Programmation en C

14. LE LANGAGE C++

IETA Pascal CANTOT
DGA / Centre d'Analyse de Défense / Méthodes de Simulation
14. LE LANGAGE C++

Le langage C++

ATTENTION

Le C++ n'est pas seulement un surensemble du C, mais aussi un langage complètement différent dans ses concepts et son approche de la programmation !

14. LE LANGAGE C++

• On voit souvent dans le C++ un C avec l'encapsulation des données et fonctions et quelques ajouts (dont les commentaires //, très important!).

• C'est une vision trop restrictive. Le C++ est un langage à part entière, avec une philosophie très différente.

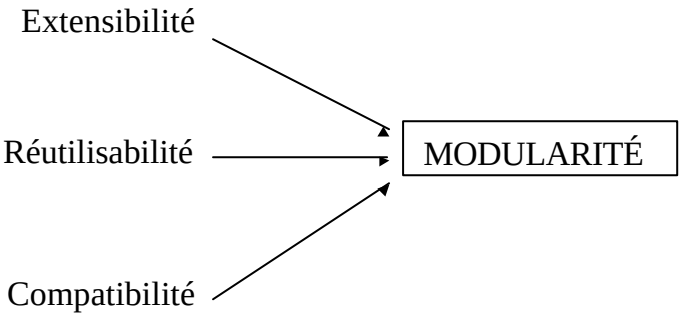
• Les programmeurs qui passent du C au C++ sont fortement déroutés, et doivent revoir leur schéma de pensée, ce qui n'est pas toujours évident. Quand ils n'y arrivent pas, le résultat de leur codage n'est pas très joli à voir...

• En outre, la complexité des mécanismes mis en œuvre par la POO demande une certaine discipline. Les projets en C++ connaissent un taux d'échec important, essentiellement dû à des problèmes de gestion de la mémoire (comme en C, voire pire en raison de son automatisation partielle en C++), mais aussi à un manque de maîtrise de l'arborescence des classes. L'héritage c'est bien, mais il faut rester simple et clair !

• Bref, le C++ est un bon langage, mais beaucoup plus difficile à maîtriser qu'il n'y paraît. Probablement aussi difficile qu'Ada95, et certainement plus que Java.

• NOTE : Voir la section 1.2 du polycopié sur C++ traitant des incompatibilités entre le C et le C++.

Nécessité de modularité



14. LE LANGAGE C++

• La complexité sans cesse croissante des développements de logiciels a fini par trouver les limites de la programmation structurée.

• Il fallait alors, pour assurer maîtriser les coûts de développement et de maintenance trouver une nouvel approche qui pousse encore plus loin ce concept de modularité.

Principes à respecter

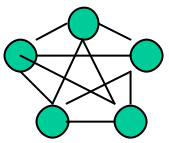
(d'après Bertrand Meyer)

- Unités linguistiques modulaires
- peu d'interfaces
- petites interfaces (couplage faible)
- Interfaces explicites
- Masquage de l'information

14. LE LANGAGE C++

• **Les modules doivent correspondre à des unités syntaxiques du langage**, c'est-à-dire que le langage doit supporter la modularité. C'est par exemple le cas d'Ada, mais pas de FORTRAN.

• L'interface doit être minimale, c'est-à-dire qu'**un module doit communiquer avec aussi peu d'autres que possible**. Ce qu'il faut notamment éviter :



•**Si deux modules communiquent, ils doivent échanger aussi peu d'information que possible** : un module n'exporte que le strict nécessaire. En effet, cela limite non seulement la complexité de l'interface pour le programmeur qui réutilisera le module, mais également les risques d'interactions non désirées.

• **Chaque fois que deux modules A et B communiquent, cela doit ressortir clairement du texte de A, de B ou des deux**. Autrement dit, toute connexion avec l'extérieur doit être déclarée (par exemple A pourra dire "j'utilise B").

• **Toute information concernant un module doit être privée, sauf si elle est explicitement déclarée publique**. Ceci rejoint le principe de limiter l'interface au strict nécessaire : tout ce qui n'est pas indispensable à une utilisation du module depuis l'extérieur doit être caché.

Réutilisabilité

- La modularité favorise la réutilisabilité mais ne suffit pas
- Il faut pouvoir adapter une solution à de nouveaux types de données
- Il faut pouvoir éventuellement étendre les capacités du module

14. LE LANGAGE C++

• Rappel : réutilisabilité = aptitude d'un logiciel à être réutilisé en tout ou partie pour une autre application.

• Il s'agit d'éviter de développer N fois le même composant logiciel, voire à récupérer un tel composant à l'extérieur.

• La politique du "pas développé ici" est coûteuse et souvent non fondée.

• Dans l'idéal, un module doit pouvoir évoluer, pour s'adapter à de nouveaux besoins. Par exemple, une routine de tri sur des entiers devrait pouvoir facilement s'adapter à des réels. Autre exemple: vous avez développé un module de gestion d'arbre binaire, mais vous n'avez pas prévu l'enregistrement sur disque de l'arbre. Or, quelques mois plus tard, la fonction est requise. Il pourrait être intéressant de rajouter la fonctionnalité sans modifier le composant d'origine, qui a été validé et diffusé dans une bibliothèque logicielle.

L'approche "objets"

- Programmation fonctionnelle : on se focalise sur les actions que réalise le système.
- Programmation objet : on base l'architecture du système sur les objets qu'il manipule.

⇨ On ne doit pas commencer par analyser ce que fait un système, mais à quoi il le fait !

14. LE LANGAGE C++

• La programmation orientée objets est avant tout une méthodologie de conception de système.

• Il est possible de faire de la POO dans n'importe quel langage (j'ai vu un jour un article sur la POO en... assembleur!), mais les langages orientés objets facilitent la mise en application de cette méthodologie.

• Les caractéristiques de base d'un langage à objets sont :

- Encapsulation
- Héritage
- Polymorphisme

• L'approche objets a énormément de qualités qui en font la méthodologie favorite du génie logiciel moderne. Les trois langages les plus en vogue en génie logiciel (C++, Ada95, Java) sont tous les trois orientés objets.

• On adapte même les anciens langages : C et Ada, bien sûr, ont ainsi des versions OO, mais aussi Cobol, Pascal, PERL, etc.

Historique

- 1967 : langage SIMULA pour la simulation
- 1972 : SMALLTALK, la référence
- 1983 : C++ (Bjarne Stroustrup)
- 1988 : EIFFEL (Bertrand Meyer)
- ~1990 : Borland sort une version OO de son très populaire compilateur PASCAL
- 1994 : Introduction de la POO dans Ada95
- 1996 : Déferlante JAVA.

14. LE LANGAGE C++

• La simulation vit apparaître le premier langage à objets (du moins à ma connaissance), ce qui n'est pas étonnant, car c'est un domaine où il s'agit de modéliser les composants d'un système physique : la POO est particulièrement efficace pour cela.

• Dès 1972, c'est la programmation d'interface utilisateur graphique, un autre domaine de prédilection de la POO, qui engendre celui qui deviendra le père de la plupart des langages à objets modernes : SMALLTALK.

• On notera que l'apparition de la POO vient de besoins que les langages structurés ou séquentiels d'alors ne pouvaient remplir de façon satisfaisante.

• Dans les années 80, le langage C, répandu avec le système d'exploitation UNIX, devenait de plus en plus populaire, mais manquait de souplesse pour les gros développements. L'idée d'en écrire une extension permettant la POO fit son chemin, et donna Objective C et C++. C'est ce dernier, supporté par AT&T, qui gagna la compétition et fut adopté, son concurrent se retrouvant marginalisé.

• Ada83, alors la référence dans le domaine du génie logiciel, subit de plein fouet la concurrence de C++, plus familier (beaucoup de gens faisaient du C), moins contraignant. Il est clair que le C++ fut un facteur déterminant dans le "semi-échec" d'Ada... La réaction fut tardive mais efficace: Ada95 permit à Ada de se remettre au goût du jour et de remonter la pente.

• Java est un cas à part dans l'histoire des langages. En à peine 3 ans, il est devenu, fait unique dans les annales de l'informatique, la coqueluche des développeurs. Héritier de C++ et d'Ada95, il a de sérieux atouts pour être le langage privilégié de la prochaine décennie.

Principes de base

- Les langages à objets manipulent des structures appelées **classes**. Ces classes correspondent aux modules.
- Ces classes correspondent à des types. Les **objets** sont des **instances** de ces classes.
- Exemple : BIBLE est une instance de la classe LIVRE.

14. LE LANGAGE C++

Les *Sept Pas vers le Bonheur par les Objets* de B. Meyer :

•**Structure modulaire fondée sur les objets** : Les systèmes sont découpés en modules sur la base de leurs structures de données

• **Abstraction des données** : Les objets doivent être décrits comme des implémentations de types de données abstraits.

• **Gestion automatique de la mémoire** : Les objets inutilisés doivent être désalloués par le système sous-jacent, sans intervention du programmeur (par exemple, si une liste chaînée n'est plus utilisée, il faut libérer l'espace mémoire occupé par tous ses éléments, ce qui n'est pas automatique en C).

• **Classes** : Tout type est un module, et tout module de haut niveau est un type.

• **Héritage** : Une classe peut être définie comme une extension ou une restriction d'une autre classe.

• **Polymorphisme et liaison dynamique** : Une entité de programme doit pouvoir faire référence à des objets de plusieurs classes, et une opération doit pouvoir avoir des versions différentes dans des classes différentes.

• **Héritage multiple et répété** : Il doit être possible de déclarer qu'une classe hérite de plus d'une classe, et plus d'une fois de la même classe.

Encapsulation

- Une classe est décrite par des données (**attributs**) et des traitements s'appliquant à ces données (**méthodes**). Méthodes et attributs sont les **membres** de la classe.
- Exemple :

```
class rectangle
{
    unsigned longueur, largeur;
    unsigned surface();
};
```

14. LE LANGAGE C++

• Le prototype de la classe `rectangle` est :

```
class rectangle
{
    unsigned longueur, largeur;
    unsigned surface();
};
```

• Le corps de la méthode `surface()` pourrait s'écrire :

```
rectangle::surface()
{
    return longueur * largeur;
}
```

• L'instanciation d'un objet de type rectangle et l'appel d'une de ses méthodes serait :

```
...
rectangle toto;
unsigned surf;
toto.longueur = 9;
toto.largeur = 4;
toto.dessiner();
surf = toto.surface();
...
```

Limitations d'accès aux membres

- On peut (doit?) limiter l'accès aux membres d'une classe à ce qui est nécessaire, à l'aide des mots clés **`private`**, **`public`**, **`protected`**.
- Principe : Les attributs sont privés, les méthodes publiques (défaut en C++)
- Utilisation méthodes pour accéder aux données.

14. LE LANGAGE C++

• L'exemple d'utilisation de la classe rectangle qui précède est donc faux, puisque rectangle::longueur et rectangle::largeur sont privés. Il aurait donc fallu écrire :

```
class rectangle
{
  public:
    unsigned longueur, largeur;
    unsigned surface();
};
```

• Une méthode plus élégante consiste à utiliser une méthode d'accès :

```
class rectangle
{
  private: // implicite
    unsigned longueur, largeur;
  public:
    void creer(unsigned longr, unsigned larg)
      { longueur = longr;  largeur = larg; }
    unsigned lit_longueur()
      { return longueur; }
    unsigned lit_largeur();
      { return largeur; }
    unsigned surface();
};
```

• Encore plus élégant : utilisation d'un constructeur à la place de `creer()` !

Constructeur

- Le constructeur d'une classe est une méthode qui est appelée lors de l'instanciation d'un objet de cette classe
- Le constructeur porte le même nom que la classe. Exemple :

```
class rectangle
{
    unsigned longueur, largeur;
    rectangle(unsigned longr, unsigned larg);
    ...
};
```

14. LE LANGAGE C++

- Un constructeur ne renvoie jamais de résultat (le *void* est implicite).
- Notre classe rectangle devient :

```
class rectangle
{
  private: // implicite
    int posx, posy;
    unsigned longueur, largeur;
  public:
    rectangle(unsigned longr, unsigned larg);
    unsigned lit_longueur()
    unsigned lit_largeur();
    unsigned surface();
};

...

rectangle::rectangle(unsigned longr, unsigned larg)
{
   longeur = longr;
   largeur = larg;
}

...

rectangle toto(9,4);          // Instanciation d'un rectangle
```

Destructeur

- Un destructeur est une méthode particulière appelée lors de la désallocation d'un objet.
- Le nom de cette méthode est le même que le constructeur, mais précédé d'un tilde (~).
- Exemple : `~rectangle`
- NOTE: Constructeurs et destructeurs par défaut.

14. LE LANGAGE C++

• De même qu'un constructeur permet de réaliser un certain nombre d'opérations automatiquement lors de la création d'un objet, le destructeur est appelé lors de sa destruction (i.e. en fin de vie de l'objet).

• Dans une classe, un constructeur par défaut est fourni. Il alloue la mémoire nécessaire au stockage des attributs. De même, un destructeur libère cette mémoire.

• Pour la classe rectangle, il n'y a pas lieu d'écrire un destructeur, car la libération de la mémoire peut être faite simplement par le système.

• En revanche, si durant sa vie un objet alloue dynamiquement de la mémoire (exemple : pour créer une liste chaînée), elle ne pourra pas être libérée par le constructeur par défaut. Il faudra donc un destructeur explicite.

• Autre exemple : un objet "fichier" pourra réaliser l'ouverture du fichier dans son constructeur et la fermeture dans son destructeur!

opérateurs **new** et **delete**

- L'opérateur **new** permet de créer des objets dynamiquement. Il est plus puissant que `malloc()`, et est surchargeable.
- L'opérateur **delete** permet de détruire les objets dynamiques créés par **new** (et seulement ceux-là!)
- Exemple :

```
rectangle *rect = new rect(9,4);
resultat = rect->surface();
delete rect;
```

14. LE LANGAGE C++

- **new** crée des objets typés, contrairement à **malloc()**.
- Le constructeur est appelé lors du **new**.
- De même, un **delete** appellera le destructeur de la classe de l'objet.
- Ce sont des opérateurs : en les surchargeant, on peut les adapter à des cas particulièrement complexes.
- Attention : le mécanisme d'allocation mémoire du C++ est disjoint de celui du C. Il ne faut pas, par exemple, libérer avec `free()` un objet créé avec **new**.

Surcharge

- Capacité d'un identificateur à avoir plusieurs significations en fonction du contexte.
- Exemples :

```
int carre(int x);
float carre(float x);
```

- Lors d'un appel à `carre(0.2)`, c'est la seconde fonction qui sera utilisée.

• La surcharge est très pratique, elle symétrise et clarifie le code.

• C'est le type des arguments qui permet de faire la différence entre les différentes versions d'une fonction. Il faut donc que ces types soient clairement différents en C++. Il faut se rappeler que le C (et le C++) n'utilisent pas de typage très fort comme Ada, de sorte que le code ci-dessous provoque une erreur, les deux fonctions apparaissant comme identiques:
distance carre(distance x)

```
typedef float distance;

float carre(float x);
distance carre(distance x);
```

• En C++ deux usages sont particulièrement remarquables : la surcharge des opérateurs et celle d'un constructeur.

• Par exemple, pour une classe Cercle, on peut définir:
Cercle(Point centre, float rayon);
Cercle(Point centre, Point un_point);
Mais Cercle(Point centre, float diametre); ne sera pas acceptée.

• En C++ on peut surcharger les opérateurs. Ainsi, si on définit une classe Matrice, il est possible de définir des opérateurs + et * sur des matrices et d'écrire:

```
Matrice A,B,C;
// Initialisation de A et B ...
C = A + B;
```

Généricité

- Capacité d'un module à être paramétré, par exemple par un type de données.
- Permet de construire une classe ou une fonction traitant un type quelconque.
- Exemple : liste chaînée générique

```
template <class T> class Liste
{
  ...
  Ajouter(T);
  ...
};
```

14. LE LANGAGE C++

• Ada83 a particulièrement bien démontré l'intérêt de la généricité pour la réutilisabilité du code.

• Les *templates* ne figuraient pas dans la première version du C++, mais ont rapidement été rajoutés.

• On peut paramétrer une classe ou une fonction par un type, mais aussi par une valeur. Exemple :

```
template <class T, int n> class Tableau
{
   ...
}
```

Pour créer un tableau de 100 réels :

```
Tableau <double, 100> mon_tableau;
```

• Les templates sont très largement utilisés dans la STL.

Héritage

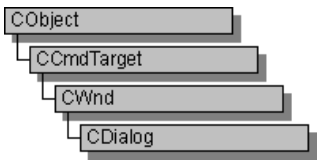
- Le mécanisme d'héritage (ou de dérivation) permet à une classe d'hériter des méthodes et attributs d'une ou plusieurs autres classes.
- Elle peut alors rajouter ses propres attributs ou fonctions.
- "hérite de" doit pouvoir être remplacé par "est un(e)".

14. LE LANGAGE C++

• L'héritage est le mécanisme le plus fondamental de la POO.

• Tout l'intérêt du mécanisme d'héritage réside dans ce que les fonctions et attributs de la classe d'origine sont accessibles à la classe dérivée. Tout ce que l'on a à écrire, c'est le "delta" entre les deux.

• L'héritage permet de particulariser (un carré est un rectangle dont les côtés sont égaux) et/ou d'étendre (une boîte de dialogue est une fenêtre, plus divers éléments de dialogue, plus des boutons "OK" et "Annuler"). Ainsi, sous Windows :



• Quand une classe hérite de plusieurs classes à la fois, on parle d'héritage multiple. Ceci est très délicat parfois à manipuler, car il peut y avoir des conflits de noms. De plus, l'arbre d'héritage devient complexe, avec des branches qui fusionnent. C'est pourquoi l'héritage multiple n'a pas été introduit en Ada95.

• Le constructeur de la classe parent est également hérité. Mais il ne s'occupera que de l'initialisation des attributs de la classe parent. Cela ne dispense pas, si nécessaire, d'écrire un constructeur pour la classe dérivée (voir page suivante).

• En principe, la classe parent est à un niveau d'abstraction (ou de généricité) plus élevé que ses classes dérivées : on part du plus général (ou abstrait) vers le plus particulier.

Exemple d'héritage

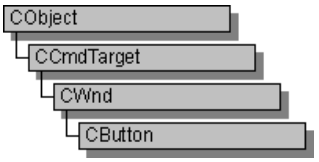
- Soit une classe Fenetre de Windows, zone rectangulaire dans laquelle on peut afficher à volonté du texte.
- Un bouton est une fenêtre avec un texte et une action associée.
- Le fait de faire hériter Bouton de Fenetre va nous éviter la réécriture de plusieurs attributs et fonctions

14. LE LANGAGE C++

• Soit la classe `Fenetre` :

```
class Fenetre
{
    int x,y;
    int larg, haut;
  public:
    Fenetre(int x, int y, int larg, int haut);
    ~Fenetre();
    Afficher(char* texte);
};
```

NOTE: Extrait de l'arbre d'héritage des MFC :



• La classe `Bouton` :

```
class Bouton : public Fenetre
{
    int x,y;
    int larg, haut;
    Action action;
  public:
    Bouton(int x, int y, char *texte, Action une_action);
};
```

• Le constructeur de `Bouton` :

```
Bouton::Bouton(int x, int y, char *texte, Action une_action)
       : Fenetre(x,y, strlen(texte), 1)
{
     action = une_action;
     Afficher(texte);
}
```

Polymorphisme

- Une classe Y dérivant d'une classe X est à la fois Y et X : c'est le **polymorphisme**.
- Exemple : la classe `Bouton` dérivée de `Fenetre` :

```
Bouton bouton;
Fenetre fenetre = bouton;
```

Ce code est correct, puisque qu'un Bouton est aussi une Fenetre !

14. LE LANGAGE C++

• En revanche, lors de l'affectation de `bouton` à `fenetre`, les attributs rajoutés lors de l'héritage ne sont pas pris en compte. Ainsi, `fenetre` n'a pas d'attribut `action`.

• Il est possible de redéfinir une méthode ou un attribut lors de l'héritage. Par exemple, si nous avons une méthode `Surface()` pour un `Rectangle` et que nous créons une classe dérivée `Carre`, nous pouvons écrire une méthode `Carre::Surface()` qui viendra masquer `Rectangle::Surface()`. Mais cette dernière pourra toujours être utilisée par les méthodes de `Carre` (sauf si elle est déclarée virtuelle dans `Rectangle` avec le mot clé **`virtual`**).

• C'est ce principe qui permet à une classe dérivée d'invoquer le constructeur de son parent.

• **Fonctions virtuelles** : si on redéfinit la méthode Afficher() dans Bouton, et que l'on fait :

```
Bouton bouton;
Fenetre fenetre = bouton;
fenetre.Afficher("Bonjour!");
```

On utilisera en fait la méthode `Fenetre::Afficher()` et non Bouton::Afficher() comme on pourrait le désirer. Pour forcer l'utilisation de la méthode la mieux adaptée (celle des classes dérivées) il faudrait déclarer virtuelle la méthode `Fenetre::Afficher()` :

```
virtual Afficher(char* texte);
```

• **Fonctions virtuelles pures** : il se peut que la méthode ne puisse pas être implémentée dans la classe de base. La déclaration suivante l'indique:

```
virtual Afficher(char* texte) = 0;
```

• Une classe ayant au moins une méthode virtuelle pure est dite **abstraite**, et on ne peut pas l'instancier (mais on peut instancier des classes dérivées, pourvu qu'elles fournissent un code pour les méthodes virtuelles pures).

Autres particularités du C++

- Commentaires //
- Type référence
- Paramètres par défaut
- Fonctions *inline*
- Exceptions
- Les flux d'entrée-sortie
- etc.

14. LE LANGAGE C++

• Beaucoup de gens estiment qu'un commentaire délimité comme en C ou en Pascal pose des problèmes de lisibilité. En C++, le marqueur // signifie que tout ce qui suit est un commentaire, jusqu'au prochain saut de ligne.

• En C++, les types références sur une variable permettent d'utiliser les pointeurs de façon transparente. C'est évidemment éminemment dangereux! Les références sont essentiellement utilisées pour le passage de paramètres: Exemple :

```
int incremente(int& x);
```

On appellera la fonction par `incremente(x)` (et non `incremente(&x)`), mais tout se passera comme si on donnait l'adresse.

• On peut préciser des paramètres par défaut aux fonctions :

```
void creer_echiquier(int couleur = BLEU, int taille = 8);
```

`creer_echiquier(ROUGE);` sera équivalent à `creer_echiquier(ROUGE,8);`

• On peut mettre le code des méthodes dans la déclaration de leur prototype:

unsigned surface() { **return** longueur*largeur; }

Cela permet d'optimiser les petites fonctions en économisant un appel, mais cet usage est déconseillé car il mélange spécification et implémentation des méthodes.

• Le C++ admet un mécanisme d'exception, avec `try`, `catch` et `throw`, qui ne sont pas aussi puissants qu'en Ada, mais ouvrent des possibilités intéressantes de traitement d'erreur.

• Les entrées/sorties en C++ peuvent se faire par les anciennes librairies du C, toujours présentes, ou par des classes de flux (*stream*), et des opérateurs de sortie que l'on peut surcharger pour les adapter à de nouveaux types.

Standard Template Library

- La norme du C++ inclut également une bibliothèque de classes très complète :
  - chaînes de caractères
  - entrées / sorties (*streams*)
  - conteneurs et itérateurs (listes, ensembles, queues...)
  - divers algorithmes et utilitaires
  - ...
- C++ montre ainsi son attachement à la réutilisation du code

14. LE LANGAGE C++

(SUITE DE LA PAGE PRÉCÉDENTE)

• Exemple de traitement d'exception pour récupérer une erreur d'allocation mémoire :

```
#include <iostream.h>

int main()
{
    char *buf;
    try
    {
        buf = new char[512];
        if (buf == 0)
            throw "Erreur d'alloc memoire!";
    }
    catch(char *str)
    {
       cout << "Exception: " << str << endl;
    }
    ...
    return 0;
}
```

Les pièges du C++

- Pratiquement les mêmes que le C, plus...
- Complexité du langage!
- **`new`** et **`delete`** peuvent poser les mêmes problèmes que `malloc()` et `free()`
- L'automatisation (obscure...) d'une partie de la gestion de la mémoire
- Complexité potentielle d'un arbre d'héritage
- Héritage multiple
- ...