Objet avancés : programmation générique.

Sujet 4 : Fonctions et fonctions d’ordre supérieur

En se basant sur le Design Pattern Foncteur, est il possible de définir des jeux de classes et de fonctions permettant la gestion homogènes d’objets « Callable », c’est a dire d’objet exposant un système similaire à celui des appels de fonctions classiques ?

Comment écririez vous une système permettant de récupérer les types des arguments et de retour d’un de ces objets et comment écrire la composée arbitraire de ces fonctions ? Inspirez vous de Boost.Function ou des articles de Alexandrescu sur le sujet.

ARENJO LOPEZ David

BRABAN Yoann

LE POTTIER Cyril

MARTIN Clément

NERET Emmanuel

Sommaire

I-Les Foncteurs, Généralité

II-Structure du projet et des objets

I/ Les Foncteurs

La plupart des algorithmes de la librairie standard, ainsi que quelques méthodes des classes qu'elle fournit, donnent la possibilité à l'utilisateur d'appliquer une fonction aux données manipulées, quel que soit leur type. Ces fonctions peuvent être utilisées pour différentes tâches, comme pour comparer deux objets par exemple, ou tout simplement pour en modifier la valeur.

Cependant, la librairie standard n'utilise pas ces fonctions directement, mais a plutôt recours à une abstraction des fonctions : les foncteurs. Un foncteur n'est rien d'autre qu'un objet dont la classe définit l'opérateur fonctionnel ‘()’. Les foncteurs ont la particularité de pouvoir être utilisés exactement comme des fonctions puisqu'il est possible de leur appliquer leur opérateur fonctionnel selon une écriture similaire à un appel de fonction. Cependant, ils sont un peu plus puissants que de simples fonctions, car ils permettent de transporter, en plus du code de l'opérateur fonctionnel, des paramètres additionnels dans leurs données membres. Les foncteurs constituent donc une fonctionnalité extrêmement puissante qui peut être très pratique en de nombreux endroits. En fait, dans l’absolue, toute fonction peut être transformée en foncteur. Les algorithmes de la librairie standard peuvent donc également être utilisés avec des fonctions classiques moyennant cette petite transformation.

Le but de ce projet et donc de gérer ces foncteurs de manière à les utiliser le plus simplement possible. Pour cela il faudra se heurter à deux problématiques : les paramètres passés à ces foncteurs notamment leurs types, mais aussi le type de la valeur retourner par ces foncteurs.

Ainsi dans un premier temps nous allons créer un exemple simple permettant de bien visualiser l’intérêt des foncteurs en réutilisant les opérateurs de bases et la librairie standard.

L’exemple le plus concret est celui du ‘for\_each’, fonction présente dans le fichier d'entête standard algorithm.

std::for\_each(debut,fin,function);

‘for\_each’ applique ici une fonction aux éléments d'une séquence, un tableau par exemple, parcourue à l’aide d’itérateurs. Le problème que pose cette fonction est donc la présence de paramètres, en effet ici elle ne doit posséder qu’un seul paramètre, et le type de celui-ci n’est pas important tant qu’il est identique au type des éléments de la séquence.

Les foncteurs sont une façon élégante de solutionner le problème de paramètre seul dans le for\_each. Car, imaginons que nous ayons un vecteur de string et que nous voudrions écrire son contenu dans un fichier. Nous voudrions naturellement pouvoir spécifier le fichier où l'écrire, mais malheureusement, le seul paramètre de la fonction sera occupé par une chaine de caractères.

Nous allons donc voir dans la partie suivante qu’avec l’utilisation des foncteurs et des templates, il nous sera possible d’éliminer cette contrainte.

II/Structure du projet et des objets

Dans cette partie nous allons expliquer comment nous souhaitons utiliser des foncteurs basiques avec plusieurs paramètres et comment nous allons gérer les valeurs et les types retournés par ces foncteurs. Pour cela nous nous sommes inspiré de la librairie standard et de ces utilisations.

L’objectif d’un foncteur est donc de pouvoir englober toutes les fonctions, pour réaliser cela, nous avons donc dû répondre à une première question : Qu’elles sont les informations strictement nécessaire à une fonction. En réponse à cette question, nous avons trouvé les éléments suivant :

* Un type de retour
* Une liste de type d’arguments
* Un pointeur vers des instructions

Un pointeur vers un objet dans le cas des méthodes.

La liste d’arguments est sans aucun doute le point le plus complexe à traiter, en effet, le nombre d’argument d’une fonction n’est pas fixe, et ses types peuvent être différents. Pour résoudre ce problème nous avons donc pensé à utiliser à un équivalent de l’objet java « Object » (l’objet « source » de tous les objets du langage) et d’en faire un vecteur, mais il n’existe pas en C++, et même si c’était le cas, la mise en place serait probablement un peu trop complexe.

Donc, pour commencer de manière simple et éventuellement travailler de manière itérative nous avons décidé de gérer uniquement des fonctions ne prennent un ou deux paramètres de même type. Les fonctions qui prennent un paramètre et un seul sont dites « unaires », alors que les fonctions qui prennent deux paramètres sont qualifiées de « binaires ». Afin de faciliter la création de foncteurs utilisables avec ces algorithmes, nous allons définir donc deux classes de base qui définissent les deux catégories de fonctions utilisés par ces algorithmes. Ces classes de base sont les suivantes :

|  |
| --- |
| template <class Arg, class Result>  struct f\_unary\_function  {  typedef Arg argument\_type;  typedef Result result\_type;  };  template <class Arg1, class Arg2, class Result>  struct f\_binary\_function  {  typedef Arg1 first\_argument\_type;  typedef Arg2 second\_argument\_type;  typedef Result result\_type;  }; |

Par la suite, on pourrait même aller plus loin en créant d’autres classes permettant d’utiliser autant de paramètres que nous le souhaitons, on se contentera de faire une librairie applicable uniquement sur les fonctions « unaires » et « binaires » décrites ci-dessus.

En ce qui concerne les foncteurs, nous souhaitions commencer par des exemples simples de manière à bien saisir le comportement de ceux-ci. Nous avons donc réalisé des foncteurs utilisant les opérateurs classiques tels que plus et moins définis ci-dessous

|  |
| --- |
| template <class T>  struct f\_plus : f\_binary\_function<T, T, T>  {  T operator()(const T &operande1, const T &operande2) const   {  return operande1 + operande2;  }  };  template <class T>  struct f\_minus : f\_binary\_function<T, T, T>  {  T operator()(const T &operande1, const T &operande2) const   {  return operande1 - operande2;  }  }; |

Nous avons choisi d’ajouter une couche permettant d’appeler les foncteurs « unaires » et « binaires », nous avons donc créé deux classes permettant d’appeler les foncteurs unaires et binaires en fonction de leur nombre d’arguments, mais nous avons aussi créer des templates permettant de gérer les types de ces arguments de manière générique.

|  |
| --- |
| template <class T, class F>  T f\_call(T i, T j, F foncteur)  {  // Applique l'opérateur fonctionnel au foncteur  // avec comme arguments les deux premiers paramètres :  return foncteur(i, j);  }  template <class T, class F>  T f\_call(T arg, F functor)  {  // Applique l'opérateur fonctionnel au foncteur  // avec comme arguments le paramètres :  return foncteur(arg);  } |

Quand à l’utilisation de ces foncteurs voici comment cela se présente :

|  |
| --- |
| int main(void)  {  // Construit le foncteur de somme :  f\_plus<int> foncteur\_plus;  f\_minus<int> foncteur\_minus;  // Utilise ce foncteur pour faire faire une addition  // à la fonction "applique" :  cout << f\_call(22, 20, foncteur\_plus) << endl;  cout << f\_call(22,20, foncteur\_minus) << endl;    return 0;  } |

Enfin pour reprendre notre exemple de départ dans lequel nous souhaitions écrire le contenue d’un fichier d’entier dans un tableau afin de le projeter sur n’importe quel flux. Un problème se pose alors, en effet l’algorithme ‘for\_each()’ applique une opération à chaque élément d’une séquence, or une fonction opère sur ses paramètres et ne tiens pas d’état à jour d’un appel a un l’autre alors que nous souhaitons que l’opération projette tous les entiers sur le même flux.

La clé de ce problème et de remplacer la fonction qui traite l’opération par un objet dont le constructeur pourrait saisir une référence sur le flux désiré et de surcharger l’operateur ‘()’ pour faire en sorte que chaque instance de la classe puisse se comporter comme une fonction capable de savoir sur quel flux écrire.

Nous allons donc développer la solution à l’aide d’un foncteur un peu moins classique que ceux vus plus haut.

Celui ci sera chargé d’écrire le contenu d’une séquence dans un flux, ainsi le flux sera passé en paramètre et le type de celui-ci.

|  |
| --- |
| template <class T>  class Afficher  {  std::ostream &os\_;  public:  Afficher(std::ostream &os) throw(): os\_(os) { }  void operator()(const T &val)  {  os\_ << val << ' ';  }  }; |

Le foncteur sera utilisé de la façon suivante :

|  |
| --- |
| int main(void)  {  using std::vector;  using std::ifstream;  using std::for\_each;  vector<int> v;  ifstream ifs ("in.txt");  for (int val; ifs >> val; v.push\_back(val));  cout<< v.size() << endl;  // le pointeur sur Afficher<int> est remplacé par une instance  // anonyme de la classe Afficher<int> appliquée au flux cout  for\_each(v.begin(), v.end(), Afficher<int>(cout));    return 0;  } |

Conclusion

Nous avons donc expliqué comment utiliser les foncteurs et surtout comment régler les problèmes de passage de paramètres a ceux-ci et gérer leurs valeurs de retour.

Nous avons également créé une librairie de foncteurs de fonction « unaires » et « binaires », c'est-à-dire, des fonctions avec un argument et deux de mêmes types, ainsi qu'une liste de fonction qui peuvent être appelé avec les foncteurs.

Le code présent dans le repository git présente les exemples un peu plus complets présentés dans ce rapport.