

Les conversions numériques en C,C++,C++/CLI

par Patrick OTTAVI MVP Visual C++ ([Site](#)) ([Blog](#))

Date de publication : 31/05/2007

Dernière mise à jour : 31/05/2007

Les conversions numériques sont des sujets qui reviennent souvent sur le forum Visual C++/MFC Il faut dire que les conversions de chaînes de caractères vers un type natif et vice-versa sont omniprésentes dans nos développements Windows. A travers cet article je vous propose de faire le point sur les techniques disponibles pour ces travaux.

- I - Définition du problème
- II - Transformation d'une CString ou du contenu d'un contrôle vers un type natif
 - A - La Bibliothèque C
 - B - Du coté des MFC
 - C - La bibliothèque standard du C++ (SL)
 - D - Le Projet BOOST
 - E - Le C++/CLI
- III - Conversion d'un type int,long,float,double vers une chaine de caractères
 - A - La Bibliothèque C
 - B - Du Coté des MFC
 - C - La bibliothèque standard du C++ (SL)
 - D - Le Projet BOOST
 - D - Le C++/CLI
- VI - Conclusion

I - Définition du problème

Comme je l'ai dit en introduction la conversion entre chaîne de caractères et type natif est centrale dans nos programmes,

Les contrôles Windows travaillent avec des chaînes de caractères tout en pouvant représenter une saisie numérique voir décimale.

La représentation interne de ces données visuelles étant le plus souvent stockée en interne dans leur type natif,

Vient alors à se poser le problème de la conversion entre le contrôle et sa variable.

La majorité des posts s'y rapportant sur nos forums se résume souvent à ces questions:

Comment faire pour convertir une chaîne en entier, double, float etc..

Comment formater une chaîne à partir d'un entier, double, float etc..

Comment récupérer un entier, double float, etc., à partir d'un contrôle, et bien sûr l'inverse qui consistera à mettre à jour le contrôle à partir d'un type natif.

Pour procéder nous disposons de plusieurs techniques issues de différentes bibliothèques :

Celles du C, les bibliothèques standards du C++, mais aussi quelques apports intéressants avec le projet BOOST.

J'ai aussi fait attention à ce que l'ensemble des codes présentés fonctionnent en **UNICODE**.

Enfin je ne pouvais ignorer le C++/CLI, je consacrerai donc une rubrique pour les deux sens de conversion.

C'est ce que nous allons découvrir maintenant.

II - Transformation d'une CString ou du contenu d'un contrôle vers un type natif

A - La Bibliothèque C

La bibliothèque C fournit un ensemble de fonctions permettant la transformation de chaîne en type de données et inversement.

Un post général y est consacré dans la faq Visual C++, aussi je ne vais pas m'attarder dessus, je préfère donner la priorité au traitement C++ du sujet :

FAQ Comment convertir une CString en int, double, long ?

B - Du côté des MFC

Les MFC comme l'api 32 permettent avec **GetDlgItemInt** la récupération du contenu d'un contrôle sous forme d'entier.

mais quid des autres types long,float,double ?

Pour ces autres types il faudrait attacher au contrôle une variable correspondante au type de donnée souhaité et appeler la méthode **UpdateData(TRUE)** pour disposer de sa valeur.

Voyons maintenant une méthode de transformation d'une chaîne dans son type natif mettant en #uvre la bibliothèque standard C++ (standard library).

C - La bibliothèque standard du C++ (SL)

Commençons par convertir une chaîne vers un type spécifié :

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

template<typename T,typename S>
bool FromString( const S & Str, T & Dest )
{
#ifdef _UNICODE
    std::wstringstream iss( Str );
#else
    std::stringstream iss( Str );
#endif
    // tenter la conversion vers Dest
    return iss >> Dest != 0;
}
```

Utilisation:

```
int nInt ;
FromString(_T("10"), nInt ); // conversion en int.

double dDouble;
```

```
FromString( _T("3.14107"), dDouble ); // conversion en double

CString str=_T("1200");
#ifdef _UNICODE
    std::wstring strstl;
#else
    std::string strstl;
#endif

strstl=_T("1200");
int n=0;
FromString(str.GetString(),n);
n=0;
FromString(strstl,n);
```

La version associée à un contrôle MFC :

La récupération du contenu d'un contrôle dans son type natif : entier, double, float, long ne cause pas plus de problèmes en modifiant un peu le code précédent on obtient :

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

template<typename T>
bool FromCtrl( const CWnd & Ctrl, T & Dest )
{
    CString Str;
    Ctrl.GetWindowText(Str);
#ifdef _UNICODE
    std::wstringstream iss( static_cast<LPCTSTR>( Str) );
#else
    std::istringstream iss( static_cast<LPCTSTR>( Str) );
#endif
    // tenter la conversion vers Dest
    return iss >> Dest != 0;
}

//Utilisation:

double d;
FromCtrl(*GetDlgItem(IDC_EDITNOM),d);
if(d==10.0)
{
    // traitement
}
// etc...
```



Ces exemples s'appuient sur la classe `istringstream` et utilisent l'opérateur surchargé `>>` pour effectuer la conversion.

Ce type de fonction se nomme fonction modèle, elle permet de rester générique par rapport au type T utilisé.

D - Le Projet BOOST

Voyons maintenant ce que propose le projet BOOST :

Il fournit une classe de conversion `lexical_cast` qui permet la conversion de chaînes représentant des nombres en base 10.

Pour utiliser l'exemple qui va suivre, vous devrez télécharger le projet  **BOOST** disponible sur SourceForge La dernière version stable est la  **1.33.1**

Vous pouvez aussi utiliser la version graphique de distribution de **BOOST** avec un outil distribué par  **Boost Consulting**, voir aussi le tutoriel **Installer et utiliser Boost sous Windows avec Visual C++ 2005**

Commençons par convertir une chaîne vers un type spécifié :

L'utilisation de lexical_cast est confondante de simplicité :

```
#include <string>
#include <boost/lexical_cast.hpp>
int n,n1,n2;
CString str=_T("1200");
#ifdef _UNICODE
    std::wstring strstl;
#else
    std::string strstl;
#endif
strstl=_T("1200");
    try
    {
        n = boost::lexical_cast<int>(static_cast<LPCTSTR>(str));
        n1 = boost::lexical_cast<int>(strstl);
        n2 = boost::lexical_cast<int>(_T("1200"));
    }
    catch(boost::bad_lexical_cast &e)
    {
#ifdef _DEBUG
        TRACE("Mauvaise conversion : %s",static_cast<const char *>(e.what()));
#endif
        return false;
    }
}
```

Plus besoin de flux, la séquence peut être utilisée directement.

Néanmoins on peut toujours encapsuler le traitement dans une fonction modèle :

```
#include <string>
#include <boost/lexical_cast.hpp>
template<typename T,typename S>
bool BoostFromString(const S &rStr, T & Dest )
{
    try
    {
        Dest = boost::lexical_cast<T>(rStr);
    }
    catch(boost::bad_lexical_cast &e)
    {
#ifdef _DEBUG
        TRACE("Mauvaise conversion : %s",static_cast<const char *>(e.what()));
#endif
        return false;
    }
    return true;
}
```

Le même exemple donnera :

```
int n=0;
// MFC
CString str=_T("1200");
BoostFromString(str.GetString(),n);
// C++ - STL : string
#ifdef _UNICODE
    std::wstring strstl;
#else
    std::string strstl;
#endif
strstl=_T("1200");
BoostFromString(strstl,n);
// une chaîne
BoostFromString(_T("1200"),n);
```

La version associée à un contrôle MFC :

```
template<typename T>
bool BoostFromCtrl( const CWnd & Ctrl, T & Dest )
{
    CString Str;
    Ctrl.GetWindowText(Str);
    try
    {
        Dest = boost::lexical_cast<T>(static_cast<LPCTSTR>(Str));
    }
    catch(boost::bad_lexical_cast &e)
    {
#ifdef _DEBUG
        TRACE("Mauvaise conversion : %s",static_cast<const char *>(e.what()));
#endif
        return false;
    }
    return true;
}
```

Utilisation:

```
int n=0;
BoostFromCtrl(*GetDlgItem(IDC_EDITNUM),n);
```

A travers ces exemples nous avons découvert différentes techniques de conversion d'une chaîne vers un type numérique, le contrôle de la conversion, et ce avec les différentes bibliothèques.

Pour ma part je trouve l'utilisation de **boost::lexical_cast** plus séduisante car très simple et utilisable directement.

E - Le C++/CLI

En C++/CLI la conversion d'une chaîne managée (String) en entier ou double est relativement simple :

```
#include "stdafx.h"
using namespace System;


int main(array<System::String ^> ^args)
{
    String ^str=L"1200,20";
```

```
double d;
int n;
try
{
    Console::Write(L"Tentative conversion double ");
    d=Convert::ToDouble(str);
    Console::Write(L"d:");
    Console::WriteLine(d);

    Console::Write(L"Tentative conversion Entier ");
    n=Convert::ToInt32(str); // provoque une erreur !!!
    Console::Write(L"n:");
    Console::WriteLine(n);
}
catch(FormatException ^e)
{
    Console::Write(e->Message);
}
return 0;
}
```

On trouvera les fonctions de conversions dans l'espace de nom  **System::Convert**.

La conversion en double ou en int ne cause pas de problème particulier.

En cas de chaîne invalide une exception  **FormatException** est levée, comme c'est le cas ici avec mon exemple lors de la tentative de conversion de la chaîne en entier.

Autre exemple mettant en oeuvre les conversions entre base:

```
String ^bin = L"1111";
int decimal = Convert::ToInt32(bin,2);
String ^hexa = Convert::ToString(decimal, 16);
Console::WriteLine(L"binaire = {0}\ndécimal = {1}\nhexadécimal = {2}", bin, decimal, hexa);
```

III - Conversion d'un type int,long,float,double vers une chaîne de caractères


Passons maintenant à l'exercice inverse qui consiste à prendre une valeur d'un type natif et de la transformer en chaîne de caractères destinée à mettre à jour un contrôle.

A - La Bibliothèque C

Classiquement on pourra utiliser la fonction `sprintf`, ou les fonctions associées au type de données : comme `itoa` pour la conversion d'un `int` vers une chaîne.

Une fois la chaîne constituée on mettra à jour le contrôle.

B - Du Coté des MFC

La classe **CString** nous aide dans la conversion en proposant une méthode **Format** fonctionnant de la même manière que la fonction  `vsprintf`, `sprintf` du C. Cet exemple de la FAQ Visual montre comment l'utiliser : [FAQ Comment convertir un entier, un double, un float, etc, en chaîne de caractères ?](#)

Toujours dans l'optique de mettre à jour directement le contrôle,

Pour les autres types il faudrait attacher au contrôle une variable correspondante au type natif souhaité et appeler la méthode [FAQ UpdateData\(TRUE\)](#) pour disposer de sa valeur.

C - La bibliothèque standard du C++ (SL)

Comme précédemment, le code qui suit permet de se passer de l'association d'une variable à un contrôle et donc d'affecter directement une valeur d'un type natif au contrôle désigné.

```
#include <sstream>
#include <string>
#include <iomanip>
#include <iostream>

class FormatNum
{
public :
    FormatNum(){}
    template <typename T>
    FormatNum(const T&t)
    {
        operator <<(t);
    }
    template <typename T>
    FormatNum & operator << (const T& t)
    {
        m_ss << t;
        return *this;
    }

public :
#ifdef _UNICODE
    std::wstringstream m_ss;
#else
    std::stringstream m_ss;
#endif
};
```

```
#endif
};

template<typename T>
void ToCtrl(CWnd & Ctrl,const T & Src,FormatNum &rFormat=FormatNum())
{
    rFormat << Src;

#ifdef _UNICODE
    std::wstring s=rFormat.m_ss.str();
#else
    std::string s=rFormat.m_ss.str();
#endif
    Ctrl.SetWindowText(s.c_str());
}
```

Utilisation :

```
// met 10.345 dans le contrôle.
ToCtrl(*GetDlgItem(IDC_EDITNUM),10.345);
// met 10.35 dans le contrôle.
ToCtrl(*GetDlgItem(IDC_EDITNUM),10.345,
    FormatNum()<<std::setprecision(4));
```

Le traitement se décompose en deux parties :

La fonction modèle ToCtrl permettant la transformation du type utilisateur et l'affectation de la chaîne au contrôle passée en argument.

Un objet fonction FormatNum optionnel qui permet le formatage du flux pour contrôler la conversion, l'enchaînement des arguments, et qui fournit l'objet flux de conversion de la classe stringstream à la fonction modèle ToCtrl.

Voyons son utilisation dans les exemples qui suivent :

Dans le cas d'un double ou float si on veut maîtriser la précision du nombre envoyé on pourra utiliser la fonction setprecision définie dans l'entête standard **iomanip** pour fixer le nombre de digits souhaités.

Vous pouvez bien-sûr utiliser les autres fonctions et compléter le flux

Exemples:

Contrôler la longueur de la chaîne créée, et spécifier un caractère de remplissage.

```
// donne 0000010.35
ToCtrl(*GetDlgItem(IDC_EDITNUM),
    10.345,
    FormatNum()<<std::setprecision(4)<<std::setfill(_TCHAR('0'))<<std::setw(10));
```

Cet exemple impose une précision de 4 digits, une chaîne de 10 caractères remplie avec des '0'.

Dans le même ordre d'idée on pourra fixer la base de conversion ...

```
// donne 000000020
```

```
ToCtrl(*GetDlgItem(IDC_EDITNUM),
32,
FormatNum()<<std::setprecision(4)<<std::setfill(_TCHAR('0'))<<std::setw(10)<<setbase(16));
```

Enfin rajouter du texte devant la conversion :

```
// donne: Conversion Hexa: 0x000000020
ToCtrl(*GetDlgItem(IDC_EDITNUM),
32,
FormatNum()<<_T("Conversion Hexa: 0x")
<<std::setprecision(4)<<std::setfill(_TCHAR('0'))<<std::setw(10)<<setbase(16));
//ou
ToCtrl(*GetDlgItem(IDC_EDITNUM),
32,
FormatNum(_T("Conversion Hexa:
0x"))<<std::setprecision(4)<<std::setfill(_TCHAR('0'))<<std::setw(10)<<setbase(16));
```

Ou encore une syntaxe plus aérée:

```
FormatNum format;
format <<_T("Conversion Hexa: 0x")
<<std::setprecision(4)<<std::setfill(_TCHAR('0'))<<std::setw(10)<<setbase(16);

ToCtrl(*GetDlgItem(IDC_EDITNUM),32,format); // donne: Conversion Hexa: 0x000000020
```

Vous noterez aussi l'utilisation optionnelle de la spécification du format de conversion (FormatNum).

On pourra compléter notre traitement par une fonction de conversion vers une CString ou string de la STL.

```
template<typename T,typename S>
void ToString(S & rstr,const T & Src,FormatNum &rFormat=FormatNum())
{
rFormat << Src;
#ifdef _UNICODE
std::wstring s=rFormat.m_ss.str();
#else
std::string s=rFormat.m_ss.str();
#endif
rstr=s.c_str();
}
```

Utilisation :

```
CString str;
ToString(str,1200);
#ifdef _UNICODE
std::wstring strstl;
#else
std::string strstl;
#endif
ToString(strstl,1200);
```

D - Le Projet BOOST

BOOST fournit la bibliothèque **Boost Format** contenant une classe format permettant de réaliser les conversions, de plus elle autorise les spécifications de formats comme printf du C.

```

try
{
#ifdef _UNICODE
    boost::wformat f(_T("c'est %1% maniere de %2% les %3%"));
#else
    boost::format f("c'est %1% maniere de %2% les %3%");
#endif
    f % 1;
    f % _T("voir") % _T("choses");
    AfxMessageBox(boost::io::str(f).c_str());

    f.clear(); // vider le tampon avant reutilisation...

    f.parse(_T("conversion entiere %d"));
    f % 1200;

    AfxMessageBox(boost::io::str(f).c_str());

#ifdef _UNICODE
    GetDlgItem(IDC_EDITNUM)->SetWindowText(boost::io::str(boost::wformat(_T("Conversion Hexa: %010x"))
    % 32).c_str());
#else
    GetDlgItem(IDC_EDITNUM)->SetWindowText(boost::io::str(boost::format("Conversion Hexa: %010x") %
    32).c_str());
#endif
}
catch(boost::io::format_error &e)
{
    TRACE("Mauvaise conversion : %s",static_cast<const char *>(e.what()));
}
    
```

Boost Format peut être utilisée de plusieurs manières (exemples ci-dessus).

- En remplacement d'arguments: chaque nombre entouré de % doit être remplacé en utilisant l'opérateur % .
- En spécifiant un format: On utilisera alors les mêmes spécifications de format disponibles avec la fonction printf du C.

Je n'ai montré que deux aspects de cette bibliothèque qui possède de nombreuses possibilités.

N'hésitez pas à consulter la documentation en ligne avec les exemples associés :

 <http://www.boost.org/libs/format/index.html>

L'intérêt de **Boost Format** dans un projet **MFC** peut sembler plus limité puisque nous disposons de la méthode **Format** de la classe **CString**.

Elle sera beaucoup plus utile dans des projets C++ standards, ou dans la réalisation de bibliothèques devant se passer des **MFC**.

D - Le C++/CLI

La conversion d'un type natif vers une string se fait très naturellement en C++/CLI :


```
#include "stdafx.h"
using namespace System;

int main(array<System::String ^> ^args)
{
    double d=3.14107;
    int n=100;
    String ^Str;
    Console::Write(L"Conversion double en chaine :");
    Str=d.ToString();
    Console::WriteLine(Str);

    Console::Write(L"Conversion int en chaine :");
    Str=n.ToString();
    Console::WriteLine(Str);

    Console ::WriteLine((12356).ToString()) ;
    Console ::WriteLine((0xFF).ToString());
    Console ::WriteLine((true).ToString());
}
```

d.ToString() mérite quelques explications :

En C++/CLI tous les types classiques sont des alias d'objet du framework .net, un int est donc un objet qui hérite de la classe  **System::Objet**.

Celle-ci possède (entre autre) la méthode ToString() qui permet de convertir l'objet en chaîne de caractère.

Autre avantage en C++/CLI les types classiques étant des objets ils sont initialisés à zéro.

Une autre possibilité intéressante qui est démontrée dans les dernières lignes de mon exemple :

Un littéral numérique peut être considéré comme un objet s'il est entouré de parenthèses, alors la méthode ToString peut être appliquée#

Cette forme de conversion a l'avantage d'être simple mais elle ne permet de contrôler le format de la chaîne obtenue .

On utilisera alors String::Format.

```
using namespace System;
using namespace System::Globalization;

Str=String::Format(CultureInfo::CurrentCulture,
    "(C) Currency: . . . . . {0:C}\n" +
    "(D) Decimal: . . . . . {0,4:0}\n" +
    "(E) Scientific: . . . . . {1:E}\n" +
    "(F) Fixed point: . . . . . {1:F}\n" +
    "(G) General: . . . . . {0:G}\n" +
    " (default):. . . . . {0} (default = 'G')\n" +
    "(N) Number: . . . . . {0:N}\n" +
    "(P) Percent: . . . . . {1:P}\n" +
    "(R) Round-trip: . . . . . {1:R}\n" +
    "(X) Hexadecimal: . . . . . {0:X}\n",
    100, 3.14107);
```

```
Console::WriteLine(Str);

Str=String::Format( "3.14107->: {0:#,###};", 3.14107);//3.14
Console::WriteLine(Str);
Str=String::Format( "3.555->: {0:#,###};", 3.555);//3.56
Console::WriteLine(Str);
Str=String::Format( "100->: {0,5:00###}", 100 );//00100
Console::WriteLine(Str);
```

Le format se décompose de la manière suivante : {index[,alignment][:formatString]}

Vous retrouverez l'ensemble des  **descripteurs de format** dans la documentation MSDN.

VI - Conclusion

Il ne vous reste plus qu'à adopter la solution qui convient le mieux à vos besoins ou à la situation.

