Programmation Avancée
Paradigmes des langages de programmation

Khaled Barbaria
khaled.barbaria@ensta.org

Faculté des Sciences de Bizerte
Mastère de Recherche en Informatique (MSI1)

2011-2012

Objectifs du cours

Objectifs du cours

- Rappeler les concepts de base de la programmation (programmation procédurale, programmation orientée objets, etc.)
- Introduire/Maîtriser de nouveaux concepts : structures de données génériques, patrons de conception.
- Introduire/Maîtriser le langage C++ (syntaxe, encapsulation, héritage, polymorphisme, templates, exceptions, etc.)
- Introduire/Maîtriser le concept de programmation fonctionnelle (illustration par le langage Erlang)
- Introduire/Maîtriser la programmation concurrente et distribuée

Programme

1. Introduction : paradigmes des langages de programmation
   - Programmation Impérative (structurée, procédurale, orientée objets, générique)
   - Programmation déclarative (programmation fonctionnelle, programmation logique)
2. Rappels sur les concepts avancés du langage C : pointeurs, gestion mémoire, programmation générique en C.
3. Introduction à C++
4. Concepts basiques de la programmation orientée objets avec C++ (encapsulation, héritage, interface, polymorphisme, etc.)
5. C++ avancé : STL et programmation générique
6. Concept de programmation fonctionnelle avec le langage Erlang
7. Programmation concurrente et distribuée : approche fonctionnelle et approche orientée objets

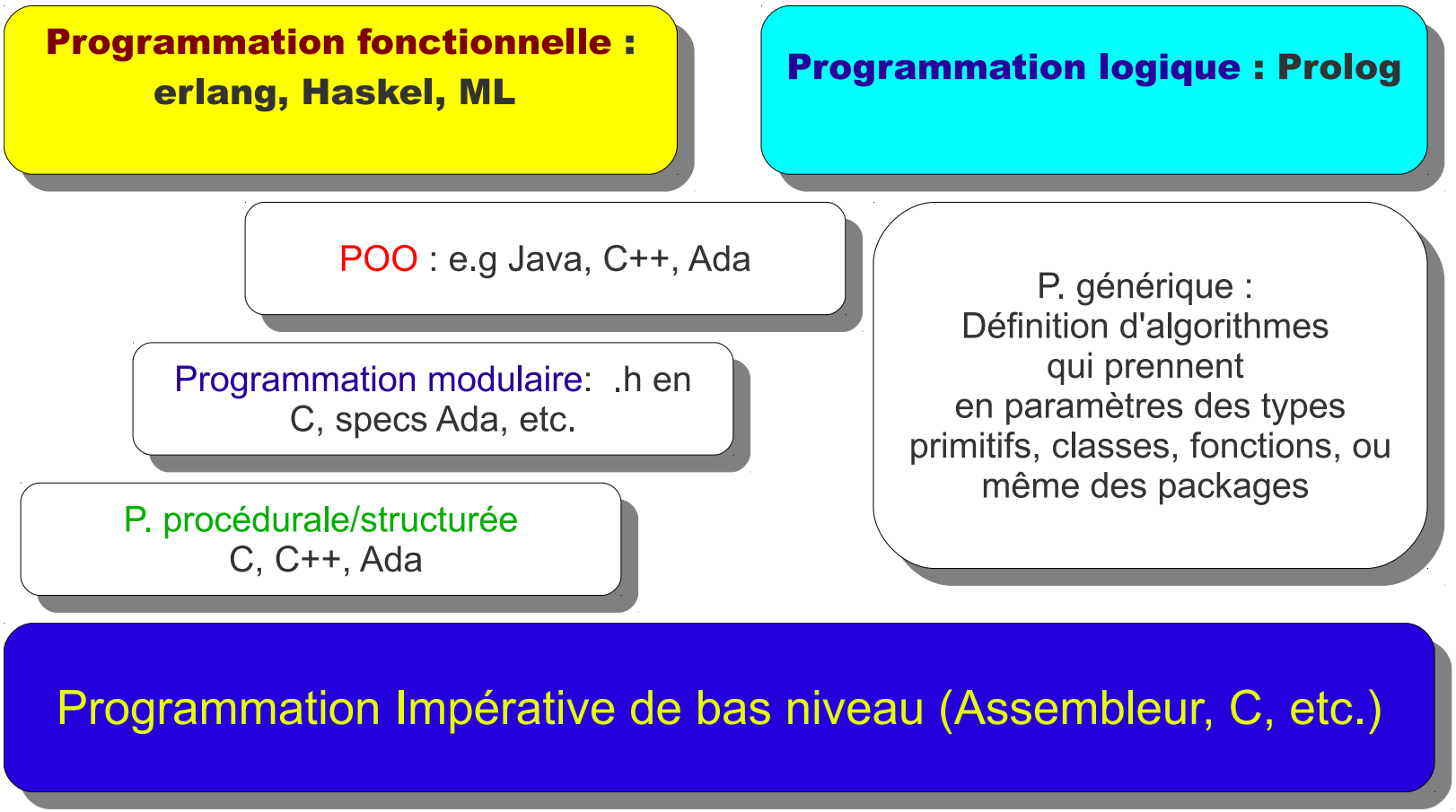
Niveaux d'abstraction des différents paradigmes I

- Niveau 0 : Programmation impérative (la machine exécute les instructions appartenant au langage qu'elle supporte)
- Niveau 1 : Programmation procédurale et structurée : définition d' unités (autonomes et réutilisables = fonctions/procédures)
- Niveau 2 : Programmation modulaire : définition de modules (e.g. fichiers .h en C) accessibles grâce à leurs interface. Plusieurs détails (allocation mémoire) restent à la charge du client de ces modules.
- Niveau 3 : Programmation orientée objets : propose le concept de l'objet qui encapsule à la fois le comportement et les données.

Niveaux d'abstraction des différents paradigmes II

- Niveau 3.5 : Programmation générique : s'applique aux autres paradigmes pour augmenter la réutilisatbilité des algorithmes indépendamment des structures de données (types, fonctions, paquetages, etc.). Le langage de programmation instancie la version générique en utilisant des types, des fonctions ou même des packages comme paramètres.
- Niveau 4 : programmation déclarative : basés sur des fondements théoriques solides (lambda calcul, logique) limite les effets de bord (e.g. Erlang) ou les élimine complètement (langages fonctionnels purs e.g. Haskel)

Niveaux d'abstraction des différents paradigmes



Programmation impérative

Définition

- Définit les programmes comme un ensemble d'*instructions* qui agissent sur un ensemble de données (état du programme) pour implémenter les algorithmes et réaliser les calculs.
  - Principales instructions : assignation (affectation), branchement conditionnel (if..then..else), branchement sans condition (goto, appels de procédures), boucles (for, while, etc.).
  - Ces instructions sont ensuite traduites (grâce aux compilateurs/interpréteurs) en instructions de plus bas niveau (langage machine spécifique au processeur).
- Les instructions sont ordonnées ; il est toujours possible de connaître la prochaine instruction ainsi que la manière avec laquelle elle doit être réalisée.
- Paradigme de base sur lequel sont construits les autres modèles !

Programmation impérative

Effets de bord

La programmation impérative se base sur la modification de variables

- Exemple 1 : écriture sur la console.
- Exemple 2 : mise à jour d'une variable globale par une procédure
- Les effets de bord sont nécessaires : input/output des calculs, communications, etc.
- Les effets de bord sont indésirables (sources d'erreurs, le résultat des calculs d'une fonction ne dépend pas uniquement de ses paramètres mais aussi de son contexte d'exécution.
- Les effets de bords sont très fréquents en programmation impértive.

Programmation impérative

Problèmes avec les effets de bord

- Les programmes avec les effets de bord sont difficiles à analyser et à déboguer
- Les expressions sans effets de bord peuvent être exécutées dans n'importe quel ordre (support de la concurrence).
- sans effets de bord si deux équations sont équivalentes une fois alors elles le sont pour toujours.
- L'interaction avec le monde (I/O, GUI, accès réseau, etc.) ne semble pas être bien supportée par la programmation impérative.
- Pour mieux contrôler les modifications de l'état des programmes deux types de programmations impératives ont été proposés : programmation structurée et programmation procédurale.

Programmation structurée

Programmation structurée

- Sous-paradigme de la programmation impérative qui consiste à supprimer l'instruction goto ou à limiter son utilisation aux cas graves (exceptions)
- Parmi les langages de programmation les plus structurants, on trouve Pascal et Ada.
- Objectif : créer des unités de code faciles à comprendre isolément, indépendamment de leur contexte

Programmation procédurale

Programmation procédurale

- Basée sur le concept de procédure/fonction ou routine ou sous programme.
- Les sous-programmes utilisent des variables locales et agissent sur des arguments fournis explicitement en paramètre, par valeur ou par référence.
- Un sous programme peut être appelé à partir d'un autre sous-programme procédure ou à partir de lui même (récursivité).
- Avantages :
  - Appliquer les mêmes traitements sur différents paramètres à partir de plusieurs endroits dans le programme.
  - Simplifier la programmation en supporter la construction de programmes structurés en sous programmes.

Programmation impérative de base : discussion

- La programmation structurée/procédurale peut s'associer à la méthodologie de développement par décompositions successives (top-down) : le problème est décomposé en sous problèmes plus petits. Une solution élémentaire est ensuite proposée, implémentée et testée pour chacun de ces sous problèmes.
- La définition de sous programmes sans effets de bord favorise un premier type de modularité. Les modules présentent une interface connue, et un comportement bien défini (indépendamment du contexte). Les langages tels que C donnent la possibilité aux fonctions de se comporter d'une manière isolée (définition et portée des variables locales, etc) mais n'empêchent pas les effets de bord (I/O, variables globales, passage par référence).
- Mais : les modèles de programmation structurée ou procédurale atteignent très rapidement leurs limites surtout pour les applications de grande taille

Programmation impérative de base : discussion

- La programmation impérative de base minimise les pertes en performances. Elle était appropriée lorsque les ressources et les outils d'analyse et de compilation étaient limitées
- Le programmeur optimise les performances et la consommation mémoire de son programme à la main.
- Mais la clarté, lisibilité et aptitude à la maintenance du code étaient sacrifiés !
- Également, la portabilité était difficilement réalisable car les architectures de bas niveau influençaient la manière de programmer. La seule solution pour réaliser la portabilité est donc de dupliquer le code et l'optimiser de nouveau pour la nouvelle architecture.

Programmation modulaire

- La programmation procédurale atteint ses limites avec l'augmentation de la taille des programmes.
- Principe de l'encapsulation des données : regroupement des sous-programmes en relation ainsi et les données qu'elles manipulent.
- La communication inter-modules se fait à travers une interface ; les détails d'implantation de chaque module sont cachés.
- Exemple en C : fichier compteur.h avec getNewValue() comme seul service.
- Il faut choisir : soit un seul compteur par module client !
- soit modifier l'interface getNewValue(&cpt). Dans ce cas le module client doit maintenant de gérer explicitement le cycle de vie du compteur (création/destruction explicite)
- Comment faire pour la gestion des types de haut niveau (structures) : allocation/libération de la mémoire, que se passe t-il si ce type doit évoluer (changement au niveau de la structure) ?

Programmation orientée objets

POO : Solution à plusieurs problèmes

- Modifications non contrôlées au niveau de l'état (encapsulation des données, règles de visibilité, définition d'interfaces)
- Réutilisation/maintenance du code (héritage, polymorphisme)

Unité fondamentale de la POO : L'objet

- Unité autonome qui interagit avec son environnement en utilisant "des messages" (appels au niveau de son interface)
- L'objet découple l'implémentation interne de son interaction avec son environnement en offrant ou en s'engageant à implémenter une ou plusieurs interfaces
- Les programmes sont écrits sous forme d'ensembles d'objets qui interagissent et subissent des modifications au niveau de leurs états ; les objets sont responsables de réaliser leurs contrats. Ils ont également des parties internes invisibles depuis l'extérieur.

Programmation orientée objets

Héritage

- L'héritage Favorise la réutilisation du code en permettant le raffinage d'objets existants.
- L'héritage et le polymorphisme (l'habilite d'un objet d'un type d'être utilisé comme s'il était un autre type), permettent de réutilise une très grande portion du code initial.

Avantages

- Favorise une conception modulaire autour du concept de l'objet, une architecture claire, et limite les effets de bords
- Supporte à merveille le top-down design.

Discussion

Limites de la POO

- Les objets ne permettent pas de capturer toutes les abstractions : par exemple les algorithmes opèrent sur les objets mais ne peuvent pas être classifiés eux même comme objets.
- Les compilateurs et les environnement d'exécution pour l'OO sont de loin plus complexes que ceux conçus pour la programmation impérative (exemple : gestion du polymorphisme).
- Les instructions qui peuvent paraître simples (au niveau des interfaces des objets) peuvent cacher une vraie complexité. Le développeur doit être au courant des coûts en mémoire et performances de ces instructions.

Programmation déclarative

- La programmation impérative décrit les étapes à suivre (structures des données, instructions de niveau plus ou moins élevé) pour résoudre un problème donné
- La programmation déclarative sépare ce qui doit être calculé de la méthode de calcul. Le programmeur définit le problème, le compilateur le résout.
- Exemples :
- Paradigmes
  - Programmation fonctionnelle
  - Programmation logique

Programmation logique

Programmation logique

- Déclaration de fait et de règles informations avérées : (base de connaissances)
- Définition des requêtes (buts)
- Le compilateur/interpréteur tente de répondre à la requête en se servant des informations contenues dans la base de connaissance
- Application : Utilisée en intelligence artificielle
- Exemple de langage supportant la programmation logique : Prolog

Prolog

Prolog

- Calculabilité Aussi puissant que les machines de Turing
- un moteur d'inférence particulier permet larésolution des règles particulières (clauses de Horn)
- Avantages : rapidité et simplicité de la programmation.
- Inconvénients : temps d'exécution potentiellement très long : la maîtrise du fonctionnement du moteur d'inférence est nécessaire ! !

Programmation fonctionnelle

- Les paradigmes de programmation impérative et OO formulent explicitement les algorithmes et programmes comme séquences d'instructions agissant sur un ensemble de variables (état)
- Le paradigme de programmation fonctionnelle se base sur la notion de fonction mathématique et ne se base plus sur la notion d'état
- Le programmeur ne se préoccupe pas de l'état de la mémoire
- Un programme est donc une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble de valeurs en entrée (Exécution d'un programme = Evaluation d'une fonction)
- Conséquences : les variables ne varient plus, plus de boucles, plus de type/classe string, plus de classes/objets du tout..

Programmation fonctionnelle

Avantages

- Processus légers et autonomes, plus de problèmes de concurrence,
- Les fonctions sont des constructions de première catégorie
  - Les fonctions peuvent être utilisées dans n'importe quel contexte y compris comme paramètre en entrée ou comme résultat d'une fonction,
  - Donc, en C, les fonctions ne sont pas des objets de première classe (il n'est pas possible de créer de nouvelles fonctions à l'exécution contrairement à d'autres "objets".
  - En effet ce sont objets de seconde classe car elles peuvent être manipulées grâce aux pointeurs de fonctions.
- Pattern matching
- Cadre de travail théorique très puissant basé sur le λcalcul
- Le polymorphisme très coûteux au niveau de la programmation déclarative, se trouve naturellement supporté.