

Editions ENI

C++
Les fondamentaux du langage
(2^e édition)

Collection
Ressources Informatiques

Table des matières

Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence ENI de l'ouvrage **RI2CPP** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

- 1. Objectifs de ce livre 11
- 2. Travaux pratiques. 12
 - 2.1 Le langage de script Lambda Basic. 12
 - 2.2 Le tableur InCell 13
 - 2.3 Les pages web dynamiques EZ-Pages 14
- 3. À qui s'adresse ce livre ? 15

Chapitre 1 Introduction

- 1. Notions clés. 17
 - 1.1 Principales caractéristiques du langage C++ 17
 - 1.2 Programmation orientée objet 19
 - 1.3 Environnement de développement et fichier makefile 21
 - 1.3.1 Choix d'un EDI 21
 - 1.3.2 Construction d'un fichier makefile 22
 - 1.4 Organisation d'un programme C++ 26
 - 1.4.1 Codes sources 27
 - 1.4.2 Modules objets 30
 - 1.4.3 Bibliothèques (librairies) 33
 - 1.4.4 Exécutable 34
 - 1.5 Préprocesseur. 35
 - 1.6 Choix d'un compilateur 36
 - 1.7 Éditeur de liens 37

2.	Bases de la programmation C++	37
2.1	Déclaration de variables	38
2.1.1	Utilité des variables	38
2.1.2	Portée des variables	40
2.1.3	Syntaxe de déclaration	42
2.1.4	Types de données	42
2.2	Instructions de tests et opérateurs	51
2.2.1	Instructions de tests	51
2.2.2	Opérateurs	54
2.3	Instructions de boucle	60
2.3.1	La boucle for	60
2.3.2	La boucle while	62
2.3.3	La boucle do	62
2.3.4	Les instructions de débranchement	63
2.4	Tableaux	63
2.5	Fonctions et prototypes	65
2.5.1	Déclaration d'une fonction	66
2.5.2	Fonctions et procédures	66
2.5.3	Appel des fonctions	68
2.5.4	Gestion des variables locales	68
2.5.5	Définition de fonctions homonymes (polymorphisme)	69
2.5.6	Fonctions à nombre variable d'arguments	70
2.5.7	Attribution de valeurs par défaut aux arguments	72
2.5.8	Fonctions en ligne	73
2.5.9	Fonctions externes de type C	73
2.5.10	Fonctions récursives	74
2.5.11	La fonction main()	75
2.6	Pointeurs	77
2.6.1	Pointeurs sur des variables	78
2.6.2	Pointeurs et tableaux	82
2.6.3	Allocation de mémoire	84
2.6.4	Arithmétique des pointeurs	87
2.6.5	Pointeurs de pointeurs	88

2.6.6	Pointeurs de fonctions	89
2.7	Références	97
2.8	Constantes	100
2.8.1	Constantes symboliques	100
2.8.2	Le type void.	101
2.8.3	Les alias de type : typedef	102
2.8.4	Constantes et énumérations	102
3.	Exceptions	103
3.1	Les approches de bas niveau.	103
3.1.1	Drapeaux et interruptions	103
3.1.2	Traitement des erreurs en langage C.	105
3.2	Les exceptions plus sûres que les erreurs.	107
3.3	Propagation explicite	108
3.4	Types d'exceptions personnalisés	109
3.4.1	Définition de classes d'exception.	109
3.4.2	Instanciation de classes	110
3.4.3	Classes d'exception dérivées.	111
3.5	Prise en charge d'une exception et relance	112
3.6	Exceptions non interceptées	113
3.7	Acquisition de ressources	113
4.	Travaux pratiques.	116
4.1	Prise en main de l'interprète Lab	116
4.1.1	Structure de la solution	116
4.1.2	Le dossier framework	117
4.1.3	Le dossier langage	118
4.1.4	Le dossier scriptboxes.	118
4.1.5	Utiliser l'interprète	119
4.2	Le code de la boucle principale.	120
4.3	Affichage dans Labshow.	122

Chapitre 2

De C à C++

1. Programmation structurée	125
1.1 Structures	126
1.1.1 Constitution d'une structure	127
1.1.2 Instanciation de structures	128
1.1.3 Instanciation avec l'opérateur new	129
1.1.4 Pointeurs et structures	130
1.1.5 Organisation de la programmation	131
1.2 Unions	132
1.3 Copie de structures	135
1.4 Création d'alias de types de structure	137
1.5 Structure et fonction	138
1.5.1 Passer une structure par valeur comme paramètre	138
1.5.2 Passer une structure par référence comme paramètre	139
1.5.3 Passer une structure par adresse comme paramètre	139
1.5.4 De la programmation fonctionnelle à la programmation objet	140
2. Gestion de la mémoire	141
2.1 Alignement des données	142
2.2 Allocation de mémoire interprocessus	143
3. La bibliothèque standard du C	143
3.1 Les fonctions communes du langage C <stdlib.h>	143
3.2 Chaînes <string.h>	145
3.3 Fichiers <stdio.h>	147
4. Travaux pratiques	152
4.1 Chargement de scripts dans Lab	152
4.2 Supprimer les erreurs liées à la librairie non sécurisée	154

Chapitre 3

Programmation orientée objet

1. Classes et instances	157
1.1 Définition de classe.....	158
1.1.1 Les modificateurs d'accès	159
1.1.2 Organisation de la programmation des classes.....	162
1.2 Instanciation.....	164
1.3 Constructeur et destructeur	166
1.3.1 Constructeur.....	166
1.3.2 Le pointeur this.....	167
1.3.3 Destructeur	168
1.3.4 Destructeur virtuel.....	169
1.4 Allocation dynamique	170
1.5 Constructeur de copie	172
2. Héritage.....	173
2.1 Dérivation de classe (héritage).....	173
2.1.1 Exemple de dérivation de classe.....	174
2.1.2 Héritage public, protégé et privé	178
2.1.3 Appel des constructeurs.....	179
2.2 Polymorphisme	180
2.2.1 Méthodes polymorphes	180
2.2.2 Conversions d'objets.....	181
2.3 Méthodes virtuelles et méthodes virtuelles pures	181
2.4 Héritage multiple	186
2.4.1 Notations particulières.....	187
2.4.2 Conséquences sur la programmation	190
3. Autres aspects de la POO	192
3.1 Conversion dynamique	192
3.1.1 Conversions depuis un autre type.....	192
3.1.2 Opérateurs de conversion	194
3.1.3 Conversions entre classes dérivées	195

3.2	Champs et méthodes statiques	196
3.2.1	Champs statiques	196
3.2.2	Méthodes statiques	197
3.3	Surcharge d'opérateurs	202
3.3.1	Syntaxe	202
3.3.2	Surcharge de l'opérateur d'indexation	204
3.3.3	Surcharge de l'opérateur d'affectation	204
3.3.4	Surcharge de l'opérateur de conversion	205
3.4	Fonctions amies	205
3.5	Adressage relatif et pointeurs de membres	207
3.5.1	Notations	208
3.5.2	Construction d'un middleware orienté objet	209
3.6	Programmation générique	215
3.6.1	Modèles de fonctions	216
3.6.2	Modèles de classes	220
4.	Travaux pratiques	225
4.1	Utilisation de l'héritage de classes dans l'interprète Lab	225
4.2	Des pointeurs de membres pour des fonctions callback	227

Chapitre 4

La bibliothèque Standard Template Library

1.	Introduction	231
2.	Organisation des programmes	232
2.1	Espaces de noms	232
2.1.1	Utilisation complète d'un espace de noms	234
2.1.2	Espace de noms réparti sur plusieurs fichiers	235
2.1.3	Relation entre classe et espace de noms	236
2.1.4	Déclaration de sous-espaces de noms	238
2.2	Présentation de la STL	239

3. Flux C++ (entrées-sorties)	239
3.1 Généralités	240
3.2 Flux intégrés	241
3.3 État d'un flux	241
3.4 Mise en forme	241
3.5 Flux de fichiers	243
3.6 Flux de chaînes	245
3.7 Paramètres locaux	246
4. Classe string pour la représentation des chaînes de caractères	248
4.1 Représentation des chaînes dans la STL	249
4.2 Mode d'emploi de la classe string	250
4.2.1 Fonctions de base	250
4.2.2 Intégration dans le langage C++	252
4.2.3 Fonctions spécifiques aux chaînes	254
5. Conteneurs dynamiques	256
5.1 Conteneurs	257
5.1.1 Insertion d'éléments et parcours	258
5.1.2 Itérateurs	258
5.1.3 Opérations applicables à un vecteur	259
5.2 Séquences	260
5.2.1 Conteneurs standards	260
5.2.2 Séquences	262
5.2.3 Adaptateurs de séquences	264
5.2.4 Conteneurs associatifs	268
5.3 Algorithmes	270
5.3.1 Opérations de séquence sans modification	270
5.3.2 Opérations de séquence avec modification	271
5.3.3 Séquences triées	272
5.3.4 Algorithmes de définition	273
5.3.5 Minimum et maximum	273

5.4	Calcul numérique	273
5.4.1	Limites des formats ordinaires.	274
5.4.2	Fonctions de la bibliothèque	275
5.4.3	Fonctions de la bibliothèque standard et classe valarray	276
6.	Travaux pratiques.	277
6.1	La classe Variant	277
6.2	La méthode to_string	280
6.3	Prise en charge des tableaux dans Lab	281

Chapitre 5

Les univers de C++

1.	L'environnement Windows	285
1.1	Les programmes Win32	285
1.2	Choix du mode de compilation	286
2.	L'environnement .NET	287
2.1	Le code managé et la machine virtuelle CLR	287
2.2	Les adaptations du langage C++ CLI	288
2.2.1	La norme CTS	288
2.2.2	La classe System::String	290
2.2.3	Le garbage collector	292
2.2.4	Construction et destruction d'objets.	294
2.2.5	La référence de suivi % et le handle ^	298
2.2.6	Le pointeur interne et les zones épinglées.	301
2.2.7	Les tableaux et les fonctions à nombre variable d'arguments.	303
2.2.8	Les propriétés	305
2.2.9	Les délégués et les événements	307
2.2.10	Les méthodes virtuelles	309
2.2.11	Les classes abstraites et les interfaces	311

2.3	Le framework .NET	312
2.3.1	Les références d'assemblages	313
2.3.2	L'espace de noms System::IO	314
2.3.3	L'espace de noms System::Xml	315
2.3.4	L'espace de noms System::Data	316
2.3.5	L'espace de noms System::Collections	317
2.3.6	L'espace de noms System::Collections::Generic	318
2.3.7	Le portage de la STL pour le C++ CLI	319
2.4	Les relations avec les autres langages : C#	319
3.	Travaux pratiques	321
3.1	Une application en C++ pour .NET : le tableur InCell	321
3.1.1	Architecture du tableur	321
3.1.2	La feuille de calcul	323
3.1.3	L'ordonnanceur de calcul	331
3.1.4	Zoom sur l'évaluateur	336
3.1.5	L'interface graphique	337
3.2	Les pages web dynamiques EZ-Pages	338
3.2.1	Fonctionnement des pages web dynamiques	338
3.2.2	Implémentation de BufferBox	340
3.2.3	Réalisation du questionnaire web EZHandler	340
3.2.4	Tests	342
3.2.5	Pour aller plus loin	343

Chapitre 6 Des programmes C++ efficaces

1.	Dépasser ses programmes	345
1.1	Oublier les réflexes du langage C	345
1.2	Gestion de la mémoire	347
1.3	Concevoir des classes avec soin	348
1.4	Y voir plus clair parmi les possibilités de l'héritage	349
1.5	Analyser l'exécution d'un programme C++	350

2. La conception orientée objet (COO).....	351
2.1 Relation entre la POO et la COO	351
2.1.1 L'approche initiale de C++	351
2.1.2 UML et C++	352
2.2 Les design patterns	355
 Index	 357

Editions ENI

Maîtrisez Qt 5

Guide de développement d'applications professionnelles

(2^e édition)

Collection
Epsilon

Table des matières



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.

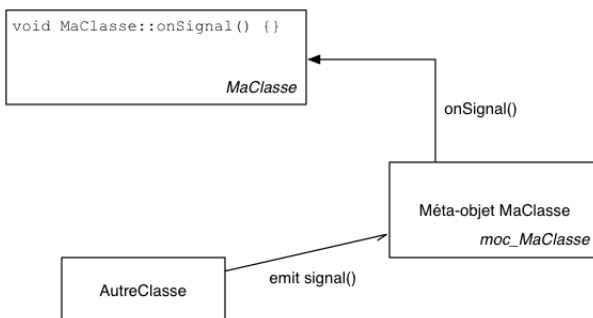
3.1 Le méta-objet

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.



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 destructeur 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;
```


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*.