Classes génériques et opérateurs

Objectifs

- Fonctions/classes amies
- Redéfinition des opérateurs en C++
- Principe des classes génériques et aborder la syntaxe en C++

Fonctions/classes amies

- Une fonction, une méthode et une classes amies ont accès aux membres privés et protégés d'une classe dont elles ont été déclarées comme « amies »
- Le mot clef utilisé est « *friend* »
- Pour les classes (comme « en amour ») :
  - Si A est amie de B ⇏ B est amie de A
  - Si A est amie de B et que B est amie de C ⇏ A est amie de C
  - Si A est amie de B et que F est fille de A ⇏ F est amie de B

Fonctions amies – Exemple

```
class NombreV1 {
public:
  NombreV1(const int& p_valeur) : m_valeur(p_valeur) { ; }
  inline const int& valeur() { return this->m_valeur; }

  friend NombreV1 additionner(const NombreV1& p_nombre1, const NombreV1& p_nombre2);
private:
  int m_valeur;
};
```

```
inline NombreV1 additionner(const NombreV1& p_nombre1, const NombreV1& p_nombre2) {
  return NombreV1(p_nombre1.m_valeur + p_nombre2.m_valeur);
}
```

```
    NombreV1 n1(23), n2(19);
    NombreV1 n = additionner(n1, n2);
    std::cout << "n1 + n2 = " << n.valeur() << std::endl;
```

Méthodes amies – Exemple

```
class NombreV1 {
public:
  NombreV1(const int& p_valeur) : m_valeur(p_valeur) { ; }
  inline const int& valeur() { return this->m_valeur; }

  friend NombreV1 CalculatriceV1::additionner(const NombreV1& p_nombre1, const
NombreV1& p_nombre2);

private:
  int m_valeur;
};
```

```
class NombreV1;
class CalculatriceV1 {
public:
  NombreV1 additionner(const NombreV1& p_nombre1, const NombreV1& p_nombre2);
};
```

```
NombreV1 CalculatriceV1::additionner(const NombreV1& p_nombre1, const NombreV1& p_nombre2) {
  return NombreV1(p_nombre1.m_valeur + p_nombre2.m_valeur);
}
```

```
    CalculatriceV1 calculatriceV1;
    NombreV1 na2 = calculatriceV1.additionner(n1, n2);
    std::cout << "méthode amie - n1 + n2 = " << na2.valeur() << std::endl;
```

Classes amies – Exemple

```
class NombreV1 {
public:
  NombreV1(const int& p_valeur) : m_valeur(p_valeur) { ; }
  inline const int& valeur() { return this->m_valeur; }

  friend class CalculatriceV2;

private:
  int m_valeur;
};
```

```
class CalculatriceV2 {
public:
  inline NombreV1 additionner(const NombreV1& p_nombre1, const NombreV1& p_nombre2) {
    return NombreV1(p_nombre1.m_valeur + p_nombre2.m_valeur);
  }
};
```

```
    CalculatriceV2 calculatriceV2;
    NombreV1 na3 = calculatriceV2.additionner(n1, n2);
    std::cout << "classe amie - n1 + n2 = " << na3.valeur() << std::endl;
```

Quand utiliser les amitiés ?

- Quand l’utilisation d’une méthode membre est impossible
- Quand l’utilisation d’une méthode membre brise les principes SOLID
- Pour certains surcharge d’opérateurs (voir plus loin)

Surcharge d’opérateurs

- En C++, vous pouvez surcharger pratiquement tous les opérateurs, aussi bien les préfixes, les suffixes (cas unaire) et les infixes (cas binaire)
- Quelques exemples :
  - operator++() – préfixe : une fois l’opération effectuée, vous pouvez renvoyer l’objet courant
  - operator++(int) – suffixe : ne pas oublier de faire un copie de la valeur avant de faire l’opération d’incrémentation
  - operator<<(std::ostream&, <Type Obj>) : écrire dans le flux passé en paramètres. Passer l’objet à écrire dans le stream en référence constante. Renvoyer le flux
  - operator<<(std::istream&, <Type Obj>) : écrire dans le flux passé en paramètres. Passer l’objet à écrire dans le stream en référence. Renvoyer le flux

Surcharge d’opérateurs

- Pour la majorité des opérateurs, vous pouvez les déclarer et les définir comme étant des méthodes membres de la classe ou des méthodes amies
  - Certains opérateurs ne vous laisse pas le choix (« () », « [] », « = », « -> »)
  - Si vous surchargez les opérateurs de décalage de bits « << » et « >> » sur les flux (« stream »), vous **devez** les déclarer et les définir comme fonctions amies : le flux est l’objet de gauche
  - Le choix dépend aussi de l’endroit où vous désirez écrire le code
- Je vous conseille de prendre les paramètres par références constantes (évite la copie et n’est pas modifiable) et de renvoyer un nouvel objet (quand cela à du sens, donc pas pour les opérateurs « << » et « >> »)

Surcharge d'opérateurs – Exemple 1

```
std::string operator*(const std::string& p_chaineARepeter, const int& p_nbFois);
```

```
#include <sstream>

std::string operator*(const std::string& p_chaineARepeter, const int& p_nbFois) {
  std::stringstream ss;
  for (size_t fois = 0; fois < p_nbFois; fois++) {
    ss << p_chaineARepeter;
  }

  return ss.str();
}
```

```
    std::string s = "chocolatine ";
    //s *= 3; // erreur : operator*= non défini
    std::cout << "s = \"" << s << "\"" << std::endl;
    s = s * 3;
    std::cout << "s = s * 3 : \"" <<  s << "\"" << std::endl;
    std::cout << "s * 3 : \"" << s * 3 << "\"" << std::endl;
```

Surcharge d'opérateurs – Exemple 2

```
class NombreV1 {
public:
  // ...
  friend std::istream& operator>>(std::istream& p_istream, NombreV1& p_valeur);
  friend std::ostream& operator<<(std::ostream& p_ostream, const NombreV1& p_valeur);
};
```

```
inline std::ostream& operator<<(std::ostream& p_ostream, const NombreV1& p_valeur) {
  p_ostream << "NombreV1(" << p_valeur.m_valeur << ")" << std::endl;

  return p_ostream;
}

inline std::istream& operator>>(std::istream& p_istream, NombreV1& p_valeur) {
  p_istream >> p_valeur.m_valeur;

  return p_istream;
}
```

```
    NombreV1 nombreV1(42);
    std::cout << "nombreV1 : " << nombreV1 << std::endl;
    std::cout << "Saisir un entier : ";
    std::cin >> nombreV1;
    std::cout << "L'entier saisi est : " << nombreV1 << std::endl;
```

Quand surcharger les opérateurs ?

- Quand nous voulons simplifier l'utilisation d'une classe avec l'utilisation d'opérateurs plus « naturels »
- Quand la sémantique de l'opérateur n'est pas ambiguë : toujours utiliser le sens commun des opérateurs
  - Exemple « maVariable += maValeur » a un sens d'ajout, ici de « maValeur » à « maVariable »
- Toujours surcharger l'ensemble des opérateurs qui ont la même sémantique
  - Exemple :
    - « + » binaire, « + » unaire, « += », « ++ » préfixe, « ++ » suffixe, etc. (« - », « / » , « * » , « % »)
    - « == », « != », « < », « > », « <= », « >= »
    - Etc.

Méthodes et classes génériques

- Une fonction, une méthode ou une classe générique est une entité qui peut être paramétrée par un ou plusieurs types de données
- Cela permet souvent d'éviter les répétitions de code
  - Exemple : ListeEntiers, ListeChainesCaractères, etc. devient Liste<TypeValeur>. TypeValeur pourra alors prendre les types « int », « std::string », etc.
- En C++, il faut déclarer les paramètres de type avant l'entité avec le mot clef « template »
- Les entités doivent être **déclarées et définies** dans le fichier d'entête : le compilateur écrit le code spécifique par rapport aux paramètres utilisés lors de l'utilisation de l'entité
- C++ permet de donner une implantation spécifique pour des paramètres : il part des types les plus spécifiques aux plus génériques

Méthodes et classes génériques – Exemple

```
template <class TypeNombre>
class Nombre {
public:
  Nombre(const TypeNombre& p_valeur) : m_valeur(p_valeur) { ; }

  template<class TypeNombre>
  friend std::istream& operator>>(std::istream& p_istream, Nombre<TypeNombre>& p_valeur);
  template<class TypeNombre>
  friend std::ostream& operator<<(std::ostream& p_ostream, const Nombre<TypeNombre>& p_valeur);

  Nombre<TypeNombre> operator+(const Nombre<TypeNombre> p_valeur) {
    return Nombre<TypeNombre>(this->m_valeur + p_valeur.m_valeur);
  }

private:
  TypeNombre m_valeur;
};
```

Méthodes et classes génériques – Exemple

```
template<class TypeNombre>
std::ostream& operator<<(std::ostream& p_ostream, const Nombre<TypeNombre>& p_valeur) {
  p_ostream << "Nombre(" << p_valeur.m_valeur << ")" << std::endl;

  return p_ostream;
}

template<class TypeNombre>
std::istream& operator>>(std::istream& p_istream, Nombre<TypeNombre>& p_valeur) {
  p_istream >> p_valeur.m_valeur;

  return p_istream;
}
```

Quand utiliser les génériques ?

- Souvent dans les cadriciels
- Pour éviter de répéter du code basé sur un simple type (Liste, Dictionnaire, etc.)
- Pour généraliser un algorithme (exemple tri, recherche, filtre, etc.)

Références

- Livre Programmer en C++ moderne : Chapitres 15, 16, 18 et 19
- Aller plus loin : https://stackoverflow.com/questions/4421706/what-are-the-basic-rules-and-idioms-for-operator-overloading