Chapitre 1: Concepts la Programmation Orientée Objet

I52 POO & IHM — C++

Valérie Gillot

Septembre 2019

Plan du chapitre

1. Conception Orientée Objet
2. Programmation Orientée Objet
3. C++

Section 1

Conception Orientée Objet

- Modularité
- Mariage entre modularité et ré-utilisabilité

Cycles de vie d'un logiciel

1. Expression des besoins
2. Spécification
3. Analyse des besoins
4. Conception
   - Choix des solutions techniques : architecture, performance et optimisation
   - Définition des structures et des algorithmes
5. Implémentation
6. Tests et vérification
7. Validation
8. Maintenance et évolution

Conception Orientée Objet : principes

- Basée sur les concepts de génie logiciel
- Favorise la conception, la fabrication et la maintenance de systèmes logiciels complexes
- Pour optimiser le coût en temps et en argent, le Génie logiciel rationalise :
  - le développement
  - la maintenance

Conception Orientée Objet : principes

- Basée sur les concepts de génie logiciel
- Favorise la conception, la fabrication et la maintenance de systèmes logiciels complexes
- Pour optimiser le coût en temps et en argent, le Génie logiciel rationalise :
  - le développement
  - la maintenance

Conception Orientée Objet : principes

- Basée sur les concepts de génie logiciel
- Favorise la conception, la fabrication et la maintenance de systèmes logiciels complexes
- Pour optimiser le coût en temps et en argent, le Génie logiciel rationalise :
  - le développement
  - la maintenance

Conception Orientée Objet : principes

- Basée sur les concepts de génie logiciel
- Favorise la conception, la fabrication et la maintenance de systèmes logiciels complexes
- Pour optimiser le coût en temps et en argent, le Génie logiciel rationalise :
  - le développement
  - la maintenance

Conception Orientée Objet : principes

- Basée sur les concepts de génie logiciel
- Favorise la conception, la fabrication et la maintenance de systèmes logiciels complexes
- Pour optimiser le coût en temps et en argent, le Génie logiciel rationalise :
  - le développement
  - la maintenance

Critères de qualité d'un logiciel

Pour les utilisateurs

ergonomie et efficacité

Pour les concepteurs

- Validité (ou exactitude) : réalisation exacte des tâches spécifiées
- Robustesse : capacité à fonctionner hors des cas de spécifications
- Extensibilité : facilité d'adaptation aux changements de spécifications
- Modularité : aptitude à pouvoir combiner les modules
- Ré-utilisabilité des modules grâce à une conception normalisée pour créer d'autres applications

Critères de qualité d'un logiciel

Pour les utilisateurs

ergonomie et efficacité

Pour les concepteurs

- Validité (ou exactitude) : réalisation exacte des tâches spécifiées
- Robustesse : capacité à fonctionner hors des cas de spécifications
- Extensibilité : facilité d'adaptation aux changements de spécifications
- Modularité : aptitude à pouvoir combiner les modules
- Ré-utilisabilité des modules grâce à une conception normalisée pour créer d'autres applications

Critères de qualité d'un logiciel

Pour les utilisateurs

ergonomie et efficacité

Pour les concepteurs

- Validité (ou exactitude) : réalisation exacte des tâches spécifiées
- Robustesse : capacité à fonctionner hors des cas de spécifications
- Extensibilité : facilité d'adaptation aux changements de spécifications
- Modularité : aptitude à pouvoir combiner les modules
- Ré-utilisabilité des modules grâce à une conception normalisée pour créer d'autres applications

Critères de qualité d'un logiciel

Pour les utilisateurs

ergonomie et efficacité

Pour les concepteurs

- Validité (ou exactitude) : réalisation exacte des tâches spécifiées
- Robustesse : capacité à fonctionner hors des cas de spécifications
- Extensibilité : facilité d'adaptation aux changements de spécifications
- Modularité : aptitude à pouvoir combiner les modules
- Ré-utilisabilité des modules grâce à une conception normalisée pour créer d'autres applications

Critères de qualité d'un logiciel

Pour les utilisateurs

ergonomie et efficacité

Pour les concepteurs

- Validité (ou exactitude) : réalisation exacte des tâches spécifiées
- Robustesse : capacité à fonctionner hors des cas de spécifications
- Extensibilité : facilité d'adaptation aux changements de spécifications
- Modularité : aptitude à pouvoir combiner les modules
- Ré-utilisabilité des modules grâce à une conception normalisée pour créer d'autres applications

Modularité

Deux catégories de méthodes modulaires

- méthodes descendantes : décomposition itératives du problème en sous-systèmes (peu réutilisable)
- méthodes ascendantes : composition d'éléments simples (bibliothèques)

Modularité

Deux catégories de méthodes modulaires

- méthodes descendantes : décomposition itératives du problème en sous-systèmes (peu réutilisable)
- méthodes ascendantes : composition d'éléments simples (bibliothèques)

Propriétés d'une méthode modulaire

Selon Bertrand Meyer (1990)

- Décomposition des problèmes en sous problèmes résolubles séparément
- Composition des composants logiciels pour créer de nouveaux programmes
- Compréhension de chaque module séparément
- Continuité : une petite modification induit la modification de peu de modules
- Protection : un comportement indésirable doit impacter un ou peu de modules

Propriétés d'une méthode modulaire

Selon Bertrand Meyer (1990)

- Décomposition des problèmes en sous problèmes résolubles séparément
- Composition des composants logiciels pour créer de nouveaux programmes
- Compréhension de chaque module séparément
- Continuité : une petite modification induit la modification de peu de modules
- Protection : un comportement indésirable doit impacter un ou peu de modules

Propriétés d'une méthode modulaire

Selon Bertrand Meyer (1990)

- Décomposition des problèmes en sous problèmes résolubles séparément
- Composition des composants logiciels pour créer de nouveaux programmes
- Compréhension de chaque module séparément
- Continuité : une petite modification induit la modification de peu de modules
- Protection : un comportement indésirable doit impacter un ou peu de modules

Propriétés d'une méthode modulaire

Selon Bertrand Meyer (1990)

- Décomposition des problèmes en sous problèmes résolubles séparément
- Composition des composants logiciels pour créer de nouveaux programmes
- Compréhension de chaque module séparément
- Continuité : une petite modification induit la modification de peu de modules
- Protection : un comportement indésirable doit impacter un ou peu de modules

Propriétés d'une méthode modulaire

Selon Bertrand Meyer (1990)

- Décomposition des problèmes en sous problèmes résolubles séparément
- Composition des composants logiciels pour créer de nouveaux programmes
- Compréhension de chaque module séparément
- Continuité : une petite modification induit la modification de peu de modules
- Protection : un comportement indésirable doit impacter un ou peu de modules

Modularité dans la pratique...

Un module

- doit avoir une unité sémantique
- correspondre à une unité syntaxique du langage
- doit pouvoir être compilé séparément
- toute information propre à un module doit rester privée (*information hiding*)
- i.e. : un module est connu par une description (*interface*), les communications entre modules sont réduites et explicitées

Modularité dans la pratique...

Un module

- doit avoir une unité sémantique
- correspondre à une unité syntaxique du langage
- doit pouvoir être compilé séparément
- toute information propre à un module doit rester privée (*information hiding*)
- i.e. : un module est connu par une description (*interface*), les communications entre modules sont réduites et explicitées

Modularité dans la pratique...

Un module

- doit avoir une unité sémantique
- correspondre à une unité syntaxique du langage
- doit pouvoir être compilé séparément
- toute information propre à un module doit rester privée (*information hiding*)
- i.e. : un module est connu par une description (*interface*), les communications entre modules sont réduites et explicitées

Modularité dans la pratique...

Un module

- doit avoir une unité sémantique
- correspondre à une unité syntaxique du langage
- doit pouvoir être compilé séparément
- toute information propre à un module doit rester privée (*information hiding*)
- i.e. : un module est connu par une description (*interface*), les communications entre modules sont réduites et explicitées

Modularité dans la pratique...

Un module

- doit avoir une unité sémantique
- correspondre à une unité syntaxique du langage
- doit pouvoir être compilé séparément
- toute information propre à un module doit rester privée (*information hiding*)
- i.e. : un module est connu par une description (*interface*), les communications entre modules sont réduites et explicitées

Mariage entre modularité et ré-utilisabilité

Les quatre conditions essentielles selon Meyer

1. Abstraction des données : les opérations qui s'appliquent à un même modèle de données sont regroupées
2. Adaptabilité : un module doit être adaptable à des structures de données ayant des implantations différentes
3. Généricité un module doit être adaptable à plusieurs types de données
4. Factorisation dans un module d'informations communes à plusieurs autres

Mariage entre modularité et ré-utilisabilité

Les quatre conditions essentielles selon Meyer

1. Abstraction des données : les opérations qui s'appliquent à un même modèle de données sont regroupées
2. Adaptabilité : un module doit être adaptable à des structures de données ayant des implantations différentes
3. Généricité un module doit être adaptable à plusieurs types de données
4. Factorisation dans un module d'informations communes à plusieurs autres

Mariage entre modularité et ré-utilisabilité

Les quatre conditions essentielles selon Meyer

1. Abstraction des données : les opérations qui s'appliquent à un même modèle de données sont regroupées
2. Adaptabilité : un module doit être adaptable à des structures de données ayant des implantations différentes
3. Généricité un module doit être adaptable à plusieurs types de données
4. Factorisation dans un module d'informations communes à plusieurs autres

Mariage entre modularité et ré-utilisabilité

Les quatre conditions essentielles selon Meyer

1. Abstraction des données : les opérations qui s'appliquent à un même modèle de données sont regroupées
2. Adaptabilité : un module doit être adaptable à des structures de données ayant des implantations différentes
3. Généricité un module doit être adaptable à plusieurs types de données
4. Factorisation dans un module d'informations communes à plusieurs autres

Mariage entre modularité et ré-utilisabilité

Les réponses de la POO

1. la notion d'objet permet de regrouper dans un module une structure de donnée et les opérations spécialisées qui s'y rattachent
2. la redéfinition (ou surcharge, surdéfinition) : un opérateur ou une fonction peut porté le même nom (identificateur) dans des modules différents
3. le polymorphisme : les objets peuvent avoir plusieurs types différents
4. la généricité : traitement identique d'objets de types différents

Mariage entre modularité et ré-utilisabilité

Les réponses de la POO

1. la notion d'objet permet de regrouper dans un module une structure de donnée et les opérations spécialisées qui s'y rattachent
2. la redéfinition (ou surcharge, surdéfinition) : un opérateur ou une fonction peut porté le même nom (identificateur) dans des modules différents
3. le polymorphisme : les objets peuvent avoir plusieurs types différents
4. la généricité : traitement identique d'objets de types différents

Mariage entre modularité et ré-utilisabilité

Les réponses de la POO

1. la notion d'objet permet de regrouper dans un module une structure de donnée et les opérations spécialisées qui s'y rattachent
2. la redéfinition (ou surcharge, surdéfinition) : un opérateur ou une fonction peut porté le même nom (identificateur) dans des modules différents
3. le polymorphisme : les objets peuvent avoir plusieurs types différents
4. la généricité : traitement identique d'objets de types différents

Mariage entre modularité et ré-utilisabilité

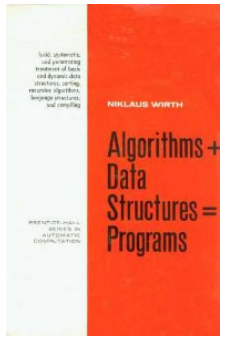
Les réponses de la POO

1. la notion d'objet permet de regrouper dans un module une structure de donnée et les opérations spécialisées qui s'y rattachent
2. la redéfinition (ou surcharge, surdéfinition) : un opérateur ou une fonction peut porté le même nom (identificateur) dans des modules différents
3. le polymorphisme : les objets peuvent avoir plusieurs types différents
4. la généricité : traitement identique d'objets de types différents

Conception d'un programme

Selon N. Wirth

programme = algorithme + structure de données



Traitement ou données ?

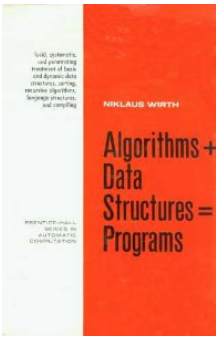
La conception d'un programme doit-elle se faire par le traitement ou les données ?

- dans la conception par le traitement, la ré-utilisabilité et la modularité ne sont pas favorisées
- ✓ la conception par les données est favorable pour le développement de programme de taille importante

Conception d'un programme

Selon N. Wirth

programme = algorithme + structure de données



Traitement ou données ?

La conception d'un programme doit-elle se faire par le traitement ou les données ?

- dans la conception par le traitement, la ré-utilisabilité et la modularité ne sont pas favorisées
- ✓ la conception par les données est favorable pour le développement de programme de taille importante

Section 2

Programmation Orientée Objet

Programmation Orientée Objet : historique et définition

La POO est un paradigme de programmation basée sur la COO pour le développement de logiciels de taille importante

- introduit par O.J. Dahl & K. Nygaard au début des années 1960, poursuivi par A. Kay dans les années 1970
- qui consiste en la définition et l'interaction de briques logicielles appelées objets
- un objet représente un concept, une idée ou toute entité du monde physique
  - représenter ces objets et leurs relations ;
  - l'interaction entre les objets via leurs relations
  - concevoir et réaliser les fonctionnalités attendues

Programmation Orientée Objet : historique et définition

La POO est un paradigme de programmation basée sur la COO pour le développement de logiciels de taille importante

- introduit par O.J. Dahl & K. Nygaard au début des années 1960, poursuivi par A. Kay dans les années 1970
- qui consiste en la définition et l'interaction de briques logicielles appelées objets
- un objet représente un concept, une idée ou toute entité du monde physique
  - représenter ces objets et leurs relations ;
  - l'interaction entre les objets via leurs relations
  - concevoir et réaliser les fonctionnalités attendues

Programmation Orientée Objet : historique et définition

La POO est un paradigme de programmation basée sur la COO pour le développement de logiciels de taille importante

- introduit par O.J. Dahl & K. Nygaard au début des années 1960, poursuivi par A. Kay dans les années 1970
- qui consiste en la définition et l'interaction de briques logicielles appelées objets
- un objet représente un concept, une idée ou toute entité du monde physique
  - représenter ces objets et leurs relations ;
  - l'interaction entre les objets via leurs relations
  - concevoir et réaliser les fonctionnalités attendues

Programmation Orientée Objet : historique et définition

La POO est un paradigme de programmation basée sur la COO pour le développement de logiciels de taille importante

- introduit par O.J. Dahl & K. Nygaard au début des années 1960, poursuivi par A. Kay dans les années 1970
- qui consiste en la définition et l'interaction de briques logicielles appelées objets
- un objet représente un concept, une idée ou toute entité du monde physique
  - représenter ces objets et leurs relations ;
  - l'interaction entre les objets via leurs relations
  - concevoir et réaliser les fonctionnalités attendues

Programmation Orientée Objet : historique et définition

La POO est un paradigme de programmation basée sur la COO pour le développement de logiciels de taille importante

- introduit par O.J. Dahl & K. Nygaard au début des années 1960, poursuivi par A. Kay dans les années 1970
- qui consiste en la définition et l'interaction de briques logicielles appelées objets
- un objet représente un concept, une idée ou toute entité du monde physique
  - représenter ces objets et leurs relations ;
  - l'interaction entre les objets via leurs relations
  - concevoir et réaliser les fonctionnalités attendues

Programmation Orientée Objet : historique et définition

La POO est un paradigme de programmation basée sur la COO pour le développement de logiciels de taille importante

- introduit par O.J. Dahl & K. Nygaard au début des années 1960, poursuivi par A. Kay dans les années 1970
- qui consiste en la définition et l'interaction de briques logicielles appelées objets
- un objet représente un concept, une idée ou toute entité du monde physique
  - représenter ces objets et leurs relations ;
  - l'interaction entre les objets via leurs relations
  - concevoir et réaliser les fonctionnalités attendues

Programmation Orientée Objet : Paradigmes dérivés

- Programmation orientée prototype : fondée sur la notion de prototype. Un prototype est un objet à partir duquel on crée de nouveaux objets. L'héritage est dynamique : tout objet peut changer de parent à l'exécution, n'importe quand. Exemple Javascript
- Programmation orientée classe (C++, Java, Python, OCamel)
- Programmation orientée composant : une approche objet, non pas au sein du code, mais au niveau de l'architecture générale du logiciel

Programmation Orientée Objet : Paradigmes dérivés

- Programmation orientée prototype : fondée sur la notion de prototype. Un prototype est un objet à partir duquel on crée de nouveaux objets. L'héritage est dynamique : tout objet peut changer de parent à l'exécution, n'importe quand. Exemple Javascript
- Programmation orientée classe (C++, Java, Python, OCamel)
- Programmation orientée composant : une approche objet, non pas au sein du code, mais au niveau de l'architecture générale du logiciel

Programmation Orientée Objet : Paradigmes dérivés

- Programmation orientée prototype : fondée sur la notion de prototype. Un prototype est un objet à partir duquel on crée de nouveaux objets. L'héritage est dynamique : tout objet peut changer de parent à l'exécution, n'importe quand. Exemple Javascript
- Programmation orientée classe (C++, Java, Python, OCamel)
- Programmation orientée composant : une approche objet, non pas au sein du code, mais au niveau de l'architecture générale du logiciel

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré
  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Autres paradigmes de programmation

- Programmation impérative, paradigme originel et le plus courant
  - Programmation structurée, visant à structurer les programmes impératifs pour en supprimer les instructions goto

Exemple

- C est un langage impératif structuré,
- Python (multi-paradigmes) peut être utilisé comme langage impératif et structuré

  - Programmation procédurale, à comparer à la programmation fonctionnelle
- Programmation orientée objet et ses dérivées
- Programmation déclarative, consistant à déclarer les données du problème, puis à demander au programme de le résoudre
  - Programmation descriptive (par exemple, HTML, XML ou LaTeX)
  - Programmation fonctionnelle (Camel)
  - Programmation logique (par exemple Prolog)
  - Programmation par contraintes

Programmation Orientée Objet : les principes

1. Objet
2. Typage et polymorphisme
3. Redéfifinition
4. Classe ou prototype

Programmation Orientée Objet : les principes

1. Objet
2. Typage et polymorphisme
3. Redéfifinition
4. Classe ou prototype

Programmation Orientée Objet : les principes

1. Objet
2. Typage et polymorphisme
3. Redéfifinition
4. Classe ou prototype

Programmation Orientée Objet : les principes

1. Objet
2. Typage et polymorphisme
3. Redéfifinition
4. Classe ou prototype

Principes : Objet

L'objet est une structure de données valuées et cachées qui répond à un ensemble de messages.

- Les données (ou champs) qui décrivent sa structure interne sont appelées ses attributs ou données membres
- L'ensemble des messages forme ce que l'on appelle l'interface de l'objet, description du comportement, ce sont les méthodes ou fonctions membres
- Certains attributs et/ou méthodes sont cachés (privés) : c'est le principe d'encapsulation.

Principes : Objet

L'objet est une structure de données valuées et cachées qui répond à un ensemble de messages.

- Les données (ou champs) qui décrivent sa structure interne sont appelées ses attributs ou données membres
- L'ensemble des messages forme ce que l'on appelle l'interface de l'objet, description du comportement, ce sont les méthodes ou fonctions membres
- Certains attributs et/ou méthodes sont cachés (privés) : c'est le principe d'encapsulation.

Principes : Objet

L'objet est une structure de données valuées et cachées qui répond à un ensemble de messages.

- Les données (ou champs) qui décrivent sa structure interne sont appelées ses attributs ou données membres
- L'ensemble des messages forme ce que l'on appelle l'interface de l'objet, description du comportement, ce sont les méthodes ou fonctions membres
- Certains attributs et/ou méthodes sont cachés (privés) : c'est le principe d'encapsulation.

Principes : typage, polymorphisme, redéfinition, classe

1. En POO chaque objet est typé. Le type définit la syntaxe (« Comment l'appeler ? »). Un objet peut appartenir à plus d'un type : c'est le polymorphisme
2. La POO permet à un objet de raffiner la mise en œuvre d'un message défini pour des objets d'un type parent, autrement dit de redéfinir la méthode associée au message : c'est le principe de redéfinition des messages (ou overriding en anglais).
3. La classe est une structure qui décrit la structure interne des données et elle définit les méthodes qui s'appliqueront aux objets de même famille (même classe) ou type.

Principes : typage, polymorphisme, redéfinition, classe

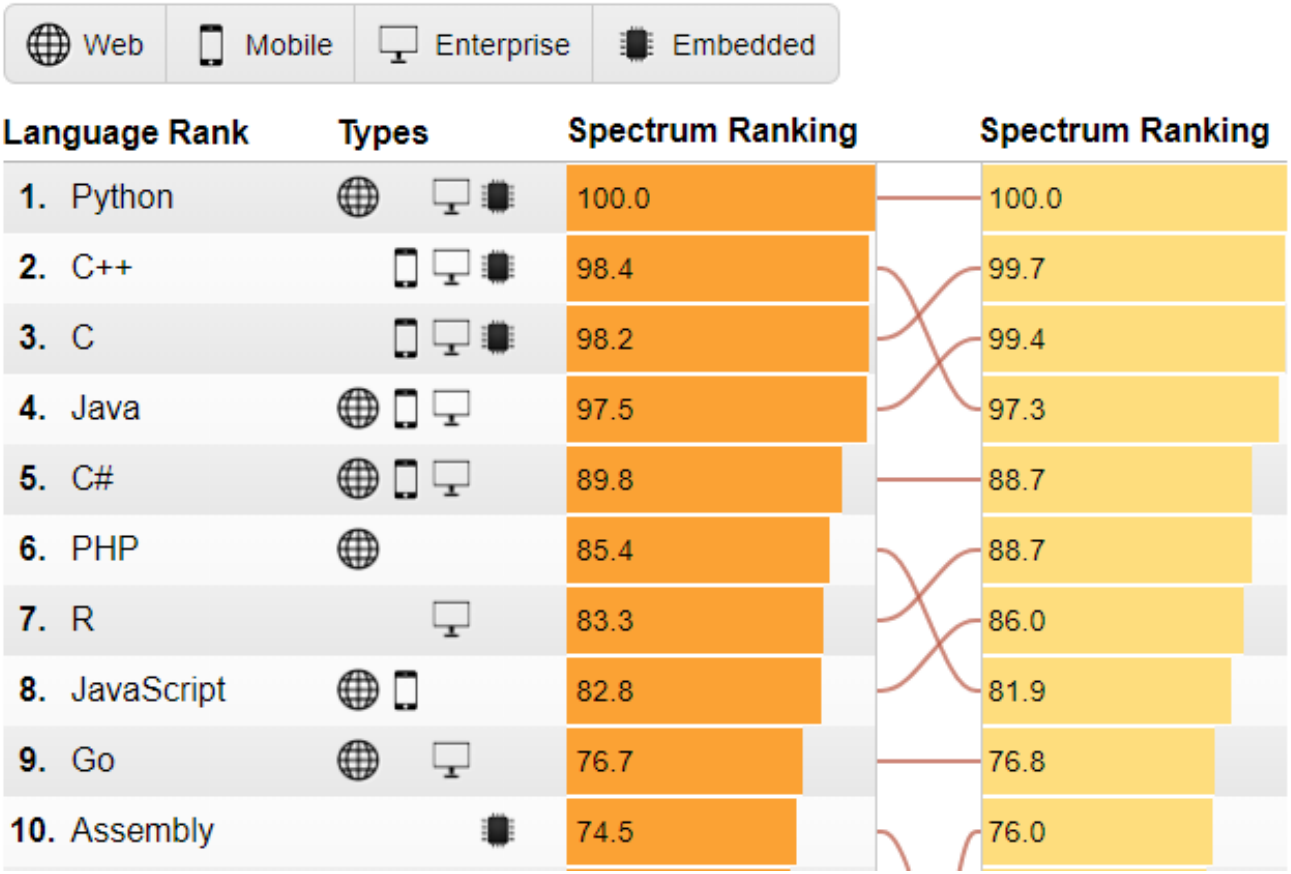
1. En POO chaque objet est typé. Le type définit la syntaxe (« Comment l'appeler ? »). Un objet peut appartenir à plus d'un type : c'est le polymorphisme
2. La POO permet à un objet de raffiner la mise en œuvre d'un message défini pour des objets d'un type parent, autrement dit de redéfinir la méthode associée au message : c'est le principe de redéfinition des messages (ou overriding en anglais).
3. La classe est une structure qui décrit la structure interne des données et elle définit les méthodes qui s'appliqueront aux objets de même famille (même classe) ou type.

Principes : typage, polymorphisme, redéfinition, classe

1. En POO chaque objet est typé. Le type définit la syntaxe (« Comment l'appeler ? »). Un objet peut appartenir à plus d'un type : c'est le polymorphisme
2. La POO permet à un objet de raffiner la mise en œuvre d'un message défini pour des objets d'un type parent, autrement dit de redéfinir la méthode associée au message : c'est le principe de redéfinition des messages (ou overriding en anglais).
3. La classe est une structure qui décrit la structure interne des données et elle définit les méthodes qui s'appliqueront aux objets de même famille (même classe) ou type.

Classement IEEE des langages 2018

Le top 10 des langages en 2018 (colonne de gauche) en comparaison au classement de 2017 (colonne de droite).



Section 3

C++

- Historique
- Propriétés et Avantages

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI

1983 *C with classes* devient C++

1985 commercialisation de la version 1.0

1987 compilateur GNU (GCC 6.3 released [2016-12-21])

1989 version 2.0 (héritage multiple, norme ANSI)

1990 classes génériques, traitement des exceptions

1991 normalisation par l'ISO

1994 normalisation de la compatibilité C/C++

1997 normalisation de la version 3.0.

1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)

2003 publication d'une version corrigée du standard C++.

⋮

2014 ISO/CEI 14882 :2014.

2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !

2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.

⋮

2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

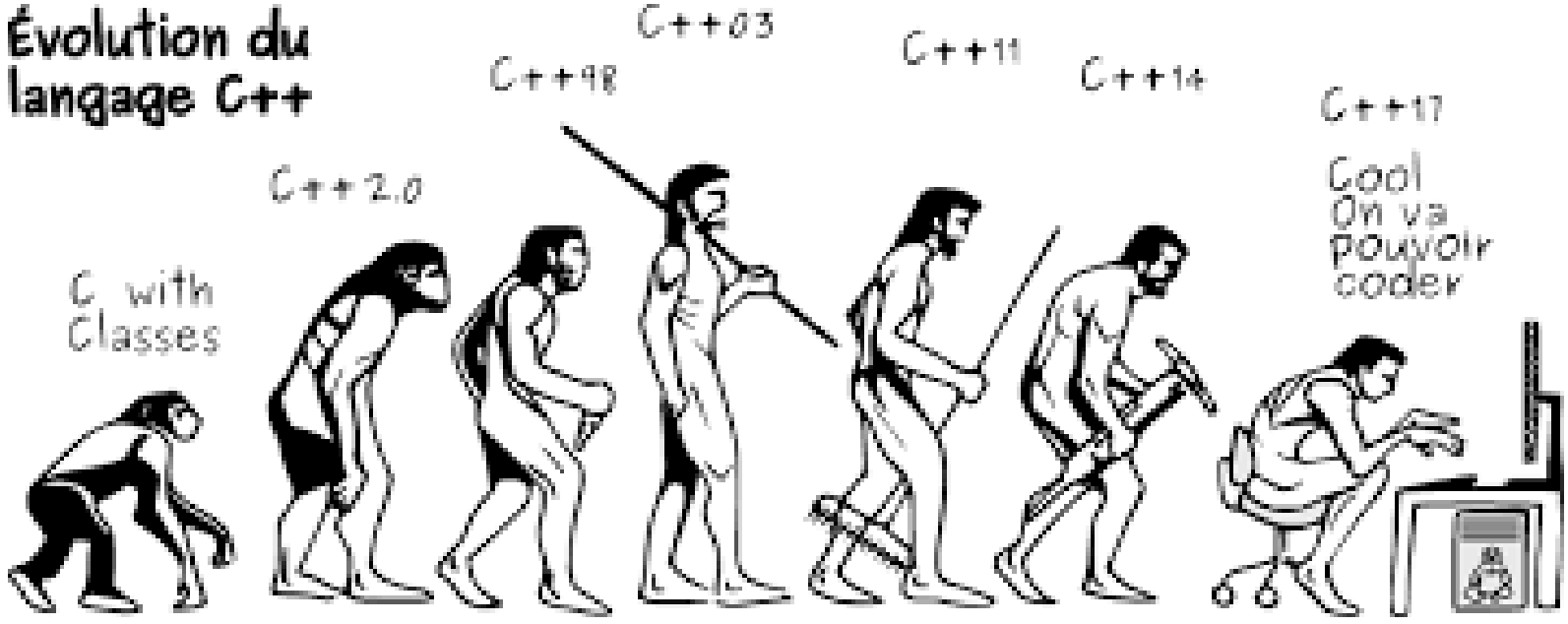
C++ est libre de droits

Historique

1979 création de *C with classes* par Bjarne Stroustrup, inspiré de SIMULA, compatible C ANSI
1983 *C with classes* devient C++
1985 commercialisation de la version 1.0
1987 compilateur GNU (GCC 6.3 released [2016-12-21])
1989 version 2.0 (héritage multiple, norme ANSI)
1990 classes génériques, traitement des exceptions
1991 normalisation par l'ISO
1994 normalisation de la compatibilité C/C++
1997 normalisation de la version 3.0.
1998 standardisation de C++(ISO/CEI 14882 :1998) (réunit à Sophia Antipolis)
2003 publication d'une version corrigée du standard C++.
⋮
2014 ISO/CEI 14882 :2014.
2017 C++17 depuis mars 2017 : ISO/CEI 14882 :2017 !
2020 prochaine version de C++

C++ est libre de droits

Évolution du langage C++



Naissance du C++, la petite histoire

Bjarne Stroustrup, danois, a développé C++ au cours des années 1980, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T. L'idée de créer un nouveau langage vient de l'expérience en programmation de Stroustrup pour sa thèse de doctorat. Il s'agissait en l'occurrence d'améliorer le langage C. Il l'avait d'ailleurs nommé C with classes (« C avec des classes »).

Les concepts de POO du C++

C++ est un langage multi-paradigmes : programmation procédurale, programmation orientée objet et programmation générique (généricité).

Les concepts supportés

- Encapsulation par la classe : des objets similaires partagent les mêmes méthodes mais se distinguent par leurs données privées.
- Généralisation-Spécialisation par l'héritage : permet de définir une classe fille à partir d'une (ou plusieurs) classe mère.
- Polymorphisme par les surcharges : le traitement associé à un message est déterminé dynamiquement à l'exécution en fonction de l'objet auquel il est adressé.

Les concepts de POO du C++

C++ est un langage multi-paradigmes : programmation procédurale, programmation orientée objet et programmation générique (généricité).

Les concepts supportés

- Encapsulation par la classe : des objets similaires partagent les mêmes méthodes mais se distinguent par leurs données privées.
- Généralisation-Spécialisation par l'héritage : permet de définir une classe fille à partir d'une (ou plusieurs) classe mère.
- Polymorphisme par les surcharges : le traitement associé à un message est déterminé dynamiquement à l'exécution en fonction de l'objet auquel il est adressé.

Les concepts de POO du C++

C++ est un langage multi-paradigmes : programmation procédurale, programmation orientée objet et programmation générique (généricité).

Les concepts supportés

- Encapsulation par la classe : des objets similaires partagent les mêmes méthodes mais se distinguent par leurs données privées.
- Généralisation-Spécialisation par l'héritage : permet de définir une classe fille à partir d'une (ou plusieurs) classe mère.
- Polymorphisme par les surcharges : le traitement associé à un message est déterminé dynamiquement à l'exécution en fonction de l'objet auquel il est adressé.

“Hello world !”

Dans le fichier `hello.cc` on écrit

```
#include <iostream>
using namespace std;

int main()
{
        cout << "Hello world!" << endl;
        return 0;
}
```

On compile avec la commande

```
g++ hello.cc -o hello
```

Ensuite on exécute avec la commande

```
./hello
```

On obtient

```
Hello world!
```