment(int& a) { return ++a; }
#define INCREMENT(A)  (++(A))
```

Apport de langage OO : les classes d'objets

encapsulation des membres, manipulation au travers des méthodes.

```
class Entreprise {
    // membres
  int nbEmployes
  Employe* listePersonnel[MAX_Pers];
public:
    // constructeur/destructeur
  Entreprise();
  Entreprise(Entreprise*);
  ~Entreprise();
private:
    // méthodes
  void embauche(Employe*, int sal=SAL_SMIC)
  void debauche(Employe*);
  void debauche(Employe*, int primeDepart); //surcharge
public:
    // méthodes d'opérateur
  Entreprise& operator += (Employe* e)
  inline Entreprise& operator -= (Employe* e)
                    { debauche(e); return *this; }
}
```

Déclarations Imbriquées

```
class List {
  class Element { public: ...
    give(Obj& o);
  };
public:
  insert(Obj& o);
};

class List {
  class Element { public: ...
    alloc(Obj& o);
  };
public:
  insert(Obj& o);
};
Array::Element::give(Obj& o) { ... }
List::Element::alloc(Obj& o) { ... }

class Process { public:
   enum Status { IDLE, RUNNING };
    Status st;
    ...
};
schedule() {
    Process::Status st = IDLE;
    ...
}
```

Apport de langage OO : manipulation des objets

```
main() {
  Entreprise IBM; // constructeur 1
  Entreprise* Bell = new Entreprise; // ex-malloc + constructeur 1

  Employe durant;
  Employe* p_durant = &durant
  Employe martin;

  IBM.embauche(p_durant);
  IBM -= p_durant;
  Bell->embauche(p_durant);
  (&Bell) += &martin;

  Entreprise ATT(Bell); // constructeur 2
                        // instanciation au milieu du code
  delete Bell; // desallocateur + destructeur
  ATT -= p_durant; //
} // en sortie de bloc
  // destructeur ~Entreprise() pour IBM, ATT
  // destructeur ~Employe() pour durant, martin
```

Constructeurs, Destructeur

Constructeurs MyClass(...)

MyClass() est généré par défaut mais il peut être surchargé

Il est invoqué lors de la déclaration des variables locales (et paramêtres) et par l'opérateur MyClass::new()

NB : MyClass(MyClass&) peut être invoqué lors d'une affectation

Destructeur ~MyClass()

généré par défaut mais il peut être surchargé une seule fois

Il est invoqué sur les variables locales (et paramêtres) à la sortie d'un bloc (ou de la fonction) et par l'opérateur MyClass::delete()

Allocateur, Désallocateur

Allocateur operator MyClass::new(...)

généré par défaut mais il peut être surchargé

Allocateur operator ::new()

l'opérateur ::new() est l'allocateur globale

il existe par défaut mais il peut être surchargé

Désallocateur operator MyClass::delete()

généré par défaut mais il peut être surchargé

Désallocateur operator ::delete()

l'opérateur ::delete() est l'allocateur globale

il existe par défaut mais il peut être surchargé

Constructeurs, Destructeur, Allocateur, Désallocateur (Exemple)

```
class Str { public:
  char* cs;
  Str()          { cs=(char*)NIL; };
  Str(char* cs) {
    this->cs=new char[strlen(cs)+1]; strcpy(this->cs,cs);};
  Str(const Str& s){
    if(this->cs!=(char*)NIL) delete this->cs;
    this->cs=new char[strlen(s.cs)+1]; strcpy(this->cs,cs.s);};
  ~Str()         { if(this->cs!=(char*)NIL) delete this->cs; };
};

Str& func(Str p){        // Str::Str(const Str&) pour p
  Str v1;                // Str::Str() pour v1
  ...
  {
    Str v2(v1);          // Str::Str(const Str&) pour v2
    Str v3 = v2;         // Str::Str(const Str&) pour v3
    ...
  }                      // Str::~Str pour v3, v2
  ...
}                        // Str::~Str pour v1, p
int main(){    Str m1("toto");    // Str::Str(char*) pour m1
               Str m2 = func(m1); // Str::Str(const Str&) pour m2
}
```

Opérateurs

Définition des Opérateurs

méthodes appelées avec une forme syntaxique algébrique

```
class Obj;

class ListObj { public: ...
              ListObj();
              ListObj(const ListObj&);
  ListObj&   operator =(const ListObj&);
  ListObj&   operator +=(const ListObj&);
  ListObj&   operator +=(const Obj&);
  const Obj& operator [](long&);
};

ListObj& operator + (const ListObj&, const ListObj& );
```

Utilisation des Opérateurs

```
ListObj l2, l3;
ListObj   l1(l3);  //    ListObj(const ListObj&);
                   // vs ListObj();l1.operator=(l3);
l1 += l2;    // l1.operator+=( l2 )
l1 += l3[0];   // l1.operator+=( l3.operator[](0) )
l1 = l2 + l3;  // l1.operator=( operator+(l2,l3) );
```

Héritages Simple et Multiple (les Constructeurs)

```
class Person { public:
       char* nom;
     Person(char* nom) {
       this->nom = new char[strlen(nom)+1];
       strcpy(this->nom, nom);
    };
   }
class Etudiant : public Person { public:
       char* etude;
     Etudiant (char* nom, char* etu) : Person(nom) {
       etude = new char[strlen(nom)+1];
       strcpy( etude, etu);
     };
   }
class Employe : public Person { public:
       int echelon;
     Employe(char* nom, int ech) : Person(nom), echelon(ech) {};
   }
class Etudiant_en_FormCont: public Etudiant, virtual public Employe {
    public:
     Etudiant_en_FormCont(char* nom, char* etu, int ech)
       : Etudiant(nom,etu), Employe(nom, ech) {};
   }
```

Méthodes Virtuelles
Méthodes non Virtuelles

```
class Person { public:
    char* nom;
    virtual   printVIRT() { cout << nom;};
              print()     { cout << nom;};
};
class Employe : public Person { public:
    int echelon;
    virtual printVIRT() { Person::printVIRT(); cout << echelon; };
            print()     { Person::print();     cout << echelon; };
            print2()       { Employe::print(); };
};

f(){
      Employe Bill;
      Employe e = & Bill;
      Person  p = e;     // Person p = & Bill;

      e->printVIRT();    // Employe::printVIRT()
      e->print();           // Employe::print()
      e->print2();          // Employe::print2()
      p->printVIRT();    // Employe::printVIRT() (abstraction)
      p->print();           // Person::print()
      p->print2();          // Erreur de Compilation
}
```

Surcharge d'opérateurs

Opérateurs +=, -=

Utilisé pour alléger la notation (opération sur les collections, …)

Opérateurs <<, >>

Utilisé pour alléger la notation (opération sur les streams, …)

Opérateur de déférenciation ->

Utilisé pour intercepter les accès à la mémoire (implantation de la persistance, du contrôle de concurrence, …)

Héritage Multiple et Méthodes

```
  class Person { public:
      char* nom;
    virtual print() { cout << nom;};
  }
class Employe : public Person { public:
      int echelon;
    virtual print() {            // spécialisation
            Person::print();
            cout << echelon;
   };
  }
class Etudiant : public Person { public:
      char* etude;
    virtual print() {            // spécialisation
            Person::print();
            cout << etude;
   };
  }
class Etudiant_en_FormCont: public Etudiant, virtual public Employe {
   public:
    virtual print() {     Etudiant::print();
                        Employe::print();
   };
```

Classes Abstraites (pure virtual)

pas d'instanciation possible pour ces classes

```
class Figure { public:
  virtual void draw()=0;
};
class Circle : public Figure {
  virtual void draw() { ... };
};

Figure fig;     // KO
Circle circle;  // OK
```

Remarque: définition optionelle de **`Figure::draw()`**

Contrôle d'Acces (I - Définition)

Private (défaut)

seulement
- méthodes membres
- initialiseurs de la classe
- méthodes **friend**

Protected

en plus
- méthodes membres d'une classe dérivée

Public

- toutes méthodes ou fonctions

Contrôle d'Acces (II - Utilisation)

```
class Base {
  private:
    int   a;
  protected:
    int   b();
  public:
    int   c;
  private:
    int   g();
};
```

```
class Derive : public Base {
    int   d();
  protected:
    int   e;
  public:
    int   f;
};
```

```
Base::g() {
  a++;  // OK
  b();  // OK
  c++;  // OK
}
```

```
Derive::g() {
  a++;  // OK
  b();  // OK
  c++;  // OK

  d();  // OK
  e++;  // OK
  f++;  // OK
}
```

```
f(Base &bs, Derive&
dr){
  bs.a++;  // Error
  bs.b();  // Error
  bs.c++;  // OK

  dr.d();  // Error
  dr.e++;  // Error
  dr.f++;  // OK
}
```

Contrôle d'Acces (III - Heritage)

```
class Base {
  private:   int   a;
  protected: int   b();
  public:    int   c;
  private:   int   g();
};
```

```
class Derive
   : XXXXX Base {
             int   d();
  protected  int   e;
  public:    int   f;
};
```

```
class DD : Derive {
  fDD(Derive&);
};
```

```
XXXXX = public

DD::fDD(Derive& dr){
  dr.a++;  // Error
  dr.b();  // OK
  dr.c++;  // OK
}
f(Derive& dr){
  dr.a++;  // Error
  dr.b();  // Error
  dr.c++;  // OK
}
```

la classe Derive rend public le contrôle d'accès de sa super-classe à tous

```
  XXXXX = protected

DD::fDD(Derive& dr){
  dr.a++;  // Error
  dr.b();  // OK
  dr.c++;  // OK
}
f(Derive& dr){
  dr.a++;  // Error
  dr.b();  // Error
  dr.c++;  // Error
}
```

la classe Derive rend public le contrôle d'accès de sa super-classe uniquement à ses sous classes et aux friends

```
  XXXXX = private

DD::fDD(Derive& dr){
  dr.a++;  // Error
  dr.b();  // Error
  dr.c++;  // Error
}
f(Derive& dr){
  dr.a++;  // Error
  dr.b();  // Error
  dr.c++;  // Error
}
```

la classe Derive ne rend pas public le contrôle d'accès de sa super-classe

Contrôle d'Acces (IV - Friend)

Friend

fonction "amie"

classe "amie"

autorise les accès private et protected aux fonctions non membres

```
class Matrix {
  Vector& Diagonal();
};

class Vector { private:
  friend class Matrix;
  friend Matrix& multiple(Matrix&,Vector&);

  int size;
};

Vector& Matrix::Diagonal() { .. = size; }
Matrix& multiple(Matrix& m,Vector& v) { .. = size; }
```

Const

```
class Obj { friend g(const Obj&);
  int i;
public:
  f();
  f_CONST() const;
};

g(const Obj* o) {
  cout << o->i;   // OK
  o->i++;         // Erreur
  ((Obj*)o)->i++;    // OK
}

main() {
       Obj*  o    = new Obj;
 const Obj*  oc   = o;          // OK
       Obj*  onc  = oc;         // Erreur
       Obj*  onc2 = (Obj*)o;       // OK

       o->f();         // Erreur
       o->f_CONST();   // OK
}
```

Généricité : classes paramétrées

Déclaration

```
template <class T> class List {
    struct Element { public:
       T*        value;
       Element*  next;
      };

     Element* first;

  public:
            List<T>();
            List<T>(const List<T>&);
  List<T>&  operator =(const List<T>&);
  List<T>&  operator +=(const List<T>&);
  List<T>&  operator +=(const T&);
  const T&  operator [](long&);
};
List<T>& operator + (const List<T>&, const List<T>& );
```

Utilisation

```
List<int> li;       li += 1;       cout << li[0];
List<double> ld;    ld += 3.1416;
List<Employe> mcrosft;     mcrosft += new Employe("Bill");
List<Person>  wldcny;      wldcny  += mcrosft;
```

Exceptions C++

Traitement des erreurs dans un langage séquentiel

```
err_t& fn( res_t& r ) {
  ...
  if(...){
    r = result; return OK; }
  else          return KO;
}

for(i=0; (err=fn(i))==OK && i<MAX; i++ ) {}
if(err==KO) { traitement d'erreur }
```

Execptions C++:

mécanisme alternatif de traitement des erreurs “synchrones”

- simplification du code de l’application
  notanment au niveau des constructeurs
- efficacité
  parfois si MAX grand
- récupération d’une erreur “levée” dans une fonction de bibliothèque

Exceptions C++ : Mécanisme

Définition

```
class MathError{};
class DivideZero : MathError {};
class Overflow   : MathError { char* _str; Overflow(char* str); };
double add(double d1; double d2) {
 ...
 if(...) throw Overflow("+");
}
f(){
      try {
        add(maxdouble, maxdouble); // raise Overflow
      }
      catch(DivideZero) {
        // traitement pour DivideZero
      }
}
g(){
      try {
        f();
      }
      catch(Overflow& o) {
        cerr << "Overflow sur " << o._str; exit();
      }
      catch(MathError) {
        // traitement pour MathError et autres dérivés
      }}
```

Manques dans le C++ (en 1998)

- Espaces de nommage (Namespace)
- Méta Modèle
- Dynamic Binding
- Multi Activité (thread Java ou task ADA)
- Ramasse Miette
- Une bibliothèque standard

les Ramasses Miette (Garbage Collector)

C++:

Destruction explicite par l'opérateur de désallocation delete()

Smalltalk, Java, C# :

Destruction implicite

quand l'instance n'est plus référencée par aucune autre instance

nécessite un "Ramatte Miette"

c.a.d. récupération des instances inaccessibles

Techniques: Compteur de référence, Mark-and-Sweep, Incrémentale, Générationnel ...

bibliothéque C++ de Ramasses Miette

par héritage avec des techniques différentes classes par classes

Ramasses Miette (exemple : Compteur de Référence)

```
template <class CGC> class Ref {
    CGC*        ptr;
  public:
      CGC* operator =(const CGC* t) {
          if(ptr) ptr->decref();
          ptr = t; t->incref();
          return ptr;
      };
  };
class ObjectGC {
    int          cpt;
              ObjectGC() : cpt(0) {}
    virtual  ~ObjectGC() {};
  public:
    void incref()  { cpt++;  };
    void decref()  { if(--cpt==0) delete this;   };
  };

class MyClass : public ObjectGC { ... };

  Ref<MyClass> ref1 = new MyClass;    // cpt==1
  Ref<MyClass> ref2 = ref1;           // cpt==2
               ref1 = 0;              // cpt==1
               ref2 = ref3;           // cpt==0 Destruction effective
```

Méta Classes

Construction dynamique du schéma

Implantation difficile dans une approche compilé

Applications : IA

Dynamic Binding

Ajout des bibliothèques d'objets au Runtime

Dépend de système mais reste compliqué
Objective-C (GNU) utilisé dans NeXTStep, Java dans la JVM

Appel de méthodes distantes

RPC - OO, Corba

Multi Activité

Pas de spécification de Coroutines ou de Taches dans le langage

But : Objets réactifs

Solution: encapsuler des librairies système (pthread POSIX) si le système le permet

Espaces de nommage (NameSpace)

Motivation : résolution des conflits de nommage

2 bibliothèques utilisent le même nom
pour 2 classes aux fonctionnalités différentes

```
// car engine
class Engine {
  public:
    void ignition(){…};
    void stop(){…};
    int  getRunningSpeed(){…};
    char *getFuelType(){…};
    …
};
```

```
// speech synthesis engine
class Engine {
  public:
    void spell(char* sentence){…};
    void stop(){…};
    void setVoice(int gender){…};
    int  getVoice(){…};
    …
};
```

La solution : les espaces de nommage

Espaces de nommage (NameSpace)

La solution

```
namespace car {
// car engine
class Engine {
  public:
    void ignition(){…};
    void stop(){…};
    int  getRunningSpeed(){…};
    char *getFuelType(){…};
    …
};
}
```

```
namespace voicesynthetis {
// speech synthesis engine
class Engine {
  public:
    void spell(char* sentence){…};
    void stop(){…};
    void setVoice(int gender){…};
    int  getVoice(){…};
    …
};
}
```

```
int main() {
  car::Engine e
          =new car::Engine();
  e.ignition();
}
```

```
using namespace voicesynthetis;

int main() {
  Engine e=new Engine();
  e.spell("My car is great");
}
```

RTTI (TODO)

Identification des types au Runtime

Motivation : introspection des classes

Utile aux conteneurs (CCM, …), Gestionnaires de Plugin, …

Conclusion

Gros point fort :

l'efficacité du C

Un sur ensemble de la syntaxe du C

Apprentissage plus facile pour un programmeur C

Migration incrémentale des applications C vers le C++

Extensible via des bibliothèques

Mais malheureusement tardivement « standardisée »

Un standard de fait pour le Génie Logiciel (avec Java et C#)

grand nombre de compilateurs et AGL (public ou payant)

choisi très tôt par l'OMG (Object Management Group - i.e. CORBA)

pour les SGDB - OO

choisi par l'ODMG (Object Database Management Group)

Des héritiers (Java et C#) qui lui font une très forte concurrence

Bibliographie

Ellis et Stroustrup, ***The Annotated C++ Reference Manual***

Addison-Welley Publishing Company

La sainte bible du developpeur C++ (en anglais)

Bjarne Stroustrup, Le langage C++, 1096 pages (2001), Ed. Campus, ISBN : 2744010898

une introduction pédagogique (en anglais et en français)

Bjarne Stroustrup, Le langage C++, Ed Pearson Education, 1100 pages, Mars 2003, ISBN : 2-7440-7003-3

Massini, ***Les langages à Objets***

un bon panorama des langages objets (syntaxte et implantation)

`comp.object` & `comp.std.c++`

les derniers potins, les trucs, les FAQ, des tutoriaux ...

Le site C++ de Bjarne Stroustrup

http://www.research.att.com/~bs/C++.html

Que manque t'il à ce cours ?

La bibliothèque standard, …

Josuttis, Nicolai M., The C++ standard library: a tutorial and reference, Ed Addison Wesley, 1999