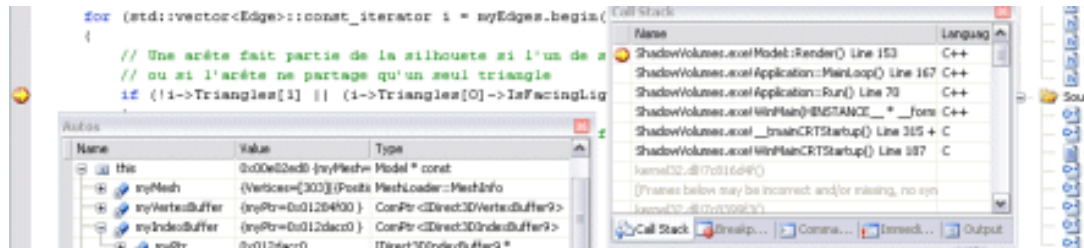


# Utiliser efficacement le débogueur de Visual Studio



par [Laurent Gomila](#)

Date de publication : 05/01/2007

Dernière mise à jour : 05/01/2007

Ce tutoriel vous dévoile toutes les ficelles du débogage sous Visual Studio, et vous aidera à corriger rapidement les comportements anormaux de vos programmes.

- 1 - Introduction
- 2 - Préparer un projet pour le débogage
  - 2.1 - Pour déboguer, il faut travailler en... debug
  - 2.2 - Les fichiers de symboles
  - 2.3 - Déboguer une DLL
  - 2.4 - Passer des arguments à la ligne de commande
  - 2.5 - Localiser l'emplacement d'une erreur fatale
- 3 - Les outils de débogage
  - 3.1 - Les espions
    - 3.1.1 - Les infobulles
    - 3.1.2 - Les fenêtres Espion
    - 3.1.3 - Les symboles de formatage
    - 3.1.4 - Visualiser le contenu d'une longue chaîne de caractères
  - 3.2 - La pile des appels
  - 3.3 - Les points d'arrêt
  - 3.4 - L'exécution pas à pas
    - 3.4.1 - Pas à pas détaillé spécifique
  - 3.5 - Modifier & Continuer
- 4 - Manipuler le débogueur depuis le code
  - 4.1 - Utiliser la sortie du débogueur
  - 4.2 - Générer des points d'arrêt depuis le code
- 5 - Les possibilités avancées
  - 5.1 - Les points d'arrêt évolués
  - 5.2 - Les points de trace
  - 5.3 - Les "debugger visualizers"
  - 5.4 - Faciliter le débogage des threads
  - 5.5 - La sortie assembleur
  - 5.6 - Les registres
- 6 - Conclusion

## 1 - Introduction

Beaucoup de développeurs, principalement débutants, utilisent très peu voire pas du tout cet indispensable outil qu'est le débogueur. Et ceux qui l'utilisent passent bien souvent à côté des fonctionnalités les plus intéressantes. Non pas parce que son utilisation est difficile, bien au contraire, mais parce qu'on ne connaît bien souvent pas l'étendue des possibilités qu'il nous offre.

Visual Studio fournit l'un des débogueurs les plus performants du marché, possédant beaucoup de fonctionnalités, et surtout très intuitif à utiliser. Voici donc un petit guide détaillé de chacune de ces fonctions, leur mode d'emploi, ainsi qu'un petit exemple montrant leur utilisation.

*Ce tutoriel a été réalisé à l'aide de Visual C++ 2005, mais l'utilisation du débogueur est similaire avec les versions antérieures ainsi que les autres langages.*

## 2 - Préparer un projet pour le débogage

### 2.1 - Pour déboguer, il faut travailler en... debug

Afin que le débogueur soit capable de vous assister correctement, il faut l'aider un minimum. La première chose à faire est bien entendu de compiler votre projet en mode **Debug** lorsque vous souhaitez utiliser le débogueur, sans quoi vous ne pourrez accéder à (presque) aucune de ses fonctionnalités. Cela doit paraître évident, mais il est toujours bon de le rappeler.

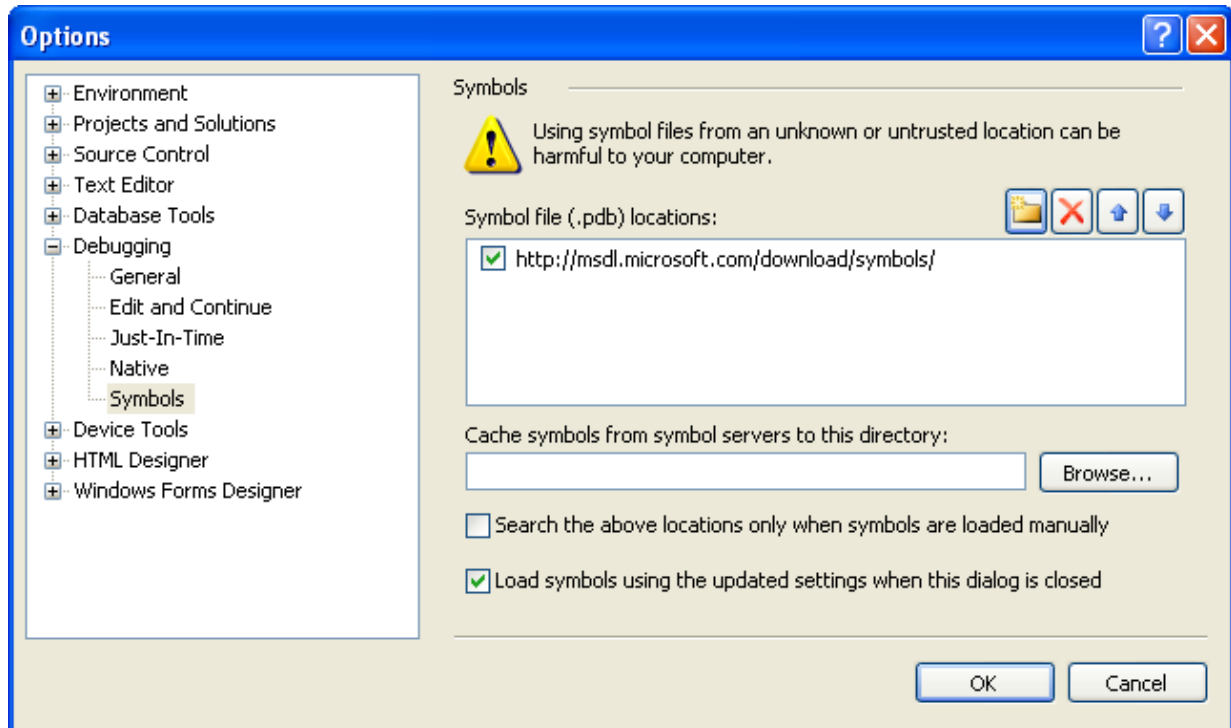
Lorsque vous utilisez une ou plusieurs bibliothèques, il faut également vous assurer que vous liez bien avec la version **Debug** de celle-ci. La plupart des bibliothèques fournissent une telle version (généralement identifiée par le préfixe "\_d"), elle vous permettra non pas de déboguer la bibliothèque en question, mais au moins d'obtenir beaucoup plus de renseignements si elle provoque un plantage dans votre application.

Si la version **Debug** n'est pas livrée, vous pourrez également la compiler vous-même si vous possédez le code source de la bibliothèque.

### 2.2 - Les fichiers de symboles

Il existe également des cas où les versions de **Debug** ne peuvent tout simplement pas être fournies, c'est le cas par exemple si vous utilisez l'API Windows. Ainsi ce qui est généralement proposé, est de télécharger ce que l'on appelle les *symbol files* (fichiers de symboles), qui ne sont rien d'autre que des fichiers contenant les informations de débogage d'un binaire ; pour Visual C++ il s'agit des fichiers .PDB.

Leur utilisation n'est cependant pas automatique, il faut indiquer au compilateur comment les trouver. Sous Visual C++ 2005, allez dans le menu *Outils->Options...->Débogage->Symboles*, puis ajoutez dans *Emplacement du fichier de symboles (.pdb)* : l'adresse suivante : <http://msdl.microsoft.com/download/symbols>. Il s'agit du serveur de téléchargement des fichiers symboles, qui fournira donc tous les fichiers symboles dont vous aurez besoin pour déboguer une DLL Microsoft. Vous pouvez au-dessous spécifier un chemin, qui sera celui dans lequel Visual C++ va placer les fichiers symboles une fois téléchargés. Un bon emplacement peut être *C:\Windows\Symbols*, par exemple.



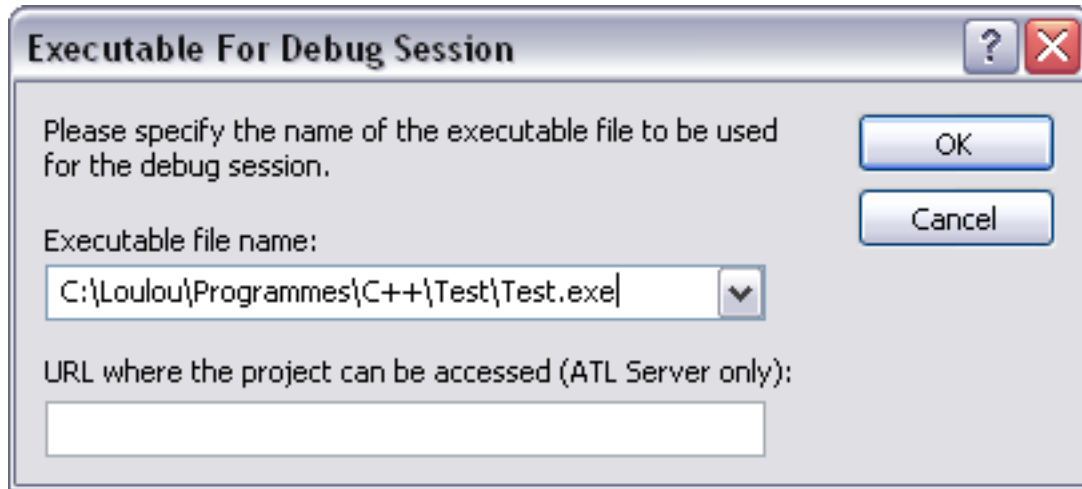
Une fois ceci fait, vous constaterez un temps de chargement plus long lorsque vous lancerez un débogage : Visual C++ va aller télécharger tous les fichiers symboles dont votre programme a besoin, et les charger automatiquement. Ce téléchargement ne se fait bien sûr que la première fois, ou lorsque le fichier est mis à jour sur le serveur.

Avec ces nouveaux symboles chargés, vous pourrez constater que votre débogueur sera maintenant beaucoup plus causant qu'auparavant.

## 2.3 - Déboguer une DLL

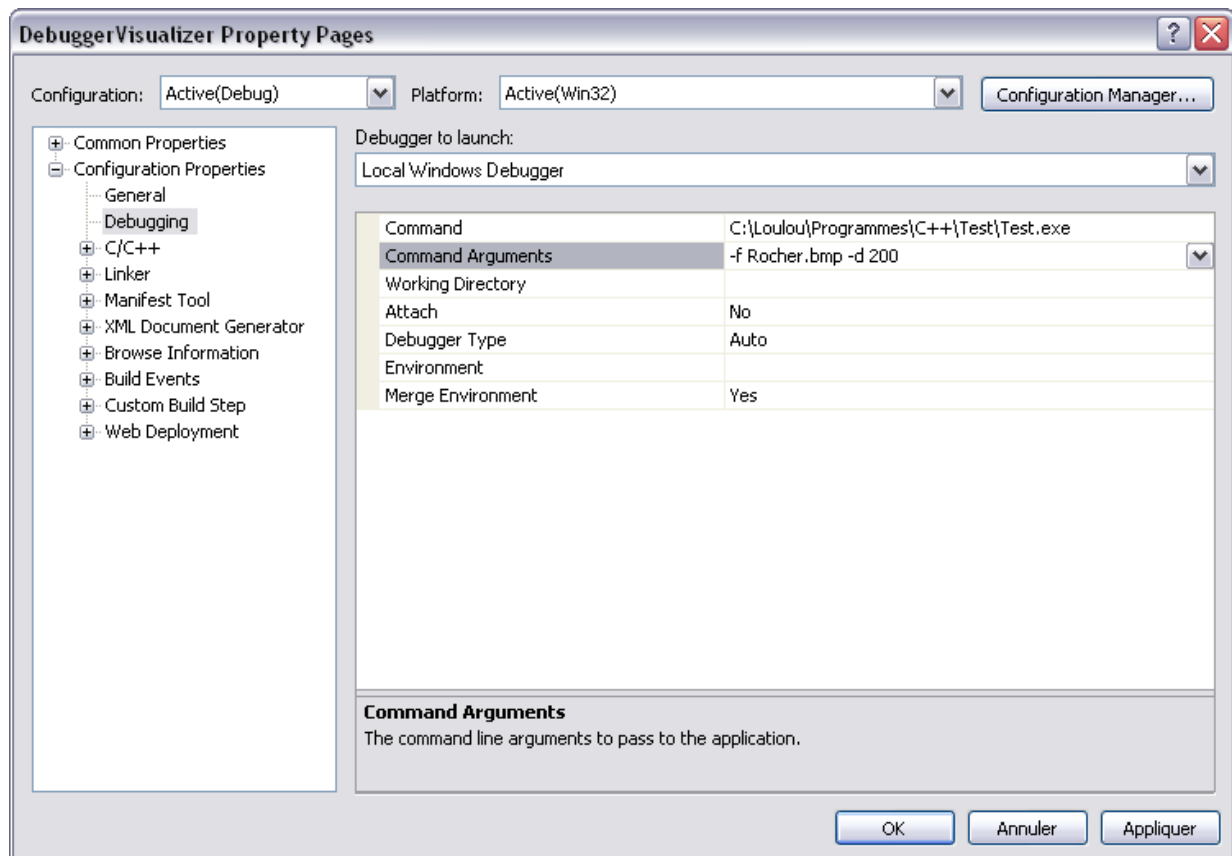
Une DLL n'étant pas exécutable seule, il est donc impossible de la déboguer directement. Ainsi pour lancer le débogage d'une DLL, Visual Studio nous permet de choisir quel exécutable lancer.

Vous pouvez spécifier le chemin de cet exécutable dans les options du projet->*Débogage*->*Commande*. Vous pouvez aussi simplement tenter de lancer le débogage directement, Visual Studio se chargera de vous demander quel exécutable vous souhaitez utiliser via une petite boîte de dialogue.



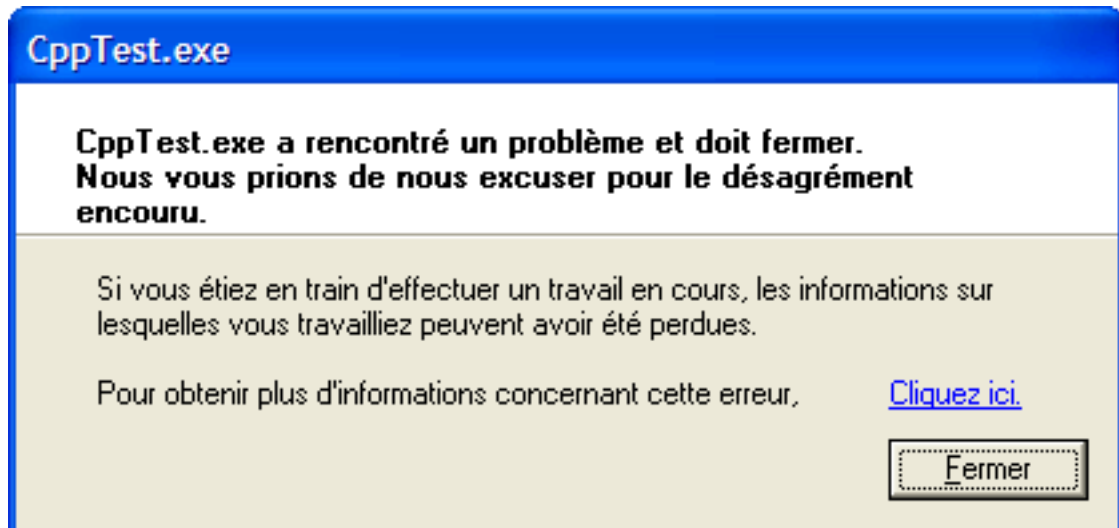
## 2.4 - Passer des arguments à la ligne de commande

Le même problème se pose pour les exécutables qui nécessitent des paramètres : comment leur passer sans utiliser la console ? De la même manière que précédemment, vous pouvez spécifier ceux-ci dans les options du projet->*Débogage*->*Arguments de la commande*. Ainsi à chaque fois que vous lancerez votre exécutable depuis Visual Studio, les arguments que vous aurez spécifiés seront ajoutés à la ligne de commande.

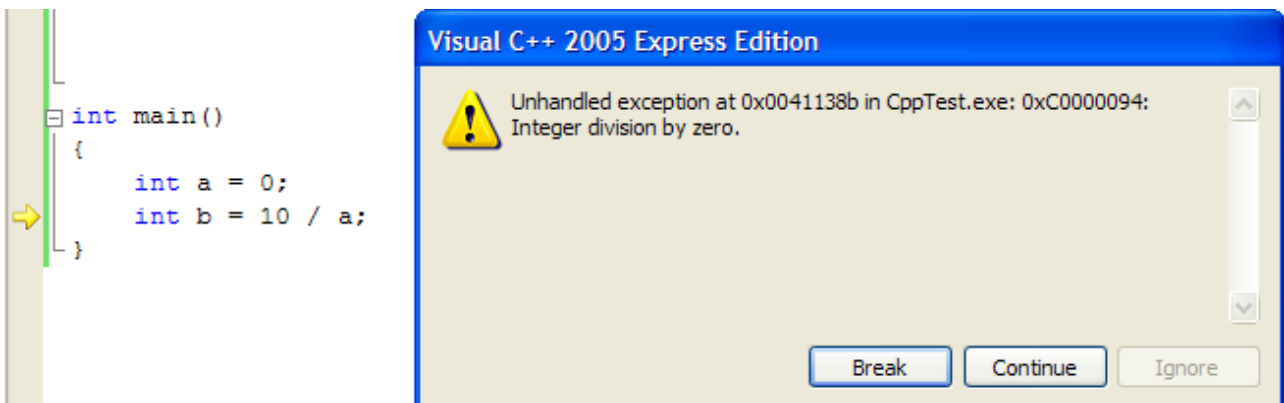


## 2.5 - Localiser l'emplacement d'une erreur fatale

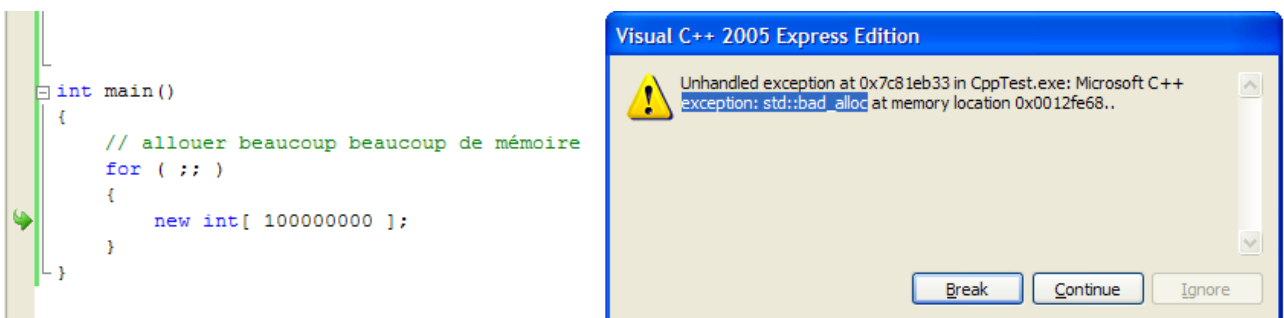
L'utilité du débogueur commence à se faire sentir en cas de plantage dans votre programme. Typiquement, vous lancez manuellement l'exécution de ce dernier, quand soudain, c'est l'accident :



Sans débogueur, vous passeriez un certain temps à disperser des messages de trace dans votre code afin de localiser le lieu du sinistre, en recompilant et re-exécutant à chaque fois. Avec un débogueur, cette approche préhistorique est un gaspillage de temps, puisque ce dernier est capable de vous emmener précisément et instantanément sur la ligne de code ayant provoqué l'erreur. Pour s'en convaincre, il suffit d'exécuter le même programme depuis le débogueur cette fois-ci, simplement en le lançant via le menu *Déboguer->Démarrer le débogage* (F5) :



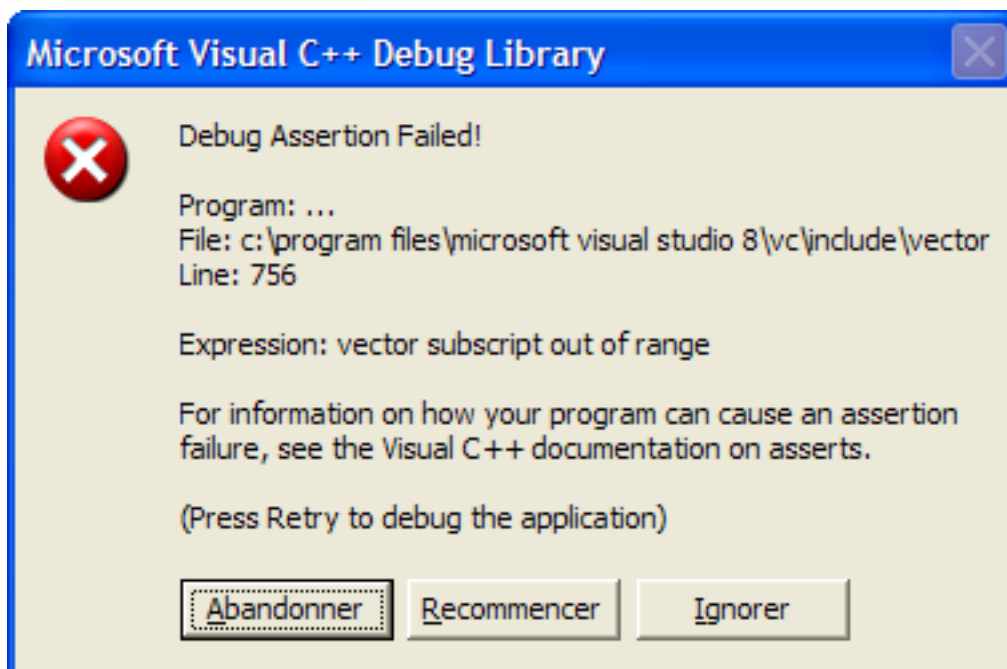
Notez dans la colonne de gauche que le curseur indique l'emplacement exact de l'erreur. Ce principe est valable pour de nombreuses erreurs, comme les exceptions non gérées.



Dans l'exemple ci-dessus, il s'agit d'une exception standard en cas d'échec d'allocation de mémoire (`std::bad_alloc`), mais cela aurait pu être n'importe quelle autre exception.

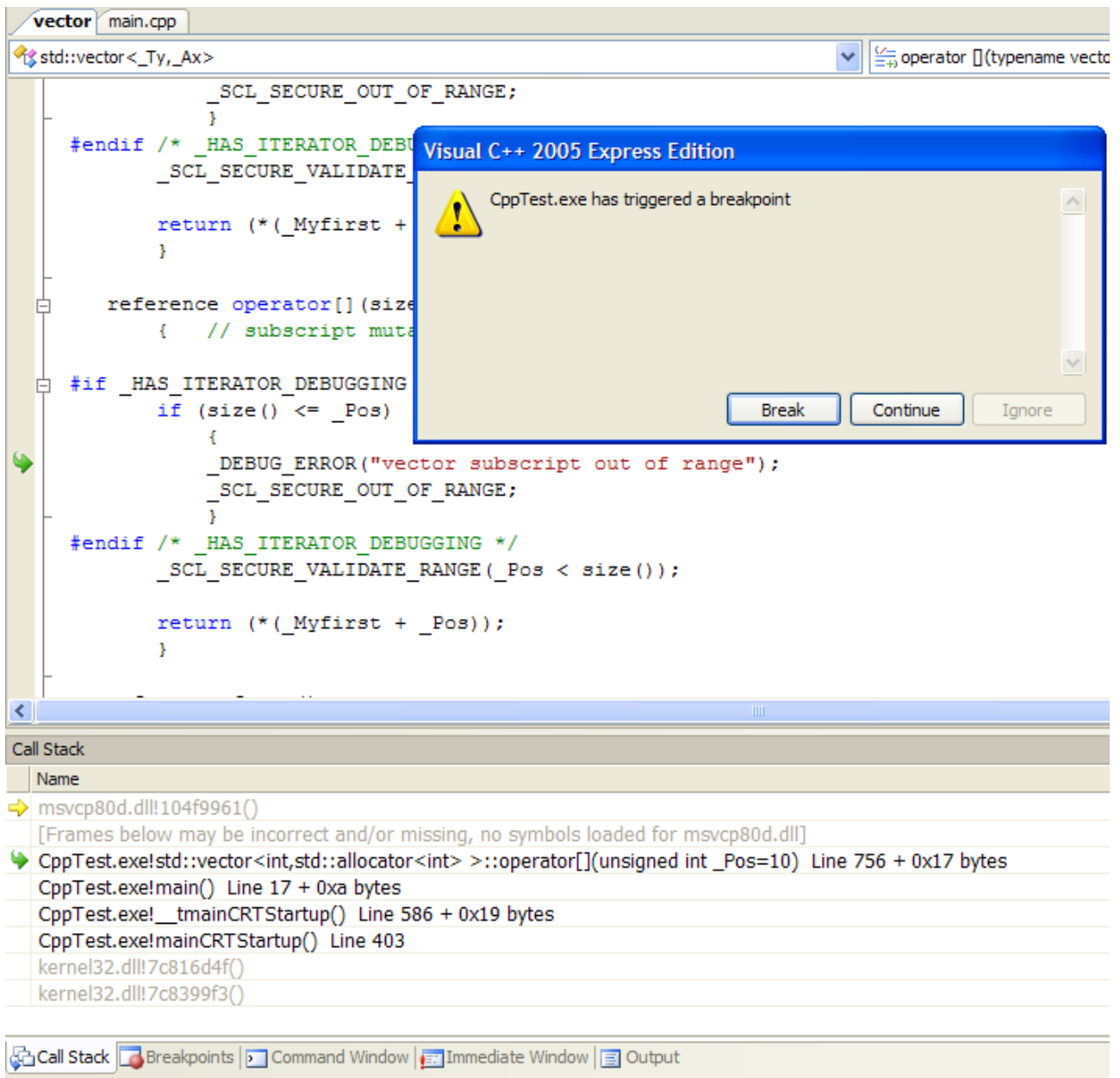
Un autre cas typique d'erreur est l'*assertion failure*, provoqué par l'échec d'un test supposé toujours vrai, mais qui précisément dans votre cas ne l'est, d'où une erreur fatale. Ces tests sont généralement réalisés au moyen de divers macros de Visual C++ ([assert](#), [ASSERT](#), [ATLASSERT...](#)) ou au moyen de macros "maison" (dont le principe est détaillé plus loin dans ce cours).

La particularité de ces erreurs est que le débogueur vous indique l'emplacement de l'assertion fautive, mais cette dernière n'est que l'endroit où une erreur a été détectée, et non là où elle se trouve. Souvent, l'erreur se situe en amont dans le code appelant. Par exemple, dans l'exemple suivant :

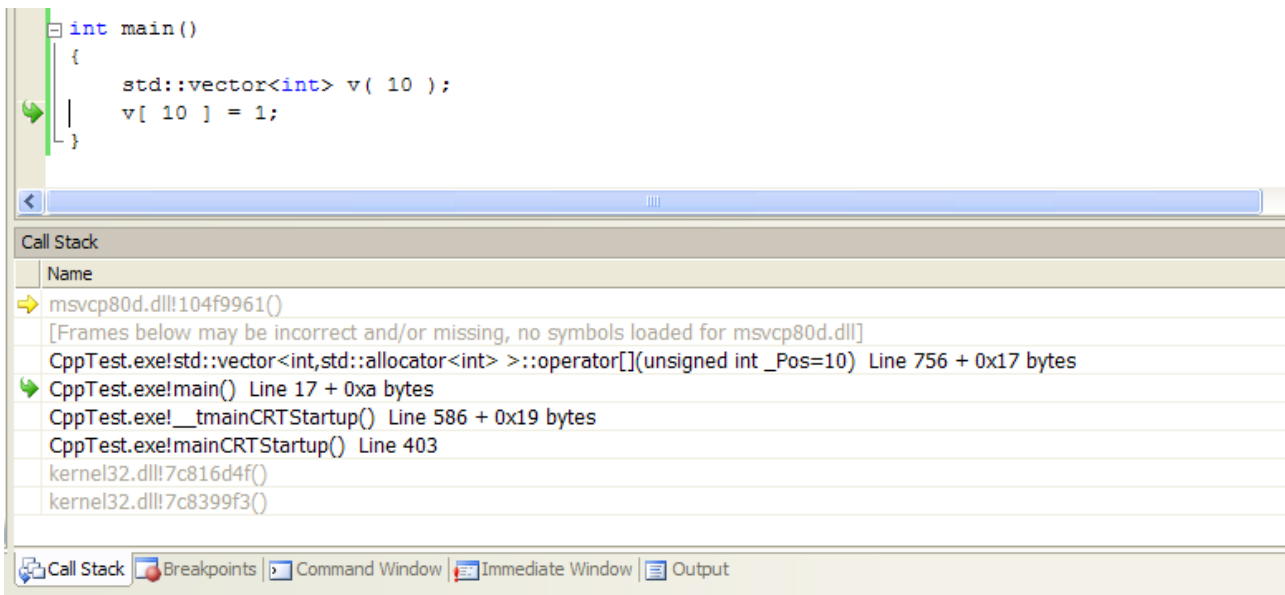


Après avoir cliqué sur Recommencer, le débogueur nous emmène dans le code de `std::vector` :





Il est peu probable que `std::vector` soit en cause, ce dernier ne fait d'ailleurs que déclencher une erreur *vector subscript out of range*. L'erreur est plutôt à rechercher du côté d'une mauvaise utilisation de `std::vector`, à priori dans le code appelant. Pour cela, il faut remonter la pile des appels (visible en bas de l'image ci-dessus -- voir chapitre 3.2), jusqu'à atteindre une fonction connue qui pourrait être responsable de l'erreur. En l'occurrence ici, il s'agit de la fonction appelante, c'est à dire `main()`. En remontant d'un niveau la pile des appels, on parvient ainsi dans le code `main()` :



Un programmeur C++ aura vite compris le problème : un tableau de 10 éléments est créé, et l'instruction `v[ 10 ] = 1;` tente d'accéder au 11ème. Or, Visual C++ 2005 introduit diverses options de sécurité qui font (entre autre) que `std::vector` vérifie que son opérateur `[]` est correctement utilisé et constate dans notre cas que ce n'est pas le cas, d'où une erreur de dépassement d'indice.

Selon ce principe, vous savez désormais comment réagir aux erreurs de type *Debug Assertion Failed!*, *Debug Error*, *Unhandled exception*, etc. Certains cas sont plus délicats, comme les détections de corruption mémoire car l'erreur peut se situer n'importe où. Nous verrons plus loin comment traiter ces cas particuliers.

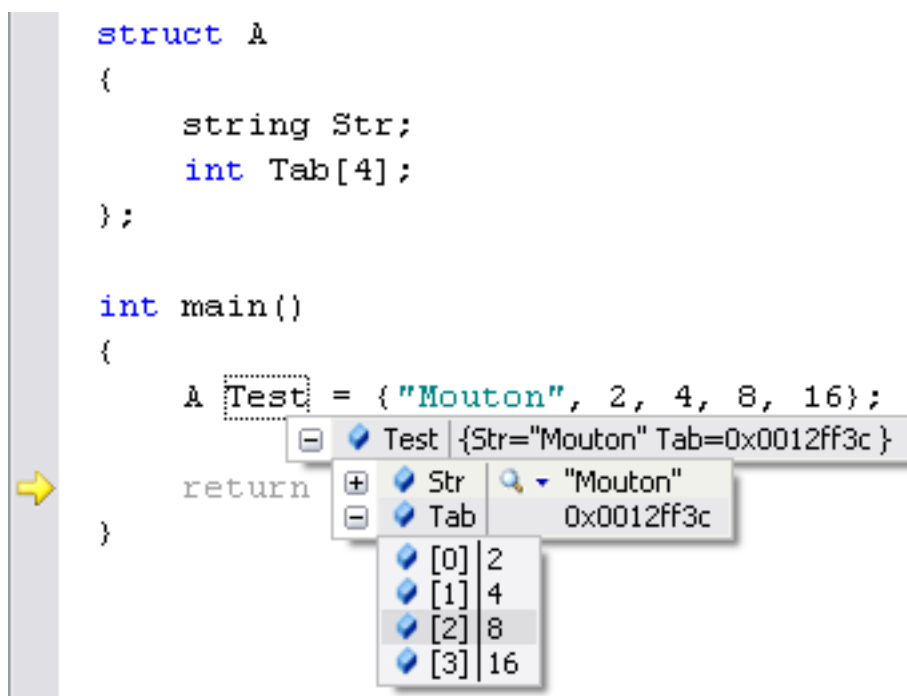
## 3 - Les outils de débogage

### 3.1 - Les espions

Une première fonctionnalité primordiale du débogueur, est la possibilité de scruter la valeur des variables, et ce à tout moment. Il existe plusieurs façons de visualiser ces valeurs, lorsque le programme est stoppé en mode débogage.

#### 3.1.1 - Les infobulles

La première manière de visualiser une variable est de simplement placer le pointeur de la souris sur celle-ci : une infobulle vous indiquera sa valeur. Pour les structures ou les tableaux, vous pouvez même dérouler et inspecter plus en profondeur les membres / éléments, et ce sur plusieurs niveaux de profondeur.



#### 3.1.2 - Les fenêtres Espion

Si vous souhaitez suivre la valeur d'une variable sans avoir à pointer dessus, ou si vous voulez évaluer des expressions plus complexes, alors vous pouvez utiliser les fenêtres Espion (*Watch windows*) prévues à cet effet. Si elles ne sont pas visibles par défaut, vous pouvez les afficher via le menu *Déboguer->Fenêtres->Espion*. Plusieurs vous sont proposées, le but est simplement de vous permettre de faire des regroupements ou de ne pas surcharger les fenêtres si vous inspectez beaucoup de variables. Vous avez également à disposition les fenêtres *Automatique* (*Ctrl+Alt+V, A*) et *Variables locales* (*Alt+4*), qui scrutent par défaut les variables définies dans le contexte courant.

Pour ajouter une variable à la fenêtre Espion, il suffit de faire un clic droit sur la variable en question (pendant la phase de débogage), et de cliquer sur *Ajouter un espion*. Ou encore plus simple : un glisser-déposer de la variable dans la fenêtre Espion. Vous pouvez également éditer les lignes de cette dernière, et entrer directement le nom de

la variable. Si le nom entré n'est pas trouvé dans le contexte courant, le débogueur vous indiquera qu'il ne peut pas évaluer le symbole. Avec cette technique il est également possible d'inspecter des expressions complexes, comme par exemple une opération mathématique, l'accès à des membres d'une structure, ou encore les deux en même temps.

```

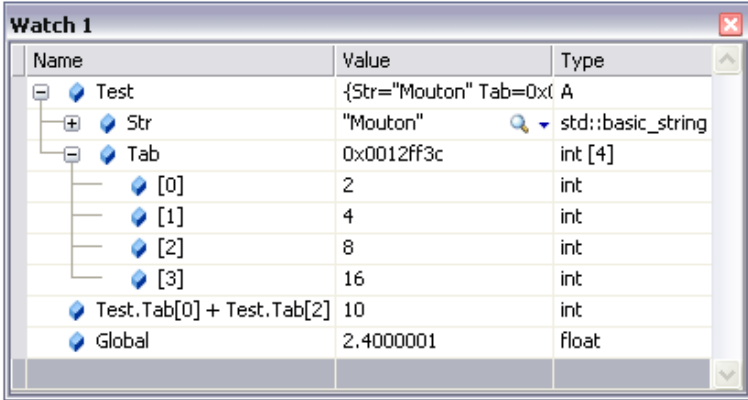
struct A
{
    string Str;
    int Tab[4];
};

float Global = 2.4f;

int main()
{
    A Test = {"Mouton",
              2, 4, 8, 16};

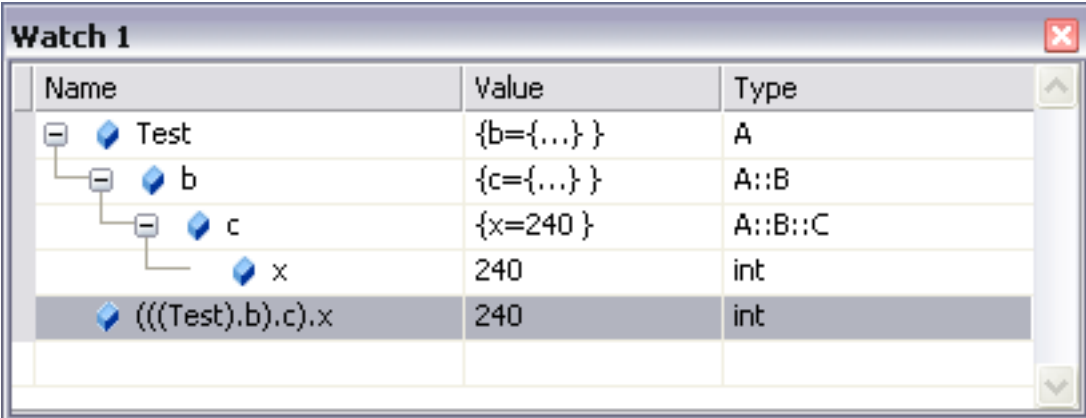
    return 0;
}

```



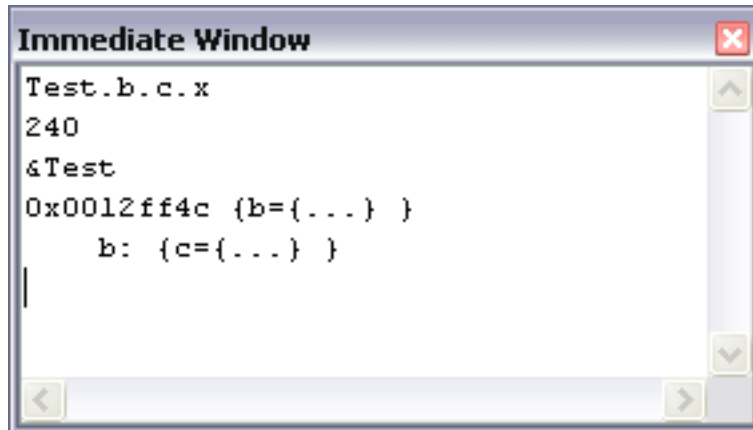
Name	Value	Type
Test	{Str="Mouton" Tab=0x00012ff3c}	A
Str	"Mouton"	std::basic_string
Tab	0x0012ff3c	int [4]
Tab [0]	2	int
Tab [1]	4	int
Tab [2]	8	int
Tab [3]	16	int
Test.Tab[0] + Test.Tab[2]	10	int
Global	2.4000001	float

Si vous voulez inspecter une variable profondément imbriquée, mais sans avoir à dérouler toute la structure qui la contient, vous pouvez également glisser la ligne de la fenêtre Espion qui vous intéresse sur une nouvelle ligne, et ainsi l'avoir directement au premier plan.



Name	Value	Type
Test	{b={...}}	A
b	{c={...}}	A::B
c	{x=240}	A::B::C
x	240	int
(((Test).b).c).x	240	int

Dernière chose : il existe une fenêtre *Immédiate* (*Ctrl+Alt+I*) qui permet également d'évaluer à la volée des expressions complexes, mais son intérêt est a priori plutôt limité par rapport à la fenêtre Espion, qui permet de faire la même chose.



### 3.1.3 - Les symboles de formatage

Si la représentation d'une variable espionnée ne correspond pas tout à fait à ce que vous voudriez, il est possible de modifier celle-ci via l'utilisation de symboles, un peu comme avec *printf* en C. La liste de tous les symboles de formatage utilisables se trouve sur la MSDN :

[Symbols for watch variables.](#)

Voici par exemple un affichage de variables par défaut :

Name	Value	Type
ERR	2	unsigned long
Message	273	unsigned int
Color	4285558784	unsigned long
Infinity	1.0000000e+020	float

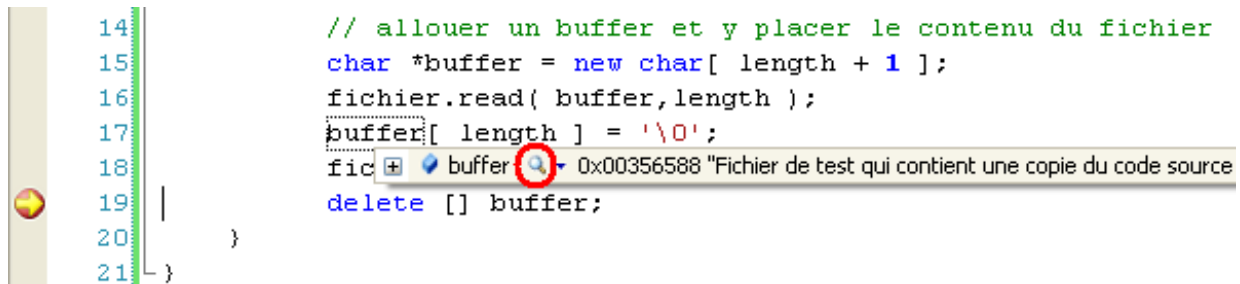
Et les mêmes avec les symboles de formatage appropriés :

Name	Value	Type
ERR,hr	0x00000002 Le fichier spécifié est introuvable	unsigned long
Message,wm	WM_COMMAND	unsigned int
Color,x	0xff707000	unsigned long
Infinity,g	1e+020	float

Petite astuce au passage : la variable *ERR* est définie automatiquement par le débogueur, et représente le code d'erreur courant ; elle a donc la valeur que vous renverrait la fonction *GetLastError()*.

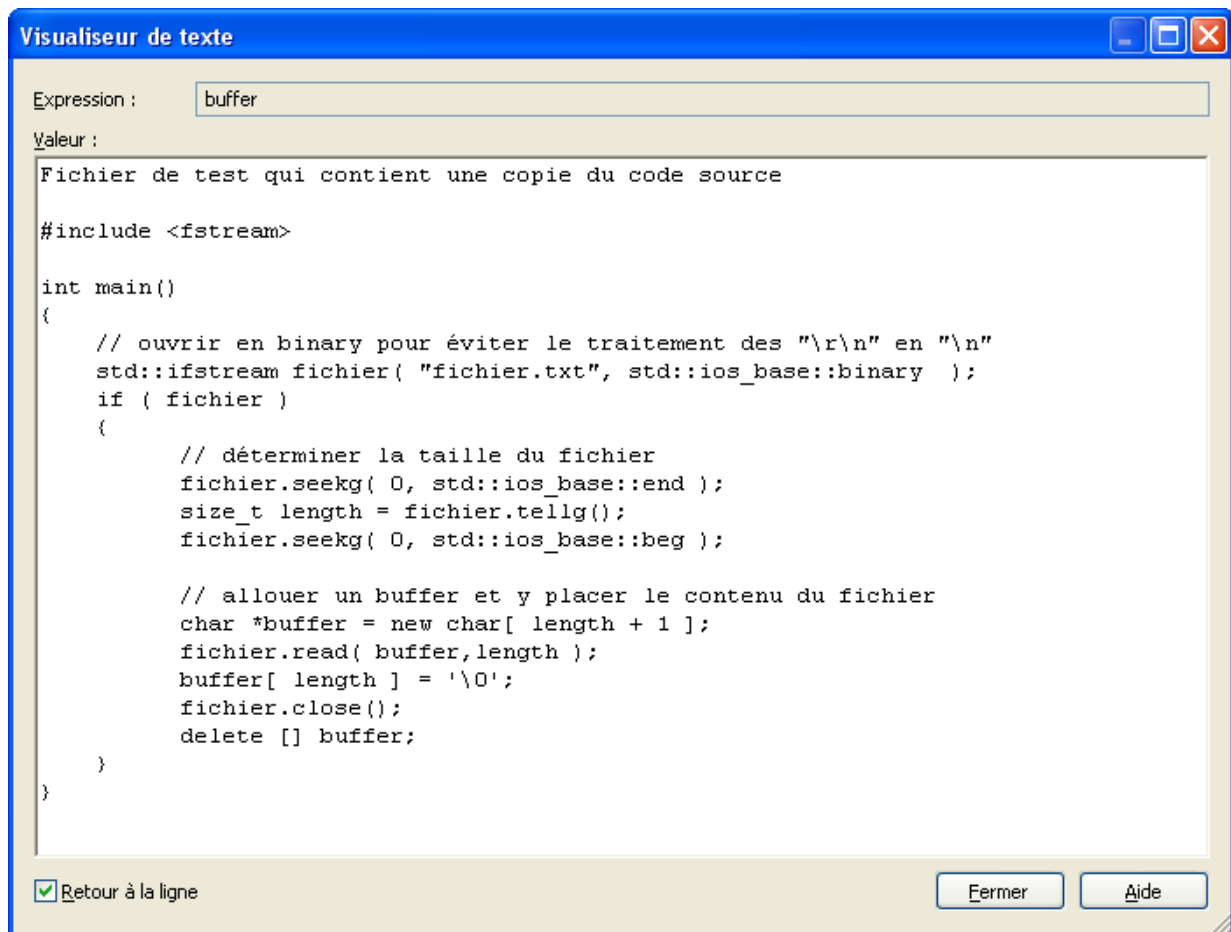
### 3.1.4 - Visualiser le contenu d'une longue chaîne de caractères

Les infobulles sont très pratiques, mais il arrive de temps en temps qu'elles soient trop petites pour afficher en totalité la valeur des variables qui nous intéressent. En particulier avec les données de type texte, dont il arrive régulièrement qu'on ne puisse pas en lire l'intégralité dans l'infobulle ou ailleurs dans les fenêtres du débogueur :



Visualisation d'une longue chaîne de caractères

Dans un tel cas, il suffit de cliquer sur la petite loupe que Visual Studio affiche dans l'infobulle à côté de la variable en question pour qu'une fenêtre *Visualiseur de texte* s'ouvre et permette de consulter aisément un volume important de données :



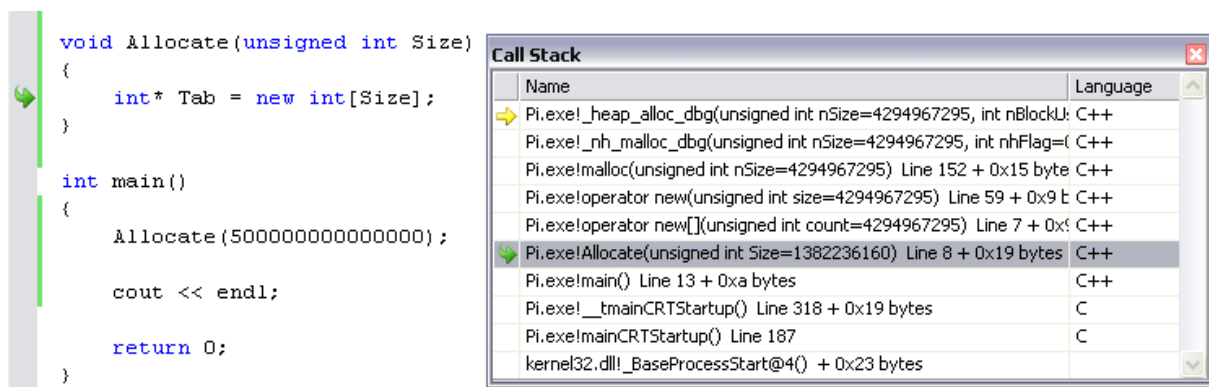
Par défaut, Visual Studio propose trois types de visualisations pour les contenus conséquents : texte, XML et

HTML (accessibles en cliquant sur la flèche qui se trouve à côté de la petite loupe).

## 3.2 - La pile des appels

La pile des appels de fonction (*call stack*) est la liste, à un instant donné, des fonctions qui ont été appelées mais qui n'ont pas terminé leur exécution. Une fonction pouvant en appeler une autre, qui va elle-même en appeler une autre, ... on obtient donc logiquement une pile, dont le premier élément sera la fonction dans laquelle l'exécution s'est arrêtée, et le dernier sera très certainement la fonction `main()`.

Si la fenêtre de pile des appels n'est pas visible, vous pouvez l'afficher via le menu *Déboguer->Fenêtres->Pile des appels (Alt+7)*.



Dans cet exemple, on voit que le programme s'est arrêté (ici : a planté) dans une fonction système nommée `_heap_alloc_dbg`. On devine qu'il s'agit d'une fonction d'allocation de mémoire dynamique, mais cela ne nous aide pas beaucoup à trouver la source de notre plantage. Grâce à la pile des appels, nous pouvons remonter les fonctions jusqu'à en trouver une qui soit issue de notre code, en l'occurrence ici la fonction `Allocate`. En double-cliquant dessus, le débogueur nous emmène jusqu'à l'endroit de cette fonction où l'exécution s'est arrêtée. Nous voyons d'une part que c'est l'appel à `new` qui provoque ce plantage, nous pouvons donc inspecter la valeur de la variable `Size` (par ailleurs déjà donnée dans la pile des appels puisqu'il s'agit d'un paramètre de fonction), et voir ce qui nous intéresse, à savoir qu'elle possède une valeur beaucoup trop élevée.

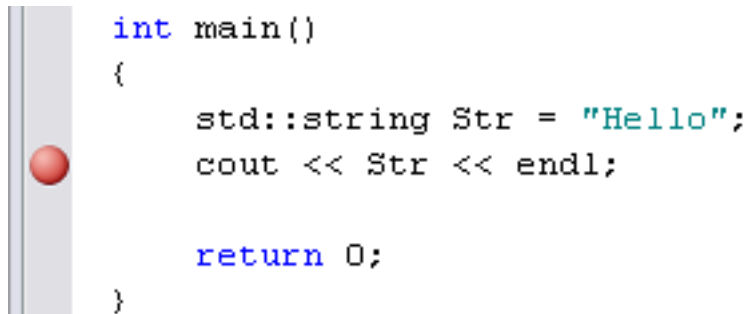
A l'aide de la pile des appels vous avez pu localiser la source de l'erreur en quelques secondes, vous évitant d'inspecter des parties de code inutilement ou encore de placer des affichages un peu partout jusqu'à tomber sur l'endroit fautif.

A noter que vous pouvez également inspecter la pile des appels de tous les threads actifs au moment où le programme s'est arrêté. La liste de ces threads est disponible dans le menu *Déboguer->Fenêtres->Threads (Ctrl+Alt+H)*. Vous obtenez ensuite la pile d'appel associée à un thread par un simple clic sur celui-ci. Très pratique si vous manipulez plusieurs threads et commencez à avoir par exemple des problèmes d'interblocages ou d'accès concurrents.

## 3.3 - Les points d'arrêt

Les points d'arrêt (*breakpoints*) sont un élément essentiel du débogueur : à chaque fois que le débogueur va en rencontrer un, il mettra en pause l'exécution à cet endroit. Pratique notamment pour inspecter la valeur de certaines variables à un moment précis, ou encore pour s'assurer que le programme passe bien par certains endroits.

Pour placer un point d'arrêt, il suffit de cliquer sur la ligne choisie, dans la colonne située à gauche du code. Vous pouvez également utiliser le raccourci F9. Vous verrez un point rouge apparaître : votre point d'arrêt est maintenant placé. Notez que Visual Studio peut déplacer vos points d'arrêt, notamment si ceux-ci se trouvent sur des portions ne correspondant pas à des instructions (un commentaire par exemple).



```
int main()
{
    std::string Str = "Hello";
    cout << Str << endl;

    return 0;
}
```

Lancez l'exécution en mode débogage : si tout se passe bien, votre programme va s'arrêter sur le premier point d'arrêt rencontré. Vous pouvez maintenant si vous le souhaitez inspecter la pile des appels à cet endroit, ou encore la valeur de variables locales. Vous pourrez ensuite continuer l'exécution jusqu'au prochain point d'arrêt, ou jusqu'à la fin s'il n'y en n'a plus.

A noter que si vous vous perdez dans les points d'arrêt, l'option du menu *Déboguer->Supprimer tous les points d'arrêt* (*Ctrl+Maj+F9*) permet de retirer tous les points d'arrêt du programme.

### 3.4 - L'exécution pas à pas

Parfois, il arrive que l'on ait besoin de voir comment se comporte un programme ligne par ligne. Pas question de poser des points d'arrêt partout, bien sûr. Visual Studio propose un mode d'exécution pas à pas, qui permet d'avancer d'une (ou plus) instruction à la fois, vous permettant ainsi d'observer en détail le parcours du programme, ainsi que de suivre l'évolution de quelques variables si vous le souhaitez.

L'exécution pas à pas peut être démarrée après l'arrêt du programme sur un point d'arrêt, mais également directement à partir du menu *Déboguer->Pas à pas détaillé* (*F11*). L'exécution pas à pas se poursuit également avec *F11*, ou avec *F10* (*Pas à pas principal*). *F11* va s'engouffrer dans le code, et entrer dans les appels de fonctions par exemple, alors que *F10* ne va jamais quitter la fonction courante même s'il croise un autre appel de fonction.

Vous pouvez également démarrer le pas à pas à partir d'un certain endroit du code, soit en posant un point d'arrêt, soit en utilisant la commande du clic droit *Exécuter jusqu'au curseur* (*Ctrl + F10*).

L'exécution pas à pas peut se révéler très utile lorsque par exemple une variable prend une valeur invalide, mais on ne sait pas où ni quand exactement. En combinant l'exécution pas à pas et une bonne utilisation de l'espion, on pourra déterminer très rapidement l'endroit à partir duquel la variable prend une valeur fantaisiste, et donc quelles en sont les causes.

#### 3.4.1 - Pas à pas détaillé spécifique

Lorsque l'on analyse l'exécution du programme pas à pas, on est souvent confronté à une situation où le traçage d'un appel à l'intérieur d'une fonction est rendu pénible par le fait que le débogueur vous emmène dans le code de constructeurs de variables temporaires, ou dans celui de fonction dont le résultat de l'appel est directement donné en paramètre. Par exemple, dans l'exemple suivant:



```
#include <string>

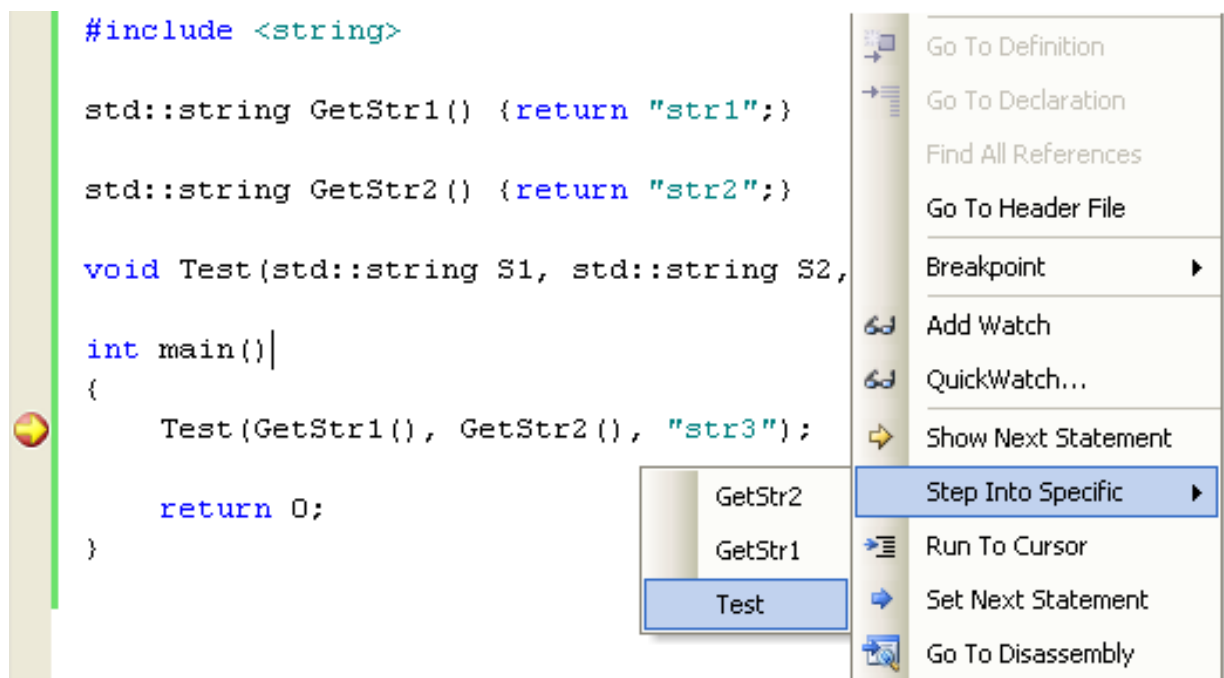
std::string GetStr1() { return "str1"; }
std::string GetStr2() { return "str2"; }

void Test( std::string S1, std::string S2, std::string S3 ) {}

int main()
{
    Test( GetStr1(), GetStr2(), "str3" );
}
```

Un point d'arrêt posé sur l'appel de *Test* suivi d'une exécution pas à pas afin de rentrer dans le code de la fonction sera "court-circuité" par un passage dans *GetStr2*, puis *GetStr1* ainsi que plusieurs fois dans le constructeur de *std::string*.

Pour directement se rendre dans le code de *Test* en ignorant toutes ces étapes, faites un clic droit et choisissez *Pas à pas détaillé spécifique->Test*, comme le montre l'image suivante:



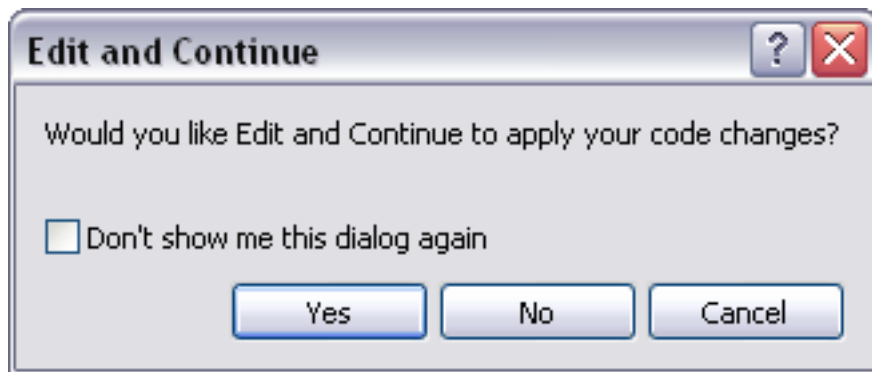
A noter que si vous vous êtes engouffré dans une fonction qui ne vous intéresse pas, vous pouvez toujours en sortir instantanément et revenir au code appelant avec la commande *Débogueur->Pas à pas sortant (Shift+F11)*.

### 3.5 - Modifier & Continuer

La fonctionnalité *Modifier & Continuer (Edit and Continue)* est assez évoluée et très pratique : elle permet de modifier le code en cours de débogage, sans avoir à tout recompiler et tout relancer. Imaginez par exemple que vous ayez trouvé un bug qu'il vous sera difficile à reproduire, ou que vous vous trouviez à un endroit de l'exécution qui suit de longues manipulations, ou encore que vous vouliez essayer plusieurs changements dans le code sans tout relancer à chaque fois.

L'option *Modifier & Continuer* est normalement activée par défaut, mais si ce n'est pas le cas vous pouvez aller cocher quelques cases dans le menu *Outils->Options...->Débogage->Modifier & Continuer*. Pour activer la

fonctionnalité il n'y a ensuite rien à faire : si vous tentez de reprendre le débogage après une modification du code, Visual C++ vous préviendra, recompilera ce qu'il faut, et poursuivra l'exécution ; vous n'y verrez que du feu. Vous pouvez également cliquer sur *Déboguer->Fenêtres->Appliquer les modifications du code (Alt+F10)*, qui peut parfois même se faire sans avoir à stopper l'exécution du programme.



*Modifier & Continuer*

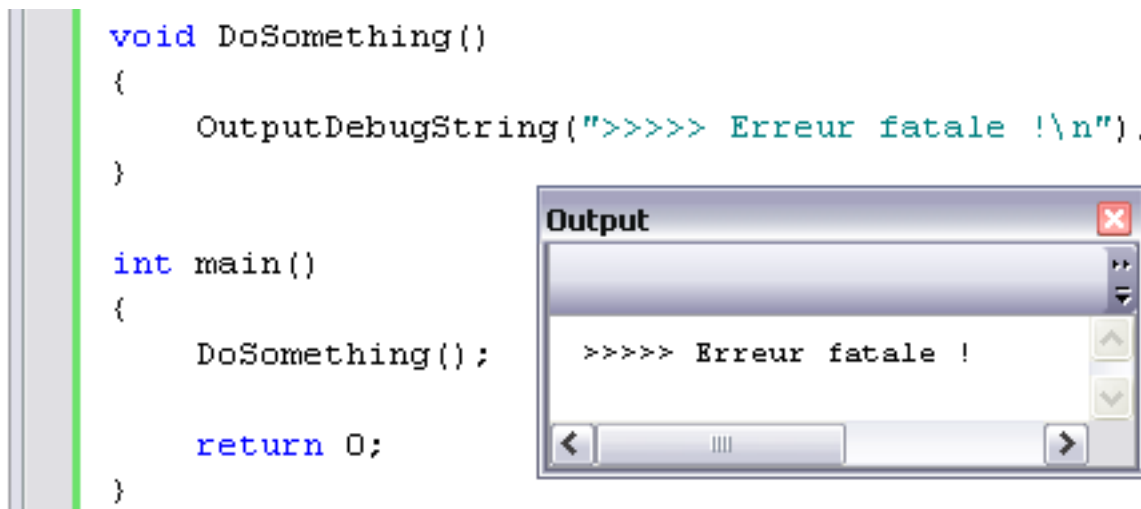
Le *Modifier & Continuer* n'est malheureusement pas toujours possible, notamment si vous effectuez trop de modifications. La liste de ses limitations est disponible sur la MSDN :

#### [Limitations of Edit and Continue](#)

## 4 - Manipuler le débogueur depuis le code

### 4.1 - Utiliser la sortie du débogueur

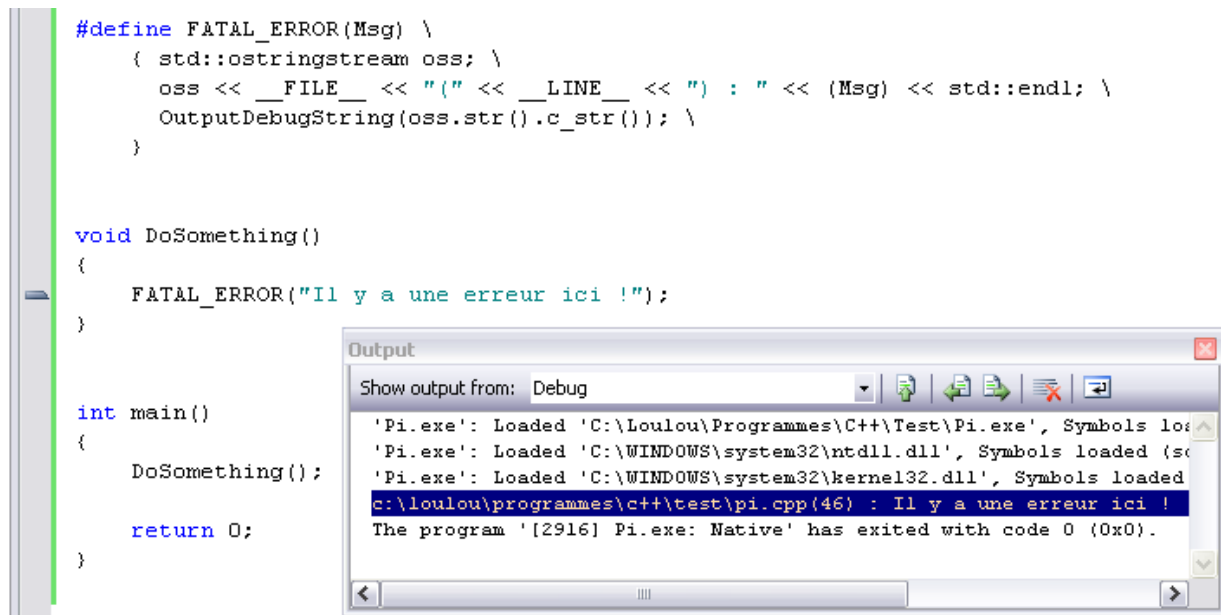
La fonction `OutputDebugString` permet de réaliser la première tâche, à savoir écrire dans la sortie du débogueur. Elle est définie dans l'en-tête `<winbase.h>`, incluez donc simplement `<windows.h>` pour l'utiliser. Son utilisation est ensuite on ne peut plus simple : passez lui une chaîne de caractère type C (terminée par un caractère nul), et elle affichera cette dernière dans la sortie du débogueur.



Ceci est très pratique pour se passer de fichiers logs et pouvoir consulter en temps réel ce qu'il se passe dans votre programme. Par exemple, la version **Debug** de DirectX utilise cette fonction pour afficher en temps réel des messages d'information, d'avertissement ou d'erreur dans la sortie du débogueur.

Nous pouvons même pousser le bouchon encore plus loin, et profiter d'un fonctionnement très simple de Visual C++ : forcer celui-ci à nous amener sur une ligne de code, comme lorsqu'il vous affiche une erreur. L'astuce est simple : pour que Visual C++ reconnaisse le fichier / numéro de ligne et vous y emmène, il suffit de respecter la syntaxe qu'il utilise, à savoir "fichier(ligne)".

Le nom de fichier et le numéro de ligne peuvent être placés automatiquement, via respectivement les macros `__FILE__` et `__LINE__`. Ces macros sont définies dans la norme, et vous assurent d'être remplacées par ce que vous attendez d'elles quelque soit l'endroit du code où vous les insérez.



Comme vous le voyez, si nous double-cliquons sur le message que nous avons affiché dans la sortie du débogueur, l'éditeur nous amène gentiment jusqu'à la ligne et le fichier inscrits.

## 4.2 - Générer des points d'arrêt depuis le code

Le débogage c'est bien chouette, mais il est parfois des moments où l'on aimerait un peu le personnaliser. Par exemple, afficher des messages persos sur la sortie du débogueur, ou encore stopper le programme à certains moments automatiquement (un peu comme des points d'arrêt dynamiques). Voyons donc comment réaliser ces deux tâches, à l'aide de morceaux de code.

Seconde tâche que nous aimerions réaliser : placer des points d'arrêt automatiquement lorsque certains actions non souhaitées se produisent. Ceci est possible grâce à une interruption dédiée des processeurs x86, à savoir l'interruption 3. Lorsqu'elle est déclenchée et que votre débogueur tourne, celui-ci l'intercepte et la gère comme un point d'arrêt.

Pour l'appeler depuis votre code, il suffit donc d'insérer un (tout petit) morceau d'assembleur :

```
__asm {int 3}
```

A noter que Visual C++ fournit aussi la fonction intrinsèque `__debugbreak()` qui équivaut à utiliser cette instruction en assembleur.

Cependant, cette instruction ne fonctionnera que si votre programme a été compilé en mode **Debug**. Si l'interruption 3 est lancée dans un programme en utilisation normale (sans débogueur), l'interruption ne sera pas interceptée et vous aurez tout simplement droit à un crash. Il convient donc de ne définir cette fonctionnalité qu'en mode **Debug**, ce que l'on peut tester via la macro `_DEBUG`, qui est définie par défaut en mode **Debug** par Visual C++ :

```

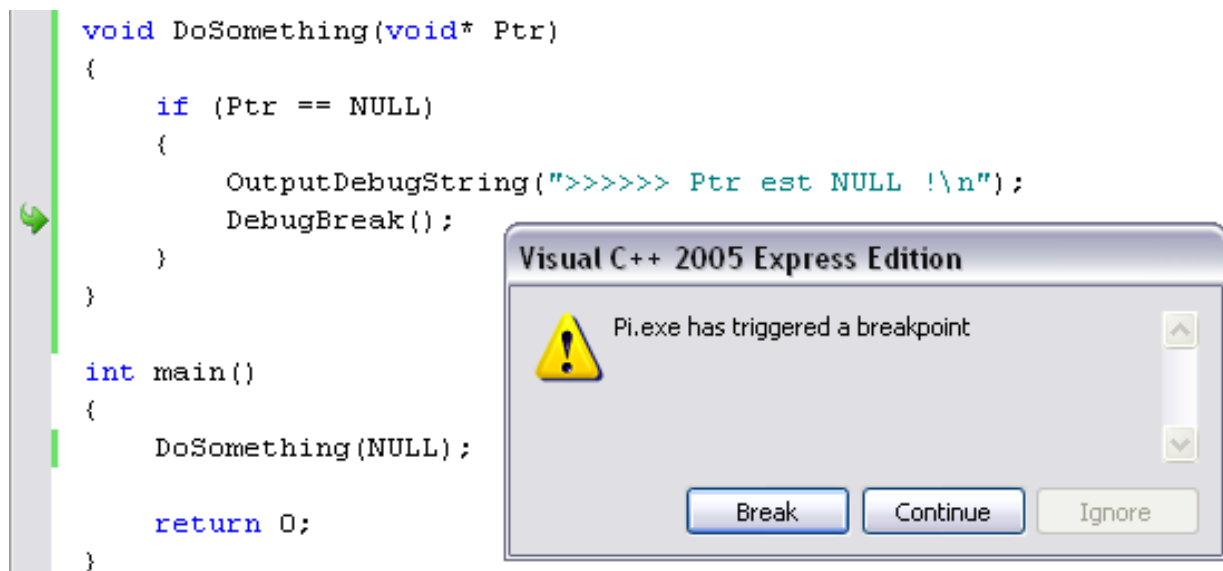
#ifdef _DEBUG
#define BREAK __asm {int 3}
#else
#define BREAK
#endif

```

C'est mieux, mais pas parfait. En effet cela ne fonctionnera que sur les CPU x86, ce qui n'est pas génialissime. Heureusement pour nous, Windows fournit une fonction qui fait exactement ce boulot, et qui fonctionnera sur tout type de processeur : il s'agit de la fonction `DebugBreak()`, définie dans l'en-tête `<windows.h>`. De plus, elle assure de ne pas planter le programme lorsque celui-ci n'est pas exécuté en mode débogage, ce qui nous affranchit des macros vues précédemment.

A noter que le point d'arrêt ne sera pas forcément toujours exactement là où vous le souhaiteriez, notamment si quelques appels de fonction sont insérés dans le processus, mais grâce à la pile des appels vous pourrez toujours remonter à l'endroit du code qui vous intéresse.

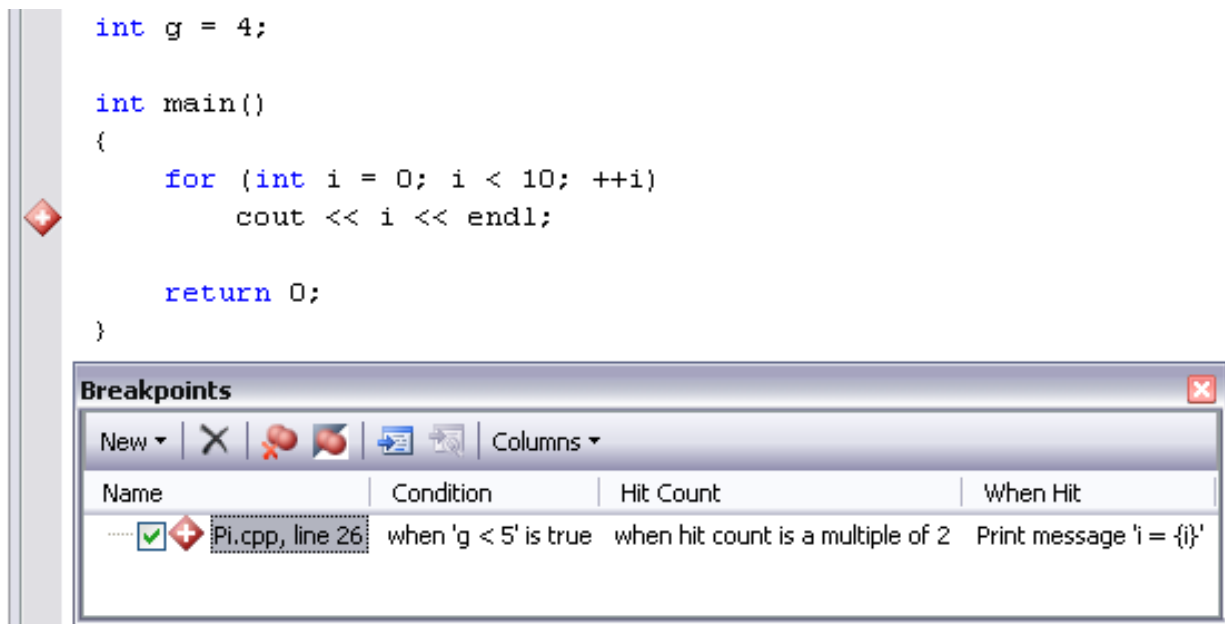
Vous vous demandez peut-être ce que l'on peut faire avec tout ça ? Par exemple, certaines bibliothèques (dont DirectX) permettent de traquer très efficacement les fuites mémoires. Première étape : votre programme est lancé en mode **Debug**, et les fuites mémoires vous sont indiquées dans la sortie du débogueur avec un identifiant. Ensuite, vous pouvez indiquer à la bibliothèque (pour DirectX cela se fait dans le panneau de configuration) de placer un point d'arrêt là où un certain identifiant a été alloué. Lors de la prochaine exécution de votre programme, vous serez amené comme par magie à une ligne du code comportant une allocation qui n'a pas sa désallocation. Très pratique non ?



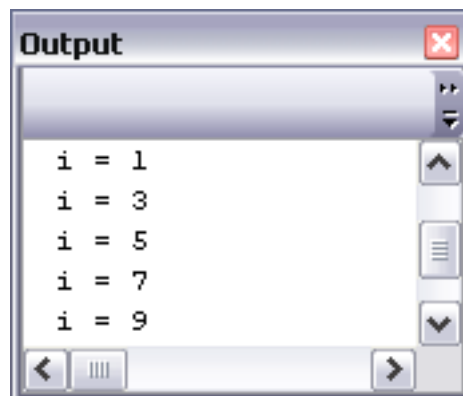
## 5 - Les possibilités avancées

### 5.1 - Les points d'arrêt évolués

Visual Studio propose également une utilisation plus poussée des points d'arrêt, accessible via la fenêtre *Points d'arrêt*. Si elle n'est pas visible, vous pouvez l'afficher via le menu *Déboguer->Fenêtres->Points d'arrêt (Alt + F9)*. Vous pouvez alors rendre vos points d'arrêt plus intelligents : poser une condition, un filtre, exécuter une action, ou encore ne l'activer que sur certains passages.



Voici la sortie générée par notre point d'arrêt après exécution du programme :

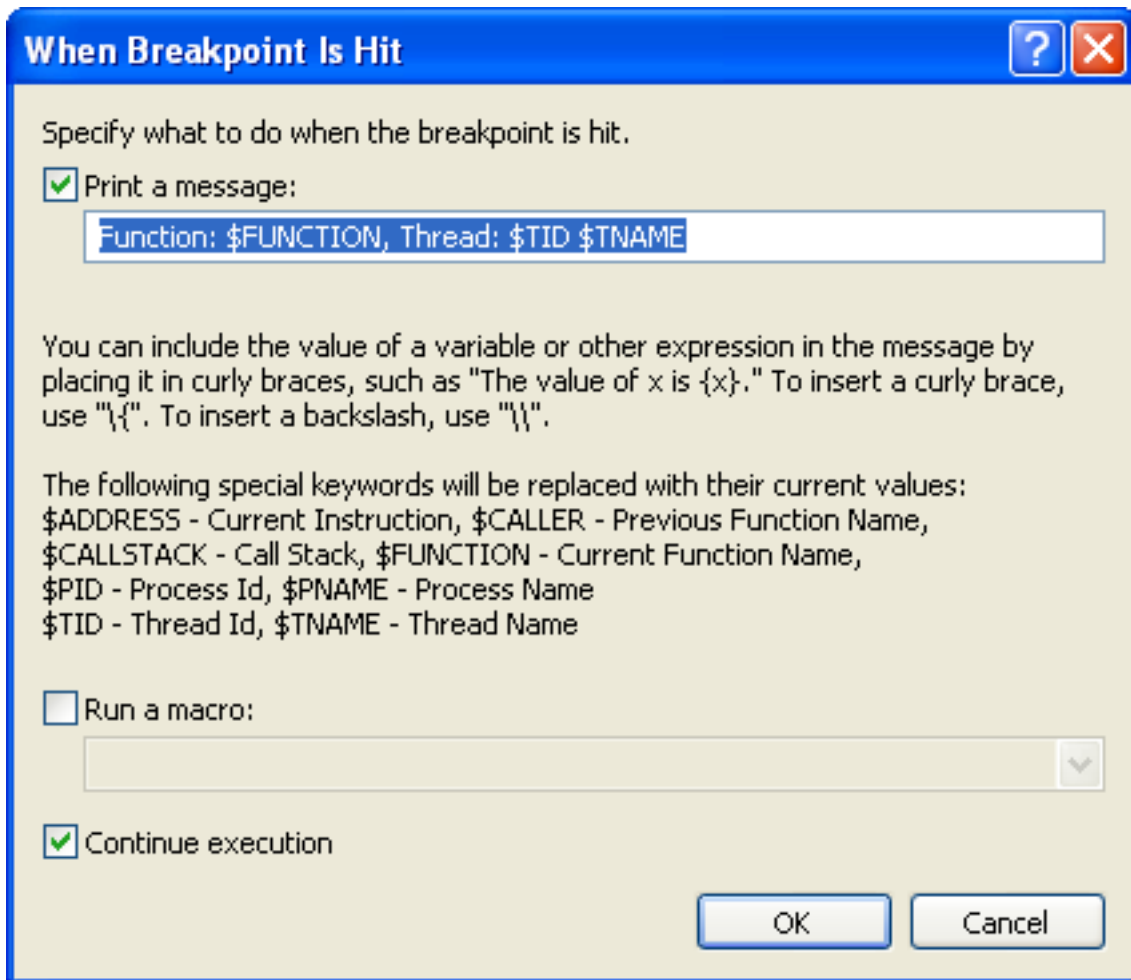


### 5.2 - Les points de trace

Les points de trace (*Tracepoint*) sont une nouveauté de Visual Studio 2005. Il s'agit en fait d'un cas particulier des points d'arrêt évolués présentés précédemment. Leur particularité par rapport aux points d'arrêt est de ne pas provoquer d'arrêt justement, mais de simplement effectuer une action. L'action de base qui leur vaut leur nom est d'afficher un message de trace. Mais les mêmes actions que les points d'arrêts évolués sont possibles, en

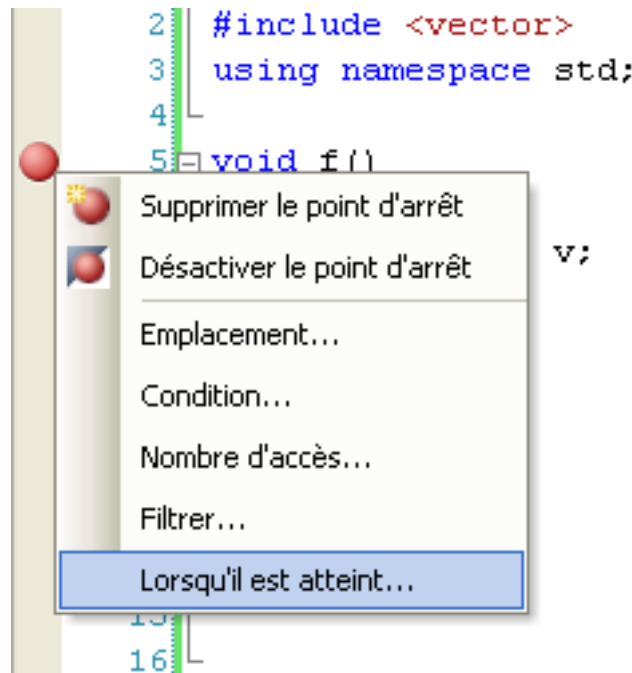
particulier le fait d'arrêter le programme.

Pour créer un point de trace, faites un clic droit sur la ligne où vous voulez le placer et choisissez *Point d'arrêt->Insérer un point de trace*. Une fenêtre s'ouvre alors :



*Création d'un point de trace*

Un autre moyen d'obtenir cette fenêtre est de créer un point de trace à partir d'un point d'arrêt. Pour cela, faites un clic droit sur le point d'arrêt concerné, et sélectionnez *Lorsqu'il est atteint...* :



Transformation un point d'arrêt en point de trace

A chaque passage au niveau du point de trace, le message renseigné sera affiché dans la fenêtre de sortie du débogueur (voir la partie *Utiliser la sortie du débogueur*) :

```
Function: f(void), Thread: 0x388 __tmainCRTStartup
```

### 5.3 - Les "debugger visualizers"

Pour certaines variables "compliquées", la représentation fournie par Visual Studio dans les fenêtres Espion n'est malheureusement pas suffisante, et il vous faudra quelques efforts pour visualiser ce qui vous intéresse vraiment dans votre classe. Nous avons vu que l'utilisation de symboles de formatage pouvait aider à afficher les variables de manière plus correcte, mais ce n'est pas toujours suffisant. Heureusement, Visual Studio permet d'aller encore plus loin, et de personnaliser l'affichage de types dans le débogueur comme bon nous semble.

La manière la plus simple (mais aussi la plus limitée) de personnaliser l'affichage d'une classe, est d'éditer le fichier *autoexp.dat* situé dans *C:\Program Files\Microsoft Visual Studio 8\Common7\Packages\Debugger\*. La syntaxe pour ajouter un type est expliquée au début du fichier, et n'est pas très compliquée. Prenez par exemple cette structure :

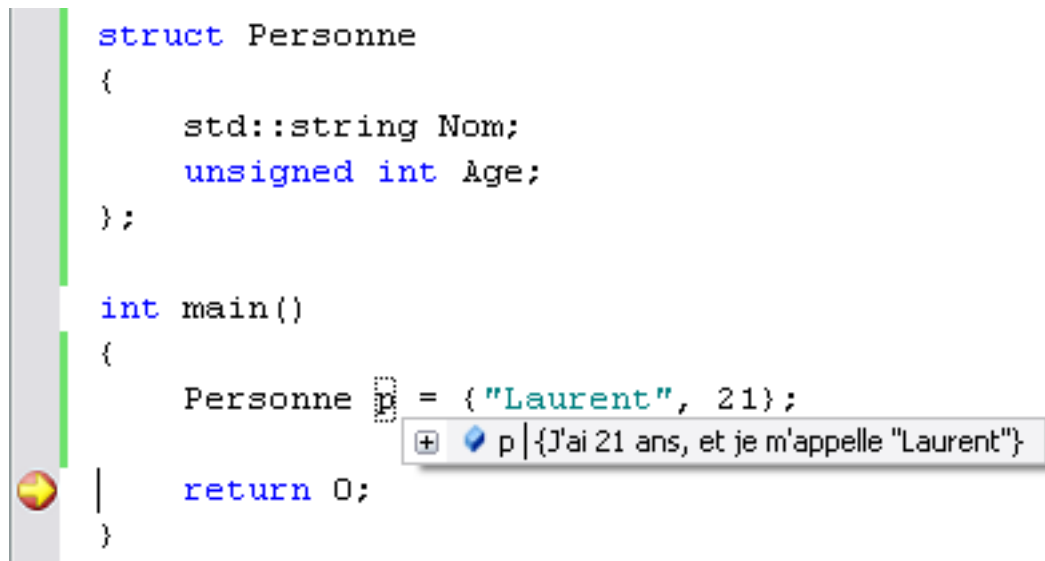
```
struct Personne
{
    std::string Nom;
    unsigned int Age;
};
```

Dans le fichier *autoexp.dat*, ajoutons cette ligne :

```
Personne = J'ai <Age> ans, et je m'appelle <Nom>
```

Plutôt simple n'est-ce pas ? Voyons maintenant ce que cela donne au niveau du débogage :





Cette façon de procéder est très pratique pour des petites personnalisations, mais pas vraiment adaptée à des traitements compliqués. Heureusement le mécanisme peut être étendu, et permet d'écrire les personnalisations dans des DLL plutôt que directement avec la syntaxe prédéfinie dans `autoexp.dat`. Pour associer une DLL à un certain type, écrivez la ligne suivante dans `autoexp.dat` :

```
Personne = $ADDIN(MonVisualiseur.dll, DebugPersonne)
```

Le premier paramètre de `$ADDIN` est le nom de la DLL (qui doit se trouver dans `$(DevEnvDir)` ou dans un chemin défini dans le PATH), le second paramètre est le nom de la fonction à appeler, tel qu'il est exporté (attention aux décorations de noms par exemple).

La manière de procéder pour écrire la DLL est expliquée sur [cette page](#) de la MSDN, et un exemple *EEAddin* est fourni (visualiseur d'expressions régulières).

Voici un exemple commenté d'une DLL qui permet d'afficher la structure suivante de manière élaborée :

```

struct Test
{
    enum FLAGS
    {
        FLAG1 = (1 << 0),
        FLAG2 = (1 << 1),
        FLAG3 = (1 << 2),
        FLAG4 = (1 << 3)
    };
};

unsigned long Flags;
unsigned char Mask;

```

```

#include <windows.h>
#include <stdio>
#include <string>
#include "Test.h"

// Voici la structure que le débogueur va passer à notre fonction, et qui permettra d'accéder
// à l'objet à afficher
struct DEBUGHELPER
{
    DWORD dwVersion;

```

```

BOOL (WINAPI *ReadDebuggeeMemory)(DEBUGHELPER *pThis,
                                  DWORD dwAddr,
                                  DWORD nWant,
                                  VOID* pWhere,
                                  DWORD *nGot);

// from here only when dwVersion >= 0x20000

DWORDLONG (WINAPI *GetRealAddress)(DEBUGHELPER *pThis);

BOOL (WINAPI *ReadDebuggeeMemoryEx)(DEBUGHELPER *pThis,
                                    DWORDLONG qwAddr,
                                    DWORD nWant,
                                    VOID* pWhere,
                                    DWORD *nGot);

int (WINAPI *GetProcessorType)(DEBUGHELPER *pThis);
};

// Voici la fonction que l'on va exporter, et qui permettra
// de renvoyer un message personnalisé au débogueur
extern "C" __declspec(dllexport) HRESULT WINAPI Debug(DWORD dwAddress,
                                                       DEBUGHELPER *pHelper,
                                                       int nBase,
                                                       BOOL bUniStrings,
                                                       char *pResult,
                                                       size_t max,
                                                       DWORD reserved)
{
    // Première chose à faire : récupérer le contenu de l'objet à afficher
    Test Obj;
    DWORD nGot = 0;
    pHelper->ReadDebuggeeMemory(pHelper, dwAddress, sizeof(Test), &Obj, &nGot);

    // On affiche son masque sous forme binaire
    std::string Result = "Masque : ";
    for (int i = 0; i < 8; ++i)
    {
        unsigned char Byte = 1 << i;
        if (Obj.Mask & (1 << i))
            Result += "1";
        else
            Result += "0";
    }

    // On affiche ses options par leur nom
    std::string Options = "";
    if (Obj.Flags & Test::FLAG1) Options += "FLAG1, ";
    if (Obj.Flags & Test::FLAG2) Options += "FLAG2, ";
    if (Obj.Flags & Test::FLAG3) Options += "FLAG3, ";
    if (Obj.Flags & Test::FLAG4) Options += "FLAG4, ";

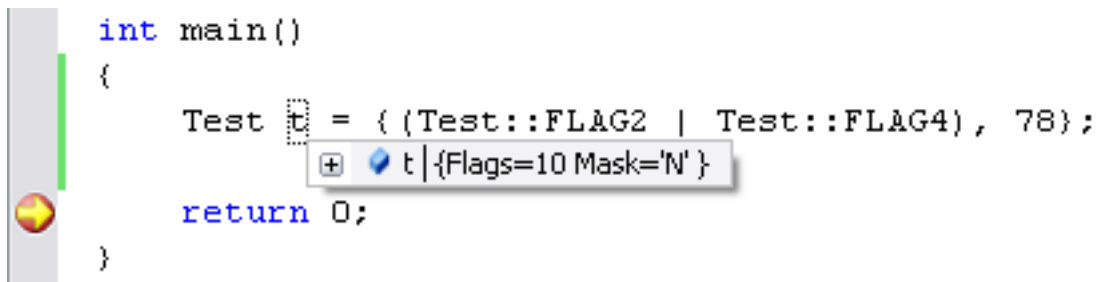
    if (Options == "")
        Result += " - Aucune option";
    else
        Result += " - Options : " + Options.substr(0, Options.size() - 2);

    // Enfin on recopie le résultat dans la variable prévue à cet effet
    strcpy(pResult, Result.c_str());

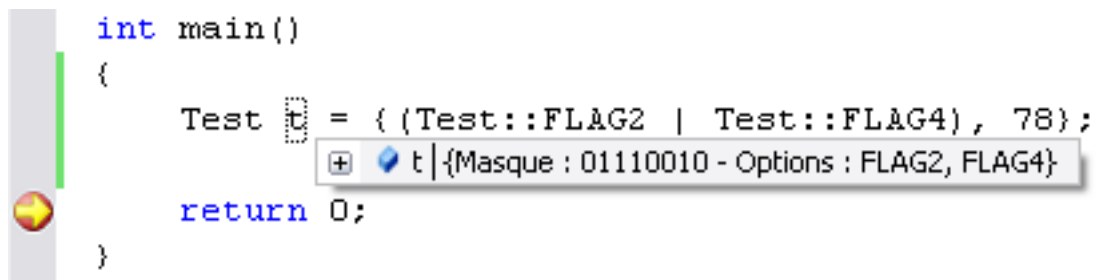
    return S_OK;
}

```

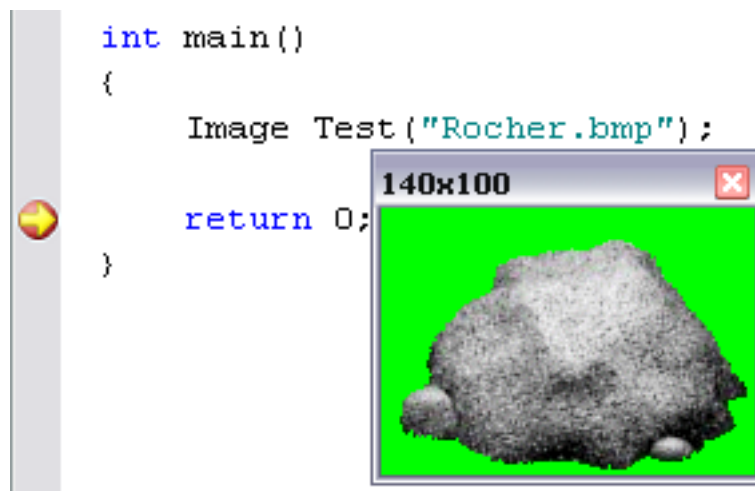
Sans notre visualiseur :



Avec notre visualiseur :



On pourrait également penser à des visualiseur plus évolués, qui affichent les informations avec une interface graphique. Pour l'exemple un visualiseur d'images a été développé, la DLL ainsi que les sources commentées et les fichiers projet VC++ 2005 sont téléchargeables ici : [sources-visu.zip \(20.1 Ko\)](#).



A noter que pour les langages managés, l'écriture d'un visualiseur est différente et beaucoup plus simple, voir par exemple ces tutoriels :

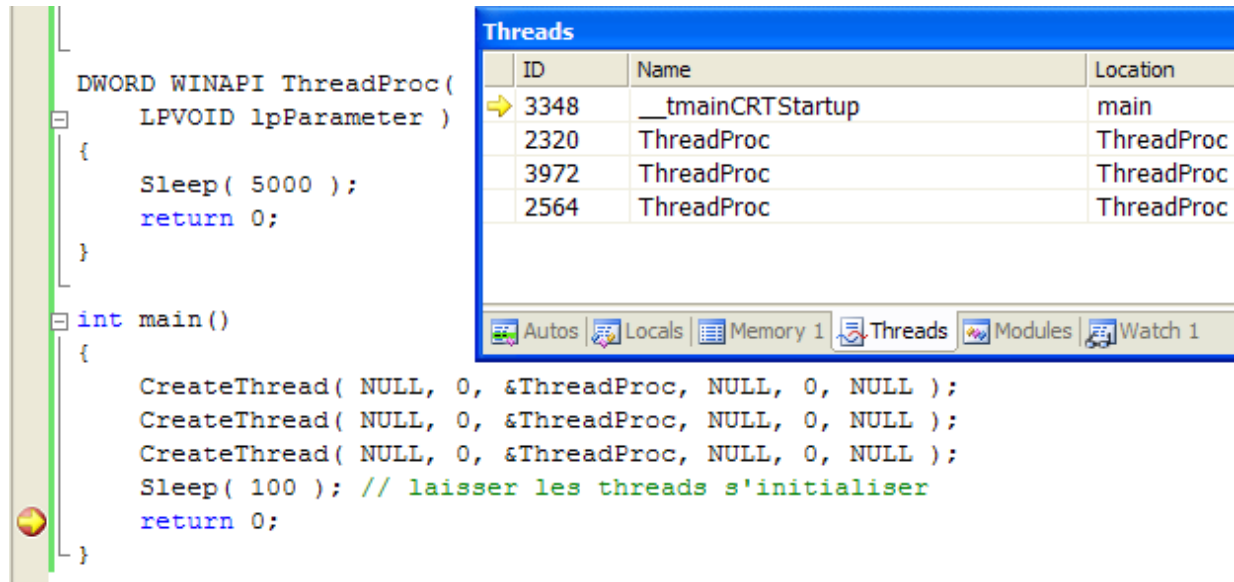
[Création d'un visualiseur de debuggage avec Visual Studio .Net 2005.](#)

[Create a debugger visualizer in 10 lines of code.](#)

## 5.4 - Faciliter le débogage des threads

La liste des threads du processus débogué est listée dans la fenêtre des *Threads* (*Ctrl+Alt+H*). Vous avez peut

être déjà remarqué que le thread principal (celui de la fonction main) possède un nom particulier (`__tmainCRTStartup`), alors que les autres threads possèdent comme nom celui de leur fonction.



#### Nom par défaut des threads

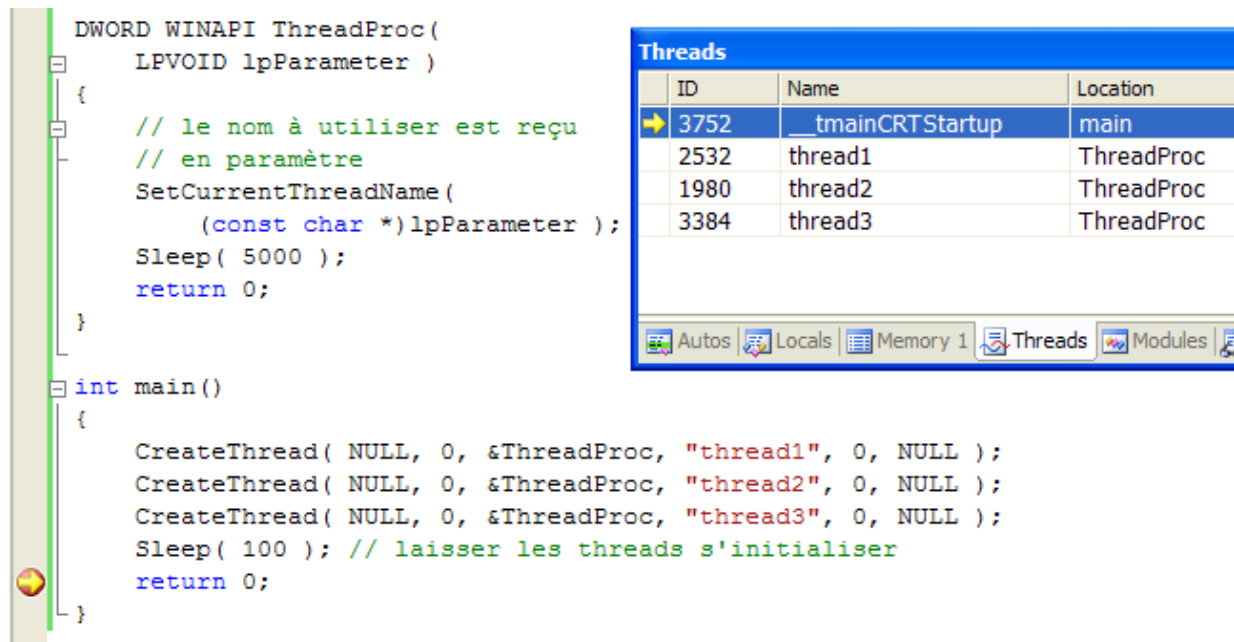
Et bien il est possible de modifier ce nom avec celui de votre choix, afin de mieux vous y retrouver. La fonction suivante modifie le nom du thread depuis lequel elle est appelée avec celui reçu en paramètre, qui doit être une chaîne littérale et surtout pas une chaîne temporaire.

```
// Donne un nom au thread appelant cette fonction.
// Ce nom apparaît dans le débogueur de VC++
void SetCurrentThreadName( const char * ThreadName )
{
#ifdef _MSC_VER // spécifique VC++
    typedef struct tagTHREADNAME_INFO
    {
        DWORD dwType; // must be 0x1000
        LPCSTR szName; // pointer to name (in user addr space)
        DWORD dwThreadID; // thread ID (-1=caller thread)
        DWORD dwFlags; // reserved for future use, must be zero
    } THREADNAME_INFO;

    THREADNAME_INFO info;
    info.dwType = 0x1000;
    info.szName = ThreadName;
    info.dwThreadID = (DWORD)( -1 ); // thread courant
    info.dwFlags = 0;

    __try
    {
        ::RaiseException(
            0x406D1388,
            0,
            sizeof info / sizeof(DWORD),
            (DWORD*)&info );
    }
    except ( EXCEPTION_CONTINUE_EXECUTION )
    {}
#else
    (void)ThreadName; // éviter les warnings
#endif
}
```

Le nom ainsi renseigné apparaît alors dans la fenêtre des threads:

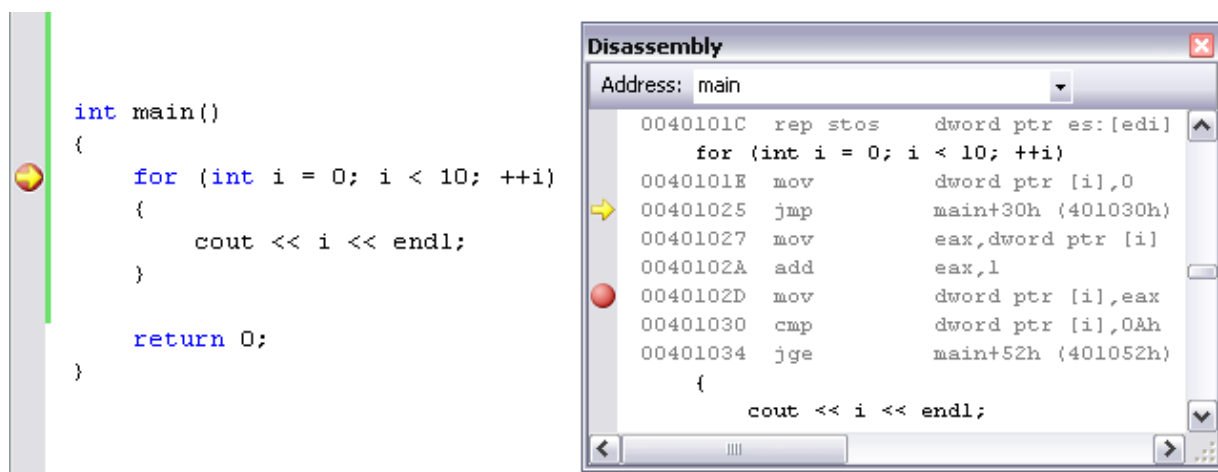


Pour plus d'informations sur ce procédé mystérieux, consultez [How to: Set a Thread Name \(Unmanaged\)](#).

## 5.5 - La sortie assembleur

Le mot peut faire peur, mais rassurez-vous, rares sont ceux qui vont jusque là. Cependant, il peut parfois être utile d'aller encore plus profondément dans le débogage, jusque dans les entrailles du code, et de déboguer directement l'assembleur généré par votre programme. Utiliser la sortie assembleur permet notamment de déboguer du code optimisé.

Pour passer à la représentation assembleur durant le débogage, il suffit d'aller dans le menu *Déboguer->Fenêtres->Code Machine (Alt+8)*. La vue assembleur permet par exemple d'avancer instruction par instruction avec l'exécution pas à pas, de poser des points d'arrêt plus précis, ou encore de visualiser avec un espion les valeurs des différents registres manipulés.

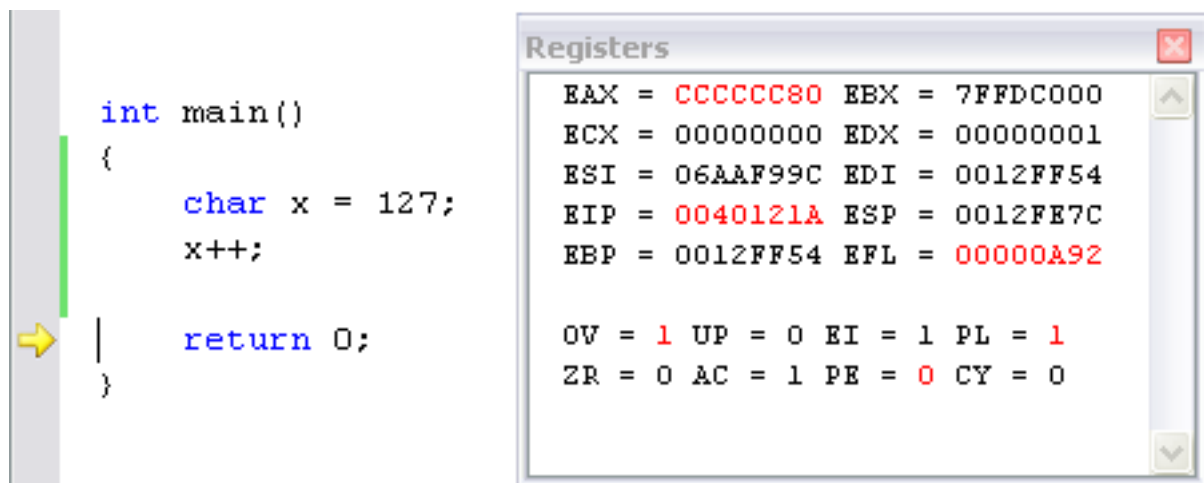


## 5.6 - Les registres

Pour ceux qui voudraient avoir une vue d'ensemble des différents registres sans passer par la fenêtre d'assembleur, il est possible d'afficher la fenêtre des registres (Menu *Déboguer->Fenêtres->Registres* (*Alt+5*)). Cette fenêtre affiche la valeur de toutes sortes de registres (CPU, SSE, MMX, floating point, flags, ...).

Par défaut seuls les registres CPU (EAX, etc.) sont affichés, un clic droit sur la fenêtre permet d'en sélectionner d'autres.

A noter que la fenêtre des registres est éditable : vous pouvez changer la valeur courante des registres (à vos risques et périls) si besoin est.



Dans l'exemple précédent, nous pouvons voir par le flag OV (*overflow*) que nous venons de provoquer un dépassement de capacité. Les autres flags nous fournissent également d'autres indications, dont voici la signification :

Flag name	Set	Clear
Overflow	<b>ov</b>	<b>nv</b>
Direction	<b>dn</b> (decrement)	<b>up</b> (increment)
Interrupt	<b>ei</b> (enabled)	<b>di</b> (disabled)
Sign	<b>ng</b> (negative)	<b>pl</b> (positive)
Zero	<b>zr</b>	<b>nz</b>
Auxiliary Carry	<b>ac</b>	<b>na</b>
Parity	<b>pe</b> (even)	<b>po</b> (odd)
Carry	<b>cy</b>	<b>nc</b>

Cet exemple ne vous a peut-être pas beaucoup parlé, mais nul doute que cela sera bien utile à ceux qui

manipulent de près ces registres.

## 6 - Conclusion

Nous venons de faire un tour d'horizon assez complet du débogueur de Visual Studio, et nous avons vu que celui-ci nous proposait bien plus qu'il nous en faut pour déboguer efficacement nos applications, la plupart du temps très facilement.

Au-delà des fonctionnalités de base telles que les points d'arrêt ou la pile des appels, il met à la disposition des programmeurs une interface simple et intuitive, permet une personnalisation bien sympathique, et offre aux plus courageux de quoi aller déboguer en profondeur leurs applications.

Si vous n'aviez jamais touché à un débogueur, j'espère vous en avoir donné l'envie et les capacités ; dans tous les cas j'espère que vous avez pu retirer quelques bons tuyaux de ce tutoriel.

Une version PDF de cet article est disponible : [Télécharger \(606 Ko\)](#)

Si vous avez des suggestions, remarques, critiques, si vous avez remarqué une erreur, ou bien si vous souhaitez des informations complémentaires, n'hésitez pas à me contacter !

Je tiens également à remercier [Aurélien](#) pour ses ajouts et correction, ainsi que [nico-pyright\(c\)](#) pour son aide tout au long de l'élaboration de ce tutoriel.