faire plusieurs chapitres sur l'internationalisation. Séparer les chaines, la localisation, le formatage. Peut être faire un chapitre sur ICU et sur Qt::tr ?

Les chaînes de caractères internationales

De nos jours, avec les progrès des moyens de communication, en particulier d'internet, il est très facile de partager ses programmes à l'international. Des outils de partage du code, comme par exemple GitHub que vous verrez par la suite, permettent de mettre en place des équipes de développement collaboratif, même sur des projets lancés par une seule personne. Il est donc conseillé, dès le début d'un projet, de penser en termes de travail en équipe et d'écrire ses programmes en anglais (code, commentaires, documentation).

Le corollaire à cela est qu'un programme pourra facilement être utilisé par des personnes qui ne parlent pas l'anglais (en premier lieu vous peut-être) et il sera intéressant de pouvoir afficher des chaînes de caractères dans d'autres langages.

Et c'est à ce niveau que la gestion des chaînes de caractères peut devenir complexe. Par défaut, les chaînes en C++ sont basées sur le système anglais, qui ne contient aucun caractère accentué. En français par exemple, nous utilisons des accents aigu, grave ou circonflexe, des trémas. D'autres langues utilisent plus de types d'accents (Suédois). D'autres encore utilisent des alphabets complètement différents (Russe, Chinois).

En plus des alphabets différents, il faut également gérer d'autres problématiques. Par exemple, en anglais, le point est utilisé comme séparateur décimal. En français, nous utilisons la virgule. En anglais américain, une date s'écrit sous la forme mois-jour-année. En français, nous écrivons jour-mois-année.

Certaines langages s'écrivent de gauche à droite, d'autres de droite à gauche, d'autres encore de haut en bas.

On voit bien, par ces quelques exemples, que la gestion des langages est quelque chose de complexe. Dans ce chapitre, nous allons commencer par étudier les principaux types d'encodage des caractères et les chaînes utilisables dans ce contexte en C++.

Pour une gestion complète des langues ayant des alphabets complexes, il sera préférable d'utiliser des bibliothèques dédiées, comme ICU. Ce chapitre est une simple introduction aux problématiques posées lorsque l'on souhaite prendre en compte l'internationalisation des programmes.

La norme ASCII

Vous avez vu dans les chapitres précédents que le type `char` est un type d'entier un peu particulier. Il est possible de manipuler une variable de ce type comme un nombre (initialisation avec un nombre, addition, soustraction, etc.), mais lorsque l'on affiche une variable de ce type, cela affiche un caractère. Il est également possible d'afficher la valeur numérique d'un caractère de la façon suivante :

main.cpp

```
#include <iostream>

int main() {
    int const i { 'a' };
    std::cout << std::showbase << std::hex << i << ' ' <<
std::dec << i << std::endl;
}
```

Dans ce code, le littérale caractère 'a' est de type `char`. Il est converti en type `int` lors de l'initialisation de la variable `i`. Ce code ne produit pas d'erreur de conversion implicite puisque le type `char` est plus petit que le type `int`, il n'y a pas de risque de perte d'information.

Ce code affiche :

```
0x61 97
```

En d'autres termes, cela veut dire que la valeur hexadécimale 0x61 (97 en décimal) sera interprétée comme étant le caractère 'a' pour le type `char`. Cette correspondance entre la valeur en mémoire et le caractère qui est affiché s'appelle l'encodage des caractères.

L'opération inverse est également possible : on peut initialiser une variable de type `char` avec une valeur entière et afficher le caractère correspondant à cette valeur. Par exemple :

main.cpp

```cpp
#include <iostream>

int main() {
    char const c { 0x61 };
    std::cout << c << std::endl;
}
```

affiche :

```
a
```

Naturellement, en utilisant la même valeur numérique, le caractère affiché est le même.

L'encodage des caractères est une simple convention, dans l'absolu rien n'empêche pour chaque ordinateur et système d'exploitation d'utiliser son propre encodage. Mais cela voudrait dire qu'une chaîne affichée sur un ordinateur ne sera pas la même sur un autre ordinateur. Cela serait compliqué de créer des programmes dans cette situation.

Heureusement, la situation n'est pas aussi catastrophique. Pour permettre d'afficher une chaîne correctement sur différents systèmes, différents systèmes d'encodage ont été normalisés. Le plus connu est l'ASCII (American Standard Code for Information Interchange), qui est l'encodage par défaut en C++ et donc celui que l'on a utilisé depuis le début sans le savoir.

Le tableau suivant permet de retrouver la correspondance entre valeur numérique et caractère affiché. Pour le lire, il faut regarder les en-têtes de ligne et de colonne pour trouver la valeur. Par exemple, le caractère 'a' se trouve dans la ligne “0x6.” et la colonne “0x.1”, ce qui correspond donc à la valeur “0x61”, ce que l'on a vu dans les codes précédents.

	0x.0	0x.1	0x.2	0x.3	0x.4	0x.5	0x.6	0x.7	0x.8	0x.9	0x.A	0x.B	0x.C	0x.D	0x.E	0x.F
0x2.	espace	!	”	#	$	%	&	'	(	)	*	+	,	-	.	/
0x3.	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7.	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Il est possible d'écrire directement des littérales caractère ou chaîne en utilisant les valeurs hexadécimales des caractères en utilisant la syntaxe suivante : “\x” suivi du code hexadécimal. Par exemple :

main.cpp

```
#include <iostream>
#include <string>

int main() {
    std::cout << '\x24' << '\x58' << std::endl; // littérales caractère
    std::cout << "\x64\x34" << std::endl;       // littérale chaîne
}
```

affiche :

```
$X
d4
```

Les valeurs comprises entre 0x00 et 0x1F correspondent à des caractères spéciaux, dont l'utilisation est réservée.

Les normes d'encodage sur 8 bits

	ASCII	Accent		Arabe		Chinois		
Caractères affichés (UTF-8)	a	é		ع		漢		
Représentation en mémoire	0x61	0xC3	0xA9	0xD8	0xB9	0xE6	0xBC	0xA2
OEM 850 (console Windows)	a	├	®	Ï	╣	µ	╝	ó
Windows-1252 (Qt Creator)	a	Ã	©	Ø	¹	æ	¼	¢

Les plus attentifs auront peut-être remarqué un détail sur la norme ASCII. Si on vérifie les valeurs limites du type `char`, on peut remarquer que celui-ci permet de coder 256 valeurs possibles (de -128 à +127). Or, dans la norme ASCII, seules les valeurs de 0x00 (0 ou 0b00000000) à 0x7F (127 ou 0b01111111) sont définies, les valeurs de 0x80 (0b10000000) à 0xFF (0b11111111) ne sont pas définies dans la norme. Cela revient à dire que la norme ASCII encode les caractères sur 7 bits dans un type sur 8 bits.

L'autre problème avec la norme ASCII est qu'elle ne permet pas d'encoder les accents ou les caractères différents de l'alphabet latin. Historiquement, cela s'explique du fait que l'anglais s'est imposé dans le passé comme langue par défaut de l'informatique. Cependant, avec le temps et la démocratisation de l'informatique, il a été nécessaire de pouvoir afficher d'autres langages que l'anglais, ce qui a abouti à l'apparition de nombreuses normes d'encodage.

La première solution pour ajouter ces caractères supplémentaires a donc été d'utiliser le huitième bit qui est inutilisé dans la norme ASCII.

http://fr.wikipedia.org/wiki/Page_de_code_850 EOM850, par défaut dans la console windows

http://fr.wikipedia.org/wiki/ISO_8859 ISO 8859, codage internationale, selon la version (ISO 8859-1 à ISO 8859-16 pour les autres alphabets)

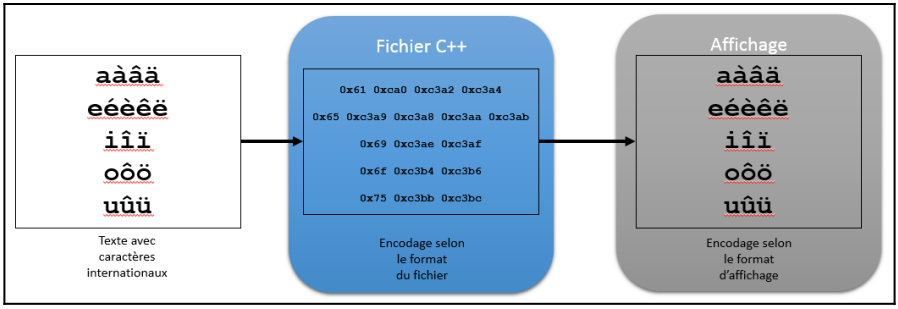
http://fr.wikipedia.org/wiki/Windows-1252 spécifique à Windows, dérivé de ISO 8859-1 (pas international)

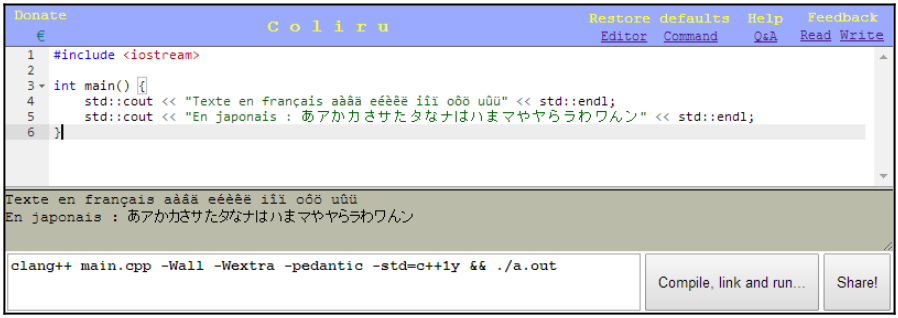
Les normes d'encodage multi-octets

Représentation des caractères internationaux sur plusieurs octets.

http://fr.wikipedia.org/wiki/Table_des_caract%C3%A8res_Unicode_%280000-0FFF%29 et http://fr.wikipedia.org/wiki/UTF-8 sur 8 bits, respecte ASCII. Si bit0 = 1, alors encodage sur plusieurs octets

UTF-16 et UTF-32 : encodage de base sur 16 ou 32





ideone.com

edit fork download copy

```
1.  #include <iostream>
2.
3.  int main() {
4.      std::cout << "Texte en français aàâä eéèêë iîï oôö uûü" << std::endl;
5.      std::cout << "En japonais : あアかカさサたタなナはハまマやヤらラわワんン" << std::endl;
6.  }
```

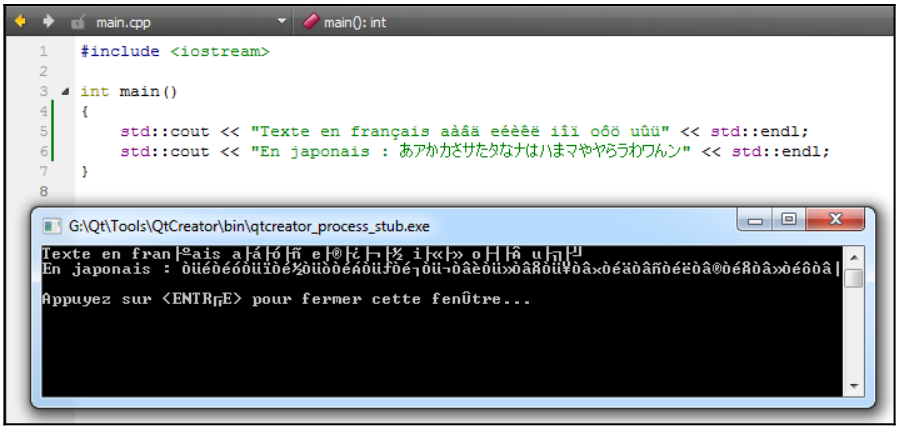
Success comments (0)

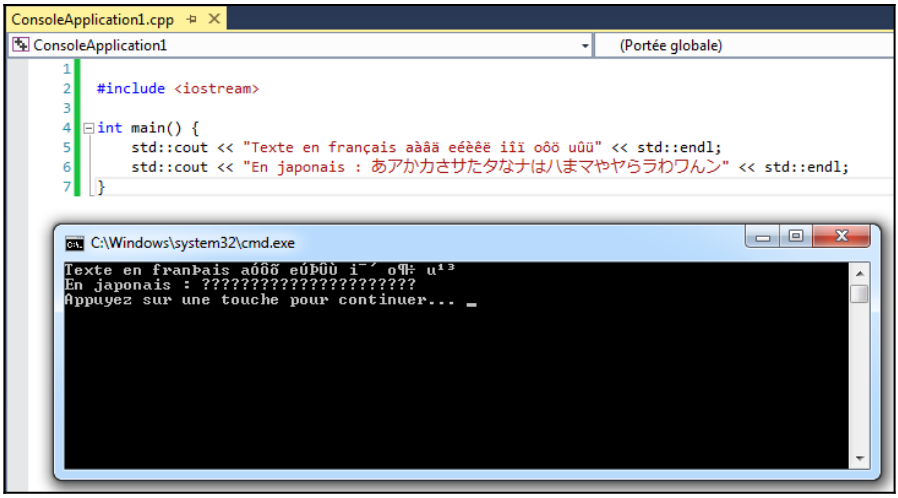
stdin copy

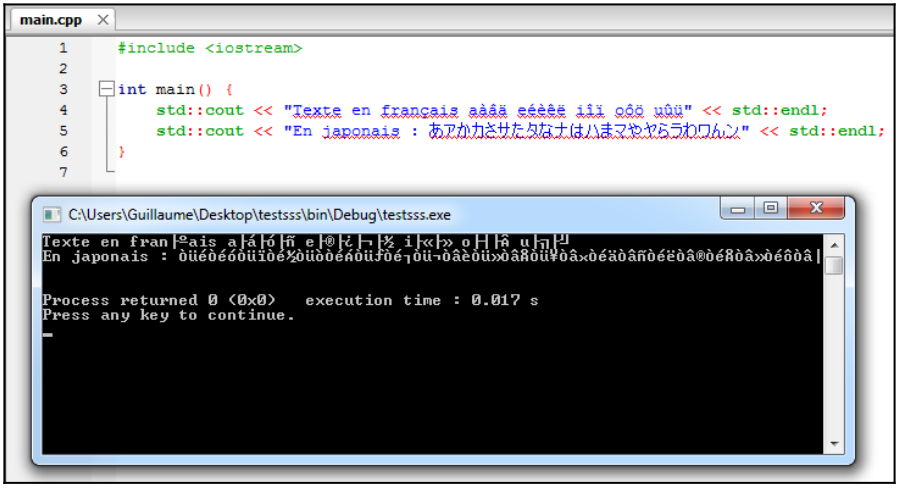
```
Standard input is empty
```

stdout copy

```
Texte en français aàâä eéèêë iîï oôö uûü
En japonais : あアかカさサたタなナはハまマやヤらラわワんン
```







wstring, u32string, u12string

caractère de 16 ou 32 bits (wchar_t), préfixe L'a' et L"bla bla"

également wcout, wcerr, wclog, wofstream, etc

- char : 8 bits, UTF-8

- wchar : implémentation spécifique
- char16_t : 16 bits, UTF-16
- char32_t : 32 bits, UTF-32

Différencier affichage et données en mémoire

Utilisation des expressions régulières

main.cpp

```cpp
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex pattern { "た" };
    std::cout << std::boolalpha << std::regex_search("あかさたなはまやらわ", pattern) << std::endl;
    std::cout << std::boolalpha << std::regex_search("あかさなはまやらわ", pattern) << std::endl;
}
```

Conversion :

main.cpp

```cpp
#include <iostream>
#include <string>
#include <regex>
#include <map>

int main()
{
    std::wstring source { L"こんにちは" }; // bonjour
    std::map<wchar_t, std::string> katakana = {
        {L'こ', "ko"},
        {L'ん', "n"},
        {L'に', "ni"},
        {L'ち', "chi"},
        {L'は', "ha"}
```

```
    };
    std::string result {};
    for (auto c: source) {
        result += katakana[c];
    }
    std::cout << result << std::endl;
}
```

Localisation (pour plus tard)

Affichage d'une chaine dans une autre langage. Par défaut, écrire en anglais dans ses codes et prévoir une fonctionnalité de traduction. Lib dispo

Éviter d'écrire : `cout « “bla bla” « i « “bla bla” « endl;` parce que dans une autre langue, l'ordre peut être différent. Par exemple : `cout « “bla bla bla bla” « i « endl`. Écrire une chaine avec placeholder : `cout « tr(“bla bla %1 bla bla, i) « endl;`.

Exercices : implémentation simple tr() avec find/replace. Implémenter tr() avec regex.

Chapitre précédent	Sommaire principal	Chapitre suivant