Editions ENI

C++
Les fondamentaux du langage

(2e édition)

Collection
Ressources Informatiques

Extrait

Chapitre 5
Les univers de C++

1. L'environnement Windows

1.1 Les programmes Win32

La plate-forme Win32 a très vite été supportée par C++ car Microsoft a lancé dès 1992 l'un des premiers environnements intégrés pour ce langage ; il s'agissait de Visual C++, et à cette époque, de nombreux architectes étaient convaincus de l'intérêt d'un langage objet proche du langage C pour créer des applications fonctionnant dans des environnements graphiques.

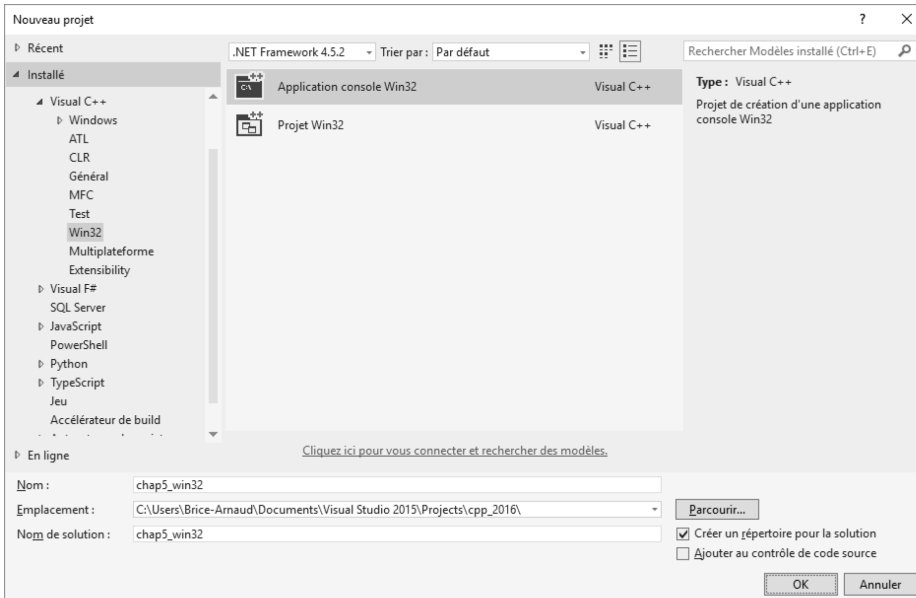
Toutefois, la programmation d'une application fenêtrée, sans framework, n'est pas une tâche aisée. Il n'en demeure pas moins que la plateforme Win32 propose plusieurs formats d'applications : applications "console", services Windows, bibliothèques dynamiques (DLL) et bien entendu les applications graphiques fenêtrées.

Remarque

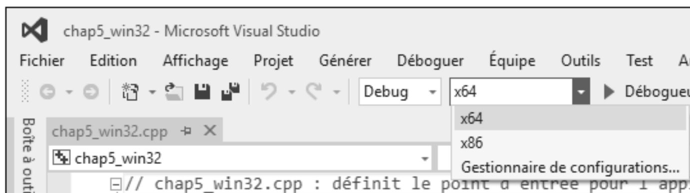
Il n'existe pas de modèle de projet "Win64" car Windows reste un système très ancré dans le mode 32 bits. Toutefois, Visual Studio sait compiler en mode 64 bits depuis un projet Win32.

1.2 Choix du mode de compilation

La création d'un projet s'effectue avec la commande **Fichier - Nouveau Projet** :



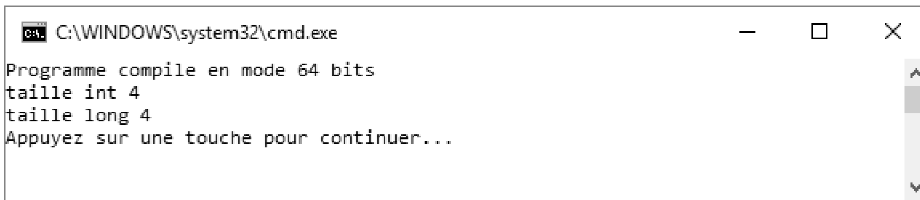
Depuis le gestionnaire de configuration, le mode de compilation 64 bits est activé :



Le programme qui suit est intéressant ; il indique que la taille des types `int` et `long` est identique à un mode 32 bits.

```
printf("Programme compilé en mode 64 bits\n");
printf("taille int %d\n", sizeof(int));
printf("taille long %d\n", sizeof(long));
```

© Editions ENI - All rights reserved



Le mode 64 bits recommande au compilateur de produire des instructions microprocesseur 64 bits, mais cela ne veut pas dire que les types sont ajustés en conséquence. Autrement dit, un calcul sur un type `__int64` pourra prendre plus de temps en 32 bits qu'en 64, mais `__int64` reste toujours 64 bits et `int` reste sur 4 octets quel que soit le mode de compilation. Il s'agit là d'une particularité du compilateur de Microsoft ; d'autres compilateurs peuvent adopter un comportement différent.

2. L'environnement .NET

2.1 Le code managé et la machine virtuelle CLR

Comme Java, .NET fait partie de la famille des environnements virtualisés. Le compilateur C++ managé ne produit pas du code assembleur directement exécuté par le microprocesseur mais un code intermédiaire, lui-même exécuté par une machine "virtuelle", la CLR (*Common Language Runtime*). Cette couche logicielle reproduit tous les composants d'un ordinateur – mémoire, processeur, entrées-sorties… Cette machine doit s'adapter au système d'exploitation qui l'héberge, et surtout optimiser l'exécution du code.

Remarque

Il y a là une différence importante entre .NET et Java. Alors que Microsoft a évidemment fait le choix de se concentrer sur la plate-forme Windows, d'autres éditeurs ont porté Java sur leurs OS respectifs – Unix, AIX, Mac OS, Linux… La société Novell a cependant réussi le portage de .NET sur Linux, c'est le projet Mono.

Cet environnement virtualisé a une conséquence très importante sur les langages supportés ; les types de données et les mécanismes objets sont ceux de la CLR. Comme Microsoft était dans une démarche d'unification de ses environnements, plusieurs langages se sont retrouvés éligibles à la CLR, quitte à adapter un peu leur définition. Le projet de Sun était plutôt une création ex nihilo, et de ce fait seul le langage Java a réellement été promu sur la machine virtuelle JVM.

2.2 Les adaptations du langage C++ CLI

Le terme CLI signifie *Common Language Infrastructure* ; c'est l'ensemble des adaptations nécessaires au fonctionnement de C++ pour la plate-forme .NET / CLR.

2.2.1 La norme CTS

Le premier changement concerne les types de données. En successeur de C, le C++ standard a basé sa typologie sur le mot machine (`int`) et le caractère de 8 bits (`char`). Confronté à des problèmes de standardisation et d'interopérabilité, Microsoft a choisi d'unifier les types de données entre ses différents langages.

Les types primitifs, destinés à être employés comme variables locales (boucles, algorithmes...), sont des types "valeurs". Leur zone naturelle de stockage est la pile (*stack*), et leur définition appartient à l'espace de noms **`System`** :

`wchar_t`	`System::Char`	Caractère unicode
`signed char`	`System::SByte`	Octet signé
`unsigned char`	`System::Byte`	Octet non signé
`double`	`System::Double`	Décimal double précision
`float`	`System::Float`	Décimal simple précision
`int`	`System::Int32`	Entier 32 bits
`long`	`System::Int64`	Entier 64 bits
`unsigned int`	`System::UInt32`	Entier 32 bits non signé

© Editions ENI - All rights reserved

`unsigned long`	`System::UInt64`	Entier 64 bits non signé
`short`	`System::Int16`	Entier court 16 bits
`bool`	`System::Boolean`	Booléen
`void`	`System::Void`	Procédure

Voici un exemple utilisant à la fois la syntaxe habituelle de C++ et la version "longue" pour définir des nombres entiers. Cette dernière ne sera utilisée qu'en cas d'ambiguïté, mais il s'agit en fait de la même chose.

```
int entier = 4; // alias de System::Int32
System::Int32 nombre = entier;
```

Les structures managées (**`ref struct`**) sont composées de champs reprenant les types ci-dessus. Elles sont créées sur la pile et ne sont pas héritables.

```
ref struct Point
{
public:
    int x,y;
} ;

int main(array<System::String ^> ^args)
{
    Point p; // initialisation automatique sur la pile : pas de gcnew
    p.x = 2;
    p.y = 4;

    return 0;
}
```

Comme tous les types valeurs, les structures managées n'autorisent pas l'effet de bord à moins qu'une référence explicite n'ait été passée (voir ci-dessous les références suivies).

Par opposition les classes managées (**`ref class`**) sont créées sur le tas et correspondent davantage au fonctionnement des classes (structure) du C/C++ standard. Évidemment, elles sont héritables, manipulées par références, et instanciées par l'opérateur **`gcnew`**.

```
ref class Personne
{
public:
```

```
    String^ nom;
    int age;
} ;

int main(array<System::String ^> ^args)
{
    Personne^ pers = gcnew Personne();
    pers->nom = "Marc";
    pers->age = 35;

    return 0;
}
```

Naturellement, les structures et les classes managées supportent la définition de méthodes, C++ est bien un langage objet !

Les langages .NET sont fortement typés, en évitant autant que possible la généricité "fourre-tout" du `void*` issu du C, ou du `Variant` de VB. En opposition, la norme CTS prévoit que l'ensemble des types de données (valeurs ou référence) héritent d'un comportement commun `System::Object`. Nous découvrirons un peu plus loin comment cette caractéristique est exploitée dans les classes de collections d'objets.

2.2.2 La classe System::String

Parmi les types intrinsèques à .NET on trouve la classe `System::String` pour représenter les chaînes. Elle n'est pas aussi intégrée au langage C++ CLI que dans C#, mais cependant elle offre exactement les mêmes services. Microsoft l'a de plus aménagée pour faciliter la manipulation et la conversion avec `char*` :

Voici un premier exemple de construction de chaînes. On remarquera l'emploi facultatif du symbole `L` devant les littérales de chaîne, ainsi que le symbole ^ dont la signification sera expliquée un peu plus tard.

```
char* c = "Bonjour C++";
String ^ chaine1 = gcnew String(c); // construction à partir d'un char*
String ^ chaine2 = L"C++ CLI c'est formidable"; // littérale de chaîne .NET
String ^ chaine3 = "Un univers à découvrir"; // idem

Console::WriteLine(chaine1);
Console::WriteLine(chaine2 + ". " + chaine3);
```

© Editions ENI - All rights reserved

Editions ENI

Maîtrisez Qt 5

Guide de développement d'applications professionnelles

(2e édition)

Collection
Epsilon

Extrait

Chapitre 6
QtCore

1. Objectifs

Dans ce chapitre nous vous présenterons le module principal, le plus important de Qt : le module QtCore. Celui-ci contient un certain nombre de classes indispensables pour la création d'applications, telles que `QString` pour les chaînes de caractères, `QTimer` pour la programmation d'événements, `QFile` et `QDir` pour la gestion des fichiers et encore bien d'autres.

Ce module est un des plus importants, car sur lui reposent tous les autres modules. Il intègre une grande partie des fonctionnalités offertes par Qt et décrites dans le chapitre Les fondations de Qt, en particulier la classe `QObject`.

2. Intégration et interactions

Le module QtCore est intégré à votre projet grâce à l'ajout du mot-clé `core` dans la déclaration `QT` de votre fichier `.pro`.

```
QT += core
```

Remarque

Il est pratiquement impossible de s'en passer, sauf si vous faites de la programmation purement C++ sans utiliser aucune des fonctionnalités de Qt. N'importe quel autre module de Qt aura besoin de celui-ci.

3. Découverte de QtCore

QtCore est un ensemble d'API qui fournit essentiellement des types de base comme les chaînes de caractères Unicode, les types fichier et répertoire pour l'accès aux disques, des interfaces pour les entrées-sorties et des mécanismes de gestion des événements.

Cet ensemble de classes est nommé, dans l'univers Qt, un module. Il contient notamment la classe `QObject`, la mère des classes contenues dans les autres modules de Qt et de vos futures classes. En outre, la plupart des classes de QtCore sont utilisées par les autres API de Qt.

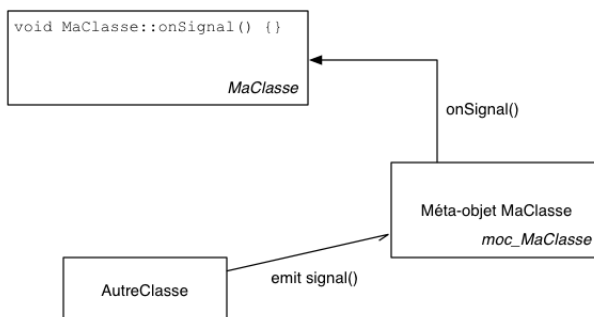
3.1 Le méta-objet

Comme expliqué dans le chapitre Anatomie d'une application, Qt utilise un système de méta-objets pour faciliter le travail du développeur dans la gestion des signaux et l'appel asynchrone des fonctions.

La méta-classe correspondant au méta-objet est directement issue de l'héritage de la classe `QObject` et de l'usage de la macro `Q_OBJECT`.

Lorsque vous utilisez la macro `Q_OBJECT`, vous ajoutez plusieurs déclarations de fonctions à votre classe. Ces fonctions seront utilisées par Qt pour réaliser les appels de slots et générer les signaux.

Comme vous n'implantez pas vous-même ces fonctions surnuméraires, c'est Qt qui va le faire pendant la phase de construction du programme. L'utilitaire `moc` est chargé de générer le code de ces fonctions associées à votre classe.



© Editions ENI - All rights reserved

Les fonctions ajoutées par Qt permettent d'étendre les capacités de votre code source afin de le rendre capable d'émettre des signaux et d'en recevoir.

La fonction `emit()` ou l'opérateur `emit` ainsi que les fonctions déclarées dans les sections `signals` et `slots` ne sont jamais appelés directement par vos classes mais redirigés par et vers les méta-objets.

Remarque

Qt ajoute des fonctionnalités et des attributs à votre classe et lui permet de s'intégrer parfaitement dans le système de gestion des signaux et slots.

Ce mécanisme est totalement transparent pour vous, les fonctions sont générées automatiquement et leur contenu dépend à la fois du contexte de votre programme et de la manière dont vous faites dépendre vos objets les uns des autres, c'est-à-dire la manière dont vous émettez vos signaux ou faites vos appels asynchrones ainsi que l'affinité des objets (voir chapitre Les fondations de Qt - La classe QObject - Affinité de thread).

3.2 Introspection

Chaque classe qui hérite de QObject a un méta-objet associé, une instance de la classe `QMetaObject`. Cette classe fournit des mécanismes d'introspection pendant l'exécution :

`QMetaObject::className() : const char*` - retourne le nom de la classe.

`QMetaObject::superClass() : QMetaObject*` - retourne une instance du méta-objet de la classe mère.

`QMetaObject ::constructorCount() : int` - retourne le nombre de constructeurs de la classe.

`QMetaObject::constructor(int index) : QMetaMethod` - retourne les métadonnées d'un constructeur.

`QMetaObject::methodCount() : int` - retourne le nombre de méthodes de la classe.

`QMetaObject::method(int index) : QMetaMethod` - retourne les métadonnées d'une fonction particulière de la classe.

`QMetaObject::propertyCount() : int` - retourne le nombre de propriétés de la classe, incluant celles de la classe mère.

`QMetaObject::property(int index) : QMetaProperty` - retourne les métadonnées d'une propriété de la classe.

3.3 Le méta-type

Pour être en mesure de traiter vos classes dans les signaux et les appels asynchrones, Qt utilise les méta-types de la classe `QMetaType`.

Il s'agit d'un système générique qui traite chaque classe comme un type générique et est capable, en quelque sorte, d'encapsuler un objet d'une classe inconnue dans un signal ou un appel de fonction.

Ainsi, si vous avez besoin de transmettre une de vos classes dans un signal vous devez la déclarer comme un méta-type :

```
qRegisterMetaType<MaClasse>("MaClasse") ;
```

Les conditions pour utiliser cette fonction sont que votre classe doit posséder un constructeur par défaut sans argument, un constructeur de copie et un déstructeur public.

De plus, si vous déclarez des `typedef` sur des types déjà enregistrés, vous devez enregistrer ces `typedef` aussi :

```
typedef QString MaChaine ;
qRegisterMetaType<MaChaine>("MaChaine") ;
```

Si vous souhaitez utiliser votre classe avec les conteneurs à template de Qt (`QList<T>`, `QHash<T>`, etc), vous devez effectuer une déclaration du méta-type en utilisant la macro `Q_DECLARE_METATYPE()` :

```
class MaClasse {...};
Q_DECLARE_METATYPE(MaClasse);

QList<MaClasse> maListe;
```

© Editions ENI - All rights reserved

3.4 Instanciation dynamique

Le méta-type vous permet aussi d'instancier dynamiquement n'importe quelle classe par son nom, sous la forme d'une chaîne de caractères.

Par exemple, vous pouvez avoir un fichier texte qui comprend une liste de noms de classes à instancier et à exécuter et effectuer ce travail pendant le fonctionnement de l'application.

La fonction `QMetaType::type(const char*) : int` retourne un identifiant de méta-type instanciable, ou `QMetaType ::UnknownType` si le méta-type n'existe pas ou s'il n'est pas déclaré.

La fonction `QMetaType::create(int) : void*` retourne un pointeur vers une nouvelle instance du méta-type pour lequel vous avez fourni un identifiant.

```
QStringList nomsDeClasse;
QFile fichier("lefichier.txt");
//Lecture du fichier et récupération des noms de classes à instancier

foreach(QString nomDeClasse, nomsDeClasse) {
   qDebug() << "Instanciation du type " << nomDeClasse;

   //Conversion du nom de classe en const char* ASCII
   int id = QMetaType::type(nomDeClasse.toLatin1());
   if(id != QMetaType::UnknownType) {
      void *instance = QMetaType::create(id);

      if(!instance) { //L'instanciation a échoué }
   } else { // Le type n'existe pas }
}
```

4. Créer et piloter des threads

La gestion des *threads* est un point extrêmement délicat, surtout lorsque l'on souhaite que notre application soit multiplateforme.

Il existe principalement deux types de gestion des *threads* très différents : POSIX et WIN32. Le premier est une implantation Unix et le second existe uniquement sous Windows.

Pour satisfaire l'interopérabilité des programmes, Qt propose une classe de gestion des *threads* : `QThread`. Attention toutefois, il ne s'agit pas d'un *thread* à proprement parler, mais davantage d'un contrôleur pour un *thread* système.

4.1 Paralléliser des tâches

Il faut être très prudent lorsque l'on décide de créer des *threads*. La plupart du temps, un développeur peu expérimenté créera un ou plusieurs *threads* pour résoudre un problème de performances ou de blocage de l'interface graphique. C'est une mauvaise interprétation du problème et une solution inadaptée, le blocage de l'interface graphique est révélateur d'un problème conceptuel (voir chapitre Les fondations de Qt).

Avec Qt, il y a très peu besoin de créer des *threads*, et surtout pas pour résoudre ce genre de problèmes. Les threads doivent être utilisés uniquement pour paralléliser des sous-processus. Par exemple, le parcours d'un arbre avec un algorithme de graphes peut être parallélisé avec un ou plusieurs *threads*.

S'il s'agit simplement d'un long travail à effectuer, sans parallélisme, il faudra utiliser la *run-loop* du thread principal et les signaux/*slots* pour intégrer ce travail de manière fluide dans le flux de l'application.

Il existe plusieurs façons de créer et de gérer des *threads* : les *threads* pilotés dont on souhaite connaître l'état et que l'on souhaite pouvoir démarrer et arrêter de manière maîtrisée ainsi que les tâches autonomes que l'on exécute et qui se terminent « dans l'anonymat ».

4.1.1 Les threads pilotés

La classe `QThread` propose un mécanisme d'encapsulation d'une tâche à exécuter dans un *thread*.

Il s'agit d'un conteneur pour une tâche et non d'un *thread* au sens du système. Un `QThread` possède sa propre *run-loop*.

© Editions ENI - All rights reserved