Développement Orienté Aspect et Optimisation

Pierre Schweitzer

Mardi 4 Juin 2013

LIMOS UMR CNRS 6158

Clermont Université - Université Blaise Pascal

LPC UMR CNRS 6533
Clermont Université - Université Blaise Pascal

Introduction

- Insuffisance de l'objet
- Notion d'aspect
- Garder un code simple
- Adapter un code selon l'aspect
- Focus sur le C++

Plan

Plan

Problématiques

Comment gérer simplement...

- La concurrence ?
- L'authentification ?
- La trace d'exécution ?
- La sécurité d'exécution ?
- ...

Programmation Orientée Aspect

Notion d'aspect

- Surcouche à la notion d'objet déjà existante
- Donne un contexte d'exécution au code
- Pas (ou peu) besoin de modifier le code pour utiliser l'aspect
- Fonctionnalités supplémentaires

Programmation Orientée Aspect

Notion d'aspect

- Surcouche à la notion d'objet déjà existante
- Donne un contexte d'exécution au code
- Pas (ou peu) besoin de modifier le code pour utiliser l'aspect
- Fonctionnalités supplémentaires

Surcouche à l'objet ?

⚠ L'aspect peut être présent sur du code non objet

L'aspect, une longue existence

Implémentation en C

- Via le préprocesseur et les annotations
- Les pragmas permettent de modifier le code
- Également certains mot-clés du langage

L'aspect, une longue existence

Implémentation en C

- Via le préprocesseur et les annotations
- Les pragmas permettent de modifier le code
- Également certains mot-clés du langage

Exemple en C

```
1  #pragma pack(2)
2  static const int a = 2;
```

Plan

Approches naïves

Variadic macros

- Permet d'encapsuler des fonctions
- Néanmoins, lourdeur de développement
- Nécessite de modifier l'appel des fonctions

Approches naïves

Variadic macros

- Permet d'encapsuler des fonctions
- Néanmoins, lourdeur de développement
- Nécessite de modifier l'appel des fonctions

Variadic templates

- Permet également d'encapsuler des fonctions
- Plus facile d'utilisation que les macros
- Code plus complexe à écrire
- Fait appel à des notions très spécifiques du C++11

Approches naïves

Variadic macros

- Permet d'encapsuler des fonctions
- Néanmoins, lourdeur de développement
- Nécessite de modifier l'appel des fonctions

Variadic templates

- Permet également d'encapsuler des fonctions
- Plus facile d'utilisation que les macros
- Code plus complexe à écrire
- Fait appel à des notions très spécifiques du C++11

Limitation des normes

- ✖ Ceci n'est possible qu'avec C++11
- ✖ Ou C99 pour les macros

Utilisation des templates 1/5

Idées directrices

- Utilisation des pointeurs de fonction
- Appel de l'aspect qui appelle la fonction

Utilisation des templates 1/5

Idées directrices

- Utilisation des pointeurs de fonction
- Appel de l'aspect qui appelle la fonction

Démonstration avec un aspect simple

- Objectif : afficher la valeur de chaque argument de la fonction
- Appel de l'aspect qui appelle la fonction AspectDisplayArgs()
- Affichage effectif dans AspectpDisplayArgs()

Utilisation des templates 2/5

Fonction de l'aspect

```
1  template<typename FunctionType, typename... ArgsType>
2  auto AspectDisplayArgs(FunctionType * Function, const
       ArgsType&... FunctionArgs) -> decltype(Function(
       FunctionArgs...))
3  {
4      AspectpDisplayArgs(0, FunctionArgs...);
5      return Function(FunctionArgs...);
6  }
```

Utilisation des templates 3/5

Affichage effectif des arguments

```
1  template<typename ArgType, typename... ArgsType>
2  void AspectpDisplayArgs(unsigned int Count, const ArgType&
       FunctionArg, const ArgsType& ... FunctionArgs)
3  {
4      std::cout << "Arg " << Count << ": " << FunctionArg
           << std::endl;
5      AspectpDisplayArgs(++Count, FunctionArgs...);
6  }
```

Utilisation des templates 3/5

Affichage effectif des arguments

```
1  template<typename ArgType, typename... ArgsType>
2  void AspectpDisplayArgs(unsigned int Count, const ArgType&
       FunctionArg, const ArgsType& ... FunctionArgs)
3  {
4      std::cout << "Arg " << Count << ": " << FunctionArg
           << std::endl;
5      AspectpDisplayArgs(++Count, FunctionArgs...);
6  }
```

Fin de récurrence

```
1  void AspectpDisplayArgs(unsigned int Count) {
2  }
```

Utilisation des templates 4/5

Fonction de test

```
1  int Add(int a, int b) {
2      return a + b;
3  }
```

Utilisation des templates 4/5

Fonction de test

```
1  int Add(int a, int b) {
2      return a + b;
3  }
```

Appel de l'aspect

```
1      Res = AspectDisplayArgs(&Add, 9, 11);
2      std::cout << "Sum of 9 and 11 is: " << Res << std::
           endl;
```

Utilisation des templates 5/5

Sortie

```
1  Arg 0: 9
2  Arg 1: 11
3  Sum of 9 and 11 is: 20
```

Utilisation des templates 5/5

Sortie

```
1 Arg 0: 9
2 Arg 1: 11
3 Sum of 9 and 11 is: 20
```

Inconvénients

- ⚠ Forte dépendance à C++11
- ⚠ Concepts assez peu maîtrisés

Utilisation des templates 5/5

Sortie

```
1 Arg 0: 9
2 Arg 1: 11
3 Sum of 9 and 11 is: 20
```

Inconvénients

- ⚠ Forte dépendance à C++11
- ⚠ Concepts assez peu maîtrisés

Code complet et commentaires

- Disponible sur http://www.pschweitzer.fr/?p=23

Utilisation d'un préprocesseur

But du préprocesseur

- Parsera le code
- Effectuera des ajouts de code à des points de coupure ("pointcuts")
- Utilise sa propre syntaxe / son propre code
- Nécessite une compilation du code généré ensuite

Utilisation d'un préprocesseur

But du préprocesseur

- Parsera le code
- Effectuera des ajouts de code à des points de coupure ("pointcuts")
- Utilise sa propre syntaxe / son propre code
- Nécessite une compilation du code généré ensuite

Outils

- AspectC++
- Outils développés en interne

AspectC++

Le préprocesseur

- "Imitation" de AspectJ (Java)
- Projet de recherche de Olaf Spinczyk
- Logiciel open source et gratuit

AspectC++

Le préprocesseur

- "Imitation" de AspectJ (Java)
- Projet de recherche de Olaf Spinczyk
- Logiciel open source et gratuit

Utilisation

- Définition de pointcuts (expressions régulières)
- Implémentation de l'aspect (advices) dans un fichier
- Parsing du code source avec l'aspect (recherche des join points)
- Compilation du rendu final

Plan

Profiling avec AspectC++

Règle d'or

- Ne jamais optimiser sans avoir profilé !
- Profiling : récupération des informations de performance
- Identification des zones lentes et critiques du logiciel

Profiling avec AspectC++

Règle d'or

- Ne jamais optimiser sans avoir profilé !
- Profiling : récupération des informations de performance
- Identification des zones lentes et critiques du logiciel

Outils de profiling

- Callgrind (via valgrind)
- gprof (via gcc/g++)

Profiling avec AspectC++

Règle d'or

- Ne jamais optimiser sans avoir profilé !
- Profiling : récupération des informations de performance
- Identification des zones lentes et critiques du logiciel

Outils de profiling

- Callgrind (via valgrind)
- gprof (via gcc/g++)

Limites des outils existant

- ✖ Absence du véritable support de JNI (valgrind)
- ✖ Output peu lisible (gprof)

Implémentation d'un aspect TProfiler 1/4

Lignes directrices

- Utilisation de pointcuts sur toutes les fonctions
- Implémentation d'un before & d'un after
- Requête des données de performances
- Utilisation des ticks et du temps CPU
- Hooking sur main() pour avoir ses propres paramètres
- Code dans un fichier séparé, compilable sans aspect

Implémentation d'un aspect TProfiler 1/4

Lignes directrices

- Utilisation de pointcuts sur toutes les fonctions
- Implémentation d'un before & d'un after
- Requête des données de performances
- Utilisation des ticks et du temps CPU
- Hooking sur main() pour avoir ses propres paramètres
- Code dans un fichier séparé, compilable sans aspect

Génération de graphes d'appel

- Pour fournir des données cohérentes en sortie
- Permet de visualiser le fonctionnement du programme
- Utilisable avec kcachegrind

Implémentation d'un aspect TProfiler 2/4

Déclaration de l'aspect

```
1  aspect TProfiler {
2      pointcut functions() = "% ...::%(...)";
3      pointcut mainargs() = "% main(int, char**)";
4      pointcut profiler() = "% TProfiler::%(...)";
```

Implémentation d'un aspect TProfiler 2/4

Déclaration de l'aspect

```
1  aspect TProfiler {
2      pointcut functions() = "% ...::%(...)";
3      pointcut mainargs() = "% main(int, char**)";
4      pointcut profiler() = "% TProfiler::%(...)";
```

Implémentation des fonctionnalités

```
1  advice execution(functions() && !profiler()) : before() {}
2  advice execution(functions() && !profiler()) : after() {}
```

Implémentation d'un aspect TProfiler 2/4

Déclaration de l'aspect

```
1  aspect TProfiler {
2      pointcut functions() = "% ...::%(...)";
3      pointcut mainargs() = "% main(int, char**)";
4      pointcut profiler() = "% TProfiler::%(...)";
```

Implémentation des fonctionnalités

```
1  advice execution(functions() && !profiler()) : before() {}
2  advice execution(functions() && !profiler()) : after() {}
```

Implémentation complète

- https://github.com/HeisSpiter/acprof

Implémentation d'un aspect TProfiler 3/4

Sortie

```
1 Function: double Madhava(unsigned int) (functions.cpp:31),
      calls: 1, total ticks: 1270000, total user time: 576000ms,
       total kernel time: 696000ms
2 Function: double Pow(double,double) (functions.cpp:7), calls:
      999999, total ticks: 730000, total user time: 328000ms,
      total kernel time: 424000ms
3 Function: double Sqrt(double) (functions.cpp:3), calls:
      1999997, total ticks: 1280000, total user time: 852000ms,
      total kernel time: 328000ms
4 Function: int main(int,char **) (main.cpp:4), calls: 1, total
      ticks: 6900000, total user time: 700000ms, total kernel
      time: 218000ms
5 Function: unsigned int AskMax(const std::basic_string<char>
       &) (functions.cpp:44), calls: 5, total ticks: 0, total
      user time: 0ms, total kernel time: 0ms
```

Implémentation d'un aspect TProfiler 4/4

Inconvénients

- ✖ Impossibilité de définir un pointcut sur les bibliothèques
- ✖ C++11 & CUDA non supportés
- ✖ Impossibilité de définir un pointcut sur des fonctions avec template
- ⚠ Logiciel encore en cours de développement

Plan

Conclusions

Aspect

- Nombreux avantages à l'utilisation de l'aspect
- Souvent utilisé en entreprise
- Néanmoins peu aisé d'utilisation en C++

Conclusions

Aspect

- Nombreux avantages à l'utilisation de l'aspect
- Souvent utilisé en entreprise
- Néanmoins peu aisé d'utilisation en C++

Optimisation

- Maître mot : profiling
- Outils dédiés pour ça
- Possibilité de le faire en aspect

Des questions ?

Merci pour votre attention !