



Centre Informatique pour les **L**ettres
et les **S**ciences **H**umaines

TD 2 : Création de variables, de types et de références



| | |
|---|----|
| 1 - Préparatifs | 2 |
| Création du projet | 2 |
| Configuration de l'environnement | 2 |
| 2 - Les variables de type standard et leur initialisation | 3 |
| Une variable non initialisée | 3 |
| Erreurs fréquentes | 4 |
| Utilisation de caractères interdits | 4 |
| Fragmentation du nom | 5 |
| Oubli du point-virgule final | 5 |
| Une variable initialisée | 5 |
| Erreurs d'initialisation | 6 |
| Conversion automatique | 6 |
| Une variable de type pointeur | 7 |
| 3 - Création et utilisation d'un type énuméré | 7 |
| Définition "sur place" | 7 |
| Définition dans un fichier .h | 8 |
| Utilisation | 8 |
| 4 - Création et utilisation d'une classe | 8 |
| Définition | 9 |
| Instanciation | 9 |
| 5 - Création de références | 9 |
| Problèmes de conformité des types | 9 |
| 6 - Qu'avons nous appris ? | 10 |


Contrairement aux autres, le programme que vous allez créer au cours de ce TD ne comporte aucune interface graphique. Notre étude du langage C++ est en effet encore trop peu avancée pour que vous puissiez écrire du code dont l'exécution présente un intérêt quelconque pour un utilisateur. A ce stade, mieux vaut sans doute rester entre programmeurs et ne s'intéresser qu'au fonctionnement interne du programme...


1 - Préparatifs



Du fait qu'il est dépourvu d'interface utilisateur, le programme que vous allez écrire ne nécessite pas un projet Qt, et nous allons donc nous contenter d'un projet Visual C++ simple.


Création du projet

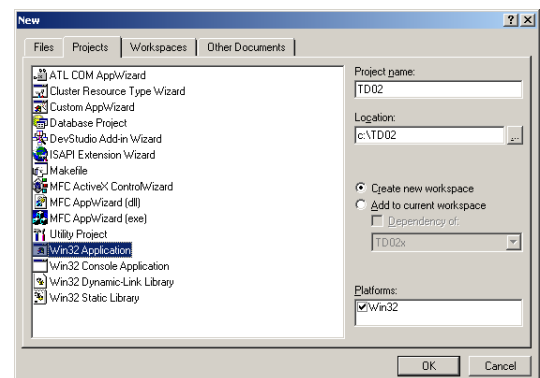
Après avoir lancé Visual C++ , déroulez le menu **File** et choisissez la commande **"New"** .

Dans le dialogue qui s'ouvre alors (cf. ci-contre), choisissez l'onglet **"Projects"** .


Dans la liste des types de projets proposés, sélectionnez l'option **"Win32 Application"** .



Fournissez un nom dans la zone d'édition **"Project name"** , et, dans la zone **"Location"**, indiquez à quel endroit vous souhaitez voir créé le dossier qui contiendra les fichiers correspondants à ce projet .


A la différence de celui rencontré lors de la création d'un projet Qt, le dialogue ouvert par le bouton  permet ici de désigner normalement le dossier où sera créé le dossier projet.




Le dialogue de création de projet de Visual C++

Refermez ce dialogue en cliquant sur le bouton **"OK"** .

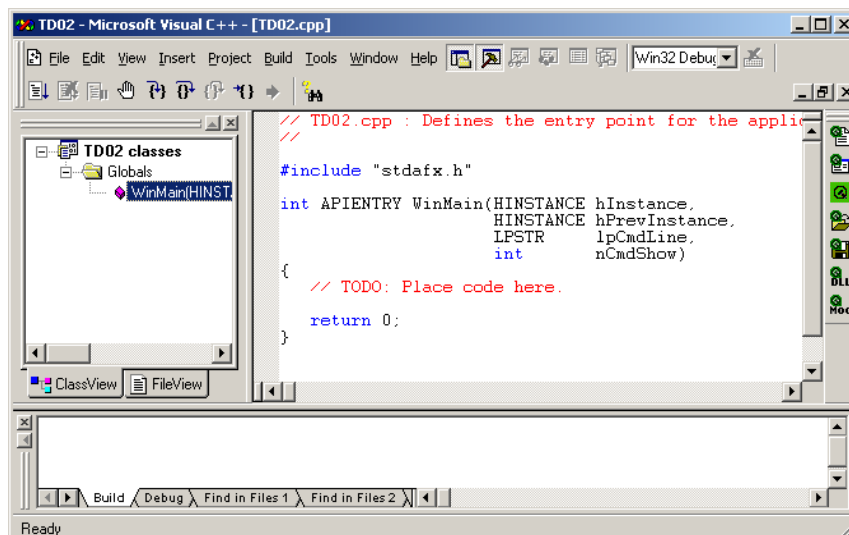
Dans le dialogue suivant, sélectionnez l'option **"A simple Win32 Application"**  et cliquez sur le bouton **"Finish"** .

Une dernière fenêtre vous informe que le projet créé correspond à un programme qui "run and exit immediately" (c'est à dire "s'exécute et se referme immédiatement"), ce qui veut simplement dire qu'il est dépourvu d'interface. Cliquez sur le bouton **"OK"** .

Configuration de l'environnement

Dans l'onglet **"ClassView"** de la fenêtre **"Workspace"**, déployez le dossier **"Globals"** et double-cliquez sur l'unique entrée qu'il contient (**"WinMain(...)"**) .

La scène obtenue devrait alors ressembler à la suivante :



Nota bene :


Utilisez, si nécessaire, les commande **"Output"** et **"Workspace"** du menu **"View"** pour rendre visible les fenêtres qui seraient manquantes.

La taille des différentes fenêtres peut être ajustée en déplaçant leurs frontières à l'aide de la souris.

L'interface utilisateur de Visual C++

Visual C++ a préparé pour vous un fichier nommé TD02.CPP, qui contient un programme composé d'une unique fonction. Cette fonction, nommée WinMain(), sera automatiquement exécutée lorsque le programme sera lancé et c'est donc ici que vous allez créer quelques variables et références.

Toutes les instructions créant des variables ou des références que vous allez écrire au cours de ce TD devront figurer entre la ligne `//TODO : Place code here` et la ligne `return 0;`

Cliquez avec le **bouton droit** sur la ligne `"return 0;"`, et, dans le menu qui surgit alors, choisissez la commande **"Insert/remove breakpoint"** .

Un point rouge doit apparaître dans la marge de gauche, sur la ligne `return 0.`



Une ligne de code dotée d'un point d'arrêt

Ce point rouge signale la présence d'un point d'arrêt, c'est-à-dire d'une ligne sur laquelle l'exécution du programme fera une pause pour nous permettre d'examiner l'état des variables.

L'utilisation de points d'arrêts et l'examen des variables ne sont rendus possibles que par l'exécution du programme sous la supervision d'un programme spécial, appelé un debugger.



2 - Les variables de type standard et leur initialisation

Pour avoir des variables à examiner, il vous faut bien entendu introduire dans le programme des lignes de code ayant pour effet d'en définir...

Une variable non initialisée

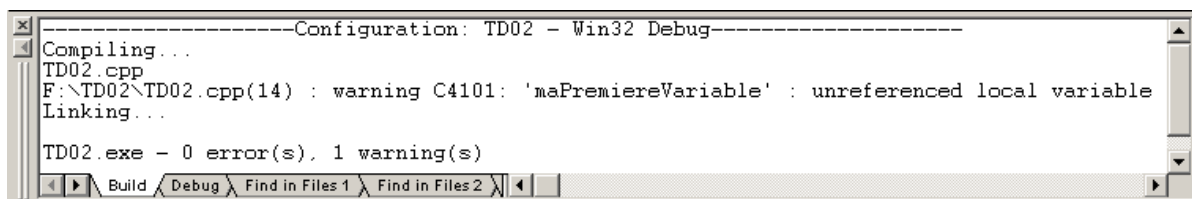
Insérez, dans la fonction WinMain(), la ligne suivante .

```
int maPremiereVariable;
```


Vérifiez que votre ligne de code est parfaitement conforme au modèle (pas d'accent, pas d'espaces intempestifs et un point virgule final) , puis compilez votre programme .

Rappel : la compilation peut être obtenue en pressant la touche **F7**, qui correspond à la commande **"Build"** du menu **"Project"** (compilation et édition des liens).


La compilation de cette ligne de code produit un avertissement :



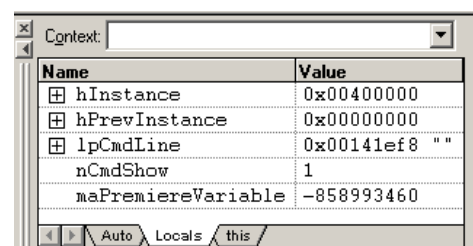
Le compilateur nous signale que notre variable est "unreferenced", ce qui signifie qu'elle est définie, mais que notre programme n'ordonne aucune action la concernant. Un programme qui définit une variable pour n'en rien faire ensuite contient vraisemblablement une erreur, et ce warning est une invitation à la corriger. Avant de procéder à cette correction, nous allons toutefois procéder à un certain nombre d'observations et d'expériences.

Dans le menu **"Build"**, sélectionnez la ligne **"Start debug"** et, dans le sous-menu qui s'ouvre alors, la commande **"Go"** (vous pouvez aussi, plus simplement, presser la touche **F5**) .

Le point rouge symbolisant le point d'arrêt est maintenant surchargé d'une flèche jaune, qui indique que le debugger a interrompu le programme avant l'exécution de l'instruction `return 0.`

Repérez la fenêtre **"Variables"** du debugger .

Si cette fenêtre n'est pas visible, sélectionnez la commande du même nom dans le sous-menu **"Debug windows"** du menu **"View"**.




La fenêtre "Variables" du debugger

Dans son onglet "Locals", cette fenêtre propose un tableau contenant les noms et valeurs courantes de toutes les variables de la fonction où a eu lieu l'interruption du programme. Si les quatre premières de ces variables ne nous concernent pas (ce sont les paramètres de la fonction, objets que nous n'étudierons qu'au cours de la Leçon 5), la dernière ligne indique effectivement la présence de notre variable. Comme vous pouvez le constater,

Une variable non initialisée n'est pas "vide", et ne contient pas forcément la valeur 0.

Les programmeurs débutants ont souvent l'intuition qu'une variable est une sorte de boîte et que, par conséquent, elle est vide tant qu'on n'a rien mis dedans. Si la métaphore de la boîte n'est pas fondamentalement mauvaise, il ne faut pas oublier qu'une variable correspond à une zone mémoire, que cette dernière a forcément un état électrique déterminé, et que tous les états électriques sont interprétables comme des valeurs de la variable. La notion de vide n'a tout simplement aucun sens dans le cas des variables.


Comme la "valeur" d'une variable non initialisée est imprévisible, l'utiliser dans un programme est rarement une bonne idée. Il est donc préférable de suivre le conseil implicitement donné par le compilateur et d'initialiser les variables.

Mettez fin à l'exécution du programme en choisissant, dans le menu "Debug", la commande "Stop debugging" .


Erreurs fréquentes

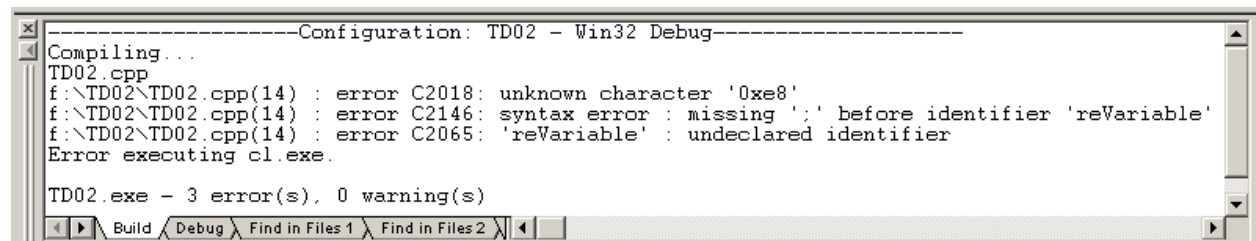
La saisie du texte source donne souvent lieu à des erreurs bénignes, qui empêchent néanmoins la compilation du programme. Il est donc important d'apprendre à reconnaître la source du problème à partir des messages de protestation émis par le compilateur.

Utilisation de caractères interdits

Modifiez ainsi la ligne de code définissant la variable  :

```
int maPremièreVariable;
```

Lors de la compilation (F7) , les messages d'erreur suivants sont émis :



Une unique faute de frappe peut causer plusieurs erreurs de compilation.

Le compilateur ne cherche pas à compter vos erreurs pour vous mettre une note, il cherche à traduire le texte source en une séquence d'instructions exécutables par le processeur. Lors de son analyse du texte, la présence du 'è' fautif le laisse face à trois fragments de texte :

```
1 int maPremi
2 è
3 reVariable;
```

Ainsi, si la présence du caractère inconnu 'è' (en anglais : *unknown character*) est bien la première erreur rencontrée, elle donne naissance à (1) une définition dépourvue de point-virgule (en anglais : *missing ';'*) et à (3) une instruction mentionnant un objet non déclaré (en anglais : *undeclared identifier*). Remarquez que la logique des deux derniers messages d'erreur n'apparaît qu'après que l'on ait compris la cause du premier. Par conséquent,

Lorsque plusieurs erreurs sont signalées, il faut d'abord chercher à comprendre la première.

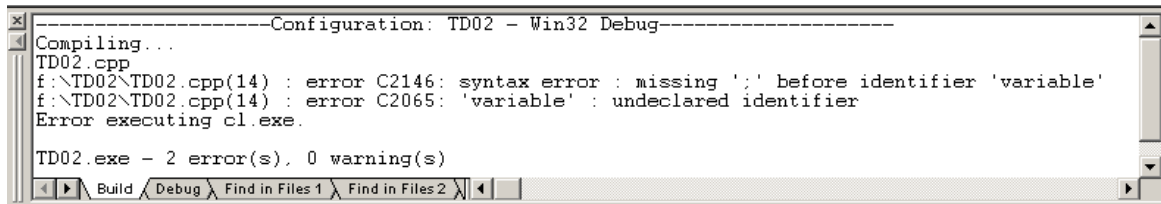
Lorsque les erreurs signalées sont nombreuses, il faut utiliser la barre de défilement vertical pour remonter à l'affichage de la première.

Fragmentation du nom

Modifiez ainsi la ligne de code définissant la variable ☐ :

```
int maPremiere variable;
```

Lors de la compilation (F7) ☐, les messages d'erreur suivants sont émis :



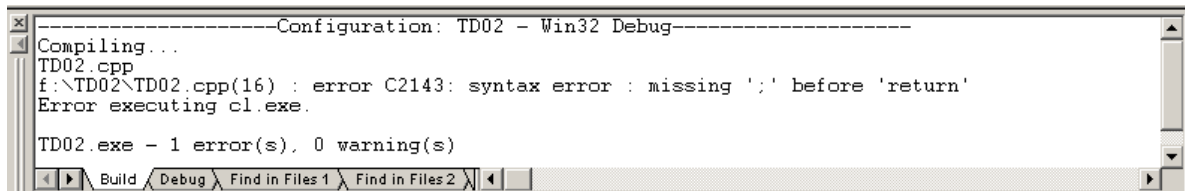
A la lumière de la discussion précédente, ces messages devraient vous sembler limpides.

Oubli du point-virgule final

Modifiez ainsi la ligne de code définissant la variable ☐ :

```
int maPremiereVariable
```

Lors de la compilation (F7) ☐, le message d'erreur suivant est émis :



Il arrive que le compilateur émette un message unique décrivant exactement l'erreur commise.

C'est cependant assez rare (l'oubli d'un seul point-virgule peut, dans certains cas, provoquer l'émission de plus de cent messages d'erreur).

Une variable initialisée

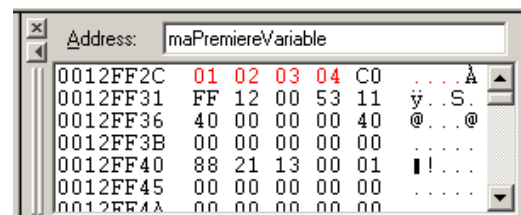
Modifiez le programme de façon à ce que la variable soit initialisée avec la valeur 67305985 ☐.

Compilez (F7) ☐ et lancez l'exécution du programme (F5) ☐. Le programme s'arrête à nouveau sur la ligne return 0, qui comporte toujours un point d'arrêt.

Dans le sous-menu "Debug windows" du menu "View", choisissez la commande "Memory" ☐.

Cette fenêtre (cf. ci-contre) propose une représentation (en hexadécimal) du contenu de la mémoire.

L'annexe 0 propose un rappel sur la notation hexadécimale, mais la maîtrise de cette façon d'écrire les nombres n'est pas indispensable pour l'usage que vous allez faire de la fenêtre "Memory".




La fenêtre "Memory" du debugger

La partie inférieure de la fenêtre "Memory" comporte trois zones :


- la zone centrale contient des nombres hexadécimaux à deux chiffres qui représentent l'état des cases mémoires
- la zone de gauche propose une colonne de nombres à huit chiffres qui sont les adresses correspondant à la première case mémoire dont l'état est représenté sur la ligne concernée.
- la zone de droite donne, pour sa part, une interprétation de l'état de la mémoire supposant que celle-ci contient du texte.

La partie supérieure de la fenêtre "Memory" contient une zone d'édition intitulée "Address:". Effacez le contenu de cette zone et remplacez-le par le nom de votre variable ☐. Lorsque vous pressez la touche "Entrée" ☐, les données affichées dans la partie inférieure de la fenêtre sont ajustées pour que la zone de mémoire représentée débute à l'adresse de la variable.

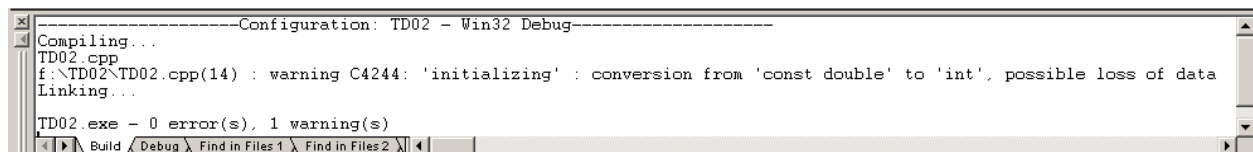
La fenêtre "Memory" représentée ci-dessus indique donc que, au cours de l'exécution du programme pendant laquelle cette photo a été prise, `maPremiereVariable` s'était vu attribuer la zone de mémoire débutant à l'adresse `0x12FF2C` (c'est à dire 1 244 972 en décimal). Il est, bien entendu, possible que la zone de mémoire attribuée à votre propre variable soit différente. Si vous avez bien initialisé la variable avec la valeur suggérée, l'état des quatre premières cases de mémoire affichées devrait en revanche être identique à celui représenté ci-dessus, puisque la séquence 01 02 03 04 correspond au codage de 67305985 dans une variable de type `int`.


Mettez fin à l'exécution du programme en choisissant, dans le menu "Debug", la commande "Stop debugging" .

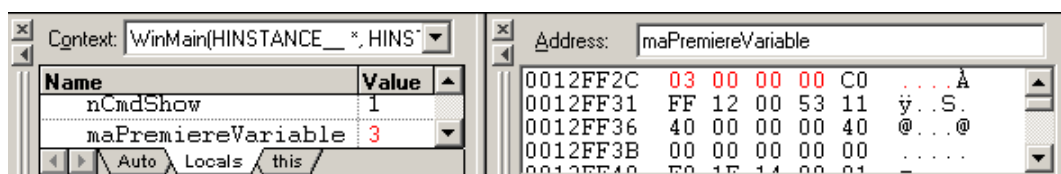
Erreurs d'initialisation

Modifiez le programme de façon à ce que votre variable, bien que de type `int`, soit initialisée avec la valeur 3.14 .

Lors de la compilation (F7) , l'avertissement (*warning*) suivant est émis :



Une compilation qui s'achève sans erreur produit un exécutable qui fonctionne, même si la compilation s'est accompagnée d'avertissements. Lancez l'exécution du programme et observez la valeur de `maPremiereVariable` au moment où le point d'arrêt est atteint .



Etat d'une variable de type `int` initialisée avec la valeur 3.14



L'erreur que nous venons de commettre ne rend pas la compilation impossible, car le compilateur procède à une conversion automatique, qui lui permet d'obtenir la représentation de 3 au format `int` à partir de la représentation de 3.14 au format `double`. Cette conversion s'accompagnant d'une perte de précision, le compilateur émet un avertissement. Remarquez toutefois qu'un programmeur qui initialise une variable avec une valeur comportant une partie décimale n'a sans doute pas en tête la création d'un objet contenant une valeur entière, et risque d'être fort déçu par les résultats obtenus, par la suite, en utilisant cette variable.

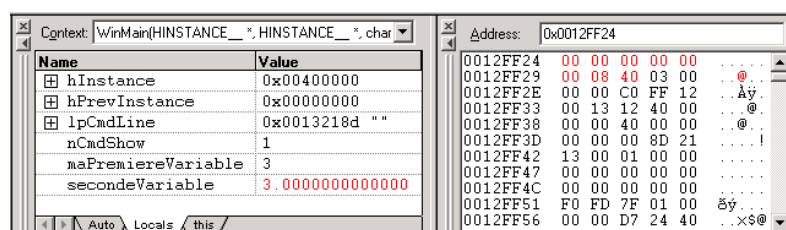
N'ignorez pas les avertissements, ils peuvent signaler une erreur lourde de conséquences.

Conversion automatique

Certaines conversions n'entraînent aucun risque de perte de précision, et sont donc effectuées sans que le moindre avertissement ne les signale. Modifiez la déclaration de `maPremiereVariable` et ajoutez celle d'une seconde variable, de sorte que votre code prenne la forme suivante :

```
1 int maPremiereVariable = 3;
2 double secondeVariable = maPremiereVariable;
```

Après compilation  et exécution , l'état de notre `secondeVariable` apparaît ainsi :



Etat d'une variable de type `double` initialisée avec la valeur 3

La ligne 2 garantit que les deux variables adoptent la même valeur. Il est pourtant clair que leur différence de type conduit ces variables à représenter cette valeur de façon différente.

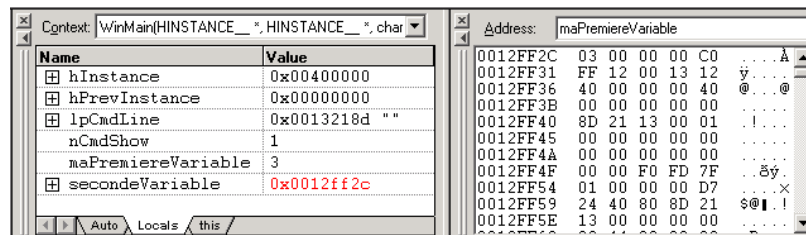
Pour procéder à l'initialisation de la secondeVariable, le compilateur a donc du convertir (silencieusement) la valeur trouvée dans maPremiereVariable (03 00 00 00) pour lui donner la représentation convenant à une variable de type double (00 00 00 00 00 00 08 40).

Une variable de type pointeur

Modifiez la déclaration de secondeVariable, de sorte que votre code prenne la forme suivante :

```
1 int maPremiereVariable = 3;
2 int * secondeVariable = &maPremiereVariable;
```

Après compilation ☐ et exécution ☐, l'état de notre secondeVariable apparaît ainsi :



Si vous demandez à la fenêtre "Memory" d'afficher l'état de la zone mémoire correspondant à maPremiereVariable (en tapant le nom de celle-ci dans la zone "Adress") ☐, vous pouvez constater (dans la colonne de gauche de la fenêtre "Memory"), que cette zone commence effectivement à l'adresse contenue dans secondeVariable (0x12FF2C, sur la copie d'écran proposée ci-dessus).

3 - Création et utilisation d'un type énuméré

La création d'un type énuméré est extrêmement simple, et la seule question qui se pose réellement est la suivante : où allons nous faire figurer sa définition ?

Définition "sur place"

Une première façon de procéder est d'insérer la **définition de notre type** au début du fichier dans lequel il est utilisé. Dans notre cas, cela conduirait à un texte source se présentant ainsi :


```
// TD02.cpp : Defines the entry point for the application.
1 #include "stdafx.h"
2 enum EPersonnage {ROSE, MOUTARDE, LEBLANC, OLIVE, PERVENCHE, VIOLET};
3
4 int APIENTRY WinMain(HINSTANCE hInstance,
5                     HINSTANCE hPrevInstance,
6                     LPSTR lpCmdLine,
7                     int nCmdShow)
8 {
9     // TODO: Place code here.
10    return 0;
11 }
```




Cette méthode a l'avantage d'être simple et directe. Son défaut est que, si des fonctions définies dans un autre fichier entendent, elles aussi, se servir du type EPersonnage, la définition de celui-ci devra également figurer dans cet autre fichier. La répétition de définitions identiques crée un risque : l'accumulation des modifications apportées au texte source peut finir par rendre ces définitions différentes. Cette erreur, difficile à repérer pour un lecteur humain, n'est souvent pas détectable par le compilateur et a toutes les chances de se traduire par un fonctionnement incorrect du programme. Par conséquent,


On préfère habituellement éviter de définir les types "sur place".

Définition dans un fichier .h



L'alternative consiste à créer un fichier ad hoc. La coutume veut que les fichiers contenant des définitions de types portent l'extension ".h".

Dans le menu "File", choisissez la commande "New..." .

Dans l'onglet "Files" du dialogue qui s'ouvre alors, sélectionnez l'option "C/C++ Header File"  et, dans la zone d'édition "FileName:", saisissez le nom `personnages.h` , puis cliquez sur le bouton "OK" .

Vous disposez alors un fichier vierge, dans lequel vous pouvez entrer votre définition .

```
//personnages.h : contient la définition du type EPersonnage, qui sert à
//créer des variables capables de représenter un personnage du jeu de Cluedo
enum EPersonnage {ROSE, MOUTARDE, LEBLANC, OLIVE, PERVENCHE, VIOLET};
```



Enregistrez le fichier (commande "Save" du menu "File")  et refermez-le .

Un tel fichier permet d'éviter d'avoir à répéter la définition du type énuméré : chaque fois que vous aurez besoin d'utiliser celui-ci, il vous suffira d'indiquer dans quel fichier se trouve sa définition. Cette indication est donnée en faisant figurer, en début de fichier, la directive

```
#include "personnages.h"
```

N'insérez jamais vos propres directives `#include` avant le `#include "stdafx.h"` de Visual C++

Utilisation

Une fois le type défini, il s'utilise exactement comme un type standard. Corrigez  le code suivant, de façon à ce que le programme puisse être compilé  (sans erreur ni avertissement).

```
// TD02.cpp : Defines the entry point for the application.
//
1  #include "stdafx.h"
2  #include "personnages.h"
3
4  int APIENTRY WinMain(HINSTANCE hInstance,
5                      HINSTANCE hPrevInstance,
6                      LPSTR     lpCmdLine,
7                      int       nCmdShow)
8  {
9      // TODO: Place code here.
10     EPersonnage moi = MOUTARDE
11     EPersonnage suspectPrincipal(PERVANCHE);
12     EPersonnage autreSuspect = MOI;
13     return 0;
14 }
```

Si le compilateur se plaint que `EPersonnage` est un "undeclared identifier", c'est sans doute que vous avez oublié d'insérer la ligne 2...

C'est parce que les directives `#include` figurent normalement au début des textes source que les fichiers sur lesquels elles portent sont nommés des "headers" (fichiers d'en-tête, en français). Cette dénomination est, à son tour, la cause du choix de l'extension .h

4 - Création et utilisation d'une classe

En l'état actuel de vos connaissances, la création d'une classe n'est pas très différente de la création d'un type énuméré. Dès la Leçon prochaine, toutefois, les classes vont comporter des fonctions membre, et il ne sera plus du tout envisageable de les définir "sur place".

Si les types (classes ou types énumérés) peuvent être redéfinis dans le même programme autant de fois qu'il est nécessaire (lorsqu'ils sont utilisés par des fonctions placées dans des fichiers différents, par exemple), ce n'est pas le cas des fonctions membre (qui ne peuvent être définies qu'une seule fois). Cette contrainte nous force pratiquement à utiliser des fichiers spécifiques pour définir les classes.

Définition

En vous inspirant de la procédure suivie pour le type énuméré, créez un fichier nommé "position.h" ☐.

Placez dans ce fichier le code suivant ☐.

```
1 class CPosition
2 {
3 public:
4     double latitude;      //en degrés
5     double longitude;     //en degrés
6     int altitude;         //en mètres
7 };
```

Enregistrez le fichier ☐ et refermez-le ☐.

Instanciation

Modifiez le code présent dans le fichier TD02.CPP de façon à ce que la fonction WinMain() comporte une variable de type CPosition ☐.

Vous n'êtes pas, pour l'instant, en mesure d'initialiser cette variable.

Vérifiez que votre code peut être compilé sans erreur ni avertissement ☐.

5 - Création de références

L'attribution d'un second nom à une variable suppose, bien entendu l'existence préalable de cette variable. Insérez donc le code suivant dans votre fonction WinMain() ☐.

```
1 int ageDuCapitaine = 15;
2 int &adc = ageDuCapitaine;
```

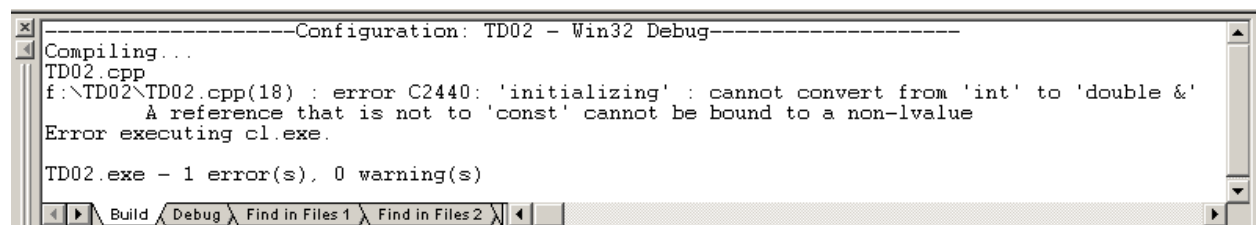
Vérifiez que ce code peut être compilé sans erreur ni avertissement ☐.

Problèmes de conformité des types

Outre l'omission pure et simple de l'initialisation de la référence (qui provoque évidemment une erreur de compilation : *"references must be initialized"*), le seul problème que vous risquez de rencontrer lors de la création d'une référence est une divergence entre le type déclaré pour celle-ci et celui de l'objet qu'elle est censée désigner. Essayez, par exemple,

```
1 int ageDuCapitaine = 15;
2 double age = ageDuCapitaine;
3 double &adc = ageDuCapitaine;
```

La compilation de cette séquence de code ☐ provoque l'émission d'un message d'erreur :



La ligne 2 ne crée pas de problème, car une conversion automatique de la valeur 15 contenue dans ageDuCapitaine produit la valeur 15.0, qui peut être stockée dans la variable age.

Le cas de la ligne 3 est bien différent, car il ne s'agit pas là de ranger une valeur dans une variable, mais d'associer un nouvel identificateur à la variable ageDuCapitaine. Cette variable étant un int, il n'est pas question de lui associer un identificateur qui laisserait croire qu'il s'agit d'un double.

Le message d'erreur émis par le compilateur est accompagné d'une remarque rappelant qu'une référence à un objet non constant ne peut être associée à un objet non modifiable. Cette remarque peut sembler hors sujet, dans la mesure où la ligne 3 ne mentionne aucun

objet non modifiable. Il faut, pour la comprendre, savoir qu'une référence à un objet constant peut être associée à une constante littérale :

```
const double & CONSTANTE = 15; //une façon exotique de créer une constante
```

Si la ligne 3 de l'exemple précédent avait été

```
const double & adc = ageDuCapitaine;
```

le compilateur aurait tout simplement associé l'identificateur adc à une constante double de valeur 15.0 (valeur obtenue par conversion automatique de celle contenue dans ageDuCapitaine). L'erreur rencontrée sur la ligne 3 peut donc avoir deux origines : soit le programmeur souhaitait créer un surnom pour ageDuCapitaine (et il a écrit double au lieu d'int), soit le programmeur souhaitait créer une constante de type double initialisée avec la valeur courante de la variable ageDuCapitaine (et il a oublié le mot const). C'est à cette seconde hypothèse que fait allusion la remarque émise par Visual C++.

6 - Qu'avons nous appris ?

- 1 - Un programme peut être dépourvu de toute interface utilisateur.
- 2 - Les variables non initialisées sont dans un état imprévisible.
- 3 - Le nombre d'erreurs signalées par le compilateur n'est pas le nombre d'erreurs contenues dans le code source.
- 4 - Il faut toujours commencer par corriger la première erreur signalée.
- 5 - Error(s) et warning(s) méritent le même traitement : il faut éliminer leur cause.
- 6 - On définit un type (classe ou type énuméré) dans un fichier .h
- 7 - Pour pouvoir utiliser un type défini dans un autre fichier, il faut "#include" le fichier en question.