Comment concevoir correctement une bibliothèque en C++11 ? Eric Niebler essaie de répondre à cette question dans une présentation lors du dernier Meeting C++.

C++11 Library Design

Pour ceux qui ne connaissent pas Eric Niebler, c'est le mainteneur de Boost Proto, on avait commencé à traduire sa série d'article sur le C++ Expressif sur Developpez.com (très intéressant, mais pas forcement simple à appréhender).

Certains points de cette présentation sont intéressants à mettre en avant. Elle décompose le design des bibliothèques en 3 niveaux et proposer un certain nombre de questions à se poser lorsque l'on crée une bibliothèque :

1. les fonctions
2. les classes
3. les modules

Il faut penser nos codes en termes de bibliothèques (diapo 5)

Une bibliothèque “est une collection d'implémentation de comportements, écris en termes de langage, qui possède une interface bien définie permettant d'invoquer ces comportements. [...] Le comportement est prévu pour la réutilisation [...].” (traduction de Wikipedia, “Library (software)”, Octobre 2013, cité dans la présentation).

Cette définition devrait correspondre à tous les codes que l'on écrit et donc tous les codes que l'on écrit devrait être des bibliothèques :) (je simplifie un peu l'idée).

Le design des fonctions (diapo 14)

- Est-ce que ma fonction est facile à appeler correctement ?
- Est-ce que ma fonction est difficile à appeler de façon incorrecte ?
- Est-ce que ma fonction est efficace à appeler ?
- Est-ce que ma fonction est efficace à appeler avec un minimum de copie ?
- Est-ce que ma fonction est efficace à appeler avec un minimum d'aliasing ?
- Est-ce que ma fonction est efficace à appeler sans allocations inutiles de ressources ?
- Est-ce que ma fonction est facile composer avec d'autres fonctions ?
- Est-ce que ma fonction est utilisable dans les constructions d'ordre supérieur ?

Eric Niebler donne les règles suivantes pour le design des fonctions (les explications ne sont pas détaillées ici, voir la présentation pour les détails) :

1. Guideline 1 : continuez de passer les valeurs en lecture seule par référence constantes (sauf les types simples).
2. Guideline 2 : passez les arguments “sink” par valeur (je ne sais pas comment traduire ce terme, cela correspond aux arguments qui doivent être conservés par la fonction, par exemple un objet à mettre dans un conteneur. Ces arguments doivent donc être copié ou déplacer, pas simplement avoir une référence).
3. Guideline 3 : encapsulez les états d'un algorithme dans un objet qui implémente cet algorithme.

Le design des classes (diapos 37 et 38)

- Comment concevoir une classe en C++11 qui utilise au mieux le C++11 ?
- Comment concevoir une classe en C++11 qui utilise correctement les fonctionnalités du langage ?
  - Qui utilise correctement la copie, affectation, déplacement, range-based for, etc. ?
  - Qui se compose bien avec les autres types ?
  - Qui peut être utilisé n'importe où ? (pile, tas, stockage statique, dans une expression constante, etc.)
- Comment concevoir une classe en C++11 qui utilise correctement les fonctionnalités de la bibliothèque ?
  - Avec les algorithmes génériques ?
  - Avec les conteneurs ?
- Est-ce que mes types peuvent être copiés et affectés ?
- Est-ce que mes types peuvent être passés et retournés efficacement comme paramètres ?
- Est-ce que mes types peuvent être insérés efficacement dans un vecteur ?
- Est-ce que mes types peuvent être triés ?
- Est-ce que mes types peuvent être utilisés dans une map ? Dans une map ordonnée ?
- Est-ce que mes types peuvent être utilisés avec des itérateurs ? (si c'est une collection)
- Est-ce que mes types peuvent être utilisés avec des flux (streams) ?
- Est-ce que mes types peuvent être utilisés pour déclarer des constantes globales ?

Et les guidelines suivants :

1. Guideline 4 : rendez vos types réguliers (dans la mesure du possible).
2. Guideline 5 : rendez vos opérations de déplacement noexcept (dans la mesure du possible).
3. Guideline 6 : l'état “déplacer depuis un autre objet” doit faire partie de votre invariant de classe.
4. Guideline 7 : si le guideline 6 n'a pas de sens, le type n'est pas déplaçable.
5. Corollaire : tous les types déplaçables doivent avoir un état valide par défaut, peu coûteux à construire.

Le design des modules (diapos 56)

Pour le design des modules, Eric Niebler fait un constat simple : le C++11 offre peu de fonctionnalités pour garantir la création de modules évolutifs et réutilisables. Que peut-on attendre d'un langage pour cette problématique ?

- Quelles sont les fonctionnalités pour éviter les dépendances cycliques et obtenir une hiérarchie des composants ?
- Quelles sont les fonctionnalités pour décomposer de gros composants en composants plus petits ?
- Quelles sont les fonctionnalités pour faciliter l'extensibilité des composants ? (ajout de nouvelles fonctionnalités)
- Quelles sont les fonctionnalités permettant de versionner les composants ?

Le C++ ne propose rien pour le support des modules, le chargement dynamique de bibliothèques et pour le versioning des interfaces ou des implémentations. Le C++11 a apporté une petite évolution avec les espaces de noms inline (si vous n'avez jamais compris l'intérêt pratique des inline namespaces, regardez les diapos 58 à 63, Eric Niebler présente un exemple concret de leur utilisation), mais il manque encore beaucoup de choses. Les modules (study group 2 du comité de normalisation du

C++) devrait répondre à cette problématique, mais ils ne seront pas disponibles avant le C++17.

Les guidelines :

1. Guideline 8 : mettez tous les éléments d'interface dans un espace de noms de versioning.
2. Guideline 9 : mettez l'espace de noms courant en inline.
3. Guideline 10 : mettez les définitions de type dans des espaces de noms (non inline) bloquant l'ADL (Argument-dependent name lookup) et exportez les en utilisant using.
4. Guideline 11 : préférez des objets-fonctions globaux constant (constexpr) aux fonctions libres nommées (sauf pour les points de personnalisation documentés).

Conclusion

On voit bien avec cette présentation en quoi le C++11 nécessite de changer ses habitudes de codage si on souhaite améliorer l'évolutivité de ses codes, mais aussi le travail qui reste à faire pour améliorer encore le C++. La transition peut se faire en douceur, puisque le C++ reste rétro-compatible (un code ancien continuera de fonctionner). L'idéal est donc de faire évoluer le code existant (en totalité ou en partie) à l'occasion d'une maintenance ou d'une évolution. Il ne faut pas non plus avoir peur de repenser l'architecture globale de son code (la difficulté étant de réussir à “oublier” l'architecture existante pour trouver la “bonne” architecture et pas simplement mettre des pansements sur une architecture bancale).