

Pensées sur la compatibilité binaire


Qt by Nokia

par Thiago Macieira traducteur : Thibaut Cuvelier ([Site web](#)) ([Blog](#)) Qt Labs

Date de publication : 13/08/2009

Dernière mise à jour : 03/10/2010

Vous avez toujours rêvé de savoir à quoi correspondait *ABI*, *name mangling*, et autres *virtual tables* ? Vous savez déjà ce dont il s'agit, mais vous aimeriez en savoir plus ? Alors cet article est pour vous !

Cet article est une traduction autorisée de  **Some thoughts on binary compatibility**, par Thiago Macieira.

N'hésitez pas à commenter cet article !

| | |
|----------------------------------|---|
| I - L'article original..... | 3 |
| II - Introduction..... | 3 |
| III - Historique..... | 3 |
| IV - Ce dont l'ABI a besoin..... | 4 |
| V - Aujourd'hui..... | 5 |
| VI - Divers..... | 6 |

I - L'article original




Les *Qt Labs Blogs* sont des blogs tenus par les développeurs de Qt, concernant les nouveautés ou les utilisations un peu extrêmes du framework.

Nokia, Qt et leurs logos sont des marques déposées de *Nokia Corporation* en Finlande et/ou dans les autres pays. Les autres marques déposées sont détenues par leurs propriétaires respectifs.

Cet article est la traduction du billet  **Some thoughts on binary compatibility**, par Thiago Macieira.

II - Introduction


Ces derniers mois, j'ai été très peu présent dans la blogosphère. Je rassemblais des idées pour un blog, que je compte toujours écrire, concernant la suprématie fonctionnelle de Qt. Bien que ça ne vienne pas, j'ai décidé de coucher par écrit quelques pensées sur la compatibilité binaire.

J'ai récemment mis à jour la  **KDE Techbase** ( **aussi disponible en français**). Plus précisément, l'article sur la  **compatibilité binaire avec le C++**. Soit dit en passant, c'est le troisième résultat sur Google avec "binary compatibility". J'ai essayé de détailler les explications des *dos* et *don'ts*, des choses à faire et à ne pas faire. Après que j'aie écrit la partie sur la surcharge de virtuels d'une base non primaire, quelqu'un, par IRC, m'a demandé d'écrire **quelques exemples**.

Pour écrire ces exemples, j'ai dû revoir mes notions de *name mangling*, de *virtual tables*... J'ai même dû essayer d'apprendre l'ABI de Microsoft Visual Studio. Cela m'a pris un certain temps, mais j'ai trouvé **un article** avec quelques informations sur le *name mangling* et sur Visual Studio. Je suis aussi heureux d'avoir pris le temps de parfaire mes connaissances, puisque j'ai trouvé un autre exemple de choses à ne pas faire (le cas "*virtual override with covariant return of different top address*").

III - Historique

Commençons avec un peu d'histoire.

Tandis qu'UNIX a toujours été fortement lié au C, le marché du DOS n'a jamais eu de telle relation. Bien sûr, les applications étaient développées en C, même au début des années 80, mais le DOS n'a jamais fourni de librairie C. Non. Pour accéder aux services du DOS, on devait modifier quelques valeurs dans un registre et déclencher une interruption ( **Int 21**). Ceux qui implémentaient un compilateur C devaient aussi développer leur propre librairie C.

Aussi, rappelez-vous cette époque, où il n'existait ni DLL, ni librairie partagée : pas de compatibilité binaire à maintenir. La conséquence est que chaque compilateur décidait de lui-même comment implémenter la séquence d'appel et l'ABI. Ce qui confère quelques responsabilités à l'appelant et à l'appelé, comme désigner quels registres du processeur sont utilisés pour passer les paramètres, lesquels doivent être utilisés pour le travail, lesquels doivent être préservés, qui nettoie la pile, la taille de certains types, l'alignement...

Et, comme tout le monde pouvait l'espérer, chaque compilateur gérait cela différemment.

Dans le monde UNIX, les choses étaient un peu plus standardisées, puisqu'une librairie C existait depuis un certain temps, et qu'un compilateur de référence était précisé sur chaque poste. Pour pouvoir utiliser la même librairie C, les compilateurs devaient implémenter une ABI identique.

Mais la situation devient plus intéressante quand on parle de C++ ! Si, sur les systèmes UNIX, les conventions d'appel C sont plus ou moins bien standardisées, il n'en est pas encore question pour le C++. Le C est un langage de bas niveau, au point que, en y regardant assez longtemps, il est possible de lire le code assembleur derrière chaque

instruction en C. Cependant, d'expérience, quand une telle chose arrive, on a des visions, et il vaudrait mieux rentrer chez soi se reposer... Le C++ introduit de nombreux concepts par dessus le C (surcharge, appels virtuels, héritage multiple, héritage virtuel, retours covariants, polymorphisme, *templates*, références...). Ce qui signifie autant de points sur lesquels les compilateurs peuvent différer.

Une chose intéressante est arrivée en 2000 : les processeurs Itanium. Pas à cause des processeurs eux-mêmes, mais à cause de la documentation qu'ils suscitèrent.

Il n'était pas suffisant de connaître l'ensemble des instructions de l'architecture (voir, à ce sujet, le **Software Developer's Manual**), les développeurs en voulaient plus, avaient besoin de plus que tout cela, et Intel s'est incliné.

- **Software Conventions & Runtime Architecture Guide** ;
- **Processor-specific Application Binary Interface** ;
- **C++ ABI**.

GCC a clairement adopté cette ABI sur Itanium, mais, puisque le code était là, et que le code était supérieur à ce que GCC avait, GCC l'a appliqué aussi aux autres plateformes. Il est ainsi intéressant de voir cette ABI utilisée sur des systèmes qui n'ont rien à voir avec l'Itanium, ni avec UNIX, comme Symbian sur un système ARM.

IV - Ce dont l'ABI a besoin

Il est clair que l'ABI doit correspondre à tout programme C++. C'est-à-dire qu'elle doit supporter toutes les fonctionnalités du langage. En débutant avec la plus simple innovation du C++ par rapport au C, on peut voir à quel point les choses deviennent intéressantes.

En C, une fonction n'est identifiée que par son nom. Il ne peut pas y avoir deux fonctions avec le même nom dans la portée globale. C++, juste à côté, lui, supporte la surcharge. Il peut donc y avoir des fonctions avec le même nom, qui ne diffèrent que par les arguments qu'elles prennent. Nous venons à la conclusion que chaque ABI doit encoder ces différentes fonctions avec différents noms. Elle doit encoder toutes les différences par rapport au langage C, mais elle pourrait aussi choisir d'encoder des informations utiles au débogage.

Ensuite viennent les appels virtuels. Lors d'un appel à une fonction virtuelle d'une classe, le compilateur doit se débrouiller pour écrire le code qui permet d'appeler toute fonction réimplémentée, sans savoir, *a priori*, quelles sont ces réimplémentations. Le seul moyen d'y arriver est de stocker, quelque part dans la classe, l'endroit où l'appel virtuel est censé arriver. La majorité, voir l'ensemble, des compilateurs se contentent d'ajouter un pointeur, quelque part dans l'objet, sur la *virtual table*, qui est une liste de pointeurs de fonctions pour chaque appel de virtuel. Chaque classe C++ avec des fonctions virtuelles possède une telle table, liste des virtuels de la classe, les hérités et les surchargés.

Mais les *virtual tables* contiennent, d'habitude, plus d'informations que de simples pointeurs sur fonction, comme le *typeinfo* d'une classe C++ et les *offsets* des bases virtuelles internes à l'objet. Ce dernier type d'information est illustré par le cas de l'héritage multiple en forme de diamant. Une classe Base, deux classes A et B, dérivant virtuellement de Base, ainsi qu'une classe finale X, dérivant de A et de B. Quand elles sont prises séparément, A et B sont similaires, le contenu de Base est stocké quelque part dans les structures de A et de B. Cependant, à l'intérieur de X, les choses changent, puisqu'il doit allouer une copie de A, une copie de B, mais une seule et unique copie de Base.

Le compilateur doit donc encoder quelque part l'endroit où il a placé le sous-objet Base. Une manière de faire est de simplement garder un pointeur, membre à la fois de A et de B. Une autre est de mettre l'*offset* de début de A et de B dans la *virtual table* - ce qui permet de sauver deux octets en mémoire.

En combinant ces trois concepts, on couvre 99% des besoins d'une ABI pour un programme C++ moyen.

V - Aujourd'hui

Nous pouvons classer les ABI C++ en trois grands groupes : les systèmes utilisant l'ABI C++ pour Itanium, l'ABI C++ de Microsoft, et les autres. La dernière catégorie est un groupe pour tous les autres compilateurs, comme le compilateur de Sun Studio pour Solaris, IBM Visual Age pour AIX, HP aCC pour HP-UX sur PA-RISC. Ce dernier peut aussi être utilisé sur Itanium : sur cette plateforme, il utilisera l'ABI C++ pour Itanium. Nous ne testons pas activement la compatibilité entre les binaires de Qt et les spécificités de ces trois compilateurs pour la simple et bonne raison que nous n'avons pas d'indice quant à leur nature, voire leur existence. Je n'ai connaissance d'aucun document décrivant l'ABI qu'ils implémentent, et je n'ai pas vraiment l'envie de les étudier, vu la valeur ajoutée que cela apporterait. Après tout, la majorité des utilisateurs de ces plateformes et de Qt le compilent depuis les sources.

L'ABI C++ pour Itanium est un concept moderne, créé après la standardisation du C++, avec toutes ses fonctionnalités connues. Elle a été créée par des gens qui essayaient de résoudre un problème : comment rendre le C++ possible, sans superflu ? Ils sont arrivés avec une ABI assez élégante : les classes comportant des fonctions virtuelles se voyaient ajouté, pour premier membre, un pointeur caché sur leur *virtual table*. La table étant elle-même créée en même temps que la première fonction membre virtuelle non *inline*. La *virtual table* contient, aux *offsets* positifs, les pointeurs vers les fonctions virtuelles membres ; aux *offsets* négatifs, les *typeinfo* et les *offsets* qui doivent gérer l'héritage multiple.

Même le *name mangling* est lisible, pour les types simples ! La règle de base : le C ne devrait pas l'utiliser, pour éviter tout risque de collision. Ils ont donc choisi le préfixe `_Z`.

Prenons un exemple : `_ZN7QString7replaceEiiPK5QChari`. Nous pouvons le diviser comme suit.

```
_Z N 7QString 7replace E i i PK5QChar i
```

Ce que nous pouvons lire !

- `_Z` : préfixe des symboles C++ ;
- `N 7QString 7replace E` : nom composé :
- `7QString` : nom de 7 caractères, `QString` ;
- `7replace` : nom de 7 caractères, `replace`.
- `i` : entier ;
- `P` : pointeur ;
- `K5QChar` : nom constant de 5 caractères `QChar`, soit `const QChar`.

Mises bout à bout, ces informations nous donnent le prototype de la fonction : `QString::replace(int, int, const QChar*, int)`.

À l'opposé, Microsoft. Leurs compilateurs ont choisi d'encoder les noms de fonction avec autant de détails que possible, comme, par exemple, sa visibilité. De plus, pour d'obscures raisons, qui n'ont, sans doute, plus de sens de nos jours, le *mangling* à la sauce Microsoft est... insensible à la casse ! Ce serait comme si, un jour, quelqu'un avait actionné un levier, et, hop, le C++ est devenu insensible à la casse : voici le schéma qu'ils utiliseraient. GCC, de toute évidence, serait perdu dans un monde C++ insensible à la casse.

C'est clairement un héritage du DOS. Cela se montre encore quand on remarque que la taille du pointeur (proche ou lointain) est aussi encodée, comme l'appel de fonction (proche ou lointain). Ces choses ne sont plus utilisées de nos jours, mais l'ABI peut encore les stocker.

La même fonction que précédemment reçoit donc un *mangling* assez différent, avec MSVC.


```
?replace@QString@@QAEAAV0@HHPBVQChar@@H@Z
```

Mais nous pouvons quand même le lire.

- ? : préfixe du C++ ;
- replace : le nom le plus à droite ;
- @ : séparateur ;
- QString : classe clôturant ;
- @@ : fin du nom de fonction ;
- Q : publique, proche (ni virtuelle, ni statique) ;
- A : pas de qualificateur CV (ni constante, ni volatile) ;
- E : __thiscall ;
- A : référence (possiblement proche) ;
- A : référence non modifiée (pas un const X &) ;
- V @ : classe et délimiteur ;
- 0 : premier nom de classe précédemment vue ;
- H : entier ;
- H : entier ;
- P : pointeur normal (non constant) ;
- B : type constant (PB correspond à const X * ; QA, à X * const) ;
- V @ : classe et délimiteur ;
- VQChar@ : classe QChar avec délimiteur ;
- H : entier ;
- @ : fin de la liste des paramètres ;
- Z : fonction, ou classe de stockage de texte ou de code.

Ce qui se lit ainsi :

```
public: class QString & near __thiscall QString::append(int, int, const class QChar *, int)
```

 *Il est important de noter quelques points.*

- L'utilisation de ?, au lieu de quelque chose que l'on peut écrire en C ;
- La même lettre peut avoir des significations différentes en fonction de sa place ;
- Les types des variables sont assignés par ordre alphabétique depuis une liste, et n'essayent pas de ressembler au nom du type ;
- class est encodé explicitement (V), tout comme les structures (U), les unions (T) et les énumérations (W4) ;
- L'encodage de la séquence d'appel (__thiscall) et du déplacement (near).

D'un côté, le *mangling* à la Microsoft permet de produire des messages d'erreur plus détaillés, et différencie les types, les séquences d'appel, qui ne se résolvent pas en un même symbole. D'un autre côté, il encode des détails qui n'ont aucune importance à l'appel (il différencie struct et class ; protected, private et public).

VI - Divers

Merci à **yan** et **superjaja** pour leurs relectures et encouragement lors de la traduction, et à **koopajah** pour sa relecture orthographique ! Mais aussi à **dvdibly**, pour son autre correction orthographique !

Cet article n'est qu'une partie de l'iceberg ABI dévoilé par Thiago Macieira. Après avoir dégrossi la chose, il s'est attaqué à une partie, très importante : **les conventions d'appel**.