Modules in C++

Coming soon …

Modules in C++

Coming soon …

40 ans après Modula (Niklaus Wirth – 1975)

… probablement pas dans le standard ISO C++17

Modules in C++

Coming soon …

40 ans après Modula (Niklaus Wirth – 1975)

… probablement pas dans le standard ISO C++17

Un avant goût aujourd'hui grâce à Clang

Modules in C++

Qu'entend-t-on derrière le terme “module” ?

- Une bibliothèque avec une interface bien définie
- Le contrôle précis de ce qui est exposé (exporté) au clients
- Le masquage des détails d'implémentation
- Le masquage des dépendances

Modules in C++

N'est-on pas capable de faire tout cela aujourd'hui ?

- Au niveau binaire : oui
- Au niveau du langage C+ : partiellement

Modules in C++

N'est-on pas capable de faire tout cela aujourd'hui ?

- Au niveau binaire : oui

- Shared/Dynamic library
- Export et masquage supportés au niveau de la phase d'édition de lien (linker), et runtime (“loader”)
- Editeur de lien et runtime sont language-agnostic : dans le code source, on ne fait qu'instruire l'éditeur de lien en accolant des attributs. Hors du périmètre du langage C++, non standardisé.
- Variation irréconciables dans les détails de fonctionnement entre les plateformes (format ELF vs Mac-O vs PE, PIC vs Non-PIC, export/import implicite vs explicite)

→ Effort de standardisation possible pour les fonctionnalités de base, mais pas de révolution : pas de format binaire spécifique C++.

Modules in C++

N'est-on pas capable de faire tout cela aujourd'hui ?

- Au niveau du langage C++: partiellement

- Contrôle d'accès (ex. “private”) dans les classes,
- Forward-declaration
  Idiome PImpl (Pointer Implementation)
- Utilisation d'un namespace “detail” pour séparer les détails d'implantation (ex. Boost)

Modules in C++

Masquage des dépendances ?

Contrôle d'accès “private” != Visilibité :

- Insuffisant : l'inclusion des en-têtes est transitive et exposée aux clients

```
#include <bar.hpp>

class foo {
        private:
                bar bar_;
};
```

Modules in C++

Masquage des dépendances ?

Forward-declaration :

- Insuffisant : les déclarations sont exposées aux clients
- Contraignant : force l'utilisation de pointeur/référence

```
class bar;

class foo {
        private:
                bar & bar_;
};
```

Modules in C++

Masquage des dépendances ?

L'idiome “PImpl” (Pointer Implémentation) :

- Masquage complet
- Contraignant : force l'utilisation de pointeur/référence, sans type!
- Lourd à mettre en oeuvre

```
class foo {
        private:
                void * pimpl_;
};
```

Modules in C++

Le coupable ?

Modules in C++

Le coupable ?
Le préprocesseur

Modules in C++

Effets indésirables du préprocesseur :

- #include => pas de possibilité de contrôle ce qui est importé / rexporté aux clients, donc modularité incomplète / floue
- Temps de compilation accrus par l'inclusion repétées des mêmes en-têtes pour chaque unité de compilation (compensé par l'utilisation des en-têtes précompilées)
- Les macros polluent l'espace de nom (cf. la bibliothèque standard) et impactent jusqu'à la lisibilité des msg d'erreurs
- Abus des macros (surtout en C)
- IDE à la traîne (mauvaise indexation, outils de refactoring bloqués par les macros ...)
- Include-guards == boilerplate

→ Aucune évolution pour palier à ces défauts : blocus du comité de standardisation (ex, même “#pragma once” est non-standard)

- #if est indispensable pour la portabilité

Modules in C++

Vers une évolution :

- Commité de standardisation : “*SG2, Modules. Work on possible refinement or replacement for the header-based build model*”
- Probablement pas pour C++17
- L'implémentation précède la standardisation (ex. Clang)

Modules in C++

Commité de standardisation : Paper N4465

```
module modfoo;

import modbar;

export {
        class foo {
                private:
                        bar b_;
        };
}
```

Modules in C++

Evolution en douceur avec Clang :

- aucun changement dans le code source!

```
module modfoo; // déclaré dans fichier xxx.modulemap

#include <bar.hpp> // interprété comme import modbar; si fichier xxx.modulemap présent

export { // implicite
        class foo {
                private:
                        bar b_;
        };
}
```

Modules in C++

Evolution en douceur avec Clang :

- aucun changement dans le code source!
- simples ajout de fichiers xxx.modulemap à côté des en-têtes

```
module modfoo {
        header "foo.hpp"
        link "foo"
        use modbar
}

module modbar {
        header "bar.hpp"
        link "bar"
}
```

Modules in C++

Evolution en douceur avec Clang :

- aucun changement dans le code source!
- simples ajout de fichiers xxx.modulemap
- implémenté par évolution de la technique de pré-compilation des en-têtes (AST sérialisé, indexé, chargement partiel au besoin)
- aucun changement dans le build system : totalement transparent, cache géré par clang
- déployer les fichiers modulemap