Ch. 8 - Classes et objets

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

11 octobre 2021

Table des matières

Introduction

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

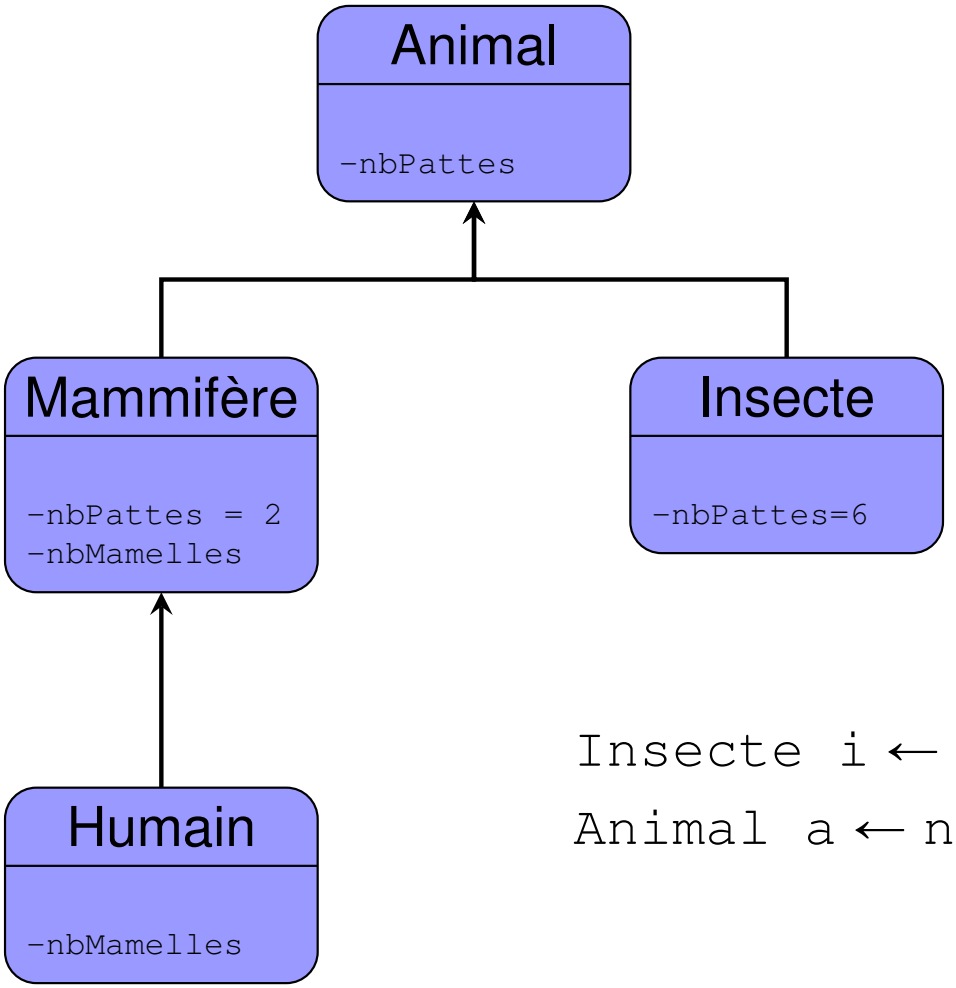
- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type `A` par un objet de type `B`

Exemple




```
Insecte i ← nouvel Insecte
Animal a ← nouvel Insecte
```

Classes

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,
  1. les membres d'une `class` sont privés, ceux d'une `struct` sont publics
  2. les membres d'une mère au sein d'une fille sont privés dans une `class`, publics dans une `struct`

Exemple classe

- Fichier `point-class.cpp`

```
1  class point
2  {
3    double x, y;
4
5    public:
6      point(int x, int y)
7      {
8        this->x = x;
9        this->y = y;
10     }
11
12     inline double getX()
13     {
14       return x;
15     }
16
17     inline double getY()
18     {
19       return y;
20     }
21
22     double dist(point p)
23     {
24       return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
25     }
26 };
```

Exemple classe

- Fichier `point-struct.cpp`

```
1  struct point
2  {
3    point(int x, int y)
4    {
5      this->x = x;
6      this->y = y;
7    }
8
9    inline double getX()
10   {
11     return x;
12   }
13
14   inline double getY()
15   {
16     return y;
17   }
18
19   double dist(point p)
20   {
21     return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
22   }
23
24   private:
25     double x, y;
26 };
```

Utilisation

- Fichiers `point-class.cpp` et `point-struct.cpp`

```
1  int main()
2  {
3    point p1(1,1);
4    //cout << p1.x << " " << p1.y << endl; //ko
5    cout << p1.getX() << " " << p1.getY() << endl;
6    point p2(2,2);
7    cout << p2.getX() << " " << p2.getY() << endl;
8    cout << "dist = " << p1.dist(p2) << endl;
9  }
```

Remarque

- Ne pas oublier le '`;`' après la déclaration d'une classe ou d'une structure

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
  - « Même fichier »
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Exemple (1/2)

- Fichier `point-decl-impl.h`

```
1  //no include, no using namespace std;
2
3  class point
4  {
5    double x, y;
6
7    public:
8      point(int x, int y);
9      inline double getX();
10     inline double getY();
11     double dist(point p);
12 };
13
14 double point::getX()
15 {
16     return x;
17 }
18
19 double point::getY()
20 {
21     return y;
22 }
```

Exemple (2/2)

- Fichier `point-decl-impl.cpp`

```
1  #include "point-decl-impl.h"
2
3  point::point(int x, int y) {
4      this->x = x;
5      this->y = y;
6  }
7
8  double point::dist(point p) {
9      return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
10 }
```

- Fichier `point-decl-impl-main.cpp`

```
1  #include <iostream>
2  #include "point-decl-impl.h"
3
4  using namespace std;
5
6  int main() {
7    point p1(1,1); point p2(2,2);
8    cout << p1.getX() << " " << p1.getY() << endl;
9    cout << p2.getX() << " " << p2.getY() << endl;
10   cout << "dist = " << p1.dist(p2) << endl;
11 }
```

Illustration pour `inline`

- Fichier `inline.h`

```
1  struct A {
2    void f() { //inline
3      cout << "Brol::f" << endl;
4    }
5  };
6
7  struct B {
8    inline void f(); //inline
9  };
10
11 void B::f() {
12   cout << "Foo::f" << endl;//defined in same file
13 }
14
15 inline double sum(double a, double b) { //inline
16   return a + b;
17 }
18
19 struct C {
20   void f();//not inline
21 };
```

- Suite dans `inline.cpp` et `inline-main.cpp`

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Exemple

- Fichier `const-class.cpp`

```
1  class A
2  {
3      int i;
4
5      public:
6          A(int i) { this->i = i; }
7          void set(int i) { this->i = i; }
8          int& get() { cout << "g " ; return i; }
9          const int& get() const { cout << "cg " ; return i; }
10 };
11
12 int main()
13 {
14     A a(2);
15
16     a.set(3);
17     cout << a.get() << endl;
18     a.get() = 5;
19     cout << a.get() << endl;
20
21     const A ca(42);
22     //ca.set(5);
23     cout << ca.get() << endl;
24 }
```

Classes d'allocation des objets

- Souvent, l'adresse d'un objet est l'adresse du premier attribut
- Les attributs non dynamiques et non statiques ont la même classe d'allocation que l'objet
- Les attributs `static` sont statiques
- Attributs dynamiques
  - Données en classe d'allocation dynamique
  - Adresses de même classe d'allocation que l'objet
- Les fonctions membres sont généralement allouées dans le segment de code
  - Pas les fonctions `inline`
- Les remarques en terme de portée et de durée de vie sont valides sous ces conditions

Illustration

- Fichier `class-alloc.cpp`

```
1  struct Array {
2      int i;
3      int * arr;
4
5      Array(int i)
6      {
7        this->i = i;
8        arr = new int[i];
9      }
10 }; //missing sooo many things to make it safe...
11
12 int main() {
13     Array a(2); //a automatic
14                 //i automatic
15                 //tab automatic
16                 //*tab dynamic
17
18     static Array b(2); //b static
19                        //i static
20                        //tab static
21                        //*tab dynamic
22
23     Array * c = new Array(2); //c dynamic
24                               //i dynamic
25                               //tab dynamic
26                               //*tab dynamic
27 }
```

Constructeurs et destructeurs

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 13)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 15)

Exemple

- Fichier `point-cstr.cpp`

```
1  class point {
2    double x, y;
3    bool copie;
4
5    public:
6      point(int x, int y) {
7        this->x = x; this->y = y;
8        copie = false;
9
10       cout << "Construction de " << x << " " << y << endl;
11     }
12
13     point(const point& p) {
14       this-> x = p.x; this-> y = p.y;
15       copie = true;
16
17       cout << "Copie de " << x << " " << y << endl;
18     }
19
20     ~point() {
21       cout << "Destruction de " << x << " " << y;
22       if(copie)
23         cout << " (copie)";
24       cout << endl;
25     }
26 }
```

Exemple (2/2)

- Fichier `point-cstr.cpp`

```
1  void sayHello(point p) { //fct indep, try with &, *
2    cout << "Hello␣Mr␣point␣" << p.getX() << "␣" << p.getY() << endl;
3  }
4
5  int main() {
6    point p1(0,0); point p2(1,1);
7    cout << p1.getX() << "␣" << p1.getY() << endl;
8    sayHello(p1);
9    cout << p2.getX() << "␣" << p2.getY() << endl;
10   cout << "dist␣=␣" << p1.dist(p2) << endl;
11
12   point p3(p1);//explicit copy
13   p3 = p2;
14 }
```

Remarques

- Copies implicites effectuées
- Destructions implicites effectuées
- Affectation muette

Liste d'initialisation

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs
     - Y compris appels superconstructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Exemple

- Fichier `point_init.cpp`

```
1  class point
2  {
3    double x, y;
4
5    public:
6      point(int x = 0, int y = 0) : x(x), y(y) {}
7
8      double getX() const
9      {
10       return x;
11     }
12
13     double getY() const
14     {
15       return y;
16     }
17
18     double dist(point p)
19     {
20       return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
21     }
22 };
```

Délégation

- Fichier `deleg.cpp`

```
1  class A {
2    int i;
3    const int k;
4
5    private:
6      A() : k(5) {
7        cout << "Init " << endl;
8        //k = 5; //ko
9      }
10
11   public:
12     A(int x) : A(), i(x) {
13       //i = x; //ok
14     }
15
16     void print() { cout << "A : " << i << endl;}
17 };
18
19 struct B : A {
20     B() : A(7), j(3) {}
21
22     B(int  i, int j) : A(i), j(j) {}
23
24     private:
25         int j;
26 }
```

Absence de constructeur par défaut

- Fichier `no-cstr.cpp`

```
1  struct A
2  {
3    int i;
4    A(int i) : i(i) {}
5  };
6
7  struct B
8  {
9    A a;
10   B(A a) : a(a) {};//ok
11
12   //B(A a)//ko
13   //{
14   //  this -> a = a;
15   //}
16 };
17
18 int main()
19 {
20   A a(2);
21   B b(a);
22 }
```

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Exemple

- Fichier `dataset.cpp`

```
1  class DataSet
2  {
3    double sum;
4    int count;
5
6    public:
7      DataSet() : sum(0), count(0) {}
8      DataSet(const initializer_list<double>& data) : DataSet() { update(data); }
9
10     void update(const initializer_list<double>& data)
11     {
12       for(double d : data)
13       {
14         update(d);
15       }
16     }
17
18     inline void update(double d)
19     {
20       sum += d;
21       count++;
22     }
23
24     inline double mean() const { return sum / count; }
25 };
```

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Constructeur par défaut

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Utilité

- Fichier `delete-cstr.cpp`

```
1  struct A { int i; };
2  struct AD
3  {
4      int i;
5      AD() = delete;
6  };
7
8  int main()
9  {
10     A a1;      //i not init
11     A a2{};    //i = 0
12     A a3{42}; //i = 42
13
14     //AD ad;
15     AD ad{};
16     AD ad2{42};
17 }
```

Appels (1/2)

- Fichier `call-cstr.cpp`

```
1  struct A {
2      int x;
3      A(int x = 1): x(x) {} // user-defined default constructor
4  };
5
6  struct B : A {}; // B::B() implicitly-defined, calls A::A()
7
8  struct C {
9      A a;
10 }; // C::C() implicitly-defined, calls A::A()
11
12 struct D: A {
13     D(int y): A(y) {}
14 }; // D::D() not declared
15
16 struct E: A {
17     E(int y): A(y) {}
18     E() = default; // explicitly defaulted, calls A::A()
19 };
20
21 struct F {
22     int& ref;
23     const int c;
24 }; // F::F() is implicitly defined as deleted
```

Appels (2/2)

- Fichier `call-cstr.cpp`

```
1  int main()
2  {
3      A a;
4      B b;
5      C c;
6  //  D d;
7      E e;
8  //  F f;
9  }
```

Constructeur de recopie

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Exemple (1/2)

- Fichier `call-copy.cpp`

```
1  struct A {
2      int n;
3      A(int n = 1) : n(n) { }
4      A(const A& a) : n(a.n) { }
5  };
6
7  void f1(A a) {}
8  void f2(A& a) {}
9
10 A f3()
11 {
12     A a;
13     return a;
14 }
15
16 int main()
17 {
18     A a1(7);
19     A a2(a1);   // copy
20     A a3 = a2; //copy
21
22     f1(a1); //copy
23     f2(a1);
24     A a4 = f3();
25 }
```

Exemple (2/2)

- Fichier `call-copy.cpp`

```
1  struct B
2  {
3      B();
4      B(const B&) = delete;
5  };
6
7  void f4(B b) {}
8  void f5(B& b) {}
9
10 /*
11 B f6()
12 {
13     B b;
14     return b;
15 }*/
16
17 int main()
18 {
19     B b;
20     //f4(b);
21     f5(b);
22
23     //B b2 = f6();
24 }
```

Destructeur

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Suppression de destructeur

- On peut empêcher la génération d'un destructeur avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être détruits
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non destructible
  - `T` hérite d'une classe ayant un destructeur `delete`

Exemple

- Fichier `destr.cpp`

```
1  void open_f(const string& path) { cout << "Opening " << path << endl;   }
2
3  void flush_f(const string& path) {  cout << "Flushing " << path << endl;   }
4
5  void close_f(const string& path) {  cout << "Closing " << path << endl;   }
6
7  class InputFileStream {
8      const string& path;
9        public:
10           InputFileStream(const string& path) : path(path) {
11               open_f(path);
12           }
13
14           ~InputFileStream()  {
15               flush_f(path);
16               close_f(path);
17           }
18 };
19
20 int main()  {
21     InputFileStream("brol.txt");
22 }//désallocation implicite
```

- Autres exemples au chapitre 5